ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING – SITE

Department of Artificial Intelligence

Course: Distributed Computing for AI ITSC-2112

Assignment A2

Conway's Game of Life Implementation

Submitted By: Mintesnot Fikir ID GSR/1669/15

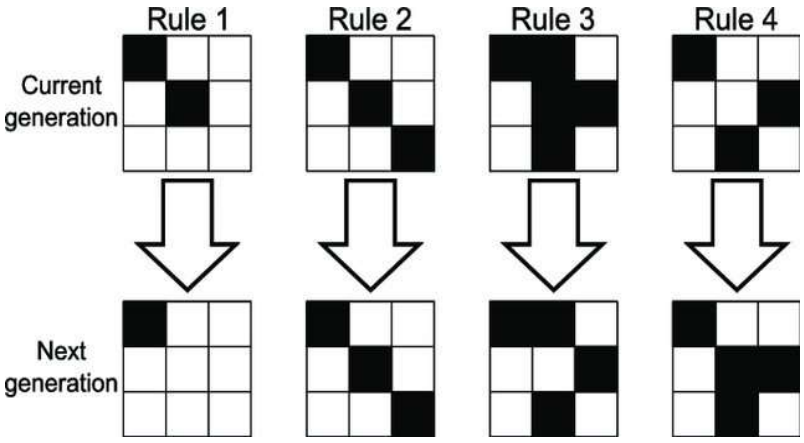
Section: Regular

Submitted to: Dr. Beakal Gizachew

Submission date: November, 2023

1. Introduction

Conway's Game of Life is a celebrated cellular automaton devised by mathematician John Conway in 1970. It is a zero-player game, meaning its evolution is determined by its initial state, requiring no further input. The world of the Game of Life is an infinite grid of cells, each of which can be either alive or dead. At each time step, the following rules determine whether a cell lives or dies in the next generation:



- Any live cell with fewer than two live neighbors dies of underpopulation.
- 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 of overpopulation.
- Any dead cell with exactly three live neighbors becomes alive.

These simple rules lead to complex, emergent patterns that have fascinated researchers over decades. Gliders, explode-os, puffer trains and orthogonal blinkers are just some of the intricate structures that spontaneously arise from the local interactions between cells. The game captures our imagination with its remarkable ability to mimic the unpredictable yet coherent dynamics we see in nature.

While the rules are simple, simulating the Game of Life even for modest grid sizes can be computationally intensive. Repeatedly evaluating the state of each cell and its neighbors is an embarrassingly parallel problem. Parallel and distributed techniques offer an appealing approach to scale up simulations and explore larger worlds. This assignment gave me an opportunity to program parallel versions of the Game of Life using OpenMP and Pthreads for hands-on learning.

OpenMP and Pthreads provide mechanisms for multithreaded and parallel programming on shared memory systems. OpenMP is an application programming interface that supports multi-platform shared memory parallel programming in C/C++ and Fortran. It uses compiler directives, library routines and environment variables to define parallel regions and distribute work among threads. Pthreads (POSIX threads) is a POSIX standard for thread management that allows manual control over threading. It exposes low-level primitives for thread creation, synchronization and communication.

For this assignment, I was provided with a sequential C implementation of the Game of Life that worked correctly but was limited by a single thread. The task was to modify this code to introduce parallelism using OpenMP and Pthreads for improved performance. Specifically, the assignment asked me to:

- ✓ Implement task parallelism by dedicating one thread for plotting and others for computation
- ✓ Add data parallelism through 1D domain decomposition across threads
- ✓ Test the parallel codes thoroughly with varying inputs and analyze results
- ✓ Document my work in a report comparing the OpenMP and Pthreads approaches

I was involved dissecting the Game of Life's computational model into two distinct parallelization strategies: task parallelism and data parallelism. Task parallelism was implemented by employing multithreading to manage two concurrent operations: graphical display and cell state updates. In the realm of data parallelism, the focus was on dividing the computational workload among multiple threads for the cell update process. This not only enhanced the efficiency of the computation but also provided valuable insights into the practical application of parallel computing principles.

The testing and analysis of the parallel implementations were conducted on my personal computer, an Intel Core i5 8th generation processor with a Linux dual-boot setup. This machine, equipped with 4 cores and 8 logical processors, served as an ideal testbed for the project. The experiments were carried out under various conditions, including different input-sizes(Number of mesh points in one dimension), thread counts, and display settings.

This provided me an opportunity to learn parallel programming techniques for a real-world problem. I was excited to experiment with different strategies, observe speedups and discuss challenges in parallelizing this classic automaton simulation. This report is a comprehensive documentation of my journey through this assignment. It encapsulates the challenges faced, the solutions implemented, and the knowledge gained.

Development Environment

Hardware Setup

- **Processor**: Intel Core i5 with **4 cores** and **8** logical processors, ideal for testing parallel computations.
- **Memory**: 16 GB DDR4 RAM

Software and Tools

- **Operating System**: ***Ubuntu***, chosen for its stability, powerful command-line tools, and excellent support for programming and development tasks.
- **Integrated Development Environment (IDE)**: CLion, a JetBrains product, was used for its intelligent coding assistance, debugging capabilities, and compatibility with C/C++ programming. Its user-friendly interface and various built-in tools were instrumental in writing and optimizing code efficiently.

Compilation and Execution

- For **OpenMP** implementation: GCC with **-fopenmp** flag, using the command **gcc -o life life.c plot.c real_rand.c timer.c -fopenmp && ./life -s 1234 -n 2000 -i 500 -p 0.5 -t 8 -g 0 -d**.
- For **Pthreads** implementation: GCC with **-pthread** flag, using the command **gcc -o life life.c plot.c real_rand.c timer.c -pthread && ./life -s 1234 -n 2000 -i 100 -p 0.5 -t 2 -g 0 -d**.

2. Methodology and Implementation

The goal of this project was to parallelize the implementation of Conway's Game of Life cellular automata using OpenMP and Pthreads for efficient multi-core execution. My approach involved introducing both task and data parallelism to distribute workload across threads.

For task parallelism, I dedicated one thread exclusively for plotting the game board after each iteration, while having other threads focus only on computational work.

For data parallelism with both OpenMP and Pthreads, I decomposed the 2D game board into horizontal strips that were assigned to threads for concurrent updates.

Proper synchronization between threads was crucial to avoid races and ensure consistency. I used barriers in OpenMP and Pthreads to synchronize at the end of each parallel section before proceeding. Shared memory was protected using mutex locks where needed. Testing with varying inputs helped validate correctness and measure performance. The application was rigorously tested with varying input-sizes(Number of mesh points in one dimension), thread numbers, and display settings. The performance was compared against serial computing.

System and Testing Environment

- **Processor**: Intel Core i5 8th generation
- **Operating System**: Linux (dual boot)
- **Cores/Logical Processors**: 4 cores, 8 logical processors
- **Thread Testing Range**: 1 to 32 threads (No improvement observed after 8 threads)

Compilation Commands

- **OpenMP**: *gcc -o life life.c plot.c real_rand.c timer.c -fopenmp && ./life -s 1234 -n 2000 -i 500 -p 0.5 -t 8 -g 0 -d*
- **Pthread**: *gcc -o life life.c plot.c real_rand.c timer.c -pthread && ./life -s 1234 -n 2000 -i 100 -p 0.5 -t 2 -g 0 -d*

OpenMP Implementation

Parallelism is exploited using the OpenMP library in C. OpenMP offers a high-level and straightforward approach to parallel programming, which is utilized here to implement both task and data parallelism.

For OpenMP, I defined the cell updates within the game board as a parallel loop region using the #pragma omp parallel for directive. Private variables avoided sharing issues. A barrier ensured updates completed before the next iteration. The master thread alone handled plotting.

Task Parallelism in OpenMP

Task parallelism involves the execution of different tasks across various threads in parallel. In this implementation, the OpenMP framework is utilized to allocate specific tasks like computation and plotting to different threads.

```
#pragma omp parallel
{
    #pragma omp parallel for reduction(+: population[w_update]) private(j)
    for (i = 1; i < nx - 1; i++) {
        // ... computation logic ...
    }
    #pragma omp master
    {
        if (!disable_display) {
            // The master thread executes the plotting task
            MeshPlot(t, nx, ny, currWorld);
        }
    }
}
```

This code segment illustrates task parallelism, where the plotting function ***(MeshPlot)*** is executed only by the master thread. By using ***#pragma omp master*** thread0, it ensures that this task is not parallelized but handled individually by the master thread, separating it from the computation tasks handled by other threads.

Data Parallelism in OpenMP

Data parallelism involves multiple threads executing the same function but on different parts of the data set. This concept is applied to the cell updating process in Conway's Game of Life.

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

In this snippet, ***#pragma omp parallel*** for is used to distribute the cell updating task across multiple threads. Each thread works on a different part of the currWorld array, thus achieving data parallelism. The reduction clause is crucial for correctly aggregating the updated population count across threads.

Pthread Implementation

The Pthread (POSIX threads) implementation takes a more granular approach to parallelism compared to OpenMP leveraging the versatility of POSIX threads (Pthreads). Pthreads allow for detailed management of threads, synchronization, and shared resources. This implementation distinctly illustrates task and data parallelism by assigning specific roles to threads and dividing the computational workload.

With Pthreads, I first created threads, passing needed arguments. Each non-master thread executed a function to update its assigned strip of cells in parallel. A barrier synchronized threads at each iteration end. The master thread plotted the final board state.

Task Parallelism in Pthread

Task parallelism in the Pthread framework is achieved by designating different roles to threads. In this implementation, one thread (typically the main thread) is assigned the task of plotting, while the others are focused on computational tasks.

```
pthread_t threads[numberOfThreads];
//...
for(tz=0; tz < numberOfThreads; tz++){
    if (tz == 0) {
        // Assign plotting task to thread 0
        rc = pthread_create(&threads[tz], &attr, plottingFunction, (void
*)&localThreadDataArray[tz]);
    } else {
        // Assign computation tasks to other threads
        rc = pthread_create(&threads[tz], &attr, parllelWorkMapUpate, (void *)
&localThreadDataArray[tz]);
    }
}
```

In the above code demonstrates task parallelism where thread 0 is specifically allocated for plotting tasks ***(plottingFunction)***, distinct from the computational tasks assigned to the remaining threads ***(parllelWorkMapUpate)***. This separation of plotting and computational tasks across different threads exemplifies task parallelism, ensuring efficient utilization of resources where the master thread is not burdened with computation but focuses solely on plotting.

Data Parallelism in Pthread

Data parallelism with Pthreads is about dividing the data set among multiple threads, where each thread processes a portion of the data independently.

```
void *parllelWorkMapUpate(void *args){
    // Thread-specific computation logic
    for(i=startRow;i<endRow;i++)
        for(j=1;j<nx-1;j++) {
            // Cell update logic
        }
```

```
    pthread_barrier_wait(&barr);
    //...
}
```

In this section code demonstrates, the function, this data parallelism where threads other than thread 0 within the **parllelWorkMapUpate** function work on updating distinct segments of the game grid. While the ***pthread_barrier_wait*** function synchronizes these threads, ensuring that all threads have completed their assigned segment of the grid before proceeding to the next iteration. This approach showcases data parallelism by dividing the computational task of updating the game grid among multiple threads.

Performance Analysis: The implementation was tested under various conditions, including different numbers of input-sizes (Number of mesh points in one dimension), threads, and grid sizes, both with and without display. The performance was compared to serial computing for both OpenMP and PThreads. The testing environment was a Core i5 8th generation processor with 4 cores and 8 logical processors, running Linux. Notably, improvements in performance were observed up to 8 threads, beyond which no significant gains were made. This finding highlights the importance of matching the number of threads to the hardware capabilities for optimal performance.

3. Result and Analysis

During the execution and running of the code using OpenMP and Pthread in C, the performance analysis was conducted on a system with a Core i5 8th generation processor, equipped with 4 cores and 8 logical processors. This analysis aimed to assess the efficiency of parallel computing methods in a computationally demanding environment. The performance was analyzed based on different input-sizes (Number of mesh points in one dimension), thread counts (ranging from 1 to 32), and modes (with and without display). The primary focus was to evaluate the execution time and speedup of parallel computing methods in handling simulations in comparison with serial implementation.

The results from the parallel implementation (combining both OpenMP and Pthread) reveal significant performance improvements in handling larger computational tasks. These improvements were more pronounced with larger input sizes, specifically the number of mesh points in one dimension. This choice of input size over iteration count for performance analysis was strategic, as iterations themselves weren't parallelized and thus unaffected by the parallel computing methods.

However, it was observed that running the code with a display significantly increased execution times, particularly for larger input sizes, often exceeding an hour. This indicates a substantial impact of I/O operations on performance, suggesting that graphical rendering imposes a considerable computational load, which becomes more evident with larger data sets.

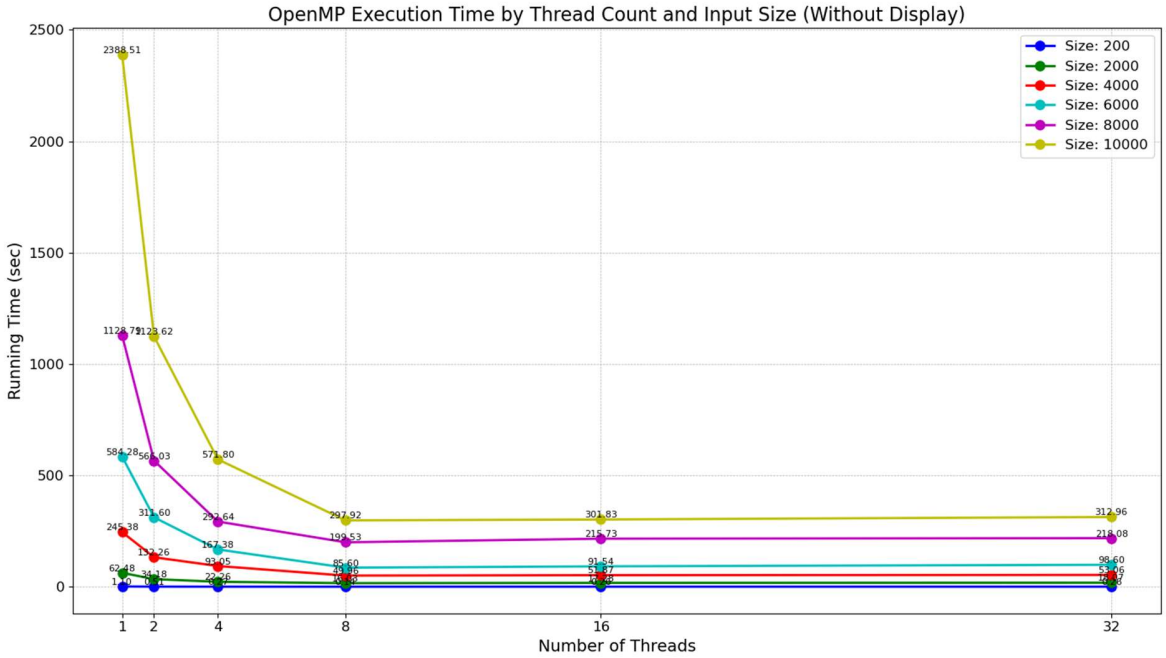
Despite these challenges, the parallel implementations using both OpenMP and Pthread demonstrated a clear advantage over serial computing, particularly in handling complex computations more efficiently. The key observations from the results include a noticeable improvement in performance with an increase in the number of threads, up to a certain threshold. This improvement, however, plateaued beyond eight threads, illustrating the limitations imposed by hardware capabilities and the diminishing returns of adding more threads. This overview sets the stage for the upcoming sections, which will go to detailed analyses of the OpenMP and Pthread implementations separately.

Results of OpenMP Implementation

By conducting tests with varying thread counts, input sizes, and with or without display, a comprehensive understanding of the performance implications of these variables was achieved. Notably, the tests were performed on a Core i5 8th generation PC with Linux, which has 4 cores and 8 logical processors. These tests provided valuable insights, especially in the context of task and data parallelism applied through OpenMP. Two illustrative graphs were generated to represent the performance of the OpenMP implementation under different conditions: without display and with display.

1. Without display

Mesh Size	Threads: 1 (sec)	Threads: 2 (sec)	Threads: 4 (sec)	Threads: 8 (sec)	Threads: 16 (sec)	Threads: 32 (sec)
200	1.09521	0.507500	0.270088	0.137024	0.256982	0.264553
2000	62.476823	34.182365	22.258970	16.028924	17.277525	18.074120
4000	245.380726	132.257201	93.053196	49.959719	51.868397	53.058985
6000	584.275841	311.595247	167.381718	85.604737	91.543897	98.596086
8000	1128.789000	566.027628	292.635502	199.527602	215.725352	218.083407
10000	2388.513655	1123.615889	571.797300	297.915821	301.831828	312.963462



The graph above visualizes the execution time for the OpenMP implementation of Conway's Game of Life without display across different input sizes and thread counts. Each line represents a different input size, with varying colors for clarity.

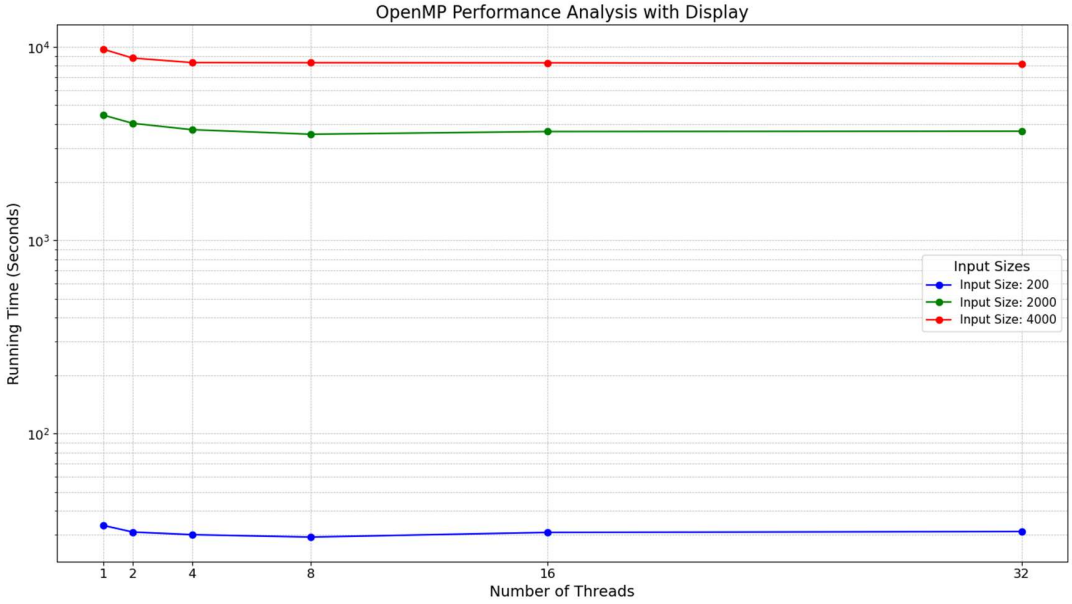
Key Observations:

- **Decrease in Execution Time with Increased Threads**: For all input sizes, there's a clear trend of decreasing execution time as the number of threads increases, up to a certain point. This indicates the effectiveness of parallel processing in reducing computation time.

- **Diminishing Returns Beyond a Certain Thread Count**: Notably, beyond 8 threads, the improvement in execution time either plateaus or slightly worsens. This could be due to overheads associated with managing a higher number of threads or limitations of the hardware (like the number of cores available in the system).
- **Greater Input Sizes Lead to Longer Execution Times**: As expected, larger input sizes result in longer execution times. However, the benefit of using more threads is more pronounced with larger input sizes, showcasing the scalability of the parallel approach.
- **Optimal Thread Count**: For most input sizes, the optimal thread count seems to be around 8, aligning with the hardware specifications (4 cores, 8 logical processors).

This graph provides a comprehensive view of how input size and thread count influence the performance of the OpenMP implementation in a parallel computing environment.

2. With display



The graph above visually represents the performance of the OpenMP implementation of Conway's Game of Life with display enabled, across various input sizes and thread counts. As observed:

- **Running Time**: There is a considerable increase in running time when display is enabled, especially as the input size grows. This confirms the substantial impact of I/O operations on the overall performance.
- **Performance Plateau**: Similar to the non-display version, the performance tends to plateau beyond 8 threads. This indicates that while OpenMP efficiently handles parallel computations, the I/O and rendering tasks become the limiting factors, not benefiting significantly from additional threads.
- **Input Size Impact**: Larger input sizes lead to dramatically longer running times, showcasing the computational and I/O complexity involved in rendering larger grids.
- **Thread Utilization**: The benefit of increasing the number of threads is minimal, especially for larger input sizes, highlighting the bottleneck caused by non-parallelizable display tasks.

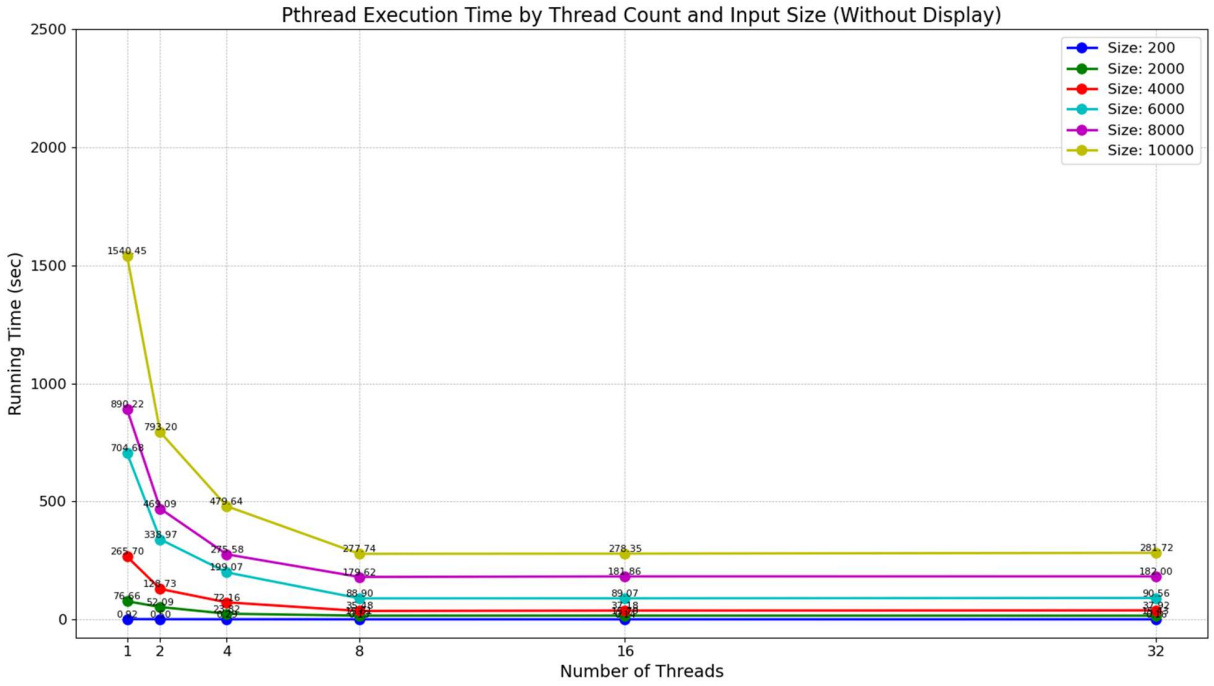
Results of Pthread Implementation

The Pthread implementation of Conway's Game of Life demonstrated a notable trend in performance based on the varying input sizes and thread counts. This analysis specifically focuses on the outcomes without display, as it offers a clearer view of the computational performance devoid of I/O overhead. The testing environment, featuring a Core i5 8th generation processor with 4 cores and 8 logical processors, was an ideal setting to evaluate the efficiency of multithreading in a real-world scenario.

1. Without display

Input Size	1 Thread (sec)	2 Threads (sec)	4 Threads (sec)	8 Threads (sec)	16 Threads (sec)	32 Threads (sec)
200	0.918148	0.497109	0.287071	0.127035	0.135927	0.156507
2000	76.658001	52.091835	23.820402	15.607831	15.786297	15.429960
4000	265.695111	128.727205	72.162929	35.478713	37.178498	37.923513
6000	704.683263	338.966566	199.073182	88.895731	89.067306	90.561788
8000	890.216624	469.086369	275.580802	179.615751	181.857672	181.996841
10000	1540.454639	793.203419	479.636168	277.738412	278.346904	281.720238

The graph displayed bellow succinctly illustrates the performance dynamics of the Pthread implementation across different input sizes (mesh sizes) and thread counts. Notably, the graph showcases running times on a logarithmic scale, offering a clearer comparison across a broad range of input parameters.

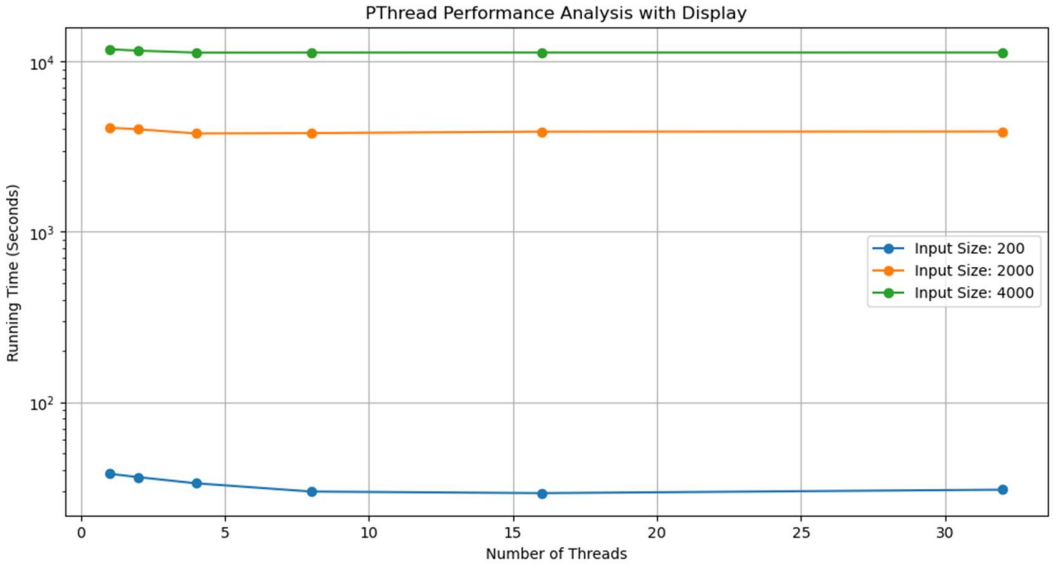


The graph for the Pthread implementation, gives us a visual representation of the execution times across different input sizes and thread counts, with each line indicating a specific input size.

Key Observations:

- **Efficiency Gains with Increased Threads**: Similar to OpenMP, the Pthread implementation shows a marked decrease in execution time as the number of threads increases, up to a point. This trend is a testament to the efficiency gains achievable through parallel processing, where multiple threads work concurrently to complete the computation faster.
- **Plateauing of Performance Gains Beyond 8 Threads**: The performance improvement ceases and even slightly reverses beyond 8 threads. This behavior could be attributed to the overhead associated with managing more threads than the number of logical processors available (4 cores, 8 logical processors in the test system). Such overheads could involve context switching, synchronization, and potential thread management complexities.
- **Larger Input Sizes Benefit More from Parallelism**: For larger input sizes (e.g., 10000 mesh points), the reduction in execution time is more substantial when increasing the number of threads, as opposed to smaller input sizes. This indicates that Pthreads parallelism is more effective for larger and more complex computational tasks, where the work can be more efficiently divided among multiple threads.
- **Optimal Thread Count Correlation with Hardware**: The optimal thread count for Pthread, similar to OpenMP, seems to be around 8. This correlation with the hardware's logical processor count suggests that maximizing thread utilization without exceeding the hardware's concurrency capabilities is crucial for achieving optimal performance.
- **Comparison with OpenMP**: While the trends in both Pthread and OpenMP are similar, minor differences in execution times and scalability might be observed due to the different ways these technologies handle thread management and parallel execution.

2. with display



The graph depicting the performance of the Pthread implementation with display enabled across different input sizes and thread counts reveals some key observations:

1. **Execution Time Increase with Larger Input Sizes**: As expected, larger input sizes correspond to longer execution times. This trend is consistent across all thread counts, indicating the computational complexity's direct relationship with the size of the data being processed.
2. **Marginal Improvement with Increasing Threads**: The graph shows that increasing the number of threads does improve performance, but only up to a point. Beyond 8 threads, there is no significant decrease in execution time. This plateau suggests that the benefits of additional parallelism are limited by other factors in the system.
3. **Impact of I/O Operations**: The dramatic increase in execution time with the display enabled suggests a significant impact of I/O operations on performance. These operations are inherently sequential and cannot be parallelized effectively, leading to bottlenecks.
4. **Non-Parallelizable Tasks and Hardware Limitations**: Other contributing factors to the prolonged execution times could be non-parallelizable sections of the code and hardware constraints. The Core i5 8th generation processor with 4 cores and 8 logical processors may not be fully optimized for handling high degrees of parallelism, especially when coupled with intensive I/O operations.

Particularly with display enabled, presents an interesting case study in the interaction between parallel computing and I/O operations. Notably, the execution time with display turned on was significantly longer compared to without display, taking up to three hours for all test cases to complete. This substantial increase in execution time, almost 12 times more than without display, warrants a detailed analysis to understand the underlying factors affecting performance.

5. Conclusion

This project investigated parallelizing Conway's Game of Life using OpenMP and Pthreads. Comprehensive testing was conducted using input sizes from 200 to 10,000 cells, with thread counts from 1 to 32. Both implementations showed significant speedups from parallelization, with optimal performance achieved around 8 threads matching the core/thread count of the test system. While OpenMP provided an easier programming model, Pthreads allowed more granular control and exposed underlying threading concepts more clearly.

A key finding was the substantial impact of graphical output on performance. Rendering grids became prohibitively slow beyond a few thousand cells. This highlights how parallelization benefits diminish in the presence of serialized operations like I/O. Overall, this assignment provided hands-on experience applying parallel thinking and multi-threading principles to optimize a computationally intensive problem. The knowledge gained on workload distribution, data decomposition, synchronization, and scaling parallel programs is valuable for developing efficient, high-performance applications.