ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ϯ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν
Πανεπιστήμιον Αθηνών
ΙΔΡΥΘΕΝ ΤΟ 1837

Assignment 2019-2020: Conway's Game of Life

National and Kapodistrian University of Athens, DiT. 18/9/2020

Jakub Maciejewski | sdi1700080
Dimitrios Foteinos | sdi1700181

Sections and Chapters

Contents

1 Introduction

This particular programming assignment has been done in the context of the course: Parallel Systems, during the 5th Semester. This course taught us about the aspects of parallel programming (MPI, OpenMP, Cuda , mpi.h). For compiling and running our programs we used a parallel computer (ARGO: 80 cores and 2 GPUs) provided by our Professor, **Mr. Yiannis Kotronis**. During this assignment we had to implement the famous Game of Life by John Conway. For further information, check this site: Game Of Life

Each implementation has its own folder. Within it, there is a ready script used to push our program into the cluster queue for it to be executed in the near future. The names of the executables are obvious after the usage of the given makefiles or through the script.

User can change the parameters of the script including the number of nodes and processes in the cluster, as well as the 3 arguments following each one of the 3 implementations. You can change the number of -rw (rows), -cl (columns), bearing in mind that they have to be a multiple of 840 (least common multiplier of the number of processes requested by our instructor). Iterations, represented by -it, can be changed as well, if you want to test the program on smaller or even bigger challenges.

2 Data Sharing Design and Assumptions

- **2.1: Data Management**

Initially, we thought about separating the plain into whole trapezoids following the top-down approach. Finally, analyzing the functionality of the MPI library functions, we came to the realization that the separation of the plain into blocks would be the most efficient approach, as it allows the development of the cartesian communicator in which each block communicates with 8 neighbours, corresponding to other processes' blocks.

As an example, lets consider a small (for Argo's capabilities) plain with 10 x 10 dimensionality. Following the cartesian topology semantics, we separate the plain into 4 blocks. As a result, we have blocks corresponding to squares covering the following opposite coordinates [0,0]/[5,5] (top left), [0,5]/[5,10] (top right), [5,0]/[10,5] (bottom left) and [5,5]/[10,10] (bottom right).

In order to enhance the speed of the communication, we added extra border pixels to each block, which are being used as a buffer for bordering pixels, that are being managed by neighbouring processes. As a result, each block has two extra columns and rows. This addition, allows for halo points, that are used to communicate efficiently with diagonal neighbours in terms of cartesian topology.

Being a little pedantic, we decided that we ought to fill all of the coordinates of the MPI 2D cartesian communicator, that we will be using. As a result, the number of processes is always a power of 2 of an integer, bounded by 64, as Argo can supply us with 80 processes at best. As a result, in order for our plain to be equally separated through processes, we start our tests with a plain of the impressive size of 840 x 840. Following, we just multiply each side by 2, quadrupling the final size of the plain. We keep up with this incremental manner up to 6720 x 6720 plain, where our serial program starts to get a little too slow for its parallel counter part.

- **2.2: Inter-process Communication**

As stated before, each process is represented as coordinate in a two dimensional space, taking care of a specific block of the initial plain. As a result, each process, has 8 neighbours, 4 being normal and the other 4 being diagonal. The core idea behind the inter-process communication is that the information about the border pixels (that are needed in the computations for each block) is being stored by neighbouring process, which have to send that information to their neighbours.

In order to make our program faster, we implemented our own MPI data types, specifically the row and column datatype, comprised of multiple characters, representing a bordering column/row of pixels. In this way, when we communicate by sending or receiving whole columns/rows of data, limiting the amount of requests, ergo the total execution time, that otherwise would be wasted for repetitive communication establishment.

In order to make sure, which part of the border we are receiving or sending to our neighbours, we used a tag system, with each number corresponding to a specific part of the border. Each time, we are iterating through our neighbours, we make use of an array containing the topological names of our neighbours, translating their position into a specific tag (integer), that then is being applied to the sent message. In this way, the recipient will know which part of the plain he has received, so it can be temporarily stored in the augmented border and used in the computation of the updated partial block.

To make it more understandable, we will continue with an example. Lets consider the 8 neighbours of a block in a circular manner starting from the upper left one being represented with a 0 and ending with the left neighbour that is represented with a 7. As a result, when we want to receive the contents of the upper left pixel, we are topologically on the right bottom of the recipient of our request. This means, our request must be accompanied with the tag of 4. In this way, our upper left neighbour will be able to identify that it has to send some helpful information about the corner pixel with tag 0.

- **2.2: Topology**

Following our instructor's tips, we used the cartesian topology in order to represent our processes in a more understandable manner. As stated before, each process is abstractly represented as a coordinate on a two dimensional plain, with each axis being periodic in MPI terms. Which means that processes situated at the border of the plain, communicate with the opposite ones. For example, lets say that we have 9 processes. In this case, the neighbours of the [3,3] coordinate are not only [3,2], [2,2] and [2,3], but also [2,0], [3,0], [0,0], [0,3] and [0,2]. This approach, allows for easier data manipulation and cleaner code syntax when it comes to communication using tags. There is no need to check if our neighbours are existent or not, as the plain is full and periodic.

Living in a world of static, euclidean geometry, our neighbouring coordinates remain the same. As a result, we use MPI function to get the cartesian identificators of our neighbours, store them in an array, that we simply traverse in each iteration, receiving and sending the bordering pixels to each of our neighbours. This simple model, allows for clear and understandable data exchange.

3 MPI Design and Optimization

- **3.1: Communication Design**

Having described the topology of our program's processes, we will continue with some communication patterns and intricacies. In order to maximize the speedup caused by the usage of multiple processes, we are using the non-blocking form of communication functions. In this way, the execution of a command can be instant, there is no need for acceptance on the other side. In order to make sure, that no request will remain unanswered, we are using request arrays and waiting for each one of the 8 possible requests that a process can receive from its neighbours.

- **3.2: Optimizations**

As stated before, our local blocks are augmented, which means that they have extra space, where they save information about the pixels, that are affected by both their area and the one

belonging to their neighbour. The other remaining block, without this border, can be reevaluated in each iteration without a need for communication, as all data of its pixels is stored locally. In this way, we are limiting the need for communication to the minimum.

In order to take advantage of the fact that the speed at which each process will send and receive needed data is undefined and fluctuate, we are not waiting for a specific request. We simply wait for any request, 8 times, in order to make sure that all of the requests of the neighbouring processes have been received. We don't care about the order of reception, as the inner block is already updated and can be combined with the received data, to update the corresponding border part of the block.

During the simulation, the plain is being updated. Trying to implement that, we came up with the idea of two blocks, the previous and current one. For each one, we have an array of send and receive requests for each of the neighbours. In this way, we can gather and send all the necessary border info of each specific block with no need of copying data to one, single array, which would result in a enormous fall in efficiency.

Even though, we were suggested to use the iSend and iReceive functions for non-blocking communication, as stated before, we followed the request array approach. We are simply initializing sending and receiving communication for each process. As a result, we are not losing precious resources, that would be used to reestablish the communication for each iteration of the plain updates. After that, we are simply executing the receive and send requests stored in the two specified arrays for both previous and current block.

Efficiency lies at the heart of our project, so we took into consideration other useful remarks of our professor and implemented them. To be specific, as we stated before, special data types for columns and rows were used, limiting the amount of requests, allowing for transportation of bigger chunks of data. Our arrays are initialized as contiguous pieces of data, so there is no technical issue with the transportation of whole columns or rows. Also, the neighbour IDs are being stores in an array, before the execution of the main program, so they don't have to be requested each time. Using cartesian communicator, allows us to automatically put communicating processes in the same cluster node, resulting in faster data exchange. Finally, as we stated before, we are using two arrays to represent the previous and the current state of the local block. During each iteration, we are updating the contents of the current block based on the ones of the previous block. After that, we simply swap the pointers, so we don't have to copy the block during each iteration, saving us plenty of time for hard disc and RAM communication. All these design patterns, lead to a pretty impressive result in terms of speed and satisfy all the requests implied by the project description.

- **3.3: Equality check**

As stated by the instructor, we had to supply an allReduce form of our program, in which we made sure that after each 10 iterations, our program would check whether previous and current

blocks are the same. This check was requested, so we could see a fall in efficiency caused by the array comparisons and the reduction command that had to make sure the two states of each local block were taken into consideration, while checking for changes in the whole plain. Specifically, there is a small piece of commented code with an if statement in the main GOL-Simulation function. If the user, wants to make use of the program in the allReduce form, they should uncomment the given part of code.

4 MPI Measurements

4.1: Simple MPI:

In order to compute the following tables, we had to use these particular formulas:

For Speedup:

$$S_{(n,p)} = \frac{T_s(n,1)}{T(n,p)}$$

For Efficiency:

$$E_{(n,p)} = \frac{S_{(n,p)}}{p}$$

- **4.1.1: Without MPI-Allreduce**

Time	1	4	9	16	25	36	49	64
840	1.169926	0.882704	0.407218	0.226557	0.150253	0.108937	0.082382	0.077189
1680	4.649761	3.564575	1.596063	0.892072	0.572847	0.404285	0.300311	0.231870
3360	18.470627	14.534478	6.346416	3.558249	2.278190	1.587565	1.178846	0.898188
6720	74.116574	58.106551	25.321231	14.273432	9.120796	6.331043	4.664689	3.583527

Speedup	1	4	9	16	25	36	49	64
840	1	1.325388	2.872972	5.163936	7.786373	10.739473	14.201233	15.156641
1680	1	1.304436	2.913269	5.212315	8.116933	11.501119	15.483152	20.053310
3360	1	1.270814	2.910402	5.190931	8.107588	11.634564	15.668396	20.564321
6720	1	1.275528	2.967025	5.192624	8.126108	11.706850	15.888856	20.682577

Efficiency	1	4	9	16	25	36	49	64
840	1	0.331347	0.319219	0.322746	0.311454	0.298318	0.289821	0.236822
1680	1	0.326109	0.323696	0.325769	0.324676	0.319475	0.315982	0.313332
3360	1	0.317703	0.323378	0.324433	0.324303	0.323182	0.319863	0.321317
6720	1	0.318882	0.329696	0.324539	0.325044	0.325187	0.324262	0.323165

- **4.1.2: With MPI-Allreduce**

Time	1	4	9	16	25	36	49	64
840	1.184059	1.071006	0.483726	0.267621	0.179175	0.128227	0.100678	0.079306
1680	4.667475	4.195742	1.888972	1.055612	0.685721	0.478661	0.359099	0.274957
3360	18.594917	17.182013	7.504008	4.207627	2.700544	1.881983	1.403818	1.065218
6720	74.178113	68.848490	29.990947	16.865818	10.805125	7.514474	5.509548	4.222589

Speedup	1	4	9	16	25	36	49	64
840	1	1.105557	2.447788	4.424387	6.608394	9.234084	11.760851	14.930357
1680	1	1.112431	2.470907	4.421581	6.806667	9.751107	10.166598	16.975290
3360	1	1.082231	2.477998	4.419335	6.885618	9.880491	13.245959	17.456442
6720	1	1.077410	2.473350	4.398133	6.865086	9.871364	13.463556	17.566974

Efficiency	1	4	9	16	25	36	49	64
840	1	0.276389	0.271976	0.276524	0.264335	0.256502	0.240017	0.233286
1680	1	0.281082	0.274545	0.276348	0.272266	0.270864	0.207481	0.265238
3360	1	0.270557	0.275333	0.276208	0.275424	0.274458	0.270325	0.272756
6720	1	0.269352	0.274816	0.274883	0.274603	0.274204	0.274766	0.274483

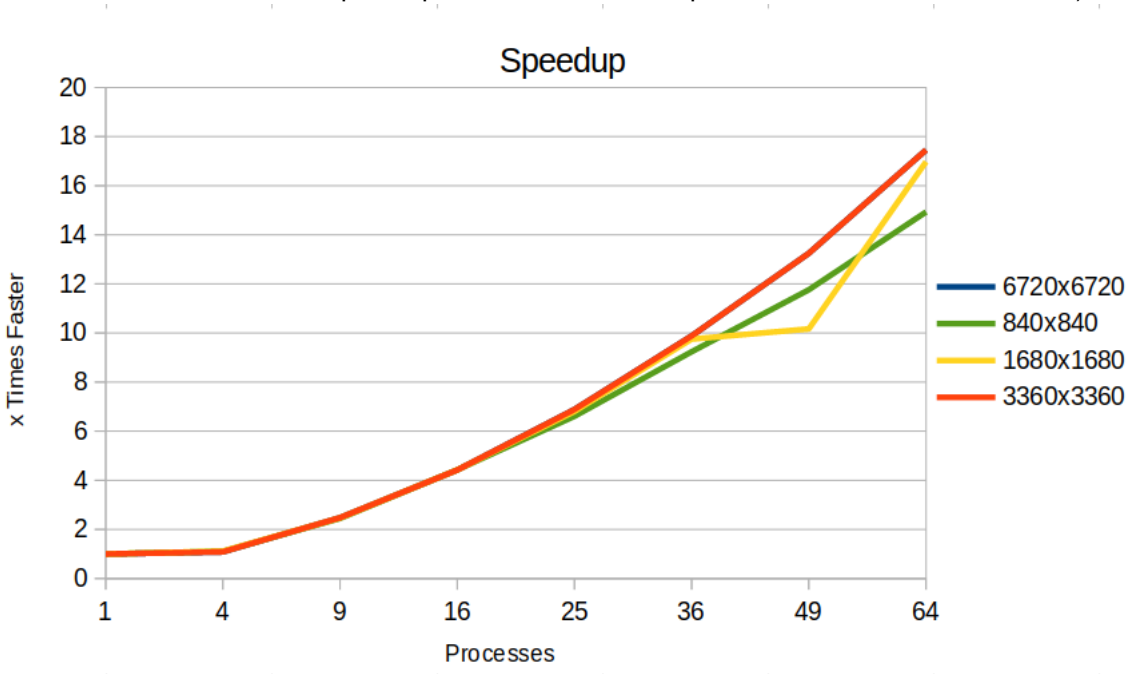
- **4.2.1: Time**

As one can imply from the given tables, the increase in the amount of processes used to calculate the subsequent plains, results in the decrease of needed time to conduct all the necessary calculations. The saved time becomes even more apparent when we analyze the row corresponding to the biggest plains. Execution time falling from 74 seconds in the serial program to 3.5 seconds in the respective parallel one with 64 processes is a pretty impressive development. But an expected one, as the enormous plain is being broken into smaller pieces, calculated simultaneously by equally capable processes, resulting in smaller execution time in total. It is also important to note that the serial program was executed on the same device with the parallel one, so our speed up and efficiency metric can be more precise. Also, each execution consisted of 50 iterations, independently of the amount of processes or the size of the plain.

- **4.2.2: Speedup**

Speedup results are equally, if not more impressive. Speedup as a mathematical term refers to the amount of times, the parallel program is faster in comparison to the serial one. As we can see, when the size of the problem is smaller, pixel calculations are pretty fast, resulting in communication being the main aspect, that our processes spend time on. As the size of the plain

increases and the number of processes we are using is getting bigger, the speedup increases in a big way. Enormous number of pixels is being distributed among a big number of processes, resulting in smaller sub problems to solve. Communication, limited only to the exchange of the border pixels, is no more the main concern of the process and while the sub plain to calculate decreases in size with the increase of capable workers, the speedup is only getting bigger. It is important to point out that 64 processes executing a parallel form of the program are 17.5 times faster in total than our serial program. We can also visualise these statistics with the follow chart (The chart below refers to the Speedup table which was implemented with MPI-Allreduce) :



- **4.2.3: Efficiency**

Efficiency as a mathematical concept is used to quantify the contribution of single process to the speed up that is being observed in the parallel program case. Even though, the speedup seems to be increasing in a big way with each increase of the number of processes, the corresponding efficiency metric is slowly decreasing. It means that the introduction of many new processes, even though it separates the initial problem into very small sub problems, at the same time it results in a complicated network of processes. In such network, the amount of communication requests is much bigger, resulting in big time penalties caused by communication delays. As a result, the more processes we are embracing, the smaller the individual process contribution becomes.

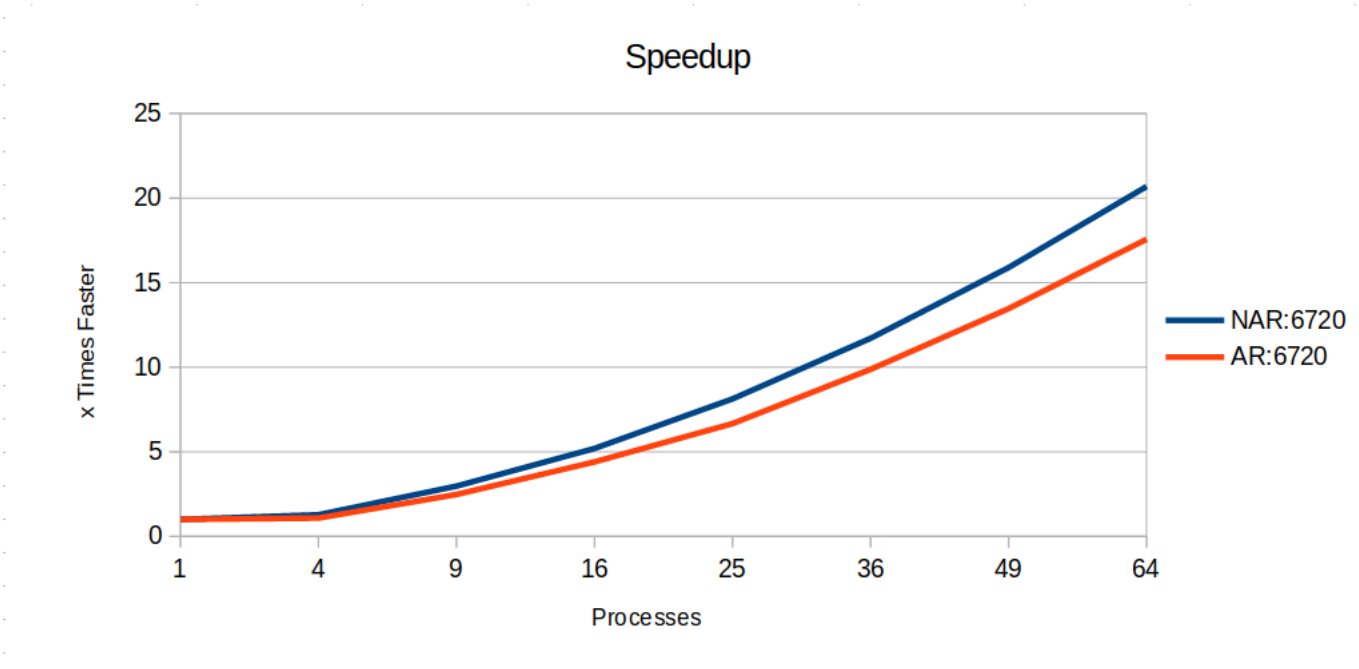
In our case, this decrease is minimal, as the communication is limited only to the border pixels of each local block and most time is still being spent on inner block pixels' calculation, resulting in small latency. The only discernible efficiency decrease is observed in the 840 x 840 plain case, where the problem to solve is small, but the communication penalty still remains. In all the other size cases, the efficiency falls in a minimal manner, almost remaining stable, implying that our program is characterized by respectable scalability.

- **4.2.4: AllReduce Case**

Periodic checks for sequential blocks' equality and the need for global change calculation that appear in the allReduce program version, imply a fall in efficiency, resulting in extra calculations when it comes to array comparison and communication penalties caused by the update of the global change variable, with which, we can decide whether the whole plain remained intact or not. These assumptions can be proved correct, when we analyze the mathematical data in the form of efficiency and speedup.

The allReduce version of the program shows smaller speedups, regardless of size of the problem, in comparison with the normal, parallel one. This is caused by the severe penalties imposed by the extra checks and global updates. Speedup is used in the calculation of efficiency and they are linearly dependent, so the efficiency is also smaller. We expected bigger drawbacks, but the function that we used to compare the two arrays, shows fast response and efficiency. It may be caused by the fact, that pixels show similar distribution in all the partial blocks, so when we make sure that our plain will remain alive during the 50 iterations, the chance that different pixels are found early on in the search is high. Still, the implications of the allReduce are discernible.

Regardless, both programs show good scalability, as we can deduct from the following graph, where we compare the speedups of both programs for the most complicated case of 6720 x 6720 pixels plain with the non allReduce version being the clear winner:
(**NAR** stands for Non All Reduce and **AR** stands for All Reduce)



4.2: MPI+OpenMp:

- **4.2.1: MPI+OpenMp Conventions:**

In this section, P stands for number of processes employed by each node of the cluster and T represents the number of threads that are being forked for each process. Following the previous convention of number of processes being a square of an integer and having to run our openMP implementation on 2, 4, 8 and 10 nodes, we run two different measurements. One being for 2 processes with 4 threads on each node and the other one being 4 processes with 2 threads. In this way, we were able to run our program on 1, 2, 4 and 8 nodes of the cluster. In this way, we have openMP measurements for 4 and 16 processes, forked into 2 and 4 threads respectively. We chose these values to represent a wide range of suggested processes/threads ratios.

Speedup is being calculated as the fraction of the serial implementation's time and the time spent on the openMP implementation. When it comes to efficiency measurements, speedup is being divided by the total number of threads employed. All measurements were based on the allReduce model with 50 iterations. Checks for subsequent plain equality are emerging every 10 iterations.

- **4.2.2: OpenMp Design**

The differences between the openMP and MPI implementation is that the first one is multithreaded. The program is informed about the fact that multithreaded calls will follow by the MPI-init-thread call. We followed the funelled thread implementation, which translates to only the master thread being able to use MPI calls, making the program a little slower than the MULTI-THREAD approach, which is more unstable and harder to implement.

The cpu intensive inner plain update calculation is being implemented with the usage of omp for, allowing all the threads to contribute, as every iteration takes constant time and there is no need for extra scheduling. In order to make sure that all threads are taking part in the same iteration of the game, we are using the omp barrier command, just before the plain update calls. After many trials and errors, we came to the realization that employing any type of scheduling in our program, collapse being a good example, only shows worse results. That is why all of our measurements were taken with the collapse call being commented. Many parts of the GOL simulation functions relate to global updates that can take place only one time on each iteration, that is why they are preceded by an omp single call. One example would be the swapping of the old and current plains at the end of each iteration.

- **4.2.3: MPI+OpenMp: 2P/4T**

Time	2 nodes	8 nodes
840	0.284081	0.100830
1680	0.993595	0.272998
3360	3.927184	1.035170
6720	14.833272	3.865137

Speedup	2 nodes	8 nodes
840	4.118283	11.602955
1680	4.679834	17.032216
3360	4.703275	17.843085
6720	4.996643	19.175665

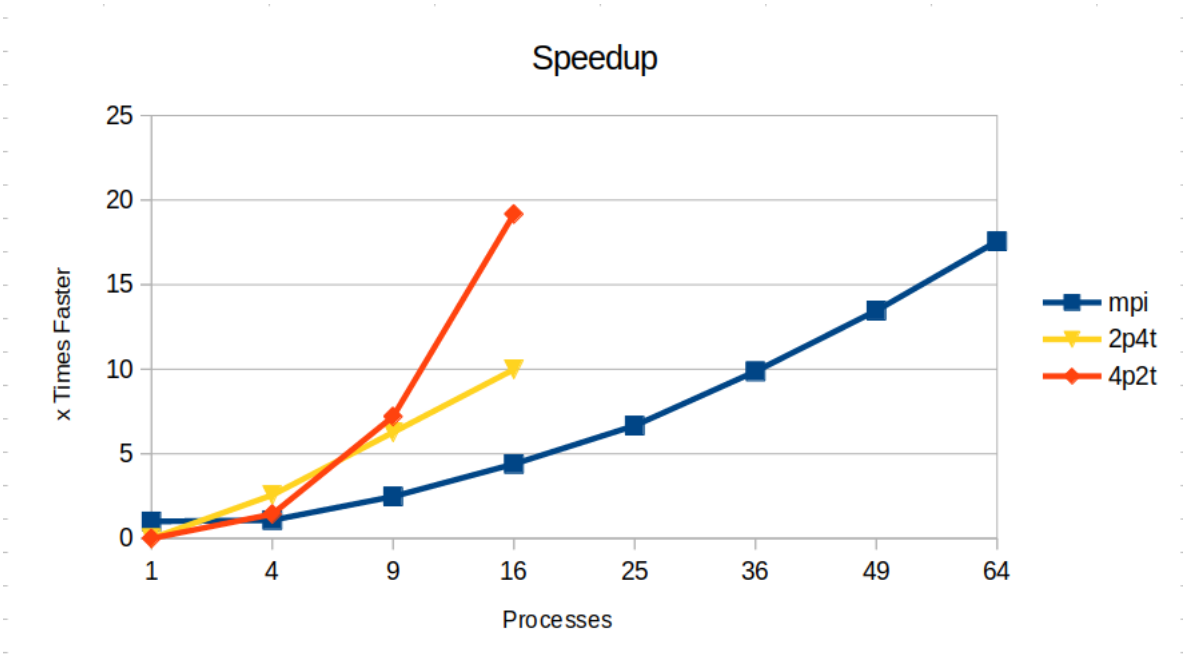
Efficiency	2 nodes	8 nodes
840	0.257392	0.181296
1680	0.292489	0.266128
3360	0.293954	0.278798
6720	0.312290	0.299619

- **4.2.3: MPI+OpenMp: 4P/2T**

Time	1 node	4 nodes
840	0.485396	0.137883
1680	1.902771	0.486383
3360	7.465133	1.880417
6720	28.964985	7.421881

Speedup	1 node	4 nodes
840	2.410250	8.484918
1680	2.443678	9.559875
3360	2.474252	9.823145
6720	2.558883	9.986225

Efficiency	1 node	4 nodes
840	0.301281	0.265153
1680	0.305459	0.298746
3360	0.309281	0.306973
6720	0.319860	0.312069



- **4.2.4: Scalability and Comparison**

Analyzing the speedup graph, one could infer that the multithreaded implementation, especially the one with 4 processes and 2 threads for each node, shows impressive speedups. The biggest speedup approaches 20 times the speed of the serial program, while the best for the corresponding clear MPI implementation with allReduce is only 17 and is accomplished with 64 processes, instead of only 16.

Nevertheless, speedup is not the only metric that we should be taking into consideration. The final result of the openMP gets less impressive, when we compare the two implementations on efficiency basis. In the 2P/4T case for the smallest input and the number of nodes equal to 8, efficiency seems to be terrible in comparison with the MPI result, where 16 processes show an efficiency bigger by a margin of 0.6. Results get more promising when we increase the input plain,where in some cases, especially the most complicated one openMP seems to win by a margin of 0.4, peaking at 0.312290.

This analysis, broughts about the main concern surrounding the openMP implementation. Differences in efficiency on small data and process counter increase are discernible, showing bad scalability when it comes to minimal inputs. Scalability becomes impressive when we increase the size of the plain. In the 4P/2T case for the 6720x6720 array , the quadrupled amount of processes has minimal impact on the efficiency, which stays still at 0.31. OpenMP behaves better than simple MPI for strenuous tasks with way way smaller total execution times, resulting in more impressive speedups, simultaneously showing similar (minimally smaller), a little decreasing scalability when it comes to big input arrays.

When it comes to the two different process/thread ratios analyzed in the openMP implementation, in terms of speedup, the clear winner is the 2P/4P. OpenMP seems to speed up the execution time when we choose to increase the amount of threads per process, instead of employing a bigger amount of processes with less forks. This approach though, is pretty unstable when it comes to small data, as stated previously for the base case of 840x840 plain. When we increase the size of the plain, the thread focused approach, shows the power of the openMP paradigm.

5 CUDA

- **5.1: CUDA Conventions**

In this section we will be using the letter T to represent the number of threads comprising a block in our CUDA implementation. Block stands for a group of threads, having access to the same, shared memory, which is a part of the L1 cache, allowing for fast data exchange and retrieval between the same block's threads.

In our implementation each pixel of the plain is managed by a different thread. Our measurements were taken, considering a different amount of threads per block. As a result, the number of blocks is equal to the fraction of total pixels and the number of threads per block. Specifically, we have taken into consideration the cases of 256, 512 and 1024 threads, in order to point out specific aspects of CUDA capabilities and the effect that waste of resources can have on the final performance of a really fast program.

The program was run on the 11th node of Argo's cluster, equiped with two powerful Dual Nvidia Quatro P4000, 8GB graphic cards. Trying not be wasteful, we chose to run our program, using only one of them. Efficiency table is not included in our presentation, as the insane number of threads used, results in minimal efficiency distribution among the plentiful, but not that powerful cores of the graphic card. There is not reason to analyze the efficiency metric, when analyzing programs run on graphic cards, that follow a totally different design approach, when compared to CPUs.

- **5.2: CUDA Measurements**

Time	**256T**	**512T**	**1024T**
840	0.003795	0.003827	0.003988
1680	0.014630	0.015193	0.015800
3360	0.058067	0.060403	0.063182
6720	0.226464	0.239594	0.248516

Speedup	**256T**	**512T**	**1024T**
840	308.280896	305.703162	293.361585
1680	317.823718	303.046271	294.288671
3360	318.091636	305.789895	292.340018
6720	327.277510	309.342362	298.236629

- **5.3: CUDA Design**

The CUDA implementation is a classic example of such programming practice. The CPU program allocates and inhabits the memory of the two subsequent plains. Then, the memory of these plains is being copied into seperate memory spaces in the GPU memory. The distribution of cells and updates are taken charge by the CPU, using a kernel call with the name GOL-Simulation.

We decided that each cell will be taken care by a different thread. Using its identificator within its block, we can imply which cell it is taking care of. The CUDA code is using a single dimensional vector representing the values in the initial 2 dimensional arrays. Each thread is using special mathematical formulas for coordinate projection from 2 dimensional to single dimensional space, in order to spot its neighbours and then using their state, in order to update its own.

No extra device functions are being called, as stack calls are really expensive, when it comes to efficient GPU programming. We decided to keep our code a little less clean, with all the updates being made in a single global function call, for the sake of efficiency.

- **5.4: CUDA-MPI Confrontation**

CUDA implementation is the clear winner, when it comes to speed-up metric. The CUDA program in the most demanding case was able to execute all the needed updates 327 faster than the serial implementation, when the number of threads was kept up to 256 per block. This is an amazing result, showing the true power of GPU programming when it comes to concurrent execution of the same operations.

The two plains are being stored in the global memory of the GPU, but there is no need of direct communication between threads, as shared memory is enough for the subsequent updates. This results in amazing scores, even in comparison with the MPI implementation, which wastes a lot of resources on interprocess communication.

For the ones that are into numbers, MPI implementation peaked at 20 times speedup, which means it was 15 slower than the best-case CUDA execution. Even though, we didn't expect such a difference, it was something not that improbable, as the process inactivity-waiting time was a serious inefficiency issue when it comes to MPI communication, that was analyzed in the case of allRecude and simple MPI implementations comparison.

6 Conclusions

Our excursion into the world of parallel programming was a big surprise. We didn't expect that such an approach would have these amazing effects in terms of program efficiency and execution speed. MPI library showed us that even though, processes waste a lot of time on intercommunication, when proper design patterns are being followed, the latency is indiscernible, when compared to the speedup that we have as a final result. Simple, interprocess communication and problem distribution into smaller subproblems, resulted in execution times that were 20 times smaller than the ones showed by the serial program that we initially implemented.

Even though impressive, MPI results were nothing in comparison with the latest technology developments in the field of GPU parallel programming. The design architecture of graphic cards that puts empasis on the big number of small processors, that are being able to execute similar operations and access tha same memory in matter of nanoseconds, is ideal for parallel programming.

Even though, the speed-up metric is impressive when it comes to CUDA implementation, we shouldn't forget the amount of threads employed to pull out such results. This means that thread level efficiency is really small, even though, the changes in efficiency are negligible, showing really good scalability.

Our implementation, which didn't even scratch the efficient, blocking techniques surrounding shared, block memory, was able to make execution time 300+ faster than the one presented by the serial program. This result is indeed impressive and makes us really positive that future developments in the field, will bring a brand new light on the still obscure world of parallel programming.

Still, if not for proper design patterns like self defined datatypes in MPI implementation or the usage of one dimensional arrays with smart mathematical formulas in the CUDA case, we wouldn't be able to achieve such amazing results. Parallel programming is a powerful tool, but demands deep understanding of how memory and interprocess communication work on a very low level.