ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Parallel Systems

Game of Life Using MPI + OpenMP + CUDA

Project implemented by :

Nikolaos Galanis - sdi1700019
Pantelis Papageorgiou - sdi1700115

Contents

1 Abstract

1.1 Game of Life definition

John Conway came up with the Game of Life in 1970, at Cambridge University. The game demonstrates the fact that some simple local rules can lead to interesting large-scale life behavior(birth, reproduction and death). The game is played in a 2 dimensional grid ($N \times N$), made of cells, that can be either alive, or dead. The game does not have any players, thus it does not require any input by the user. Each cell has at most 8 neighbours, that determine its state in the next generation. The re-formation of the grid from generation to generation is done simultaneously, meaning that each state in the next generation depends exclusively in the state of the cell and its neighbours.

1.2 Our goal

Our goal is to implement this game, using Parallel programming. This program is a pure example of the value of parallelization, because of the grid, and the ability to split it to equal pieces.

1.3 Achieving satisfactory results

We want to parallelize as much of the program as we can, thus we will use parallel I/O, as well as simulating the whole game in different processors. All the testings are made in a cluster named ARGO, offering up to 80 processors.

1.4 Methods of Parallelism

We are going to develop the game using 3 different techniques: MPI, combination of MPI and OpenMP, and finally CUDA.

1.5 Amdhal's law

It is proven that despite using parallel programming, our program will not have a good speed-up compared to a serial one, unless we parallelize a great portion of it. The Game of Life has actually 3 stages: **Input**(reading the initial grid), **Simulation**(applying the rules in each generation), and **Output**(extracting stats from the final generation). If we did not parallelize the I/O, our speed-up would be negligible, so our goal moving on is to parallelize all of the instructions of the program.

1.6 Code Structure

Our code for this assignment is spitted into many files. We have a directory which contains common files used by all the methods that we have developed. For each method (MPI, OpenMP, CUDA), we have a directory containing all the useful files. Finally, in the directory misc, there are various input and output files,

1.7 Input

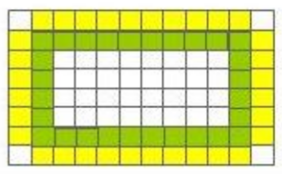
The program takes as input a $(N \times N)$ grid, which is provided in a simple text file. The data (0s and 1s) must be all in one row. So, if for example we want a (4×4) grid, we are going to provide a file containing 16 characters.

2 Data Distribution Design

2.1 Splitting the Grid

Our goal is to equally split the data into a 2D array of processes. Thus, having a 2d grid, we must compute how many rows and how many columns will each process consider. We accomplish that with the function "split_grid". We have a struct called subgrid_info that stores the number columns and rows that each process if held accountable of, and a division factor, that will help us later in order to identify which portion of the grid belongs to each process.

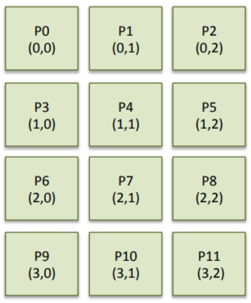
In each subgrid, we add 2 extra rows and 2 extra columns, in order to have halo points. The structure of each subgrid is like the following picture.



Those points are used to get the edge cells from the process' neighbours, in order to make the correct computations for all the cells of the current process, both internal and external.

2.2 Processes' topology

Inside main function, we create the Cartesian topology of the processes, by calling the appropriate MPI functions. The result is the following topology for an example of 12 processes.



However, we are only going to set the number of processes into a number who's square root is a natural number, in order to easily split the grid in equal square pieces.

After constructing the topology, we must find the neighbors of each process, in order to establish a communication between them later on. This code can be found in the file mpi_utils, in the function compute_neighbors. We use the MPI_Cart function family, as instructed, in order to easily find all the 8 neighbors of each process. If a process does not have a specific neighbor, we denote it as MPI_PROC_NULL.

3 MPI Code Design

Our goal was to design the MPI code by following the given instructions. Our main objectives were:

- Reduction of idle time
- Reduction of unnecessary computations
- Parallelization of a great portion of the program
- Use of datatypes

The abstract goal is to have all the process communicating by exchanging their outer-matrix data in every loop. Therefore, having already established the processes and their neighbors, we now must find the type of the data we want to swap.

3.1 Datatypes

An option is to exchange characters, one at a time. This however will cause great delay, thus we opted for the creation of datatypes. We have 2 different datatypes: a row_datatype and a column_datatype, both of them being defined in the run_game function.

3.2 Communication

In the same function, we implement the algorithm for the communications between the processes, as defined in the instructions. We opt to use MPI_Send_Init and MPI_Recv_Init, because in every loop we are dealing with the same neighbours. This type of persistent communications, save as time, and results into a more clean code. In order for the processes to synchronize correctly, we use the MPI_WaitAll function. Also, another twitch that we apply, is the use of the receive functions before the send ones, thus the process are ready to receive messages.

3.3 Checking for lack of changes

Every once in a while, we check the grid in order to see if they are any changes to our universe. Using MPI_AllRreduce, we are able to see whether the grid remains the same repeatedly. If that's the case, we shall stop the simulation, because there is not much more to observe.

3.4 Limiting serial computations in the main loop

We use dynamic arrays to implement the grid, thus we are able to switch between the pointers used to represent the grid after each loop. With this trick, we are able to skip a nested for loop, to update the before and after states after a generation.

In order to lower even more the switching between registers and main memory(which require valuable time), we limit value assignments by not having temporary variables, but instead having bigger operations.

3.5 Using the advice from instructions

In general, we have used each one of the instructions that we were given in the document "Design and development instructions".

4 Time, Speedup and Efficiency Measurements

The most important aspect when using parallel computing, is the time benefits that we get. There are a few metrics that we are taught, that can show whether or not the application of parallelism was successful. Those metrics are:

- **Parallel Run Time**: he time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes execution.
- **Speedup**: The ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

$$S = \frac{T_s}{T_p}$$

- **Efficiency**: The ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized.

$$E = \frac{S}{p} = \frac{T_s}{p \times T_p}$$

We are going to test our program in the cluster Argo.

When it comes down to MPI and Hybrid MPI + OpenMP code, we are going to use 1,4,9,16,25,36,49 and 64 processes, all powers of 2 so we can equally split our grid. We are going to test the above metrics for big sizes of grid, beginning with all the previous processes' selections' LCM, and each time by doubling the grids' size. or better representation we decided to separate the plots into those of Small and Big sizes. All our observations are summarized in the conclusions.

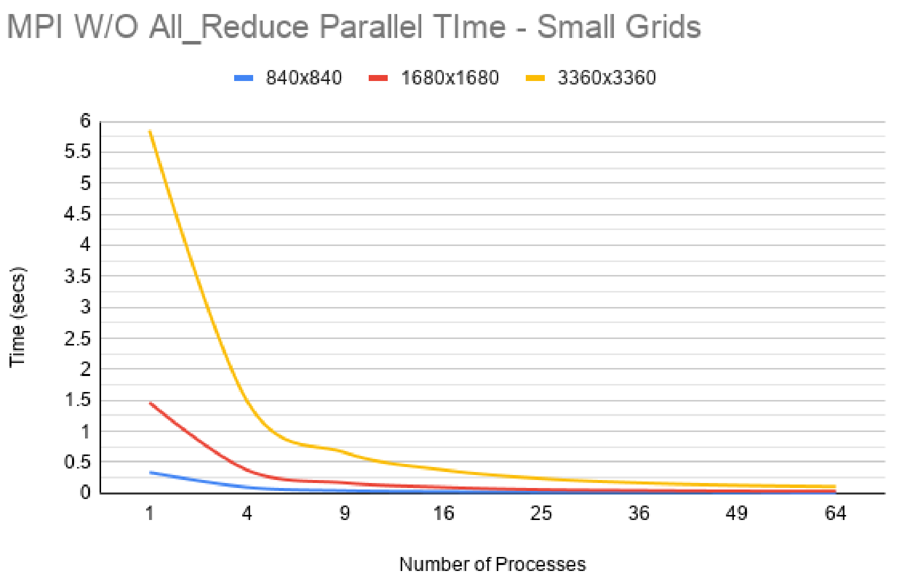
4.1 MPI

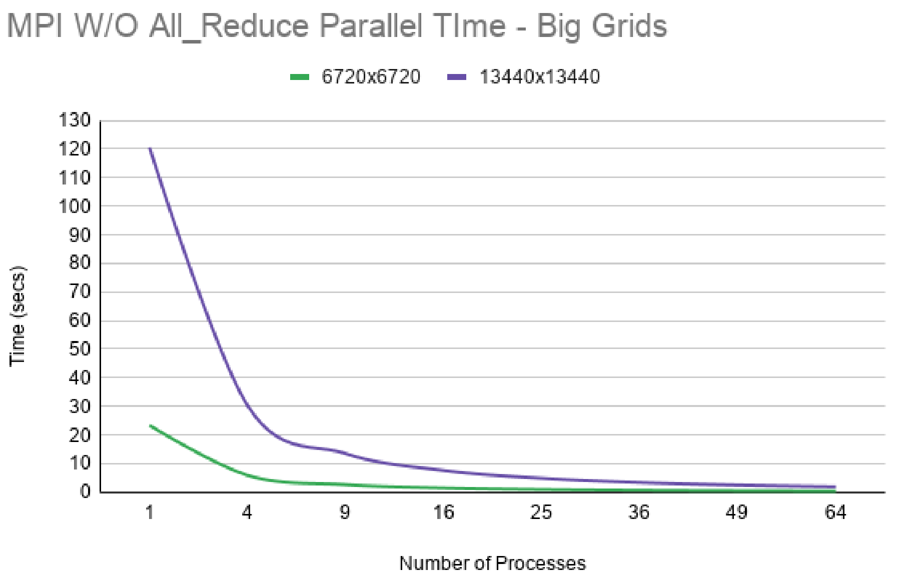
In our code, we have implemented an if statement, that checks whether 2 generations are equal, with the use of MPI_All_Reduce. We are going to check the metrics with, and without this option enabled.

4.1.1 Without AllReduce

Parallel Time

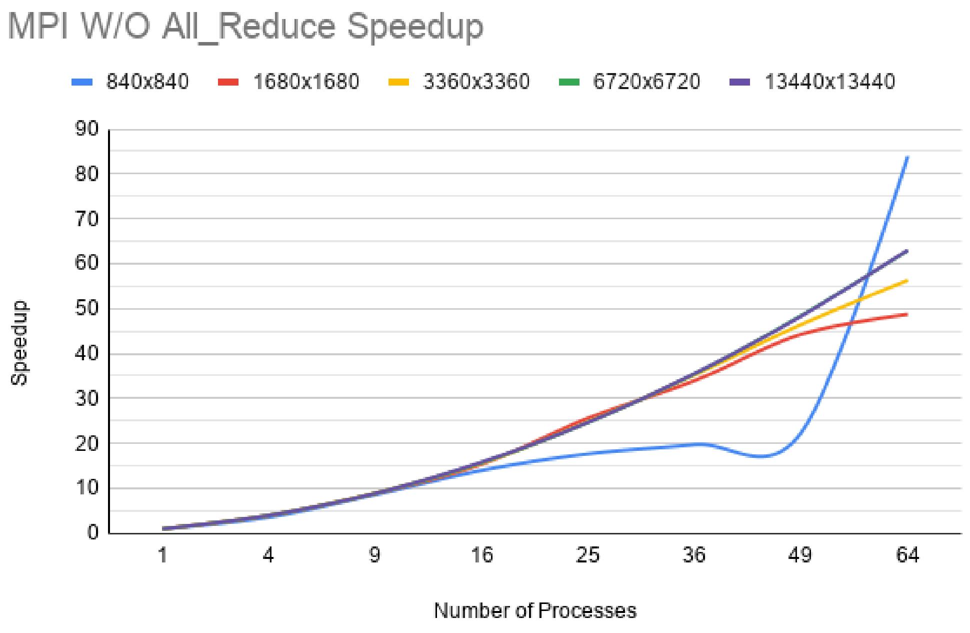
Grid Size / Processes	1 = 1 * 1	4 = 2 * 2	9 = 3 * 3	16 = 4 * 4	25 = 5 * 5	36 = 6 * 6	49 = 7 * 7	64 = 8 * 8
840 x 840	0.336	0.094	0.039	0.024	0.019	0.017	0.015	0.004
1680 x 1680	1.462	0.369	0.165	0.095	0.057	0.043	0.033	0.03
3360 x 3360	5.858	1.47	0.652	0.375	0.236	0.166	0.126	0.104
6720 x 6720	23.415	5.886	2.631	1.488	0.9484	0.659	0.483	0.372
13440 x 13440	120.577	30.243	13.501	7.6054	4.877	3.387	2.494	1.912





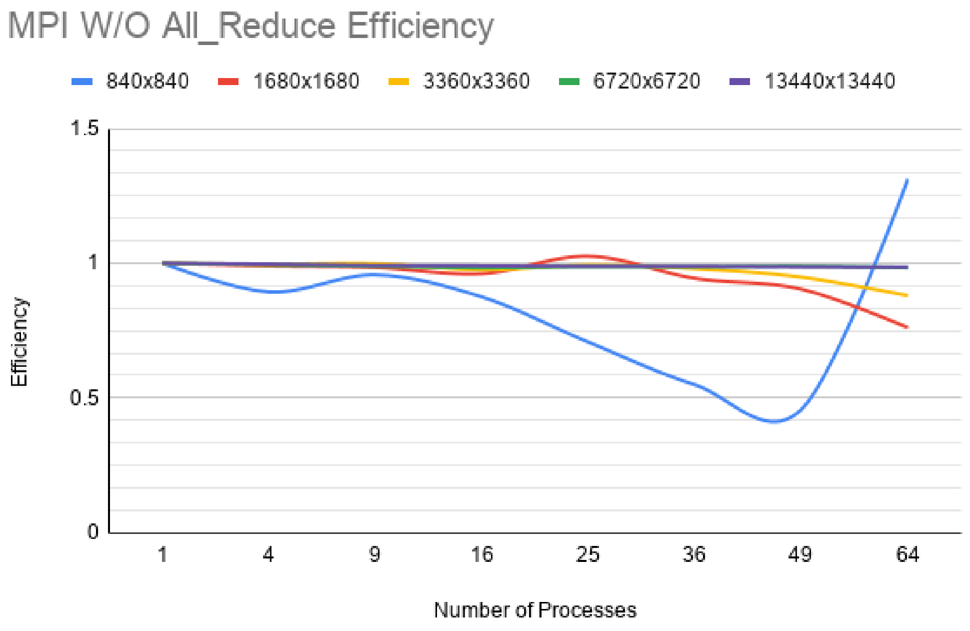
Speedup

Grid Size / Processes	1 = 1 * 1	4 = 2 * 2	9 = 3 * 3	16 = 4 * 4	25 = 5 * 5	36 = 6 * 6	49 = 7 * 7	64 = 8 * 8
840 x 840	1	3.574468085	8.615384615	14	17.68421053	19.76470588	22.4	84
1680 x 1680	1	3.962059621	8.860606061	15.38947368	25.64912281	34	44.3030303	48.73333333
3360 x 3360	1	3.985034014	8.984662577	15.62133333	24.8220339	35.28915663	46.49206349	56.32692308
6720 x 6720	1	3.978083588	8.899657925	15.7358871	24.68894981	35.53110774	48.47826087	62.94354839
13440 x 13440	1	3.986939126	8.930968076	15.85412996	24.72360057	35.59994095	48.3468324	63.06328452



Efficiency

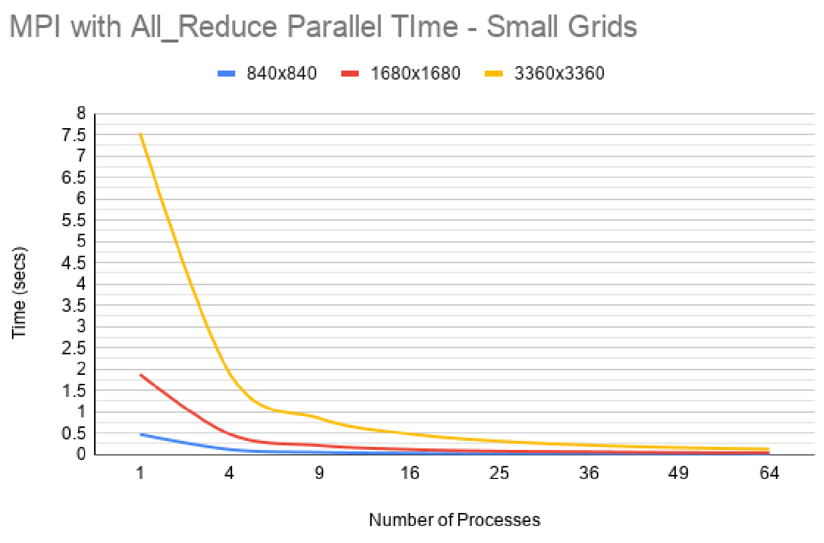
Grid Size / Processes	1	4	9	16	25	36	49	64
840 x 840	1	0.8936170213	0.9572649573	0.875	0.7073684211	0.5490196078	0.4571428571	1.3125
1680 x 1680	1	0.9905149051	0.9845117845	0.9618421053	1.025964912	0.9444444444	0.9041434756	0.7614583333
3360 x 3360	1	0.9962585034	0.9982958419	0.9763333333	0.9928813559	0.9802543507	0.9488176223	0.8801081731
6720 x 6720	1	0.994520897	0.9888508805	0.9834929435	0.9875579924	0.986975215	0.9893522626	0.9834929435
13440 x 13440	1	0.9967347816	0.9923297863	0.9908831225	0.988944023	0.9888872486	0.9866700489	0.9853638206

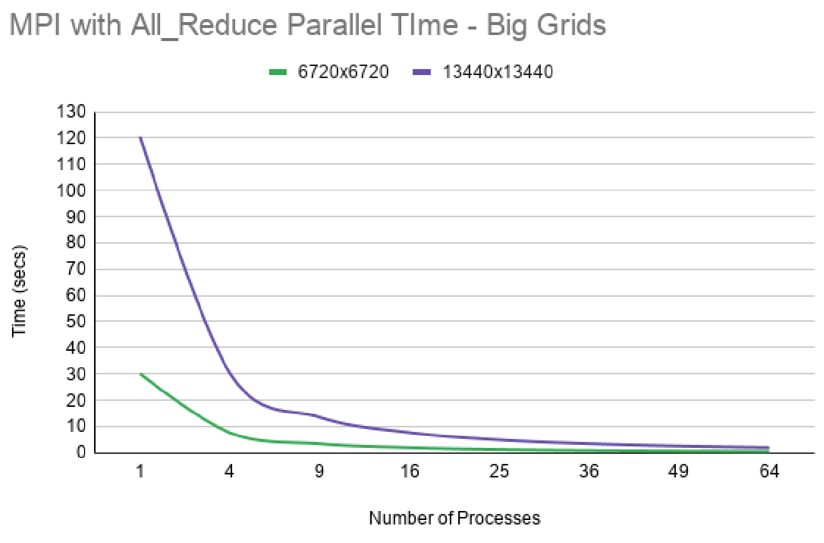


4.1.2 With AllReduce

Parallel Time

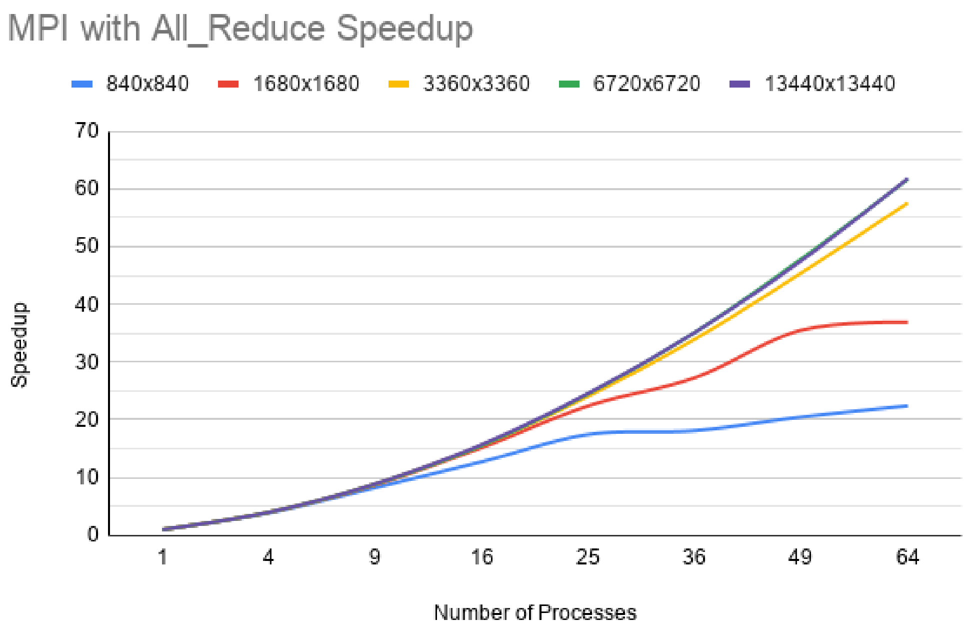
Grid Size / Processes	1 = 1 * 1	4 = 2 * 2	9 = 3 * 3	16 = 4 * 4	25 = 5 * 5	36 = 6 * 6	49 = 7 * 7	64 = 8 * 8
840 x 840	0.471	0.122	0.057	0.037	0.027	0.026	0.023	0.021
1680 x 1680	1.883	0.476	0.215	0.124	0.084	0.069	0.053	0.051
3360 x 3360	7.541	1.892	0.848	0.484	0.313	0.222	0.166	0.131
6720 x 6720	30.154	7.582	3.397	1.938	1.23	0.858	0.63	0.489
13440 x 13440	120.603	30.249	13.512	7.667	4.902	3.432	2.537	1.952





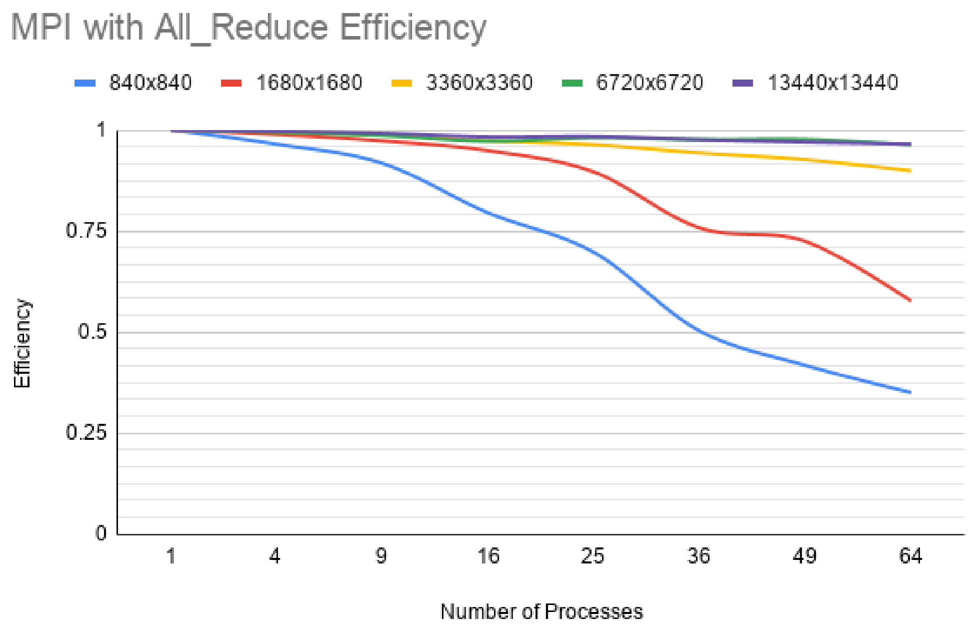
Speedup

Grid Size / Processes	1 = 1 * 1	4 = 2 * 2	9 = 3 * 3	16 = 4 * 4	25 = 5 * 5	36 = 6 * 6	49 = 7 * 7	64 = 8 * 8
840 x 840	1	3.574468085	8.615384615	14	17.68421053	19.76470588	22.4	84
1680 x 1680	1	3.962059621	8.860606061	15.38947368	25.64912281	34	44.3030303	48.73333333
3360 x 3360	1	3.985034014	8.984662577	15.62133333	24.8220339	35.28915663	46.49206349	56.32692308
6720 x 6720	1	3.978083588	8.899657925	15.7358871	24.68894981	35.53110774	48.47826087	62.94354839
13440 x 13440	1	3.986939126	8.930968076	15.85412996	24.72360057	35.59994095	48.3468324	63.06328452



Efficiency

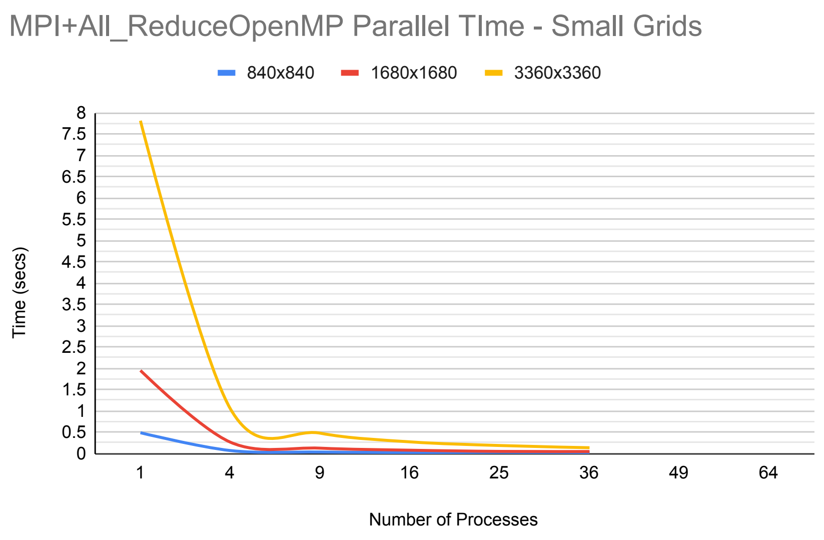
Grid Size / Processes	1	4	9	16	25	36	49	64
840 x 840	1	0.9651639344	0.918128655	0.7956081081	0.6977777778	0.5032051282	0.4179236912	0.3504464286
1680 x 1680	1	0.9889705882	0.973126615	0.9490927419	0.8966666667	0.7580515298	0.7250673854	0.5768995098
3360 x 3360	1	0.9964323467	0.9880765199	0.973786157	0.9637060703	0.9435685686	0.9270961397	0.8994513359
6720 x 6720	1	0.9942627275	0.986295097	0.9724587203	0.9806178862	0.9762367262	0.9768059605	0.9635097137
13440 x 13440	1	0.9967519587	0.9917357411	0.9831338855	0.9841126071	0.9761315074	0.9701559773	0.9653800589

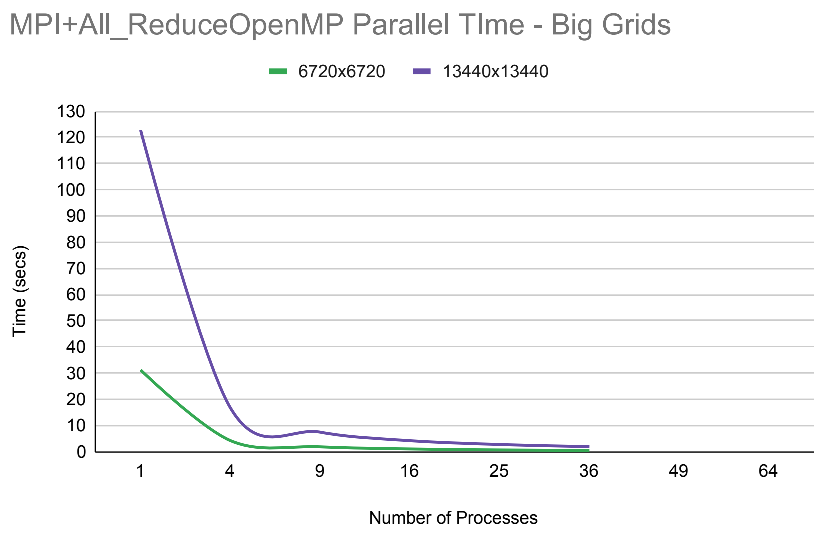


4.2 MPI+AllReduce+OpenMP

Parallel Time

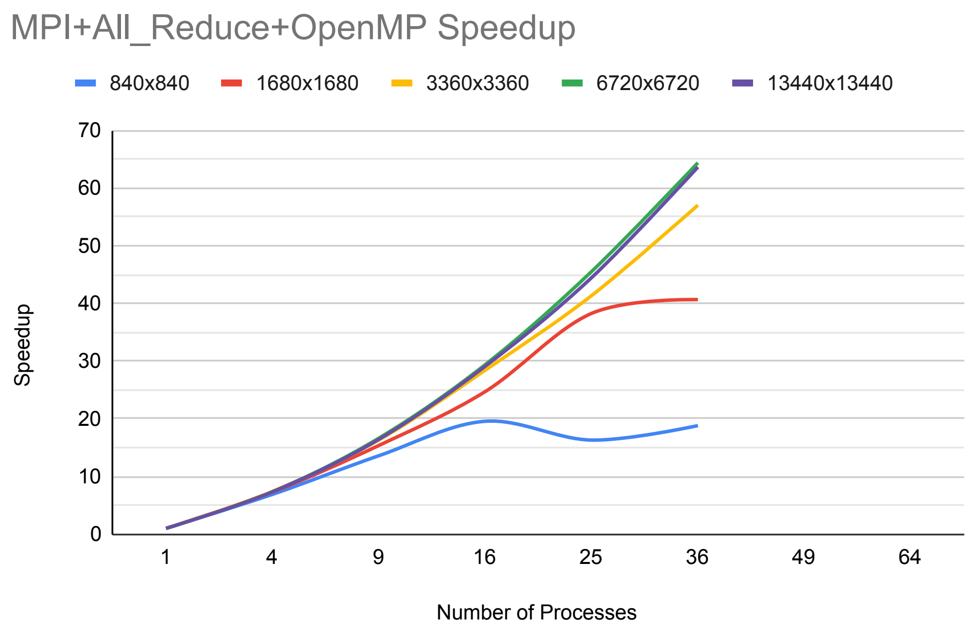
Grid Size / Processes	1 = 1 * 1	4 = 2 * 2	9 = 3 * 3	16 = 4 * 4	25 = 5 * 5	36 = 6 * 6
840 x 840	0.489	0.071	0.036	0.025	0.03	0.026
1680 x 1680	1.951	0.268	0.127	0.079	0.051	0.048
3360 x 3360	7.812	1.055	0.479	0.275	0.189	0.137
6720 x 6720	31.218	4.26	1.878	1.063	0.686	0.485
13440 x 13440	122.794	16.764	7.477	4.221	2.763	1.93





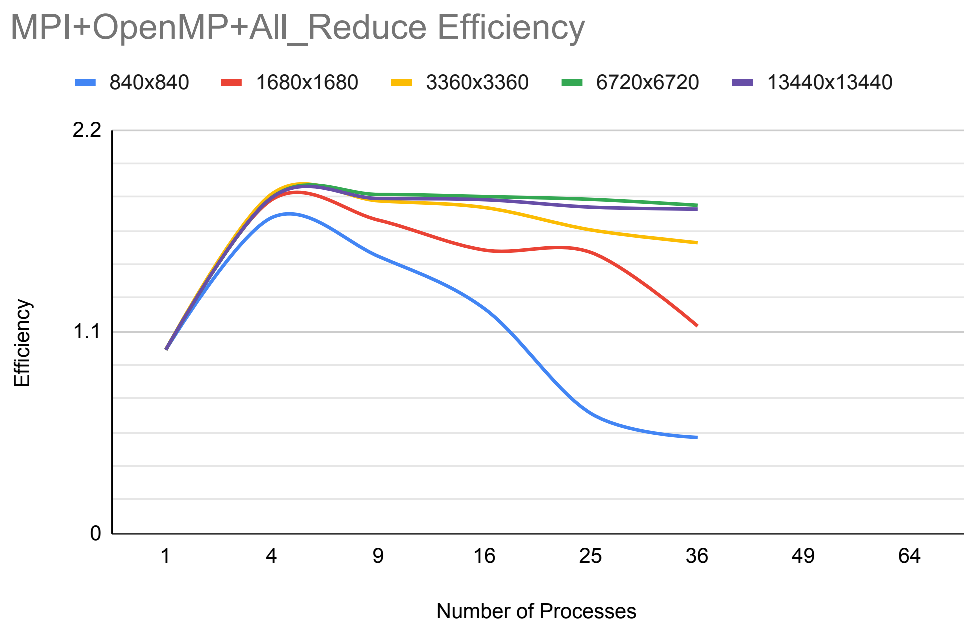
Speedup

Grid Size / Processes	1 = 1 * 1	4 = 2 * 2	9 = 3 * 3	16 = 4 * 4	25 = 5 * 5	36 = 6 * 6
840 x 840	1	6.887323944	13.58333333	19.56	16.3	18.80769231
1680 x 1680	1	7.279850746	15.36220472	24.69620253	38.25490196	40.64583333
3360 x 3360	1	7.404739336	16.30897704	28.40727273	41.33333333	57.02189781
6720 x 6720	1	7.328169014	16.62300319	29.3678269	45.50728863	64.36701031
13440 x 13440	1	7.324862801	16.42289688	29.09121061	44.44227289	63.6238342



Efficiency

Grid Size / Processes	1	4	9	16	25	36
840 x 840	1	1.721830986	1.509259259	1.2225	0.652	0.5224358974
1680 x 1680	1	1.819962687	1.706911636	1.543512658	1.530196078	1.129050926
3360 x 3360	1	1.851184834	1.812108559	1.775454545	1.653333333	1.583941606
6720 x 6720	1	1.832042254	1.847000355	1.835489182	1.820291545	1.787972509
13440 x 13440	1	1.8312157	1.82476632	1.818200663	1.777690916	1.767328728



For the Hybrid version of MPI with All Reduce and OpenMP, our experiments showed that that the best combination of processes and threads in one node is 2 Processes - 2 Threads. From this point, we continued the study of scaling with the following configuration:

- For 9 Processes: 6 Nodes
- For 16 Processes: 8 Nodes
- For 25 Processes: 8 Nodes
- For 36 Processes: 10 Nodes

We observed that for more processes, our program failed to achieve any speedup, so we gave up further measurements.

5 CUDA

CUDA is a platform designed jointly at software and hardware levels to make use of the GPU computational power in general-purpose applications at three levels:

- Software: It allows to program the GPU with minimal but powerful SIMD extensions to enable heterogeneous programming and attain an efficient and scalable execution.
- Firmware: It offers a driver oriented to GPGPU programming, which is compatible with that used for rendering. Straightforward APIs to manage devices, memory, etc.
- Hardware: It exposes GPU parallelism for general-purpose computing via a number of multiprocessors endowed with cores and a memory hierarchy.

In this project, we will use the GPUs provided in ARGO in order to measure our timings for our CUDA implementation. Before the metrics though, we are going to briefly explain our code for CUDA, which can be found in cuda/gol.cu.

5.1 Data Distribution

In CUDA, instead of processes, we are dealing with **blocks** and **threads**. Those blocks and threads, all share the same grid, thus there is no need for data exchange. In every call of the device functions, we compute the index of each thread, by the function:

$$\begin{aligned} index = {} & (\text{blockIdx.x} * \text{BLOCK\_SIZE} + \text{threadIdx.x}) \times \text{size} \\ & + (\text{blockIdx.y} * \text{BLOCK\_SIZE} + \text{threadIdx.y}) \end{aligned}$$

5.2 Code Design

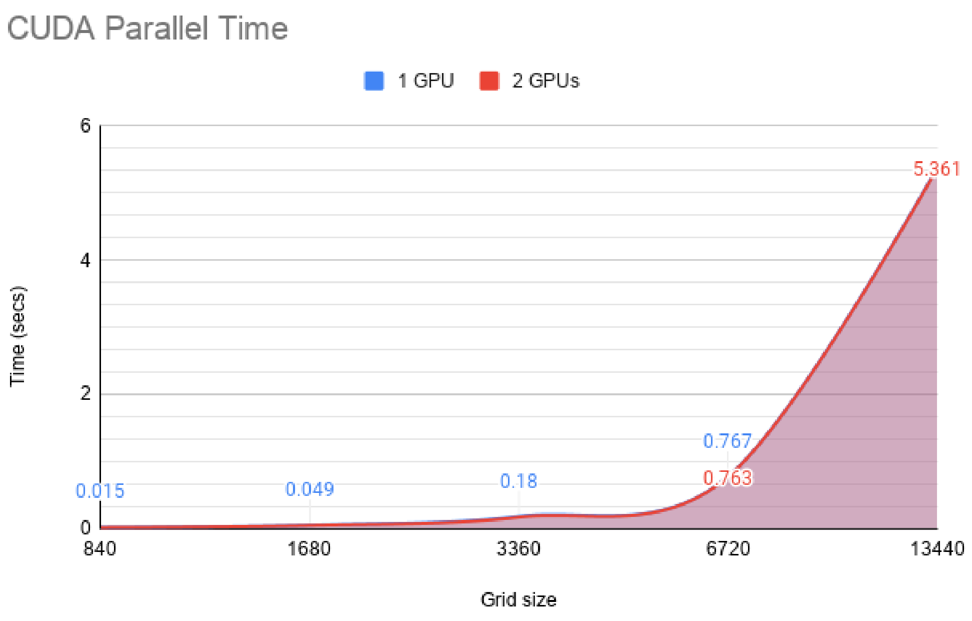
As mentioned above, each thread has its own start row and column in the grid. In order to compute the neighbours, it just gets out of bounds of its own indexing, if of course that is allowed by the big grid. We accomplished that by multiple calls of the handy function count_alive_neighbours.

In our main function, we allocate memory in the CPU and the GPU, according to the grid size, and we define our block size and the number of blocks in order to achieve an equal square splitting of our grid, just like in MPI. We run as many loops as the user has specified. In each loop, we call the device function to evolve our grid, and then exchange our grid pointers. Finally, we measure the time elapsed, and we copy the final data in the main memory, for output reasons.

5.3 Measurements

Parallel Time

Grid Size / GPUs	1	2
840 x 840	0.015	0.014
1680 x 1680	0.049	0.045
3360 x 3360	0.18	0.168
6720 x 6720	0.767	0.763
13440 x 13440	5.373	5.361



6 Conclusions

During this project, it was made clear to us that parallel programming is very useful in the era of big data, and supercomputers. Using many processes, or many threads, a 2GB file (our biggest grid), can be processed trillions of times within 1 second, if we use parallel computing. During our implementations we used many different kinds of techniques. It is unclear which is best: it always depends on the program and the data given. Our conclusions regarding the different types of parallel computing are the following:

- The naive MPI version of this problem achieves remarkable results in relation to the serial one. This comes with the trade of its quite hard to implement by a intermediate programmer, because of the topologies and the inter-process communication.
- MPI without All Reduce and MPI with All Reduce does not make a significant difference in such big data size. This is totally reasonable as it takes thousands (in some cases up to million) iterations to converge. For our experimental evaluation, for 840 x 840 grid we tried MPI + All Reduce up to 10,000 iterations and we observed that it could not converge. In contrast, in relative small data the use of All Reduce can be extremely helpful by interrupting unnecessary repetitions from the moment it converges.
- In our case, as we are dealing with big data, All Reduce does not offer any improvement, on the contrary it causes a slight delay due to the calculations performed every 10 repetitions to check the condition of convergence.
- The addition of OpenMP is quite beneficial in the Game Of Life, and in general in problems that deal with serial loops in each process. It helps parallelize a greater portion of our program, which helps as live up to Amdhal's Law. However, in order to achieve a satisfactory scaling, one needs to experiment thoroughly at a node for the right combination of processes and threads.
- With the escalating rise of computing power of GPUs, CUDA seems to be the state of the art on the parallel programming. Indeed our results justifies this statement, as it can be up to x24 faster than MPI versions.
- For CUDA results, we observe that there was slight to no difference between the usage of 1 or 2 GPUs.

7 Implemented Bonus

We chose to work on several bonus points that were given, in order to further familiarize with the Game of Life. We will now present those extra tasks that we performed in our project.

7.1 Parallel I/O

In order to parallelize a bigger portion of the problem, we chose to implement parallel input and output. Both functions (parallel_read and parallel_write) can be found in the file io_utils, under mpi_openmp/src. Given the rank of the current process, the programs reads or writes accordingly, where it should.

7.1.1 Reading

For our comfort, we read files where the data is serially presented, and not as a 2d matrix. We describe the starting row and column based to the rank of the process, and then we allocate a line buffer in order to easily transfer the data, and then a 2d char array in order to store it. We use the MPI_File_Read function, that copies the data into continuous pages of the memory, thus we do not allocate the 2d array in the traditional way. We make sure that all the data are together in the memory, so that the MPI_File_Read function works as it should. Finally, we fill the 2D array with the data we just read.

7.1.2 Writing

Using the same methods as in the reading section, we output a given generation in an output file. Our goal is to place all the generations on the same file, in order to visualize it later.

7.2 Visualization

In order to gain some intuition on how the grid (aka the generations) changes over time, we decided to provide a visualization utility. To run the script to generate the GIF of Game of Life:

1. From the project folder, change directory to misc
2. Run the command: python vizualize.py -o outputfile -d outputdir -s size

where *outputfile* stands for the generated file from our MPI/CUDA code, *outputdir* for the directory in which will export our GIF and *size* for the size of grid. Inside the folder miscgrids, you can find the GIFs we have generated for grids size of 50 and 100 respectively, ran for 90 loops.