Game of Life in Parallel

Kelsey Knowlson, Victoria Lien, Bryce Palmer

1 Introduction

The Game of Life is a cellular automaton simulation that was the first of its kind in 1970 when it was created by John Horton Conway.[1] Conway had wanted to implement an idea from the mathematician John Von Neumann to devise a machine that could create replicas of itself. Conway successfully implemented of this concept in a simulated environment and with it an entirely new branch of the study of cellular automata.[2] Today modified implementations of this game are used for a wide ranging array of topics from mirroring and predicting pattern formation in nature to analysing dynamic systems such as traffic flow or epidemiological spread.

The game is a zero-player simulation of cellular automata that runs entirely from a given initial state. It consists of a two dimensional grid that is divided into square cells each of which holds representational cell state data as either a 1 or 0. Each cell in the simulation grid must communicate its cell state with each of its eight adjacent neighbors. If a cell contains a 1 it is deemed 'alive' and if it holds a 0 it is 'dead.' The rules on culling and birthing cells can vary but for our purposes, any living cell with less than two or more than three living neighbors will die as a simulation of under or over population respectively. Any dead cell with exactly three neighbors will be resurrected as a simulation of reproduction. Each cell in the grid must communicate with all 8 of its neighbors at each time step, which can create heavy computation costs and a bottle neck effect, particularly as grid dimensions grow. To mitigate this cost, parallelization has been implemented as a method to spread weight of computation among multiple processes.

2 Hypothesis

The problem of finding the optimal partition size that maximizes the number of cells computed per unit time, constraints and limitations must be considered. Computational cost grows exponentially in terms of dimension size while communication cost grows linearly. Maximum hardware memory must be large enough to hold giant matrices that may be required to yield positive results. Under these assumptions, it is expected that using an implementation of concurrency, the optimal partition size is to be found when computation time and communication time are equivalent.

3 Naive Implementation

The naive version of Conway's Game of Life involves creating a finite dense matrix with user specified dimensions. The dimensions are strictly enforced to be square, with a size of $n \times n$. An additional 2 rows and columns of zeros were added to create padding surrounding the matrix. The padding acts as a pseudo-barrier serving the purpose of preventing out-of-bounds checks and ensuring a finite domain while allowing expedient computation.

There are user defined limitations on the number of time steps ranging from 1 to infinity. In the case of infinity, the game will terminate when all cell states have reached a stasis. Initial states were generated to maintain uniformity at each matrix size to avoid an inadvertent variable of matrix congestion.

4 Parallelization of Naive

Global grid sizes were evenly partitioned among perfect square numbers of processes with a mandated uniform distribution. Each process was allocated exclusive access to a local subgrid of size $\frac{n}{\sqrt{processcount}}$, where 'n' represents a global dimension and the process count is the total number of processes. The subgrids also include padding on their edges, initialized to zero, which will be populated with adjacent cells states post communication. At the start of each time step, all processes exchange binary vectors to represent edge cell states that will populate the padding in their respective subgrids. Inter-process communication can extend to up to eight other processes due to the nature of the game. Following communication, each process executes local computation on its subgrid.

Unfortunately, this implementation imposes a synchronization at each time step as the processes wait to receive all data before computation begins. This implementation hinders optimal parallelization and reduces concurrency. Later implementations address this issue and attempt to further optimize the algorithm.

5 Naive Implementation with Concurrency

In each individual subgrid, there exists an internal section of the grid that remains isolated from the exterior edges, therefore requiring no additional information from outside processes. In the parallelized rendition of the naive method, no distinctions are made between inner cells and outer cells of the local matrices. Instead, they are both handled as a unified computational 16384block. The amalgamation of these subgrid portions mandates full completion of communication to populate cell padding before inter-subgrid computation can initiate.

To address this limitation on speed, unique computational methods were created to make a distinction between interior and exterior cells. This design allows interior computation to commence while communication is still in progress. Once communication has terminated, the exterior cells will initiate their computation, leading to a reduction in the overall weight of computation occurring post communication.

6 Experiments

To gather experimental results for the several Game of Life implementations, a few aspects were fixed in order to maintain consistency across all tests. Firstly, the Game of Life was initialized with an agar pattern, a pattern that spans an entire plane and is periodic in space and time.[3] In other words, an agar is a pattern that will not dissipate and will continue to exist across all iterations of states. The specific pattern utilized is known in the Game of Life community as "zebra stripes." This agar pattern alternates rows of dead cells and alive cells (e.g., the first row contains dead cells, the second row contains alive cells, the third row contains dead cells, and so forth). The zebra stripes pattern requires a grid size of even dimensions, so this was taken into account for the matrix sizes tested with.

Secondly, the number of iterations (a single step or generation in the simulation) was fixed to 1,000 ticks across all tests. Experiments were ran on UNM's CARC machines, including Hopper, Wheeler, and Xena. Tests on Hopper and Wheeler were ran on the serial implementation and 1, 4, 16, and 64 processes. On Wheeler, if the quantity of processes exceeded 8, the processes were distributed among unique nodes, with 8 processes assigned to each node. For instance, the trials involving 16 processes were allocated across 2 nodes, while the experiments with 64 processes were distributed among 8 nodes. Within the "General" partition on Hopper, each individual node contains 32 CPUs. To mitigate potential overhead arising from inter-node communication, the allocation strategy aimed at optimizing intra-node communication by maximizing the utilization of CPUs within a single node. This strategy aimed to enhance efficiency by leveraging the proximity within a node, thereby minimizing the latency and potential bottlenecks associated with communication across nodes. Thus, only experiments involving 64 processes on Hopper were distributed among unique nodes, precisely two nodes in total. The experiments involved testing matrix dimensions that started at 64 and underwent a doubling progression in magnitude until reaching a size of 131,072. Note that not all tests were able to run until size 131,072 due to memory limitations on the machines or tests timing out. For example, the serial implementation was only able to calculate up to matrix size 32,768 on both Hopper and Wheeler. Tests on Xena were handled differently due to CPU allocation limitations. A maximum of 12 nodes could be requested on Xena. The experiments were conducted with 1, 4, and 9 processes, with one GPU assigned per node. The matrix sizes for the tests conducted with 1 and 4 processes remained consistent with those of the Hopper and Wheeler tests. The matrix sizes for the tests conducted with 1 and 4 processes remained consistent with those of the Hopper and Wheeler tests. However, for the experiments involving 9 processes, the matrix sizes were selected to be divisible by both 2 and 9. The initial matrix dimensions were set to 96 and underwent a doubling pattern until reaching dimensions of 98,304.

7 Results

For the Wheeler and Hopper tests, the utilization of 64 processes typically yielded the fastest completion speeds. The serial version demonstrated the least favorable performance, and the performance of parallel implementations with a single process was only marginally faster than the serial implementation. The performance of implementations with 4 and 16 processes lied between the results of the aforementioned tests, with the configuration utilizing 16 processes

outperforming the ones with 4 processes. There was some variability at smaller matrix dimensions in such a way that tests employing higher process counts exhibited less favorable performance than those with smaller process count. However, as matrix dimensions increase, the performance discrepancy subsided to the expected performance levels. For Hopper tests, it is noteworthy that tests on 64 processes generally demonstrated inferior performance compared to all other process counts, including the serial method, until around matrix size 1024×1024. This is likely due to other tests only utilizing intra-node communication while 64 processes necessitated inter-node communication. As the matrix dimension increased, the impact of inter-node communication began to diminish. The increasing computation costs made the relative impact of communication costs less extreme, leading to improvements in the performance of the 64-process experiments.

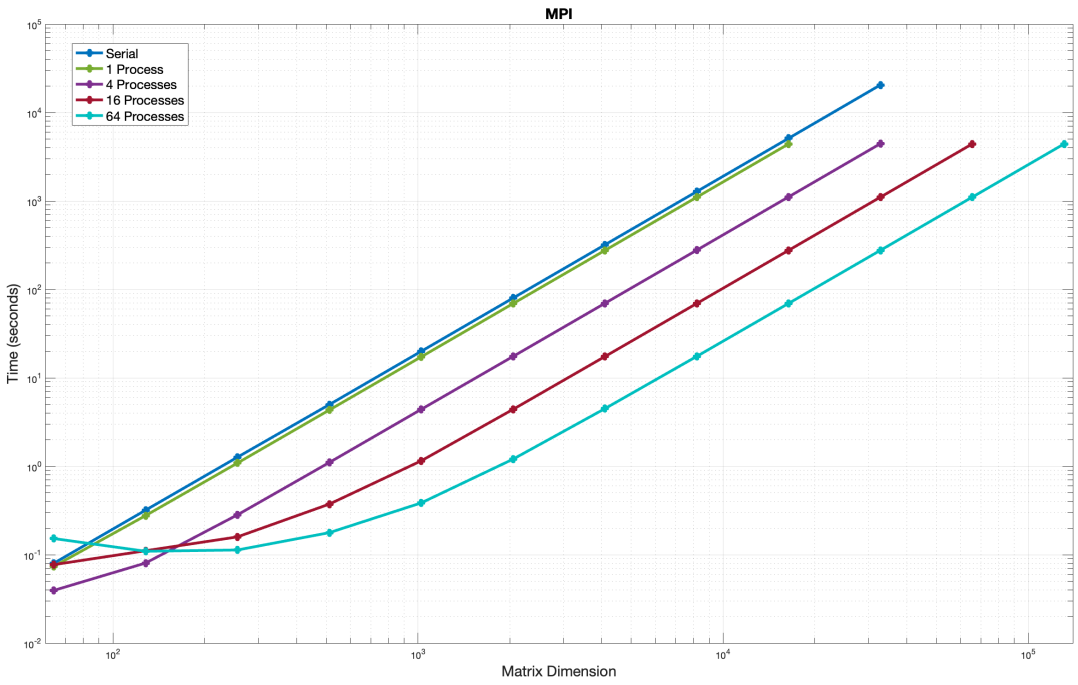

Figure 1: Serial and MPI implementation on Wheeler with 1, 4, 16, and 64 processes

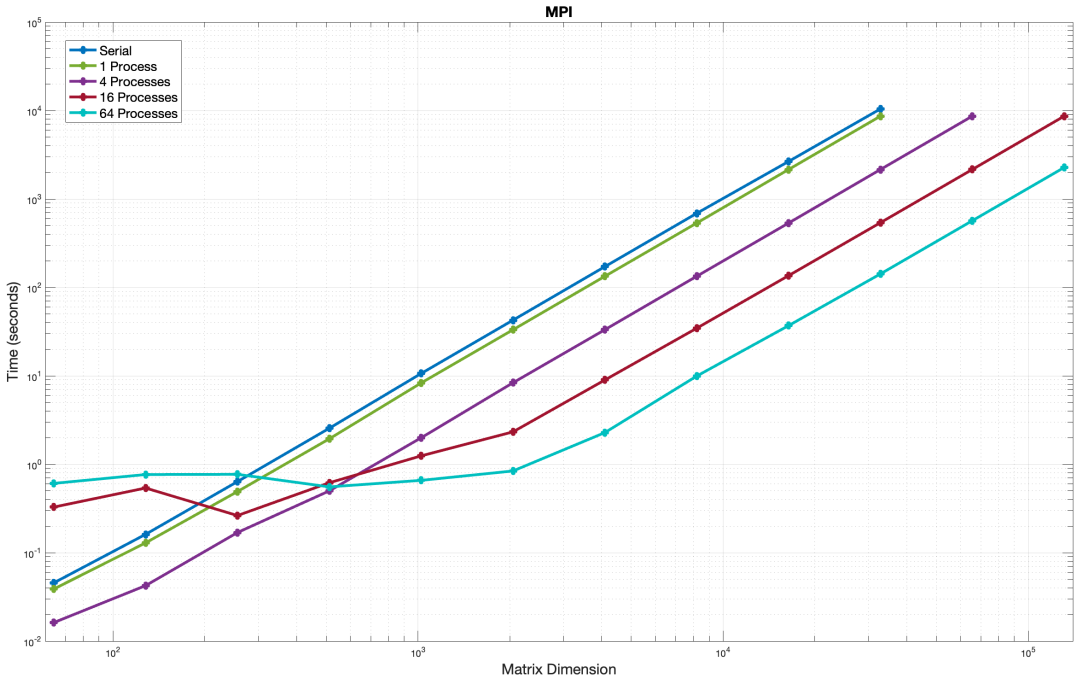

Figure 2: Serial and MPI implementation on Hopper with 1, 4, 16, and 64 processes

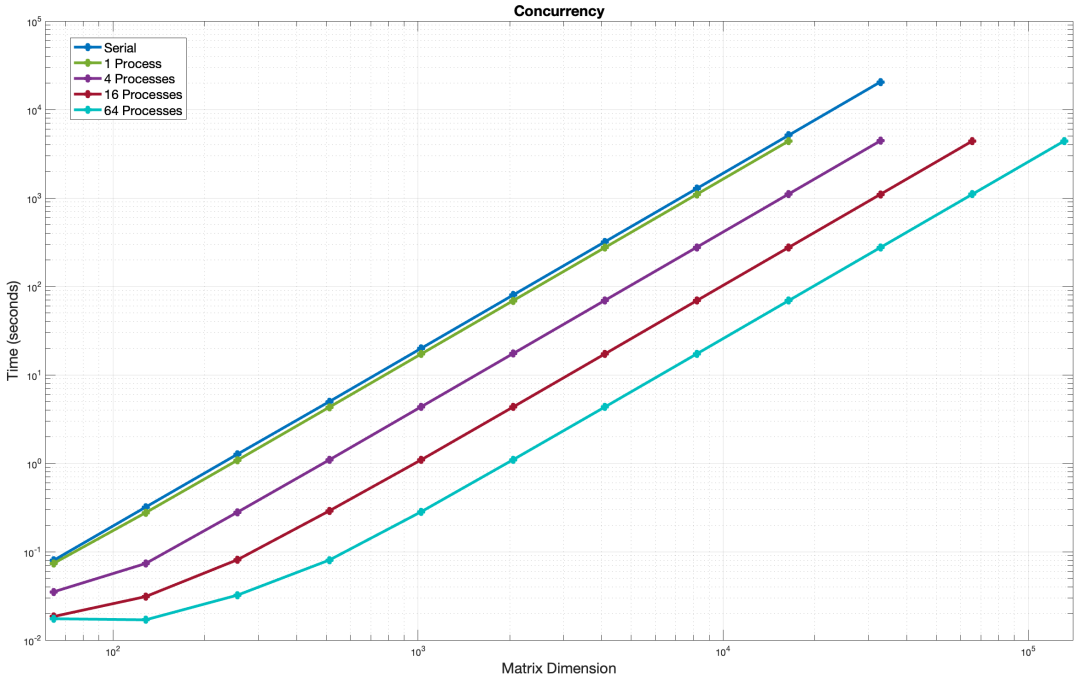

Figure 3: Serial and Concurrency version on Wheeler with 1, 4, 16, and 64 processes

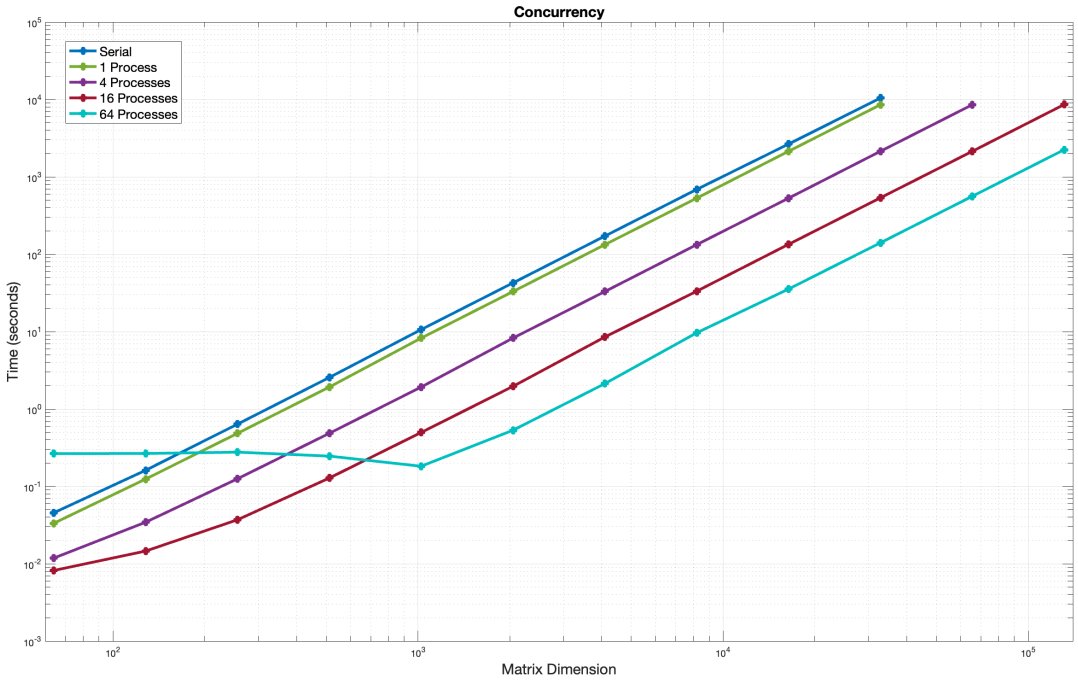

Figure 4: Serial and Concurrency version on Hopper with 1, 4, 16, and 64 processes

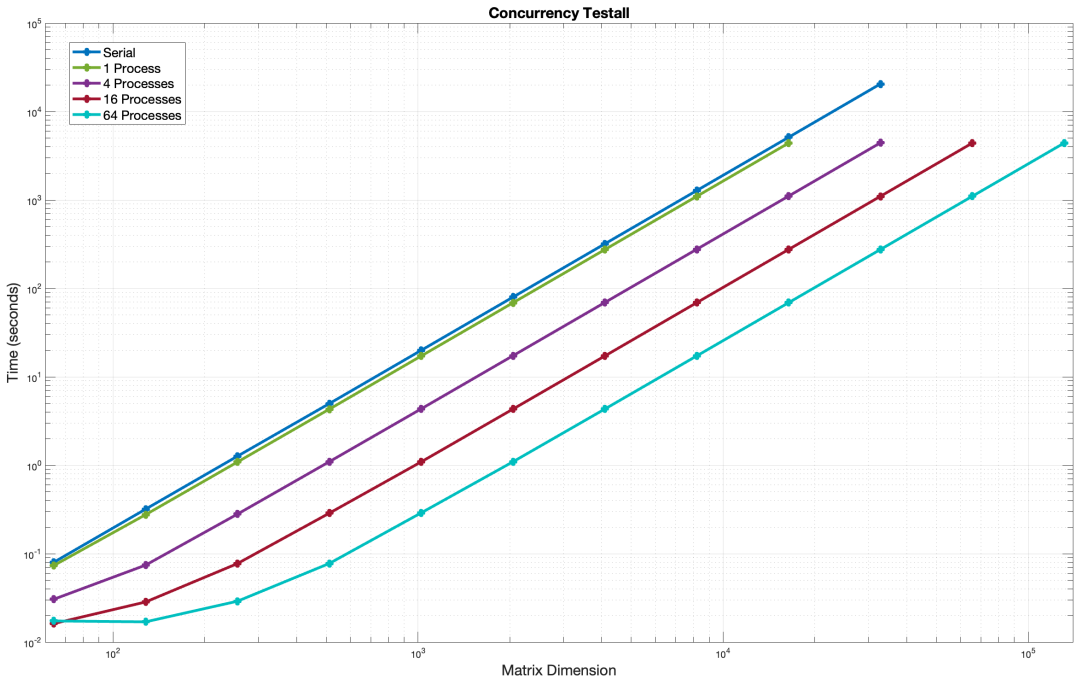

Figure 5: Serial and Concurrency with Testall on Wheeler with 1, 4, 16, and 64 processes

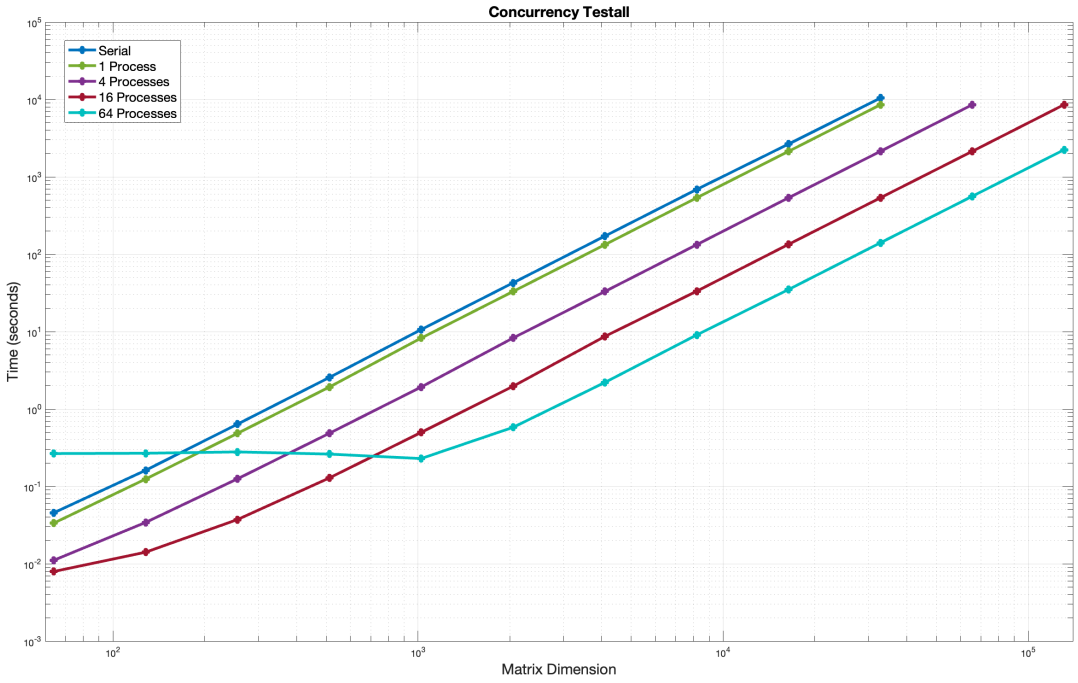

Figure 6: Serial and Concurrency with Testall on Hopper with 1, 4, 16, and 64 processes

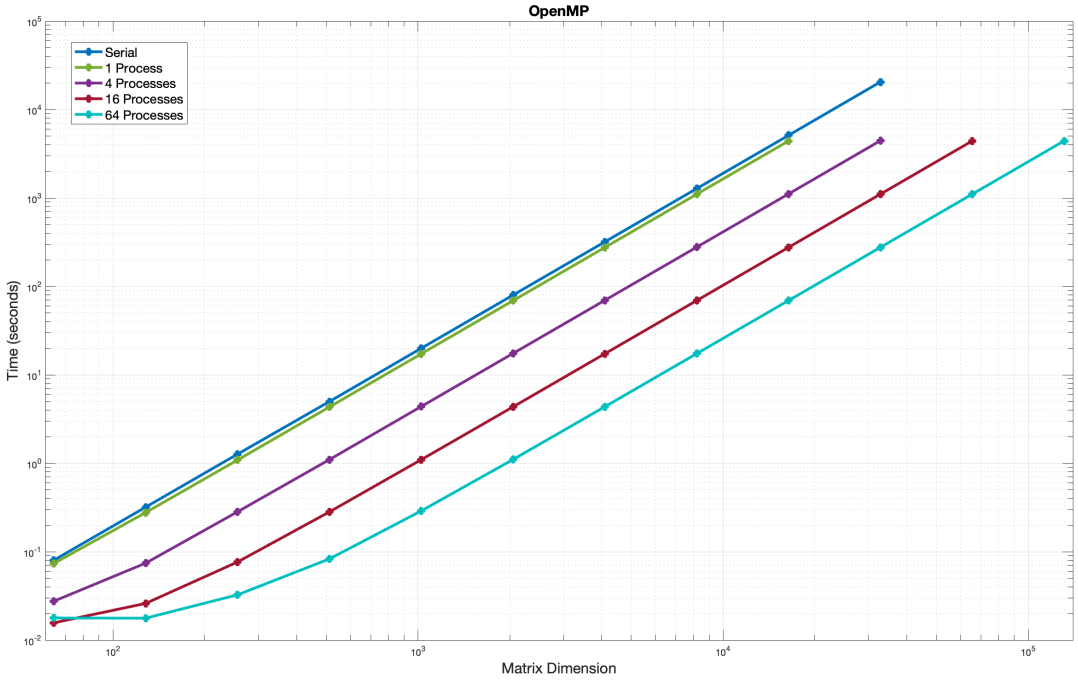

Figure 7: Serial and OpenMP on Wheeler with 1, 4, 16, and 64 processes

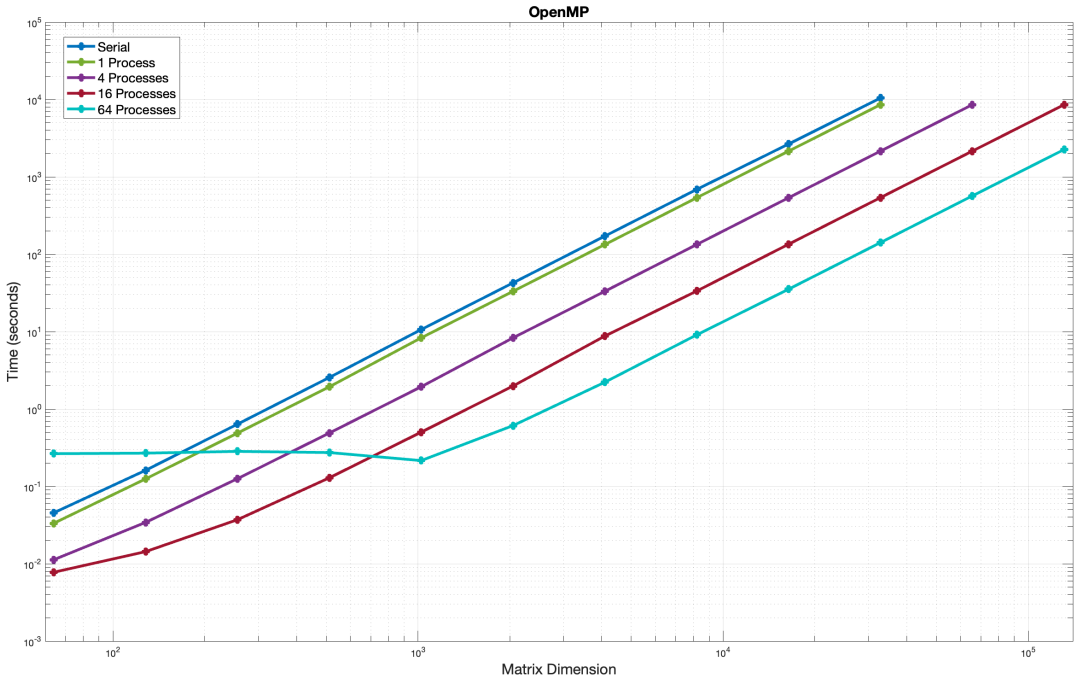

Figure 8: Serial and OpenMP on Hopper with 1, 4, 16, and 64 processes

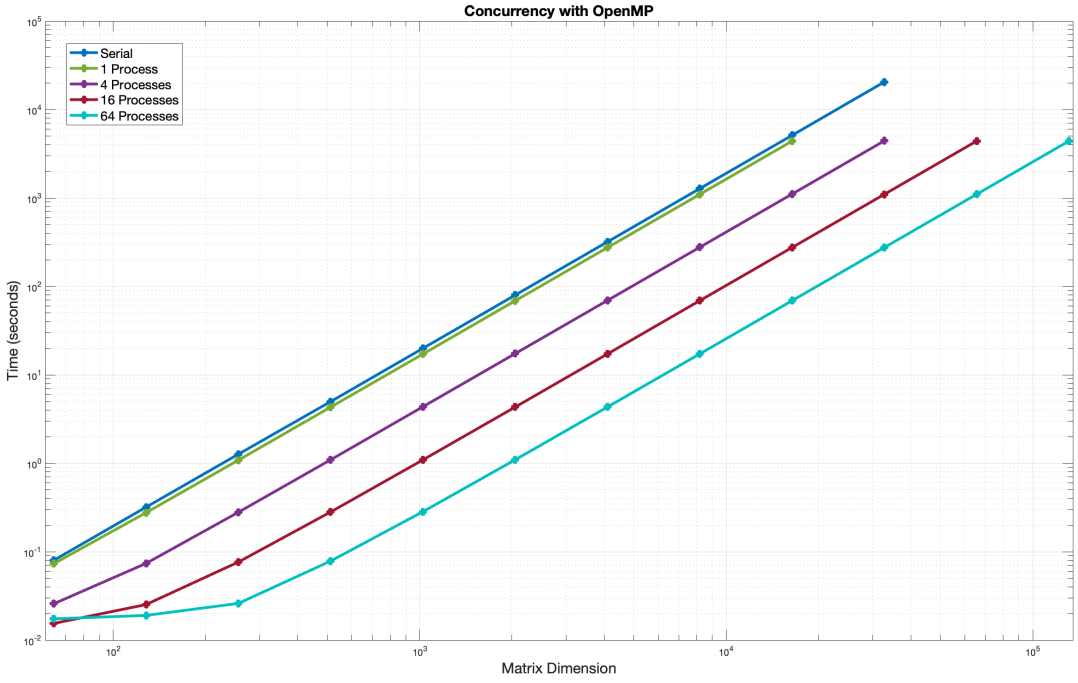

Figure 9: Serial and Concurrency with OpenMP on Wheeler with 1, 4, 16, and 64 processes

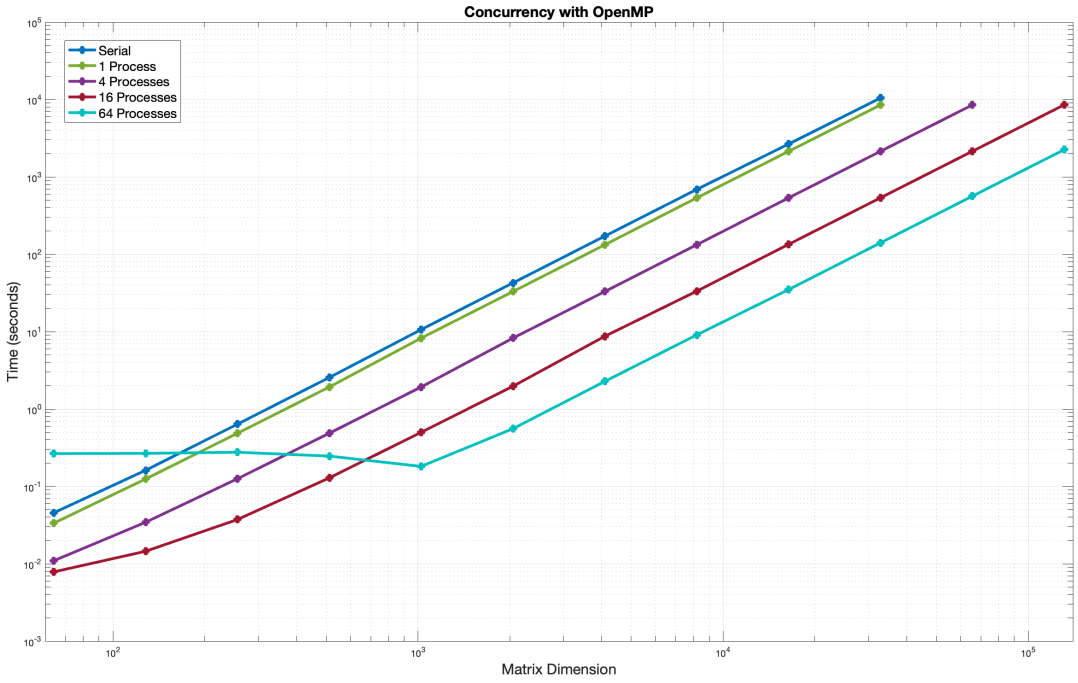

Figure 10: Serial and Concurrency with OpenMP on Hopper with 1, 4, 16, and 64 processes

It is noteworthy that, with the increase in matrix dimensions, there is a trend of convergence in the performance of the various implementations (excluding the serial version). As the matrix size increases, the variability in performance between different parallel implementations becomes less pronounced. This convergence implies that the effect of parallelization methods on performance decreases with increasing matrix dimensions, resulting in more similar performance amongst the various implementations. The diminishing effects suggest that the differences in parallelization strategies are overshadowed at larger matrix sizes due to computational needs. At smaller matrix sizes, the best performance was generally observed in the implementation employing concurrency with OpenMP. Following a decreasing order of efficiency, the subsequent implementations were the OpenMP version, concurrency utilizing Testall, the concurrent version, and lastly, the MPI version.

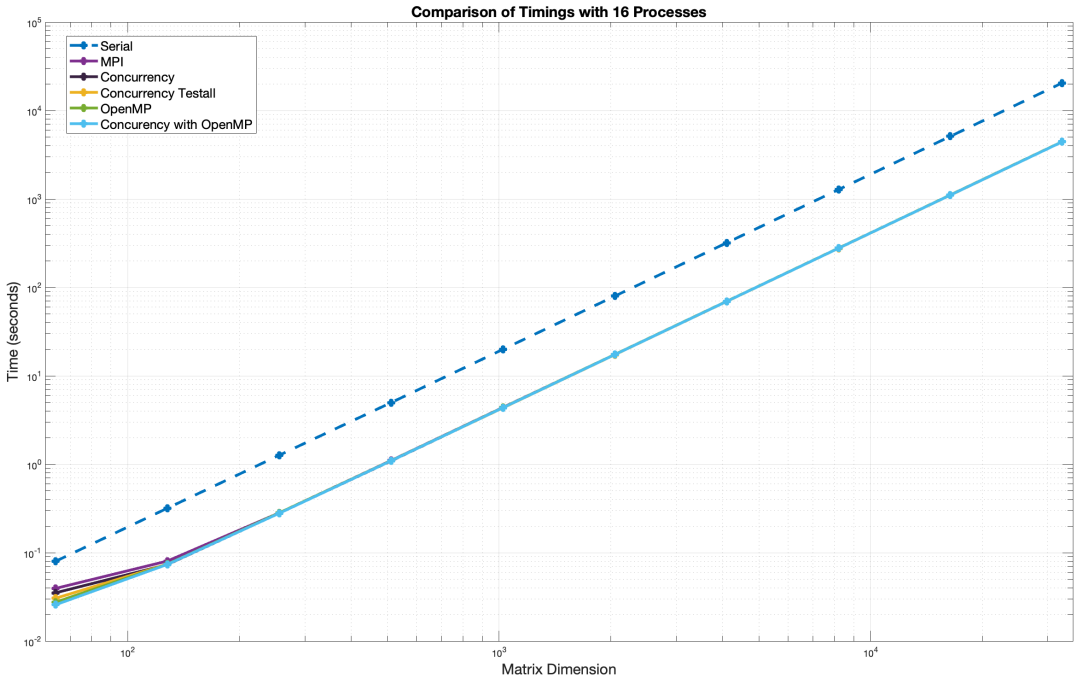

Figure 11: Comparison of different implementations at 16 processes on Wheeler

By plotting the time taken for communication and computation for the different process count, it can be observed that initially, communication costs outpace computations costs. This indicates overhead associated inter-process communication, including but not limited to latency and synchronization costs. However, as the matrix dimension increases, computational costs gradually overtake the time needed to communicate, and at matrix dimension approximately 512, communication for all process counts significantly outperforms computation. This signifies a point in which computational workload becomes a dominant factor in the overall run-time of the implementations. Notably, computation costs exhibit a substantial increase with increasing matrix sizes, but communication costs do not grow to the same extent.

Nonetheless, there appears to be an outlier in the case of 4 processes. The communication cost initially outperforms tests involving a single process. However, communication costs experiences a rapid escalation at around matrix dimension 16,384. Eventually, the cost to communicate reaches a point where it is equivalent or greater than the cost associated with computation. The tests comparing communication and computation for 4 processes consistently exhibited this anomalous behavior.

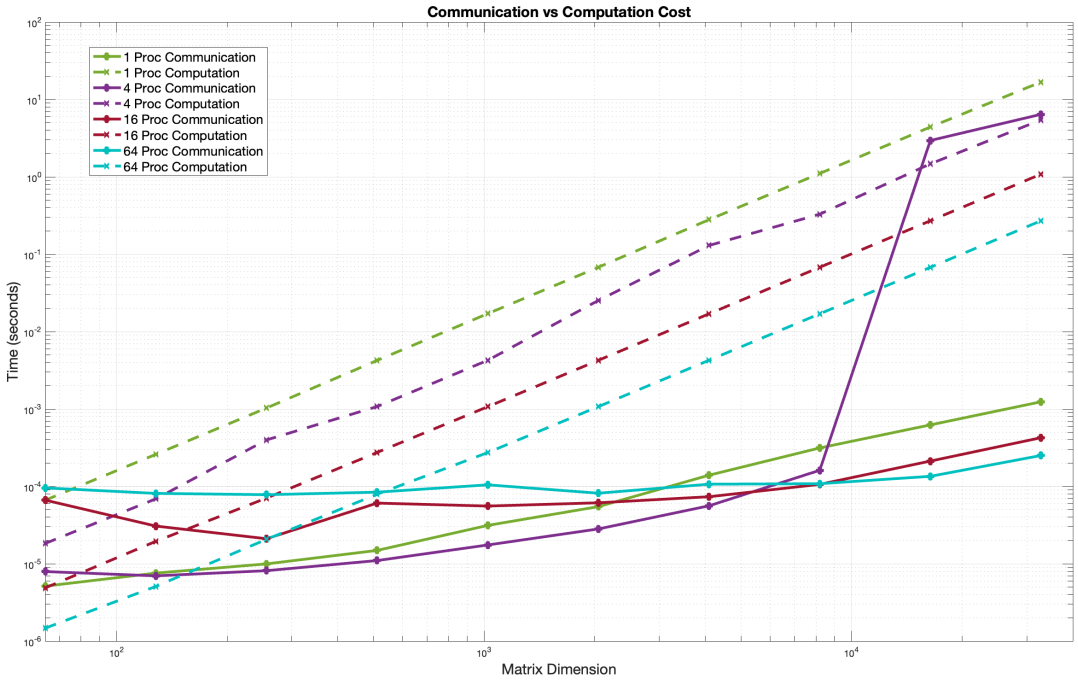

Figure 12: Communication versus computation time on Wheeler

The CUDA implementation of Game of Life demonstrated impressive speed increases compared to the other implementations by orders of magnitude. This significant speedup is due to the use of GPUs, which is consistent with the predicted increase in computational efficiency. Furthermore, the performance sclaing is consistent across the various process counts. Specifically, the execution time of the tests with 9 processes often runs faster than 4 processes, while the 4-process tests runs faster than the single-process configuration. The single-process version outperforms the other process counts initially, maintaining faster speeds until reaching a matrix dimension of approximately 256. Initially, the computational demand may not be high enough to completely utilize the GPU's resources. Consequently, the GPU can easily handle the workload of a single process, leading to faster execution speeds. However, the computational workload eventually becomes more substantial, and the advantages of parallelism using multiple processes and threads on the GPU becomes more significant.

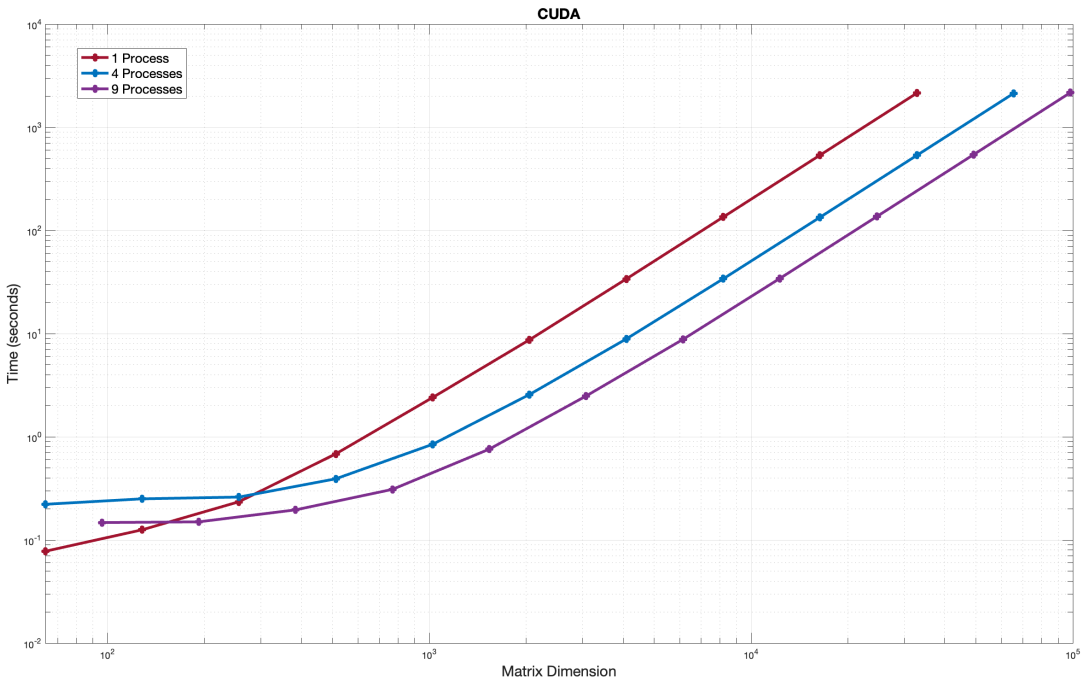

Figure 13: CUDA implementation on Xena

Additionally, the number of cells computed per second was calculated using the formula $\frac{\text{dimension}^2 \times \text{iterations}}{\text{time taken at dimension}}$. An observable trend emerges where the larger the process count, the more cells are computed per second. As the matrix dimension increases, there is also an increase in the number of cells computed per second. However, there is an eventually plateauing of cells computed per second for all process counts. This can be attributed to computation costs overtaking communication costs. As the matrix size continues to increase, the small gains in the number of cells computed per second become restricted by escalating computational costs, leading to a stabilization in the number of cells computed per unit time.

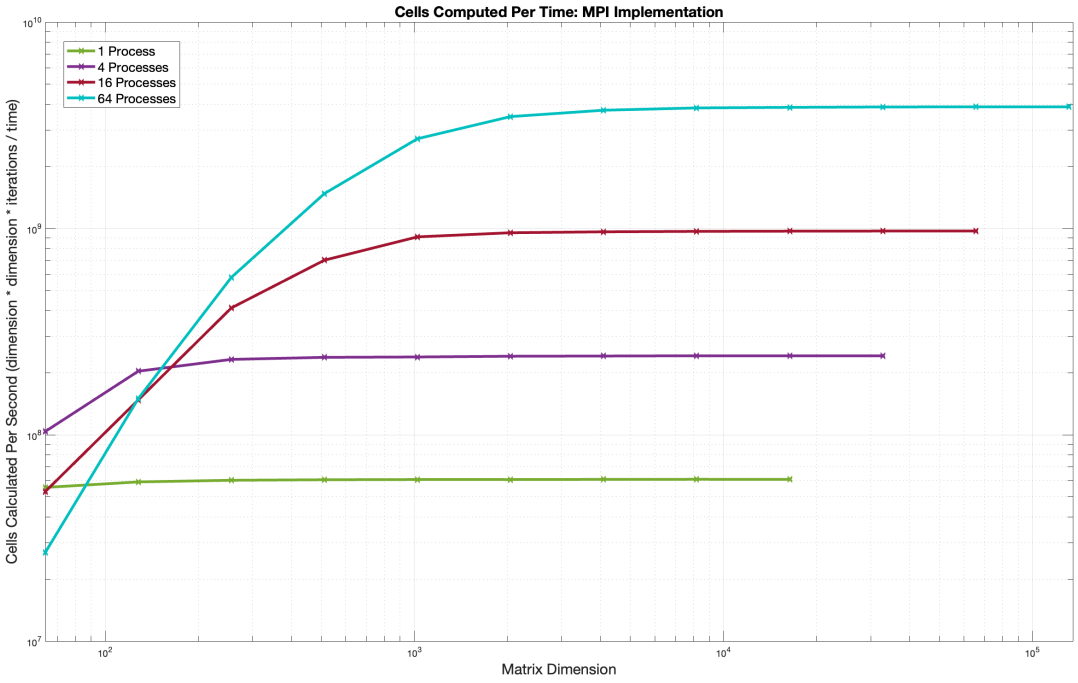

Figure 14: MPI cells computed per second on Wheeler

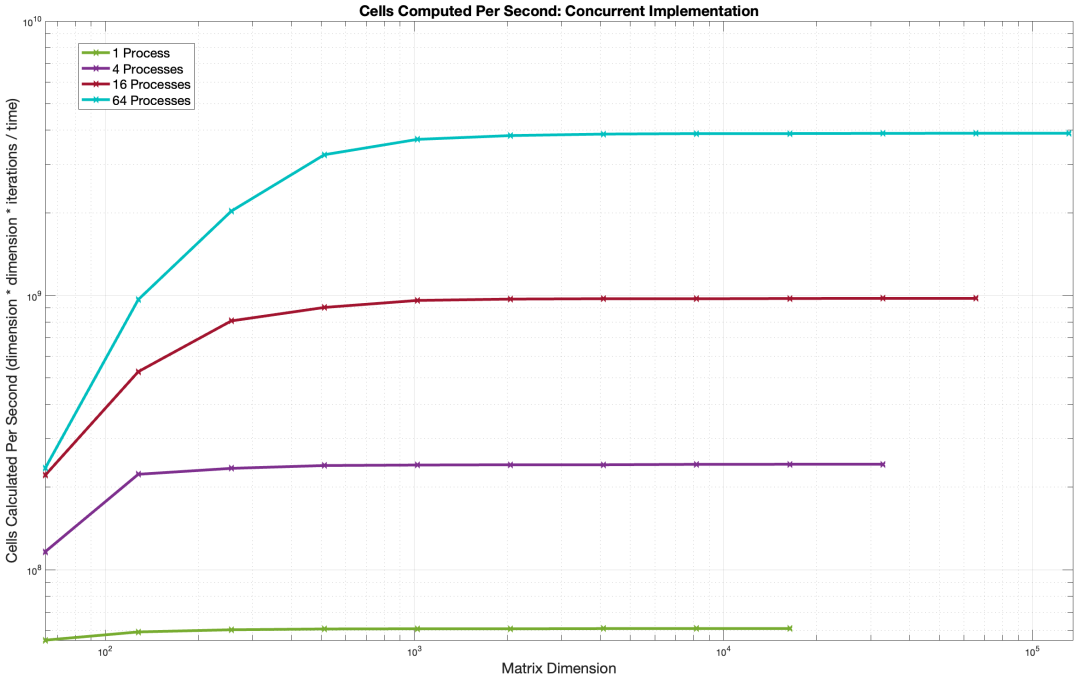

Figure 15: Concurrency cells computed per second on Wheeler

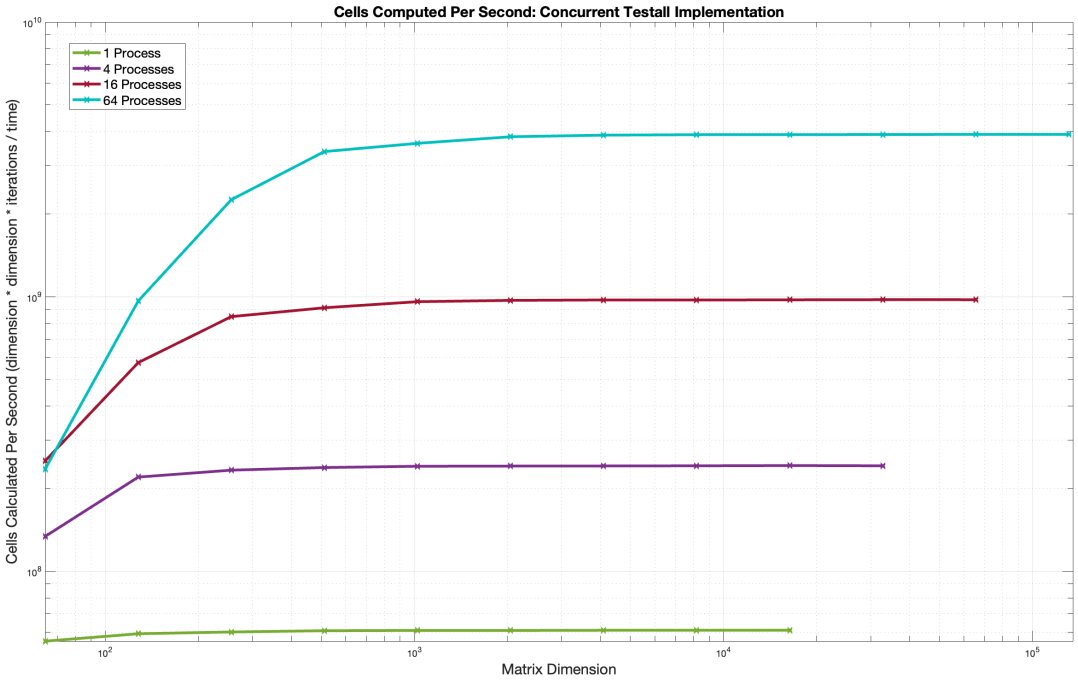

Figure 16: Concurrency Testall cells computed per second on Wheeler

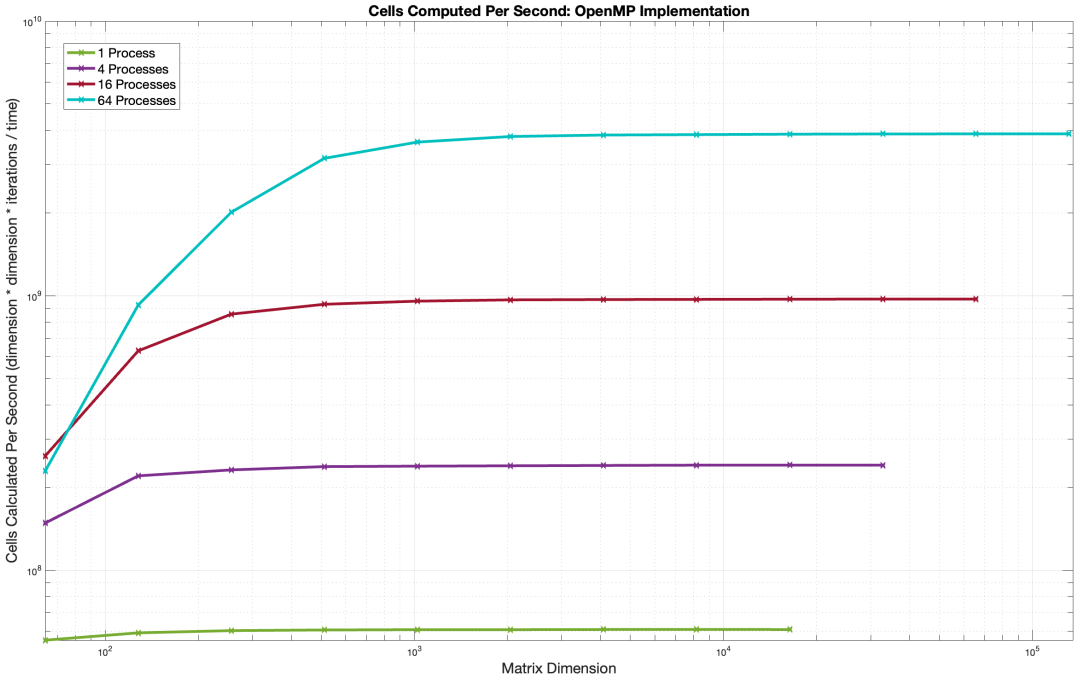

Figure 17: OpenMP cells computed per second on Wheeler

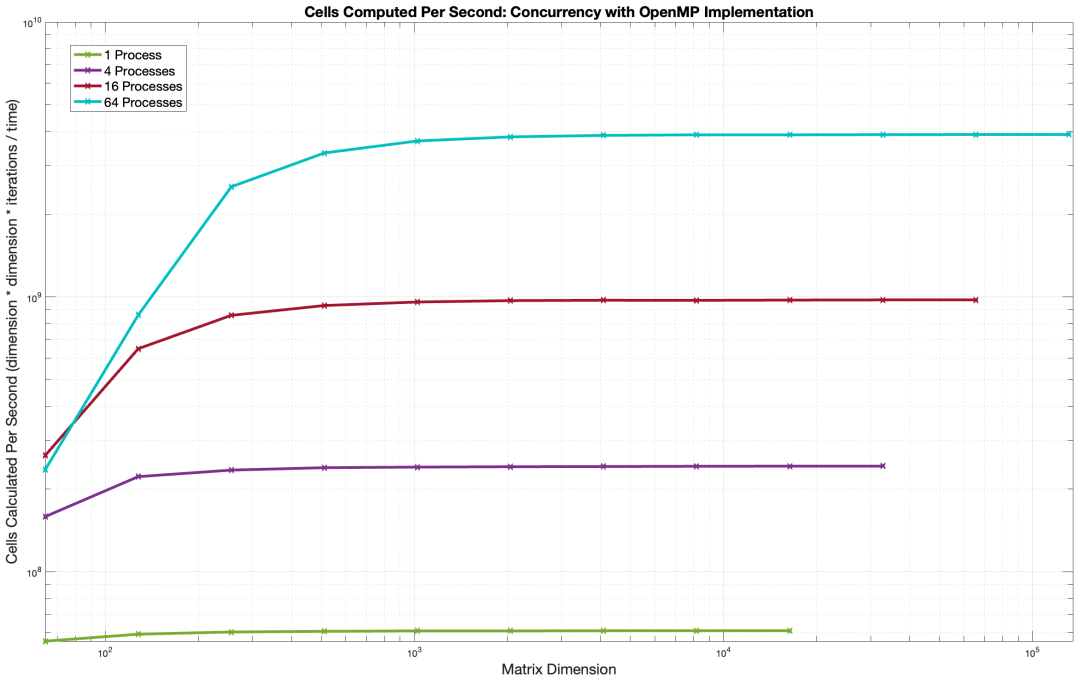

Figure 18: Concurrency with OpenMP cells computed per second on Wheeler

By utilizing MATLAB's Curve Fitting Toolbox and Symbolic Toolbox, the initial point when performance reaches a plateau was calculated. In essence, this analysis identifies when the matrix size reaches a point in which computational effectiveness reaches a threshold wherein increasing matrix size returns no further gains to computed cells per time. The resulting solution returned by MATLAB when setting the second derivative of the fitted function to zero is the matrix dimension. See table below for corresponding values.

Process Count	Matrix Dimension
1	1.149652742097743e+03
4	1.069833610275104e+03
16	7.588178873889604e+03
64	1.559829692658747e+04

Table 1: Matrix size denoting plateau on Wheeler, MPI implementation

The results contradict the initial hypothesis. The communication cost was trivial in comparison to the computational cost by an order of the input. It was observed the primary factor limiting cells computed per unit time was computation speed. Further, the computational speed reaches a bottleneck per

load on the CPU(or GPU). Using a curve of best fit approximation, the point when performance plateaus was located which did not stand with the hypothesis' claim. The points of initial plateau are located far from the intersection point of computation cost and communication cost. Furthermore, a second hypothesis has been developed suggesting the optimal partition may be located at the point of diminishing returns occurring at the inflection point of the function represented by the data set per process count. However, our curve of best fit approximation was ultimately unable to locate this point.

Regarding future research, the use of concurrency to simultaneously communicate and compute could be explored further and its complications uncovered. Also, developing a curve of best fit to solve the second hypothesis would solidify the ultimate goal of this project.

The project code can be found on GitHub.

References

[1] Wikipedia contributors. "Conway's game of life." [Online; accessed 9-December-2023]. (2023), [Online]. Available: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[2] ConwayLife.com contributors. "Conway's game of life." [Online; accessed 9-December-2023]. (2023), [Online]. Available: https://conwaylife.com/wiki/Conway%27s_Game_of_Life.

[3] ConwayLife.com contributors. "Agar." [Online; accessed 10-December-2023]. (2023), [Online]. Available: https://conwaylife.com/wiki/Agar.