Foundations of HPC [586SM]
Report exam's assignments

Paolo Da Rold

September 2024

Contents

1 Exercise 1 **3**
1.1 Introduction . . . 3
1.2 Methodology . . . 5
1.3 Implementation . . . 6
1.3.1 Read and Write . . . 7
1.3.2 Initialize . . . 7
1.3.3 Static evolution . . . 8
1.3.4 Ordered evolution . . . 13
1.3.5 Main . . . 14
1.3.6 Run the game . . . 15
1.4 Results and Discussion . . . 16
1.4.1 OpenMP scalability . . . 16
1.4.2 Strong MPI scalability . . . 18
1.4.3 Weak MPI scalability . . . 19
1.5 Conclusions . . . 22

2 Exercise 2 **23**
2.1 Introduction . . . 23
2.2 Implementation . . . 23
2.3 Results and Discussion . . . 24
2.3.1 Core Scalability - EPYC . . . 24
2.3.2 Core Scalability - THIN . . . 26
2.3.3 Size Scalability - EPYC . . . 28
2.3.4 Size Scalability - THIN . . . 29
2.4 Conclusions . . . 30

1 Exercise 1

1.1 Introduction

The goal of this exercise is to implement in `C` a parallel version of a variant of the famous Conway's 'Game of Life' [1], It is a cellular automaton that operates autonomously on an infinite grid.
The game evolves on a 2D discrete world, where the tiniest position is a single cell; actually you can imagine it as a point on a system of integer coordinates. The neighbours of a cell are the 8 most adjacent cells, i.e. those that on a grid representation share an edge with the considered cell, as depicted in the Fig. 1 below.

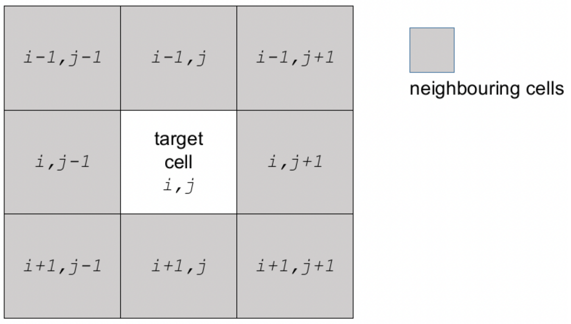


Figure 1: The cell's neighbours are the 8 immediately adjacent cells

The playground, that will be a grid of size $k \times k$, has periodic boundary conditions at the edges as seen in Fig. 2. It means that cells at an edge have to be considered neighbours of the cells at the opposite edge along the same axis. For instance, cell $(k-1)j$ will have cells $(0, j-1), (0, j)$ and $(0, j+1)$ as neighbours.

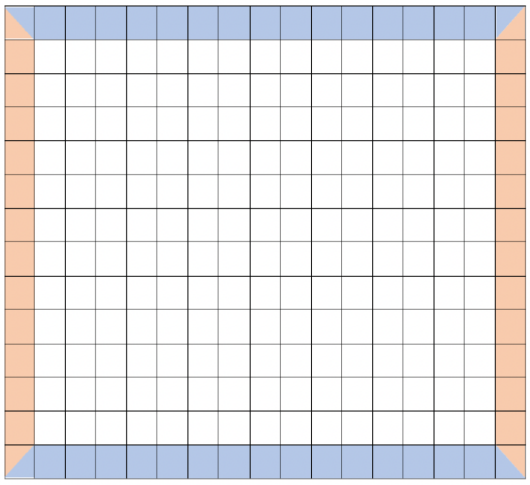


Figure 2: Periodic boundary conditions: the cells at the edges with the same colour are adjacent.

Each cell can be either "alive" or "dead" depending on the conditions of the neighboring cells:
A cell becomes, or remains, alive if 2 to 3 cells in its neighborhood are alive;

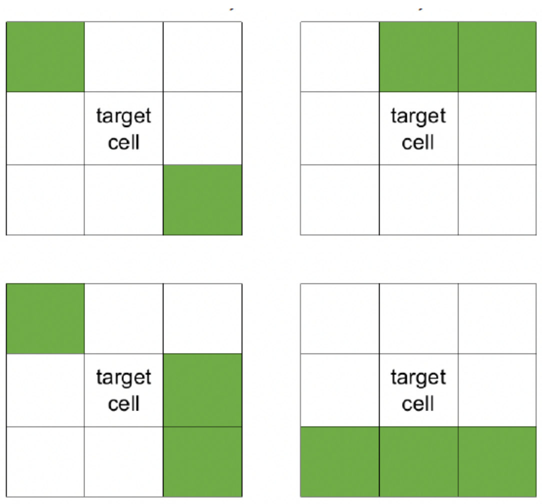


Figure 3: Examples for a cell to become or to remain alive

A cell dies, or do not generate new life, if either less than 2 cells or more than 3 cells in its neighborhood are alive (under-population or over-population conditions, respectively).

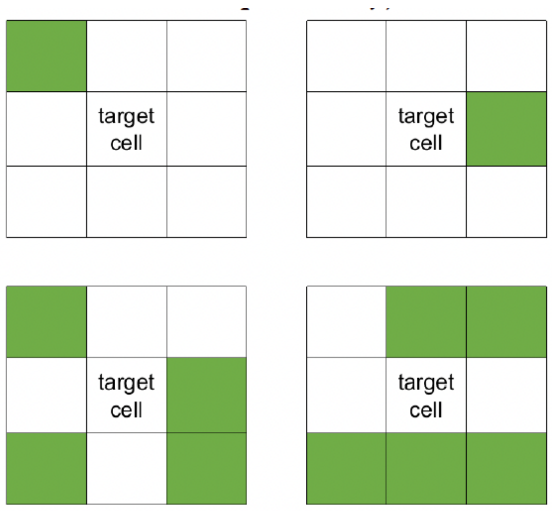


Figure 4: Examples for a cell going to die

The cell are updated following two different methods described in the next section.

The aim of this exercise is to implement a parallel version of the game using a hybrid approach of OpenMP and MPI, and analyze the scalability of each of the different evolution methods.

Three different scalability studies are considered:

- **OpenMP scalability**: see execution time for increasing number of threads
- **Strong MPI scalability**: see execution time for increasing number of MPI tasks.
- **Weak MPI scalability**: while keeping the workload per process fixed, see the execution time as the number of processes increases.

The code is implemented in order to meet the requirements provided in the assignment.

1.2 Methodology

To update the cell I'm going to implement two different evolution methods:

- **Ordered method**: Cells are updated "on the fly", meaning that evaluation and upgrade of a cell's status are done immediately, starting from cell $(0,0)$ and proceeding by line.
- **Static method**: We freeze the system, evaluate and store the status of each cell in an auxiliary matrix, and update the whole grid only afterward.

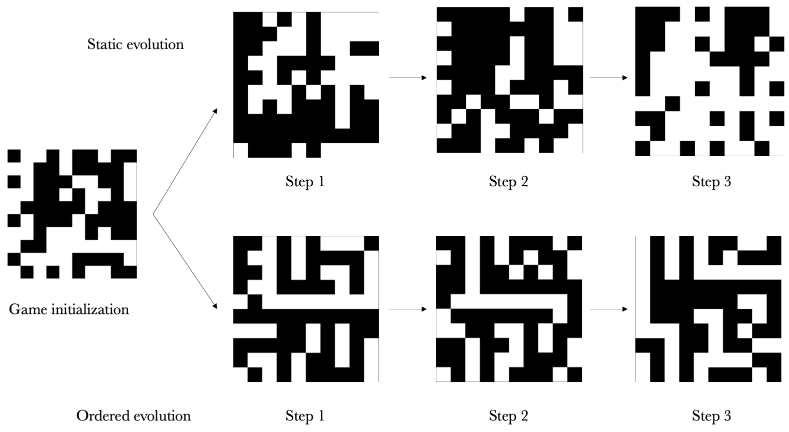


Figure 5: 3 steps for the two types of evolution on a 10×10 grid

The program start with the initialization of the playground of dimension $k \times k$ through a suitable routine.
There are two implementation in the source code: one for a serial approach that it's used when there is only 1 MPI process `initialize_serial()`; and another one that parallelized the serial code through a domain decomposition of the grid `initialize_parallel()`.
After the initialization, the grid must be saved in a `file.pgm` through the routine `write_pgm_image()` provided by the professors which works in a serial way. Then there is the evolution of the grid for a number n of steps with one of the two evolution methods: For each is implemented a serial and parallelized version:

- `ordered_ev_parallel()`
- `ordered_ev_serial()`

- `static_ev_parallel()`
- `static_ev_parallel()`

The serial code is used, as for the initialize, when there is only 1 MPI process.
These functions begin with the read of the grid from `file.pgm` with a provided routine `read_pgm_imgage()`. Next, the image is processed by the selected evolutionary algorithm.
The parallelized code use a domain decomposition of the grid and consequently I have to take care of the boundary rows to ensure the periodic boundary condition: For example consider a grid of dimension of 5×5 and 3 MPI process as in Fig. 6.

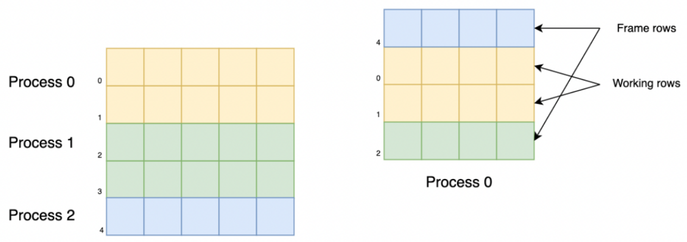


Figure 6: MPI domani decomposition

Each process could have $r = 1, 2$ rows to update, but each row needs the near rows to count the value of the neighbours, so each process will have a number of row $r + 2$. This because the evolution is rows-wise so to ensure the periodic condition for the column at the edge we have just to watch at the element of the other side of the row.

The program must have the ability to save an image every s steps, where s is provided by the user at runtime. Also the dimension of the grid k and the number of steps n can be set up at runtime. The program than is compiled and ran on ORFEO through suitable bash file. The results are saved on `.csv` files. I ran the program only on the THIN nodes because the EPYC nodes were not performing well during my experiments and ORTE daemons errors kept appearing.

1.3 Implementation

To implement the program I divided the code in different files:

- `initialize.c`: It has the routine to initialize the playground.
- `read_write.c`: It has the routine to read and write images in `.pgm` format.
- `ordered_evolution.c`: It has the routine to implement the ordered evolution.
- `static_evolution.c`: It has the routine to implement the static evolution.
- `main.c`: The main file used to run the game

The first 4 files have also an associated header file, in this way the routines can be imported in the main file.

1.3.1 Read and Write

The routines to write the images from an array to a `.pgm` file and read an images from `.pgm` file and save it in an array are provided by the professors. They work in a serial way so they must be called from the root processor in a MPI environment.

1.3.2 Initialize

To initialize the playground in a parallelized version as first thing I have to declare the needed variables like the general grid using as data type the `unsigned char` and the local grid used by each MPI processor. To understand how many rows each processor has to generate and how much memory has to allocate I simply divided the size k for the number of processor n_p.

```
unsigned char *local_grid = NULL;
unsigned char* gathered_grid = NULL;

int rows_read = k / size;
int extra_rows = k%size;
rows_read = (rank < k % size) ? rows_read+1 : rows_read;

local_grid=(unsigned char*)malloc(rows_read*k*sizeof(unsigned char));
```

After the initialization of the variables, the local gird are filled with random values between 0 or 255 through a for loop in a OpenMP environment to parallelized the process. This two values represent the cell alive or dead and they are used then to write (read) the image to (from) `.pgm` file.

```
#pragma omp parallel for
for(long i=0; i<rows_read*k; i++){
        int random_number = (rand() % (maxval+1));
        local_grid[i]= (random_number > half)?_maxval:minval;
}
```

When all the local grid are filled then all the data are gathered in the root process in the general grid. Before sending the local grid to the root processor it's necessary that the root process knows how many items receives from each processors and the location in the general grid.

```
int* list_rows_proc = NULL;
if (rank == 0) {
        gathered_grid = (unsigned char*)malloc(k*k *
        sizeof(unsigned char));
        list_rows_proc = (int*) malloc(size * sizeof(int));
}
MPI_Gather(&rows_read, 1, MPI_INT, list_rows_proc, 1, MPI_INT,
0, MPI_COMM_WORLD);

int sendcounts[size];
int displacements[size];
int total_rows = 0;

if (rank == 0){
    for (long i = 0; i < size; i++) {
        sendcounts[i] = list_rows_proc[i] * k;
        displacements[i] = total_rows * k;
        total_rows += list_rows_proc[i];
    }
}
MPI_Bcast(sendcounts, size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(displacements, size, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
```

To gather the data I use the MPI routine `MPI_Gatherv()` that allows the different processor to send data to the root processor with different dimensions. This choice is justified by the fact that the local grid could manage different number of rows.

```
MPI_Gatherv(local_grid, rows_read * k, MPI_UNSIGNED_CHAR,
gathered_grid, sendcounts, displacements, MPI_UNSIGNED_CHAR,
0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
free(local_grid);
```

After the grid is gathered in the root processor it is written in a `.pgm` file.

```
if (rank == 0) {
    char file_path[45] = "images/";
    strcat(file_path, fname);
    write_pgm_image((void *) gathered_grid, maxval, k, k, file_path);
    free(gathered_grid);
}
```

1.3.3 Static evolution

The function that implements the parallelized version of the static evolution starts with the declaration of the requested variables and the reading of the image and store of it in an array in the root processor

```
if(rank==0){
    start_time = MPI_Wtime();
    char file_path_r[45] = "images/";
    strcat(file_path_r, filename);
    completeMatrix = (unsigned char*)malloc(k*k *
    sizeof(unsigned char));
    read_pgm_image((void**)&completeMatrix, &maxval, &k,
    &k, file_path_r);
}
```

In the previous snippet code there is also the command `start_time = MPI_Wtime();` useful for counting the passing time. I will talk about how the time is measured later.
After the grid is stored on an array I have to spread the pieces of the matrix to the different MPI processes. To do this before I need to know how many rows each processor can manipulate and I do this in the same way for the initialize function. I need to be particularly careful at this point because, in addition to the actual rows that the processor has to evolve, you also need to send it

the auxiliary rows for the periodic boundary conditions. The root processor copies from the general array to its local array the data and auxiliary rows. Then it sends the data and auxiliary rows to the other processor with particular attention to the last processor. Next each processors must be receive the own local array with the auxiliary rows. The following snippet code shows the procedure I follow.

```
local_array = (unsigned char*)malloc((rows_read+2) * k *
sizeof(unsigned char)); // allocate memory on each processor
int displ = 0;
if (rank == 0) {
    int count_row = 0;
    // Processor 0 receives the last row + its rows_read rows + 1 row

    for (int i = 0; i < k; i++) {
        local_array[i] = completeMatrix[(k - 1) * k + i]; // Last row
    }
    for (int i = 0; i < (rows_read + 1) * k; i++) {
        local_array[k + i] = completeMatrix[i]; // rows_read + 1 rows
    }
    count_row = rows_read;

    // Send to other processors
    for (int p = 1; p < size - 1; p++) {
        displ = (count_row - 1) * k; // Start from 1 row back
        MPI_Send(completeMatrix + displ, (list_rows_proc[p] + 2) * k,
        MPI_UNSIGNED_CHAR, p, 0, MPI_COMM_WORLD);
        count_row += list_rows_proc[p];
    }

    // Last processor
    displ = (count_row-1) * k; // 2 rows back for the last processor
    MPI_Send(completeMatrix + displ, (list_rows_proc[size-1]+1) * k,
    MPI_UNSIGNED_CHAR, size - 1, 0, MPI_COMM_WORLD);
    MPI_Send(completeMatrix, k, MPI_UNSIGNED_CHAR, size - 1, 1,
    MPI_COMM_WORLD); // Send first row

} else if (rank == size - 1) {
    // Last processor receives (rows_read + 1) + first row
    MPI_Recv(local_array, (rows_read+1) * k, MPI_UNSIGNED_CHAR, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(local_array + (rows_read+1) * k, k, MPI_UNSIGNED_CHAR,
    0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

} else {
    // Intermediate processors receive rows_read + 2 rows
    MPI_Recv(local_array, (rows_read + 2) * k, MPI_UNSIGNED_CHAR,
    0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Now everything is ready to start with the evolution of the grid and play the game. In this method I need an auxiliary array of the same dimension of the `local_array`. I call it `new_local_array`. It's necessary to update the cell following the stating method. To update the cells, I implement a specific function that handles this. In an OpenMP environment, the number of neighbors for each cell is counted. If a cell has exactly 2 or 3 alive neighbors, the cell either stays alive or becomes alive. Otherwise, the cell dies. The new states are stored on the `new_local_array`.

```
#pragma omp parallel for
for (int i = k; i<(rows_read+1)*k; i++){
    int count = 0;
    int row, column;

    row = i/k;
    column = i%k;

    // Periodic boundary conditions for rows
    int row_above = row - 1;
    int row_below = row + 1;

    // Periodic boundary conditions for columns
    int col_left = (column == 0) ? k-1 : column - 1;
    int col_right = (column == (k-1)) ? 0 : column + 1;

    // Sum of neighbors (up, middle, down)
    int up = local_array[row_above * k + col_left] +
             local_array[row_above * k + column] +
             local_array[row_above * k + col_right];

    int middle = local_array[row * k + col_left] +
                 local_array[row * k + col_right];

    int down = local_array[row_below * k + col_left] +
               local_array[row_below * k + column] +
               local_array[row_below * k + col_right];

    count= up + down + middle;

    new_local_array[i] = (count == 765 || count == 510) ? 255 : 0;
}
```

Since the evolution has to be done for a number n of iteration than the update function is placed inside a for loop. After the update of the grid through the `update_cell()` it's necessary to update the auxiliary rows and the data rows in the `local_grid` before the next iteration. To update the auxiliary rows I use MPI non blocking commands to avoid deadlocks behaviour during the sending and receiving between the processor of the updated rows. To update the `local_grid` I simply swap the pointer from the new and old array. This way is faster than copy the data from one array to another. So the process of update is ended and it's possible to go to the next iteration of the for loop.

```
new_local_array = (unsigned char*)malloc((rows_read+2)
* k * sizeof(unsigned char));

for(int i=1; i <=t; i++){
    update_cell(local_array, new_local_array, k, rows_read);
    MPI_Request request[4];
    if(rank == 0){
        MPI_Isend(new_local_array+k, k,
        MPI_UNSIGNED_CHAR, size-1, rank + i + 1,
        MPI_COMM_WORLD, &request[0]);

        MPI_Isend(new_local_array + rows_read*k, k,
        MPI_UNSIGNED_CHAR, rank+1, rank + i,
        MPI_COMM_WORLD, &request[1]);

        MPI_Irecv(new_local_array + k + rows_read*k, k,
        MPI_UNSIGNED_CHAR, rank+1, rank+1 + i,
        MPI_COMM_WORLD, &request[2]);

        MPI_Irecv(new_local_array, k,
        MPI_UNSIGNED_CHAR, size-1, size-1 + i + 1,
        MPI_COMM_WORLD, &request[3]);
    }else if(rank == size-1){
        MPI_Isend(new_local_array+k, k,
        MPI_UNSIGNED_CHAR, rank-1, rank + i,
        MPI_COMM_WORLD, &request[0]);

        MPI_Isend(new_local_array + rows_read*k, k,
        MPI_UNSIGNED_CHAR, 0, rank + i +1 ,
        MPI_COMM_WORLD, &request[1]);

        MPI_Irecv(new_local_array + k + rows_read*k, k,
        MPI_UNSIGNED_CHAR, 0, 0+i+1,
        MPI_COMM_WORLD, &request[2]);

        MPI_Irecv(new_local_array, k,
        MPI_UNSIGNED_CHAR, rank-1, rank-1 + i,
        MPI_COMM_WORLD, &request[3]);
    }else{
        MPI_Isend(new_local_array+k, k,
        MPI_UNSIGNED_CHAR, rank-1, rank + i,
        MPI_COMM_WORLD, &request[0]);

        MPI_Isend(new_local_array + rows_read*k, k,
        MPI_UNSIGNED_CHAR, rank+1, rank + i,
        MPI_COMM_WORLD, &request[1]);
        MPI_Irecv(new_local_array, k,
        MPI_UNSIGNED_CHAR, rank-1, rank-1 + i,
        MPI_COMM_WORLD, &request[2]);

        MPI_Irecv(new_local_array + k + rows_read*k, k,
        MPI_UNSIGNED_CHAR, rank+1, rank+1+i,
        MPI_COMM_WORLD, &request[3]);
    }
    MPI_Waitall(4, request, MPI_STATUS_IGNORE);

    unsigned char *temp = new_local_array;
    new_local_array = local_array;
    local_array = temp;

    if((s!=0)&&(i%s==0)){
            write_array(local_array, i, k, maxval,
            rank, size, rows_read, list_rows_proc);
    }
}
```

As required, at the end of the loop, there is an `if` statement that is triggered if the user wants to save a snapshot of the game. To do this, I implemented a specific function that writes the array to a `.pgm` file. This function works in the same way as when the image is written after the initialization of the game.
The function for the evolution ends with the save of the final images, the release of the allocated memory and the command `end_time = MPI_Wtime();` to end the count of the time. There is also a `printf()` to print the result of the timing.

```
if (s==0){
    write_array(local_array, t, k, maxval, rank,
    size, rows_read, list_rows_proc);
}
free(new_local_array);
free(local_array);
if (rank ==0){
    free(completeMatrix);
    end_time = MPI_Wtime();
    double elapsed_time = end_time - start_time;

    printf(STATIC_PARALLEL: Dimension: %ld,\t
    Size: %d,\t Threads: %d,\t Time: %f\n ,k,size,
    omp_get_max_threads(),elapsed_time);
}
```

I implement also a serial version of the static evolution where the idea is to evolve the cell are equal to above. The unique difference is that everything is done with only one process because, of course, it's serial.

1.3.4 Ordered evolution

The ordered evolution function follows the same idea of the static evolution one:

- The grid is read from file.
- The workload is spread among the processors.
- The evolution start and goes on for n steps.
- Inside the for loop the cell is updated with a suitable function according to the ordered evolution.
- The auxiliary rows are exchanged.
- If request a snap is save at each s steps
- The allocated memory is released and is printed the final results

The code is very similar to that for the static evolution, except for the part that updates the cells and exchanges the auxiliary rows. Let's have a look. The function for the update of the cell is similar to the static evolution, but since this method is intrinsically serial it does not make any sense of use a OpenMP region here. The update is done directly on the `local_array` so there is no need of an auxiliary array. Also the part of the exchange of the auxiliary rows follow a serial setup: after one process update its local array sends the last row to the next process and the first row to the previous process. We will see from the experimental result that this method does not scale with an increase of the numper of MPI processes.

```
for (int i = 1; i <= t; i++) {
    for (int p = 0; p < size; p++){
        if (rank == p) {
            update_cell_ordered(local_array ,
            k, rows_read , p);

            int send_to = (rank + 1) % size;
            MPI_Send(local_array + rows_read * k, k,
            MPI_UNSIGNED_CHAR , send_to , 0, MPI_COMM_WORLD);
        }

        if (rank == (p + 1) % size) {
            int recv_from = (rank - 1 + size) % size;
            MPI_Recv(local_array , k, MPI_UNSIGNED_CHAR ,
            recv_from , 0, MPI_COMM_WORLD , MPI_STATUS_IGNORE);
        }
        MPI_Barrier(MPI_COMM_WORLD);

        if (rank == p) {
            int send_to = (rank - 1 + size) % size;
            MPI_Send(local_array + k, k, MPI_UNSIGNED_CHAR ,
            send_to , 0, MPI_COMM_WORLD);
        }
        if (rank == (p - 1 + size) % size) {
            int recv_from = (rank + 1 + size) % size;
            MPI_Recv(local_array + (rows_read + 1) * k, k,
            MPI_UNSIGNED_CHAR, recv_from, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    if((s!=0)&&(i%s==0)){
        write_array_ordered(local_array, i, k, maxval,
        rank, size, rows_read, list_rows_proc);
    }
}
```

Also for this method is implemented a complete serial function that works only in case of 1 MPI process.

1.3.5 Main

The main function is the core of the program. At the begin some default values for the variables are given like the dimension $k = 100$ of the grid, the number of iteration to do $n = 50$ or if and how many times save a snap of the game with te varibles $s = 0$. Also a name of default for the saved initialize image is given.The user can choose their own name for the saved images. At runtime, the user can also decide which type of evolution they want to perform. All of these runtime options available to the user are implemented in the main function using a snippet provided by the professors.
In the main function, the first step is to initialize the MPI environment by declaring how many processes there are and determining the rank of each processor. The MPI environment is initialized with a specific threading support level, `MPI_THREAD_FUNNELED`. It allows the MPI library to be used in conjunction with multiple threads, and it also tells the library how you intend to use MPI with threads.

```
MPI_Init_thread( &argc, &argv, MPI_THREAD_FUNNELED,
&mpi_provided_thread_level);
if ( mpi_provided_thread_level < MPI_THREAD_FUNNELED ) {
    printf(a problem arise when asking for
    MPI_THREAD_FUNNELED level\n);
    MPI_Finalize();
    exit( 1 );
}
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

After the MPI environment is initialized, there are two if statements: the first allows the user, via a runtime command, to decide whether to initialize the game or run it. The second if statement lets the user choose which type of evolution to perform if they decide to run the game.

```
if (action == INIT){
    if(fname==NULL){
        printf(No name was passed, default name will be used\n);
        fname=(char*)malloc(sizeof(fname_deflt)+1);
        sprintf(fname, "%s", fname_deflt);
    }
    if (size == 1){
        initialize_serial(k,fname,maxval);
    }else{
        initialize_parallel(k, fname, rank, size, maxval);
    }
}
if (action == RUN){
    if (fname == NULL) {
        fname = (char*) malloc(sizeof(fname_deflt));
        printf(fname, "%s", fname_deflt);
    }
    //run the ordered evolution
    if(e==0){
        if(size==1){
            ordered_ev_serial(fname, k, maxval, s, n);
            printf(ordered_ev_serial DONE\n);
        } else {
            ordered_ev_parallel(fname, rank, size, k, maxval, s, n);
        }
    //run the static evolution
    }else if(e==1){
        if(size==1){
            static_ev_serial(fname, k, maxval, s, n);
        } else {
            static_ev_parallel(fname, rank, size, k, maxval, s, n);
        }
    } else {
        printf(Invalid input: e can only take value in {0,1}\n);
        MPI_Finalize();
        exit( 1 );
    }
}
if (fname != NULL)
    free (fname);
MPI_Finalize();
```

Of course if there are only 1 MPI process the serial function are used. The main function end with the closing of the MPI environment at the release of memory for the `filename`

1.3.6 Run the game

To run the experiments to test the different types of scalability I made different bash files: one for each scalability test and different type of evolution algorithm. Each bash files start with some SLURM directives to allocate the necessary hardware resources. As said before all the experiment are done on THIN node. To test OpenMP scalability are used 2 nodes and for MPI 4 nodes. I set 2 tasks for node for all the experiments because there are 2 socket on each node so on each socket run 1 MPI process. The MPI tasks pool are mapped onto the hardware using the command

`--map-by node` and `--bind-to socket`. Separating the processes on different sockets ensures that each process can have up to 12 cores and exactly one is used by each thread. Concerning the binding policy for the OpenMP threads, I ran everything by using the default `OMP PLACES=cores` and opting for a `close` binding policy.
The time of running is measured, as said before, using the function `MPI_WTime()`. It gives the time relative to this reference point, so it's not an absolute timestamp but a measure of elapsed time.
For each of the three scalability experiments, different grid sizes k are used. Before each run the playground is initialized with the suitable size of the grid. Each run is repeated five times, and the mean and standard deviation are calculated as the final numerical values.
The results are stored in `.csv` files and then analyzed using Python.

1.4 Results and Discussion

In this section I'm going to analyze and comment the result obtained from the experiment for the two different evolution methods and for the three scalability studies:

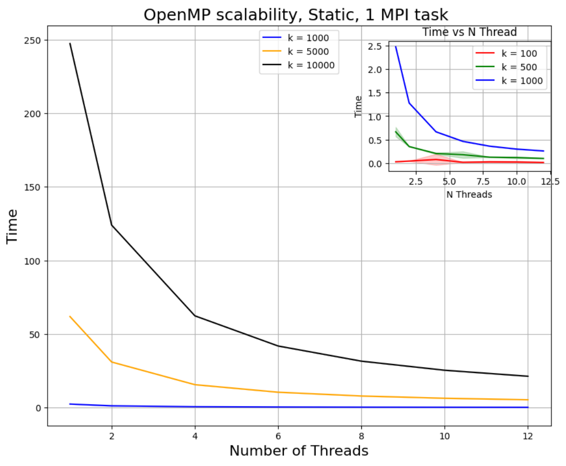
- OpenMP
- Strong MPI
- Weak MPI

1.4.1 OpenMP scalability

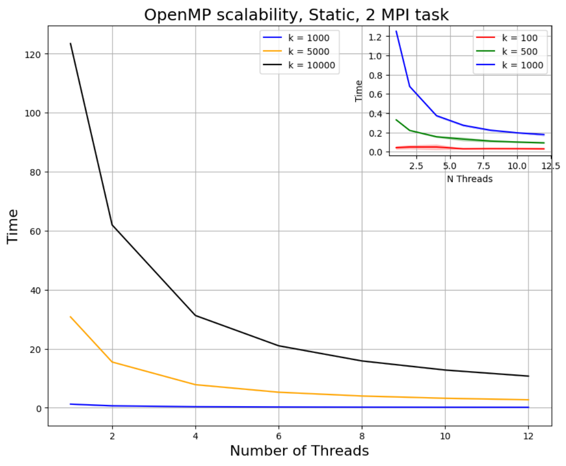
To study the OpenMP scalability the number of MPI task is fixed for the all experiment. I conducted two different experiments: one using a single MPI task and another using two MPI tasks. I also use different size dimension for the grid: $k = 100, 500, 1000, 5000, 10000$. For each of the grid dimension is measured the elapsed times for an evolution with $n = 50$ steps for both evolution methods. The number of threads goes from 1 to 12, the max number of core inside a socket for the THIN node.
From the Fig. 7 we can see that for high sizes $(5000, 10000)$ of the grid there is scaling of the elapsed time, while for little $(100, 500)$ dimension there is no evidence for a scaling.
This can be seen also from the Fig. 8 where it is reported the speedup. The speedup is defined as

$$speedup = \frac{< T_1 >}{< T_n >}, \tag{1}$$

where $< T_1 >$ is the average time for number of thread equal to 1 and $< T_n >$ is the average time for number of threads. The theoretical peak performance (TPP) is simply the line $y = x$ because we foresee that at each step the time scale down linearly with the respect of the number of thread.



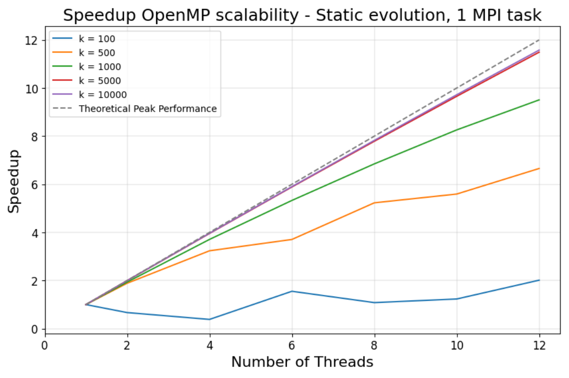
(a) Scaling with 1 MPI task



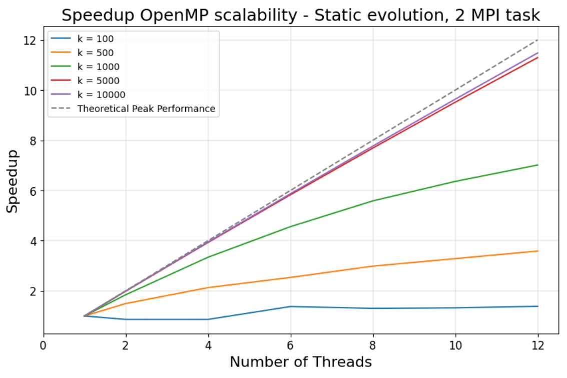
(b) Scaling with 2 MPI tasks

Figure 7: OpenMP scaling using static evolution

The better speedup is reached when the matrix have the highest dimension. For little size there is no evidence of speedup. Perhaps this is due to the fact that the matrices are so small that the execution time within the OpenMP environment is so low that the measured time is influenced by other factors, such as reading the matrix or, in the case of two MPI tasks, by communication, even though it occurs within the same node. Consequently, I believe that the method used to measure execution time is not entirely appropriate for this experiment. Measure only the OpenMP part could be a better solution. Nevertheless, it is noticeable that when the matrices are larger, the algorithm scales as the number of threads increases.
Overall, there is no substantial difference if it is used one or two MPI Task, except for the fact that, obviously, the total execution time is halved when two tasks are used, foreshadowing the results of the Strong MPI scalability as well.



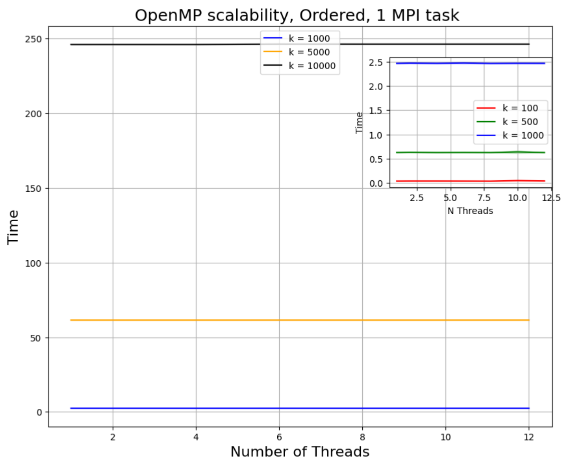
(a) Speedup with 1 MPI Task



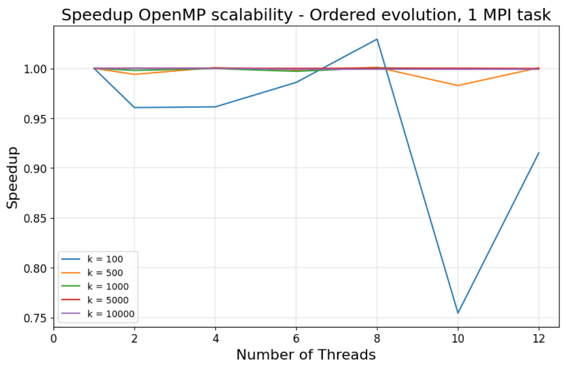
(b) Speedup with 2 MPI tasks

Figure 8: OpenMP speedup using static evolution

The ordered method respects the theoretical prediction. There is no scaling at all as we can see from Fig. 9a and Fig. 9b where is reported the 'speedup'. This is for the intrinsic nature of this evolution method. I believe there is potential to parallelize this method since, in the end, it is a deterministic evolution system where, given the initial conditions, the evolution is entirely determined by the dynamics.



(a) Scaling

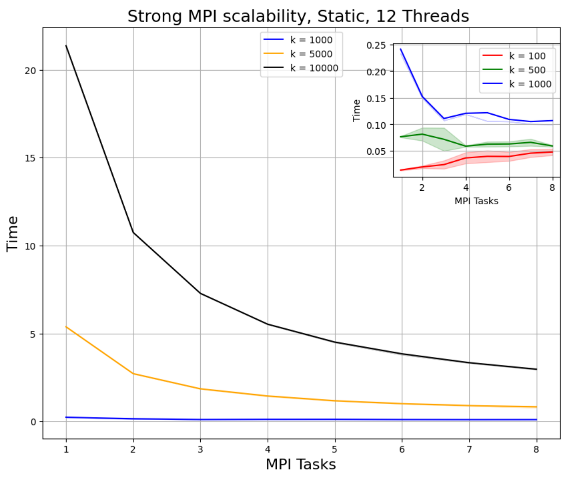


(b) Speedup

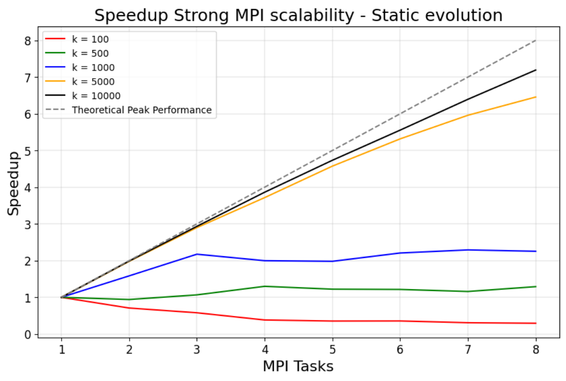
Figure 9: OpenMP scalability using ordered evolution

1.4.2 Strong MPI scalability

To study the strong MPI scalability I used up to 4 THIN nodes binding the MPI task on each socket. So I ran the experiment up to 8 MPI tasks for both evolution methods. Also here I try different matrix size as for OpenMP scalability. I fixed the number of threads equal to 12, the number of core per socket.
For the static evolution in Fig.10a we can see a scaling for high size of matrix as in the OpenMP experiments. While growing the number of MPI task for little size the time not decrease, but increase. This could be justify considering that the update of the grid is so fast for little dimension that the most of the time is spent for comunication between the MPI tasks. Indeed, the time grows with the number of MPI task. This behaviour is also reflected when is calculated the speedup. We can see that for large grid the speedup is close to the TTP while for small grid there is not at all speedup.



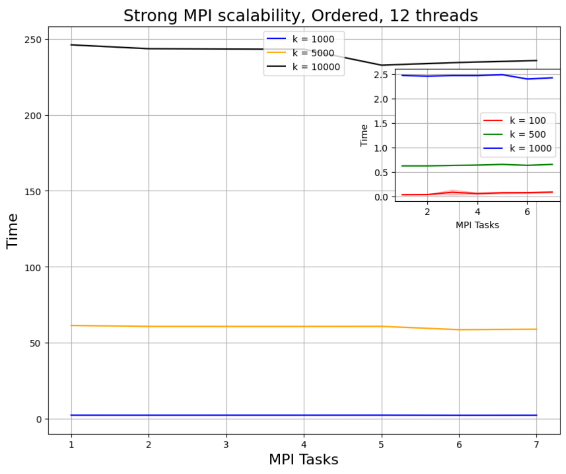
(a) Scaling



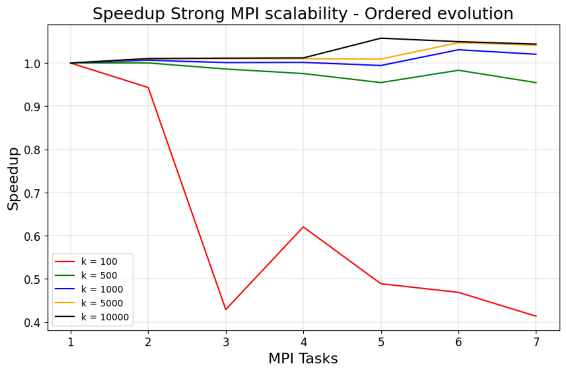
(b) Speedup

Figure 10: Strong MPI scalability using static evolution

Also for the Strong MPI scalability the oredered method in Fig.11a does not disappoint theoretical expectations. No scaling and not improvement. Also the speedup is flat as is possible to se from Fig. 11b.



(a) Scaling



(b) Speedup

Figure 11: Strong MPI scalability using ordered evolution

1.4.3 Weak MPI scalability

To study the Weak MPI scalability I have to fix the workload for each of the MPI tasks. I selected three initial size $k = 100, 1000, 10000$ for 1 MPI task and then the size k_i is increased for each tasks m_i keeping the workload constant through the relation

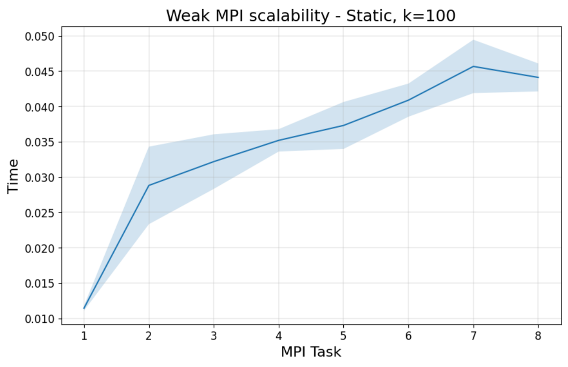
$$k_i = k_0\sqrt{m_i}. \tag{2}$$

In this way each MPI process has to update the same amount of grid cells. In table 1.

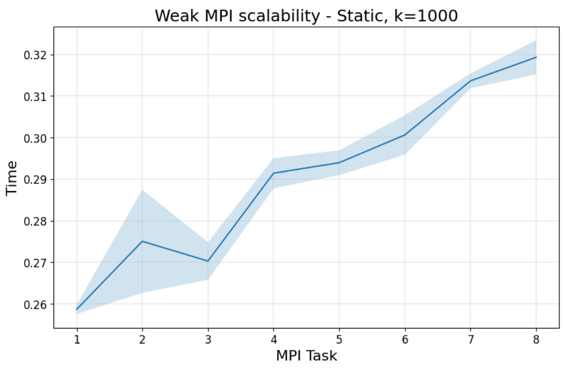
MPI task			
1	100	1000	10000
2	141	1414	14142
3	173	1732	17321
4	200	2000	20000
5	224	2236	22361
6	245	2449	24495
7	265	2646	26458
8	283	2828	28284

Table 1: Workload for MPI tasks in Weak MPI scalability

What someone will expect from this experiments is that the computational time is constant because the workload for each tasks is fixed. For the static method, from the experiment we have a slightly different behaviour. Also here as before, for $k = 100$ in Fig. 12a there is no scaling, the time grow with the number of MPI tasks. In Fig. 12b for $k = 1000$ things sound a bit better and for $k = 10000$ there is the best result as we can see from 13a. Even though the time increases slightly, the results are still fairly close to the theoretical predictions. We don't get a perfect result, of course, since with an increasing number of MPI tasks, there is more communication delay between processes. We can see the overall results in Fig. 13b where the theoretical behaviour should be a flat line along 1 (it is not reported in the plot).

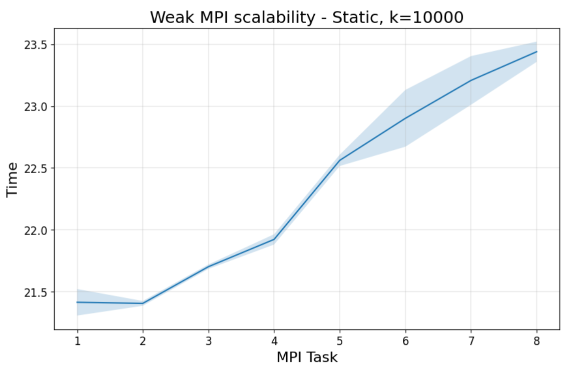


(a) Starting size $k = 100$

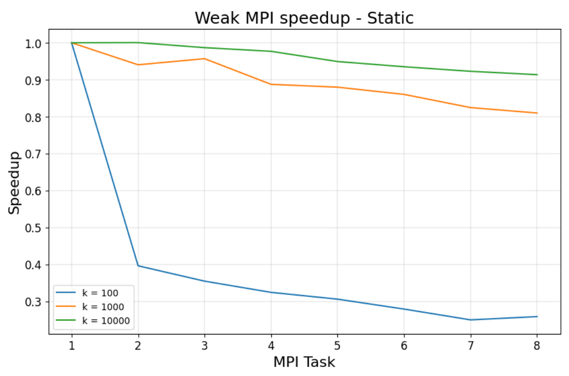


(b) Starting size $k = 100$

Figure 12: Weak MPI scalability using static evolution



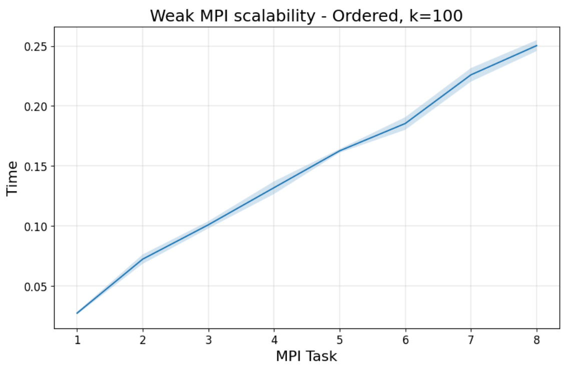
(a) Starting size $k = 100$



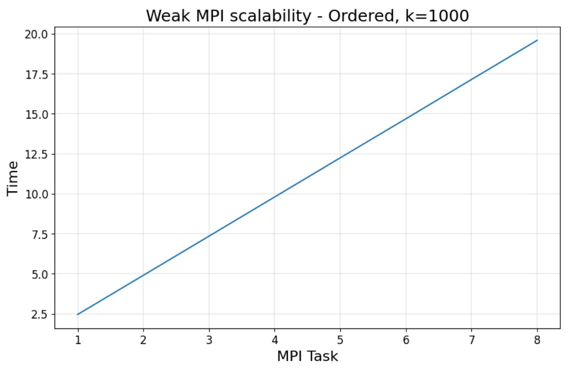
(b) Speedup analysis

Figure 13: Weak MPI scalability using static evolution

For the ordered method in Fig. 14, also here there is no hope of scalability. As the number of MPI tasks grow, the time increases almost perfectly linearly, indicating a lack of scalability.



(a) Starting size $k = 100$



(b) Starting size $k = 1000$

Figure 14: Weak MPI scalability using ordered evolution

The Fig. 15 confirm the linear behaviour of the Weak MPI scalability for the ordered method. Indeed the plots follow the the equation $f(x) = 1/x$.

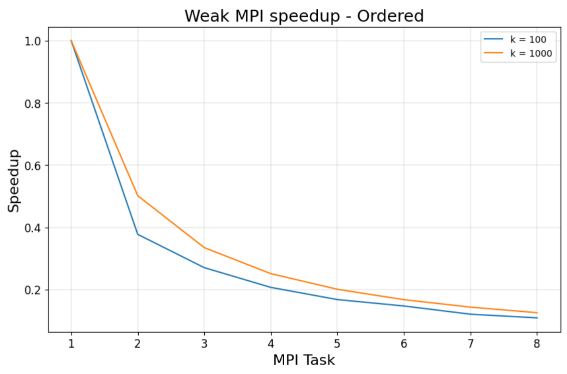


Figure 15: Speedup analysis for ordered method

1.5 Conclusions

In conclusion, I can say that the scalability study produced fairly positive results regarding the static method. All the experiments show that for larger matrices I obtain the best result, and for little matrices there is no a great advantage in the parallelization. The ordered method, as expected does not produce no one result that shows scaling behaviour due to its intrisical serial behavior.
I believe my algorithm is not one of the best implementation possible. I think it can be improved regarding the the MPI sending and receiving operations. I also think a better measure for the timing for the OpenMP scalability could be used. Other thread affinity policy and mapping of the MPI tasks could be used to see how the data change.
There was also a particular difficulty in working with ORFEO, as it often produced errors, especially with the EPYC nodes, which led me to exclude analyses using EPYC nodes.

2 Exercise 2

2.1 Introduction

The goal of this exercise was to compare the performance of three math libraries available on HPC: MKL [2], OpenBLAS [3] and BLIS [4]. The performance had to be evaluated focusing the comparison on the level 3 BLAS (Basic Linear Algebra Subprograms) function which is called gemm. Gemm stands for General Matrix Multiply, and it performs the following operation:

$$C = \alpha AB + \beta C \tag{3}$$

where α, β are scalars and C, A, B are matrices of size $(M \times N), (M \times K), (K \times N)$ respectively. Different versions of this function are implemented, one for double precision (*dgemm*) and one for single/float precision (*sgemm*).
The code for the exercise was provided by the professors and also the Makefile to compile the C code. The code which is used is a standard gemm code where three matrices A, B, C are allocated. In the code the three matrices are all square matrices. Moreover, the scalar parameter are set as $\alpha = 1$ and $\beta = 0$. Matrices A and B are filled and then the BLAS routine computes the matrix-matrix product as $C = A \times B$.
The exercise requires to perform:

- **Size scalability**: The size of the matrix with a fixed number of cores;
- **Core scalability**: The number of cores at fixed size of the matrix

The comparison of the three different libraries had to consider different variables:

- **Different architectures**: either THIN nodes or EPYC nodes;
- **Different thread allocation policies**: either spread or close cores were considered;
- **Different precision**: either single/float or double precision;

The cores are used by OpenMP threads to perform the operations in parallel.
OpenBLAS libraries was already available on ORFEO's module's list, while MKL and BLIS had to be downloaded and compiled.

2.2 Implementation

To gather data for the experiment, I wrote bash scripts as for the exercise 1. Each script follows the same structure:
At the beginning, SLURM directives are used to allocate and reserve resources for the job, and to choose the partition. Each job uses one node and one task per node, ensuring there is no connection delay between nodes.
Next, the code is compiled using a Makefile, and instructions about the working directories, OpenMP settings (such as thread policies like "spread" or "close"), and module loading are provided.
At the end, there are three nested loops that iterate through three variables: the libraries, data types (double and float), and matrix sizes or the number of cores, depending on what is being measured. Inside the loops, the executable is run, and the output is stored in a .txt file. Finally, the results are saved from the .txt file into a .csv file. All bash scripts used are available in the GitHub repository.
Performance was measured by tracking the elapsed time and the number of GFLOPs required to complete the GEMM operation with the specified settings.

2.3 Results and Discussion

The code provided allows to obtain and analyze both the time of execution and the GFLOPS achieved.
This latter measurement will also be compared with the **Theoretical Peak Performance (TPP)**: it's defined as the maximum number of floating point operation per second that can be performed on a given hardware platform. It's given by the relation

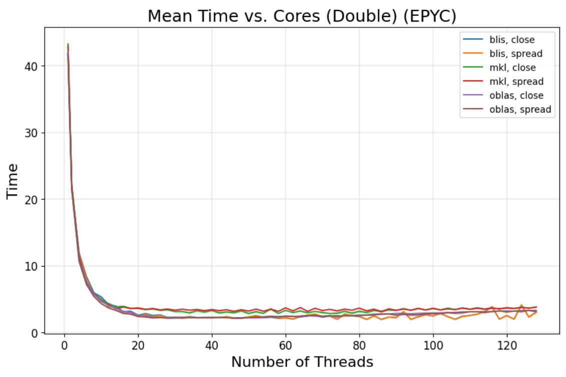
$$TPP = clock\ rate\ [GHz] \times \#cores \times \#FP\ operation\ per\ cycle. \quad (4)$$

To determine it we need to make some considerations about the two different architectures which are used. The Epyc nodes consist of two sockets each containing a *AMD Epyc 7H12 64-Core Processor*. By having a look at the specifics for the node one can determine that its clock rate is 2.6 *GHz* for each core [5].
The most modern Epyc nodes can do up to 16 FPs per cycle in double precision and 32 in single precision. As such to determine the TPP for the Epyc nodes when using n cores we can use these results to obtain $2.6 \times 32 \times n$ for single precision, and half that amount for double. Each Thin node consists of two sockets each containing a *Intel(R) Xeon(R) Gold 6126 CPU*. The theoretical peak performance for this can be found in the course slides. We are told that for a total node is 1.997 *TeraFLOPS* when using double precision, meaning that each core has a peak performance of 83.2 *GFLOPS*. As such to determine the TPP for the Thin nodes when using n cores we can use these results to obtain $2 \times 83.2 \times n$ for single precision, and half that amount for double.

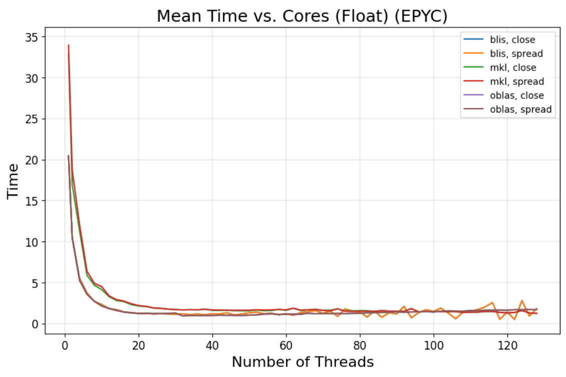
Time and GFLOPS measurement are taken 5 times to obtain an average and an estimate of thestandard deviation, which are reported in the plots. The time measurements are reported in seconds. On each of the following plots are reported the three different libraries and the thread policies used.

2.3.1 Core Scalability - EPYC

This section reports the plots for the cores scalability for EPYC node. It's used 1 node. The size of the matrix is fixed to 10000×10000 while increasing the number of cores from 1 up to 128.



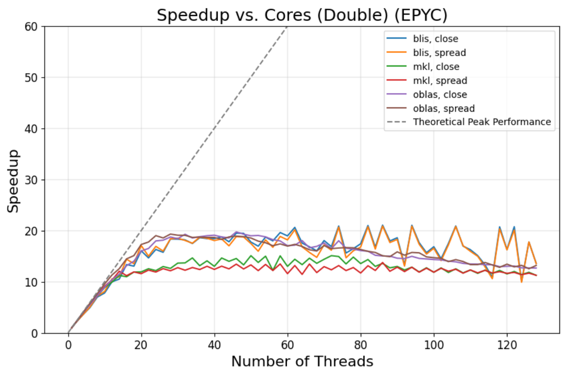
(a) Double precision



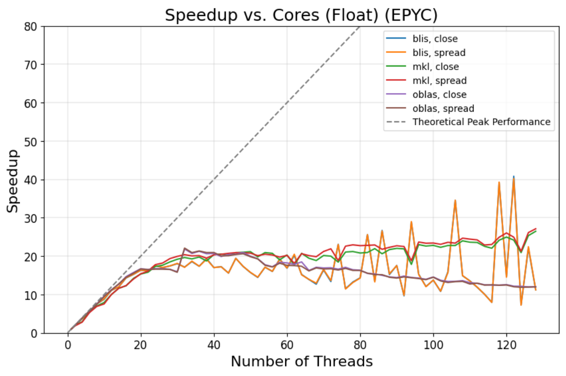
(b) Float precision

Figure 16: Time vs N of Threads for EPYC node

The plots 16 shows the mean execution time scaling with respect to the number of threads used for both double and single/float precision in EPYC node. We observe an exponential decrease in execution time for a small number of threads, but after around 20 threads, the scaling plateaus. MKL performs slightly better for double precision compared to the other two, but the advantage is not significant. Additionally, when examining the thread policies, we can see no noticeable differences. As expected, the overall execution time for single/float precision is shorter than for double precision.



(a) Double precision



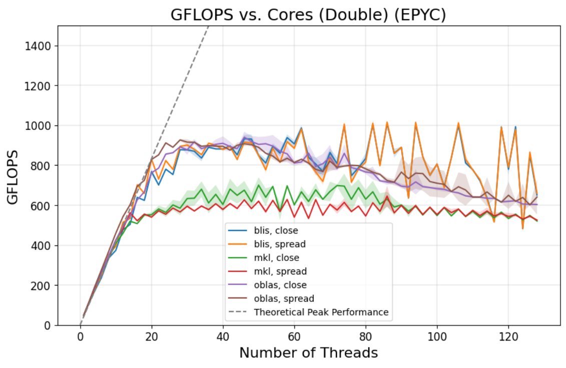
(b) Float precision

Figure 17: Speedup vs N of Threads for EPYC node

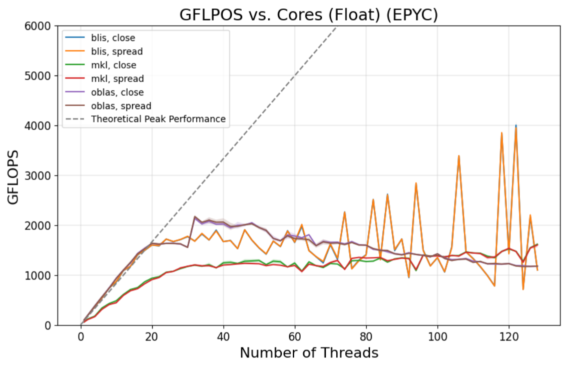
The plot 17 shows the speedup for different libraries and policy thread. The speedup is defined as

$$speedup = \frac{< T_1 >}{< T_n >}, \tag{5}$$

where $< T_1 >$ is the average time for number of thread equal to 1 and $< T_n >$ is the average time for number of threads. The theoretical peak performance is simply the line $y = x$ because we foresee that at each step the time scale down linearly with the respect of the number of thread.
We can see that this behaviour is respected more or less until 20 threads. This is in agreement with the time scale of the plot 16. After 20 threads from the experiment we see there is no more speedup, and in particular for both double and float precision we see that the BLIS library has a wide trend, while OpenBLAS and MKL are more regular.



(a) Double precision



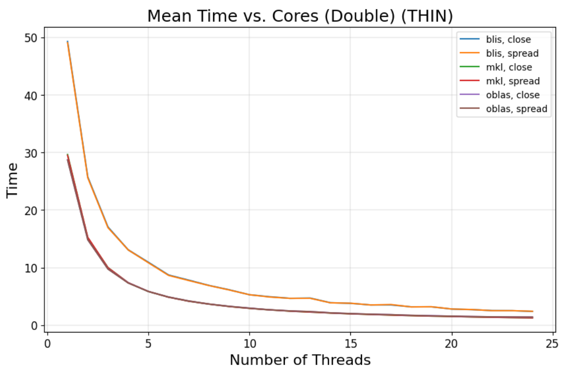
(b) Float precision

Figure 18: GFLOPS vs N of Threads for EPYC node

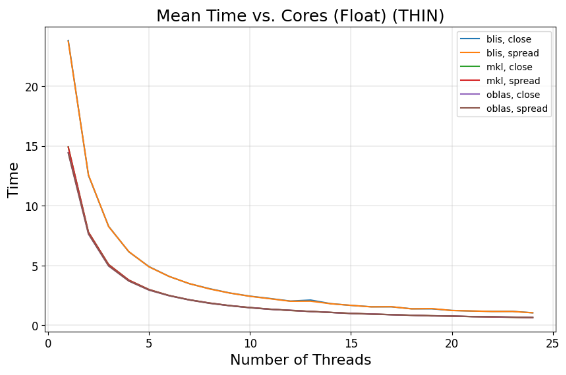
The plot 18 show the the behaviour of GFLOPS reached respect the number of Threads. Here as well the data from the experiment agree with the TPP until 20 threads and then distance with a flat behaviour according with the previous plot. Actually, we see that for single precision MKL never follow the TPP and it has the worst result compared with the other two libraries. Also in double precision MKL is the worst. And also here we see the wide trend of the BLIS library. There are not significant difference between the thread policy. The peak of GFLOPS is around 40 thread then it gets a little worse with the increasing value of number of core used.

2.3.2 Core Scalability - THIN

This section reports the plots for the cores scalability for THIN node. It's used 1 node. The size of the matrix is fixed to 10000×10000 while increasing the number of cores from 1 up to 24.



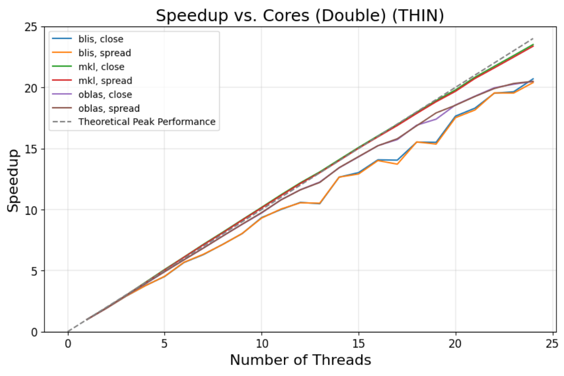
(a) Double precision



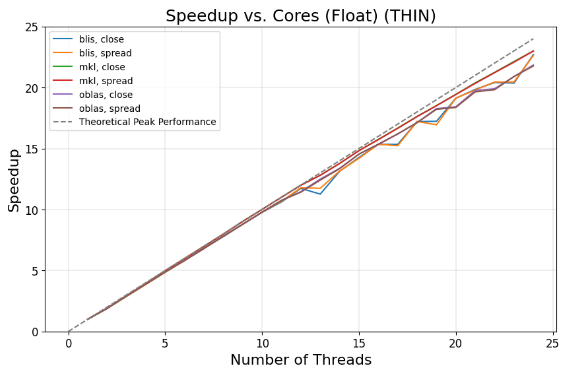
(b) Float precision

Figure 19: Time vs N of Threads for THIN node

The plots 19 shows the mean execution time scaling with respect to the number of threads used for both double and single/float precision in THIN node. As for EPYC node, we observe an exponential decrease in execution time. There is no a highlight plateau as in EPYC node. BLIS performs slightly better for double and float precision compared to the other two. Also here there is no difference in the thread policy used. As expected, the overall execution time for single/float precision is shorter than for double precision.



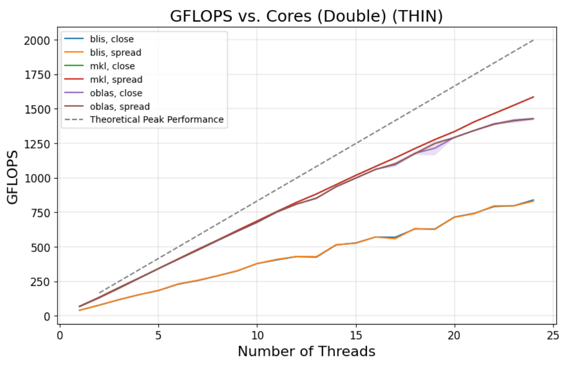
(a) Double precision



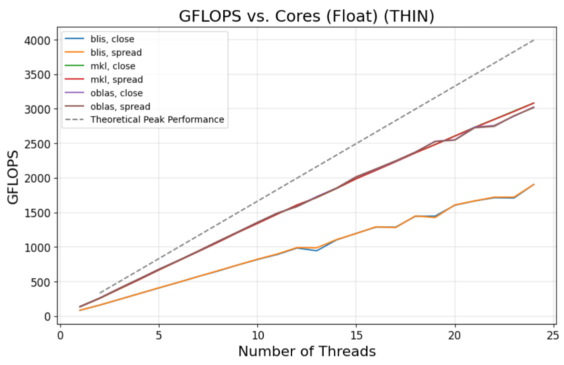
(b) Float precision

Figure 20: Speedup vs N of Threads for THIN node

The speedup behaviour for THIN node is different respect to the EPYC node. The experiment follow the TPP also for the max number of core used (but it's important to notice that here the max number is 24 respect to 128 of the EPYC node). MKL shows the best speedup performance, while OpenBLAS and BLIS perform less effectively.



(a) Double precision



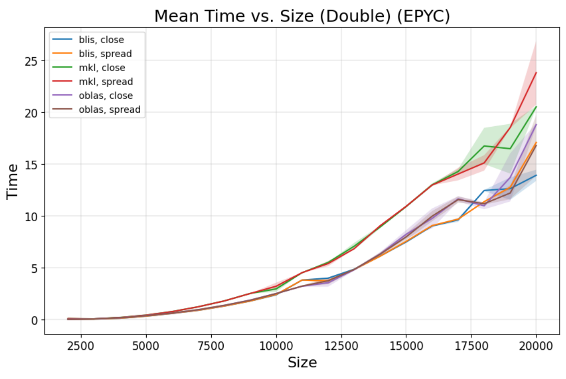
(b) Float precision

Figure 21: GFLOPS vs N of Threads for THIN node

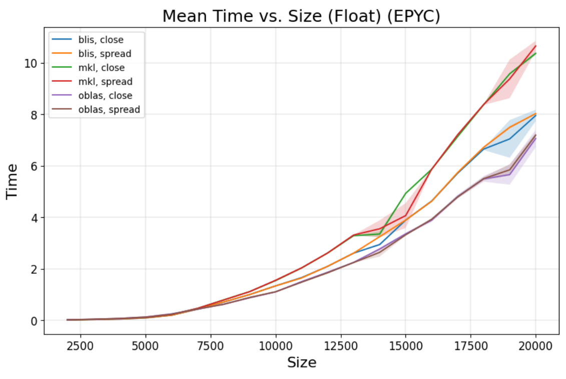
The plot 21 shows the the behaviour of GFLOPS reached respect the number of Threads for THIN node. The experimental data follows a linear trend but with a slightly lower slope compared to the theoretical peak performance. BLIS is the worst one, while MKL and OpenBLAS are closer. Also here there is no evidence in the difference of threads policy.

2.3.3 Size Scalability - EPYC

This section reports the plots for the size scalability for EPYC node. It's used 1 node with 64 core. The size of the matrix goes from 2000 × 2000 to 20000 × 20000 with a step of 1000.



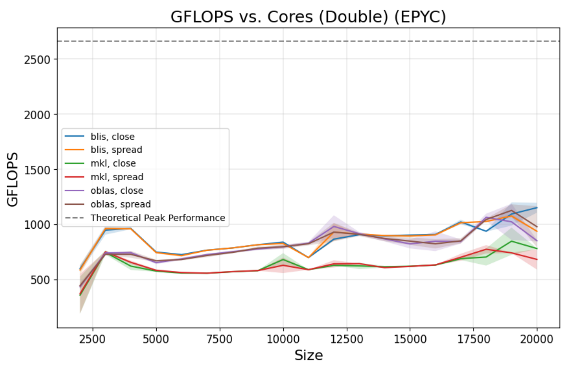
(a) Double precision



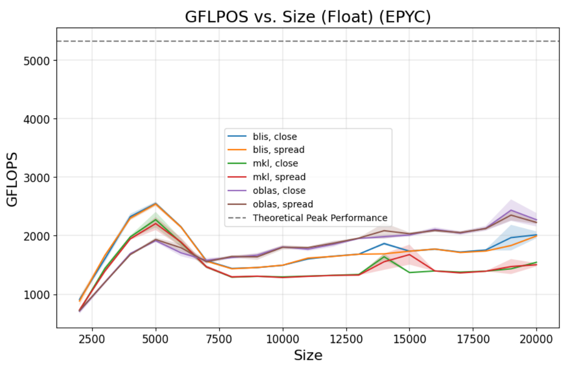
(b) Float precision

Figure 22: Time vs N of Threads for EPYC node

The plot 22 shows the execution time to respect the size of the matrix. The experimental data follow a exponential behaviour with the increasing size of the matrix. This is expected since the size of the matrix increases while maintaining constant the number of threads. The OpenBlas library that performs the best for float precisione, while for double precision also BLIS performs well. MKL is the worst one. Here we have a slightly difference for the two threads policy with the growing of the size of the matrices. Close is a bit better than spread for both precision.



(a) Double precision



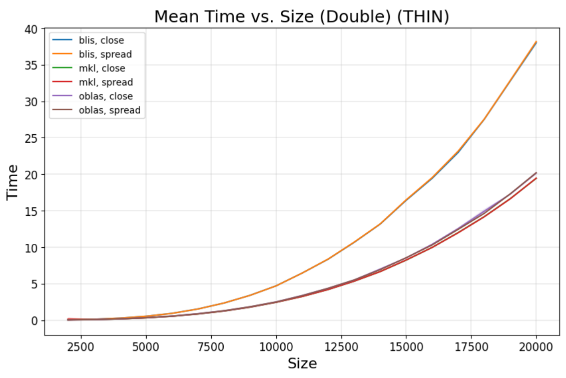
(b) Float precision

Figure 23: GFLOPS vs N of Threads for EPYC node

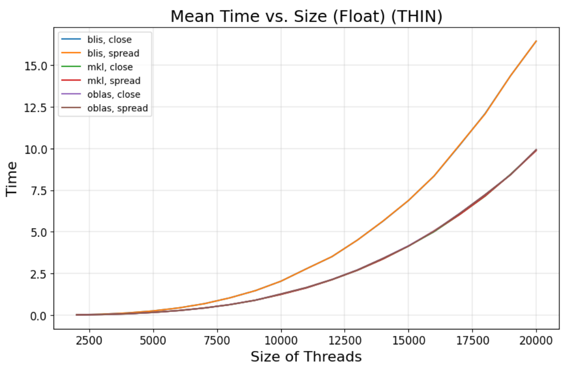
The plot 23 shows the number of GFLOPS in relation of the size of the matrices for the EPYC node. Both for double and single precision we have a globally flat behaviour with some threshold. This is good in according to the theory. It is worth nothing the distance with the theoretical peak. The experimental data are very below the theoretical value. It's constant because the number of cores is fixed. Also here OpenBLAS has the best values for float precision and for double is comparable with BLIS. Here is reflected an enhance in the peformance with close thread affinity policy respect to the spread one.

2.3.4 Size Scalability - THIN

This section reports the plots for the size scalability for THIN node. It's used 1 node with 12 core. The size of the matrix goes from 2000×2000 to 20000×20000 with a step of 1000.



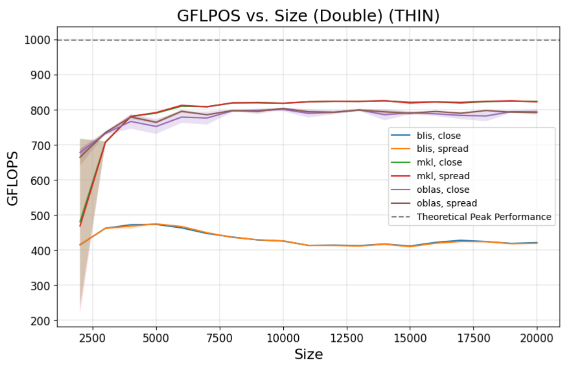
(a) Double precision



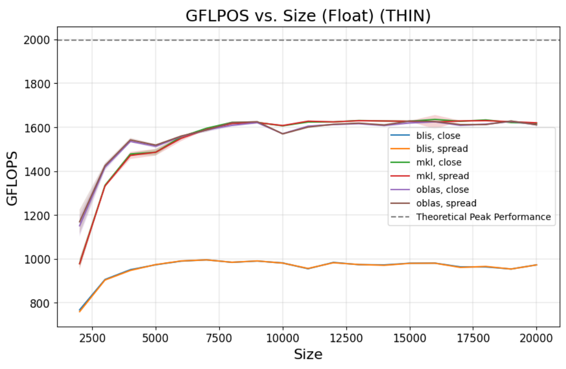
(b) Float precision

Figure 24: Time vs N of Threads for EPYC node

The plot 24 shows the execution time to respect the size of the matrix for THIN node. The behaviour, as for the EPYC node, is exponential, a bit smoother. Here the BLIS library thath performs worst, MKL and OpenBLAS are very close to each other. Here there is not difference for different thread affinity policy both for float and double precision.



(a) Double precision



(b) Float precision

Figure 25: GFLOPS vs N of Threads for EPYC node

The plot 25 shows the number of GFLOPS in relation of the size of the matrices for the THIN node. As for EPYC node, there is a plateau behaviour of the experimental data. For small matrices the values are low the platueau value that is reached for medium and large matrices. As for the running time the BLIS library is the worst one for both float and double precision. MKL and OpenBLAS are comparable. Overall, all three perform worse than the value given by the TPP. There are no noteable difference in the choice of the thread affinity policy.

2.4 Conclusions

I can conclude that among the libraries to perform the gemm operation there is no one which perform in absolut way better than the other three.
The performance depends from the chosen architecture. For example in Thin node BLIS is the worst one. In EPYC, OpenBLAS is a bit better but still comparable with BLIS, while MKL has the worst result on this architecture.
As expected the runtime for double precision is double both for core and size scalability respect the single/float precision.
For the thread allocation policies there is not characteristic difference from spread and close. Only in the analysis of size scalability is founded some significal difference for EPYC node where close policy is slightly better than the spread one.

References

[1] `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

[2] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html#gs.femdz1`

[3] `https://www.openblas.net`

[4] `https://github.com/flame/blis`

[5] `https://www.amd.com/en/products/specifications/server-processor.html`