HIGH PERFOMANCE COMPUTING PROJECT

Jacopo Dapueto

S4345255

Simone Campisi

S4341240

INTRODUCTION

Problem

The project aims to provide the parallelization of the Game of Life (GoL) program with **OpenMP**, **MPI** and **CUDA**.

The game board is two-dimensional grid of square *cells*, each of which is in one of two possible states, *live* or *dead*, (or *populated* and *unpopulated*, respectively). Every cell interacts with its eight *neighbors*, which are the cells that are horizontally, vertically, or diagonally adjacent. The game starts with a random configuration and then at each step-in time the following transitions occur:

1. Any live cell with fewer than two live neighbors die, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

In the sequential implementation of the program both the current generation and the next one are stored in two dynamic matrices, at the end of the computation of the next generation the two matrices are swapped so that in the next iteration of the time the current generation is updated, and it's done by simply swapping the references of the two structure, so that it is computed in constant time.

Assessment

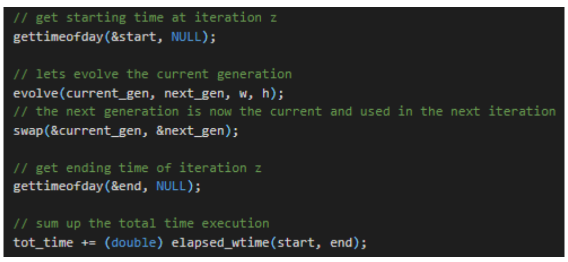
The programs have been executed on a cluster composed of 11 nodes, each one equipped with an **Intel(R) Xeon Phi (TM) CPU 7210 @ 1.30GHz** having **64 cores**, with the possibility to execute **4 threads per core**.

We provide the results in terms of the **speedup**, which is described by the formula:

$$S = \frac{T_{sequential}}{T_{parallel}}$$

where the higher the S the better is the result, and in terms of the **strong scaling** which is performed by keeping the size of total problem fixed and increasing the number of the processors *p*: so that the total amount of work remains constant and the amount of work for a single processor decreases as *p* increases.

We decide to consider the sum of the execution times of the function that computes the evolution of one generation for each time step, as can be seen from the image on the right.



```
// get starting time at iteration z
gettimeofday(&start, NULL);

// lets evolve the current generation
evolve(current_gen, next_gen, w, h);
// the next generation is now the current and used in the next iteration
swap(&current_gen, &next_gen);

// get ending time of iteration z
gettimeofday(&end, NULL);

// sum up the total time execution
tot_time += (double) elapsed_wtime(start, end);
```

The serial program was compiled with -O0 and -*O2* vectorization flags of the **ICC** compiler, in the first case the compiler does not optimize the program and in the second case the program is vectorized. The following table shows the execution times and the speedups.

Width	Height	Iterations	Serial O0	Serial O2	Speedup
100	100	10	52.35 ms	18.55 ms	2.82
500	500	10	1342.03 ms	469.74 ms	2.85
1000	1000	10	5762.69 ms	1871.70 ms	3.07
5000	5000	10	149873.14 ms	45914.28 ms	3.26
10000	10000	10	597232.41 ms	186158.10 ms	3.20
15000	15000	10	1423490.48 ms	416311.40 ms	3.41

From the table can be seen than the speedup doesn't change too much increasing the problem size, so the optimization provided by the compiler doesn't depends on the problem size.

Our baseline times for the computation of the speedups in the next experiments are the ones obtained with the *-O2* flag.

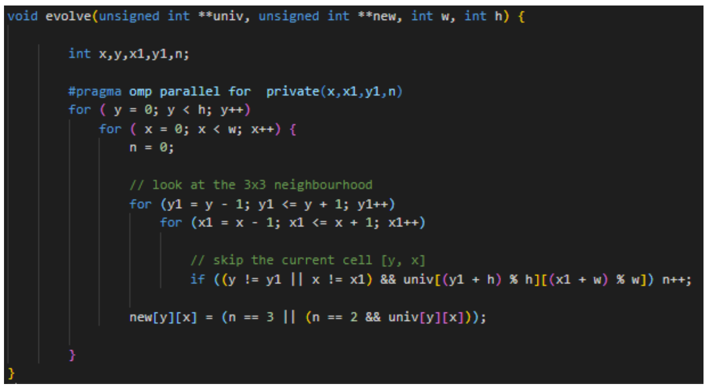
OPENMP

First, we provide a parallel implementation with **OpenMP**, which is an API to support vectorization and explicitly direct multi-threaded shared-memory parallelism.

The hotspot of the program is the evolution stage, where it starts from the current generation and compute the next one. The usage of two matrices for the computation allows to make independent many operations and so they can be parallelized, in fact the evaluation of the rules for each cell of the game board is independent from the other cells.

In this way the program executes the same simple operations on different data, such paradigm is called *SIMD* (Single instruction multiple data).

To instruct the program to behave properly must be used the directive *omp parallel for*, which schedule a given number of threads to compute the independently some of the iterations of the outer loop in any order. Adding the directive *private* each thread has its own variable to avoid race conditions, while the two matrices are shared among the threads. By default, the threads are scheduled in *static* manner because we expect the amount of work for each iteration to be almost the same.



```c
void evolve(unsigned int **univ, unsigned int **new, int w, int h) {

        int x,y,x1,y1,n;

        #pragma omp parallel for  private(x,x1,y1,n)
        for ( y = 0; y < h; y++)
            for ( x = 0; x < w; x++) {
                n = 0;

                // look at the 3x3 neighbourhood
                for (y1 = y - 1; y1 <= y + 1; y1++)
                    for (x1 = x - 1; x1 <= x + 1; x1++)

                        // skip the current cell [y, x]
                        if ((y != y1 || x != x1) && univ[(y1 + h) % h][(x1 + w) % w]) n++;

                new[y][x] = (n == 3 || (n == 2 && univ[y][x]));

        }
}
```

The inner loop can be vectorized since is made up of simple operations. As it's done in the serial program, after the computation of the generations the two matrices are swapped.

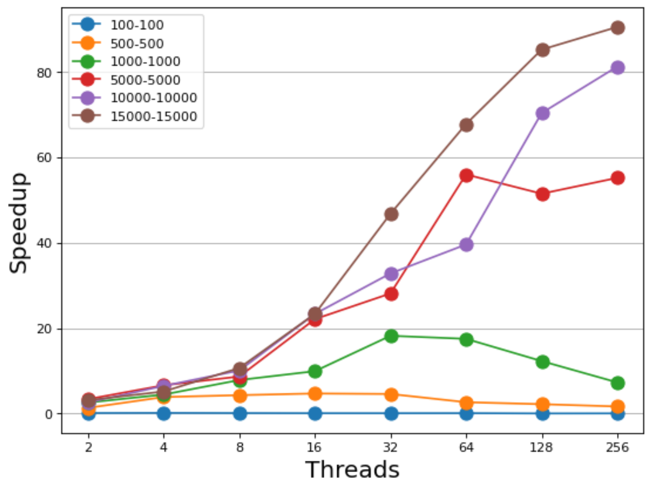
Results

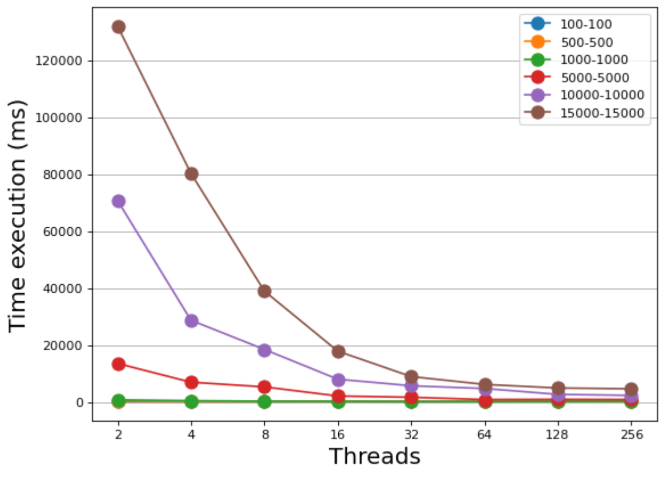
The program has been tested on a single node of the cluster with different numbers of threads and with different dimensions of the problem. Starting from the specifications of the processors, the number of threads should follow the power of 2.

Starting from 2 threads because is the minimum to exploit the parallelization, up to 256 which is the maximum number of threads supported by the given CPU.

	Inputs					Number of threads					
Width	**Height**	**Iterations**	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**
100	100	10	18.55 ms	0.14	**0.17**	0.13	0.10	0.10	0.12	0.05	0.06
500	500	10	469.74 ms	1.35	3.88	4.32	**4.71**	4.60	2.68	2.21	1.68
1000	1000	10	1871.70 ms	2.60	4.46	7.86	9.93	**18.21**	17.50	12.30	7.31
5000	5000	10	45914.28 ms	3.39	6.63	8.69	22.11	28.12	**55.96**	51.46	55.16
10000	10000	10	186158.10 ms	2.63	6.49	10.10	23.33	32.81	39.58	70.33	**81.18**
15000	15000	10	416311.409 ms	3.15	5.18	10.69	23.34	46.77	67.75	85.16	**90.46**

Speedup





As can be seen from the table and the plots above, for a very little game board (100 x 100 in particular) there is no advantages (or a little) of using OpenMP because the vectorized serial program is faster, since because also OpenMP add the cost of scheduling the threads. Instead as the problem size increases the advantages is more evident because the overhead of the threads became negligible, but in any case, as the number of threads decreases the speedup increases more slowly since it starts decreasing.

MPI

The General Idea

To provide a parallel implementation of the Conway's *Game of Life* with ***MPI***, it necessary distribute the workload across multiple machines that works in parallel, and this allows to increase the processing power, but unlike OpenMP there is not a shared memory architecture. In fact, every process has its own memory and cannot access to the memory of another process, so the way to parallelize with MPI (*Message Passing Interface*) is with the communication between processes.

So, the main idea behind the parallelization with MPI is to split the entire grid of the game into blocks that represent the local grid belonging to a single process, and the number of the local grids depends on the numbers of processes used to parallelize the program. Hence, as first step, each process must calculate the size of its own local grid, for this reason everyone must know the dimensions of the original grid.

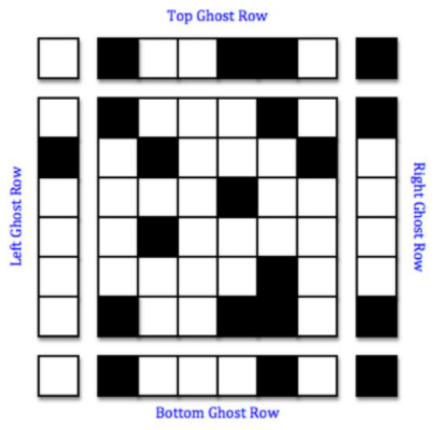


Figure 1 - Example of Ghost Rows in a block

The strategy used to split the original grid is by **stripes**, in fact each local grid has same number of columns as the starting grid, instead the number of rows is obtained dividing the total number of the original rows by the total numbers of processes in the cluster. If the latter division is not exact, the last node of the cluster computes also the remainder of division and it is summed to its number of rows.

The next important step is to manage the evolution of the cells in the border of each block. To deal with this problem, they were added the **"ghost rows",** as in the example in *Figure 1.* These extra rows are those around a local grid, and, in the case of a single block, the top ghost row is the copy of the last row of the block, instead, the bottom ghost row is the copy of first row of the block. In the same way, the left ghost column is the copy of the last column of the block, and the right ghost row is the copy of the first column of the block.

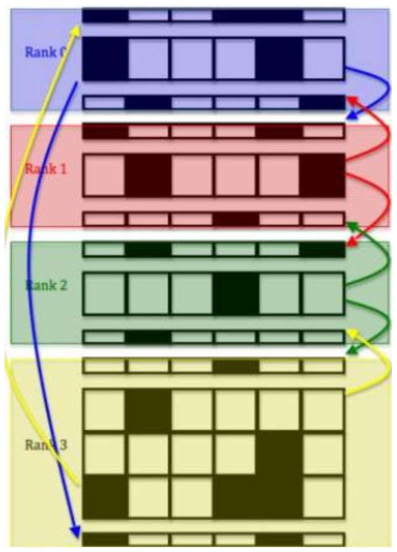


Figure 2 - Processes Communication Example

In the case of multiple blocks, instead, there are a series of stacked blocks, as many as the number of processes, and the situation a bit more complicated, because the cells that lie on top and bottom border of the block could have some cells that should arise in another block belonging to another process. Hence, the processes need to communicate with their neighbors, and this communication occurs thanks to the use of the top and bottom ghost rows. In fact, in the top ghost row, is stored the last row of the block of the upper neighbor, and in the bottom ghost row is stored the first row of the lower neighbor. The Figure-*2* shows an example of the dynamic of the communication between processes. As can be noticed, the last node communicates also with the first node, and the first node communicate also with the last process. To implement this kind of communication between processes, everyone must know who the upper and lower neighbor is, to send them, respectively, the first and the last row. Then each node needs to receive from the neighbors, thanks to the use of the ghost rows. For the ghost column, instead, each node copies the last column of the block in the left ghost row, and the first column of the block in the right ghost row (similar to the *figure 1*).

After these operations, every node can update its block computing the evolution of the cells, using the rules mentioned in the introduction, and then send its block to the root node to display the global evolution in the console.

Implementation Details

To parallelize the *GoL* across the cluster it was necessary to use the MPI methods which allows to implement the communication between the processes.

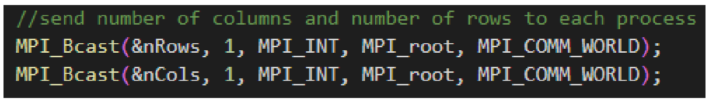

```
//send number of columns and number of rows to each process
MPI_Bcast(&nRows, 1, MPI_INT, MPI_root, MPI_COMM_WORLD);
MPI_Bcast(&nCols, 1, MPI_INT, MPI_root, MPI_COMM_WORLD);
```

Figure 3 - Send original grid size to all processes.

As mentioned in the previous section it's necessary that all the processes know some information like the numbers of rows, columns of the entire grid and the total number of timestep. To make this possible, MPI provides the method **MPI_Bcast ()** (*Figure 3*), the root node sends this information to all the other nodes. In this way each node can compute the number of rows and columns of its own block that represent a piece of the starting grid. So, each node has a struct, *gen_block,* that stores the size of the block, together with the rank of the *upper and lower neighbors*, the *rank* of the process, the size of the communicator **MPI_COMM_WORLD** (that is the default communicator) and a pointer to a matrix, *block*, that represent the local grid (Figure *4*), which was initialized with random values, which can be 0 or 1 (ALIVE or DEAD).

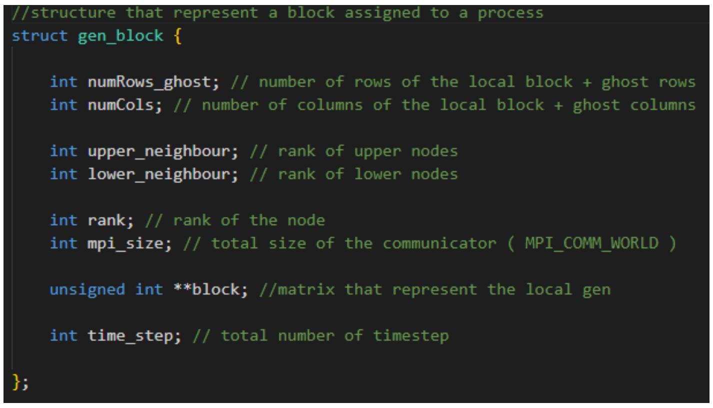

```
//structure that represent a block assigned to a process
struct gen_block {

    int numRows_ghost; // number of rows of the local block + ghost rows
    int numCols; // number of columns of the local block + ghost columns

    int upper_neighbour; // rank of upper nodes
    int lower_neighbour; // rank of lower nodes

    int rank; // rank of the node
    int mpi_size; // total size of the communicator ( MPI_COMM_WORLD )

    unsigned int **block; //matrix that represent the local gen

    int time_step; // total number of timestep

};
```

Figure 4 - Struct that represent a block assigned to a node.

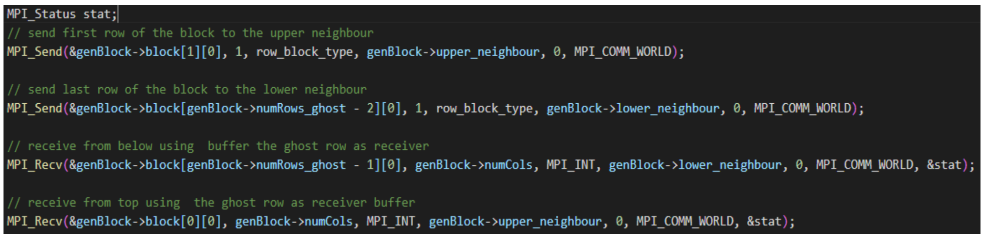

```
MPI_Status stat;
// send first row of the block to the upper neighbour
MPI_Send(&genBlock->block[1][0], 1, row_block_type, genBlock->upper_neighbour, 0, MPI_COMM_WORLD);

// send last row of the block to the lower neighbour
MPI_Send(&genBlock->block[genBlock->numRows_ghost - 2][0], 1, row_block_type, genBlock->lower_neighbour, 0, MPI_COMM_WORLD);

// receive from below using  buffer the ghost row as receiver
MPI_Recv(&genBlock->block[genBlock->numRows_ghost - 1][0], genBlock->numCols, MPI_INT, genBlock->lower_neighbour, 0, MPI_COMM_WORLD, &stat);

// receive from top using  the ghost row as receiver buffer
MPI_Recv(&genBlock->block[0][0], genBlock->numCols, MPI_INT, genBlock->upper_neighbour, 0, MPI_COMM_WORLD, &stat);
```

Figure 5 - evolve(): Send to neighbors and receive from neighbors.

As already mentioned in the previous section, before the evolve step, to manage with the problem of the cells in the border, each node sends to the last row of the local grid to the lower neighbor and the first row to the upper neighbor. So, each node call two **MPI_send(),** and also two **MPI_Recv(),** to receive from the neighbors, using the top and the bottom *ghost rows* as receiver buffer (*Figure 5*). To send the rows of the matrix in an efficient way, MPI provides the **derived datatype**. In this case was useful the **MPI_Type_contiguos** that constructs a type of map consisting of the replication of a datatype into contiguous locations. The new type is the datatype obtained concatenating copies of the old type. It was used to send the rows of the matrix in a better way since in a row the elements are contiguous in memory, passing the number of elements contiguous to send and the type. As showed in figure 6, the MPI_Type_contiguos takes as parameter the number of elements contiguous in memory to send, the type of these elements and the variable that will represent the new derived datatype, *row_block_type*.

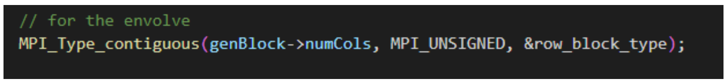

```
// for the envolve
MPI_Type_contiguous(genBlock->numCols, MPI_UNSIGNED, &row_block_type);
```

Figure 6 - MPI derived datatype to send a row of a matrix.

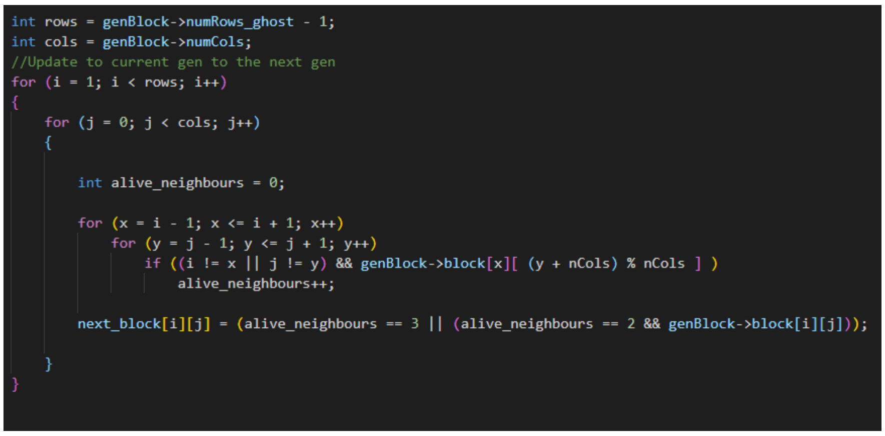
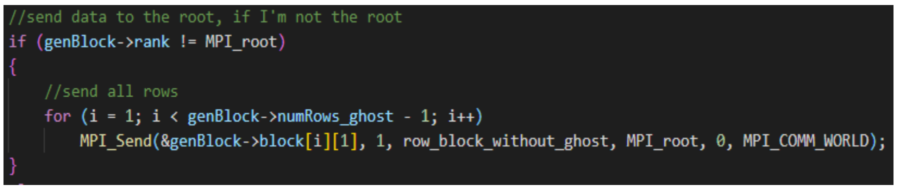

```
int rows = genBlock->numRows_ghost - 1;
int cols = genBlock->numCols;
//Update to current gen to the next gen
for (i = 1; i < rows; i++)
{
    for (j = 0; j < cols; j++)
    {

        int alive_neighbours = 0;

        for (x = i - 1; x <= i + 1; x++)
            for (y = j - 1; y <= j + 1; y++)
                if ((i != x || j != y) && genBlock->block[x][ (y + nCols) % nCols ] )
                    alive_neighbours++;

        next_block[i][j] = (alive_neighbours == 3 || (alive_neighbours == 2 && genBlock->block[i][j]));

    }
}
```

Figure 7 – evolve(): Computation of the evolution of the cells.

In this kind of implemetation, the ghost columns are not needed (left and right rows mentioned in the previous paragraph), because the cells that lie in the left and right border are managed so that if a cell must arise respectively to the left and to the right, they will arise in the opposite side of the block.



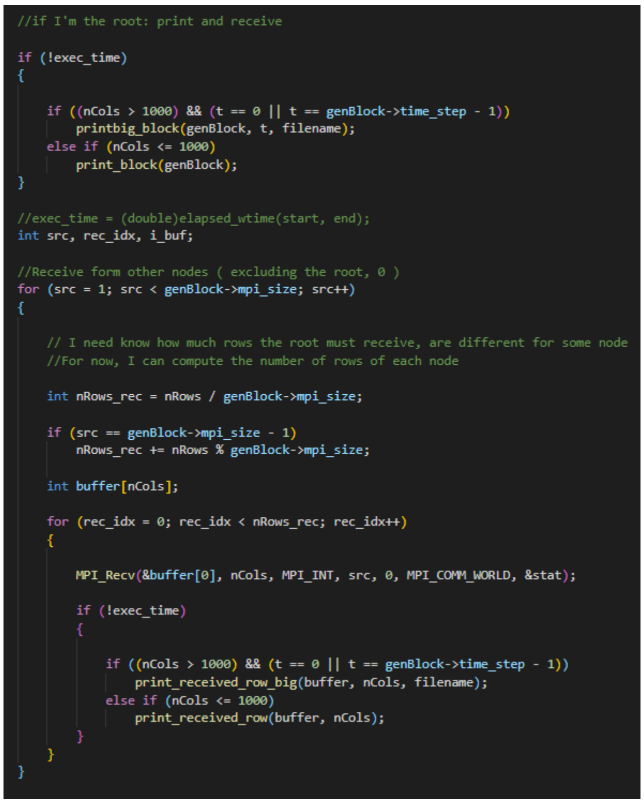
```
//send data to the root, if I'm not the root
if (genBlock->rank != MPI_root)
{
    //send all rows
    for (i = 1; i < genBlock->numRows_ghost - 1; i++)
        MPI_Send(&genBlock->block[i][1], 1, row_block_without_ghost, MPI_root, 0, MPI_COMM_WORLD);
}
```

Figure 8 – Display version 1: Each node sed its own local grid to the root node using MPI_Type_contiguos

In the *figure 7* is showed the computation of the evolution, located in the *evolve()* function, in each time step and the result of the evolution is stored in another temporary matrix, *next_block*.

At every time step is called the evolve () and the current *block*, is updated with the result stored in the *next_block*, swapping *block* and *next_block* like it's been done also for the sequential program.

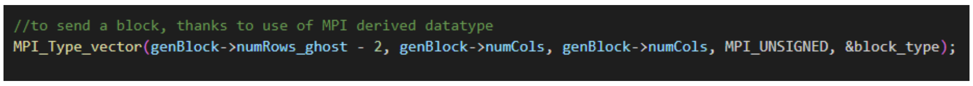
Finally, there is the need to assemble all the proccesses' local grid, the process are expected to send the evolution computed of its local grid to the root node, so that the root can reassemble all the local grids and then display it. These operations are implemented in the *display()* function.



```
//if I'm the root: print and receive

if (!exec_time)
{

    if ((nCols > 1000) && (t == 0 || t == genBlock->time_step - 1))
        printbig_block(genBlock, t, filename);
    else if (nCols <= 1000)
        print_block(genBlock);
}

//exec_time = (double)elapsed_wtime(start, end);
int src, rec_idx, i_buf;

//Receive form other nodes ( excluding the root, 0 )
for (src = 1; src < genBlock->mpi_size; src++)
{

    // I need know how much rows the root must receive, are different for some node
    //For now, I can compute the number of rows of each node

    int nRows_rec = nRows / genBlock->mpi_size;

    if (src == genBlock->mpi_size - 1)
        nRows_rec += nRows % genBlock->mpi_size;

    int buffer[nCols];

    for (rec_idx = 0; rec_idx < nRows_rec; rec_idx++)
    {

        MPI_Recv(&buffer[0], nCols, MPI_INT, src, 0, MPI_COMM_WORLD, &stat);

        if (!exec_time)
        {

            if ((nCols > 1000) && (t == 0 || t == genBlock->time_step - 1))
                print_received_row_big(buffer, nCols, filename);
            else if (nCols <= 1000)
                print_received_row(buffer, nCols);
        }
    }
}
```

Figure 9- Display version 1: The root receives from all the other nodes of the cluster.

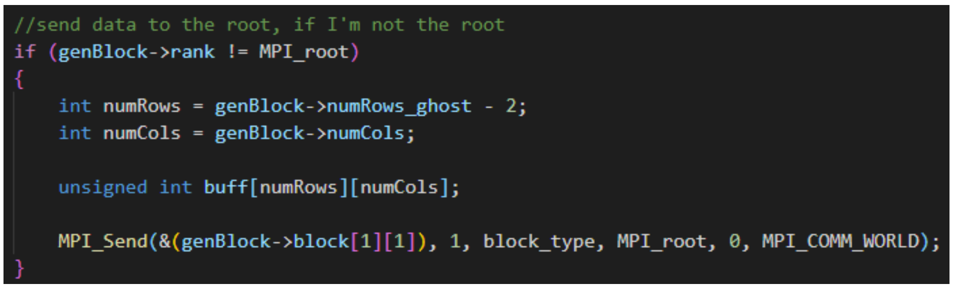
Here are implemented two version of the *display*() with 2 methods, which differ in the way the local grid are sent to the root. In the **version 1** the processes that are not root send the local grid, row by row with a series of *MPI_Send() (Figure 8).* Then the root display its own local grid, and then need to receive the rows assigned to the process. At the end of this operation the root can display the entire grid of the game *(Figure 9).*



```
//to send a block, thanks to use of MPI derived datatype
MPI_Type_vector(genBlock->numRows_ghost - 2, genBlock->numCols, genBlock->numCols, MPI_UNSIGNED, &block_type);
```

Figure 10- Display version 2 - MPI TYPE VECTOR

In the **version 2**, each node, instead of send row by row the local grid, it is sent the entirely grid using another MPI derived datatype, that is **MPI_Type_vector,** which allows to regular gaps (stride) in the displacements. It takes how many blocks to send, the number of elements to send, the stride, the type of the elements and the new type, called in this case *block_type* (*Figure 10*). So, using this datatype the nodes that are not root can send the entire block to the root node, as showed in *figure 11.*

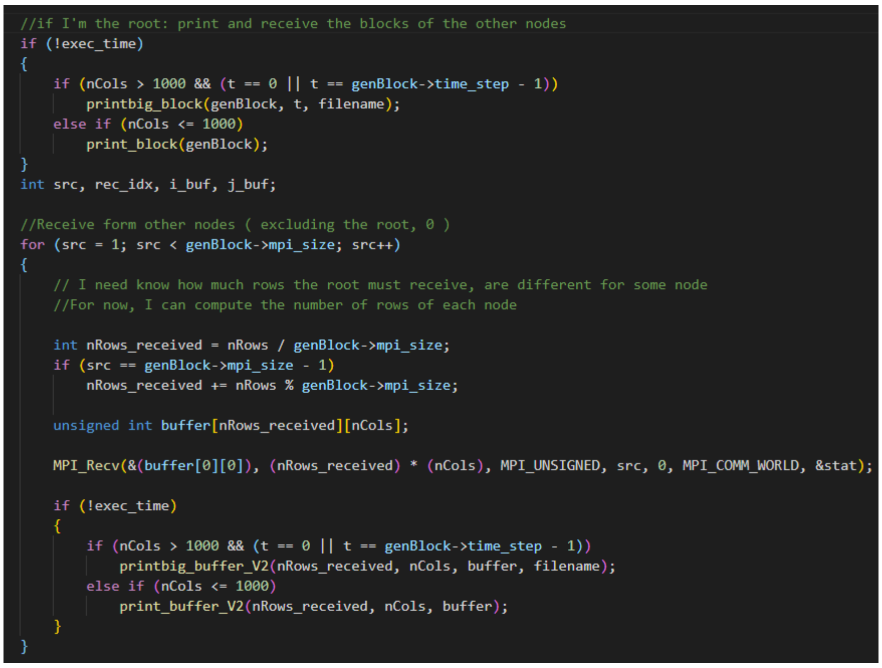


```
//send data to the root, if I'm not the root
if (genBlock->rank != MPI_root)
{
    int numRows = genBlock->numRows_ghost - 2;
    int numCols = genBlock->numCols;

    unsigned int buff[numRows][numCols];

    MPI_Send(&(genBlock->block[1][1]), 1, block_type, MPI_root, 0, MPI_COMM_WORLD);
}
```

Figure 11 – Display version 2: Each node sends its own local grid to the root node using MPI_Type_vector

Then the root node print all the grids together (*Figure 12*).

Figure 12 – Display version 2: Each node sed its own local grid to the root node using MPI_Type_vector



```c
//if I'm the root: print and receive the blocks of the other nodes
if (!exec_time)
{
    if (nCols > 1000 && (t == 0 || t == genBlock->time_step - 1))
        printbig_block(genBlock, t, filename);
    else if (nCols <= 1000)
        print_block(genBlock);
}
int src, rec_idx, i_buf, j_buf;

//Receive form other nodes ( excluding the root, 0 )
for (src = 1; src < genBlock->mpi_size; src++)
{
    // I need know how much rows the root must receive, are different for some node
    //For now, I can compute the number of rows of each node

    int nRows_received = nRows / genBlock->mpi_size;
    if (src == genBlock->mpi_size - 1)
        nRows_received += nRows % genBlock->mpi_size;

    unsigned int buffer[nRows_received][nCols];

    MPI_Recv(&(buffer[0][0]), (nRows_received) * (nCols), MPI_UNSIGNED, src, 0, MPI_COMM_WORLD, &stat);

    if (!exec_time)
    {
        if (nCols > 1000 && (t == 0 || t == genBlock->time_step - 1))
            printbig_buffer_V2(nRows_received, nCols, buffer, filename);
        else if (nCols <= 1000)
            print_buffer_V2(nRows_received, nCols, buffer);
    }
}
```

MPI – Experiments

The performance of this implementation are measured taking the execution time of the node root, measuring the time to execute the *evolve()* function plus the time the root takes to receive all the local grids from the other processes to get the total execution time. All the measurements are made using the version 2 of the *display*() because it is the most efficient, since all the processes call a single *MPI_Send()*, and the root calls a single *MPI_Recv()* for each process. Moreover, the second version use of the derived datatype, *MPI_Type_vector* allow to reduce the communication overhead, that is the waiting time in the communication between processes, with a gain in terms of execution time.

Starting from the execution time, the speedup is computed and used to fill the tables showed below. The aim of these tables is to show how MPI performs in terms of strong scaling. In each of the tables below are reported the **serial** execution time in milliseconds of the vectorized program, that is the baseline used to compute the **speedup** for each number of nodes considered and for each number of processes used to run the experiments. The experiments are executed considering the following list of grid dimensions:

$$[\ (100,100)\ (500,500)\ (1000,1000)\ (5000,5000)\ (10000,10000)\ (15000,15000)\ ]$$

The experiments are repeated with *1, 2, 4, 8 nodes* of the cluster. For 1 node the speedup has been tested exploiting also all the threads, instead for the other nodes is exploited at most all the cores. For this reason, the list of the total number of processes to use for each node is different.

The list of number of processes are the following:

Nodes	List of number of processes
1	[2 4 8 16 32 64 128 256]
2	[2 4 8 16 32 64 128]
4	[4 8 16 32 64 128 256]
8	[8 16 32 64 128 256 512]

,

In single experiment is executed with the following bash MPI command:

mpiexec -hostfile host_list_n.txt -perhost n_process_per_host -np tot_procesesses ./bin_file nRows nCols timesteps version show_result nNodes

In particular:

- *host_list_n.txt* is file that contains the list of the host used, and n is the number of nodes (1, 2, 4, 8). *n* must be replaced with the number of nodes chosen.
- *n_process_per_host* is the number of process that each node must executes, obtained computing the division between the total number of processes and the number of nodes.
- *tot_procesess* is the total number of processes.
- *nRows, nCols* are the dimensions of the grid.
- *timesteps* are the time steps with which is executed the game. In this case is fixed to 10.
- *Version* is the display version used.
- *show_result* is a flag that indicates if show or not the evolution of the game in the terminal for *nCols* lower than 1000.
- *nNodes* is the number of nodes of the cluster used.

MPI Table 1

100 x 100	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1	18.55 ms	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	-
2	18.55 ms	0.01	0.01	0.01	0.01	0.01	0.01	0.01	-	-
4	18.55 ms	-	0.01	0.01	0.01	0.01	0.01	0.01	0.01	-
8	18.55 ms	-	-	0.01	0.01	0.01	0.01	0.01	0.01	0.01

Speedup

MPI Table 2

500 x 500	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1	469.74 ms	0.21	0.22	0.22	0.23	0.23	0.23	0.22	0.21	-
2	469.74 ms	0.21	0.22	0.23	0.23	0.23	0.23	0.22	-	-
4	469.74 ms	-	0.22	0.23	0.23	0.23	0.23	0.22	0.22	-
8	469.74 ms	-	-	0.22	0.23	0.23	0.23	0.22	0.22	0.20

Speedup

MPI Table 3

1000 x 1000	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1	1871.70 ms	0.72	0.80	0.84	0.87	**0.89**	0.87	0.84	0.77	-
2	1871.70 ms	0.70	0.78	0.84	0.87	0.88	**0.89**	0.88	-	-
4	1871.70 ms	-	0.81	0.85	0.86	**0.88**	0.80	0.86	0.84	-
8	1871.70 ms	-	-	0.86	0.87	**0.88**	0.80	0.84	0.83	0.74

Speedup

The first 3 tables above (*MPI Table 1-3*) show the speedup for a relatively small size of the game. When the dimensions of the grids are too small, the parallelization with MPI is not a great choice because the speedups are all lower than zero, and this means that the sequential version of Game of Life is faster than the one parallelized with MPI because of the communication overheads.

MPI Table 4

5000 x 5000	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1	45914.28 ms	3.26	6.33	8.60	17.12	24.25	38.03	**40.57**	13.74	-
2	45914.28 ms	3.23	6.20	11.35	17.13	21.86	45.18	**57.87**	-	-
4	45914.28 ms	-	7.41	11.66	15.40	30.00	29.52	44.00	**53.20**	-
8	45914.28 ms	-	-	13.5	18.38	20.27	38.32	**49.44**	45.02	25.23

Speedup

MPI Table 5

10000 x 10000	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1	186158.10 ms	3.00	6.82	12.16	17.30	26.02	**49.60**	44.17	46.90	-
2	186158.10 ms	3.28	7.18	11.97	17.30	33.08	37.12	**62.54**	-	-
4	186158.10 ms	-	6.58	12.00	20.27	33.49	47.09	61.12	**68.19**	-
8	186158.10 ms	-	-	11.56	20.41	32.85	48.80	56.83	**67.84**	42.14

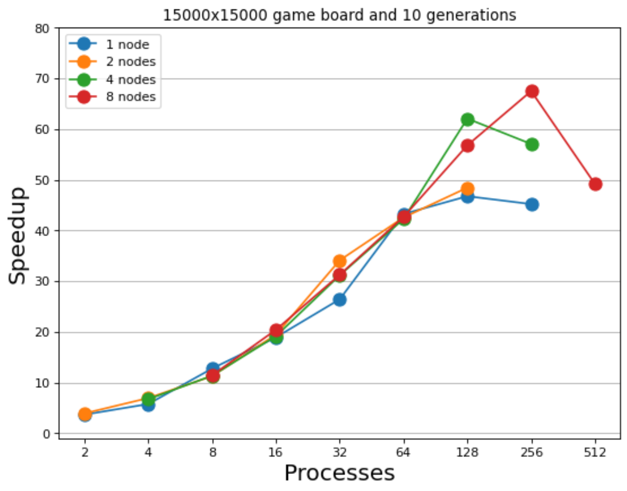
Speedup

MPI Table 6

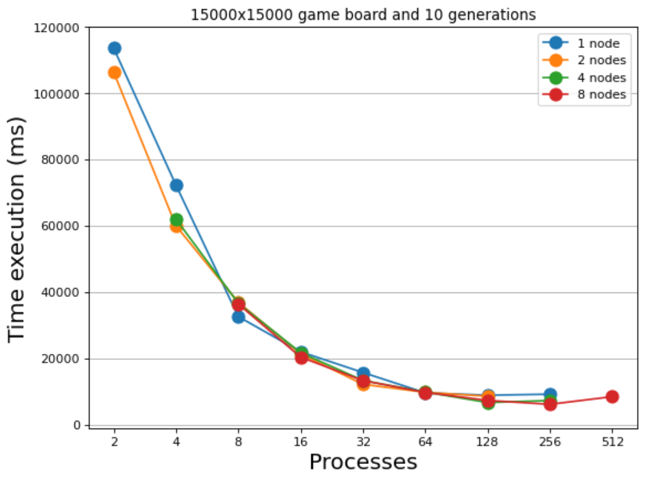
15000 x 15000	**Serial**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1	45914.28 ms	3.66	5.75	12.75	18.91	26.42	43.25	**46.75**	45.20	-
2	45914.28 ms	3.91	6.93	11.26	19.45	34.11	42.56	**48.44**	-	-
4	45914.28 ms	-	6.71	11.34	19.14	31.16	42.26	**62.06**	57.06	-
8	45914.28 ms	-	-	11.42	20.40	31.25	42.67	56.79	**67.50**	49.17

Speedup

The *MPI table 4-6* report a different situation with bigger dimensions of the grids and can be better seen the benefits of the parallelization with MPI. More the problem dimension is big the more is convenient the parallelization with MPI, this happens because the benefit of the parallelization exceeds the cost of communication between the nodes. The better results are obtained using more than two nodes of the cluster.



MPI graphic 1 – Speedup vs. number of Processes



MPI graphic 2 – Execution Time vs. number of Processes

The plots above show the execution time respect and the speedup as the number of processes increases only for the executions with a grid 15000x15000. The time execution decreases very fast until it become more stable as the number of processes increase, with the biggest number of processes the time execution seems to increase a little. This can be also seen from the speedups that at a certain point (different for each number of used nodes) they start decreasing due to the overheads of communication.

These results highlight that to obtain good performances with MPI, the problem should be evenly distributed among enough nodes. But also, the communication among the nodes can become a bottleneck if they are used to many processes.

CUDA

We provide an implementation of the algorithm also with **CUDA c**, which allows to execute part of the code on the GPU.

First, they must be identified the regions of the program to execute on the CPU (**host**) and those that must be executed on the GPU (**device**), as described in the previous sections the time-consuming part is the one computing the transition between the generations.

A good strategy for GPU programming is the **memory coalescing,** that allow to get the maximum memory bandwidth, reached when data is loaded for multiple threads in a single transaction. This happens when consecutive threads read consecutive memory addresses within a **warp.** Thanks to the use of this technique is possible to improve dramatically the performance of the execution code.

To exploit such optimization the way the game board is stored in the memory and the way the program access it must be rearranged.

Implementation Details

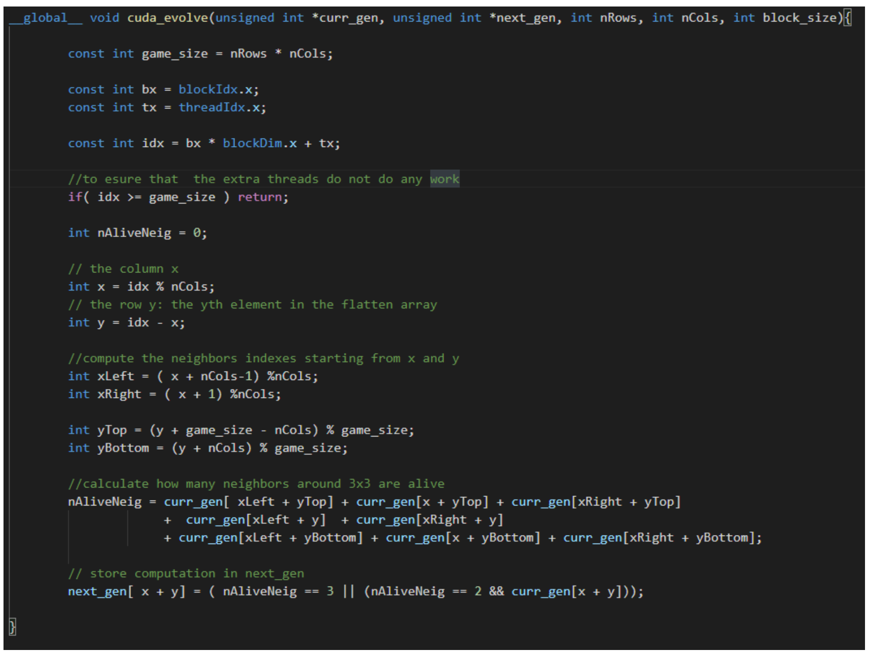
To implement a parallelization in CUDA, it was necessary follow 3 main steps:

1. **Identify data-parallel, computationally intensive portions of the original code:**

   The sequential implementation code was analyzed to find out the possible CUDA kernels candidates. So the function *evolve ()* was isolated and transformed in a **CUDA kernel**. This is the one that compute the evolution of the game.

2. **Translate identified CUDA kernel candidates into real CUDA kernels functions:**

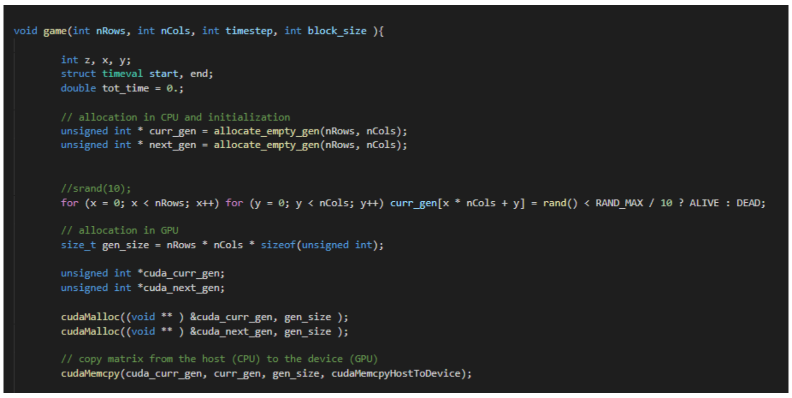
   Since it is a CUDA kernel before the returned type of the function (that must be *void*), has a qualifier keywords, **__global__**. This indicates that is a function called by the host but executed on the device.



```
__global__ void cuda_evolve(unsigned int *curr_gen, unsigned int *next_gen, int nRows, int nCols, int block_size){

        const int game_size = nRows * nCols;

        const int bx = blockIdx.x;
        const int tx = threadIdx.x;

        const int idx = bx * blockDim.x + tx;

        //to esure that  the extra threads do not do any work
        if( idx >= game_size ) return;

        int nAliveNeig = 0;

        // the column x
        int x = idx % nCols;
        // the row y: the yth element in the flatten array
        int y = idx - x;

        //compute the neighbors indexes starting from x and y
        int xLeft = ( x + nCols-1) %nCols;
        int xRight = ( x + 1) %nCols;

        int yTop = (y + game_size - nCols) % game_size;
        int yBottom = (y + nCols) % game_size;

        //calculate how many neighbors around 3x3 are alive
        nAliveNeig = curr_gen[ xLeft + yTop] + curr_gen[x + yTop] + curr_gen[xRight + yTop]
                    +  curr_gen[xLeft + y]  + curr_gen[xRight + y]
                    + curr_gen[xLeft + yBottom] + curr_gen[x + yBottom] + curr_gen[xRight + yBottom];

        // store computation in next_gen
        next_gen[ x + y] = ( nAliveNeig == 3 || (nAliveNeig == 2 && curr_gen[x + y]));

}
```

Figure 13 - Evolve converted in a CUDA kernel.

   Then, to implement the kernel, the first thing to do is to compute coordinates (x, y) of the block and the threads. This because a GPU is composed by a grid of blocks, and each block is composed by threads, and each block or thread has a unique ID. So, **threadIdx** gives the thread coordinates inside the block, (x, y), and **blockIdx** gives the coordinates of the block inside the grid. The matrix containing the game board is stored as a flatten and contiguous array so that to each thread can be associate to the index of one cell of the matrix (in this way contiguous threads in warps access contiguous cells). Starting from that the corresponding coordinates of the matrix can be found and use to compute the neighbors, as showed in figure 13. There is the possibilty that there are more threads than needed. Hence, in order to ensure that the extra threads do not any work, if the corresponding index associated is greater than the board size then the function exit.



```c
void game(int nRows, int nCols, int timestep, int block_size ){

	int z, x, y;
	struct timeval start, end;
	double tot_time = 0.;

	// allocation in CPU and initialization
	unsigned int * curr_gen = allocate_empty_gen(nRows, nCols);
	unsigned int * next_gen = allocate_empty_gen(nRows, nCols);


	//srand(10);
	for (x = 0; x < nRows; x++) for (y = 0; y < nCols; y++) curr_gen[x * nCols + y] = rand() < RAND_MAX / 10 ? ALIVE : DEAD;

	// allocation in GPU
	size_t gen_size = nRows * nCols * sizeof(unsigned int);

	unsigned int *cuda_curr_gen;
	unsigned int *cuda_next_gen;

	cudaMalloc((void ** ) &cuda_curr_gen, gen_size );
	cudaMalloc((void ** ) &cuda_next_gen, gen_size );

	// copy matrix from the host (CPU) to the device (GPU)
	cudaMemcpy(cuda_curr_gen, curr_gen, gen_size, cudaMemcpyHostToDevice);
```

Figure 15 – allocation of memory for host and device and initialization

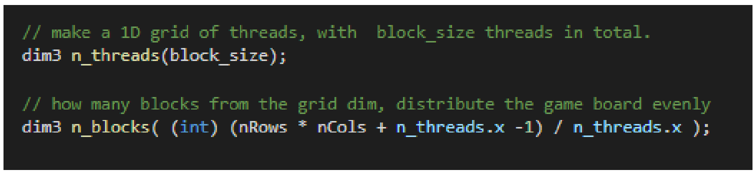
At this point is possible make the computation of the envolve, counting how many neighbors are alive or dead. So, there is a need to compute the 8 neighbors around the cell in position (x, y) in the 1-dimentional vector, finding the coordinates as they were in a matrix (Fig. 13).

Modify code in order to manage memory and kernel calls:

In the *game*() functions, similarly to OpenMP and MPI implementations, the starting matrix was allocated and initialized. But also it must be allocated space in the memory of the *device* using the **cudaMalloc()** function. Then, the starting grid is transferred from the *host* to the *device* using **cudaMemcpy()**, which takes, as parameters, the destination, the source, the size to copy and the macro that indicates the direction to transfer the data, that in this case is **cudaMemcpyHostToDevice** (Figure 16).

Before call the kernel must be define how many blocks and how many threads per block to use.

Fixed the number of threads per block, the number of blocks can be adapted depending on the dimension of the board with the following formula:



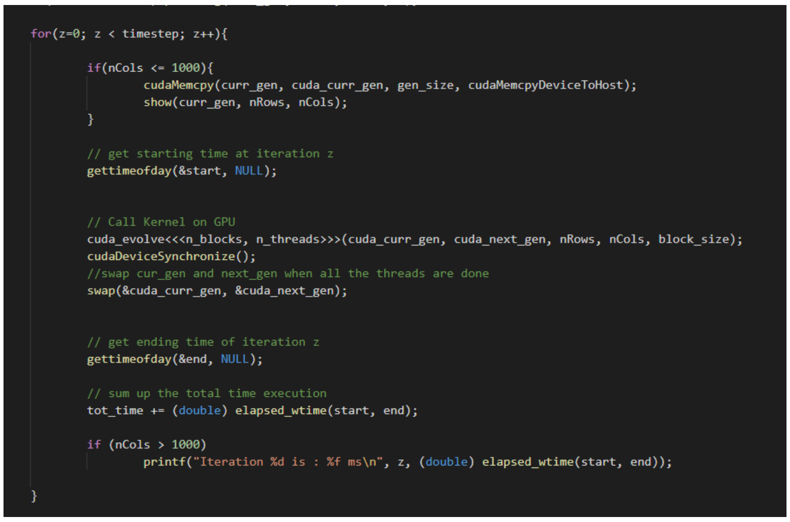
```c
// make a 1D grid of threads, with  block_size threads in total.
dim3 n_threads(block_size);

// how many blocks from the grid dim, distribute the game board evenly
dim3 n_blocks( (int) (nRows * nCols + n_threads.x -1) / n_threads.x );
```

Figure 17 –definition of number of threads per block and number of blocks

$$n\_blocks = (N + block\_size - 1) / block\_size$$

where N is the size of a 1 dimentional vector (*figure 17*), in this way the program evenly distribute the cells of the matrix between the blocks.

After the computation of *cuda_evolve ()* the function *cudaDeviceSynchronize()* allows the synchronization between the device and the host so that the current generation and the next generation are swapped to update the evolution of the game, as it is done for all the other versions presented before (*figure 18*).



```c
for(z=0; z < timestep; z++){

	if(nCols <= 1000){
		cudaMemcpy(curr_gen, cuda_curr_gen, gen_size, cudaMemcpyDeviceToHost);
		show(curr_gen, nRows, nCols);
	}

	// get starting time at iteration z
	gettimeofday(&start, NULL);


	// Call Kernel on GPU
	cuda_evolve<<<n_blocks, n_threads>>>(cuda_curr_gen, cuda_next_gen, nRows, nCols, block_size);
	cudaDeviceSynchronize();
	//swap cur_gen and next_gen when all the threads are done
	swap(&cuda_curr_gen, &cuda_next_gen);


	// get ending time of iteration z
	gettimeofday(&end, NULL);

	// sum up the total time execution
	tot_time += (double) elapsed_wtime(start, end);

	if (nCols > 1000)
		printf("Iteration %d is : %f ms\n", z, (double) elapsed_wtime(start, end));

}
```

Figure 18 – Kernel Call for each time step

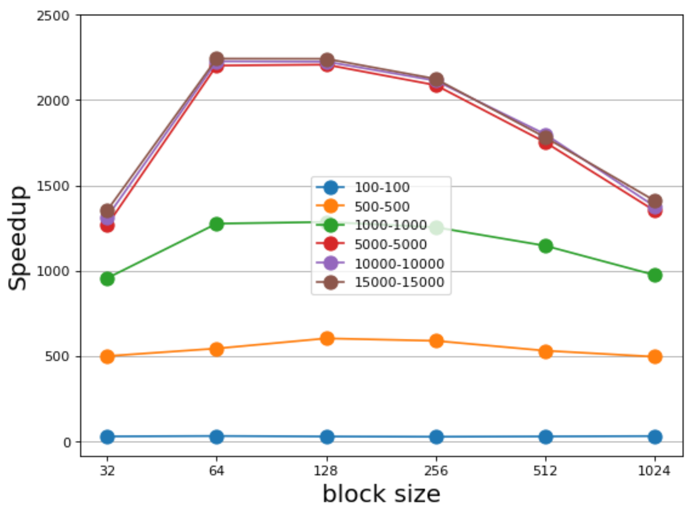
Experiments

The nodes of the cluster we used are equipped with the GeForce GTX 1650. Such GPU has a compute capability of **7.5**. With this value is possible to know the computational capabilities and in particular we are interested in the **warp** that is composed of *32 threads*, and a **block** with *1024 threads*. So, the GPU multiprocessor creates, manages, schedules, and executes threads in groups of 32, that are the **warps.**

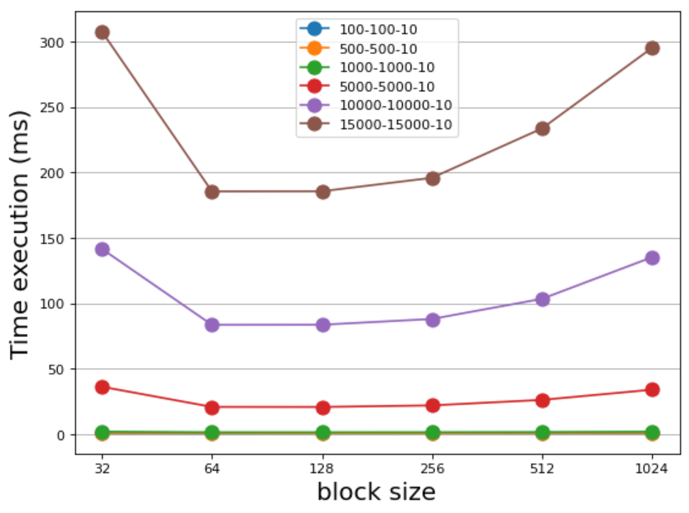
We evaluate the performances of the parallelization by changing the number of threads per block: 32 is the smallest number of threads per block used because of the hardware device, since 32 is the dimension of a warp and less than 32 would lead to an inefficient parallelization. All the other values are chosen as multiple of the warp size and 1024 is the maximum number of threads supported for a block.

Inputs				Number of threads					
Width	**Height**	**Iterations**	**Serial**	**32**	**64**	**128**	**256**	**512**	**1024**
100	100	10	18.55ms	28.15	**30.72**	27.77	26.35	28.07	30.07
500	500	10	469.74ms	499.19	543.68	**603.01**	588.65	530.78	495.51
1000	1000	10	1871.70ms	955.92	1275.87	**1285.50**	1254.49	1146.17	974.33
5000	5000	10	45914.28ms	1266.67	2202.75	**2207.94**	2087.67	1752.78	1350.97
10000	10000	10	186158.10ms	1314.55	**2227.33**	2225.44	2114.30	1799.95	1376.65
15000	15000	10	416311.40ms	1352.91	**2243.23**	2242.28	2124.76	1781.65	1410.05

Speedup



Speedup vs. block size



Time execution vs. block size

The solution provided by CUDA seems to be the best one for all the game dimensions tested, in particular for the bigger dimensions. This happen because the GPU is optimal for computation on grids of data of different dimensions, since **CUDA** provide supports the definitions of 1D, 2D, 3D grids blocks and threads. Our implementation does not include any *if* statements that can produce **warp divergence**.

As highlighted in the table the best speedup can be achieve with 64 or 128 threads per block. For small dimension of the problem (100 x 100 in particular) the program does not achieve a high speedup like in the other experiments because the sequential solution executes in a few milliseconds thanks to the vectorization.