COMS3008A Parallel Computing Project Report

William Hill (2115261)

November 2021

1 Conway's Game of Life

Conway's Game of Life[4] is a Cellular Automaton created by mathematician Jon Conway. It consists of a number of cells arranged in a two-dimensional grid. Each cell exists in one of two states at a time: alive or dead. A number of rules are defined (custom rules can be created) that govern the updating (i.e. evolution) of a cell's state from generation to generation, based on the states of the 8 neighbour cells (to the N, S, W, E, NW, NE, SW, SE), where border cells have neighbours on the opposite side of the board (i.e. wraparound):

- A living cell remains alive if it has 2 or 3 living neighbours
- A living cell dies if it has less than 2 or more than 3 living neighbours (isolation and overcrowding, respectfully)
- A dead cell resurrects if it has exactly 3 living neighbours

The Game of Life is a zero-player game, meaning that only the initial state of the cells needs to be defined, and afterwards, evolution to subsequent generations happens without further input.

1.1 Implementation and Approach

This mini-project aimed to design and implement an efficient parallelisation of Conway's Game of Life using the MPI parallel programming model. Firstly, a correct (and serial) baseline was established and, secondly, the parallel approach was implemented and verified against the serial output (for the same initial states).

1.1.1 (Serial) Baseline Implementation

The mini-project's baseline was implemented in serial, which was a simple way to determine the *correct* evolution of generations. The implementation requires, as input, the number of rows and columns of the 2D board, a seed from which a random board can be generated, and the number of generations to run the simulation for. An optional fifth argument can be provided which will enable the visualiser (obviously, printing to the screen is a costly operation, and needs to be disabled in order to gather accurate timing data).

A game board of the specified size (collapsed into one dimension, with conversion functions between the one- and two-dimensional coordinates) is initialised with a random state seeded from the specified seed: a random number in the range $[0, RAND\_MAX)$ is sampled, divided by RAND_MAX to obtain a number in the range $[0, 1)$ and values less than 0.5 correspond to a *dead* state whilst values greater than or equal to 0.5 correspond to an *alive* state.

The next generation is computed on a cell-by-cell basis by iterating over all the cell's neighbours, counting the number of living neighbours, and deciding whether the current cell's state remains the same or changes according to the pre-defined rules.

Additionally, a second, temporary board is defined to store each cell's updated state, so that dependence between cells only exists in a single generation (i.e. an updated state value will not contribute to the updating of any other cells' states in the same generation). Once all the updated states have been computed, the two boards are swapped (in $O(1)$ time), and the next generation begins.

1.1.2 Parallel Implementation

The parallel version performs the computation of the cells' next states in parallel. Assume there are n rows in the board and p processes, indexed from 0, in the MPI environment.

The processes exist in a logical ring interconnect topology, where the communication partners are the processes with the previous and next rank numbers. For example, process 0 communicates with processes $p-1$ and 1.

The board (generated from the same seed) is divided into a number of sub-boards, such that each process from 0 to $p-2$ has $round(n/p)$ rows and process $p-1$ has $n - round(n/p) * (p-1)$ rows (i.e. the remaining rows). The processes receive their row groups in order of their ranks: process 0 receives the first group of rows, process 1 receives the second group of rows and so on. The rows are distributed by making use of the `MPI_Scatterv` library function, after having calculated the number of elements to send to each process and their relative displacements.

Evidently, each process requires the last row from the "previous" communication partner and the first row from the "next" communication partner, making use of

the `MPI_Sendrecv` library functions, and alternating who sends first (based on rank parity) in order to avoid communication deadlocks. Since the board is divided into groups of rows, communication with columns is not needed. Therefore, the number of messages sent per generation is $2p$ and since each message is large (i.e. not broken down into smaller pieces), the communication overhead is minimised.

Now, each process can compute its share of the next generation's states, referencing the received rows when processing cells in its first and last rows, respectfully. As in the serial version, a second board is used to store the next generation's states: each process maintains a local version, which is then swapped (in $O(1)$ time) into the primary board.

Once the generation-limit has been reached, all the sub-boards are gathered (using `MPI_Gatherv`) into process 0 (with the same count and displacement arguments that were computed earlier). Process 0 now has the complete and correct state of the board, which can be output.

Since the communication is blocking, we will never have a case where different processes are processing different generations of the Cellular Automaton (i.e. there are no race conditions).

Since the amount of work in computing the next generation is the same for each cell, achieving load balancing is trivial beyond ensuring a similar number of rows are assigned to each process.

1.2 Testing and Validation

The serial version has no concrete baseline against which it can be compared to check correctness. Consequently, a visualiser was implemented so that the evolution of the cells could be easily seen: living cells are green; dead cells are red (*see Figure 1*). Its correctness is established by observation and identifying certain well-known patterns (such as "Blocks" and "Gliders") and can be compared to a correct, online implementation[2].

Testing the correctness of the parallel implementation is significantly more conclusive, since we can assume the serial version is correct. Because both the serial and parallel versions receive the same seed, they generate the same initial board. Consequently, we can simply compare the cell states after N generations (*note: it is important that the simulation not be left running for too long, as it may result in the board culminating to a common configuration, that can be reached via multiple, possibly incorrect, paths*). The two versions will stream their output to a text file, which are then compared using the `diff` shell command: if nothing is returned, then the outputs are the same and the parallel run is correct.

1.3 Performance Evaluation

The performance of the two implementations were compared over a number of parameters:

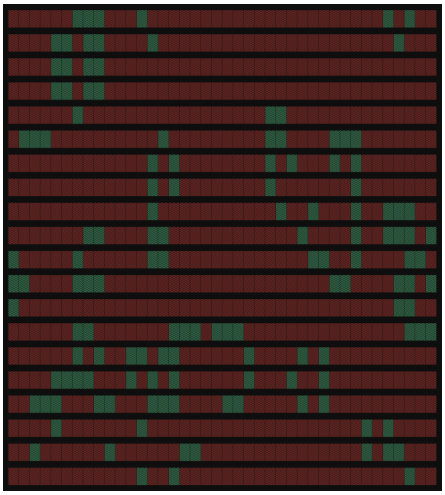

Figure 1: A generation of the Game of Life, produced by a visualisation of the serial implementation, and displayed in the terminal using unicode characters

1. Problem Size (i.e. changing board size): with dimensions $d \times d$, where $d = 400, 800, 1600, 3200, 4800, 6400$
2. Number of Processes used;

and they were evaluated according to:

1. Parallel Runtime *(see Figure 2)*
2. Speedup ($S(p) = \frac{T_{serial}}{T_{parallel}}$, for p processors) *(see Table 1 and Figure 3)*
3. Efficiency ($\frac{S(p)}{p}$, for p processors) *(see Figure 4)*

The time taken for each implementation to complete the simulation up to 50 generations was averaged over 5 runs (to account for variance in the execution time vs. the wall clock time, which was measured). Only the computational phase was timed: the initialisation and gathering of the board was not timed.

The environment on which the performance was evaluated using the university's High Performance Computing Cluster[3]. A single node was used, which has an *Intel Core i9-10940X CPU (14 cores)*.

Evidently, the Parallel Runtime (*Figure 2*) decreased exponentially over all board sizes as the number of processes increased, up to 12 processes, as expected. The reason for the increase in runtime at 14 processes is that this is nearing the physical limit at which the hardware can compute in parallel naturally. It is reasonable to expect one of the CPU cores to be responsible for other management tasks on the node *at this time*, and so when we attempt to use 14 processes, one of the physical processing elements needs to run multiple processes, thus decreased the effectiveness slightly due to increased parallel overhead. Similarly, when the number of processes is further increased, the runtime starts decreasing again,

signifying that the computational load is better balanced over the available resources.

This effect is also visible when examining the speedup (*Figure 3*): it increased linearly up to and including 12 processes (this was also where the maximum speedup achieved, for all board sizes), dropping sharply at 14 processes, before increasing linearly again, but at a slower rate than previously.

This massive speedup is indicative of an effective parallel implementation, which successfully distributes load evenly and computes concurrently.

The reason for the less-favourable performance after 12 processes is apparent when considering the Efficiency (*Figure 4*): the efficiency of the parallel implementation was fairly constant up to and including 12 processes (3.00 ± 0.25), after which it dropped by half (to 1.50 ± 0.25). This shows that the program is only scalable while the parallel overhead is low. The large initial efficiency indicates the program makes more-than-perfect use of the available resources, but as the overhead increases (when managing multiple processes per processor), the ability to use these resources fades. However, the visible effects of performance (runtime and speedup) are not notably impacted.

A similar effect would be expected when the number of nodes used increases, as the cost of communication increases significantly. Perhaps as the problem size increases even further, or the communication media improves, the efficiency would begin to return to nominal levels.

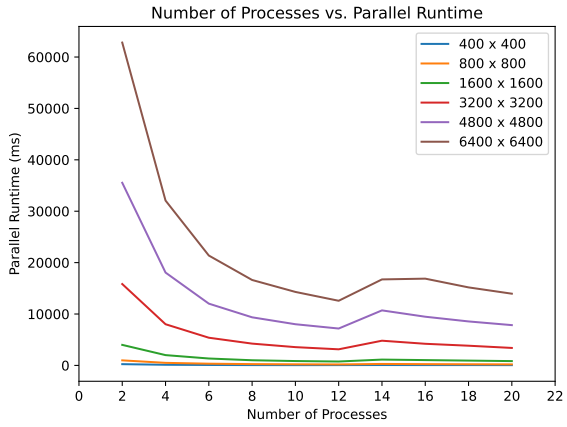

Figure 2: Graph of Number of Processes vs Parallel Runtime

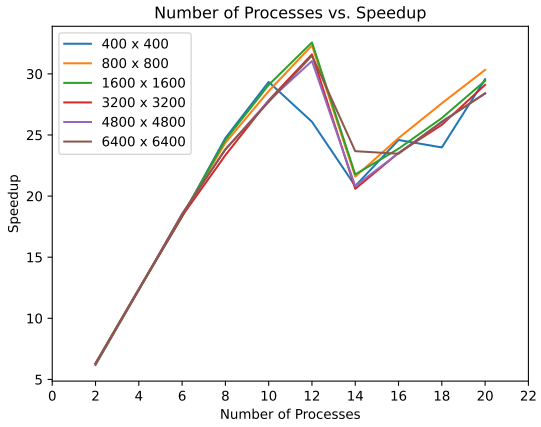

Figure 3: Graph of Number of Processes vs Speedup

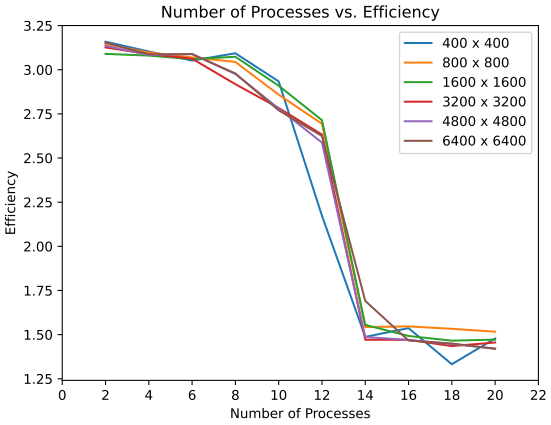

Figure 4: Graph of Number of Processes vs Efficiency

2 Dijkstra's Single Source Shortest Path (SSSP) Algorithm

Dijkstra's SSSP Algorithm is the original Shortest Pathfinding Algorithm. It is a greedy algorithm that iteratively finds the shortest path from a source node, s, to all other nodes in the graph, producing a Shortest Path Tree. The graph must be a weighted graph (even if all the edges have a constant weight), and may be directed, although it will also work on undirected graphs.

Given a graph $G(V, E, W)$, where V is the set of nodes (or vertices) in the graph, E is the set of ordered pairs representing edges, and W is the Adjacency Matrix of non-negative edge weights (if the graph is undirected, then if (a, b) exists in E, so does (b, a) and $W[a][b] = W[b][a]$), Dijkstra's Algorithm will find the shortest path, from the source $s \in V$, to another node (i.e. it becomes a *terminal* node) on each iteration, as well as update the shortest path found so far to adjacent non-terminal nodes.

2.1 Implementation and Approach

This mini-project aims to design and implement a naive parallelisation (for both Distributed and Shared Memory Systems) of Dijkstra's SSSP Algorithm. Firstly, a correct (and serial) baseline is established and, secondly, the two

Number of Processes	2	4	6	8
Board Dimensions ($d \times d$)				
400	6.31811	12.4135	18.3121	24.7476
800	6.29194	12.3921	18.4184	24.3546
1600	6.17991	12.321	18.3605	24.5879
3200	6.25297	12.3543	18.3761	23.3463
4800	6.27057	12.3338	18.5378	23.8084
6400	6.30686	12.3499	18.5298	23.831

10	12	14	16	18	20
29.3409	26.0808	20.8226	24.5905	23.9814	29.5649
28.5902	32.3285	21.5975	24.7446	27.5946	30.3297
29.0964	32.5741	21.7735	23.8747	26.3851	29.4363
27.8375	31.5973	20.5956	23.5305	25.8315	29.1094
27.8086	31.0515	20.8116	23.5189	26.0387	28.423
27.7059	31.4972	23.6744	23.4654	26.0975	28.3886

Table 1: Speedup of the parallel implementation over a various number of processes

parallel approaches are implemented and verified against the serial output (for the same input graphs).

2.1.1 Random Graph Generation

In order to solve this problem in general, an efficient way to generate random graphs is required. To solve this, the Erdős-Rényi[6] model $G(n, p), p \in [0, 1]$ is used, which can be implemented to generate a graph with n nodes, where each edge has a probability p of appearing. This can be rephrased: generate a graph with n nodes, where the edge density is p (i.e. there are $\frac{n(n-1)}{2}$ in a complete graph, so there are $p\frac{n(n-1)}{2}$ edges in the generated graph).

Implementation-wise, a random number, $r \in [0, 1]$, is sampled. If $r < p$, then the edge will be created (i.e. assigned a random positive weight in the adjacency matrix), else, the edge will be skipped (i.e. assigned a zero weight in the adjacency matrix).

However, one of the limitations of this mini-project is that the graphs must be connected, which is not something guaranteed by the Erdős-Rényi model used. To this extent, a Spanning Tree is generated first, spanning all of the nodes in the graph, and afterwards, the normal graph-generation process resumes.

Generating an undirected graph is trivial: once we have an adjacency matrix, only generate weights for the upper triangle, and copy the values to the corresponding position in the lower triangle so as to make the matrix symmetrical.

The generated graph is stored in a textfile, which will be read from when running the various forms of Dijkstra's Algorithm.

2.1.2 (Serial) Baseline Implementation

The mini-project's baseline is a serial version of Dijkstra's Algorithm. The implementation requires the path to the textfile containing the graph to solve, and constructs the graph's adjacency matrix, W, from this graph. It also requires the index of the source node, s, if all the nodes are indexed from 0 to $n - 1$.

An array, I, of the shortest distance (so far) from the source node to all other nodes is maintained (the distance from the source to itself is trivially 0). A set of terminal nodes, V_T, is maintained, to which we have already found the shortest path. On each iteration of the algorithm, the node in $V - V_T$ (i.e. non-terminal nodes) with the shortest distance is chosen to become terminal: it is added to V_T and we have found the shortest path to it. The algorithm then explores all the non-terminal nodes adjacent to the chosen node and examines if their shortest distance can be updated or not. If the chosen node has index u, then $I[v] = min\{I[v], I[u] + W[v][u]\}, \forall v \in (V - V_T)$ (i.e. if the distance to a non-terminal adjacent node through the chosen node is less than the current shortest distance, then update the distance).

The algorithm will terminate when $V = V_T$ (i.e. when all the nodes are terminal). At this point, we are guaranteed[5] to have the shortest path from the source node to all other nodes.

2.1.3 Parallel Implementation (Distributed Memory System)

The Distributed Memory System parallel version establishes an MPI environment and performs the comparison and updating of the shortest distance to each node for a single iteration in parallel. Assume there are n nodes, indexed from 0 to $n - 1$, and assume there are p processes, indexed from 0 to $p - 1$.

The processes can exist in any interconnect topology, but one which is optimised for broadcast-like operations is preferred, as this is the primary form of communication that takes place.

As with the serial version, this program requires the

name of the textfile of the graph and the start node as input arguments. Process 0 is responsible for reading the graph and establishing the adjacency matrix, W. The adjacency matrix has been collapsed to a single dimension to aid in communication, and relevant conversion functions are defined to convert between the one- and two-dimensional coordinates.

All the nodes are divided amongst the processes (into sets V_i, $\forall i \in [0, p-1]$), such that each process from 0 to $p-2$ gets $round(n/p)$ nodes, and process $p-1$ gets $n - round(n/p) * (p-1)$ nodes (i.e. the remaining nodes). Since each node and its connections are represented by a row of the adjacency matrix, the groups of rows are distributed to the processes in rank order using `MPI_Scatterv`, (into matrices W_i, $\forall i \in [0, p-1]$). To aid in this distribution, a `ROW` MPI Contiguous Datatype is defined.

Each process maintains an array, I, of the shortest distance (so far) to each of its nodes. The process with the source node will also set the distance to it to be 0, which is true trivially. Each process also maintains its own set of the overall terminal nodes, V_T.

On each iteration of the algorithm, each process determines the local non-terminal node with the shortest distance, and these are gathered (using `MPI_Gather`), to process 0, which finds the non-terminal node with the global minimum distance, and then broadcasts this node to all the processes, which add this node to their terminal sets, $V_{i,T}$. An MPI Struct Datatype, `MPI_NODE_DISTANCE`, and corresponding helper functions are defined to assist with these operations. The datatype encloses a node index and distance into a single unit.

Each process then explores all the local non-terminal nodes adjacent to the chosen node and examines if their shortest distance can be updated or not. If the chosen node has index u, then $I[v] = min\{I[v], I[u] + W_i[v][u]\}, \forall v \in (V_i - V_{i,T})$ (which is similar to the operation used in the serial implementation).

When the algorithm terminates (when $V = V_T$, or $V =_{i,T}$ for each process), the local distance arrays are gathered to process 0 (using `MPI_Gatherv`).

2.1.4 Parallel Implementation (Shared Memory System)

The Shared Memory System parallel version establishes an OpenMP environment and performs the comparison and updating of the shortest distance to each node for a single iteration in parallel. Assume there are n nodes, indexed from 0 to $n-1$, and assume there are t threads, indexed from 0 to $t-1$. The number of threads will be specified by setting the `OMP_NUM_THREADS` environment variable.

As with the other versions, this program requires the name of the textfile of the graph and the start node as input arguments. The master thread (before any team of threads is spawned) is responsible for reading the graph and establishing the adjacency matrix, W. The adjacency matrix maintains its two-dimensional structure, in contrast to the MPI version.

A single distance array, I, is defined, and a single set of terminal nodes, V_T, is defined. These two structures, as well as the adjacency matrix, will be shared amongst all threads.

For each iteration of the algorithm, t threads will be spawned at the beginning, and killed at the end. The task of finding the next node to close (i.e. to make terminal and set its final distance) will be done in two stages: firstly, the `for` construct is used to divide the distance array, I, amongst the threads, and each thread will find the non-terminal node with the minimum distance from its portion; secondly, a critical section will be defined, where each thread will compare its private minimum node (and distance) against a shared minimum node (and distance), and update it accordingly. Using a critical section prevents a race condition arising, which would lead to non-deterministic results.

A barrier has to be established at this point (using the `barrier` construct), to prevent the updating of the terminal set before all threads have completed their search. Failure to do this may result in incorrect results, since searching for the minimum non-terminal node is dependent on the contents of V_T.

The chosen node is added to terminal set, V_T, by a single thread (so as not to create unnecessary duplicates), which is achieved using the `single` construct.

Thereafter, the `for` construct is used again to divide the work of updating the shortest distance to non-terminal nodes amongst all the threads, using the same logic is the other implementations: if the chosen node has index u, then $I[v] = min\{I[v], I[u] + W[v][u]\}, \forall v \in (V - V_T)$. Since the only value in I which is read across multiple iterations is the distance to our chosen node (which will not be updated again), no race condition will arise here. However, the problem of false sharing will arise for graphs with a small number of nodes: multiple threads update elements of an array, which will lie on the same cache line, thus triggering cache coherence policies, which add extra overhead. This problem is mitigated in large graphs (especially when static scheduling is used with the `for` construct), as the distance array will become large enough to span multiple cache lines, thus limiting the number of times cache coherence policies are used.

The algorithm will terminate when $V = V_T$. Since I is shared, there is no need to do any work gathering data from the different threads.

2.2 Testing and Validation

All three programs implemented stream the contents of their distance arrays, I, to a text file when they terminate.

The serial version has no baseline implementation against which to compare it. Therefore, for several test cases (covering multiple scenarios), the results gener-

ated are compared against an online calculator[1]. Only graphs with a relatively small number of nodes were tested, due to the infeasibility of manually comparing the results for large graphs.

To test the correctness of the two parallel implementations, their results are compared against the serial results, for the same graph and start node (it is assumed that the serial version is correct). The `diff` shell command is used to do the comparisons: if nothing is returned, then the outputs are the same and the parallel runs are correct.

Each implementation also ran multiple times for the same input configuration (for the Performance Evaluation), but the results of subsequent runs were compared against the first run, and an error raised if they were different (which they may have been if non-deterministic behaviour was present). Only the results of the first run were streamed to the output files.

2.3 Performance Evaluation

The performance of the parallel implementations were compared over a number of parameters:

1. Problem Size (i.e. number of nodes in the graph, and the edge densities): with combinations `(number of nodes, density) = {(2048, 35%), (2048, 90%), (4096, 35%), (4096, 90%), (6144, 35%), (6144, 90%), (8192, 35%), (8192, 90%)}`

2. Number of Processes/Threads used;

and they were evaluated according to (for each graph size):

1. Parallel Runtime *(see Figures 5, 6, 7, 8)*

2. Speedup *(see Figures 9, 10, 11, 12)*

3. Efficiency *(see Figures 13, 14, 15, 16)*

The Edge Densities are varied to examine the effects of load imbalance: the less dense the edges are, the more likely we are to have a process/thread doing minimal works while others are still doing intensive work.

The time taken for each implementation to complete execution of Dijkstra's Algorithm was averaged over 5 runs (to account for variance in the execution time vs. the wall clock time, which was measured). Only the computational phases were timed: the initialisation of the adjacency matrix and gathering of the distance array was not timed.

The environment on which the performance was evaluated using the university's High Performance Computing Cluster[3]. A single node was used, which has an *Intel Core i9-10940X CPU (14 cores)*.

Evidently, the Parallel Runtimes *(Figures 5, 6, 7, 8)* decreased exponentially over all the graph sizes and edge densities as the number of processing elements (PEs) increased, as expected. The reason for the slight increase after 14 PEs is that the CPU we are using only has 14 cores. Therefore, after we reach 14 PEs in our implementations, multiple PEs will be mapped to the same physical processors, which obviously carries increased overhead associated with context switching. After this small spike, the decreasing trend continues, as the overall load is being better balanced over the available resources. It is also worth noting that for each Density level, then OpenMP (Shared Memory) implementation performs significantly faster than the MPI (Distributed Memory) implementation, for all numbers of PEs.

This effect is also visible when examining the speedup *(Figures 9, 10, 11, 12)*: it increased linearly (nearly perfectly) up to and including 14 PEs, before taking a sharp drop to 16 PEs (due to the imbalance of mapping PEs to physical processors), after which it started its linear increase again. This second burst of linear increase occurs at a higher rate the more nodes the graph has (the gradients are greater), showing that the parallel implementations become more effective over their serial counterpart for larger problems.

It can also be seen that for most runs, for each graph size, the runs with the higher edge density resulted in a greater speedup over time than those with the lower density graphs. This is indicative of better load balancing between the PEs, resulting in fewer idle PEs whilst computation is still ongoing.

For all but the smallest graph, the OpenMP implementation outperformed its MPI counterpart for the same graph size and edge density. This is due to larger graphs requiring far more iterations (which is linear in the number of nodes) of Dijkstra's Algorithm. This means that there are more instances where communication is required for the MPI version, which carries additional overhead not present in the OpenMP version. This additional overhead is what leads to its less-favourable performance.

In terms of Efficiency *(Figures 13, 14, 15, 16)*, the parallel implementations are fairly scalable (i.e. efficiency remains nearly constant) up to 14 PEs, only decreasing by a minuscule amount. The efficiency experiences a drop off (of nearly half its value) from 14 to 16 PEs (due to the load imbalance across the hardware), before normalising again. This shows that the parallel programs are making good use of the available resources. However, what is not expected is for the OpenMP versions to be performing more efficiently than the MPI versions (in general , Distributed System programs scale better than their Shared System equivalents). This can be attributed to the increase in communication as the graphs get larger, which implies that the MPI implementation needs to optimise its communication to either overlap with computation (which would be difficult given that the problem cannot be parallelised across iterations), or to decrease

the amount of communication occurring.

Interestingly, this Efficiency trend does not hold for the graph with 2048 nodes; rather, the Efficiency linearly decreases throughout the increase of PEs, thus demonstrating that the problem is too small to be effectively or usefully parallelised, and should only be run on a small number of PEs.

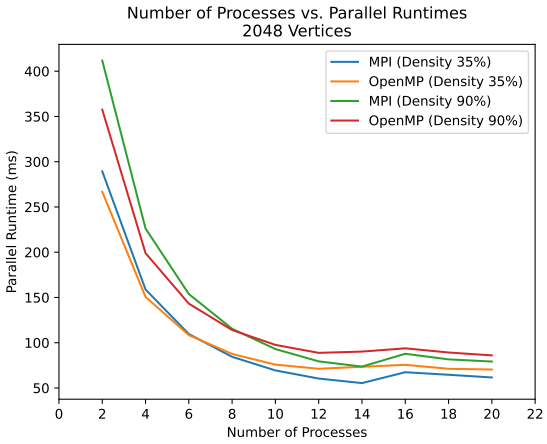

Figure 5: Graph of Number of Processes/Threads vs Parallel Runtime for a graph with 2048 nodes and various densities

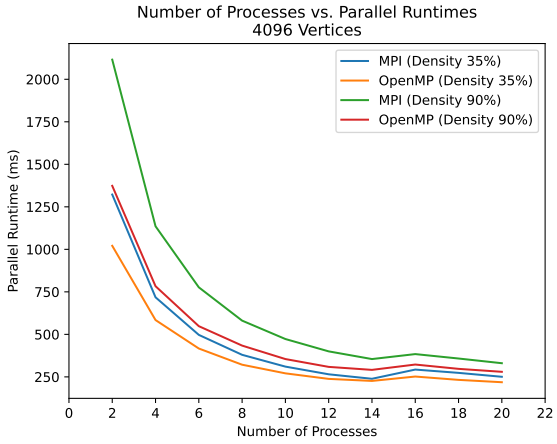

Figure 6: Graph of Number of Processes/Threads vs Parallel Runtime for a graph with 4096 nodes and various densities

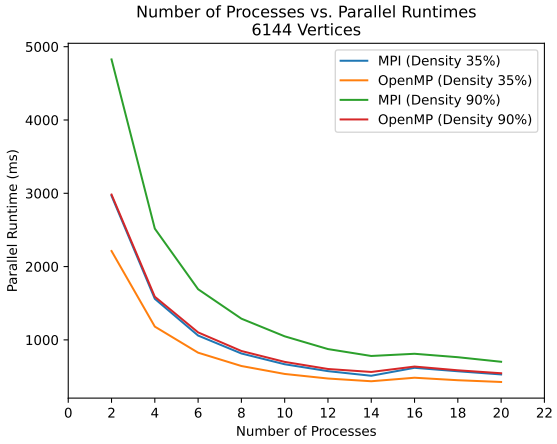

Figure 7: Graph of Number of Processes/Threads vs Parallel Runtime for a graph with 6144 nodes and various densities

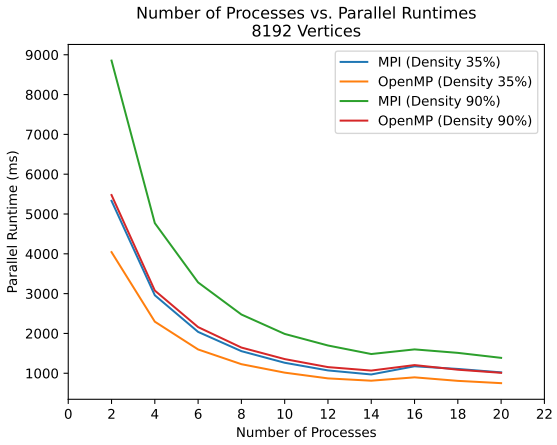

Figure 8: Graph of Number of Processes/Threads vs Parallel Runtime for a graph with 8192 nodes and various densities

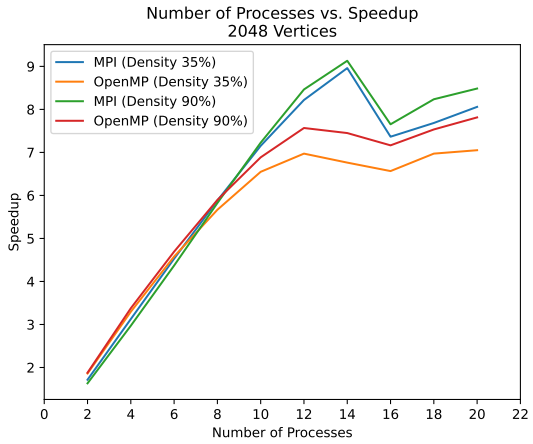

Figure 9: Graph of Number of Processes/Threads vs Speedup for a graph with 2048 nodes and various densities

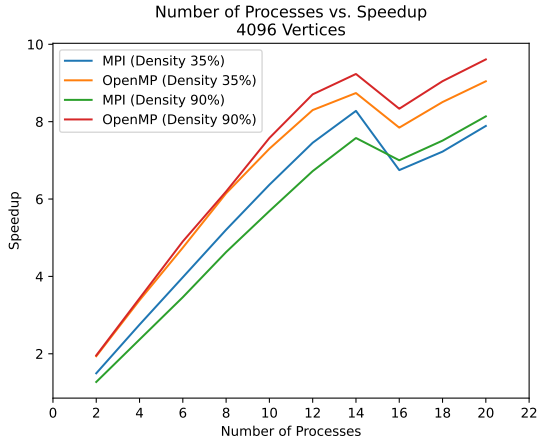

Figure 10: Graph of Number of Processes/Threads vs Speedup for a graph with 4096 nodes and various densities

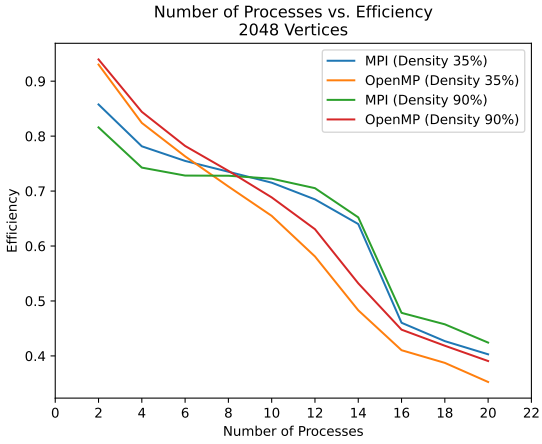

Figure 13: Graph of Number of Processes/Threads vs Efficiency for a graph with 2048 nodes and various densities

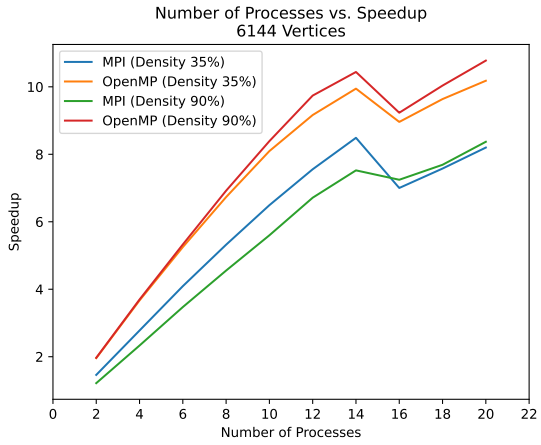

Figure 11: Graph of Number of Processes/Threads vs Speedup for a graph with 6144 nodes and various densities

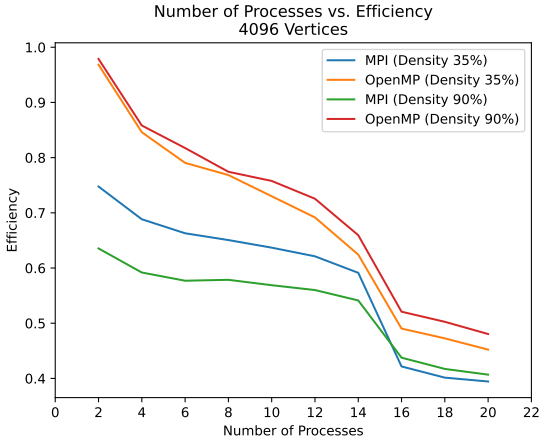

Figure 14: Graph of Number of Processes/Threads vs Efficiency for a graph with 4096 nodes and various densities

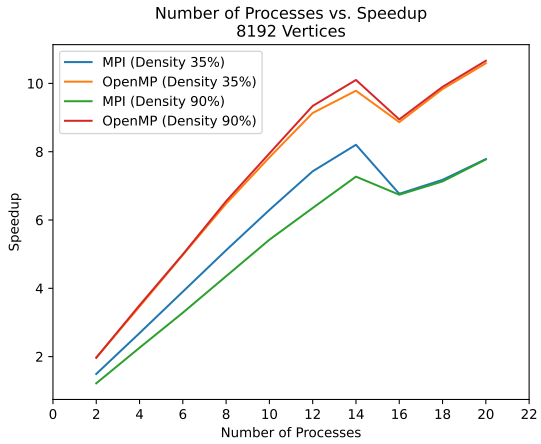

Figure 12: Graph of Number of Processes/Threads vs Speedup for a graph with 8192 nodes and various densities

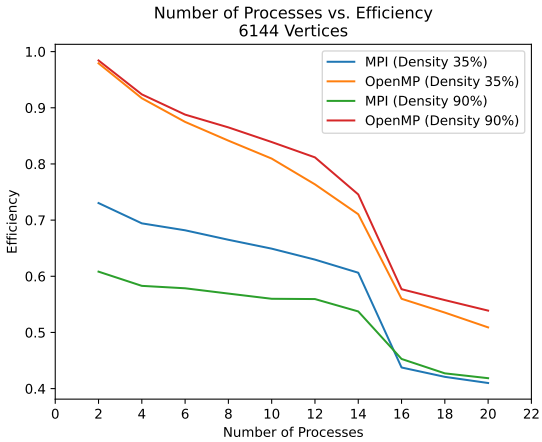

Figure 15: Graph of Number of Processes/Threads vs Efficiency for a graph with 6144 nodes and various densities

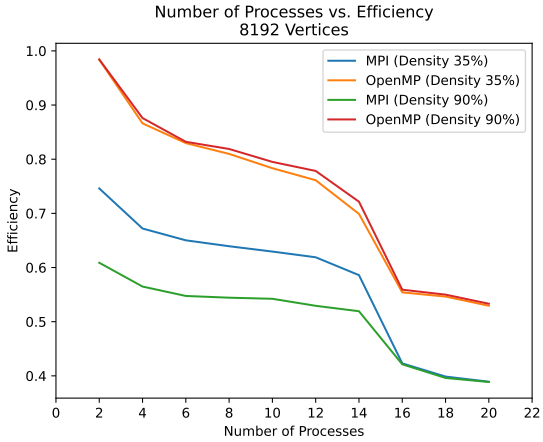

Figure 16: Graph of Number of Processes/Threads vs Efficiency for a graph with 8192 nodes and various densities

References

[1] *Graph online*, 2021. [Online]. Available: https://graphonline.ru/en/.

[2] *Play game of life*, 2021. [Online]. Available: https://playgameoflife.com/.

[3] M. S. S. unit at the University of the Witwatersrand, *High performance computing infrastructure*, 2021.

[4] M. Gardner, "The fantastic combinations of john conway's new solitaire game "life"," *Scientific American*, vol. 223, pp. 120–123, 1970.

[5] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[6] P. Erdős and A. Rényi, "On random graphs," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.