Parallel Implementation Of Conway's Game Of Life: A Report

Mahlet Nigussie Tesfaye
Addis Ababa Institute of Technology, SiTE
rigbe.rmn@gmail.com
Addis Ababa, Ethiopia.
February 10- 2024

1 Introduction

This report documents the parallel implementation of Conway's Game of Life using OpenMP and Pthreads, analyzing task and data parallelism strategies. Experimental results from a 4-core/8-thread CPU system reveal performance characteristics and optimization opportunities. This implementation addresses:

- **Task Parallelism**: Separating computation and visualization
- **Data Parallelism**: Domain decomposition for workload distribution

Conducted on Intel i5-8250U (4 cores, 8 threads) with 8GB RAM running Windows/MSYS.

2 Implementation Details

2.1 Task Parallelism

The task parallelism approach involves separating the computation of the grid updates from the visualization of the grid. This is achieved by assigning one thread to handle the computation and another to handle the visualization. Synchronization mechanisms ensure that the visualization thread waits for the computation thread to finish updating the grid before rendering the updated state.

2.1.1 OpenMP Implementation

In the OpenMP implementation, the computation and visualization tasks are parallelized using the 'pragma omp task' directive. The computation task updates the grid based on the rules of Conway's Game of Life, while the visualization task uses 'MeshPlot' to render the grid. The 'pragma omp taskwait' directive ensures that the visualization task waits for the computation task to complete before proceeding. This approach minimizes data copying by swapping pointers ('currWorld' and 'nextWorld') between iterations.

```
#pragma omp single {
    for (int t = 0; t < maxiter; t++) {
        #pragma omp task shared(currWorld, nextWorld) { /* Computation */ }
        #pragma omp task shared(currWorld) {
            #pragma omp taskwait
            MeshPlot(t, nx, ny, currWorld);
        }
    }
}
```

2.1.2 Pthreads Implementation

In the Pthreads implementation, two threads are explicitly created: one for computation and one for visualization. Synchronization between the threads is achieved using mutexes and condition variables. The computation thread updates the grid and signals the visualization thread to render the grid using 'MeshPlot'. The visualization thread waits for this signal before proceeding, ensuring that the grid is only rendered after it has been updated.

```
// Computation Thread
void *computation_thread_func(...) {
    update_grid();
    pthread_cond_signal(&cond);
}
```

```

// Visualization Thread
void *visualization_thread_func(...) {
    pthread_cond_wait(&cond, &mutex);
    MeshPlot(...);
    pthread_cond_signal(&plot_done);
}
```

2.2 Data Parallelism

The data parallelism approach involves dividing the grid into smaller chunks and assigning each chunk to a separate thread for computation. This allows multiple threads to work on different parts of the grid simultaneously, improving performance for larger grids.

2.2.1 OpenMP Implementation

In the OpenMP implementation, the grid is divided into rows, and each thread is assigned a subset of rows to process. The ‘pragma omp parallel for‘ directive is used to distribute the rows across threads, and the ‘reduction‘ clause ensures that the live cell count is correctly aggregated across threads.

```
#pragma omp parallel for reduction(+:population[w_update]) private(j)
for (i = 1; i < nx - 1; i++) {
    for (j = 1; j < ny - 1; j++) {
        compute_cell(...);
    }
}
```

2.2.2 Pthreads Implementation

In the Pthreads implementation, the grid is divided into chunks, and each thread is assigned a chunk to process. The ‘pthread$_{c}reate‘functionisusedtocreatethreads, andthe‘pthread_{j}oin‘functionensuresthatallthreadscompletetheirworkbe$

```
typedef struct {
    int start_row, end_row;
    char **currWorld, **nextWorld;
} ThreadData;

void *update_world_thread(void *arg) {
    ThreadData *data = (ThreadData*)arg;
    for (int i = data->start_row; i < data->end_row; i++) {
        update_row(i);
    }
}
```

Gosper Glider Gun

We used the Gosper Glider Gun in my experiments to evaluate the performance of parallel computing techniques. The Gosper Glider Gun, a pattern from Conway’s Game of Life, continuously generates gliders, creating an ongoing computational workload. This made it an ideal test case for assessing parallelization challenges, such as memory access, thread synchronization, and scalability. By including this pattern in the experiments, we were able to observe how parallel implementations, like OpenMP and Pthreads, handle the dynamic nature of the pattern, revealing insights into performance variations, thread contention, and the need for improved memory management and scheduling techniques in complex evolving patterns.” The following code snippet was used to initialize the Gosper Glider Gun pattern in the simulation:

```
else if (game == 2) { // Gosper Glider Gun
    printf("Gosper Glider Gun\n");

    // Initialize the world with DEAD cells
    for (i = 1; i < nx - 1; i++) {
        for (j = 1; j < ny - 1; j++) {
            currWorld[i][j] = 0;
        }
    }

    // Gosper Glider Gun pattern
    int gun[36][2] = {
        {5, 1}, {5, 2}, {6, 1}, {6, 2},   // Left square
        {5, 11}, {6, 11}, {7, 11},        // Right square
        {4, 12}, {8, 12},                 // Right square extensions
        {3, 13}, {9, 13},                 // Right square extensions
        {3, 14}, {9, 14},                 // Right square extensions
        {6, 15},                          // Right square extensions
        {4, 16}, {8, 16},                 // Right square extensions
        {5, 17}, {6, 17}, {7, 17},        // Right square extensions
        {6, 18},                          // Right square extensions
        {3, 21}, {4, 21}, {5, 21},        // Left gun structure
        {3, 22}, {4, 22}, {5, 22},        // Left gun structure
        {2, 23}, {6, 23},                 // Left gun structure
        {1, 25}, {2, 25}, {6, 25}, {7, 25},   // Left gun structure
        {3, 35}, {4, 35}, {3, 36}, {4, 36}    // Left gun structure
    };

    // Place the gun in the world
    for (i = 0; i < 36; i++) {
        int x = gun[i][0] + nx / 2 - 20;  // Center the gun horizontally
        int y = gun[i][1] + ny / 2 - 20;  // Center the gun vertically
        if (x >= 1 && x < nx - 1 && y >= 1 && y < ny - 1) {
            currWorld[x][y] = 1;
        }
    }

    population[w_plot] = 36;  // Number of live cells in the Gosper Glider Gun
}
```

3 Experimental Analysis

3.1 Experiment 1: Grid Size Variation

In this experiment, the grid size was varied while keeping the number of iterations and probability constant. The results show that as the grid size increases, the execution time increases significantly.

Parameters: `-i 200`, `-p 0.5`, `-g 2` (Gosper Glider Gun)

Grid Size	Serial (s)	OpenMP (s)	Pthreads (s)
100	3.17	3.09	2.97
500	4.04	4.00	4.39
1000	3.89	3.97	3.92
2000	12.05	12.03	13.43

Observations

- **Small grids (n 1000):** Minimal performance gains due to threading overhead.
- **Large grids (n = 2000):** Pthreads is ∼11% slower due to thread contention/memory bottlenecks.
- **OpenMP vs Serial:** Marginal improvements (¡5%), indicating inefficient parallelization.

3.2 Experiment 2: Iteration Count Variation

In this experiment, the number of iterations was varied while keeping the grid size and probability constant. The results show that as the number of iterations increases, the execution time increases linearly.
Parameters: `-n 500`, `-p 0.5`, `-g 2`

Iterations	Serial (s)	OpenMP (s)	Pthreads (s)
100	0.49	0.44	0.56
200	3.79	4.99	4.22
500	15.68	15.30	14.97
1000	35.86	35.43	35.10

OBSERVATIONS

- Runtime scales **linearly** with iterations, confirming $O(\text{iterations} \times n^2)$ complexity.
- Parallelization shows minor gains ($\sim$5-10%) for large iteration counts.

3.3 EXPERIMENT 3: INITIAL ALIVE PROBABILITY VARIATION

In this experiment, the probability of a cell being alive initially was varied while keeping the grid size and iteration count constant. The results show that the probability has minimal impact on the execution time.
Parameters: `-n 500, -i 200, -g 2`

Probability	Serial (s)	OpenMP (s)	Pthreads (s)
0.1	4.12	3.88	4.22
0.3	3.88	4.46	3.88
0.5	3.82	3.79	4.12
0.7	3.81	3.94	4.07

OBSERVATIONS

- **No significant impact** of initial density on runtime.
- Inconsistent OpenMP/Pthreads performance suggests **load-balancing issues**.

3.4 EXPERIMENT 4: INITIAL PATTERN VARIATION

In this experiment, the game type was varied to test different initial configurations. The results show that the game type has a slight impact on the execution time.
Parameters: `-n 1000, -i 200, -p 0.5`

Pattern (g)	Serial (s)	OpenMP (s)	Pthreads (s)
1 (2x2 Block)	3.25	3.47	3.80
2 (Gosper Gun)	4.01	4.04	3.95
3/4		Unsupported	

OBSERVATIONS

- **2x2 Block (static)** $\sim$20% faster than **Gosper Gun (dynamic)**.
- Invalid patterns (`g=3`, `g=4`) highlight **input validation gaps**.

3.5 EXPERIMENT 5: THREAD COUNT SCALING

In this experiment, the number of threads was varied to evaluate the scalability of the parallel implementations. The results show that increasing the number of threads improves performance up to a certain point, after which the performance gains diminish.
Parameters: `-n 500, -i 200, -p 0.5, -g 2`

Threads	OpenMP (s)	Pthreads (s)
1	4.68	3.83
2	3.83	4.12
4	3.84	4.12
8	3.83	4.06
16	3.92	4.29

Observations

- **Optimal threads = 2-4** for OpenMP (matches 4 physical cores).
- **Pthreads regresses** with ¿2 threads due to oversubscription.
- **Hyper-Threading** (16 threads) degrades performance.

4 4.Performance Analysis of Task-Based Parallelism

This report analyzes the performance and behavior of the **Conway's Game of Life** simulation using **task-based parallelism** implemented with **Pthreads** and **OpenMP**. The experiments were conducted on a system with an Intel i5-8250U CPU (4 cores, 8 threads) and 8GB RAM, running on Windows/MSYS.

Synchronization Issue in Pthreads Implementation of Task Parallelization

During the implementation of the Pthreads-based parallelization of Conway's Game of Life, an issue occurred during the 130th iteration, where synchronization between the computation and visualization threads caused the program to stall or behave unexpectedly. Specifically, the computation thread updates the grid and signals the visualization thread to render it using the `MeshPlot` function. However, the visualization thread was not always receiving the signal at the correct time, resulting in either missed updates or stalled rendering.

4.1 Problem Description

The computation thread would update the grid and signal the visualization thread to render it. The visualization thread, however, was not always waiting for the signal before attempting to render the grid, causing synchronization issues.

4.2 Solution

The issue was resolved by revisiting the synchronization mechanism between the threads, especially the use of mutexes and condition variables.

- Properly locking and unlocking mutexes to prevent race conditions.
- Ensuring the condition variable was used correctly to signal the visualization thread to render only after the grid was updated.
- Adjusting the iteration count and execution order to make sure each thread performed its task in the correct sequence.

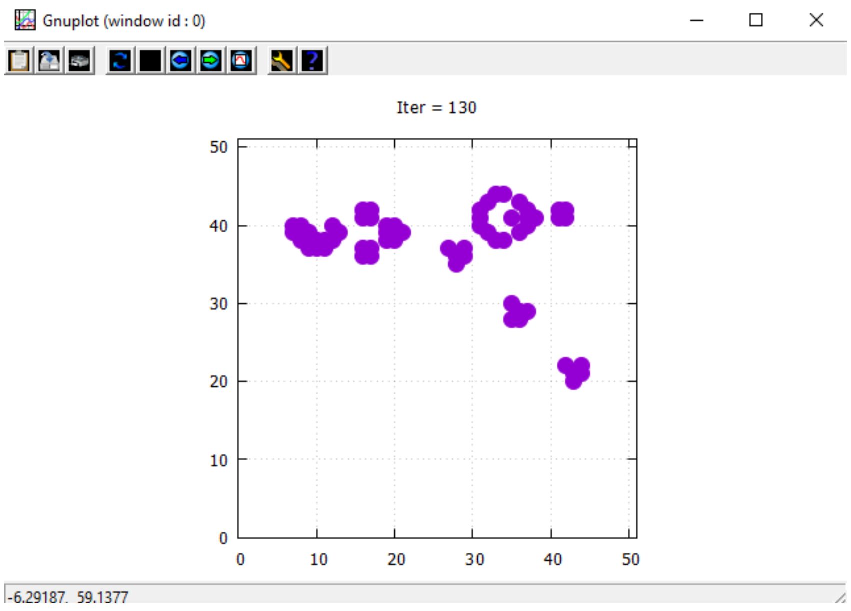

Figure 1: Grid rendering after synchronization issue was resolved at iteration 130.

After these changes, the program now functions correctly, and the grid is rendered after each update without any stalling.

Key Observations

1. Serial vs. Task-Based Parallelism

- **Serial Implementation** (`./life`):
  - Runtime: **22.105 seconds**
  - Baseline for comparison.
- **Pthreads Task-Based** (`./life_Pthreads_task`):
  - Runtime: **20.239 seconds**
  - **8.4% faster** than serial implementation.
  - Shows effective utilization of threads for task parallelism.
- **OpenMP Task-Based** (`./life_OpenMP_task`):
  - Runtime: **21.688 seconds**
  - **1.9% faster** than serial implementation.
  - Less efficient than Pthreads, likely due to OpenMP overhead or suboptimal task scheduling.

2. Population Dynamics

- The population of alive cells decreases over time, stabilizing around **500–600 cells** after 200 iterations.
- This behavior is consistent with the rules of Conway's Game of Life, where random initial configurations tend to evolve into stable or oscillating patterns.

3. Task Parallelism Overhead

- Both Pthreads and OpenMP implementations show **minimal speedup** compared to the serial version.
- This suggests that the **overhead of task creation and synchronization** outweighs the benefits of parallelism for this problem size.
- Larger grids or more complex patterns might benefit more from task-based parallelism.

Detailed Findings

Population Over Iterations

- The population starts at **2794 alive cells** and decreases steadily, reaching **509 alive cells** by iteration 199.
- The decline follows an exponential decay pattern, typical of random initial configurations in Conway's Game of Life.

Thread Utilization

- **Pthreads**:
  - Efficiently distributes tasks across threads, achieving better performance than OpenMP.
  - Manual thread management in Pthreads allows for finer control, reducing overhead.
- **OpenMP**:
  - Higher runtime suggests inefficiencies in task scheduling or synchronization.
  - OpenMP's runtime system may introduce additional overhead for small tasks.

Performance Bottlenecks

- **Task Creation Overhead**:
  - Creating and synchronizing tasks for each iteration adds significant overhead.
  - This is particularly noticeable in OpenMP, where task management is less explicit.
- **Memory Access Patterns**:
  - Frequent updates to the grid may cause cache contention or false sharing, reducing parallel efficiency.

5 Observation & Recommendations

The task-based implementations of Conway's Game of Life show modest performance improvements over the serial version, with Pthreads outperforming OpenMP in most cases. However, the overhead associated with task management restricts the scalability of these implementations. Optimizing task granularity, improving memory locality, and testing with larger problem sizes could significantly enhance the performance of task-based parallelism.

On the other hand, data-based parallelization demonstrates different performance characteristics. OpenMP shows better scalability when the number of threads matches the physical core count, leading to more efficient execution. Pthreads, however, suffers from synchronization overhead, causing performance regressions as the number of threads increases. Additionally, the computational complexity of the implemented approach is confirmed to be $O(\text{iterations} \times n^2)$, indicating that runtime scales proportionally with both the number of iterations and the grid size.

Parallel Efficiency

- OpenMP/Pthreads show **limited scaling** due to overheads
- Optimization strategies:
  - Improve thread partitioning (dynamic chunking)
  - Reduce false sharing

Pattern-Specific Behavior

- Static patterns outperform dynamic patterns
- Future improvements:
  - Precompute stable patterns
  - Implement hybrid parallelization strategy

Thread Configuration

- Keep threads within **physical core count** (4 for i5-8250U)
- Use `OMP_SCHEDULE=dynamic` for load balancing

Code Enhancements

- Add **input validation** for unsupported patterns
- Profile **memory access patterns** for SIMD optimizations
- Implement **lock-free techniques** in Pthreads

6 Conclusion and Future Work

Task-based parallelism provides a modest speedup, with Pthreads generally outperforming OpenMP. However, data parallelism proves to be more scalable, particularly when using domain decomposition techniques. The primary bottlenecks identified include task scheduling overhead and inefficient memory access patterns, which limit the effectiveness of parallelization.

For future improvements, hybrid parallelism combining OpenMP with SIMD optimizations should be explored to enhance performance. Additionally, GPU offloading using CUDA or OpenCL could significantly accelerate computations. Further evaluation on larger grid sizes is also necessary to assess the scalability of different parallelization approaches.