Università degli Studi di Trieste

HPC Final Assignement

Submitted By :
Ippolito Noemi,
Pronestì Maria

Contents

Chapter 1

Exercise 1 - Parallel Game of Life

1.1 Introduction

The goal of the first proposed exercise is to implement a hybrid parallel version of Conway's Game of Life, using both MPI and OpenMP software, and to provide a performance analysis. Before going into details of our implementation, we provide a brief introduction to the game's rules.

Conway's Game of Life is a cellular automaton that is played on a 2D rectangular grid. Each square (or "cell") of the grid can be either alive or dead, and the evolution happens according to the following rules:

- Any living cell with either fewer than two living neighbors or more than three living neighbors dies, or if already dead, remains dead.
- Any living cell with either two or three living neighbors comes to life, or if already alive, lives to the next generation.

The grid has to be considered as an infinite playground, meaning that some boundary conditions for the first and last row and column have to be enforced when considering the neighbors of each cell.
Considering the aforementioned rules, the evolution of the playground is fully deterministic and it is determined exclusively by the initial conditions. For further information, please refer to [1].
Cells are updated in row-major order, as that is how matrices are stored in memory, according to different possible evolutions. In our specific case study, we considered two kinds of evolutions:

- **Ordered evolution**, in which 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 evolution**, in which we freeze the system, evaluate and store the status of each cell in an auxiliary matrix, and update the whole grid only afterward.

1.2 Methodology

The program is written in C language and parallel parts of the code are implemented in a hybrid fashion, so that communications among different processes are provided by MPI calls and each process is possibly further parallelized using OpenMP parallel regions. In particular, we chose a "funneled" mode, that allows us to perform MPI calls within the OpenMP parallel regions, but only by the thread 0 (also called *master thread*).
The playground is a general **k** x **k** grid, that is read from a **.pgm** file, in which living cells are colored blaick and dead cells are colored white.

When running the program through the terminal, the arguments passed to the main function decide which action should be performed. In particular, there are two possibilities:

- *Initialization*, performed by passing the **-i** command-line argument. It initializes a square random playground of size k, as specified by the **-k** argument and stores it in a **.pgm** file, whose name can be freely chosen by the user using the **-f** instruction. In case no name or size are provided, their default values will be used, which are *game_of_life.pgm* and *100*.
  The initialization can be implemented in a serial or parallel fashion. For the parallel version an hybrid MPI and OpenMP code was used. The version (serial or parallel) that has to run is chosen when the program is called based on the number of MPI processes required; if that number is equal to 1 then the serial code will be executed, otherwise the parallel code will be chosen.

- *Run* mode is achieved by passing the **-r** argument. It will execute the algorithm of the game as specified by the above rules, with initial conditions given by the **.pgm** file provided by the user through the **-f** directive, for a given number of iterations, as stated by the **-n** argument and print a snapshot at the end or at each s steps. The type of evolution to be performed can be chosen by the user among the two already presented above and is specified by the **-e** argument that can be either 0, for the *ordered* evolution or 1 for the *static* one.
  Both evolutions can be implemented in a serial or parallel fashion as in the case of *initialization.*

Given the intrinsic seriality of the ordered evolution, we expect that the parallel version will be as efficient as the serial one but the parallelization has been performed nonetheless to consolidate this theory.

Before measuring performances, the correctness of the code has been manually checked, on a small 10×10 matrix.

To measure the performance of the program a scalability study has been performed. In particular, we considered:

- OpenMP scalability
- Strong MPI scalability
- Weak MPI scalability

Several tests on the EPYC and THIN nodes of the Orfeo cluster have been performed, by properly changing the number of MPI processes and OpenMP threads. In particular, the study was focused on measuring the *Speedup*, which is the ratio between the time taken by a single process to run and the one taken by n processes.

1.3 Implementation

The following section provides some details about the most technical aspects of the implementation, mainly related to how the actual program works.
The code is made of 4 files:

- `Main.c`: It contains the functions needed to initialize the playground. It also contains the main function responsible for program execution, including the retrieval of input arguments and the invocation of appropriate functions based on these inputs
- `Static_evolution.c`: It contains the code that performs the static evolution, either in parallel or serially.
- `Ordered_evolution.c`: It contains the code that performs the ordered evolution, either in parallel or serially.
- `Read_write.c`: It contains the functions `read_pgm_image()` and `write_pgm_image()`, which are used to manage the reading and writing operations, respectively, on **.pgm** images.

Inside the `Main.c` file the headers of the other files were included in order to be able to call the different functions.

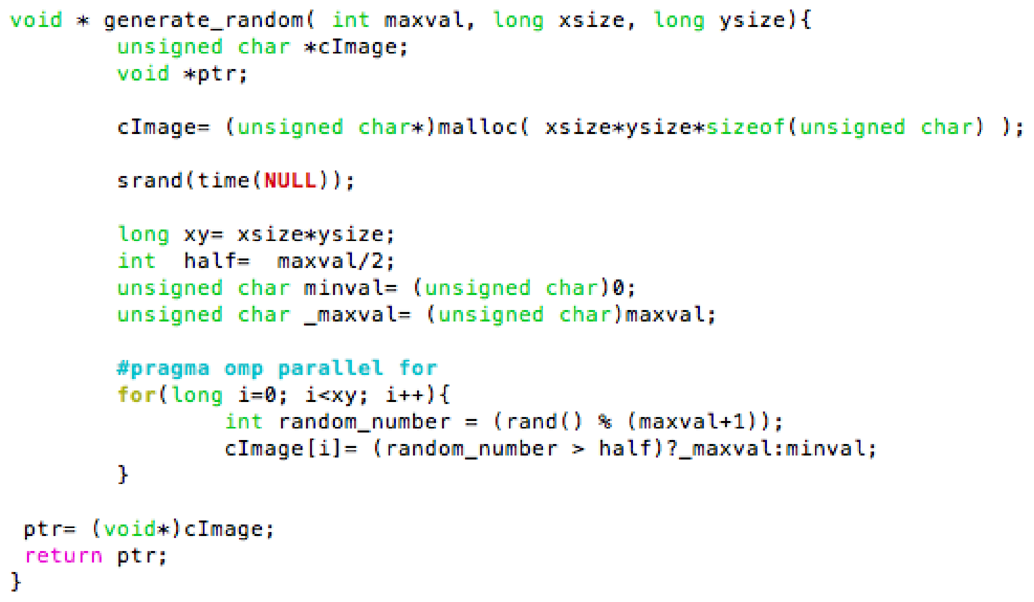
The compilation and linking processes are facilitated by a `Makefile`, which is configured with the necessary compilation flags, such as `-fopenmp` to enable OpenMP thread support, and the appropriate MPI wrapper for MPI-related operations.
Let us dig deeper in each section of the code below.

1.3.1 Main.c

The program executes different tasks depending on the specific argument provided when running it. The main function handles user-passed arguments and initializes the MPI library.
Depending on the inputs and their values either the initialization of the playground or the running of an evolution can be performed.
The function that performs the initialization of the random grid is contained in the `Main.c` file and it can either be performed serially (with the function `generate_random()`) or in parallel (with the function `generate_random_parallel()`). The playground had to be initialized randomly and it had to represent a general $k \times k$ binary matrix.
A detailed description of the two functions follows below.

Serial initialization

As previously said, the `generate_random()` function initializes the grid with random binary values. Although it's primarily a serial function, it can also operate in parallel using `OpenMP threads`. The distinction between serial and parallel versions of our code primarily relates to the use of `MPI`, as both functions utilize `OpenMP threads`. Specifically, this function is executed when only one MPI process is used. The actual function is reported below. Let us examine it step by step to get a better idea of what it does.



```c
void * generate_random( int maxval, long xsize, long ysize){
        unsigned char *cImage;
        void *ptr;

        cImage= (unsigned char*)malloc( xsize*ysize*sizeof(unsigned char) );

        srand(time(NULL));

        long xy= xsize*ysize;
        int  half=  maxval/2;
        unsigned char minval= (unsigned char)0;
        unsigned char _maxval= (unsigned char)maxval;

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

 ptr= (void*)cImage;
 return ptr;
}
```

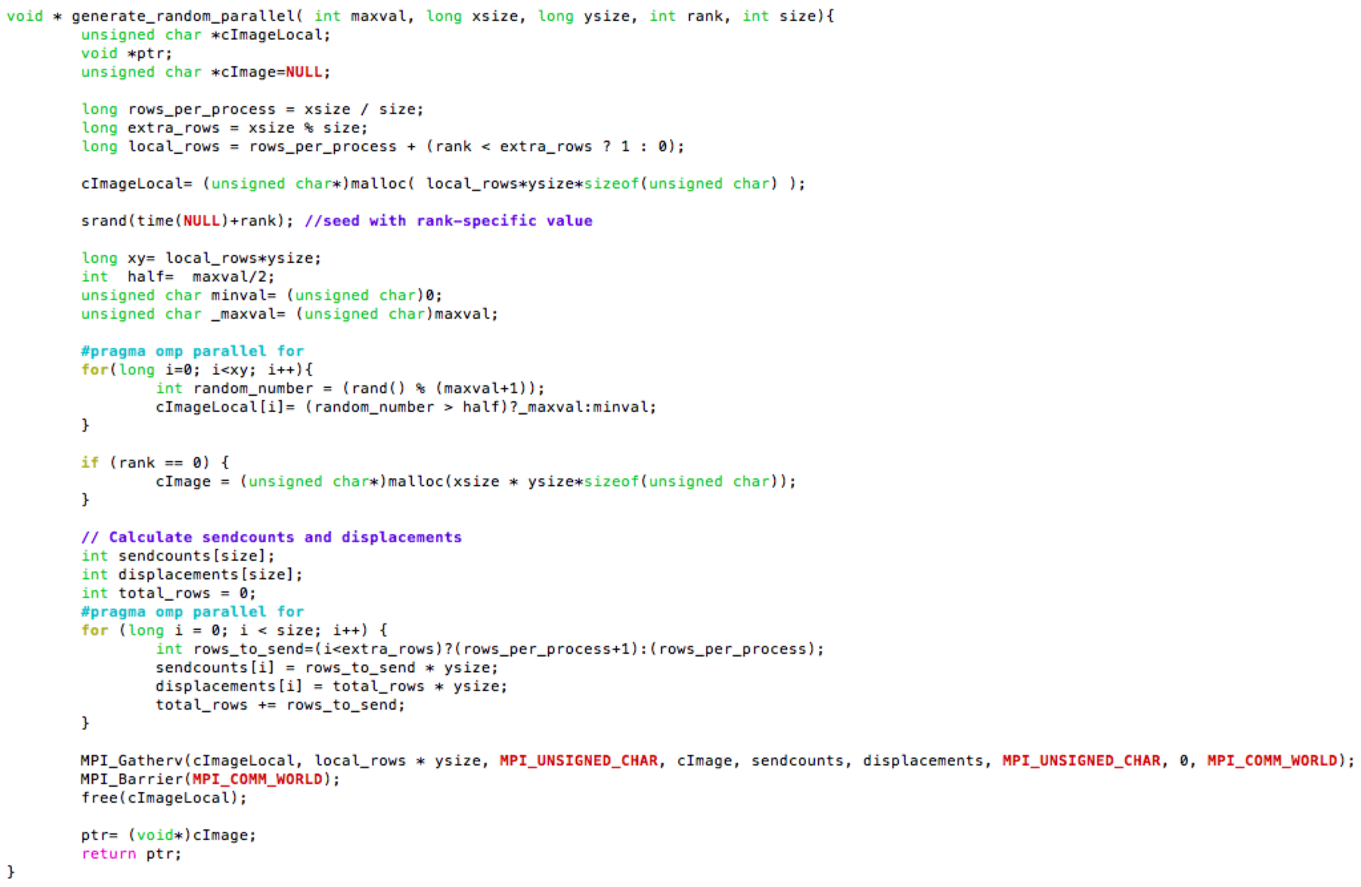
Figure 1.1: Serial initialization of the playground

We declared an array of unsigned characters and a pointer, which will later point to the initialized playground.
Next, we allocated the necessary memory space for the array using the `malloc()` function.
The main task of initializing the matrix with random values occurs within a `for loop`, parallelized using the `#pragma omp parallel for` directive. In each iteration, a random number within the range of 0 to 255 (our maximum value, given that we use one byte per pixel, allowing for 256 possible values) is generated. If the generated random number exceeds the median value (calculated as half of 255), the corresponding cell in the matrix is set to the maximum value (255). Otherwise, it is set to the minimum value (0).
After completing the loop, the matrix is entirely initialized with binary values (0s and 255s), and it can be returned through the function using the previously declared pointer.
Memory deallocation for the matrix occurs later, after the function returns the pointer and the matrix is written to another location.

Parallel initialization

The parallel version of the code, `generate_random_parallel()`, works similarly to the serial one so we are only going to highlight the differences between the 2 versions.

The function used for the parallel initialization is reported below.



```c
void * generate_random_parallel( int maxval, long xsize, long ysize, int rank, int size){
        unsigned char *cImageLocal;
        void *ptr;
        unsigned char *cImage=NULL;

        long rows_per_process = xsize / size;
        long extra_rows = xsize % size;
        long local_rows = rows_per_process + (rank < extra_rows ? 1 : 0);

        cImageLocal= (unsigned char*)malloc( local_rows*ysize*sizeof(unsigned char) );

        srand(time(NULL)+rank); //seed with rank-specific value

        long xy= local_rows*ysize;
        int  half=  maxval/2;
        unsigned char minval= (unsigned char)0;
        unsigned char _maxval= (unsigned char)maxval;

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

        if (rank == 0) {
                cImage = (unsigned char*)malloc(xsize * ysize*sizeof(unsigned char));
        }

        // Calculate sendcounts and displacements
        int sendcounts[size];
        int displacements[size];
        int total_rows = 0;
        #pragma omp parallel for
        for (long i = 0; i < size; i++) {
                int rows_to_send=(i<extra_rows)?(rows_per_process+1):(rows_per_process);
                sendcounts[i] = rows_to_send * ysize;
                displacements[i] = total_rows * ysize;
                total_rows += rows_to_send;
        }

        MPI_Gatherv(cImageLocal, local_rows * ysize, MPI_UNSIGNED_CHAR, cImage, sendcounts, displacements, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        free(cImageLocal);

        ptr= (void*)cImage;
        return ptr;
}
```

Figure 1.2: Parallel initialization of the playground

The main difference lies in the use of `MPI` (Message Passing Interface) in the parallel version, which enables the use of multiple processes to distribute and handle the workload efficiently. To achieve this, we partitioned the matrix by rows, as matrices are typically stored and accessed in a row-major order, making this division an efficient choice.
To evenly distribute the rows of the original matrix among different processes, we used the following formula to determine the number of rows assigned to each process:

$$local\_rows_n = \begin{cases} floor\left(\frac{k}{size}\right) + 1 & if \quad rank < n_{extra\ rows} \\ floor\left(\frac{k}{size}\right) & otherwise \end{cases} \tag{1.1}$$

For instance, consider that k represents the total number of rows in the grid and $size$ indicates the number of processes. If the number of rows is evenly divisible

by the number of processes, then each process will work on exactly $k/size$ local rows. However, when some `extra_rows` remain, an extra row will be assigned to each process, starting from process 0 and continuing until they are exhausted. This decomposition strategy ensures that the number of `local_rows` per process will differ by at most one, which implies that the workload is fairly balanced among all processes.
Once the number of local rows for each process is determined, each process proceeds to initialize its part of the grid using a `for` cycle that works exactly like the one presented in the serial version of the code. It's important to note that each process utilizes a distinct random seed, derived from its rank, in order to avoid possible patterns in the overall grid.
When all the processes are finished with their initialization, all pieces of data are collected into a single array on the process whose `rank` is equal to 0. The collection is achieved through an `MPI_Gatherv` call. In order to account for possible variations in the number of rows handled by each process, the arrays `sendcounts` and `displacements` were created. These arrays specify the number of elements that each process has to send to process 0 (sendcounts array) and the position in which said elements should be positioned in the final complete matrix (displacements array).
The final initialized playground is then returned by the function, as seen in the serial version of the code.

1.3.2 Static_evolution.c

The `Static_evolution.c` file contains the code that performs the static evolution of the playground.
In the "static evolution" the status evaluation and status update of each cell are disentangled, meaning that the status of all the cells (i.e., the computation of how many alive adjacent each of them has) should be evaluated at first, freezing the system, and updating the status of the cells only afterwards.
Exactly like the initialization, the static evolution can be performed serially or in parallel. We will analyze the two versions.

Serial Static Evolution

Presented below there is the code that implements the serial static evolution (which, again, is considered serial regarding the number of `MPI` processes but is actually parallelized through the use of `OpenMP` directives).

The code works in the following way. When the function `run_static()` is called two arrays of unsigned char are declared: `prevMatrix` (for storing the current state of the matrix) and `newMatrix` (for storing the updated state).

```c
void run_static(char * filename, int t, int s){
        long xsize = 0;
        long ysize = 0;
        int maxval = 0;
        unsigned char *prevMatrix;
        unsigned char *newMatrix;


        double start_time = MPI_Wtime();

        //Initialize the new matrix by reading the pgm file where it's stored
        read_pgm_image( (void**)&newMatrix, &maxval, &xsize, &ysize, filename);

        long size=xsize*ysize;

        //Allocate the space needed to store the previous matrix
        prevMatrix=(unsigned char*)malloc( (xsize+2)*ysize*sizeof(unsigned char) );

        printf("Performing the static evolution for %d times\n",t);
        //Perform the static evolution
        char fout[26];
        for(int i=1;i<=t;i++){
                #pragma omp parallel for
                for(long j=0; j< size; j++){
                        prevMatrix[j+xsize]=newMatrix[j];
                }

                #pragma omp parallel for
                for(long j=0; j< xsize; j++){
                        prevMatrix[j]=newMatrix[j+(xsize*(xsize-1))];
                        prevMatrix[((xsize+1)*xsize)+j]=newMatrix[j];
                }

                update_static_serial(prevMatrix, newMatrix, xsize);

                if((s!=0)&&(i%s==0)){
                        //Write the output image
                        sprintf(fout, "static_snapshot_%05d.pgm", i);
                        write_pgm_image((void*)newMatrix, maxval, xsize, ysize, fout);
                }
        }

        if(s==0){
                //Write the output image
                sprintf(fout, "static_snapshot_%05d.pgm", t);
                write_pgm_image((void*)newMatrix, maxval, xsize, ysize, fout);
        }

        free(prevMatrix);
        free(newMatrix);

        double end_time = MPI_Wtime();
        double elapsed_time = end_time - start_time;
        printf("%ld,%d,%d,%f\n",ysize,1,omp_get_max_threads(),elapsed_time);
}
```

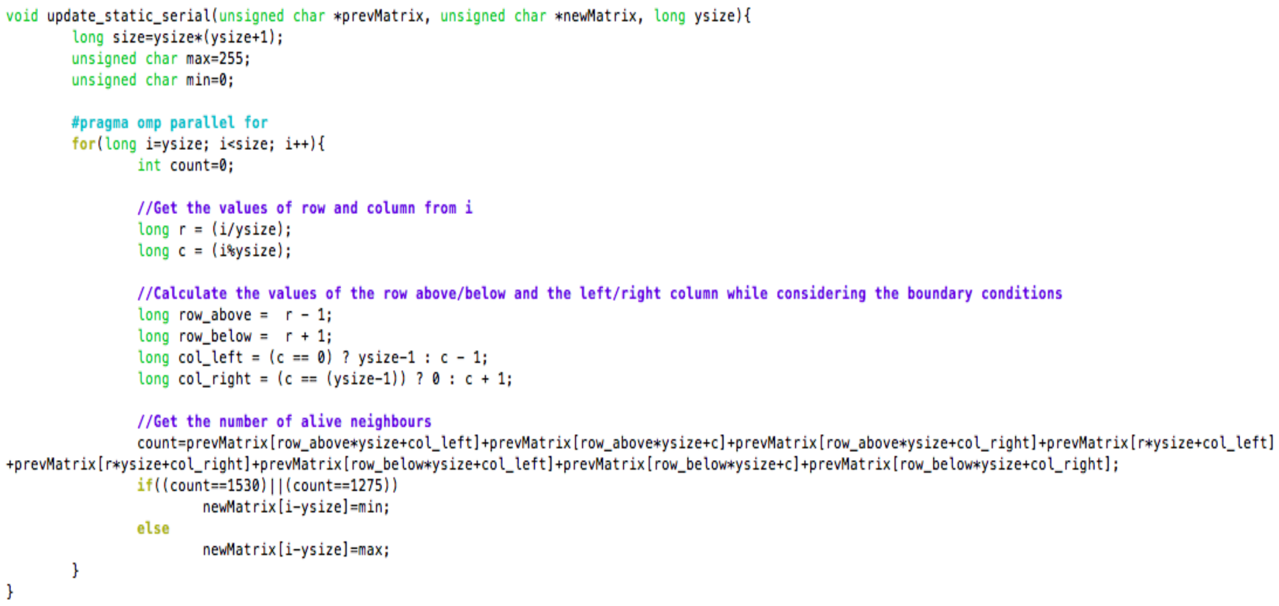
Figure 1.3: Running the static evolution

The new matrix is initialized by reading the file containing the **.pgm** image with

the initial conditions using the `read_pgm_image()` function, and the space needed to store the previous matrix is allocated. The previous matrix has two extra rows, which are used to store copies of the boundary rows to facilitate the update.
Then we enter a `for` loop that has as many iterations as the number of times that the evolution has to be performed. At each iteration the cells of the previous matrix are initialized by copying the values of the cells of the new matrix, and the two extra rows of the previous matrix are initialized as follows:

- First row: it contains the values of the last row of the new matrix
- Last row: it contains the values of the first row of the new matrix

After the initialization of the previous matrix, the function that updates the cells according to the static evolution, `update_static_serial()`, is called and every s steps a snapshot of the new matrix is written using the function `write_pgm_image()`.
At the end of the evolution the time taken to perform it is printed along with the information about the matrix size, the number of MPI processes (which is set to 1 for the serial code) and the number of OpenMP threads.
The actual update of the cells is done inside the `update_static_serial()` function, which is shown below.



```
void update_static_serial(unsigned char *prevMatrix, unsigned char *newMatrix, long ysize){
        long size=ysize*(ysize+1);
        unsigned char max=255;
        unsigned char min=0;

        #pragma omp parallel for
        for(long i=ysize; i<size; i++){
                int count=0;

                //Get the values of row and column from i
                long r = (i/ysize);
                long c = (i%ysize);

                //Calculate the values of the row above/below and the left/right column while considering the boundary conditions
                long row_above =  r - 1;
                long row_below =  r + 1;
                long col_left = (c == 0) ? ysize-1 : c - 1;
                long col_right = (c == (ysize-1)) ? 0 : c + 1;

                //Get the number of alive neighbours
                count=prevMatrix[row_above*ysize+col_left]+prevMatrix[row_above*ysize+c]+prevMatrix[row_above*ysize+col_right]+prevMatrix[r*ysize+col_left]
+prevMatrix[r*ysize+col_right]+prevMatrix[row_below*ysize+col_left]+prevMatrix[row_below*ysize+c]+prevMatrix[row_below*ysize+col_right];
                if((count==1530)||(count==1275))
                        newMatrix[i-ysize]=min;
                else
                        newMatrix[i-ysize]=max;
        }
}
```

Figure 1.4: Update of the cells for the static evolution

The function contains a for loop that iterates over the cells of the matrix excluding the first and last rows, which are only used to consider the neighbors of the cells.

During each iteration the function calculates the row and column indices based on the index `i`. These indices are then used to define the indices of the rows above and below, as well as the columns to the left and right of the current cell. These new indices are essential for enforcing the boundary conditions of the matrix. Once the indices are computed, the function accesses the 8 neighbors of each cell by referencing their positions using the respective indices.
The update of each cell follows the rules of the game. After counting the values of the neighboring cells, the function checks whether there are between 2 and 3 alive neighbors. If this condition is met, the corresponding cell in the new matrix is set to alive (black); otherwise, it is set to dead (white).
To determine the number of alive neighbors, the function counts how many of them are set to white (value 255) and checks if there are exactly 5 (value 1275, as $255 * 5 = 1275$) or 6 (value 1530, as $255 * 6 = 1530$) dead neighboring cells. Specific values like 1275 and 1530 are used in order to eliminate the need for division operations inside the loop, which would have been computationally expensive.

Parallel Static Evolution

In the parallel version of the code for the static evolution, there are several key differences compared to the serial version. Here's a breakdown of the differences:
Domain Decomposition: In the parallel version, the work is divided among different processes. Three arrays are declared: two for storing the local previous and new matrices in each process, and one for storing the complete matrix on process 0.
Broadcasting Matrix Size: Process 0 reads the matrix and its corresponding size (number of rows/columns) and broadcasts the size of the matrix to all other processes. This step is necessary for computing the number of local rows that each process will have.
Local Matrix Structure: Each process stores a certain number of rows, similar to the parallel initialization code, plus 2 more rows. These additional rows are needed to store the neighboring rows above and below, which belong to the local rows assigned to the process of the rank above and below.

Matrix Initialization: The new local matrix does not have these 2 additional rows (as they are only needed for actual updates and checking neighboring rows). Its values are obtained through a `Scatterv MPI` call that distributes the original matrix among the different processes.
Boundary Conditions: The domain decomposition ensures that boundary conditions are satisfied. Each process sends its second row to the process with a rank above and sends its second-to-last row to the process with a rank below. Each process also receives the necessary rows: its first row from the process below and its

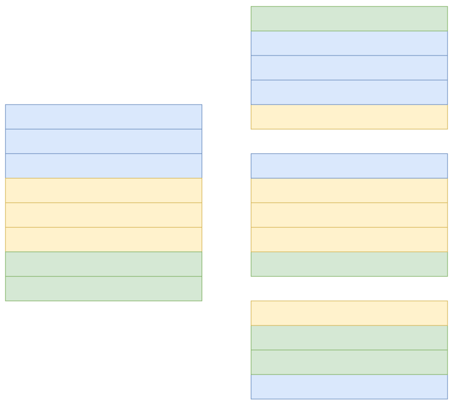
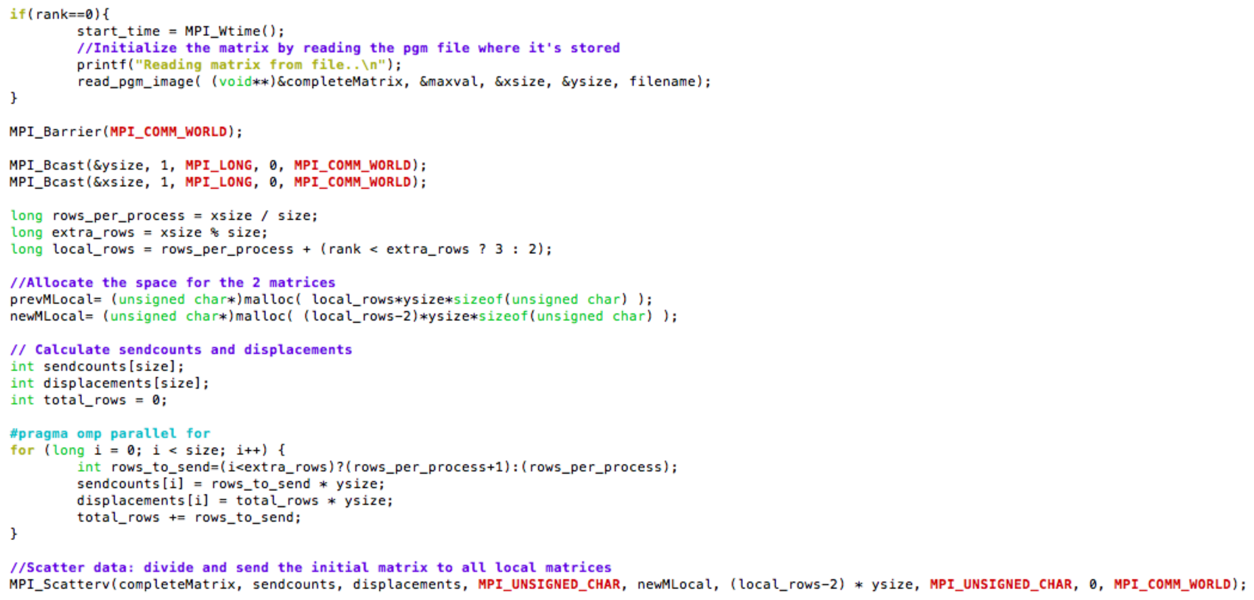


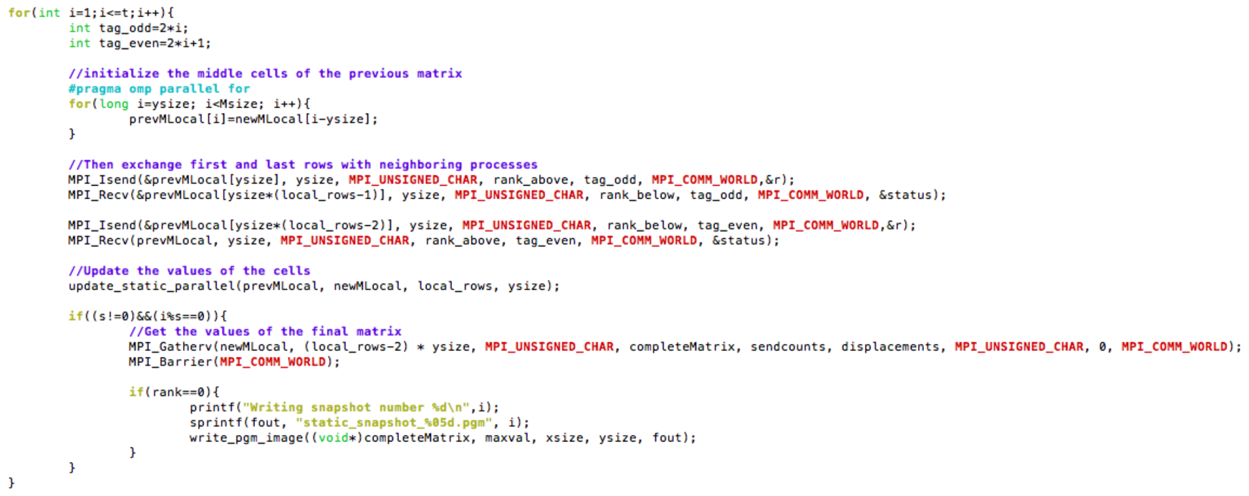
Figure 1.5: Decomposition of the original matrix into the local previous matrices.

last row from the process above.
Cell Updates: During the update, new values are directly written inside the new local matrix, and these values are then copied into the previous local matrix at the start of the following iteration.
Snapshot Creation: When a snapshot of the matrix is required (every s steps), the final matrix is composed through a `Gatherv MPI` call. This call gathers the values of all the new local matrices and merges them into the complete matrix, which is only stored on process 0. Process 0 then writes the image using the `write_pgm_image()` function.
These operations ensure that the work is distributed among processes, boundary conditions are maintained, and data is correctly gathered and composed when needed.



```c
if(rank==0){
        start_time = MPI_Wtime();
        //Initialize the matrix by reading the pgm file where it's stored
        printf("Reading matrix from file..\n");
        read_pgm_image( (void**)&completeMatrix, &maxval, &xsize, &ysize, filename);
}

MPI_Barrier(MPI_COMM_WORLD);

MPI_Bcast(&ysize, 1, MPI_LONG, 0, MPI_COMM_WORLD);
MPI_Bcast(&xsize, 1, MPI_LONG, 0, MPI_COMM_WORLD);

long rows_per_process = xsize / size;
long extra_rows = xsize % size;
long local_rows = rows_per_process + (rank < extra_rows ? 3 : 2);

//Allocate the space for the 2 matrices
prevMLocal= (unsigned char*)malloc( local_rows*ysize*sizeof(unsigned char) );
newMLocal= (unsigned char*)malloc( (local_rows-2)*ysize*sizeof(unsigned char) );

// Calculate sendcounts and displacements
int sendcounts[size];
int displacements[size];
int total_rows = 0;

#pragma omp parallel for
for (long i = 0; i < size; i++) {
        int rows_to_send=(i<extra_rows)?(rows_per_process+1):(rows_per_process);
        sendcounts[i] = rows_to_send * ysize;
        displacements[i] = total_rows * ysize;
        total_rows += rows_to_send;
}

//Scatter data: divide and send the initial matrix to all local matrices
MPI_Scatterv(completeMatrix, sendcounts, displacements, MPI_UNSIGNED_CHAR, newMLocal, (local_rows-2) * ysize, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
```

Figure 1.6: Initialization of the parallel static evolution



```c
for(int i=1;i<=t;i++){
        int tag_odd=2*i;
        int tag_even=2*i+1;

        //initialize the middle cells of the previous matrix
        #pragma omp parallel for
        for(long i=ysize; i<Msize; i++){
                prevMLocal[i]=newMLocal[i-ysize];
        }

        //Then exchange first and last rows with neighboring processes
        MPI_Isend(&prevMLocal[ysize], ysize, MPI_UNSIGNED_CHAR, rank_above, tag_odd, MPI_COMM_WORLD,&r);
        MPI_Recv(&prevMLocal[ysize*(local_rows-1)], ysize, MPI_UNSIGNED_CHAR, rank_below, tag_odd, MPI_COMM_WORLD, &status);

        MPI_Isend(&prevMLocal[ysize*(local_rows-2)], ysize, MPI_UNSIGNED_CHAR, rank_below, tag_even, MPI_COMM_WORLD,&r);
        MPI_Recv(prevMLocal, ysize, MPI_UNSIGNED_CHAR, rank_above, tag_even, MPI_COMM_WORLD, &status);

        //Update the values of the cells
        update_static_parallel(prevMLocal, newMLocal, local_rows, ysize);

        if((s!=0)&&(i%s==0)){
                //Get the values of the final matrix
                MPI_Gatherv(newMLocal, (local_rows-2) * ysize, MPI_UNSIGNED_CHAR, completeMatrix, sendcounts, displacements, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
                MPI_Barrier(MPI_COMM_WORLD);

                if(rank==0){
                        printf("Writing snapshot number %d\n",i);
                        sprintf(fout, "static_snapshot_%05d.pgm", i);
                        write_pgm_image((void*)completeMatrix, maxval, xsize, ysize, fout);
                }
        }
}
```

Figure 1.7: Running the parallel static evolution

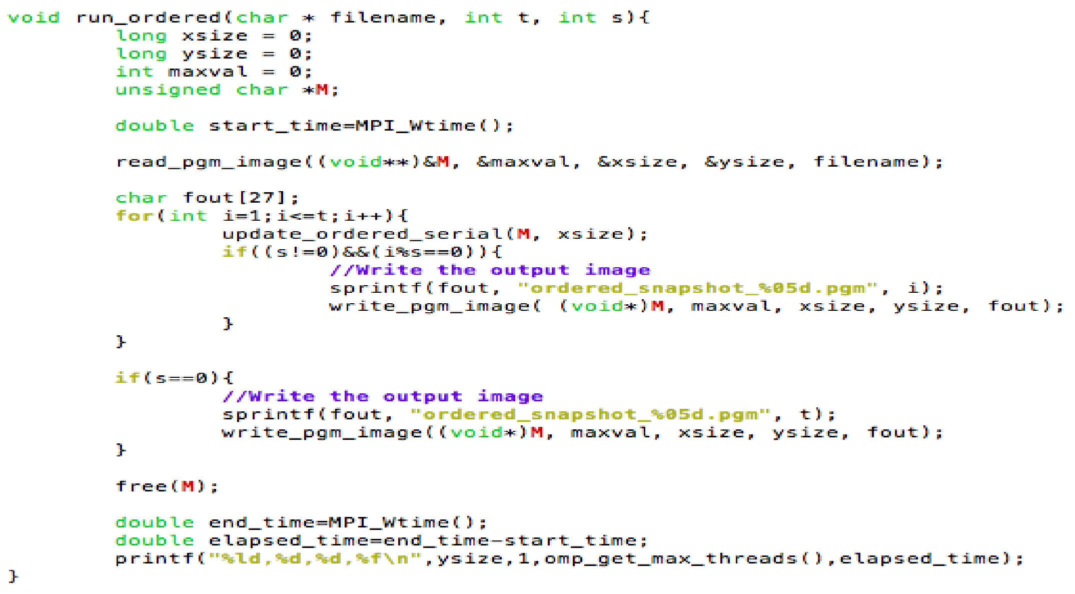
1.3.3 Ordered_evolution.c

For the "ordered evolution" the cells are upgraded in row-major order and since, obviously, changing a cell's status impacts on the fate of adjacent cells, that inserts a "spurious" signal in the system evolution. Note that this evolution is intrinsically serial because of the inherent dependency that descends from its definition.
The ordered evolution, while inherently serial, can be executed both serially and in

parallel, although parallel execution doesn't offer significant performance benefits. Differently from the other codes these functions do not use `OpenMP threads` and the only considered form of parallelization is given by the use of `MPI`. Let's analyze the two versions.

Serial Ordered Evolution

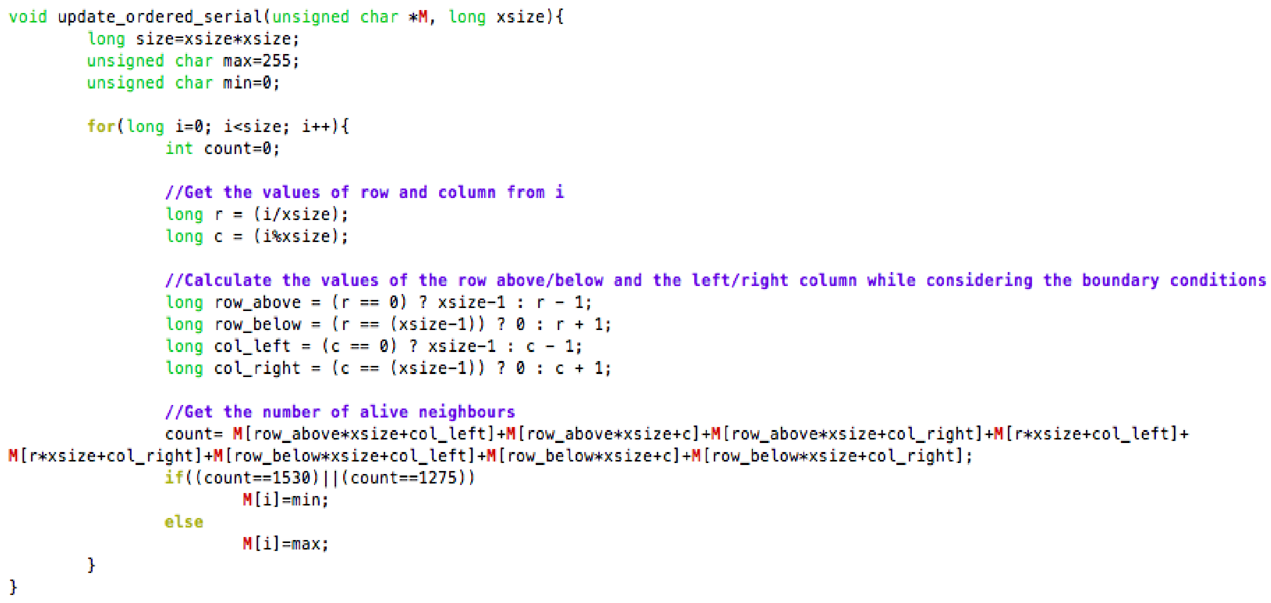
Presented below there is the code that implements the serial ordered evolution.



```c
void run_ordered(char * filename, int t, int s){
        long xsize = 0;
        long ysize = 0;
        int maxval = 0;
        unsigned char *M;

        double start_time=MPI_Wtime();

        read_pgm_image((void**)&M, &maxval, &xsize, &ysize, filename);

        char fout[27];
        for(int i=1;i<=t;i++){
                update_ordered_serial(M, xsize);
                if((s!=0)&&(i%s==0)){
                        //Write the output image
                        sprintf(fout, "ordered_snapshot_%05d.pgm", i);
                        write_pgm_image( (void*)M, maxval, xsize, ysize, fout);
                }
        }

        if(s==0){
                //Write the output image
                sprintf(fout, "ordered_snapshot_%05d.pgm", t);
                write_pgm_image((void*)M, maxval, xsize, ysize, fout);
        }

        free(M);

        double end_time=MPI_Wtime();
        double elapsed_time=end_time-start_time;
        printf("%ld,%d,%d,%f\n",ysize,1,omp_get_max_threads(),elapsed_time);
}
```

Figure 1.8: Running the ordered evolution

When the `run_ordered()` function is called, an array of unsigned characters is declared and the matrix is initialized by reading the file containing the **.pgm** image. The function `update_ordered_serial()` is called for the required number of iterations. Every s steps, a snapshot of the new matrix is written using the `write_pgm_image()` function.
The actual update of the cells happens inside the `update_ordered_serial()` function. This function is similar to `update_static_serial()`, with the main difference being that each cell is updated instantly. Its updated value is used in the next iteration of the loop when considering the values of neighboring cells, rather than using its initial value.



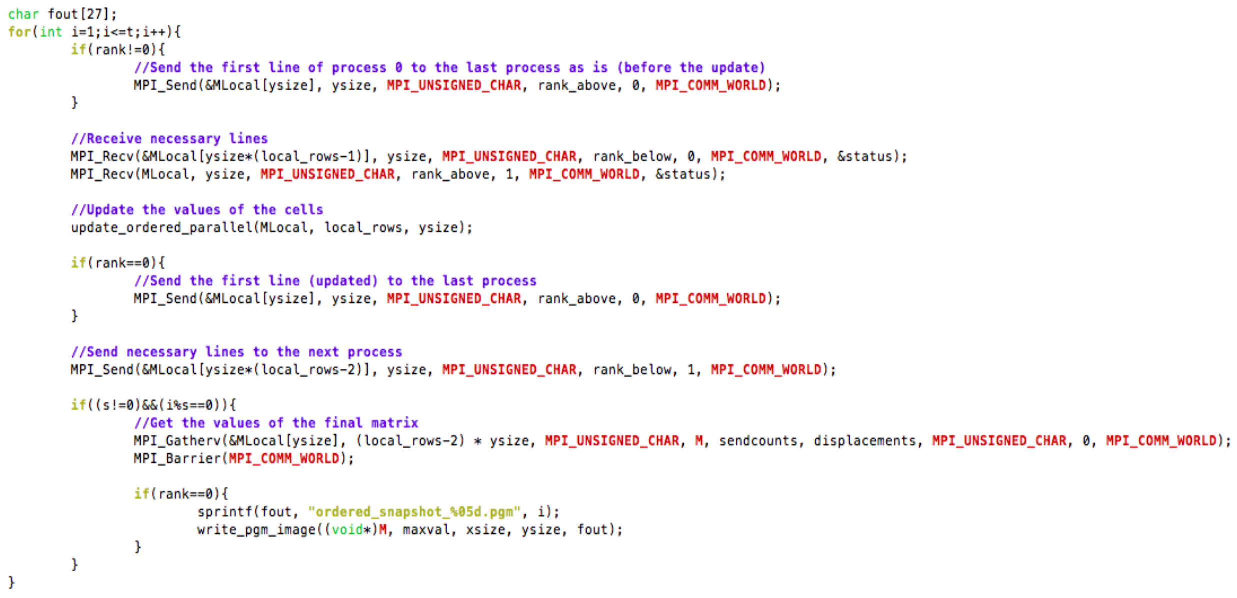
```
void update_ordered_serial(unsigned char *M, long xsize){
        long size=xsize*xsize;
        unsigned char max=255;
        unsigned char min=0;

        for(long i=0; i<size; i++){
                int count=0;

                //Get the values of row and column from i
                long r = (i/xsize);
                long c = (i%xsize);

                //Calculate the values of the row above/below and the left/right column while considering the boundary conditions
                long row_above = (r == 0) ? xsize-1 : r - 1;
                long row_below = (r == (xsize-1)) ? 0 : r + 1;
                long col_left = (c == 0) ? xsize-1 : c - 1;
                long col_right = (c == (xsize-1)) ? 0 : c + 1;

                //Get the number of alive neighbours
                count= M[row_above*xsize+col_left]+M[row_above*xsize+c]+M[row_above*xsize+col_right]+M[r*xsize+col_left]+
M[r*xsize+col_right]+M[row_below*xsize+col_left]+M[row_below*xsize+c]+M[row_below*xsize+col_right];
                if((count==1530)||(count==1275))
                        M[i]=min;
                else
                        M[i]=max;
        }
}
```

Figure 1.9: Update of the cell for the ordered evolution

Parallel Ordered Evolution

In the parallel version of the code for the ordered evolution, there are several key differences compared to the serial version.

Here's a breakdown of the differences:
Domain Decomposition: In the parallel version, the work is divided among different processes. Two arrays are declared: one for storing the local matrix in each process, and one for storing the complete matrix on process 0.
Broadcasting Matrix Size: Process 0 reads the matrix and its corresponding size (number of rows/columns) and broadcasts the size of the matrix to all other processes. This step is necessary for computing the number of local rows that each process will have.
Local Matrix Structure: Each process stores a certain number of rows, similar to the parallel initialization code, plus 2 more rows. These additional rows are needed to store the neighboring rows above and below, which belong to the local rows assigned to the process of the rank above and below.
Matrix Initialization: The Middle rows (all except the first and last) of the local matrices are initialized through a `Scatterv MPI` call that distributes the original matrix among the different processes.
Boundary Conditions: Given that cells are updated throughout the evolution and not just when the last cell is reached, sending and receiving rows between processes



```c
char fout[27];
for(int i=1;i<=t;i++){
        if(rank!=0){
                //Send the first line of process 0 to the last process as is (before the update)
                MPI_Send(&MLocal[ysize], ysize, MPI_UNSIGNED_CHAR, rank_above, 0, MPI_COMM_WORLD);
        }

        //Receive necessary lines
        MPI_Recv(&MLocal[ysize*(local_rows-1)], ysize, MPI_UNSIGNED_CHAR, rank_below, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(MLocal, ysize, MPI_UNSIGNED_CHAR, rank_above, 1, MPI_COMM_WORLD, &status);

        //Update the values of the cells
        update_ordered_parallel(MLocal, local_rows, ysize);

        if(rank==0){
                //Send the first line (updated) to the last process
                MPI_Send(&MLocal[ysize], ysize, MPI_UNSIGNED_CHAR, rank_above, 0, MPI_COMM_WORLD);
        }

        //Send necessary lines to the next process
        MPI_Send(&MLocal[ysize*(local_rows-2)], ysize, MPI_UNSIGNED_CHAR, rank_below, 1, MPI_COMM_WORLD);

        if((s!=0)&&(i%s==0)){
                //Get the values of the final matrix
                MPI_Gatherv(&MLocal[ysize], (local_rows-2) * ysize, MPI_UNSIGNED_CHAR, M, sendcounts, displacements, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
                MPI_Barrier(MPI_COMM_WORLD);

                if(rank==0){
                        sprintf(fout, "ordered_snapshot_%05d.pgm", i);
                        write_pgm_image((void*)M, maxval, xsize, ysize, fout);
                }
        }
}
```

Figure 1.10: Running the parallel ordered evolution

is crucial. However, the order of these operations is significant. Before entering the loop for evolution, the last process sends its second-to-last row to process 0 to ensure that process 0 has the necessary information for its update. Then in each iteration of the loop:

- If the current process is not 0, the second row is sent to the process with a higher rank.
- The current process receives the first row from the process with a rank above and the last row from the process with a rank below.
- Cell updates are performed.
- If the current process is process 0, it sends its second row to the last process.
- Finally, all processes send their second-to-last rows to the process with a rank below

Cell Updates: During the update, new values are directly written inside the cells of the local matrix.
Snapshot Creation: When a snapshot of the matrix is required (every s steps), the final matrix is composed through a `Gatherv MPI` call. This call gathers the values of all the local matrices and merges them into the complete matrix, which is only stored on process 0. Process 0 then writes the image using the `write_pgm_image()` function.

These operations ensure that the work is distributed among processes, boundary conditions are maintained, and data is correctly gathered and composed when needed.

1.3.4 Read_write.c

The file includes two functions: `read_pgm_image()` and `write_pgm_image()`. These functions were provided by our professors and were used as they were. For additional details, you can refer to the course's GitHub repository [2] for the specific code implementation.

1.4 Results and Discussion

1.4.1 Correctness

The initial step was to validate the program's functionality. This meant confirming that it correctly initialized a random binary image when necessary and executed the intended evolutions accurately. To ensure this, we conducted a manual assessment of our program through the following procedure. To begin, we generated a small random binary image, employing both the serial and parallel versions of the code to ensure their proper functioning. The image's dimensions were chosen as 10×10; a small size was necessary to facilitate manual verification of the correctness of the evolutions. We performed both evolutions — static and ordered — twice, each time employing the same initial conditions represented by the random image we generated. At each iteration, we captured a snapshot of the evolving image.

The resulting images are presented below. A thorough examination of these images, following the guidelines specific to each evolution, served as confirmation of the code's accuracy.

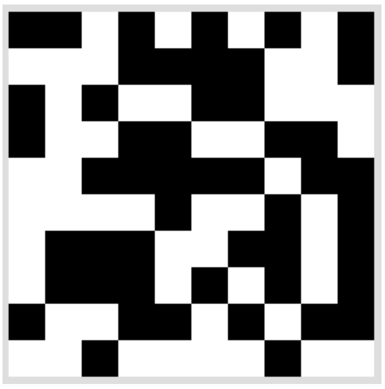


Figure 1.11: Initial Image

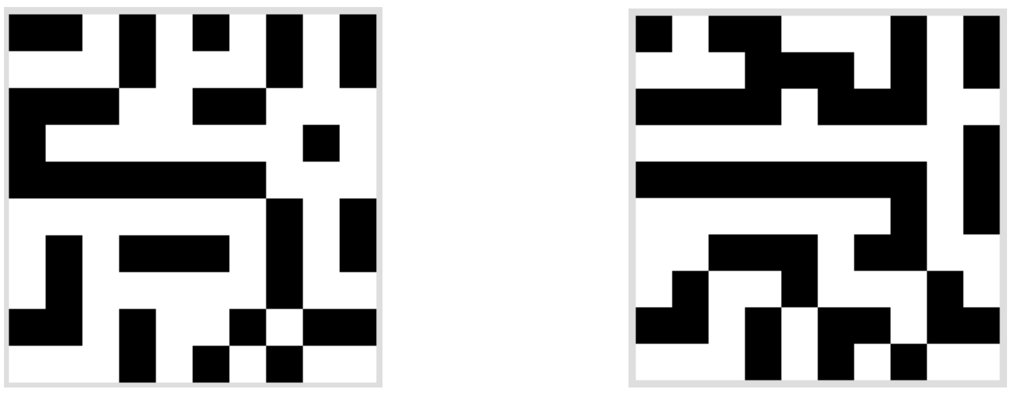


Figure 1.12: Ordered evolution

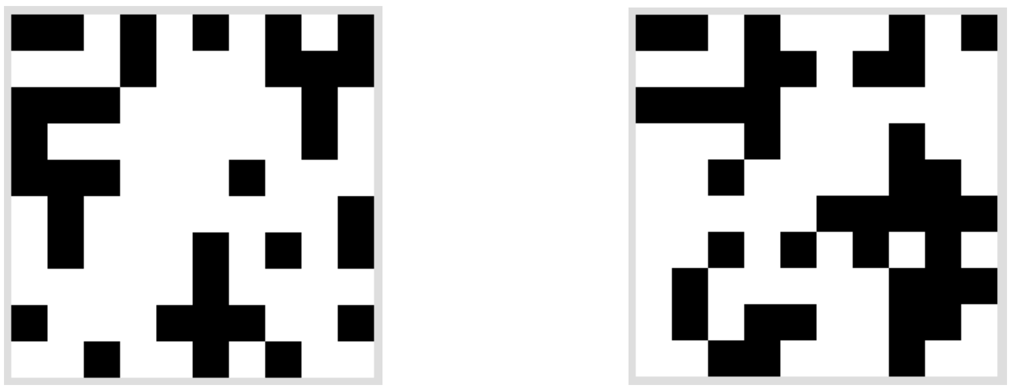


Figure 1.13: Static evolution

1.4.2 Scalability Study

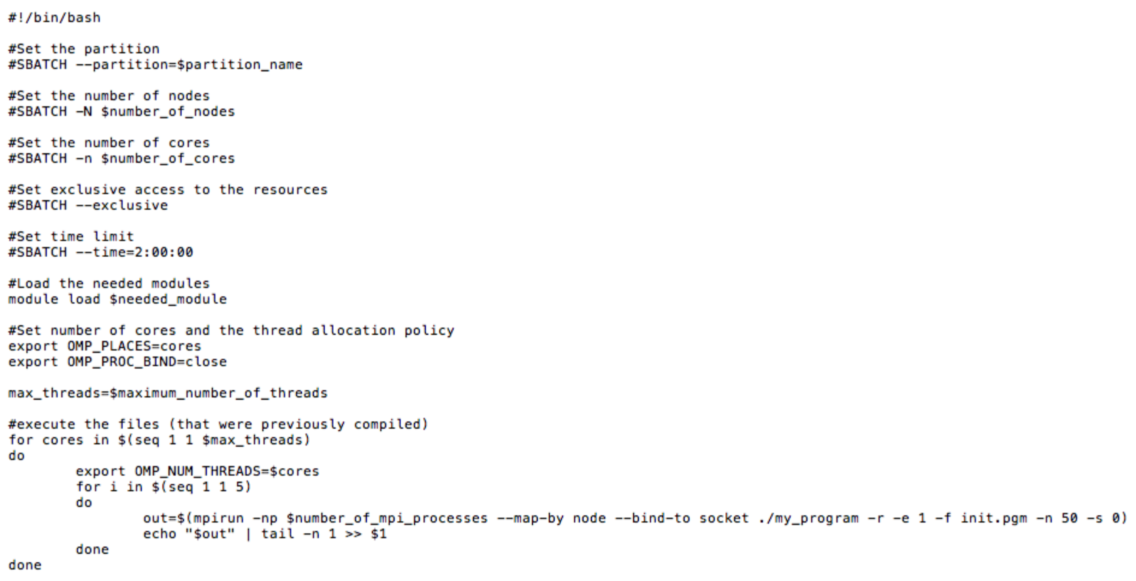
To measure the efficiency of the code (in terms of parallelization) we performed a scalability study. Precisely we considered the following scenarios:

- **OpenMP scalability**
- **Strong MPI scalability**
- **Weak MPI scalability**

SLURM, ORFEO's resource management software, offers users the flexibility to submit jobs in both interactive and batch modes. In the interactive mode, users are required to manually load modules, request resource allocation, and await approval before executing their code. Conversely, in batch mode, users can streamline the process by incorporating module loading, resource requests, and shell commands into a bash script (referred to as the job file). Subsequently, they can initiate job submission with a simple command: `sbatch job_file_name.sh`. This approach eliminates the need to wait for available resources, enabling autonomous execution. Given our objectives, the batch mode proved significantly more suitable, and we developed various job files for launching different measurement tasks.

OpenMP Scalability

We aimed to measure the program's scalability as we increased the number of OpenMP threads. The test was conducted on both EPYC and THIN nodes, with one MPI Task allocated for each socket using the `--map-by socket` option. We systematically varied the number of threads, ranging from 1 to the maximum number of cores available to saturate each socket, and repeated measurements with 1 to 8 MPI processes.
The general structure of the job files that we used to perform the OpenMP scalability can be seen below.



```
#!/bin/bash

#Set the partition
#SBATCH --partition=$partition_name

#Set the number of nodes
#SBATCH -N $number_of_nodes

#Set the number of cores
#SBATCH -n $number_of_cores

#Set exclusive access to the resources
#SBATCH --exclusive

#Set time limit
#SBATCH --time=2:00:00

#Load the needed modules
module load $needed_module

#Set number of cores and the thread allocation policy
export OMP_PLACES=cores
export OMP_PROC_BIND=close

max_threads=$maximum_number_of_threads

#execute the files (that were previously compiled)
for cores in $(seq 1 1 $max_threads)
do
        export OMP_NUM_THREADS=$cores
        for i in $(seq 1 1 5)
        do
                out=$(mpirun -np $number_of_mpi_processes --map-by node --bind-to socket ./my_program -r -e 1 -f init.pgm -n 50 -s 0)
                echo "$out" | tail -n 1 >> $1
        done
done
```

Figure 1.14: batch job for the OpenMP scalability study

The job file is made of the following components:

- SLURM directives: These directives, preceded by '#' and therefore ignored by the shell, request the necessary resources. They include the partition ('–partition'), the number of nodes ('-N'), the number of cores ('-n'), the time limit ('–time'), and '–exclusive' for exclusive node access. For EPYC nodes, the partition is named "EPYC," and the maximum number of threads is 64. THIN nodes use the "THIN" partition, and the maximum thread count is 12.

- Module loading and threads affinity policy setting: the module system is addressed to load the needed openMPI library and the threads affinity policy is configured.
- Computation section: Two nested loops are employed; one to iterate through the number of cores and another to repeat each measurement five times. Within the inner loop, the program is executed with the specified settings. The 'mpirun' option `--map-by socket` ensures that each process is placed on a different socket, spawning threads on the cores of that socket.

We opted to use the same initial image, **init.pgm**, for all measurements. This image was precomputed and stored in the same directory as the job file for accessibility during runtime.
Additionally, we created **.csv** files to store the results, passing them as inline input arguments to the jobs.

The outputs were used to obtain the *Speedup*, which was subsequently plotted, and is computed using the general formula

$$Speedup = \frac{T_1}{T_p}$$

Please note that in our study, T_1 represents the average execution time of a single process, computed from 5 iterations, while T_p denotes the elapsed time for p processes, with the same averaging process applied.

The results are presented in the following pages. Specifically, concerning the static evolution on the EPYC node, we conducted experiments using two different matrix sizes - namely 10000 and 25000 - with 50 iterations each. On the other hand, for the ordered evolution, we exclusively considered the smaller matrix size and performed fewer iterations, specifically 5.

For the THIN nodes, the results were gathered using a unique matrix size of 10000, while maintaining the same number of iterations for each evolution as in the case of EPYC nodes.

By considering just 1 socket, the results are quite satisfactory, even if performance on THIN nodes is much better than the one obtained on EPYC nodes. This is probably due to the smaller number of available threads since also EPYC nodes seem to perform quite well up to a certain number of cores. One possible cause could be that increasing the thread count leads to distributing the workload among a higher number of threads, each handling smaller amounts of data. Since caches work 'by lines', a higher core count increases the likelihood of multiple cores simultaneously

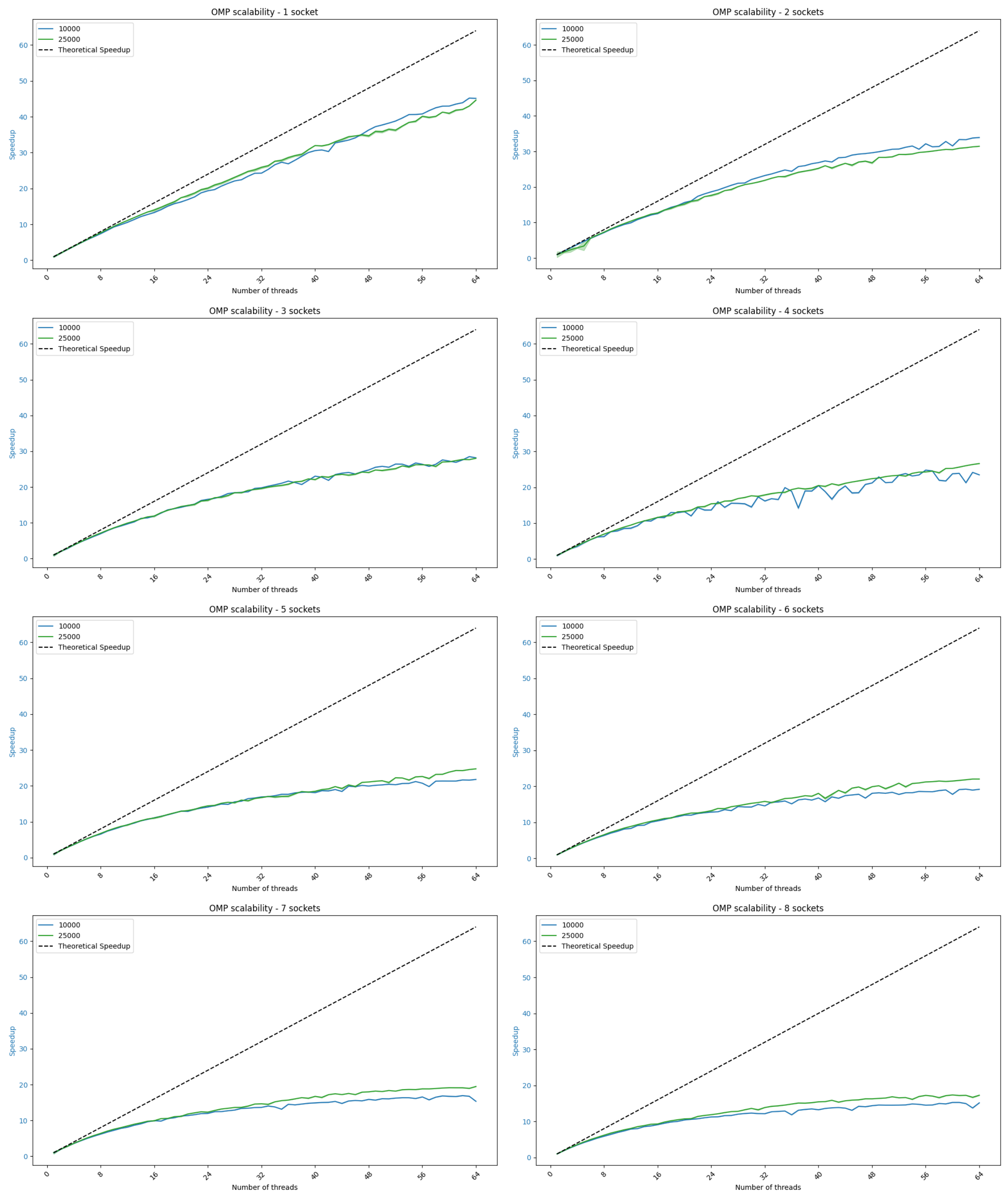


Figure 1.15: OMP scalability on EPYC nodes

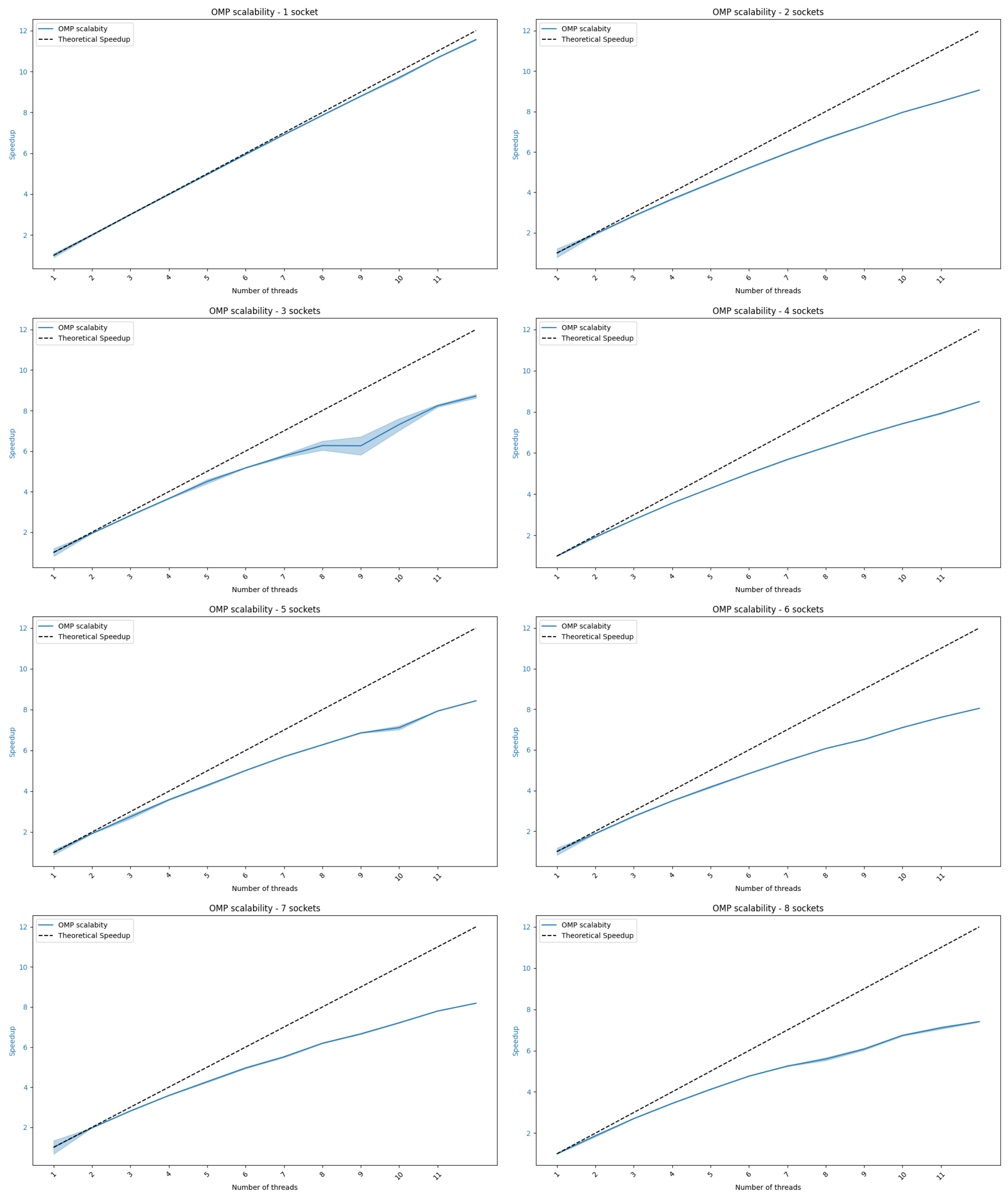
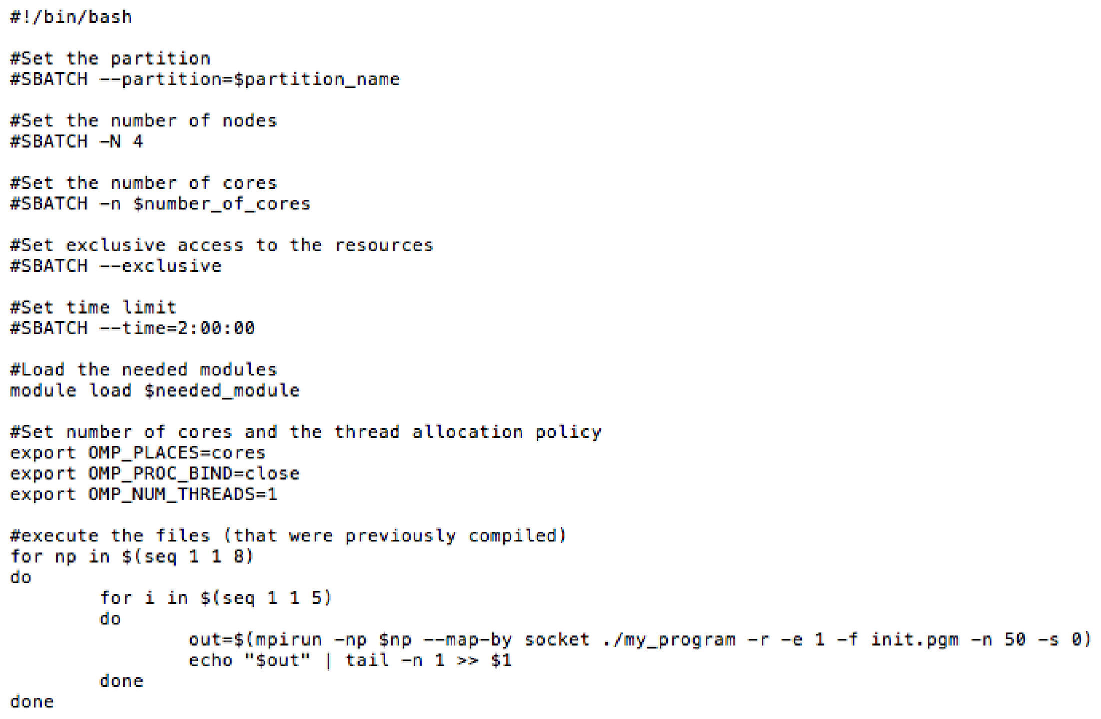


Figure 1.16: OMP scalability on THIN nodes

containing the same data in their caches. Consequently, an overhead to maintain cache coherence is introduced. It is worth noticing that, as the number of sockets grows, the increase in performance seems to slow down after a certain number of threads. This could probably be due to the intrinsic sequential fraction of the problem, as stated in Amdahl's Law.

Strong MPI Scalability

We aimed to measure the program's scalability as we increased the number of MPI processes. The test was conducted on both EPYC and THIN nodes, with one MPI Task allocated for each socket using the `--map-by socket` option. We fixed the number of OpenMP threads at 1 per MPI process and repeated measurements with 1 to 8 MPI processes.
The general structure of the job files that we used to perform the Strong MPI scalability can be seen below.



```
#!/bin/bash

#Set the partition
#SBATCH --partition=$partition_name

#Set the number of nodes
#SBATCH -N 4

#Set the number of cores
#SBATCH -n $number_of_cores

#Set exclusive access to the resources
#SBATCH --exclusive

#Set time limit
#SBATCH --time=2:00:00

#Load the needed modules
module load $needed_module

#Set number of cores and the thread allocation policy
export OMP_PLACES=cores
export OMP_PROC_BIND=close
export OMP_NUM_THREADS=1

#execute the files (that were previously compiled)
for np in $(seq 1 1 8)
do
        for i in $(seq 1 1 5)
        do
                out=$(mpirun -np $np --map-by socket ./my_program -r -e 1 -f init.pgm -n 50 -s 0)
                echo "$out" | tail -n 1 >> $1
        done
done
```

Figure 1.17: batch job for the Strong MPI scalability study

The job file is made of the following components:

- SLURM directives: These directives, preceded by '#' and therefore ignored by the shell, request the necessary resources. They include the partition ('–partition'), the number of nodes ('-N'), the number of cores ('-n'), the time limit ('–time'), and '–exclusive' for exclusive node access. The only variation when using EPYC or THIN nodes is the partition name. In both cases, we used one thread per socket, so the maximum number of cores, which differs for THIN and EPYC nodes, does not affect the setup.

- Module loading and threads affinity policy setting: the module system is addressed to load the needed openMPI library and the threads affinity policy is configured.
- Computation section: Two nested loops are employed; one to iterate through the number of mpi processes and another to repeat each measurement five times. Within the inner loop, the program is executed with the specified settings. The ‘mpirun‘ option `--map-by socket` ensures that each process is placed on a different socket, using a single thread on a single core of that socket.

We opted to use the same initial image, **init.pgm**, for all measurements. This image was precomputed and stored in the same directory as the job file for accessibility during runtime.
Additionally, we created **.csv** files to store the results, passing them as inline input arguments to the jobs.

The following results are obtained considering a matrix with 10000 rows and 50 iterations for the static evolution, while just 5 iterations for the ordered one are performed.

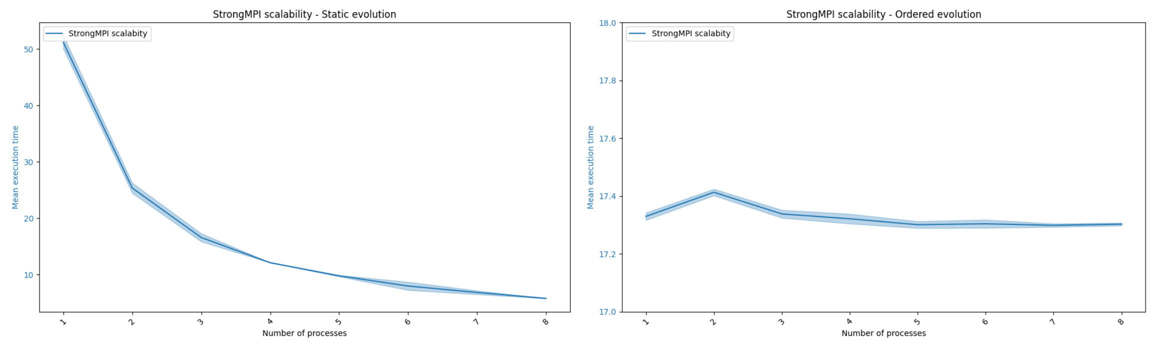


Figure 1.18: Strong MPI scalability on THIN nodes

As it can be seen from the graphs, when considering the static evolution, the run time decreases for increasing numbers of processes, as expected. This is due to the fact that the workload is distributed among a larger number of MPI processes. This does not happen for the ordered evolution in which the execution time remains constant and confirms the ”intrinsic” seriality of the evolution.

Weak MPI Scalability

We aimed to measure the program’s scalability as we increased the number of MPI processes keeping fixed the workload per MPI task. The test was conducted on both

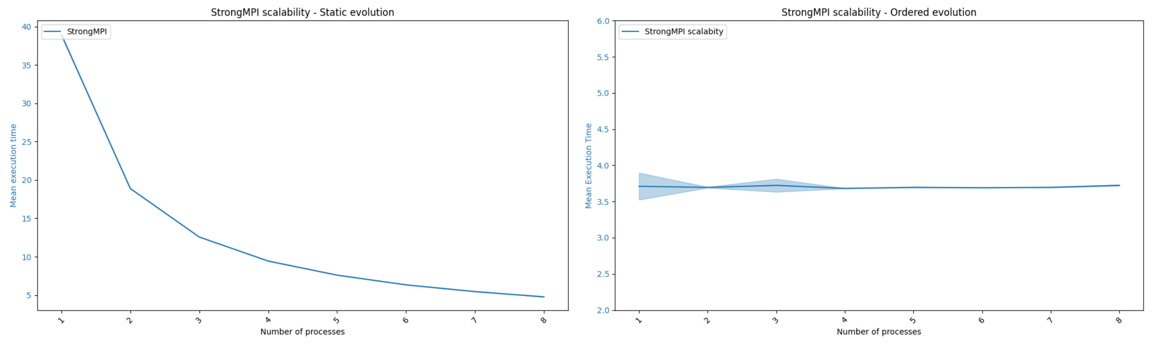
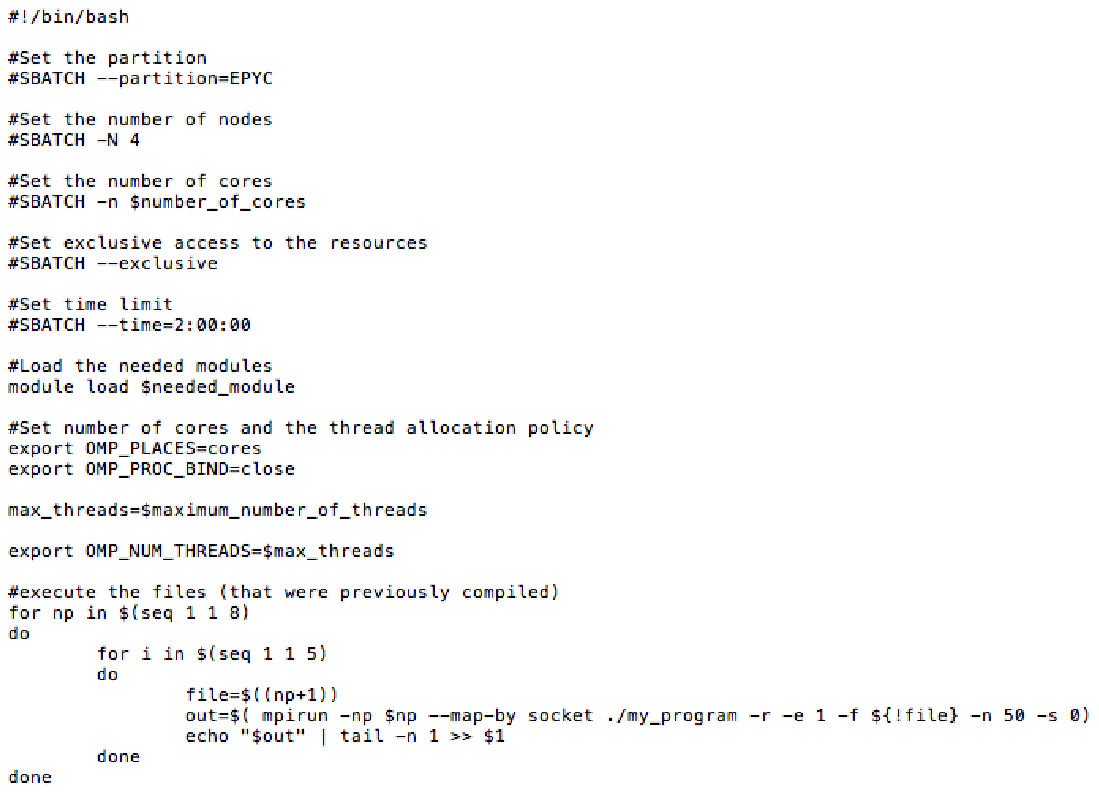


Figure 1.19: Strong MPI scalability on EPYC nodes

EPYC and THIN nodes, with one MPI Task allocated for each socket using the `--map-by socket` option. We fixed the number of OpenMP threads at the maximum number of cores per MPI process and repeated measurements with 1 to 8 MPI processes.
The general structure of the job files that we used to perform the Weak MPI scalability can be seen below.

The job file is made of the following components:

- SLURM directives: These directives, preceded by '#' and therefore ignored by the shell, request the necessary resources. They include the partition ('–partition'), the number of nodes ('-N'), the number of cores ('-n'), the time limit ('–time'), and '–exclusive' for exclusive node access. For EPYC nodes, the partition is named "EPYC," and the maximum number of threads is 64. THIN nodes use the "THIN" partition, and the maximum thread count is 12.
- Module loading and threads affinity policy setting: the module system is addressed to load the needed openMPI library and the threads affinity policy is configured.
- Computation section: Two nested loops are employed; one to iterate through the number of MPI processes and another to repeat each measurement five times. Within the inner loop, the program is executed with the specified settings, and the initial image to be read depends on the number of MPI processes being used. The 'mpirun' option `--map-by socket` ensures that each process is placed on a different socket, using a single thread on a single core of that socket.



```
#!/bin/bash

#Set the partition
#SBATCH --partition=EPYC

#Set the number of nodes
#SBATCH -N 4

#Set the number of cores
#SBATCH -n $number_of_cores

#Set exclusive access to the resources
#SBATCH --exclusive

#Set time limit
#SBATCH --time=2:00:00

#Load the needed modules
module load $needed_module

#Set number of cores and the thread allocation policy
export OMP_PLACES=cores
export OMP_PROC_BIND=close

max_threads=$maximum_number_of_threads

export OMP_NUM_THREADS=$max_threads

#execute the files (that were previously compiled)
for np in $(seq 1 1 8)
do
        for i in $(seq 1 1 5)
        do
                file=$((np+1))
                out=$( mpirun -np $np --map-by socket ./my_program -r -e 1 -f ${!file} -n 50 -s 0)
                echo "$out" | tail -n 1 >> $1
        done
done
```

Figure 1.20: batch job for the Weak MPI scalability study

We used 8 initial images that had different sizes in order to keep the workload fixed. The images were precomputed and stored in the same directory as the job file for accessibility during runtime.
In the table below, we report the size of the matrices for each number of processes.

$(\#rows)$	p
10000	1
14142	2
17321	3
20000	4
22361	5
24495	6
26458	7
28284	8

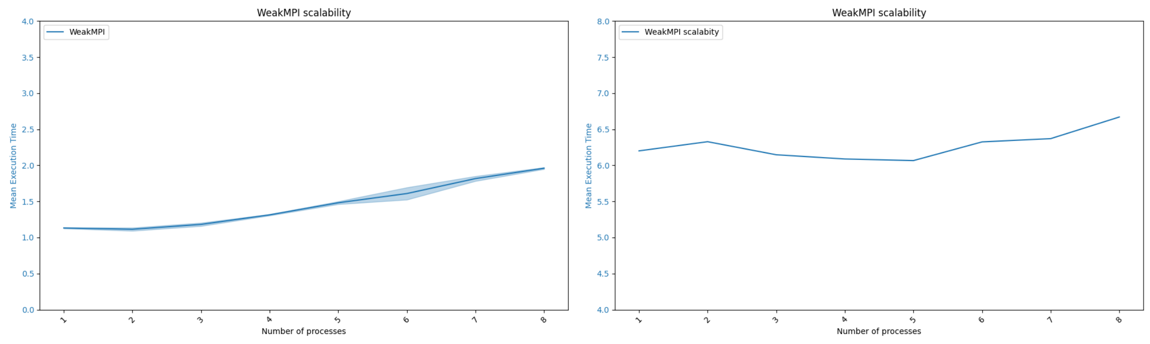
In order to keep the workload constant, they have been obtained considering the following formula

$$x^2_{MPI=1} = x_{MPI=p} \times \frac{x_{MPI=p}}{p}$$

where $x^2_{MPI=1}$ is the full size of the matrix ($\#rows \times \#columns$) when only 1 MPI process is considered and $x_{MPI=p}$ is one dimension of the matrix when p processes are considered. Since we are dealing with square matrices, $x_{MPI=p} = \#rows = \#columns$

Additionally, as always, we created **.csv** files to store the results, passing them as inline input arguments to the jobs.

Performance is measured considering 50 iterations for the static evolution and 5 iterations for the ordered one. As expected, and as evident from the provided plots, the computation time remains nearly constant for both EPYC and THIN nodes, suggesting that the program is able to handle a larger amount of data efficiently by increasing the number of processes.



(a) Weak MPI scalability on EPYC node (b) Weak MPI scalability on THIN nodes

Figure 1.21: Weak MPI Scalability

1.5 Further improvements

The overall program performance is reasonably satisfactory; nevertheless, there's room for further improvements through some adjustments. The most significant potential gain lies in parallelizing read/write operations using `MPI I/O`. We are fairly optimistic about the positive impact this could have on static evolution performance. However, we have reservations about whether it will significantly benefit the ordered evolution, as these operations don't appear to be particularly resource-intensive compared to others. Nevertheless, we believe this approach could optimize memory usage by eliminating the need to initialize the entire matrix on the master process.

Chapter 2

Exercise 2

2.1 Introduction

The goal of this exercise was to compare the performance of three math libraries available on HPC: **MKL**, **OpenBLAS** and **BLIS**. The performance had to be evaluated focusing the comparison on the level 3 BLAS function called gemm. Such a function was available in both single and double-point precision.

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS performs scalar, vector and vector-vector operations, the Level 2 BLAS performs matrix-vector operations, and the Level 3 BLAS performs matrix-matrix operations.

The BLAS level 3 contains matrix-matrix operations, including the "general matrix multiplication" (gemm), which has the following form

$$C \leftarrow \alpha AB + \beta C$$

For the mentioned exercise the specifications were $\alpha = 1$, $\beta = 0$ and only square matrices were considered. The comparison of the three different libraries had to consider different variables:

- **Different architectures**: either THIN nodes or EPYC nodes
- **Different thread allocation policies**: the choice of the considered policies was free, so spread and close cores were considered
- **Different precision**: either single-point or double-point precision

2.2 Methodology

In order to compare the performance of the different libraries two different scalability studies were required:

- **Size scaling**
- **Core scaling**

Different measures had to be performed, in order to consider all the possible combinations (nodes/ cores/ precision), and then the results, together with the Theoretical Peak Performance, had to be discussed. In order to perform this study two files were provided:

- the code containing the gemm function, which was used to perform the matrix-matrix multiplication with either single or double precision and with either one of the different libraries
- a `Makefile` which could be used to compile the gemm code

MKL and OpenBLAS libraries were already available on ORFEO's module's list, while the BLIS library had to be downloaded and compiled. The performances were measured by keeping track of the elapsed time and the number of GFLOPs required to complete the gemm operation with the considered settings.

2.3 Implementation

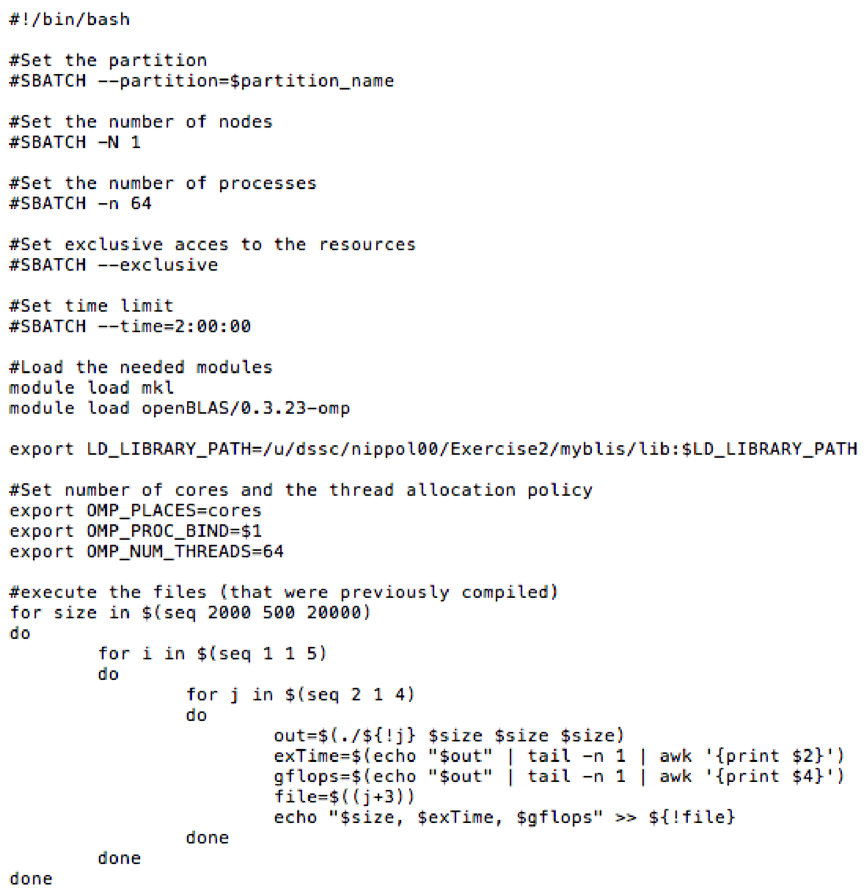
As mentioned earlier, two pieces of code were supplied: the `gemm.c` code and the `Makefile` designed for compiling `gemm.c`. These files remained unchanged, and all the necessary operations for conducting measurements were outlined in separate bash scripts. To initiate the measurement process, the initial step involved compiling the `gemm.c` file using the provided `Makefile`. Note that, before compilation, the lines referring to the BLIS library were uncommented.

Initially, we allocated the required resources and proceeded to compile the file twice. During the first compilation, it generated executable files `sgemm_blis.x`, `sgemm_oblas.x`, and `sgemm_mkl.x`, which were subsequently employed for single-precision measurements. Then, we modified the `Makefile` by replacing the command `DUSE_FLOAT` with `DUSE_DOUBLE`, resulting in the creation of executables `dgemm_blis.x`, `dgemm_oblas.x`, and `dgemm_mkl.x` for double-precision operations. Once all the executable files were available, we generated the corresponding output files (**.csv**) using the simple nano editor.

The name of the output files followed this structure:

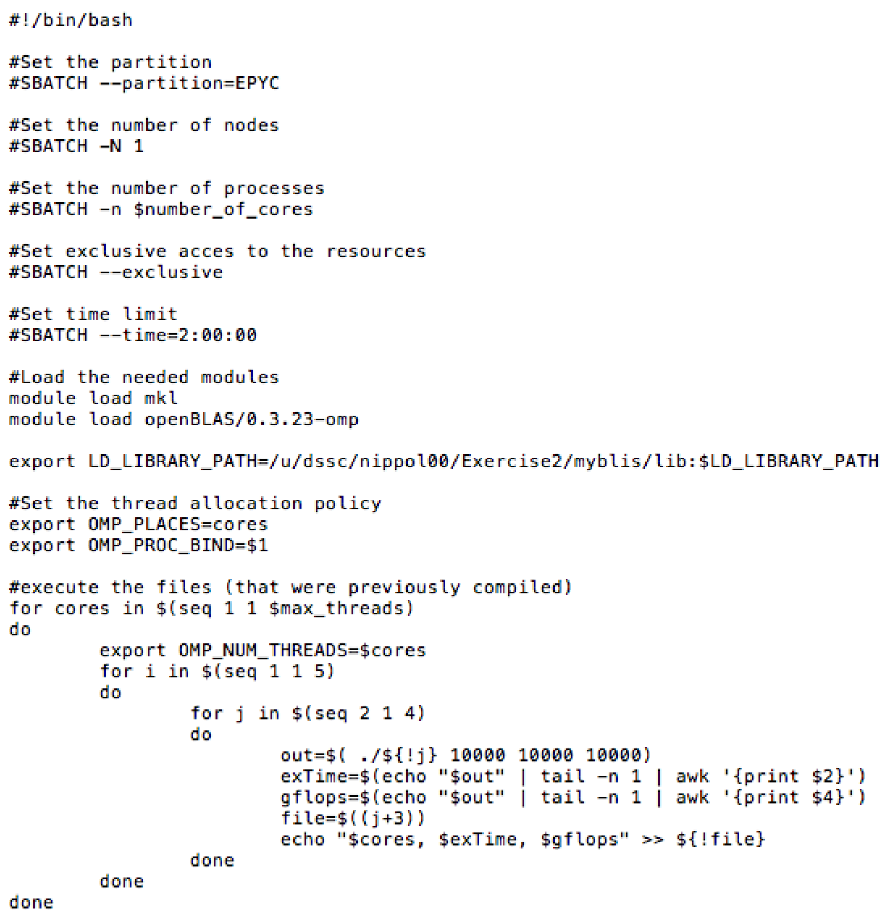
$varying_parameter-$library_name-$partition_name-$threadpolicy-$precision.csv$

So, for example, the file `size-blis-e-c-f.csv` refers to the file containing the measures taken for the execution of the *sgemm_blis.x* file for increasing matrix size and fixed number of cores, on EPYC nodes with close policy. Each of the output files contained a single line used for specifying the column names, labeled as: $varying_parameter (either **size** or **cores**), **Execution Time**, and **GFLOPs**. Subsequently, four distinct batch scripts were written, with slight variations, related to the desired partition. Two of these scripts were designed for conducting operations with increasing matrix size, while the other two were tailored for operations involving an increasing number of cores. The general structure of the two batch files is the following:



```
#!/bin/bash

#Set the partition
#SBATCH --partition=$partition_name

#Set the number of nodes
#SBATCH -N 1

#Set the number of processes
#SBATCH -n 64

#Set exclusive acces to the resources
#SBATCH --exclusive

#Set time limit
#SBATCH --time=2:00:00

#Load the needed modules
module load mkl
module load openBLAS/0.3.23-omp

export LD_LIBRARY_PATH=/u/dssc/nippol00/Exercise2/myblis/lib:$LD_LIBRARY_PATH

#Set number of cores and the thread allocation policy
export OMP_PLACES=cores
export OMP_PROC_BIND=$1
export OMP_NUM_THREADS=64

#execute the files (that were previously compiled)
for size in $(seq 2000 500 20000)
do
        for i in $(seq 1 1 5)
        do
                for j in $(seq 2 1 4)
                do
                        out=$(./${!j} $size $size $size)
                        exTime=$(echo "$out" | tail -n 1 | awk '{print $2}')
                        gflops=$(echo "$out" | tail -n 1 | awk '{print $4}')
                        file=$((j+3))
                        echo "$size, $exTime, $gflops" >> ${!file}
                done
        done
done
```

Figure 2.1: batch job for the Size Scaling



```
#!/bin/bash

#Set the partition
#SBATCH --partition=EPYC

#Set the number of nodes
#SBATCH -N 1

#Set the number of processes
#SBATCH -n $number_of_cores

#Set exclusive acces to the resources
#SBATCH --exclusive

#Set time limit
#SBATCH --time=2:00:00

#Load the needed modules
module load mkl
module load openBLAS/0.3.23-omp

export LD_LIBRARY_PATH=/u/dssc/nippol00/Exercise2/myblis/lib:$LD_LIBRARY_PATH

#Set the thread allocation policy
export OMP_PLACES=cores
export OMP_PROC_BIND=$1

#execute the files (that were previously compiled)
for cores in $(seq 1 1 $max_threads)
do
        export OMP_NUM_THREADS=$cores
        for i in $(seq 1 1 5)
        do
                for j in $(seq 2 1 4)
                do
                        out=$( ./${!j} 10000 10000 10000)
                        exTime=$(echo "$out" | tail -n 1 | awk '{print $2}')
                        gflops=$(echo "$out" | tail -n 1 | awk '{print $4}')
                        file=$((j+3))
                        echo "$cores, $exTime, $gflops" >> ${!file}
                done
        done
done
```

Figure 2.2: batch job for the Core Scaling

The job file is made of the following components:

- SLURM directives: These directives, preceded by '#' and therefore ignored by the shell, request the necessary resources. They include the partition ('–partition'), the number of nodes ('-N') which is fixed to 1, the number of cores ('-n'), the time limit ('–time'), and '–exclusive' for exclusive node access. For EPYC nodes, the partition is named "EPYC," and the maximum number of threads is 64. THIN nodes use the "THIN" partition, and the maximum thread count is 12.
- Module loading and threads affinity policy setting: the module system is addressed to load the needed libraries and the threads affinity policy is configured.
- Computation section: Three nested loops are employed; one to iterate through the varying parameter (either size or number of `OpenMP threads`), one to

repeat each measurement five times and one to perform the measure for each of the considered libraries. Within the inner loop, the program to be executed is passed to the job as an inline argument, as well as the **.csv** file where the result has to be written. After the program has been executed the needed data is extracted from the output and written to the corresponding **.csv** file.

The measurement process was performed as follows. Initially, the BLIS library required configuration and installation, a step that was iterated each time a new node partition and core count configuration was taken under consideration. To be more specific, we started with EPYC nodes equipped with 64 cores, configuring the BLIS library accordingly, and executing all measurements requiring these specifications. Afterward, the library was uninstalled and reinstalled with varying specifications (EPYC 128 cores, THIN 12 cores, THIN 24 cores), and the measurements were replicated. The rationale behind recompiling the BLIS library with distinct settings comes from its ability to generate optimized code tailored to the underlying hardware architecture during compilation. Consequently, recompiling with different settings for different architectures was deemed an essential step in the process. To perform the actual measures (with the different configurations), the sbatch script was called. In the call, the needed executable and output files were passed each time. This approach aimed to maintain the batch script's generality and reusability. Once this procedure concluded, all the results of the measurements were available for the final analysis.

2.4 Results and conclusions

The results had to be confronted and compared to gain perspective on the performance of the different math libraries. In order to obtain a deeper analysis, both EPYC and THIN nodes of the Orfeo cluster have been used. Results, coming from **.csv** outputs of the above-described sbatch scripts, have been plotted using the well-known `Matplotlib` library in Python. The Theoretical Peak Performance also had to be taken into consideration, so a brief explanation is provided below.

2.4.1 Theoretical Peak Performance

The Theoretical Peak Performance (**TPP**) is defined as the maximum number of floating point operations per second that can be performed on a given hardware platform. It is given by the following formula

$$TPP = (\text{clock rate in GHz}) \times (\#cores) \times (\#\text{FLOPS per core per cycle})$$

2.4.2 Size Scaling

In this section, we measure the scalability of the different libraries over increasing sizes of the matrix for a fixed number of cores - 64 for EPYC nodes and 12 for THIN nodes, set using `OMP_PLACES=cores` and `OMP_NUM_THREADS=64 or OMP_NUM_THREADS=12`. In order to achieve the goal, we increased the size of the matrix from 2000×2000 to 20000×20000, by steps of 500. Measures are taken 5 times for each size and the mean number of GFLOPs, with related standard deviation, is plotted for each library, both for single and double precision. Moreover, the two different threads affinity policies have been tested, by properly modifiyng the `OMP_PROC_BIND` command in the above-described sbatch files.

EPYC - Double precision

Before plotting the results, we compare them to the Theoretical Peak Performance. EPYC nodes that can be found in the Orfeo cluster - AMD EPYC 7H12 64-Core - have a base frequency of $2.6GHz$ [3] and can deliver up to 16 double-precision FLOPS per cycle per core [4]. This implies that the Theoretical Peak Performance is equal to

$$TPP = 2.6GHz \times 64 \times 16 = 2662.4GFLOPs$$

Here below, we report the real performances of each library:

	`OMP_PROC_BIND=close`	Size	`OMP_PROC_BIND=spread`	Size
OpenBlas	1085.83	16500	1045.32	17000
Blis	1055.51	3500	1136.15	17000
MKL	823.55	15500	838.05	15500

Table 2.1: Sustained performances for different policies and related size

As evident from the table, none of the libraries managed to attain the Theoretical Peak Performance. Among them, the BLIS library shows the highest number of GFLOPs per second, especially when taking into account the spread thread policy.

For a more comprehensive comparison, performance metrics are also plotted in the following graphs.

In general, the BLIS library emerges as the best-performing one. Nevertheless, it's worth noting that regardless of the specified policy, the BLIS library experiences a significant performance drop when dealing with matrices of size 11,000. This drop is likely attributed to cache-related issues. For larger matrix sizes, both BLIS and OpenBlas exhibit fairly similar performances.

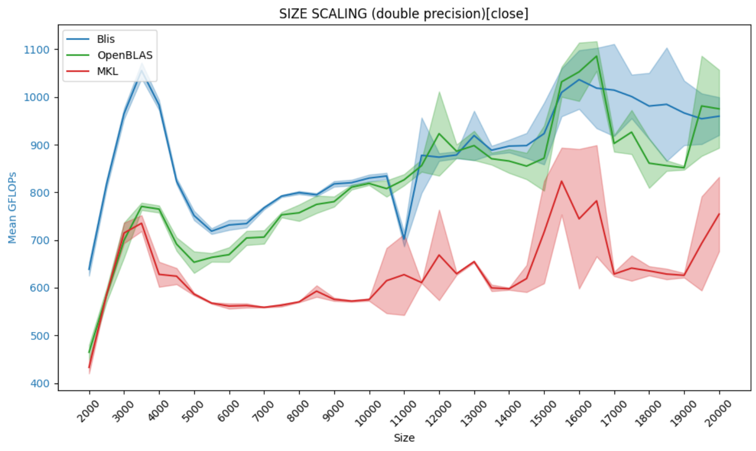


Figure 2.3: Comparison for double precision and close policy.

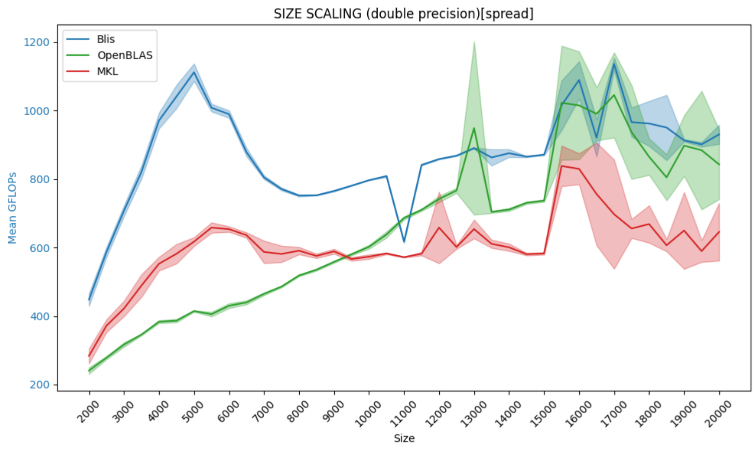


Figure 2.4: Comparison for double precision and spread policy.

EPYC - Float precision

For single precision operations, the Theoretical Peak performance is set at 5324.8 GFLOPs per second, taking into account that an EPYC node can potentially deliver up to 32 floating-point operations. Nevertheless, none of the libraries were able to attain this level of performance, suggesting that larger matrices may be required to reach this threshold. The table below shows the best actual performance achieved by each library, taking into consideration the two thread allocation policies.

Once again, the BLIS library stands out by delivering the highest number of GFLOPs per second, particularly when combined with the spread policy for thread affinity. On the other hand, when the close policy is considered, OpenBlas outperforms the others for matrix sizes exceeding 7000, while for smaller matrices, the BLIS library

	`OMP_PROC_BIND=close`	Size	`OMP_PROC_BIND=spread`	Size
OpenBlas	2402.38	18500	2155.72	20000
Blis	2486.98	5500	3098.77	6500
MKL	2001.27	5000	2335.65	7000

Table 2.2: Sustained performances for different policies and related size

is the preferred choice. However, this fact changes when adopting the spread policy, where OpenBlas performs even worse than MKL for matrices smaller than 11000 in size. Overall performances can be observed in the accompanying plots below.

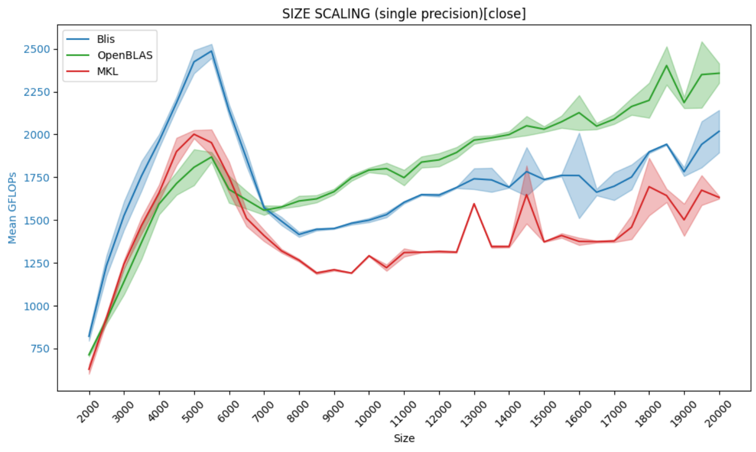


Figure 2.5: Comparison for float precision and close policy.

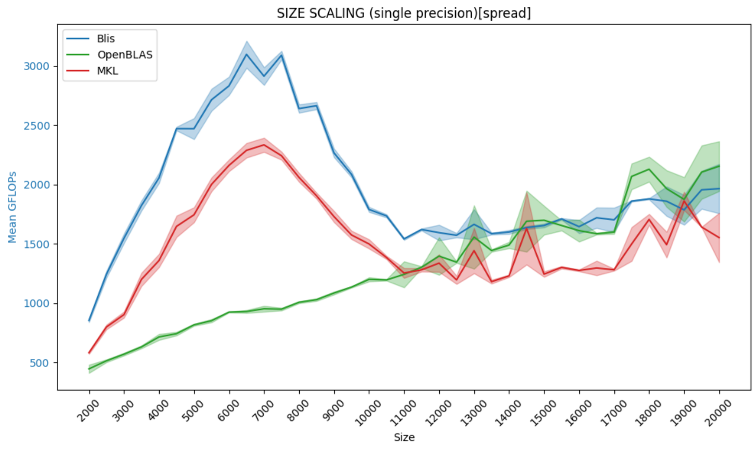


Figure 2.6: Comparison for float precision and spread policy.

Based on the outcomes presented above, whether in double or single precision, the BLIS library consistently emerges as the top-performer on EPYC nodes, exhibiting the most favorable overall performance.

THIN - Double precision

The same analysis has been performed on the Intel THIN nodes of the Orfeo cluster. Considering the case of double-precision, the Theoretical Peak Performance of an Intel Xeon Gold 6126, with a base frequency of $2.6GHz$ is equal to $998.4GFLOPs$ - assuming 32 operations per cycle per core.

	`OMP_PROC_BIND=close`	Size	`OMP_PROC_BIND=spread`	Size
OpenBlas	786.09	6500	811.01	9000
Blis	719.89	11000	756.05	14000
MKL	805.44	6000	829.81	18000

Table 2.3: Sustained performances for different policies and related size

In contrast to EPYC nodes, peak performances are attained with smaller matrix sizes, and they approach the theoretical maximum. In fact, the highest recorded number of GFLOPS is 829.81, achieved when using the MKL library in conjunction with the spread allocation policy.

As can be seen from the plot below, the number of GFLOPs of each library is quite similar for matrices whose size is larger than 5500.

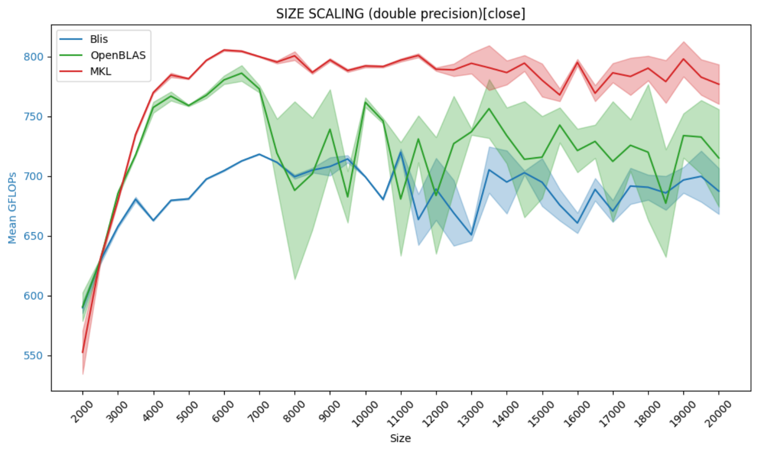


Figure 2.7: Comparison for double precision and close policy.

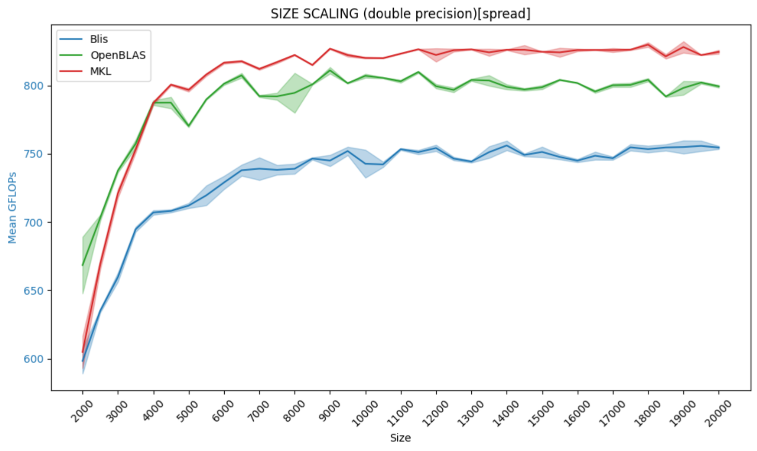


Figure 2.8: Comparison for double precision and spread policy.

It appears quite clear that, for increasing sizes of the matrix, the number of GFLOPs for each library is more stable with respect to the one performed on the EPYC nodes, given that, starting from a certain relatively small size, no particular peaks can be spotted in the graphs - except for OpenBLAS, which shows a high variance in the case of close policy. Furthermore, conversely, from what happened on EPYC nodes, MKL is the best-performing library for both types of policy. This is due to the fact that MKL is an Intel library, specifically tailored and optimized on Intel products.

THIN - Float precision

Lastly, the same measures have been taken considering single precision. The Theoretical Peak Performance, in this case, is twice the previous one - due to the change in precision - and so it is equal to 1996.8 GFLOPs per second. Once again, we present a table, supplied with plots, to show overall performance.

	`OMP_PROC_BIND=close`	Size	`OMP_PROC_BIND=spread`	Size
OpenBlas	1602.25	11500	1640.33	11500
Blis	1352.34	20000	1423.48	11500
MKL	1601.93	15500	1646.79	15500

Table 2.4: Sustained performances for different policies and related size

It is worth noticing how, in the case of float precision, MKL and OpenBlas perform a similar number of GFLOPs per second, regardless of the considered policy.

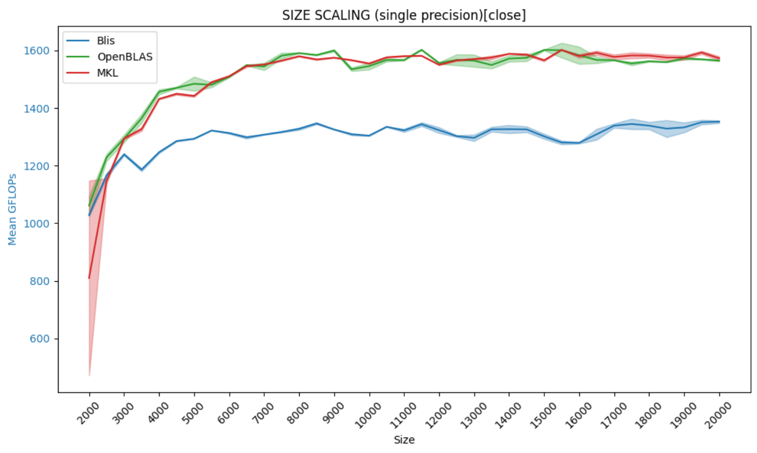


Figure 2.9: Comparison for float precision and close policy.

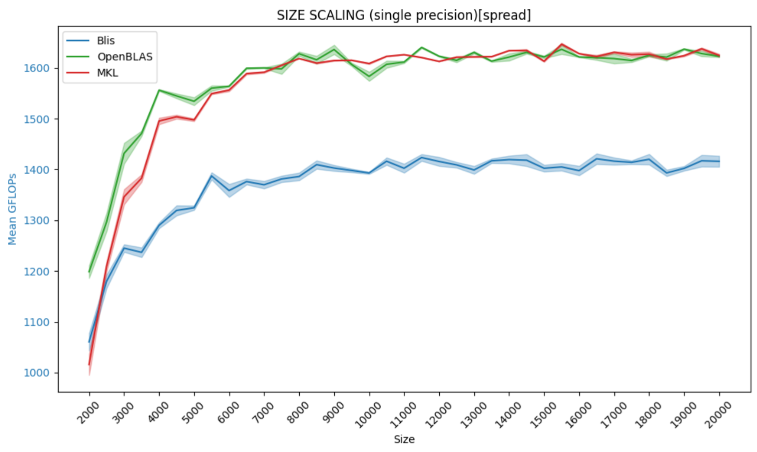


Figure 2.10: Comparison for float precision and spread policy.

2.4.3 Core Scaling

In this section, a scaling study over the increasing number of cores is performed. We fixed the size of the matrix to 10000 and let the number of cores vary from 1 to the maximum number available on the considered architecture - 128 for EPYC nodes and 24 for THIN nodes. The analysis is performed on both types of nodes and for both precisions.

EPYC - Double precision

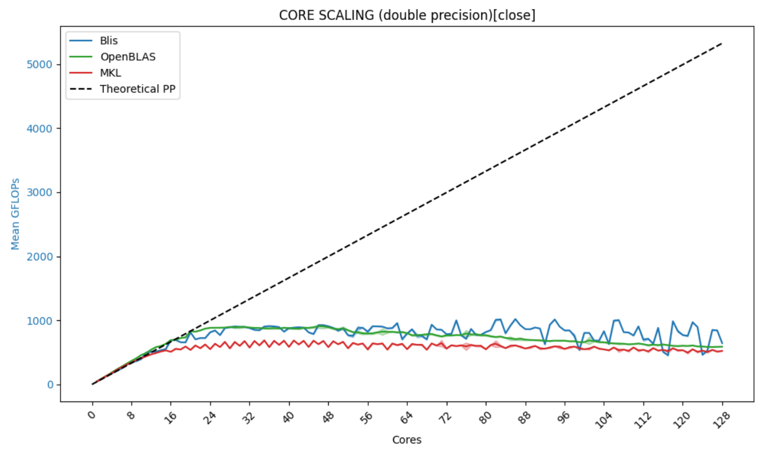


Figure 2.11: Comparison for double precision and close policy.

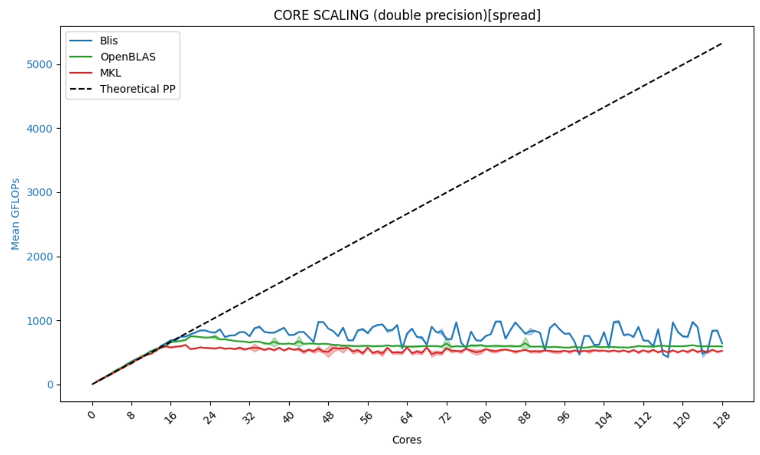


Figure 2.12: Comparison for double precision and spread policy.

EPYC - Float precision

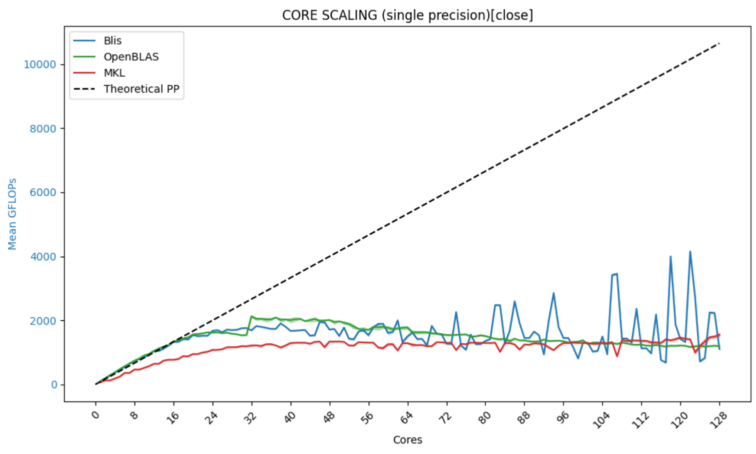


Figure 2.13: Comparison for single precision and close policy.

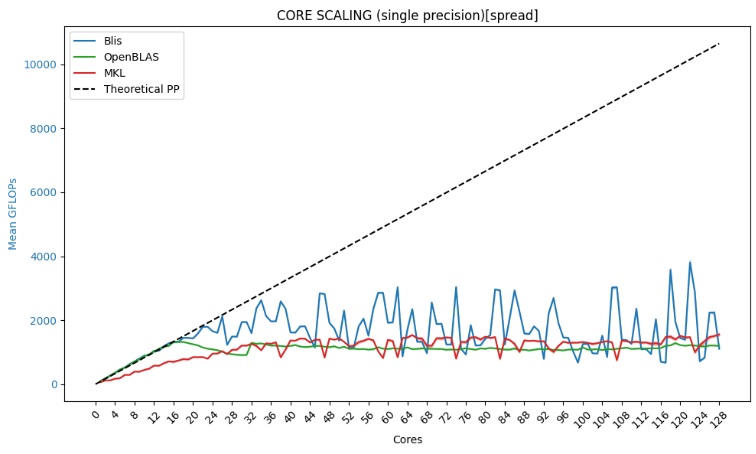


Figure 2.14: Comparison for single precision and spread policy.

For both float and double precision, none of the libraries seem to show good performances. Overall, the BLIS library emerges as the top performer, as for size scaling. When dealing with double precision and close policy, BLIS and OpenBLAS exhibit nearly identical performance trends up to 80 cores. Conversely, MKL shows the lowest performance among all libraries, no matter the specified policy. The situation varies for single precision operations and spread policy, where OpenBlas becomes the least favorable choice.

One possible reason for this general low performance could be that the size of the considered matrix is too small to allow an efficient use of cache in the case of (saturated) EPYC nodes. The idea is that each core has its own cache. When the number of processes increases, each core will work on a smaller number of data. Simultaneously, more processes will need to access data stored in other cores' cache so maintaining cache coherence will take a longer time, resulting in a lower number of GFLOPs. Some evidence in support of our hypothesis is the fact that up to 20 cores, the performances of all libraries closely align with theoretical expectations. Another plausible reason could be the fact that the matrix is initialized in serial, so all data are stored in the NUMA region of a single core. This suggests that as the number of cores increases, cores located farther from that NUMA region experience longer access times, leading to a decrease in the number of GFLOPs.

THIN - Double precision

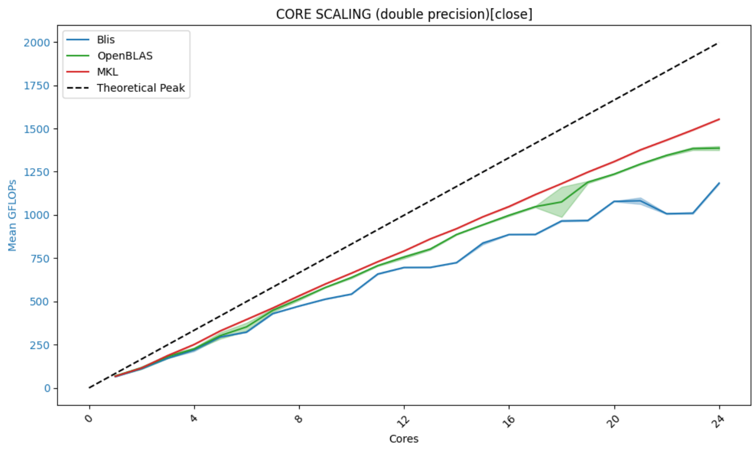


Figure 2.15: Comparison for double precision and close policy.

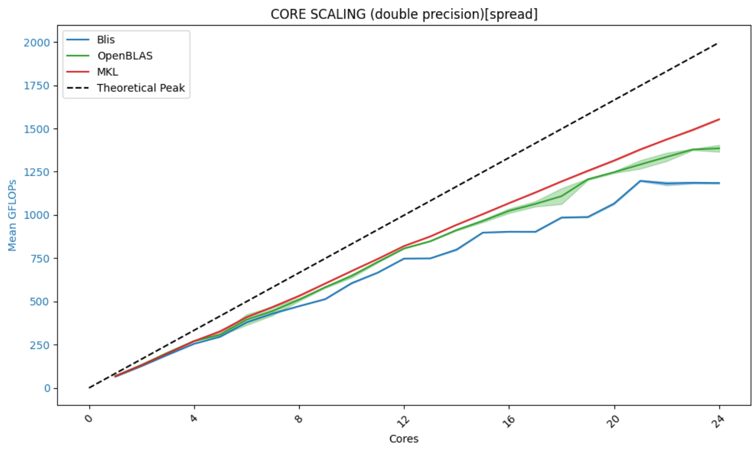


Figure 2.16: Comparison for double precision and spread policy.

THIN - Float precision

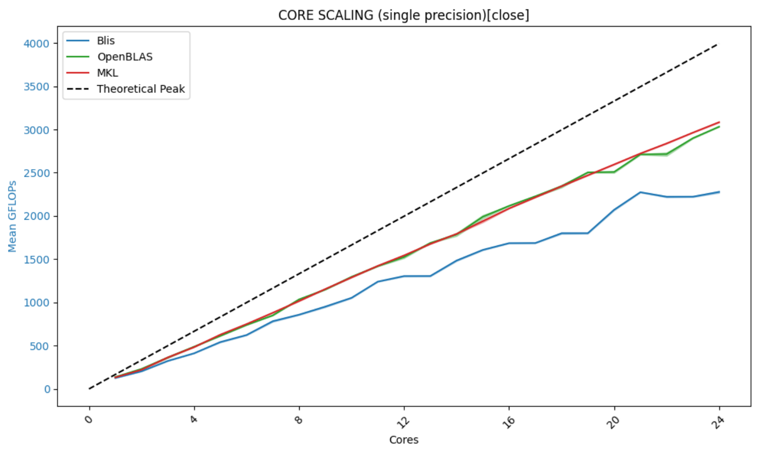


Figure 2.17: Comparison for single precision and close policy.

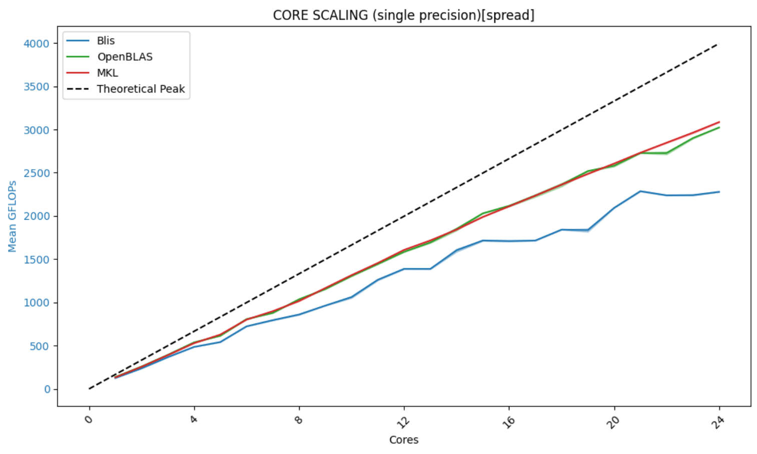


Figure 2.18: Comparison for single precision and spread policy.

Results on THIN nodes are globally better and seem to confirm our previous hypothesis. In fact, Intel THIN nodes have a smaller number of available cores, which allows a better division of the workload and a faster access time, since cores are not as distant as in the case of EPYC nodes. Considering all possible combinations of policies and precision, BLIS library is the worst-performing one. Regardless of the specified policy, MKL is slightly better than OpenBlas for double-precision operations, but the two are almost equivalent when dealing with float-precision.

A final point to consider is that, when we keep the desired partition and precision constant, it's worth noting that performance remains remarkably similar for a high number of cores, irrespective of the selected policy. This is completely reasonable since, when the nodes are saturated, performance will not be affected by the specific threading arrangement.

Bibliography

[1] Conway's game of life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life. Accessed: 2022-09-22.

[2] Github repository of the fhpc course, università di trieste 2022/2023. https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/tree/main/Assignment/exercise1. Accessed: 2022-09-22.

[3] Amd epyc™ 7h12. https://www.amd.com/en/product/9131. Accessed: 2022-09-22.

[4] Detailed specifications of the amd epyc "rome" cpus. https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-amd-epyc-rome-cpus/. Accessed: 2022-09-22.