University of Trieste

Foundations of High Performance Computing

Final project

Game of Life and GEMM benchmarking

Authors:
Michele Alessi
Marco Carollo

Teachers:
Stefano Cozzini
Luca Tornatore

Contents

1 Exercise 1 **2**
- 1.1 Introduction . . . 2
- 1.2 Methodology . . . 3
  - 1.2.1 Grid operations: initialization, reading and writing . . . 3
  - 1.2.2 Ordered evolution . . . 4
  - 1.2.3 Static evolution . . . 4
  - 1.2.4 Wave evolution . . . 5
- 1.3 Implementation . . . 8
  - 1.3.1 Grid operations: initialization, reading and writing . . . 8
  - 1.3.2 Ordered evolution . . . 11
  - 1.3.3 Static evolution . . . 13
  - 1.3.4 Wave evolution . . . 15
- 1.4 Results and Discussion . . . 20
  - 1.4.1 Writing and random initialization . . . 20
  - 1.4.2 Static evolution . . . 22
  - 1.4.3 Ordered evolution . . . 25
  - 1.4.4 Wave evolution . . . 26
- 1.5 Conclusions . . . 29

2 GEMM benchmark **31**
- 2.1 Introduction to the problem . . . 31
  - 2.1.1 The provided implementation . . . 31
  - 2.1.2 Goals . . . 31
- 2.2 Scalability over matrix size . . . 32
  - 2.2.1 Analysis on THIN node . . . 33
  - 2.2.2 Analysis on EPYC node . . . 36
  - 2.2.3 Comparison between the two architectures . . . 39
- 2.3 Scalability over the number of cores . . . 39
  - 2.3.1 Analysis on THIN node . . . 40
  - 2.3.2 Analysis on EPYC node . . . 41
  - 2.3.3 Comparison between the two architectures . . . 43
- 2.4 Conclusions . . . 45

1 Exercise 1

1.1 Introduction

This project focuses on the development of a parallelized version of a modified Conway's "Game of Life," a cellular automaton operating on an infinite 2D grid. Refer to sources [4] and [2] for foundational insights into the chosen variant. The primary objective is to explore the application of parallel computing techniques within the context of cellular automata.

The discrete nature of the world dictates that a cell's neighbors comprise the eight most adjacent cells, sharing an edge on the grid representation. The computational challenge lies in efficiently parallelizing the evolution of these neighboring cells, considering the inherent dependencies and constraints of the cellular automaton model.

Three distinct evolution methods have been implemented and are summarized as follows:

- Ordered Evolution: With this approach, the grid is traversed systematically, starting from the first entry and proceeding *sequentially* to the last. Each cell immediately inspects the state of its adjacent neighbors and evolves in an ordered manner.
- Static Evolution: In this method, the system undergoes a freeze at each time step. Each cell then assesses *at the same time* the status of its neighboring cells based on the frozen system and evolves accordingly.
- Wave Evolution: A unique approach where the ordered evolution doesn't always initiate from the same point. Instead, it commences from a random position and propagates in all directions as a square wave, as illustrated in Fig.4

The objective of this exercise involves implementing a parallel version of the game utilizing a hybrid approach combining OpenMP and MPI. The goal is to analyze the scalability of each evolution method across OpenMP and MPI: so respectively with increasing numbers of threads and processes.

1.2 Methodology

The discrete 2D world of the game is represented by a grid, with each cell functioning as the smallest unit. These cells are analogous to points within a system of integer coordinates (i, j), where i is the "row" number, while j is the "column". However, this grid has been implemented as a 1D array. The game's evolution introduces various properties, making it a subject of interest in the broader field of cellular automata. The focus here, however, is on the technical aspects of parallelizing the computation, leaving detailed exploration to individual discretion.
The grid is considered to be infinite, in the sense that each side is actually connected to the opposite side, in what is topologically a torus (see Fig. 2).

1.2.1 Grid operations: initialization, reading and writing

A fundamental step in exploiting parallelization comes directly from the handling of the operations on the grid and from the implementation of the grid itself.

Domain decomposition When parallelizing the grid computation over a multiple number of processes, enabling domain decomposition involves dividing the grid into distinct sections and assigning each segment to individual processes within the parallel environment. The chosen approach involves partitioning the grid into rows and allocating approximately $\lfloor k/\text{size} \rfloor$ number of rows to each process (Fig.1), where "k" represents the number of rows and "size" is the total number of processes. By doing so, each process operates on and stores a relatively equal fraction of the grid, promoting workload balance among the parallel entities.

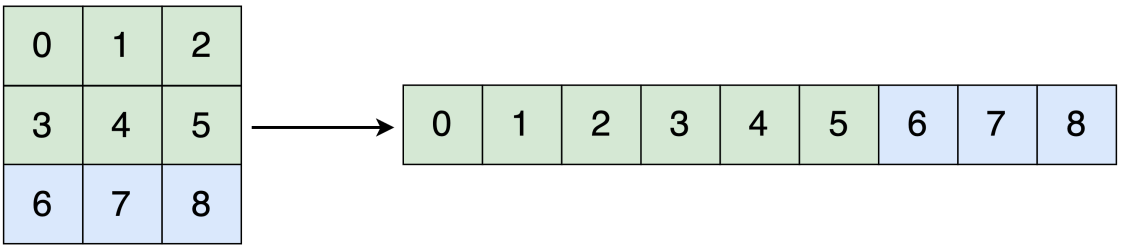


Figure 1: A 3×3 grid partition of rows with 2 processes.

This strategy aims to distribute computational tasks evenly across processes, optimizing resource utilization and enhancing parallel efficiency by reducing potential load imbalances. It also enables an efficient storing of the grid, distributing its entries across the memory of each process.

Initialization The grid is divided among the processes, and each MPI process is responsible for generating and managing its designated segment of the grid. Parallelization is achieved using a combination of MPI for inter-process communication and OpenMP threads within each process.

Writing The chosen approach requires that processes exclusively write their segment of rows to a file. Contrary to multiple files for each process, this method streamlines the storage by allowing

individual processes to directly contribute their data to the file without post-processing. Furthermore, it ensures ease in reading operations, avoiding complexities associated with merging multiple files or handling discrepancies when varying the number of processes for reading.

Reading Thanks to the previous strategy, a single file contains contributions from each process, simplifying the reading process by granting all processes access to the complete data. Leveraging MPI I/O functions, every process writes its assigned segment using distinct file pointers. Similarly, during reading operations, each process independently accesses its specific section within the file.

1.2.2 Ordered evolution

In the realm of cellular automata, the concept of ordered evolution signifies a systematic approach to updating cell states in a row-major order. Traditionally expressed through code snippets like:

```
for (int i = 0; i < k_i; i++)
    for (int j = 0; j < k_j; i++)
        upgrade_cell(i, j); // calculate the status of neighb. cells and update
```

This approach processes cells sequentially, row by row, where the status of each cell is upgraded before moving on to the next. While this method aligns with the logical structure of grid-based data, it introduces a notable challenge – the spurious signal generated by changing a cell's status, impacting the fate of its adjacent counterparts, as depicted in Figure 2. This peculiarity of the ordered evolution is what distinguishes it, for example, from the static evolution.

Referred to as "ordered evolution", this method inherently exhibits a *serial* nature due to the dependencies dictated by its definition. The sequential progression through rows implies that the status change in one cell influences the calculation for adjacent cells. Consequently, this imposes a strict order of execution, limiting opportunities for parallelism.

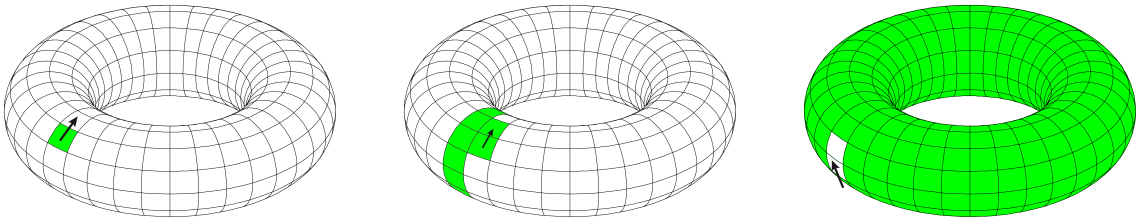


Figure 2: Ordered evolution visualized in the grid, emphasizing the torus structure. As we can see, the first cell is the only one indifferent to the evolution of the others. As the evolution progresses, the to-be-updated cell is influenced by an increasing number of cells.

1.2.3 Static evolution

The usual approach to Game Of Life is known as "static evolution". In this computational paradigm, the evaluation and update of each cell's status are decoupled into two distinct phases.
Initially, the status of all cells, typically indicating the count of alive neighboring cells for each, is computed without altering the grid. This evaluation step is independent of updating the cell states and serves to analyze the neighborhood of each cell, determining how many neighboring cells are alive.

Once this evaluation phase is completed, the system is momentarily frozen, and subsequently, the status of each cell is updated based on the evaluated neighbor counts. This *two-step process*—evaluating all cell neighborhoods before updating their statuses—defines the concept of "static evolution" in the Game of Life, enabling a clear separation between the analysis of cell neighborhoods and the subsequent adjustment of their states within the grid.
This distinction in the phases of static evolution has been obtained by providing a second grid of equal dimension to the computing agents. Each process/thread will refer to the actual grid for computing the information about the neighbors, while it will initialize the new grid copy with the cell values of the following step. At the end of this step, the new copy will be set as the current grid.

The static evolution represents a clear example in which parallelization can be exploited, given the separation between the evaluation and the updating of the grid: there's no notion, as there was on the ordered update, of seriality when updating the grid.

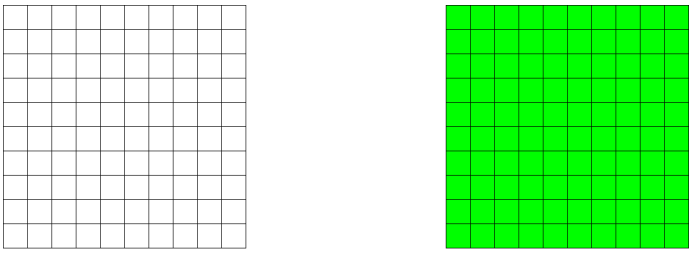
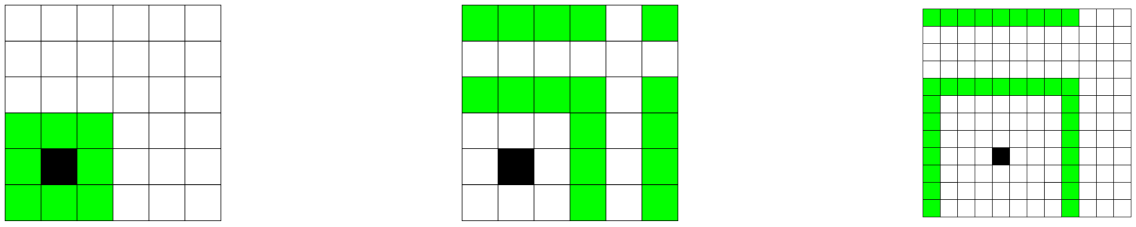


Figure 3: Static evolution. Conceptually, all cells are updated at the same time, since the evaluation and update phase are distinct. In the actual implementation, we will see this is not the case, but how the update propagates is represented by an instant change of the grid.

1.2.4 Wave evolution

The so-called "wave" update is based on the idea of propagating a wave signal across the whole grid and updating according to this. By wave, we mean the outer frame of radius r centered in a given cell. Fig.4 clarifies the situation.

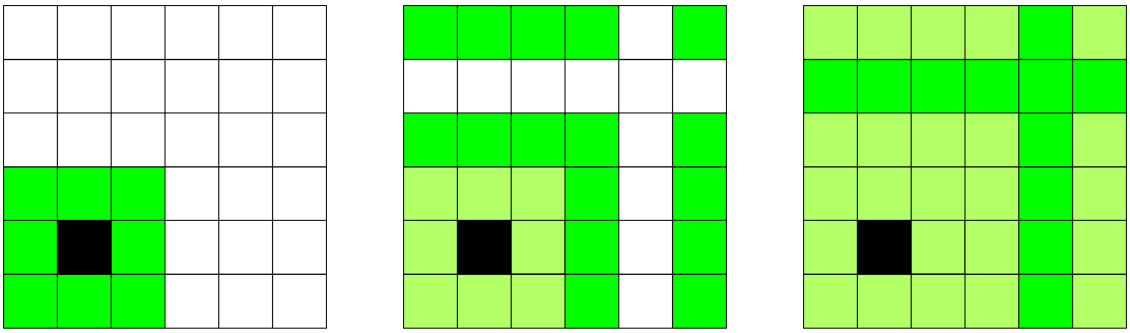


(a) $r = 1$ (b) $r = 2$ (c) $r = 4$

Figure 4: The outer frame, given a cell (black square) and a radius.

In particular, to perform one single update of the whole grid we have the following iterative steps (see Fig.5):

- pick a random cell across the grid, let's call it C
- update C, according to its neighbors like in the static update 1.2.3.
- take the outer frame of radius 1 centered in C, update its cells considering the updated version of C
- take the outer frame of radius 2 centered in C, update its cells considering the updated version of the outer frame of radius 1
- take the outer frame of radius r centered in C, update its cells considering the updated version of the outer frame of radius $r-1$
- proceed until $r = k/2$ for odd k, $r = (k-1)/2$ for even outer frame
- if k is odd, terminate the update of the grid
- if k is even, update the missing row and column



(a) At first, outer frame with $r = 1$ is updated.

(b) Then, outer frame with $r = 2$ is updated.

(c) At the end, missing row and column are updated.

Figure 5

From the point of view of pure high-level programming, it's clear that this kind of update requires a bit more care, compared to 1.2.2 and 1.2.3. First of all, we need to have a function that given a cell C over the grid and given a radius r, returns exactly the cells forming the outer frame of radius r centered in C. To deal with this problem, we tried first a naive approach, then we found a cleverer way to solve the problem. The naive (Fig. 6a) approach simply consists of having two nested for-loops, both iterating over a range from $-r$ to r, scanning the grid, and selecting all the square (i.e. the outer frame plus the interior). At this point, we simply discard the interior using an if condition, as shown in the code snippet below. The problem with this approach is that it is $O(r^2)$, so now we expose the smarter one.

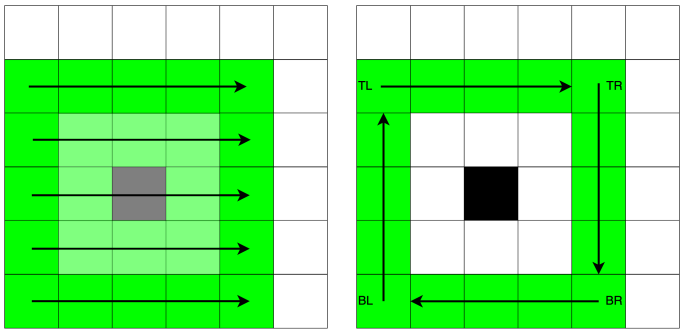
```
for (int i = -r; i <= r; ++i) {
    for (int j = -r; j <= r; ++j) {

        if (i > -r && i < r && j > -r && j < r)
            continue; // Skip the inner square
```

```
        int row = (index / k + i + k) % k;
        int col = (index % k + j + k) % k;
```

The idea is to compute 4 different general formulas (3)-(5), each giving one corner of the square, given the index of one cell C. Then, starting from each corner we iterate over the length of the edge of the square, getting the outer frame with 4 separate loops, each of them requiring $O(r)$ (Fig. 6b). Note that we use another formula (2) to compute the length of one edge, given the radius r:

$$l = 4(2r-1)+4 \tag{1}$$
$$TL = k((c/k - r + k) \mod k) + (c \mod k - r + k) \mod k \tag{2}$$
$$TR = k((c/k - r + k) \mod k) + (c \mod k + r) \mod k \tag{3}$$
$$BL = k((c/k + r) \mod k) + (c \mod k - r + k) \mod k \tag{4}$$
$$BR = k((c/k + r) \mod k) + (c \mod k + r) \mod k \tag{5}$$



(a) Naive approach to recover the outer frame.

(b) Smarter approach to recover the outer frame.

Figure 6

At this point, before getting in-depth into the details of the implementation, it's worth noticing that when k is even, we cannot have a frame with radius $k/2$, hence the last admitted length of the radius is $(k-1)/2$, meaning that we are left with one row and one column to update (see Fig. 4b for a more direct interpretation). To tackle this problem, as the last step, we need to recover which this column and row are: to do this we will recover the cell lying in the intersection of the row and the column. This is achieved using an auxiliary function, called only in case of even k.

```
int map_even_grid(int rand_cell_idx, int k){
    if (rand_cell_idx%k < k/2){
        return (rand_cell_idx+ ((k/2) *k))%(k*k) + k/2;
    }else{
        return (rand_cell_idx+ ((k/2) *k))%(k*k) - k/2;
    }
}
```

1.3 Implementation

This section delves deeper into the technical aspects of the implementation, exploiting the concepts introduced in Subsection 1.2.
When considering the actual implementation for each evolution method, we provided two functions: one to handle single process cases, exploiting parallelization only using OpenMP; the second to tackle calls of multiple processes, combining OpenMP and MPI in a hybrid approach. This avoids potential issues related to inter-process communication and mitigates additional overhead.
Calls to the functions are handled by a wrapper function.

The compiled executable encapsulates various functionalities based on user-defined parameters:

- when called with the `-i` argument, the functions initializes a random playground
- when called with the `-r` argument, the function runs a specified type of evolution, depending on the argument `-e`
- if the argument `-e $num_value` is called, where `$num_value` is a placeholder for a number in $\{0, 1, 2\}$, the evolution is run respectively in ordered, static or wave evolution
- if the argument `-f $file_name` is called, depending on previous arguments, the executable proceeds to:
  - save on `$file_name` the random initialized playground, if called in combination with `-i`
  - run the evolution starting from the snapshot on `$file_name`, if called in combination with `-e`

This multi-functional executable offers diverse functionalities catering to matrix generation, evolution, and storage, providing users with versatile options to manipulate and analyze grid evolution.

1.3.1 Grid operations: initialization, reading and writing

As previously said, our working environment is abstractly a square matrix of dimensions k $\times$ k. Actually, the matrix representation in our C implementation utilizes a one-dimensional array (per process) of the `unsigned char` type. Employing this structure ensures that each matrix entry consumes a single byte, maintaining contiguous memory allocation.
To exploit parallelization, the provided functions to write and read a PNG image have been modified to handle domain decomposition with multiple processes. This allowed us to optimize more efficiently the space in memory used by each process; in fact, with the given implementations all the grid would have been stored entirely in one process, if not in all of them.

Domain decomposition MPI processes are initialized within the `main` using `MPI_Init`, which sets up the MPI environment. The `MPI_Comm_rank` and `MPI_Comm_size` functions determine the rank and total size of the MPI processes, respectively. This enables each process to identify its unique rank among the processes and the total number of processes involved in the computation. The variable `my_rows_number` is calculated in the following way, based on the rank and the total size, ensuring that each process knows the specific portion of the grid it's responsible for handling.

```
1 int my_rows_number = (rank<(k%size)) ? k/size + 1 : k/size;
```

This calculation divides the grid into segments and allocates them among the processes, ensuring an even distribution of work among the MPI processes. This way, each process can efficiently manage its designated portion of the grid during the parallel processing. Regarding the distribution of the workload across the threads, instead, the *static* schedule is kept, given that each process handles a similar amount of workload (as we will see in the following Subsections) and a similar part of the domain. In this way, threads will also operate similarly w.r.t. workload.

Reading The function `parallel_read` is responsible for reading a grid from a file in parallel using MPI. It takes various parameters like

- `file_path` a string containing the path and the name of the file to read.
- `**grid_pointer`, a pointer to the (partial) grid where, for each process, the corresponding lines of the file to read will be stored
- `*maxval`, a pointer to the maximum value of the characters in the file (in this case it will always be 256, since we will only need black and white).
- `*xsize` and `*ysize`, pointers to the height and width of the matrix. Since we are working only with square matrices these will always be the same value of k.

The function takes also MPI-specific parameters like `my_rows_number`, `rank`, `size`, and `comm`.

The file is at first open in `FILE` format by process 0, which determines the grid's characteristics such as `xsize`, `ysize`, and `maxval`. It skips comments and reads the header information to extract these values (just like in the provided implementation of `write`).
The first process also gets the position of the file pointer after reading all the header information, and it stores its value in the variable `header_offset` (that has been previously initialized in every process). Once the header information is obtained by process 0, the `FILE` is closed.

The file is then opened as an `MPI_File` and then the data obtained previously by the process 0 is broadcasted to all other processes using `MPI_Bcast`.

```
MPI_Bcast(&header_offset, 1, MPI_OFFSET, 0, MPI_COMM_WORLD);
MPI_Bcast(xsize, 1, MPI_INT, 0, MPI_COMM_WORLD); //xsize == k
```

The partial grid of each process is then initialized based on `my_rows_number`. As previously said at the beginning of the Methodology section 1.2, the grid is actually stored as a 1D array, as we can see in the following snippet, using the function `malloc`.

```
*grid_pointer = (unsigned char*)malloc(*xsize*my_rows_number*sizeof(unsigned char));
```

Additionally, the offset for each process to start reading the grid data is determined based on the process's rank and `my_rows_number`.

For handling offsets, the code calculates the starting position in the file from which each process should read the grid. This is done by creating an array `offset_arr` that stores the offset for each process, considering the number of rows allocated to each process, and then summing over all the previous processes. This allows each process to position its file cursor at the appropriate starting point for reading the grid data.
The grid data is read from the file in parallel using `MPI_File_read`. Each process reads its allocated portion of the grid into a continuous block of memory pointed to by `grid_pointer`. The `MPI_File_seek` function ensures that each process starts reading from its designated position in the file.
Finally, after reading, the file is closed.

```
int* rows_per_process = (int*)malloc(size*sizeof(int));
for (int i=0; i<size; i++){
    rows_per_process[i] = (i<(*xsize%size)) ? *xsize/size +1 : *xsize/size;
}
int* offset_arr = (int*)malloc(size*sizeof(int));
offset_arr[0] = 0;
for (int i=1; i<size; i++){
    offset_arr[i] = offset_arr[i-1] + rows_per_process[i-1]*(*xsize);
}
MPI_Offset offset = offset_arr[rank]*sizeof(unsigned char)+header_offset;
MPI_File_seek(file, offset, MPI_SEEK_SET)
MPI_File_read(file, *grid_pointer, my_rows_number * (*xsize), MPI_UNSIGNED_CHAR, &
    status);
```

Writing The function `parallel_write` is responsible for writing a grid into a file in parallel using MPI. It takes various parameters like

- `file_path` a string containing the path and the name of the file to write
- `*grid`, a pointer to the (partial) grid where, for each process, are stored the corresponding lines of the file to write.
- `maxval`, the maximum value of the characters in the file (again, always 256).
- `k`

The function also takes MPI-specific parameters like `my_rows_number`, `rank`, `size`, and `comm`.

This function is very similar to the reading one in all aspects:

- the process 0 handles the header information, this time writing it.
- the process 0 gets the file pointer position after writing the header and broadcasts it to the other processes.
- the individual offset for each process is found precisely as in the write function, and then the `MPI_File_seek` function acts in the same way as before, setting the individual pointer of each process in the correct place.
- the grid data is written into the file in parallel using `MPI_File_read`.

Initializing the grid The `init_parallel` function, after taking the file path and k as arguments, along with MPI-specific parameters like `my_rows_number`, `rank` and `size`, utilizes the `parallel_write` function to efficiently initialize a random grid. After allocating a partial grid for each process, it is initialized randomly by each process, to be then written in parallel with the other processes using the previously defined function `parallel_write`.

It is important to notice that both the `parallel_read` and `init_parallel` functions initialize a partial grid. This could seem redundant, since after the initialization we could keep in memory the partial grids and use them later to compute the evolution, without reading first the written files. However, the assignment requires the program to divide the initialization and the evolution steps, by distinctively writing the initial grid in a file, so that the evolution can later start by reading the same file.

1.3.2 Ordered evolution

As discussed in the Methodology part and in the Introduction, the ordered evolution is intrinsically serial, so possibilities of parallelization are limited. However, given our parallel initialization, the ordered evolution can still benefit from the domain decomposition, and limit the amount of memory occupied by each process.

In our implementation, it's used the traditional evolution algorithm provided in the previous section, represented by two nested loops: one running over the rows, the second over the columns. There is one outer for running over the number of steps that we want the evolution to perform.

```
for (int step = 0; step < n; step++){
    #pragma omp parallel for ordered
    for (int i = 0; i < k; i++){ // Loop over all rows
        #pragma omp ordered
        for (int j = 0; j < k; j++){ // Loop over all columns
            int next_row = (i+1+k)%k;
            int previous_row = (i-1+k)%k;
            int next_column = (j+1+k)%k;
            int previous_column = (j-1+k)%k;
            int sum = grid[previous_row + previous_column] +
                grid[previous_row + next_column] +
                grid[next_row + previous_column] +
                grid[next_row + next_column] +
                grid[previous_row + j] +
                grid[next_row + j] +
                grid[i*k + previous_column] +
                grid[i*k + next_column];
                grid[i*k+j] = (sum > 765 || sum < 510) ? 0 : 255;
        }
    }
```

As we can see, all the `#pragma omp` directives are `ordered`, in fact representing serialization. It's also important to note that the grid is updated at each step, without keeping track separately of the updates (as will happen in static evolution).

To handle the fragmentation of the grid given by the parallel initialization, each process will handle the same rows (whose number is denoted by the variable `my_rows_number`). To tackle the problem of the ordering in the evolution, however, the following system of MPI messages has to be established.
Each process allocates two new rows, `next_row` and `previous_row`, where it will receive the cells of the grid belonging to the next and the previous row respectively.

```
unsigned char* previous_row = (unsigned char*)malloc(k*sizeof(unsigned char));
unsigned char* next_row = (unsigned char*)malloc(k*sizeof(unsigned char));
```

After this initialization, the following message operations are performed:

```
if (rank==size-1){
    MPI_Send(grid + k*(my_rows_number-1), k, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD
    );
    }
if (rank!=0){
    //send my first row to rank-1
    MPI_Isend(grid, k, MPI_UNSIGNED_CHAR, rank-1, next_tag+ step*size +(rank),
    MPI_COMM_WORLD, &request);
    //receive my last row from rank+1
```

```
8       MPI_Recv(next_row, k, MPI_UNSIGNED_CHAR, (rank + 1)%size,next_tag+ step*size+((
        rank+1)%size), MPI_COMM_WORLD, &status);
9       //receive previous row from rank-1
10      MPI_Recv(previous_row, k, MPI_UNSIGNED_CHAR, rank - 1, step*size+(rank-1),
        MPI_COMM_WORLD, &status);
11  }else{
12      MPI_Recv(previous_row, k, MPI_UNSIGNED_CHAR, size-1, 0, MPI_COMM_WORLD, &status)
        ;
13      MPI_Recv(next_row, k, MPI_UNSIGNED_CHAR, 1,next_tag+ step*size+(rank+1),
        MPI_COMM_WORLD, &status);
14  }
15  [...] //actual update of the partial grid
16  if (rank!=size-1){
17      //send my last row to rank+1
18      MPI_Isend(grid + k*(my_rows_number-1), k, MPI_UNSIGNED_CHAR, rank+1, step*size
        +(rank), MPI_COMM_WORLD, &request);
19  }
20  if (rank == 0){
21  MPI_Send(grid, k, MPI_UNSIGNED_CHAR, size-1,next_tag+ step*size +(rank),
        MPI_COMM_WORLD);
22  }
```

To better break this code down, it is useful to reason on which requests are performed **before** and which ones **after** the ordered update.
The requests performed before are:

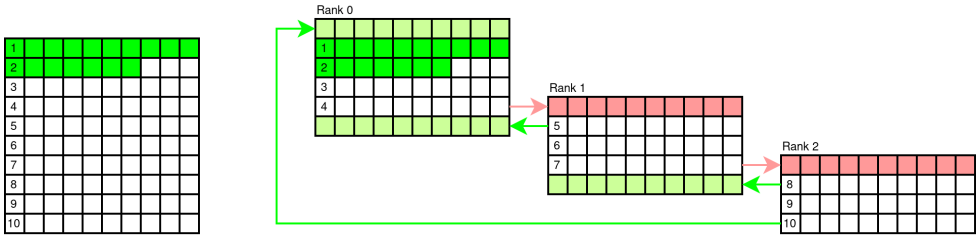
- sending and receiving of the last row of the last process to the first process, so that the ordered evolution can start right away
- sending and receiving of the first rows of each process *except the first* to the previous one so that each process will already have the last row.
  The first process **does not** send its first row to the previous (i.e. the last) process: for the intrinsically serial nature of this evolution, this sending needs to be performed after the update.
  the last process, however, already performs the receive complementary operation, so that as soon the first process finishes, it will receive its next row.
- **blocking** receiving of the previous row. This request is not fulfilled until after the previous process performs the evolution. Only the first process immediately receives its previous row.

After the ordered update these requests are performed:

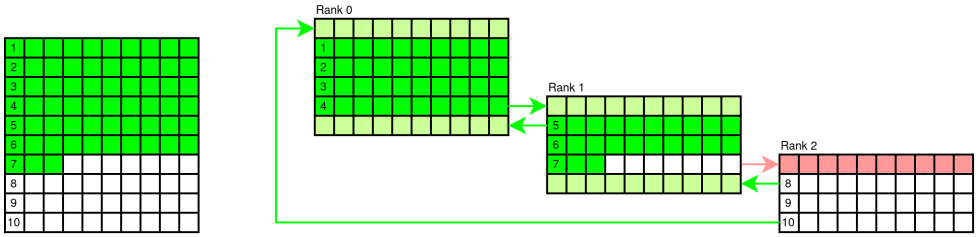
- sending of the last row to the next process, so that its blocking receive call will be fulfilled and it will proceed with the evolution. This call is not performed by the last process, since at that point the evolution of the step will be complete.
- sending of the first row by the first process to the last process.

The key about this system of messages is that the receiving call for the previous line (except for the first process) is blocking, this imposes that the evolution follows the strictly ordered path from which it takes its name.

Figure 7 represents the communication system between processes for a grid with $k = 10$ and $size = 3$ (i.e. 3 processes). The complete grid is reported only as a mean to better comprehend the message passing, it isn't actually stored in any process.



(a) At the beginning, only Rank 0 proceeds with the evolution (marked in solid green and reported on the complete matrix), since it has received both the previous row from Rank 2 and the next row from Rank 1. Rank 1 and Rank 2 processes have pending receive requests for their previous rows, while the next row of Rank 1 is already recovered. Rank 2 doesn't need a next row



(b) The computation migrates to Rank 1, as Rank 0 has fulfilled its pending receive request for the previous row. Rank 2 still has the request for the previous row pending

Figure 7: MPI implementation of static evolution.

1.3.3 Static evolution

As said a few times previously, the static evolution is the perfect playground for exploiting parallelization. This is because the steps of evaluation of the neighbors and updating are separated, so we can work with a *frozen* grid.
All the updates are temporarily written in a copy of the grid, called `next`, which is initialized in the `main`, when the static evolution is selected:

```
[...] //main.c
}else if (e == STATIC){
    unsigned char* next = (unsigned char*)malloc(k*my_rows_number*sizeof(unsigned
    char));
```

The actual evolution step (i.e. the fragment of the code in which information about the neighbors is retrieved and in which the next status of the cell is computed) is similar to the one in the Ordered evolution, with the big difference that the `#pragma omp` directives are not `ordered`. The default `static` schedule has been used, since the workload is constant for each step of the iteration.

```
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i<k; i++){
        for (int j=0; j<k; j++){
            [...] //evaluation of the neighboring cells
```

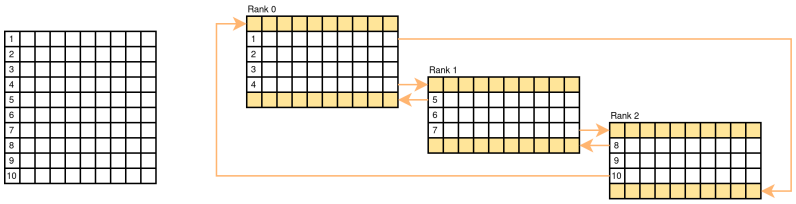
```
            next[i*k+j] = (sum > 765 || sum < 510) ? 0 : 255;
        }
    }
}
unsigned char* tmp;
        tmp = next;
        next = grid;
        grid = tmp;
```

This way, all threads for a given process can work on the task of updating the `next` grid.

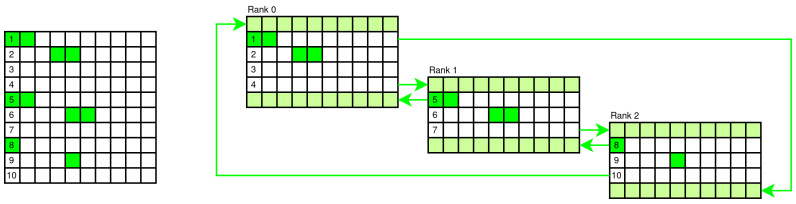
Regarding the MPI implementation, as in the ordered case, for each process to work on its partial grid a system of messages is required. However, in this case, there's no intrinsic ordering, so there's no reason to have blocking routines.

```
for (int step=0; step<n; step++){
    //send the last row, tag = step*size+rank_of_sending_process
    MPI_Isend(grid+(my_rows_number-1)*k, k, MPI_UNSIGNED_CHAR, (rank+1)%size, step*
    size+rank, MPI_COMM_WORLD, &request[0]);
    //send the first row
    MPI_Isend(grid, k, MPI_UNSIGNED_CHAR, (rank+size-1)%size, step*size+rank,
    MPI_COMM_WORLD, &request[1]);
    //receive the last row
    MPI_Irecv(previous_row, k, MPI_UNSIGNED_CHAR, (rank+size-1)%size, step*size+(
    rank+size-1)%size, MPI_COMM_WORLD, &request[2]);
    //receive the first row
    MPI_Irecv(next_row, k, MPI_UNSIGNED_CHAR, (rank+1)%size, step*size+(rank+1)%size
    , MPI_COMM_WORLD, &request[3]);
    MPI_Waitall(4, request, MPI_STATUS_IGNORE);
    [...] //actual evolution step
}
```

After all these routines, the function `MPI_Waitall` waits for all requests to be completed, so that each process can start its computation. Figure 8 provides an example of message passing with two processes.



(a) At first, all non blocking requests are performed. In the static evolution, even the last process has the next row.



(b) After the `MPI_Waitall` function, all process have received their next and previous rows, so they can start the computation. Here, the solid green doesn't mean that the update has happened, that will happen later when the pointers to **`next`** and **`grid`** will be switched; it just means that the neighboring cells of the cell in question have been evaluated. In this example, each process spawns two threads.

Figure 8: MPI implementation of static evolution.

1.3.4 Wave evolution

In this section, we will describe a possible parallel implementation of wave evolution. First of all, we will show how the implementation of a step update occurs. We will later analyze how this is adapted to work with several processes in parallel. We will then show the implementation of the outer frame recovery function and how this must be adapted to work in the case of parallel implementation.

As stated in Section 1.3.4, given a random cell C, which is uniquely identified by an index ranging in $[0, k^2 - 1]$, we need to iteratively cover the grid by increasing the radius (see Fig. 5)

```
k%2==0 ? (thresh = (k-1)/2) : (thresh =k/2); // max length of the radius

for (int radius=1; radius<= thresh; radius++){
    int tmp = (4*(2*radius - 1) + 4); // # of cells in the outer frame
    unsigned int* idxs = recoverSquare(k, rand_cell_idx, radius); // call the
    function to recover the outer frame

    for (int ii=0; ii<tmp1;ii++){
        int prev_col = (idxs[ii] -1 +(k*k))%(k*k);
        int next_col = (idxs[ii] +1 +(k*k))%(k*k);

        int sum=0;
        sum += grid[prev_col]+
        grid[(prev_col + k +(k*k))%(k*k)] +
        grid[(prev_col -k +(k*k))%(k*k)] +
        grid[(idxs[ii] - k +(k*k))%(k*k)] +
        grid[(idxs[ii] + k +(k*k))%(k*k)] +
```

```
        grid[(next_col - k +(k*k))%(k*k)] +
        grid[(next_col + k +(k*k))%(k*k)] +
        grid[next_col];
        next[idxs[ii]] = (sum > 765 || sum < 510) ? 0 : 255;
    }


        for (int ii=0; ii<tmp1; ii++){
            grid[idxs[ii]] = next[idxs[ii]]; // update the outer frame of the grid
    }
    free(idxs);
}
```

Note that the iteration over the length of the radius is intrinsically serial: in fact, when we iterate over an outer frame of radius r, all the "internal neighbors" (ideally, those that lie in the convex hull defined by the frame) correspond to the frame of the step previous.

What can instead be parallelized is the iteration over the cells that make up the outer frame. As already explained in Section 1.2.4, with the `recoverSquare` function we can find the indices of the cells that make up the outer frame, given the radius and the source cell. At this point, we can just partition the array containing these indices among all the processes. In this way, each process will update an area of the outer frame, and then, at the end of the execution of the current iteration on a given radius, it will communicate the cells it has updated to the other processes, and at the same time receive the updates from other processes.

We must mention a crucial point of this implementation: due to the difficulty in finding the neighbors given a portion of the outer frame, each process is forced to keep the entire grid in memory and communicate through an `AllGatherv` operation the entire grid. This procedure, as we will see in the Results section, is highly non-optimal, since all processes share simultaneously lots of repeated information. However, it does not create conflicts since each process updates cells which other processes will never modify.

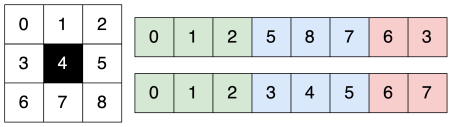
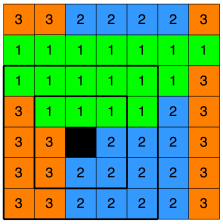


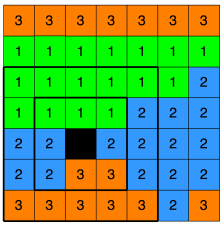
Figure 9: On the right, a 3×3 grid where cells are indexed according to the indexes of the array where the grid is stored. On the top-left is the unsorted list of the cells belonging to the outer frame, as recovered by the algorithm, and its underlying distribution over processes. On the bottom-left is the sorted version.

We mentioned the usage of the `AllGatherv` function: so let's analyze how we send information. One of the difficulties of this type of evolution is that, while the grid and the array are organized "by rows", our updates are organized "by outer frames", moreover in a toroidal topology. The way the `recoverSquare` function is defined, it iterates over the top side of the outer frame, then over the right side, and so on. Therefore the array of indices that make up the outer frame will generally be unordered (Fig. 9). This is a problem because since the grid is organized as a single array, we will have to "jump" from position to position to track the updated cells.

To solve this problem, the first thing to do is sort the array of outer frame indices and partition the sorted array among processes. This operation adds a cost of $O(r \log r)$ to our algorithm, but as



(a) Partition of the outer frame based on the unsorted list of the outer frame's cells recovered by the algorithm.



(b) Sorted version of the output of the algorithm.

Figure 10: Partition of the outer frame

we will see later the resulting advantages are very important. Fig. 10 shows how sorting the array affects the partition of the outer frame among the processes.

At this point, it will be simple to organize the displacements to be supplied to the `AllGatherv` function. Each process simply sends to all others, the portion of the grid (by portion we mean "portion of rows") starting from its first cell to update to the last cell right before the first cell of the next process (see Fig. 11 and code below). Doing so, each process sends a portion of the grid, and `AllGatherv` call will take care of "sticking together" all portions of the grid, and send to each process the updated version of the grid, ready for the next iteration over the radius.

```
int tmp1 = (4*(2*radius - 1) + 4);
unsigned int* idxs = recoverSquare(k, rand_cell_idx, radius);

int rem = tmp1 % size;
int sum = 0;
int *sendcounts = (int*)calloc(size*sizeof(int), sizeof(int));
int *displs = (int*)calloc(size*sizeof(int), sizeof(int));

for (int i=0; i<size; i++){
    sendcounts[i] = tmp1/size;
    if (rem > 0){
        sendcounts[i]++;
        rem--;
    }
    displs[i] = sum;
    sum += sendcounts[i];
}

// iteration over the cells in the outer frame
[...]

int offset;
if (rank!=size-1){
    offset = idxs[displs[rank+1]] - idxs[displs[rank]] - 1;
}else{
    offset =  idxs[tmp1-1] - idxs[displs[rank]] + 1;
}

int *recvcounts = (int*)calloc(size*sizeof(int), sizeof(int));
for (int i=0; i<size-1; i++){
```

```
    recvcounts[i] = idxs[displs[i+1]] - idxs[displs[i]];
}
recvcounts[size-1] = idxs[tmp1-1] - idxs[displs[size-1]] + 1;

int *recvdispls = (int*)calloc(size*sizeof(int), sizeof(int));
for (int i=0; i<size; i++){
    recvdispls[i] = idxs[displs[i]];
}

MPI_Allgatherv(next + idxs[displs[rank]], offset, MPI_UNSIGNED_CHAR, grid,
    recvcounts, recvdispls, MPI_UNSIGNED_CHAR, MPI_COMM_WORLD);
```

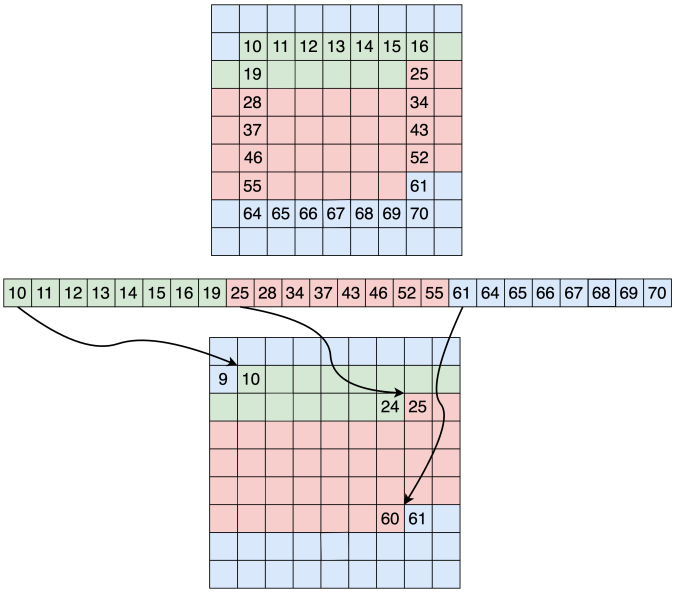
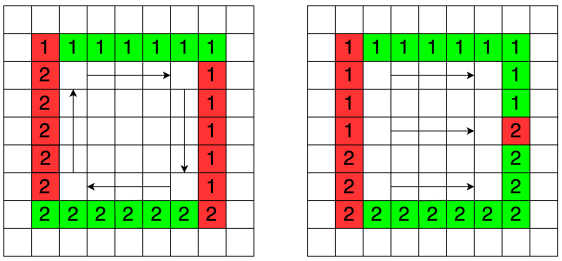
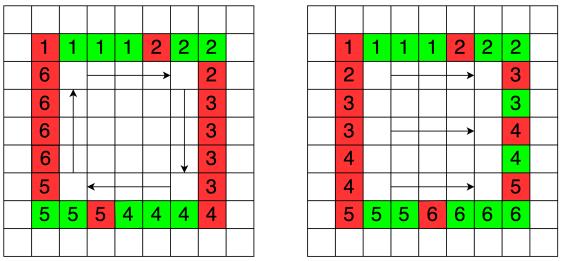


Figure 11: On the top of the figure, the outer frame shown in the figure provides the indexes of the cell. We have 24 cells in the outer frame and 3 processes. Hence each process has 8 cells to update, and the first process has exactly the first third of cells, the second one has the second third and the third one has the last third, and this is assured by the sorting operation. On the bottom side, we see the start and end indexes of each process, which correspond to the portion of the grid that a process will update.

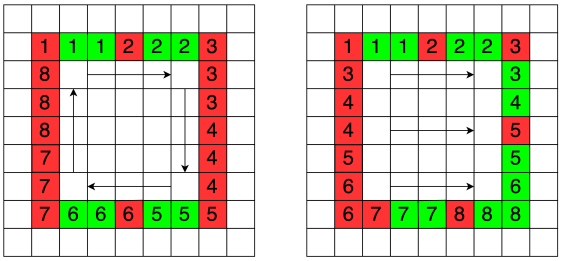
Previously we mentioned additional benefits of sorting the array containing the indexes of the cells that form the outer frame. Another important improvement obtained concerns the implementation of the algorithm with one single process, exploiting different threads (therefore within the "OpenMP" environment). In this case, sorting the array significantly reduces cache misses (Fig. 12, since with this partitioning we tend to move more by rows, and therefore we make more use of the row-major ordering due to how we map the grid in the array (i.e. by "blocks of rows", not columns).



(a) Cache hits with 2 threads.



(b) Cache hits with 6 threads.



(c) Cache hits with 8 threads.

Figure 12: On each image, on the left you can see the case when the array of indexes of the outer frame is unsorted, on the right the case in which it's sorted. Cache hits are highlighted in green, cache misses in red, and we assume that a cache line corresponds to a row of the grid.

1.4 Results and Discussion

In the present section, we will expose the results obtained by running our code on ORFEO Data Center[3]'s EPYC nodes (Fig. 13). For each of the three evolution modes, we will analyze the scalability of the code in different technical configurations. For each analysis, we will provide the

Figure 13: Configuration of one EPYC node on ORFEO: 2 sockets, 4 NUMA regions per socket, 16 cores per NUMA region.

following scalability studies:

- MPI strong scalability: keep fixed the size of the problem and the number of threads, increase the number of processes.
- MPI weak scalability: keep fixed the workload, that is the ratio between the size of the problem and the number of processes.
- OpenMP scalability: keep fixed the number size of the problem and the number of processes, increase the number of threads.

1.4.1 Writing and random initialization

Strong MPI scalability Let's start our analysis by discussing writing and initialization operations. For the strong scalability, we decided to run over 2 nodes, with 8 MPI tasks per node (hence mapping each process to a NUMA region) and 16 CPUs per task. The threads' number was fixed to 16 per task. From Fig. 14 we can observe that, when the number of processes exceeds 8, meaning that we started exploiting the second node, we see a performance degradation in the

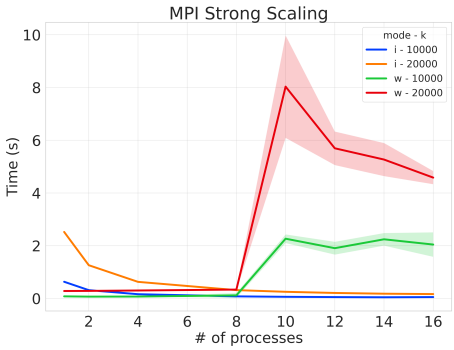


Figure 14: Strong scalability over writing and initializing operations.

writing operation. This is probably due to lower speeds in extra-node communication.
For the initialize operation, however, no noticeable differences are retrieved: this task is exploited completely inside each process, without the need for further communication with the others.

Weak MPI Scalability Regarding the weak MPI scalability, we observe a behavior similar to the previous one. When looking at Fig. 15, we would expect a horizontal line, which means perfect scalability and linear dependence between time spent and workload per process. For the writing operation, this phenomenon is observable from 1 to 8 processes and then from 8 processes onwards. However, there is a considerable jump between 8 processes and above, caused by even greater latency due to slower communications between nodes compared to communications within the node. Initializing operations is again pretty straightforward and presents results in line with the theory.

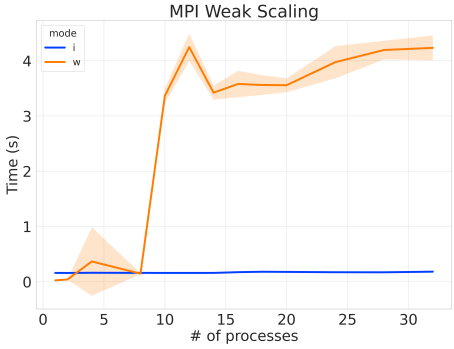
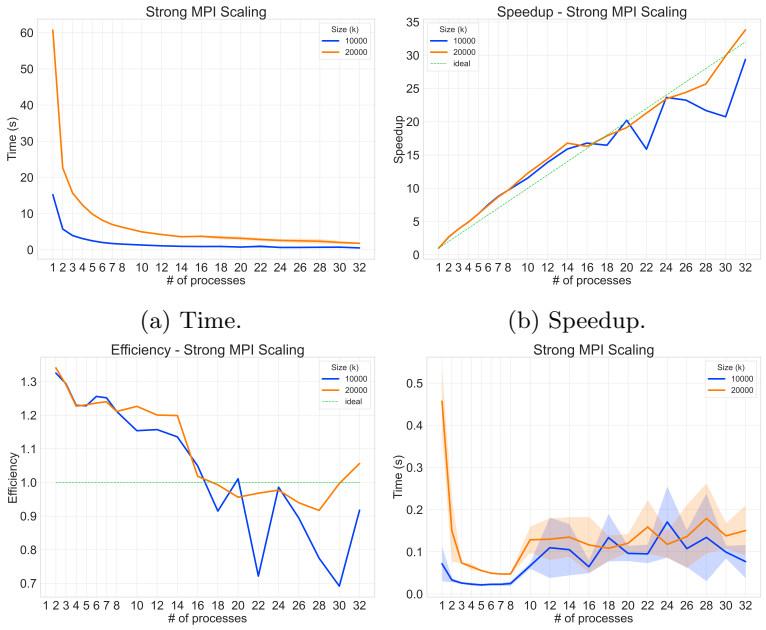


Figure 15: Weak scalability over writing and initializing operations.

1.4.2 Static evolution

Strong MPI scalability For what concerns the strong MPI scalability study for the static update, we ran our parallel code on 4 nodes, mapping processes to NUMA regions, to exploit the shared memory region within each NUMA region. The number of threads was kept constant at 16, exactly one per core (Fig. 16), and the size where fixed to 10000 and 20000. Threads are mapped to cores, using a close binding policy in order to exploit the shared L3 cache shared among close physical cores (see Fig. 13). Since this evolution is the one that is more suitable to be parallelized,



(a) Time.

(b) Speedup.

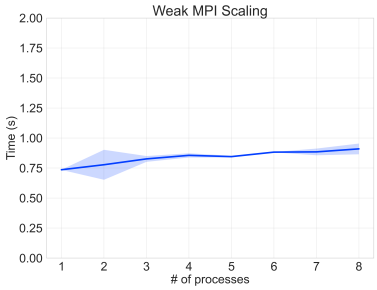
(c) Efficiency.

(d) Reading time.

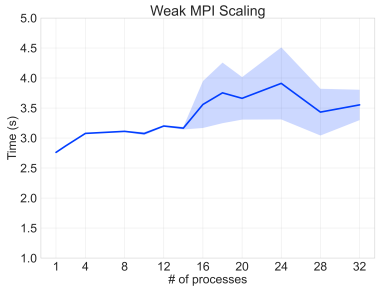
Figure 16: Strong MPI scaling. 4 nodes, 16 threads

as expected we obtain good results. Efficiency is surprisingly above 100%up to 16 processes (which means that we are using 2 nodes). However, efficiency stays always above 70%, which is a good result. By looking at Fig 16d, it is possible to see that the time in reading increases significantly when we "fill" one node with the processes, and start to populate all the other three nodes.

Weak MPI scalability In this scenario, two configurations were tried, both with 4 nodes. The first (Fig. 17a) has 2 tasks per node, hence 64 CPUs per task, and maps its task by node, binding to sockets. It exploits then 64 threads, one per physical core. The second configuration (Fig. 17b) has instead 8 tasks per node, mapping each task to a NUMA region, and 16 cores per task (with a fixed number of threads set to 16). In the first scenario, nothing interesting happens. In the second one, we can see an increase in time after 16 processes, corresponding to the moment when we have filled 2 nodes.



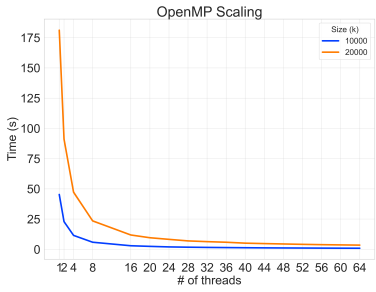
(a) Time up to 8 processes with 64 CPUs per task.



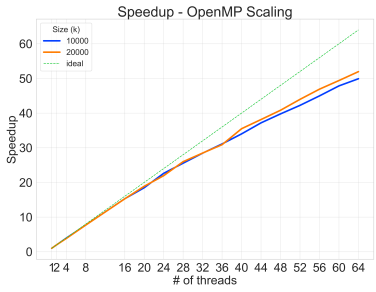
(b) Time up to 32 processes with 64 CPUs per task.

Figure 17: Weak MPI scaling on two different scenarios.

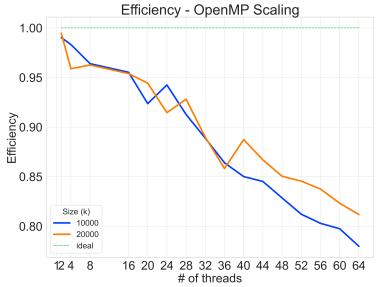
OpenMP scalability For the OpenMP scalability study, we opted for keeping the number of processes to 4, using 2 nodes, and mapping the MPI tasks to sockets.



(a) Time



(b) Speedup



(c) Efficiency.

Figure 18: OpenMP scaling with 4 processes and 64 CPUs.

Furthermore, 64 CPUs per task were used. Fig. 18 shows the scaling up to 64 threads, which are mapped on cores with a close binding policy. Note that 64 threads means exactly one thread per physical core. The speedup is consistent with the theoretical hypothesis, and the efficiency plot confirms this, as well. At this point we "played around" with the number of threads, to see what happens. We know that we have 64 physical cores per MPI task; exploiting AMD Simultaneous

Multi Threading, we can use up to 128 threads per process. But we can go further, and ask for 256 threads. At this point, each hardware thread will have 4 software threads. Fig. 20 reports the results. Let's break the analysis into two different parts. At first, we note that after 64 threads, we have a more consistent loss in efficiency. This happens because after 64 threads we start exploiting the SMT, meaning that more threads run on the same physical core. However, because of how the software threads are mapped to the machine, how the OpenMP update code is implemented, and how the L3 cache is shared among the cores, this is a problem. Let's think about the case where

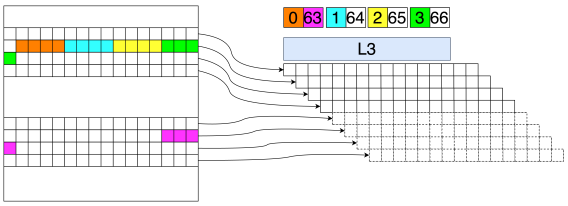
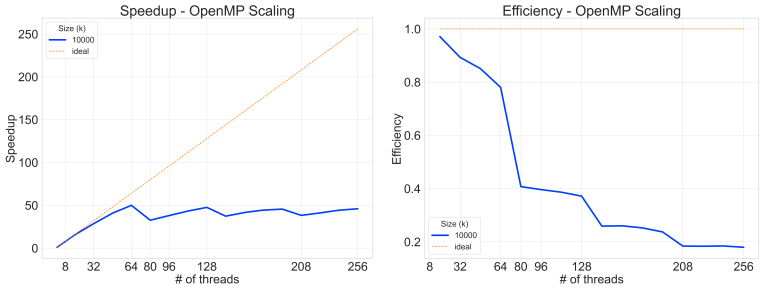


Figure 19: When we have one thread per core, all is straightforward, and ideally, we have to carry on the cache only the 4 solid rows. When SMT enters, the number of rows called in the cache increases (dashed rows in the figure). Due to the cache being limited, we encounter an overhead: this is the cache miss phenomenon.

we have one thread per core. In this case, neighboring threads access contiguous areas of the array, and to carry out their updates they will load portions of the array into the cache which will also be shared by neighboring threads. When we switch to SMT, other threads are added which must access contiguous areas of the array between them, but not contiguous to the portion of the array of the previous threads (Fig. 19). This leads to an increase in cache misses (note that we do not have false sharing, since the operations are read-only). So, the loss in efficiency observed after 64 threads is due to this phenomenon. But we've said that we have up to 256 threads, meaning that each core has to support up to 4 threads (cause each process has 64 cores): that's why we see the loss in performance on 64 (represented by a steeper decline in the speedup - SMT enters the game), 128 (3 threads per core) and 192 (4 threads per core).



(a) Speedup

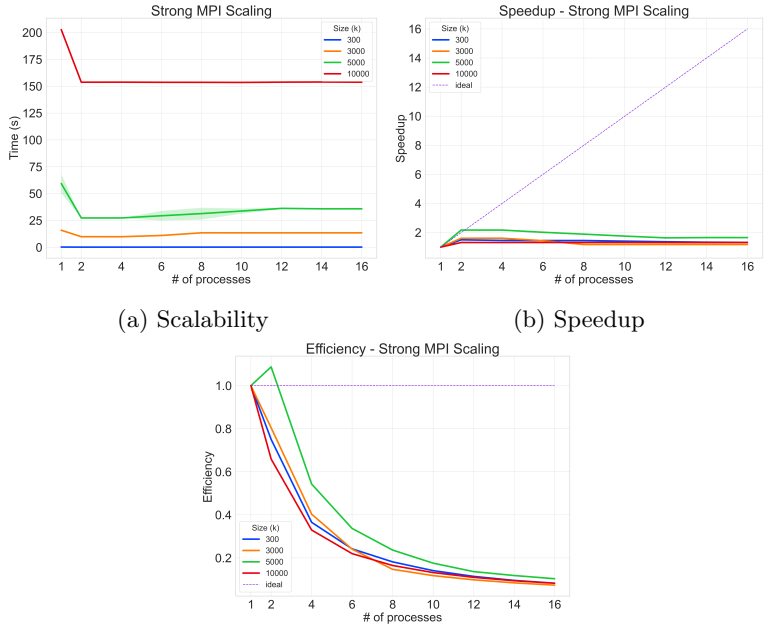
(b) Efficiency.

Figure 20: OpenMP scaling with 4 processes, each with 64 CPUs.

1.4.3 Ordered evolution

Due to the intrinsic serial implementation of this kind of evolution, we do not expect to see scalability behavior in our code, and the following results will prove this.

MPI strong scalability For the MPI strong scalability study, we decided to use 2 nodes, 8 MPI tasks per node, and 16 CPUs per task, hence studying the scalability up to 16 processes. Threads are kept fixed to one per physical core (Fig. 21). Please note that we have run with different sizes:



(a) Scalability

(b) Speedup

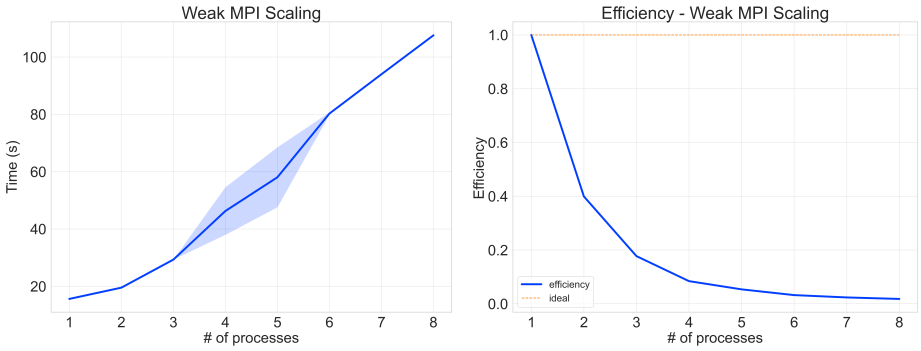
(c) Efficiency.

Figure 21: Strong MPI scalability up to 16 processes with 16 CPUs each one.

this has a precise purpose. By looking at the plot for $k = 10000$ (Fig. 21a), we can spot an elbow between 1 process and the other ones. At first, this might seem weird because since this mode is intrinsically serial, as mentioned before, we would expect a horizontal line (which means that time is independent of the number of processes). The reason is due to an overhead in reading the grid and exchanging massages, which occurs only from two processes forward. In fact, if we decrease the size of the grid, the elbow tends to get thinner and thinner until it disappears.

MPI weak scalability For the weak scalability study, as suggested by the previous analysis, we might expect an increasing function of the time with respect to the number of processes. Actually, this is what we get (Fig. 22).

OpenMP scalability As in the MPI strong scalability study, here we expect total non-dependency of the number of threads with respect to the time while keeping the size fixed.



(a) Scalability (b) Efficiency

Figure 22: Weak MPI scalability up to 16 processes - 2 nodes, 8 MPI tasks per node mapped by NUMA regions - with 16 CPUs each one, and 16 threads.

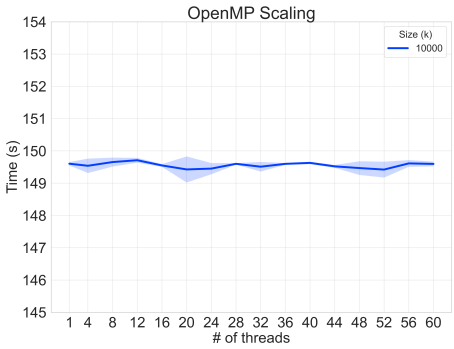


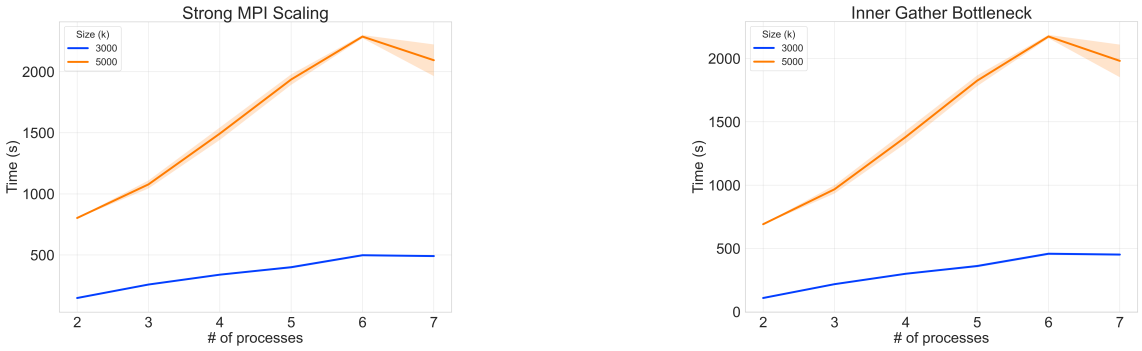
Figure 23: OpenMP scalability with 4 MPI processes mapped to socket on 2 nodes. Each MPI task has up to 64 CPUs. Each thread is associated with a physical core, adopting a close binding policy.

1.4.4 Wave evolution

Now we will discuss the MPI strong scalability and OpenMP scalability for the wave update.

MPI strong scalability As mentioned in the previous section about the wave update, our parallel implementation has a gap due to the use of the `AllGather` function, and as we will see this irremediably ruins the performance of the code (Fig. 24a). The problem is clearly the bottleneck caused by that function: the time taken to do the inner call of the function within the loop over the radius, is measured and as Fig. 24b shows, it increases as the number of processes increases. And this latency is so preponderant that the running time is completely destroyed: to demonstrate this, the two graphs in Fig. 24 are identical.

OpenMP scaling Now we expose our last result, the OpenMP scaling in wave update. We decided to run on one node, with one socket and 64 CPUs. Different configurations were tried and



(a) Scaling

(b) Inner gather operation bottleneck.

Figure 24: Strong MPI scaling. Up to 8 MPI processes, 16 CPUs per task.

subsequently analyzed. Fig. 26a- 26b show the scenarios of the threads mapped onto the cores with close and spread binding policies respectively. As highlighted by Fig. 26d, this is not true scalability,

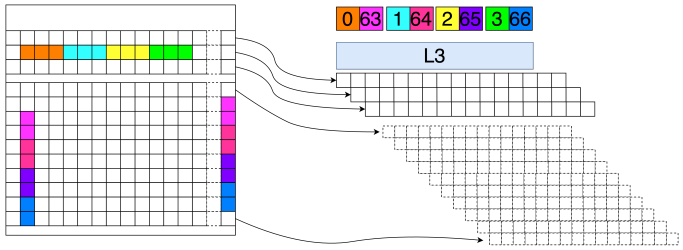
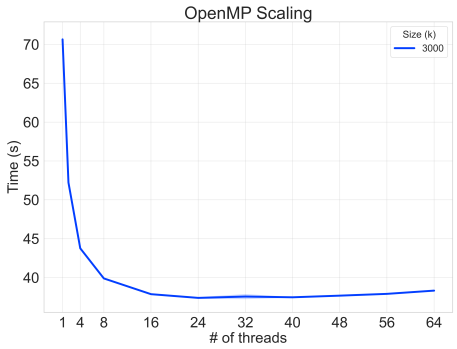
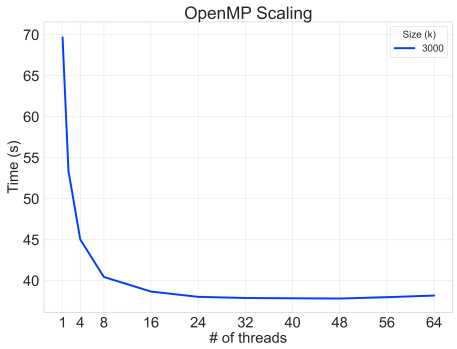


Figure 25: After threads $63, \ldots, 66$ enter, we have a significant overhead of cache lines (represented by dashed rows), caused by the topology of the problem.

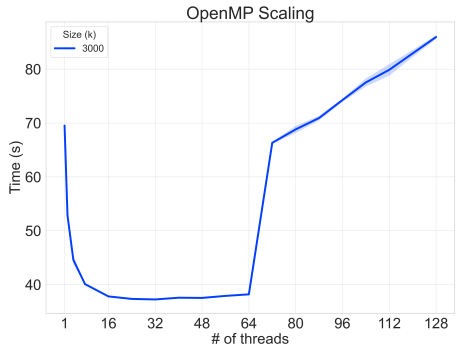
in fact, speedup and efficiency are very far from the ideal target, however, we can conclude that there is an improvement in execution time as the number of threads increases. Fig. 26c shows the scenario of threads mapped onto software threads with a close binding policy, up to 128. In this case, we observe a scenario similar to that described in the OpenMP Static Evolution section: once the 64 physical cores have been filled with 64 threads, we encounter an overhead in the cache (an increase in cash misses). In this case, given the non-continuous nature of the wave update, the effect is even more marked, as shown in Fig. 25.



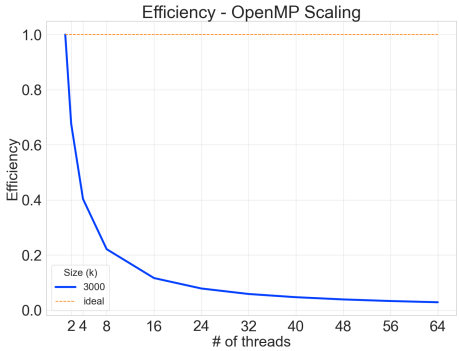
(a) Places: cores. Bind: close. This configuration was run with 64 threads, but it could have been run exploiting SMT (so up to 128 threads): in that case, we might have expected a behavior similar to Fig. 26c.



(b) Places: cores. Bind: spread. This configuration was run with 64 threads, but it could have been run exploiting SMT (so up to 128 threads): in that case, we might have expected a behavior similar to Fig. 26c.



(c) Places: threads. Bind: close.



(d) Efficiency. It's really the same in all three cases, so only efficiency for Fig. 26a is reported.

Figure 26: OpenMP scaling.

1.5 Conclusions

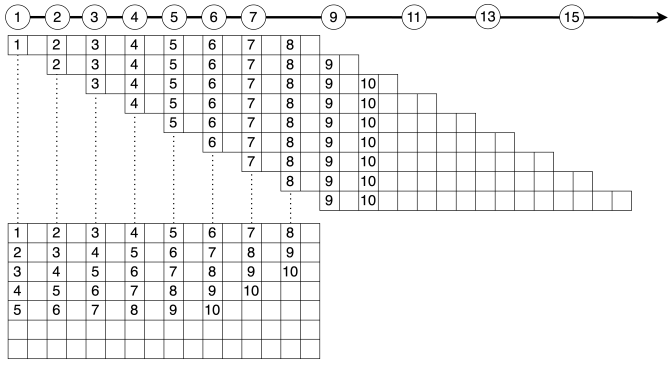
The parallelization of the Game of Life algorithm offers a fascinating insight into optimizing computational efficiency.

Static evolution Among the strategies explored, the static approach emerges as a natural fit for parallelization. By subdividing the grid into rows and assigning each process the responsibility of updating its allocated rows, the static approach can benefit from the domain decomposition with a theoretically perfect speedup. We underline once again that this opportunity for parallelization depends on the inherent 2-step nature of the update cycle: as the system freezes, we can separate the evaluation step from the actual update.

The static method's efficacy is evident in its ability to effectively distribute workload among processes, allowing for simultaneous updates without interdependence between rows. Leveraging this inherent independence among rows enables concurrent computation, significantly enhancing performance.

Ordered evolution Conversely, the ordered evolution method presents limitations in terms of parallelization. Its sequential nature, wherein each cell's update relies on the previous one, impedes opportunities for parallel execution. This inherent sequential dependency restricts the ability to distribute computation across multiple processes effectively.

In the context of a **finite** grid, however, exploring parallelization opportunities within the ordered evolution unveils a potential approach.
Just to give a grasp of the idea behind it, we imagine assigning to each row a different process: from this point, a parallelization strategy can be orchestrated.
Each process initiates the update for its respective row, while the subsequent process awaits the completion of the previous one to advance. Introducing a synchronization mechanism where each process waits until its predecessor has updated at least two additional cells enables a staged and synchronized progression. This approach allows for coordinated advancement while maintaining the sequential dependency inherent in the ordered evolution method, presenting an interesting solution for parallelization within the constraints of cell interdependency. Fig. **??** illustrates this methodology, referring to each time step with an increasing number.



Wave evolution Regarding wave evolution, our tests suggest that the implementation lacks parallelization exploiting MPI processes: the performance associated with increasing the number of processes is highly influenced by the overhead it carries, resulting in the bottleneck shown by inner gather operations. Concerning OpenMP scalability, we cannot say that the code scales, but at least we have an improvement in running time.

As a final note, we would like to introduce a possible way to tackle the parallelization problem for this kind of evolution. The idea is to split the grid in a radial way, according to the number of processes (Fig. 27). Each process is then allowed to keep a "private" zone of the grid, exploiting domain decomposition. Note that in this way, it is possible to reduce the overhead due to all processes keeping all the grid in their memory: at each time step, instead of sending the entire grid between all processes, we just send some diagonal lines. Note that this method still requires each process to wait for messages from its neighbors before going to the next iteration over the radius, since, as highlighted in the implementation section, this evolution is intrinsically serial with respect to the radius.

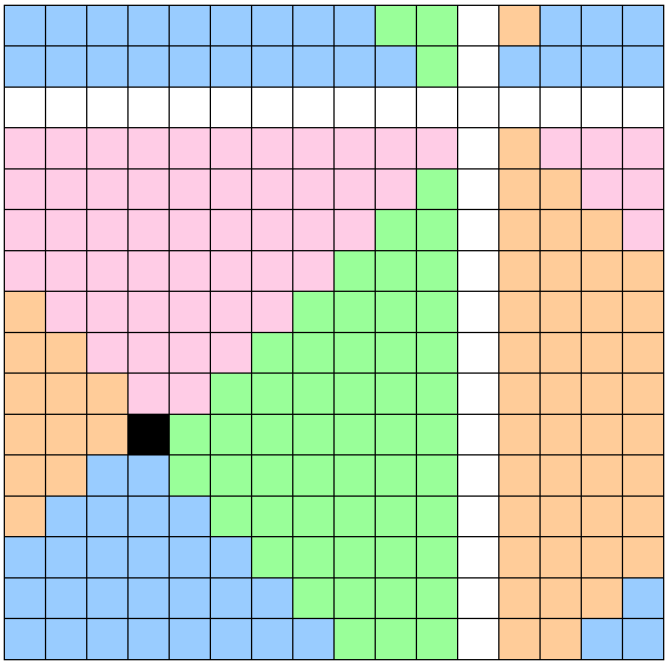


Figure 27: Ideal subdivision of a 16×16 grid with 4 processes.

2 GEMM benchmark

2.1 Introduction to the problem

In the following section, we aim to evaluate and compare the performance of three mathematical libraries, MKL, OpenBLAS, and BLIS (AMD's implementation of the BLAS library), over the level 3 BLAS (Basic Linear Algebra Subprograms) function, `gemm`.
The `gemm` (General Matrix Multiply) function is a fundamental operation within linear algebra and plays a pivotal role in matrix computations. In this context, `gemm` enables the multiplication of two matrices, A and B, resulting in the computation of matrix C using the following equation:

$$\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta$$

The function parameters include α and β as scalars and matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, where $\mathbf{A}$ is an $M \times K$ matrix, $\mathbf{B}$ is a $K \times N$ matrix and $\mathbf{C}$ is an $M \times N$ matrix.

2.1.1 The provided implementation

The code utilized in this report was directly provided alongside the exercise assignment materials. It serves as the foundational framework for conducting comprehensive evaluations and comparisons of mathematical libraries.
As a consequence, in the following analysis, all the matrices will be considered square, introducing the parameter `k` to denote their size. In addition, the parameters will be set in the following way: $\alpha = 1$, $\beta = 0$.
The provided code utilizes conditional compilation directives to support computations in both single and double precision. By defining `USE_FLOAT` or `USE_DOUBLE` flags, the code dynamically switches between single precision (float) and double precision (double) arithmetic, allowing flexibility in the numerical precision employed during `gemm` computations. This adaptation is achieved through preprocessor directives that conditionally define the data type `MYFLOAT`.
The function, invoked through the CBLAS interface (the interface of BLAS for C), showcases matrix initialization, multiplication, and timing computation, providing insights into computational efficiency and performance metrics such as GFLOPS (GigaFLOPS, or billions of floating-point operations per second) for a given matrix size, computed in the following way:

$$\text{GFLOPS} = \frac{2 \times M \times N \times K}{\text{elapsed time}} \cdot 10^{-9}$$

Parallelization To benefit from the multiple and increasing amount of cores, parallelization has been implemented in the provided code, by simply adding `#pragma omp for` directives of OpenMP in the for loops, after creating the parallel region with `#pragma omp parallel`.
Note that all these evaluations have been performed with **one single process**, exploiting parallelization only via OpenMP.

2.1.2 Goals

Our evaluation primarily concentrates on analyzing the scalability and efficiency of `gemm` calculations across varying matrix sizes and different numbers of CPU cores on a specified architecture. The assessment is crucial to understanding the libraries' behavior under different computational loads

and parallelization strategies. The theoretical peak performance of the processors used is also taken into account and set as a point of reference.

The evaluations start with an analysis of the scalability of `gemm` calculations across a spectrum of matri with increasing *size* parameters, maintaining a constant core count. Each measurement was taken 5 times: the following charts depict their means and standard deviations.
The assessments were conducted in single and double precision using MKL, OpenBLAS, and BLIS. The obtained results were scrutinized for scaling behavior and efficiency, to be finally confronted with the peak performance capabilities of the processor. Additionally, diverse thread allocation policies were explored, namely:

- `false`, the OS decides the threads' placement, allowing them to migrate.
- `close`, threads are assigned to consecutive places by their thread ID.
- `spread`, where threads are intuitively allocated, as the name suggests, spread among the node.

Another side of the analysis involved studying the scalability of `gemm` calculations increasing the number of cores, keeping matrix size constant at an intermediate value. Like the previous evaluations, the assessments comprehended single and double precision using all three libraries.

Architecture and peak performance of the processors. Both the previous tasks are exploited in two different architectures: THIN and EPYC nodes of the ORFEO Data Center[3].

THIN nodes are Intel-based, mounting in each of their 2 sockets a *Intel Xeon Gold 6126 12-Core*, with a theoretical total peak performance *for the entire node* of 1.997 TFLOPS (in double precision).
We recall that peak performance is obtained with the following expression [5]:

$$FLOPS = \text{cores} \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{FLOPs}}{\text{cycle}}$$

where $\frac{\text{FLOPs}}{\text{cycle}}$ is the number of floating point operations per cycle, and it depends on the precision used (note that FLOPs is the plural of floating point operations, while FLOPS is the number of f.p. operations *per second*). Usually, for single precision (where it's called FP32) this amount is double the one over double precision (FP64).
The peak performance is then linearly dependent on the number of cores (24, in the case of the THIN node), so given the theoretical peak performance (*t.p.p.*) of the entire node, we can calculate the *single core* peak performance of 83.21 GFLOPS. To obtain the same result in single precision, we just multiply this number by 2.

On the other hand, EPYC nodes are AMD-based; they are composed of two sockets, each one containing an *AMD EPYC 7H12*, a 64-core processor with 16 FP64 (AMD Rome[5]). AMD is claiming the Base Frequency for this processor is 2.6 GHz[1], which when multiplied by the FLOPs per cycle gives us 41.6 GFLOPS *on a single core*. This result multiplied by the 128 cores of this node returns 5.3248 PFLOPS in double precision (double that amount in single precision).

2.2 Scalability over matrix size

As said in the Introduction, in this section we will be scaling over an increasing value of k, from 2000 to 20000, analyzing the behavior of two measured numerical variables, namely:

- time to compute a matrix multiplication with the given k and library.
- average GFLOPS during the entire time of this computation.

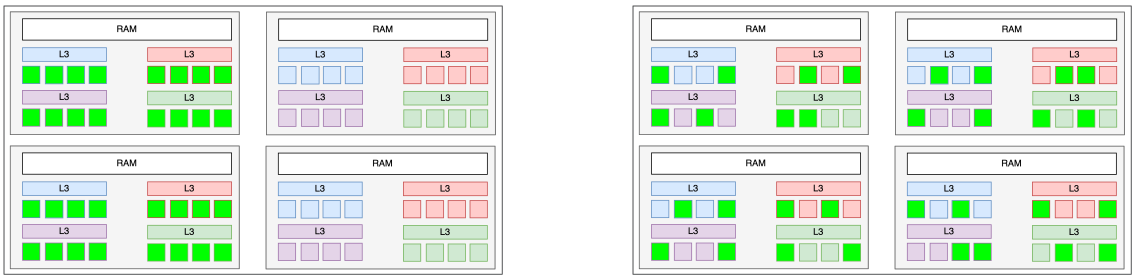
The number of cores is set as half of the entire number of cores of the node, so 12 for THIN and 64 for EPYC. However, all resources will be requested, with the following options in the batch script: `--cpus-per-task=$max_num_cpus`, where `$max_num_cpus` is a placeholder for 24 in THIN nodes and 128 in EPYC ones.

Previously we found the theoretical peak performance of both architectures; given that we now use only half of the available cores, we divide that result by 2 to obtain the t.p.p. in Table 1, for clarity.

	THIN	EPYC
single	1996.8	5324.8
double	998.4	2662.4

Table 1: Theoretical peak performance in GFLOPS for each node, in single and double precision, utilizing half of the cores.

To have a better understanding of two opposite policies for thread placement, we provide an example of how this placement would happen if, running on a single EPYC socket (64 cores), we requested only half of them. For simplicity, the internal division of hardware threads in each core is not reported.



(a) `OMP_PLACES=cores`, `OMP_PROC_BIND=close`

(b) `OMP_PLACES=cores`, `OMP_PROC_BIND=close`

Figure 28: Comparison between close and spread policy on a socket of an EPYC node, i.e. mapping of 32 threads on a socket with 64 cores.

2.2.1 Analysis on THIN node

Slicing over library This analysis keeps the categorical variable "library" fixed at one of its values (OpenBLAS, MKL, or BLIS), to observe in each case the behavior of the combinations of precision and thread allocation policies.

As we can observe from Figure 29, there's no noticeable difference between thread allocation policies, over any of the libraries. Furthermore, we can already see the poor performance of the BLIS library in the THIN node, behavior that will become more apparent in the following charts.

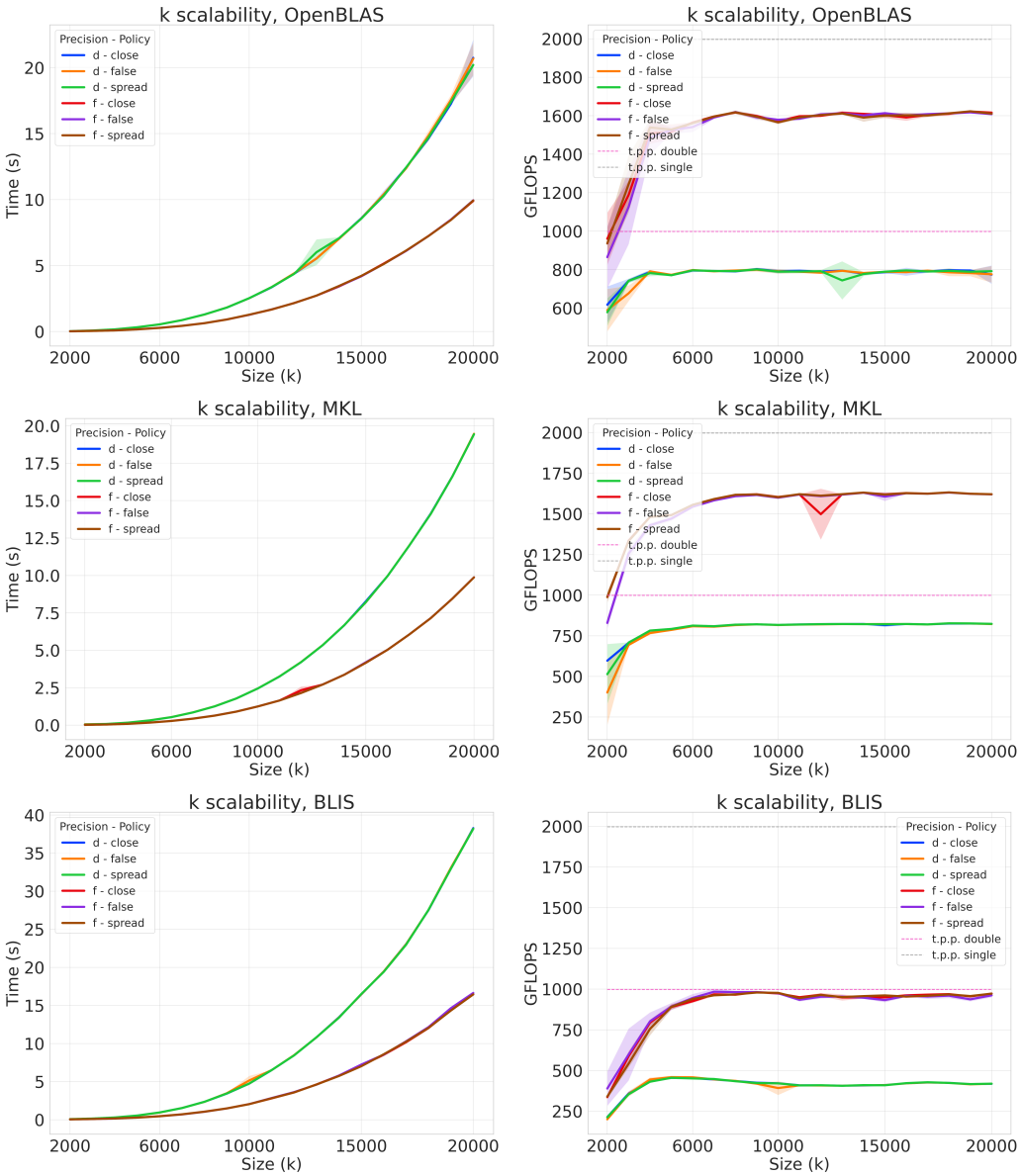


Figure 29: Scalability of k over single/double precision and thread allocation policies.

Slicing over thread allocation policy This analysis keeps the categorical variable "policy" fixed at one of its values (false, close, or spread), to observe in each case the behavior of the combinations of library and precision.

As seen in the previous slicing, the influence of thread allocation policy in this particular exercise is negligible. However, in Figure 30, we reported each case for completeness, even if the results

between graphs are very similar (in the sense that in each case we can infer the same results). In particular, it's clear how the performance of the BLIS library is poor compared to the other two libraries, taking almost double the time both in single and double precision. This results in an expected value of GFLOPS very low, such that the average result for *single* precision is along the theoretical peak performance of the *double* precision.

Regarding MKL and OpenBLAS libraries, they perform almost identically, with the MKL library

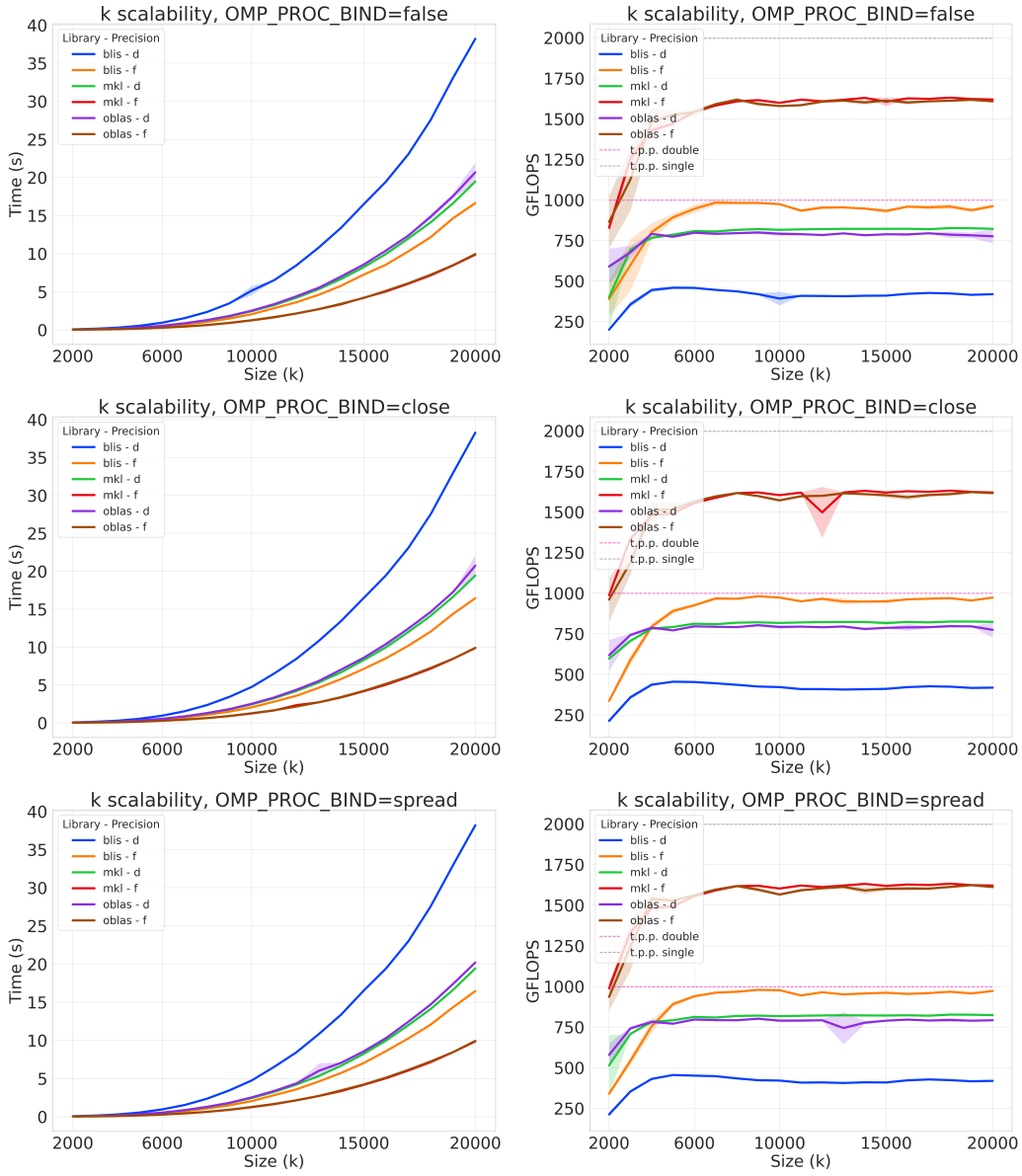


Figure 30: Scalability of k over single/double precision and different libraries.

performing slightly more both in single and double precision. Both libraries return an expected value of GFLOPS at around 80% of the theoretical peak performance.

2.2.2 Analysis on EPYC node

We will follow the same procedure we used for the THIN analysis, slicing each time on a categorical variable, and plotting different lines over the combination of the other variables.

Slicing over library This analysis keeps the categorical variable "library" fixed at one of its values (OpenBLAS, MKL, or BLIS), to observe in each case the behavior of the combinations of precision and thread allocation policies.

Taking into account Figure 31, we can see how, in the EPYC node, when operating with half of the total cores, the "false" policy (i.e. each swthread allocation is managed by the OS and they are allowed to migrate) performs slightly worse than the other two policies taken into account. This performance gap is clear, especially in the case of OpenBlas and BLIS libraries. The close and spread policies perform similarly.

Finally, the poor performance of the MKL library in the EPYC node is already noticeable (more on that later).

Slicing over thread allocation policy This analysis keeps the categorical variable "policy" fixed at one of its values (false, close, or spread), to observe in each case the behavior of the combinations of library and precision.

Looking at the results of this analysis in Figure 32, we can see how, in the EPYC node, the policy allocation produces small changes in the performance of the libraries. In particular, with close and spread policies, in double precision, the BLIS library is always faster than the OpenBLAS library; whereas, with `OMP_PROC_BIND=false`, BLIS is noticeably slower; this is also obvious in Figure 31.

On a more general note, the MKL library is always the worst performing, with average GFLOPS between 25% and 45% of the theoretical peak performance.

BLIS library is almost always the leading, at around 75% of the t.p.p., with a big gap from OpenBLAS library in the case of single precision.

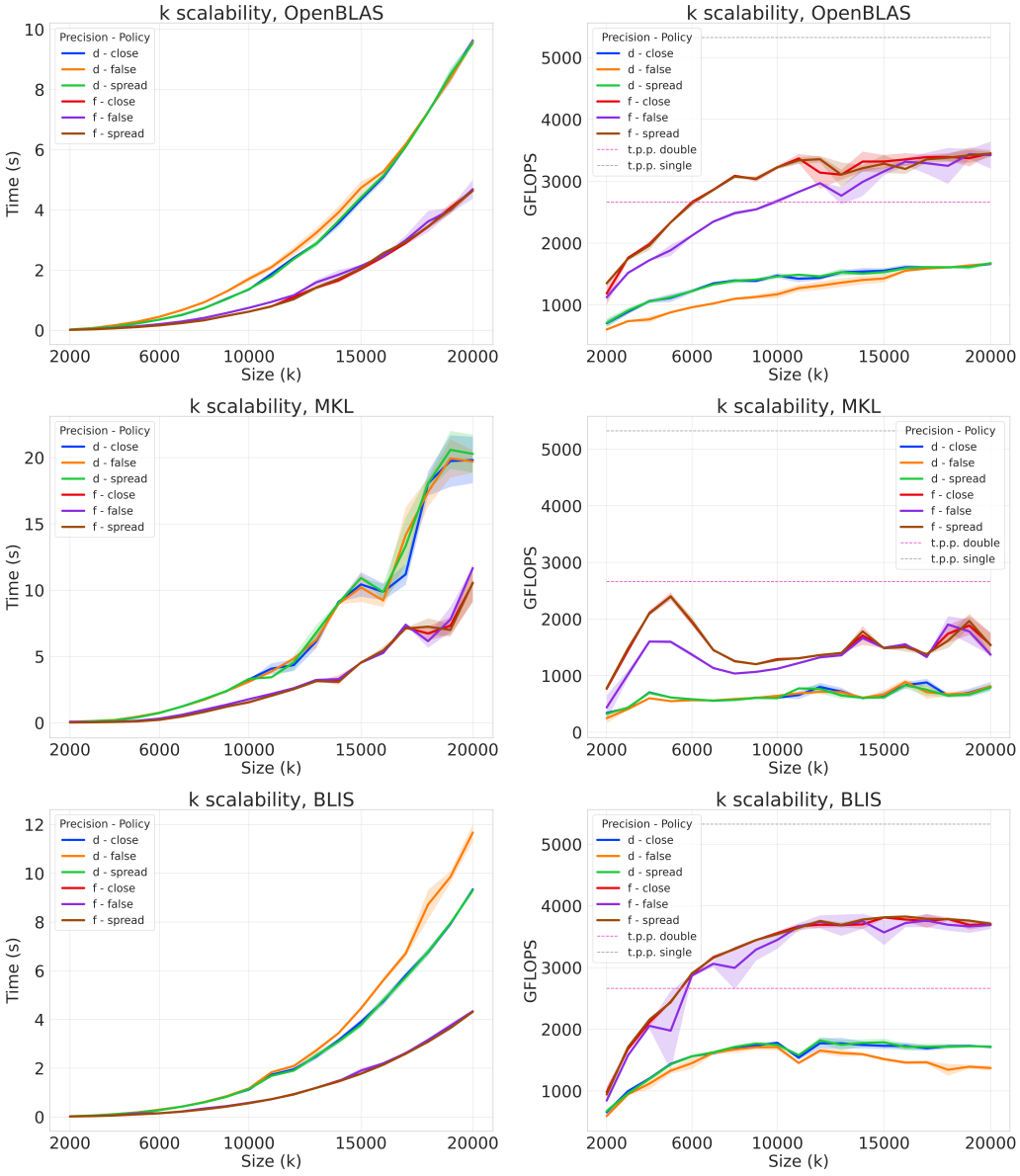


Figure 31: Scalability of k over single/double precision and thread allocation policies.

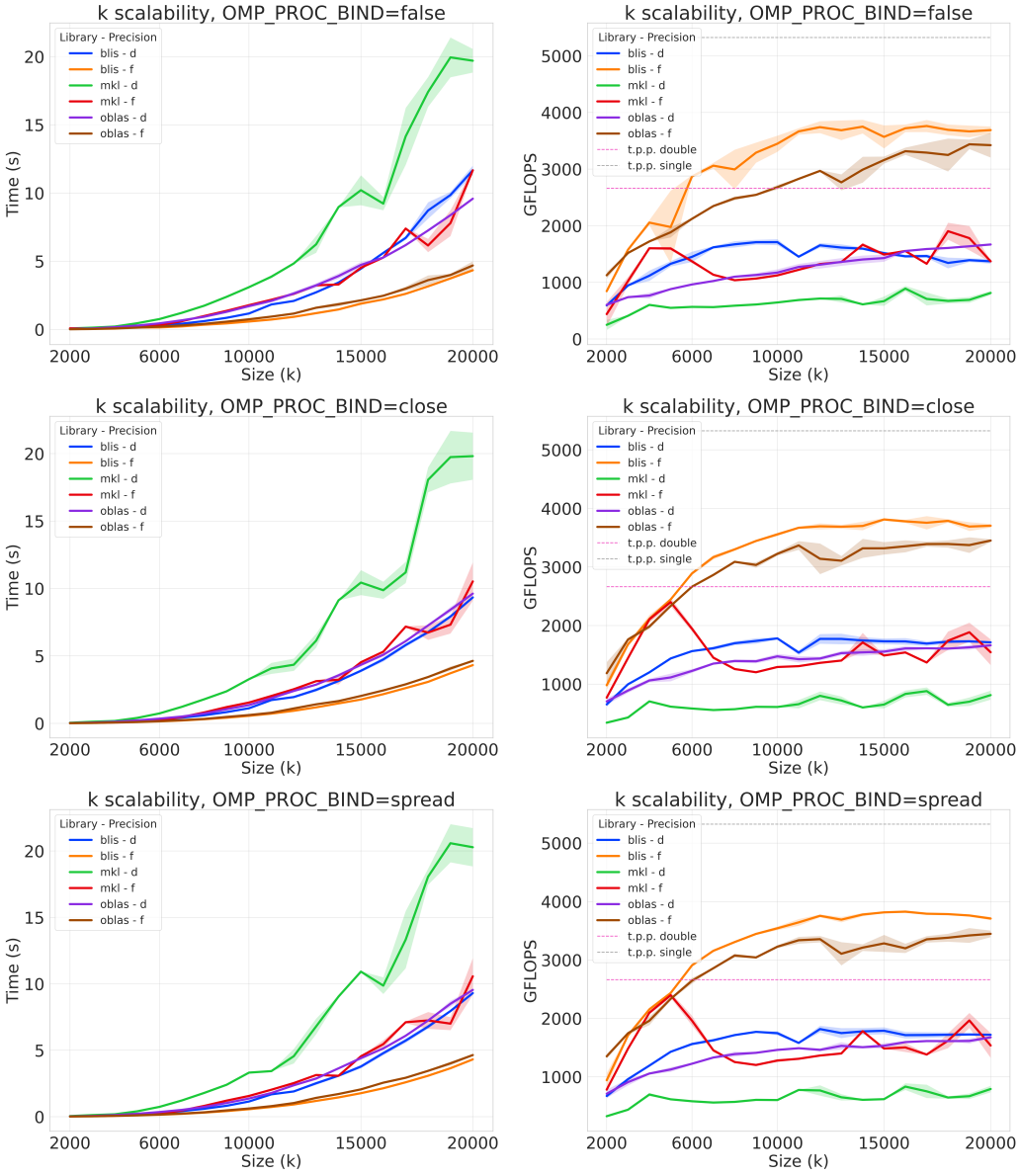


Figure 32: Scalability of k over single/double precision and different libraries.

2.2.3 Comparison between the two architectures

The focus of the analysis now migrates to comparing the **absolute** performances of the two nodes; a more balanced evaluation will be provided in the following section. Notably, each node is considered operating with a number of threads corresponding to half of its cores.
Since the "false" policy does not perform on par with "close" and "spread", and since the latter has given similar results in both nodes, the "spread" policy is chosen in the comparison setting.

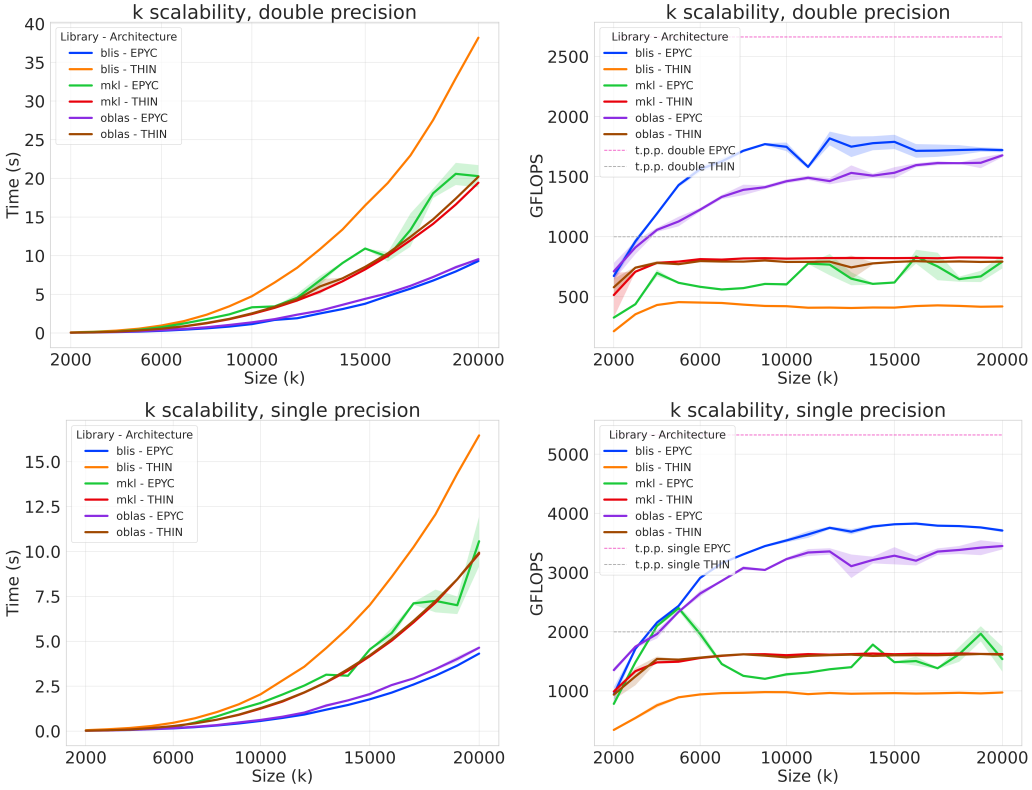


Figure 33: Scalability over k, using spread policy, with single and double precision.

From Figure 33 it's obvious that the sheer size of the EPYC node influences its results, with faster computations using OpenBLAS and BLIS libraries. MKL library, however, performs similarly to the best-performing libraries in THIN, MKL, and OpenBLAS, with higher variance.
When confronting the actual measured values for the average GFLOPS in both nodes, however, it's clear that the THIN node is more consistent and reaches a greater percentage of the theoretical peak performance.

2.3 Scalability over the number of cores

As previously said in the introduction, the goal of the following study is to analyze the performance of a given node when increasing the number of computing cores. For this task, the size parameter

	THIN	EPYC
single	$166.4n$	$83.2n$
double	$83.2n$	$41.6n$

Table 2: Theoretical peak performance in GFLOPS for each node, in single and double precision, with n the number of cores utilized.

`k` has been set to an intermediate value of 10000. In all three libraries previously cited, the `gemm` function has been applied with either single or double precision.

Previously we found the theoretical peak performance of both architectures, we group the results needed for the scalability over `n_cores` in Table2, for clarity. Alongside the charts plotting the times and the GFLOPS, the efficiency and the speedup plots will be presented.

2.3.1 Analysis on THIN node

The number of cores ranges from 1 to 24. As we can see from Figure 34, the THIN node is fairly

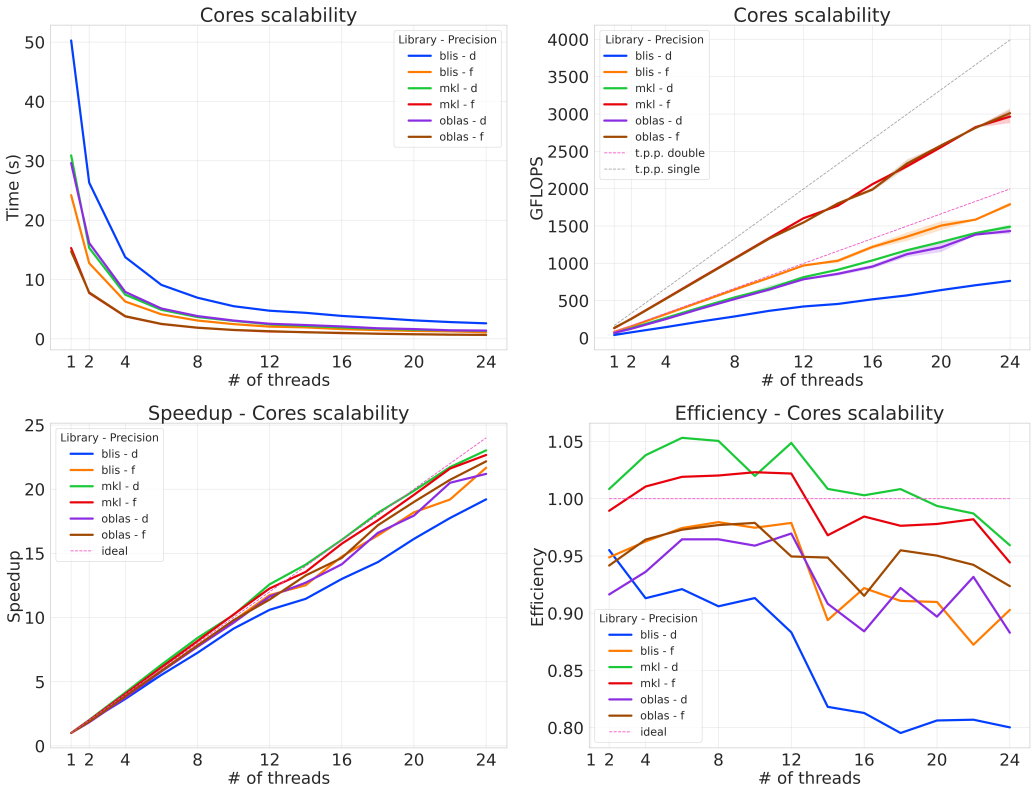


Figure 34: Scalability over the number of cores, with single and double precision.

consistent in its scalability results.

Observing the speedup plot, only the BLIS library with double precision (the worst performing in THIN, Section 2.2.1) deviates significantly from the ideal speedup. The same observation can be obtained from the efficiency plot.
Regarding the comparison with the theoretical peak performance of this node, the data points to an exploitation of t.p.p. of about 75%.
Finally, we confirm our previous analysis that the MKL library is slightly better performing than the OpenBLAS, lying rather consistently on the ideal lines.

2.3.2 Analysis on EPYC node

The number of cores ranges from 1 to 128. To facilitate the visualization of the results, in the case of EPYC nodes, slicing over the precision has been performed.

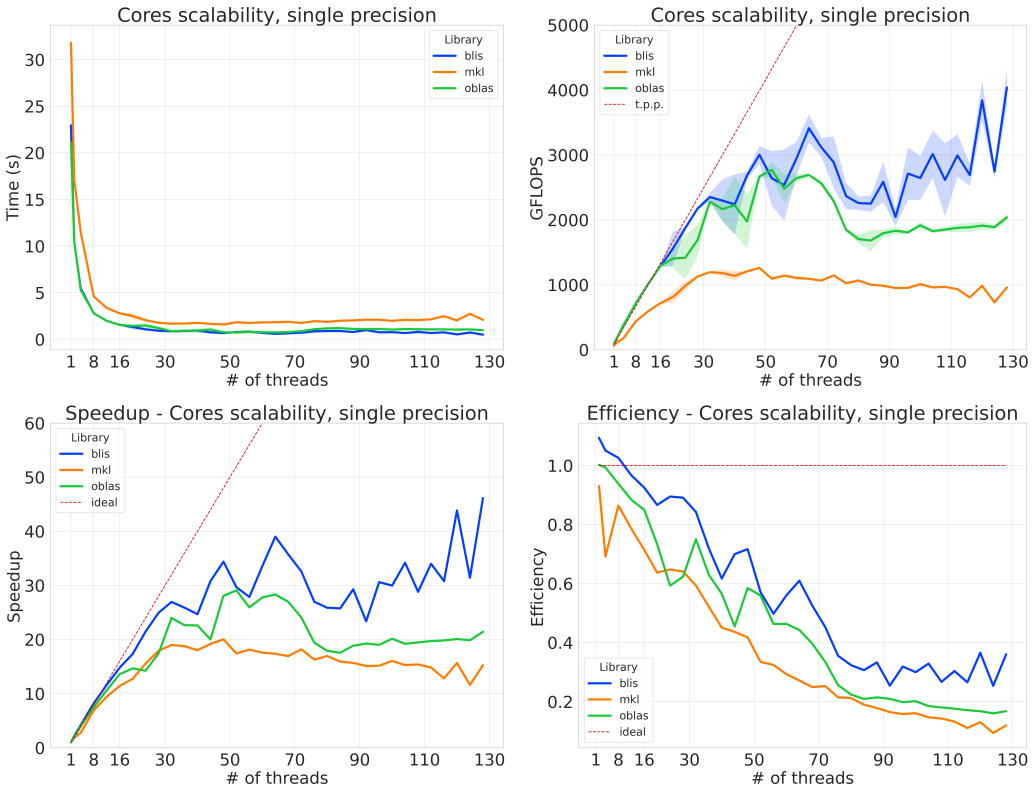


Figure 35: Scalability over the number of cores, with single precision.

Looking at Figures 35-36, the time graph initially displays a consistent decreasing trend as the number of threads increases. However, once the number of threads surpasses a certain point, varying for each library, but around 50, the time taken for both double and single precision doesn't seem to decrease consistently anymore.

Focusing on the GFLOPS plots, we can observe that when using up to 30 threads, the BLIS library achieves results close to the theoretical peak performance. For the OpenBLAS library, this

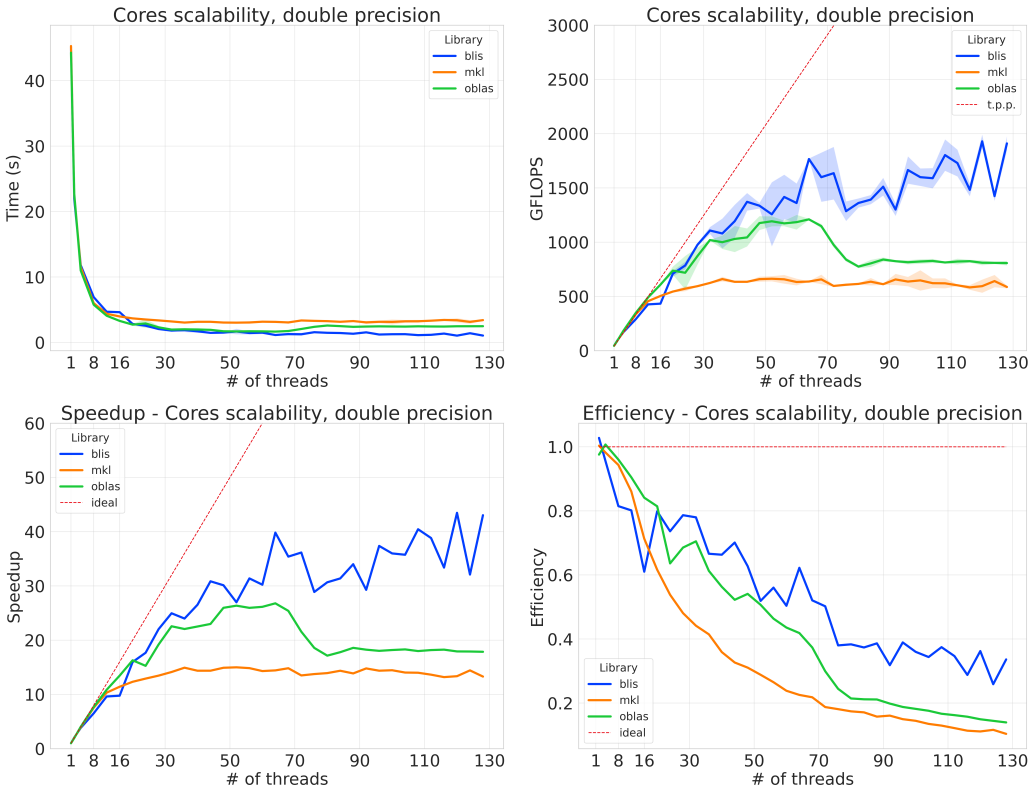


Figure 36: Scalability over the number of cores, with double precision.

number is closer to 20, while for the MKL implementation, the scaling deviates almost immediately from the ideal performance. This is also a result we underlined in the previous analysis in Section 2.2.2 When exceeding this threshold of threads, the behaviors of all libraries blend in: they present more variance while continuing on a trend of non-increasing performance. Regarding speedup, the plots offer similar insights as the GFLOPS ones, reinforcing the observations made. With a higher number of cores, in fact, performance detaches more and more from the ideal scalability.

2.3.3 Comparison between the two architectures

To provide a balanced mean of comparison between the two architectures, we will confront their results over an increasing number of cores, from 1 to 24.

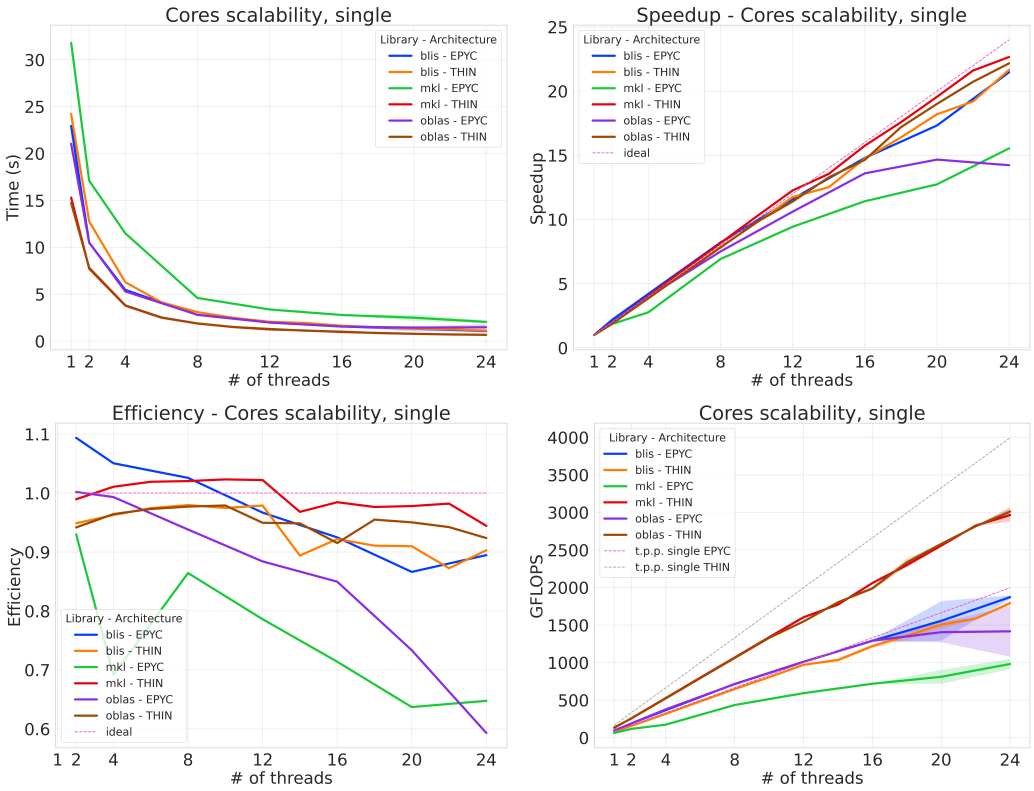


Figure 37: Comparison between THIN and EPYC nodes, scalability over the number of cores, with single precision.

Charts in Figure 37 and Figure 38 provide a balanced way to compare the performance of the libraries over EPYC and THIN nodes.
First of all, looking at the time and GFLOPS charts, it's clear that the THIN node, running at the same number of cores as EPYC, is faster and performs more operations on average. The only case in which it is slightly slower and on par with the EPYC timings is for the BLIS library, reinforcing

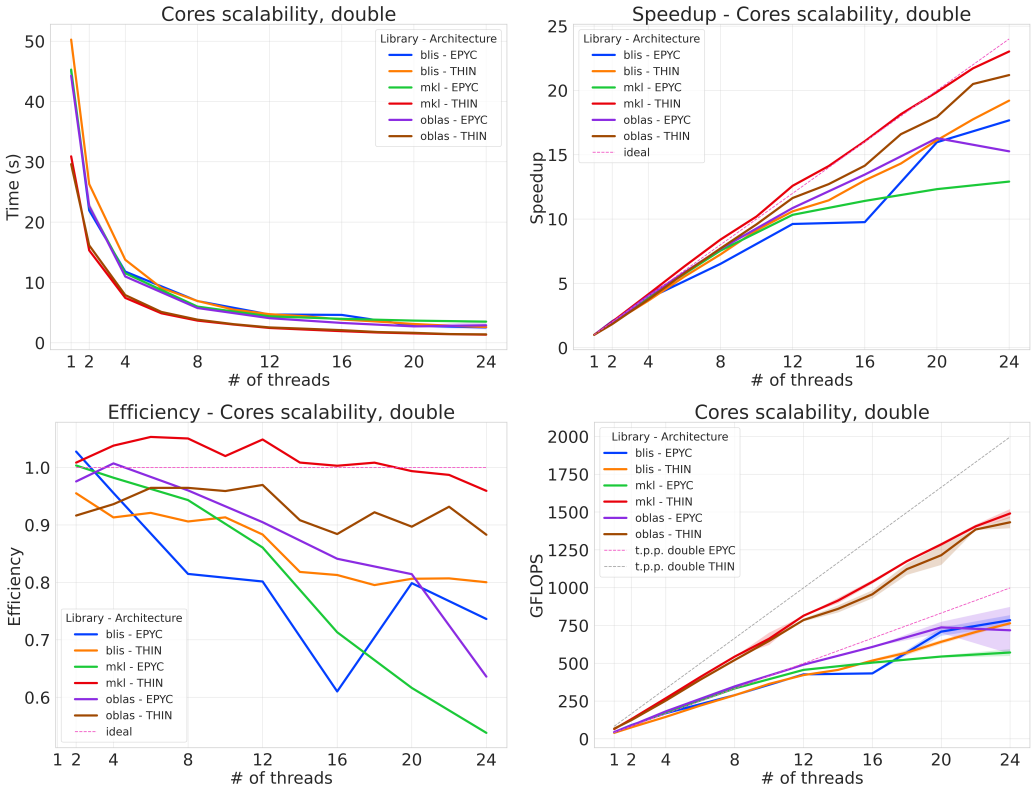


Figure 38: Comparison between THIN and EPYC nodes, scalability over the number of cores, with double precision.

once again our assessments of its generally bad performance on this node. Upon further analysis, however, we can see how this faster computation is due to the higher number of FLOPs that the Intel Xeon Gold 6126 has: 32 FP64 [5] against the 16 of the AMD EPYC 7H12. This is the reason why the theoretical peak performance of the Intel processor is steeper than the AMD one.
Taking this into account, we can see how the GFLOPS registered on EPYC are on average closer to the theoretical peak performance of the CPUs and deduce that, despite the bigger variance, the EPYC node is (at a lower number of cores) efficiently utilizing its computational capabilities. This performance level signifies that the processor is executing floating-point operations at a rate near its maximum designed capacity.
However, looking at the speedup and efficiency plots, especially for the double precision, we can see how the Intel architecture is more consistently around the ideal trajectories (excluding the BLIS library), displaying also less variance.

2.4 Conclusions

From these graphs, it's clear that the ideal library for matrix operations really depends on the kind of computer hardware you're working with. The results varied a lot based on the software we used. For instance, on the THIN nodes, the MKL library stood out as the top performer among the options. But on EPYC nodes, MKL didn't do so well—it actually performed poorly. On EPYC, the BLIS library sometimes outperformed the others, while also being the worst-performing library over THIN nodes. OpenBLAS did consistently well on both types of architectures.

One interesting takeaway was that simply adding more threads didn't always make things faster. Sometimes, choosing the right number of threads made a big difference in how well these libraries worked. The ideal scenario is finding the thread number in which, looking at the GFLOPS scalability over the number of threads plot, there's an inflection point in a knee-like plot. This way, a reasonably high value of operations is performed, while maintaining efficiency and not wasting resources.

What we can finally infer is that the best library choice for these matrix operations heavily relies on the specific hardware you are working with. These findings emphasize the importance of performing benchmark analysis within the framework of parallel computing, to be able to efficiently exploit resources and libraries available.

References

[1] AMD. Amd epyc™ 7002 series processors. https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf.

[2] Nathaniel Johnston. https://conwaylife.com/.

[3] Area Science Park. https://www.areasciencepark.it/2021/09/data-center-orfeo/.

[4] Wikipedia. Conway's game of life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[5] Wikipedia. Flops. https://en.wikipedia.org/wiki/FLOPS#cite_note-www.youtube.com-16.