Project
Foundations of High Performance Computing

Malcapi Enrico [SM3500561], Marsich Gaia [SM3500600]

Course of A.A. 2022 - 2023, Data Science and Scientific Computing

Repository with all the files: https://github.com/gmarsich/HPC-project

Contents

Part I

Assignment 1

1 Introduction

The goal of this first assignment was to implement a parallel version of "*the Game of Life*" and provide a scalability study.
The program handles the evolution of a grid of cells observing a specific update rule: basically, a cell will be alive or dead in the following step depending on the condition of the other eight surrounding cells. However, the specific update rule that was suggested for this assignment slightly differs from the standard one. The parallelization implemented in the program was an hybrid between MPI and OpenMP: as a matter of fact, the workload is splitted among different MPI processes and each of them spawns OpenMP threads.
Finally, we investigated the performances of our program with a scalability study, increasing both the MPI processes and the OpenMP threads for each process and considering different sizes of the problem.

2 Problem statement

2.1 The Game of Life

The Game of Life is a cellular automaton discovered by John Conway. It consists of a grid of $n \times m$ cells where each element is characterized by a state: either a cell is dead or it is alive. Starting from an initial set-up of the grid, at each step the cells will be subject to an evolution of their state according to the situation of the eight surrounding cells: see Figure 1 to better understand which is the neighborhood to be considered.

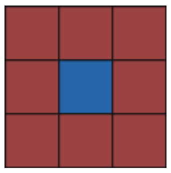


Figure 1: The neighbors (red) of a specific cell (blue)

In our case we are considering a general $k \times k$ squared grid with periodic boundaries: this means that to update the first row and the last row of the grid one needs to consider, respectively, the last row as it were the row right before and the first row as if it were the row right after; the same trick needs to be applied to the first column and the last column. The exact update rule we have been referring to differs from the standard in a few respects. In fact, observe that:

- Standard update rule:

  - any alive cell with less than two alive neighbors (underpopulation) or more than three live neighbors (overpopulation) will be marked as dead
  - any dead cell with **exactly three** alive neighbors (reproduction) will be marked as alive

- Our update rule:
  - any alive cell with less than two alive neighbors (underpopulation) or with more than three alive neighbors (overpopulation) will be marked as dead
  - any dead cell with **two or three** alive neighbors (reproduction) will be marked as alive

The state of a cell will then somehow affect the update of its neighbors. The program allows the user to decide between two possible processes of evolution:

- Ordered: the updates on the grid are performed sequentially cell after cell, following a row-major order: each element must wait for the evolution step of the previous one before updating itself. Therefore it will be affected by the new updates of the cells that have already been processed: a spurious signal will be generated, then, because the update of a specific cell depends on the new states of some cells and, on the other hand, on the current states of the remaining ones
- Static: a new grid is created. Here, the result of the evolution will be written: the grid of the previous iteration is taken as a static reference to determine the new states of the cells

2.2 PGM Format

The grid of cells is visualized and given as input to the program as a $k \times k$ image where each pixel corresponds to a cell. The state is represented by the color of the pixel, that can be either white (dead cell) or black (alive cell). The chosen format for the image is the `PGM` and is described by the following structure:

```
P5
#HEADER
number_of_columns number_of_rows
max_value
pixels_values
```

The color of a pixel is identified by either one char (1 byte) or two chars (2 byte) depending on the format. In both cases only a gray scale is available: the difference between the two choices is the range of the nuance. All the values are inserted in a single line and `max_value` represent the maximum value as integer number that can appear (255 for one char, 65535 for two chars). In our case since we just need the pixel to be either black or white the one char representation is enough: value 0, black, will be for alive cells and 255, white, for dead cells.

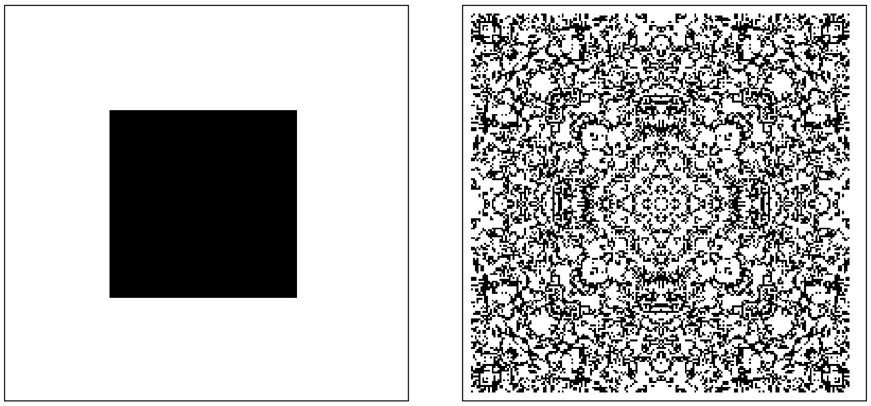

(a) Initial set-up (b) Evolution

Figure 2: Initial set-up of a 200×200 grid and its evolution after 100 iterations

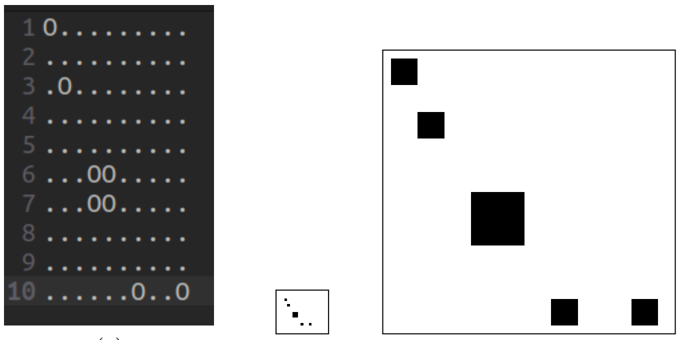
2.3 Initial set-up

As initial set-up for the evolution we choose to set the central region of the image with alive cells: in particular, the alive cells will form a square with side equal to half the side of the grid. At a certain iteration of the evolution, we will observe the situation shown in Figure 2. This specific layout allows to easily produce an initial configuration of arbitrary dimension, making possible to study the performance for different sizes of the problem.
Since the evolution of this set-up is not so intuitive we imagined a sort of environment that we called "*playground*" (in practise, it is a `.txt` file), useful to check the correctness of the evolution and also to create a new initial set-up. To create a general *playgroud* of a $k \times k$ grid one must proceed as follows:

- Create a `set_up.txt` file where to describe the desired grid configuration using `.` to denote a dead cell and `O` (big *o*) for an alive cell. With these two symbols one has just to write down the matrix, for example like:

```
..O.
.OO.
O.O.
....
```

- Use the command `./converter x y c` to launch the program that will convert `set_up.txt` into the proper image. The parameters to give as inputs are:
  - x: number of rows
  - y: number of columns
  - c: size of a cell in pixels (the dimension of a cell would be $size \times size$)



```
O.........
..........
.O........
..........
..........
...OO.....
...OO.....
..........
..........
......O..O
```

(a) (b) (c)

Figure 3: Image scaled (c) and not scaled (b) produced by an initial configuration (a) of dimension 10×10

With this procedure two images will be produced, `grid.pgm` and `grid_scaled.pgm`, respectively representing the real and scaled version of the desired grid configuration, see the example shown in Figure 3.

3 The parallelization

We took into account the fact that we deal with periodic boundaries so, for example, the first line of a new iteration also depends on the last line of the previous iteration. For this reason we supposed that thinking about a domain decomposition of the image/grid would have been a good path to follow to implement the parallelization.
Since for storing the image and the grid a row-major approach is used (so the values are stored in memory row after row, not column after column) we decided to divide the image by the rows. In this way each MPI process will work on a portion of the image composed by an appropriate chunk of rows. For the static evolution[1] each process will further divide its workload among a certain amount of OpenMP threads. The overall parallelization schema is shown in Figure 4.

4 Implementation

4.1 Overall description of the program

The files required for the compilation of the program are the following:

- the two `.c` files `game_of_life.c` and `converter.c`, respectively the main file for the Game of Life implementation and the program that converts the file of the *playground* into a PMG image

[1]The ordered evolution will not produce OpenMP threads since as already illustrated this procedure is intrinsically serial: also the MPI processes will have to wait for the updates of the previous portions of the image before starting their own update.

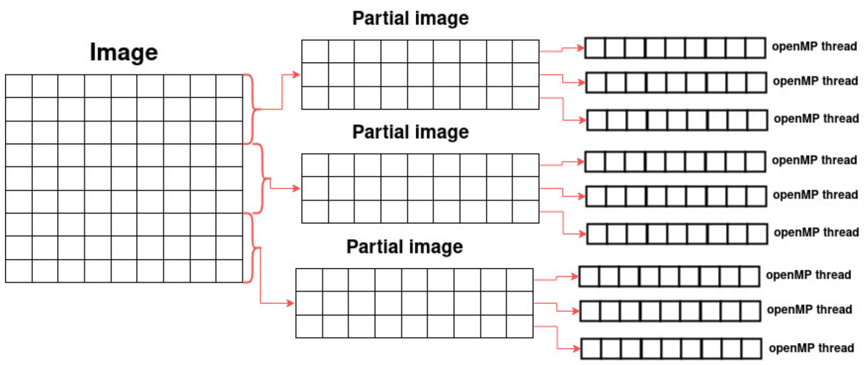


Figure 4: Schema representing the domain decomposition

- a folder containing the three headers used in `.c` files:
  - `header_general.h`: containing general functions used in the `main` of `game_of_life.c`
  - `header_update_rules.h`: with all the functions implementing the updates of the cells
  - `header_converter.h`: for all the functions used in the `main` of `converter.c`
- a `Make` file to compile everything

The `Make` file will compile the two `.c` programs linking them with the headers and it creates a folder called `results`, required to save the results of the iterations of the evolution. The two resulting compiled programs are `game_of_life.x` and `converter.x`. The `Make` file has also a "clean option" (`make clean`) to remove old files: previously compiled files, the all the images, the `results` folder and the `data.txt` file.
The `game_of_life.x` program has two functionalities depending on the parameters given as input:

- initialization of the initial set-up of a general $k \times k$ grid: this is done by using the following options:
  - `-i` to use the initialization functionality (as already presented in the Figure 2)
  - `-k <k_value>` parameter to describe the dimension of the $k \times k$ grid
  - `-f <name>` to set the name of the produced output (i.e., the initial set-up image)
- run of n steps of evolution given a starting image: this is done by using the following options:
  - `-r` to use this functionality (instead of the one to initialise the initial set-up)

  – `-e` to use the ordered evolution (by default it uses the static evolution)
  – `-n <number_of_iterations>` to give as input the desired number of steps for the evolution
  – `-s <frequency>` to indicate the frequency of savings of the evolution (by default it saves just the last step of the evolution)
  – `-f <name>` to provide the name of the image to be taken as initial set-up
  – `-c <cell_size>` if a scaled version of the result is desired (the results will be produced with each cell of dimension `cell_size`×`cell_size`)

The program will also print on the screen in the standard output some useful information: the chosen options, the dimension of the image, the number of MPI processes, the number of spawned threads for each process and the total time taken by the evolution. In addition, when the evolution mode `-r` is given the program will also create a `data.txt` file made of a series of rows: each time that the `game_of_life.c` is run in the evolution mode, a new row will be appended. Each line is composed by the indication of the number of MPI processes, the number of threads for each process and the mean of the total time taken by the evolution (the evolution is repeated 5 times).

4.2 Pseudocode and structure of the program

To implement the Game of Life program we used the following schema, a general idea of the procedure:

```
Check the options;

if option -i was given:
    Create the initial set-up;

if option -r was given:
    Read the provided initial set-up;
    Refresh the results folder;
    Set the grid to the initial set-up;

    for k in [1, number_of_iterations]:
        Allocate the New_Grid;
        Update the central rows of the New_Grid;
        Update the last and first row of the New_Grid;
        Grid=NULL;
        Grid=New_Grid;
        if k%frequency==0:
            if option -c was given:
                Scale the result;
                Save the result in the results folder;
            else:
```

```
22                  Save the result in the results folder;
23
24      if option -e was given:
25          Read the specified initial set-up;
26          Refresh the results folder;
27          Set the grid to the initial set-up;
28
29          for k in [1, number_of_iteration]:
30              Update the first row of the grid;
31              Update the central rows of the grid;
32              Update the last row of the grid;
33              if k%frequency==0:
34                  if option -c was givem:
35                      Scale the result;
36                      Save in the results folder;
37                  else:
38                      Save in the results folder;
```

During the operation highlighted in red, at line 13, the process will spawn more OpenMP threads to further parallelize the work. Observe that the pseudocode we provided is only a hint of how everything should work: for example, some operations will be performed just by the master MPI process. Moreover, if one wants to have more processes working on the image then some message passing procedure should be introduced to correctly manage and put together the data coming from the different processes, since each of these ones will deal with only a part of the initial image. Those procedures are described in the message passing procedures section (**4.6 The message passing procedures**).

4.3 Grid representation and memory allocation

The input of the program is an image in `PGM` format and in such format the image is stored in a string of chars, so it seemed a sensible idea to use an array of chars to represent the grid. In this way the elements of the grid are stored in memory following a row-major order: to access the element at the i-th row, j-th column, we used the following formula: $k(j + i * n_{col})$ with k describing the dimension of the grid (we deal with a $k \times k$ grid) and n_{col} is the total number of columns.
Considering this representation, the procedure is:

- read the image with the function `void* read_pgm_image(int* xsize, int*ysize, const char*image_name)` that extracts the image dimension from the `PGM` file and returns the pointer to the first element of the memory region containing the image
- each process will allocate a char pointer named `Grid`, pointing to the first char of the (partial) image that has been assigned to the process: `Grid = ((char*)image+rank*partial_dimension)`

- at the beginning of each iteration of the evolution a `NewGrid` char pointer will be allocated, and here the results of the associated process (considering the current step) will be stored
- at the end of the evolution step the `Grid` will be diverted to the memory address pointed by `NewGrid` and, afterwards, `NewGrid` will be set to `NULL`

4.4 The update rule

The update rules are implemented through four functions contained in the `header_update_rules.h` header. Remember that the update of a cell requires to know the state of its eight neighbors (as already described in section **2.1 The Game of Life**) and that we assumed to have periodic boundaries. The four functions that we inserted in the header are:

- `updateUP`

```
     void updateUP(const int col,
                   const int x_size,
                   const int y_size,
                   char* OldGrid,
                   char* NewGrid,
                   char* upper)

```

  that updates the element at the *col*-th column, first row, of the partial image. Since each process deals with just a part of the initial image, the update of its first row needs the information contained in the last row of the partial image located right before (take also into account the fact that we assumed periodic boundaries). A buffer called *upper*, one per process, has been designed to receive and keep in memory this external row

- `updateDOWN`

```
       void updateDOWN(const int col,
                       const int x_size,
                       const int y_size,
                       char* OldGrid,
                       char* NewGrid,
                       char* lower)

```

  has a similar purpose as `updateUP`, but in this case it is the last row of the partial image, not the first one, that needs to be updated. Analogously, also here each process has been provided of a buffer, called *lower*, to store the first row of the partial image coming right after

- `updateCell`:

```
       void updateCell(const int row,
                       const int col,
                       const int y_size,
                       char* OldGrid,
                        char* NewGrid)

```

which is used to update the central parts of the partial image, the ones that do not need to receive information from other processes (i.e., all the lines except for the first and the last row)

- `updateLINE`

```
        void updateLINE(const int col,
                        const int x_size,
                        const int y_size,
                        char* OldGrid,
                        char* NewGrid,
                        char* upper,
                        char* lower)

```

  that is exploited in the rare case where a process was just assigned a partial image that consists of only one row

4.5 The OMP section

As already mentioned before, during the update of the central rows of the static evolution each MPI process should create other children threads to further parallelize the heaviest work of the evolution. Since the memory allocation is in a row-major order, each thread should work on a chunk of rows. For this reason the complexity of the update is given by the size of rows and parallelizing also the update of the first and last row of the partial image does not decrease the complexity.
To implement the OMP parallelization we could use two possible implementations regarding the memory allocation:

- each thread uses a private buffer where to write its partial result; all the results are then combined into the `NewGrid` pointer, characteristic of the parent process
  - advantages: each thread will write on its private memory region avoiding the risk of false sharing
  - disadvantages: this procedure introduces a buffer for each spawned thread and also an overhead at the end caused by the combination of all the buffers
- each thread writes the result in the correct portion of the pointer `NewGrid`, which will be shared among all the threads
  - advantages: this procedure avoids the creation of further buffers and has no overhead because of way of combining the partial results

  - disadvantages: since each thread is writing on a shared memory region there is the risk of false sharing or of writing on the wrong portion of the `NewGrid`

We decided to adopt the second option as we expected the false sharing to be present only for low dimensions of the problem since the size of the workload for each thread rapidly increases with the size of the `Grid`.
With these decisions we created the parallel region as follows:

```
#pragma omp parallel
{   #pragma omp for schedule(static)
    for(int i = 1; i < (x - 1); i++){
        for(int j = 0; j < (y); j++){
            updateCell(i, j, y, Grid, NewGrid);
        }
    }
}
```

As we can see the outer `for` will be divided among the threads with a static schedule: this choice is due to the fact that the amount of work for each iteration is expected to be the same (check the 8 neighbours for each cell). The number of spawned threads is left to be decided through the environmental variable `OMP_NUM_THREADS`.

4.6 The message passing procedures

In order to have more MPI processes we need to define some procedures for the communications between them.
To recompose the image from the partial images of each process we used the following implementation:

```
if(rank==0){
    Grid=realloc(Grid,sizeof(char)*total_size*total_size);
    for(int z=1;z<size;z++){
        MPI_Recv(Grid+x*y*z,
        (z==size-1)?((x+total_size%size)*y):(x*y), MPI_CHAR,z,0,
        MPI_COMM_WORLD,&status);
    }
    write_pgm_image((void*) Grid,total_size,total_size,image_name);
}

if(rank!=0){MPI_Ssend(Grid,x*y, MPI_CHAR,0,0,MPI_COMM_WORLD);}
```

In this way all the processors will send their partial image in the correct position of the master processor's `Grid` ($rank = 0$) which will contain the total image.
As mentioned before in order to update the first and the last rows all the processes need to have two buffers in which to store the last row of the previous process and the first row of the following process (also the first and the last processes since we are in a periodic boundaries condition). For this reason each process will need to send and receive two rows for a total of 4 message passing procedures. In order to avoid deadlocks we used non-blocking operations with a check at the end like this:

```
MPI_Isend(Grid+(x-1)*y, y, MPI_CHAR,
            (rank+1<size)?(rank+1):0,1,
            MPI_COMM_WORLD,&request_upper_s);
MPI_Irecv(lower, y, MPI_CHAR,
            (rank+1<size)?(rank+1):0,0,
            MPI_COMM_WORLD,&request_lower_r);

MPI_Isend(Grid, y, MPI_CHAR,
            (rank-1>=0)?(rank-1):(size-1),0,
            MPI_COMM_WORLD,&request_lower_s);
MPI_Irecv(upper,y, MPI_CHAR,
            (rank-1>=0)?(rank-1):(size-1),1,
            MPI_COMM_WORLD,&request_upper_r);


MPI_Wait(&request_upper_s, &status);
MPI_Wait(&request_lower_r, &status);

MPI_Wait(&request_lower_s, &status);
MPI_Wait(&request_upper_r, &status);
```

In the ordered evolution, since each process needs to wait the end of the evolution of the previous portion of the image, we implemented the following schema of blocking operations:

```
if rank==size-1:
    Send the upper;
    Receive the lower;
if rank!=0
    Send the lower;
Every thread:
    Receive the upper;

UPDATE THE FIRST ROW;

if rank==0:
    Send the lower;

UPDATE THE CENTRAL ROWS;

if rank!=size-1
    Receive the lower;

UPDATE THE LAST ROW;

if rank!=size-1
    Send the upper;
```

With this schema using blocking message passing operations we were able to avoid deadlocks and to correctly send and receive all the required rows. Observe that such procedure actually serializes the program since each process will need to receive the *upper* buffer in order to start the evolution and, additionally, such buffer will be sent to the previous process at the end of the evolution.

4.7 The timing

In order to make a scalability study the walltime taken by the evolution process was measured with the routine `MPI_Wtime()`. Then, to have a more coherent result a mean of 5 measures was taken as follows:

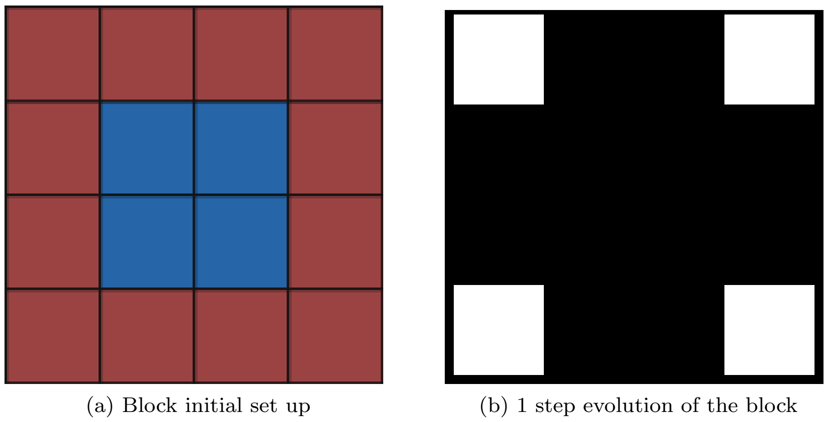
```
time=0;
for t in [1,5]:
    Inizialization of the starting set-up;
    MPI_Barrier(MPI_COMM_WORLD);
    begin=MPI_Wtime();
    EVOLUTION STEPS;
    MPI_Barrier(MPI_COMM_WORLD);
    end=MPI_Wtime();
    time=time+(end-begin);
Save on data.txt(time/5, number_of_threads,
number_of_MPI_processes);
```

As we can see the results required for a scalability study are saved on a `data.txt` file which will be used for the plots.

5 Results and discussions

5.1 The correctness of the program

Since the update rule we are using is not the standard one we could not use the standard patterns to check the correctness of the program. For example the pattern shown in Figure 5, which should be stable with the standard rule, produces a non-stable evolution in our case.


(a) Block initial set up

(b) 1 step evolution of the block pattern

Figure 5: Evolution of a stable pattern with our rules; the blue cells are those alive

For this reason, to check the correctness of the program we used the *playground*

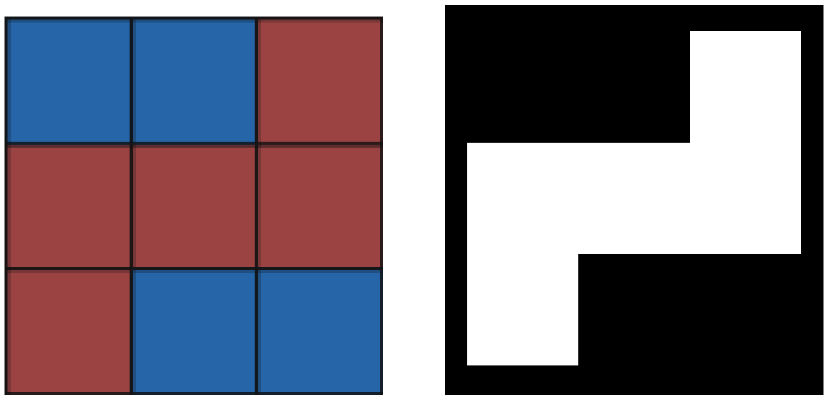

(a) Stable initial set up (b) 1000 step evolutions

Figure 6: An example of a stable pattern with the modified rules

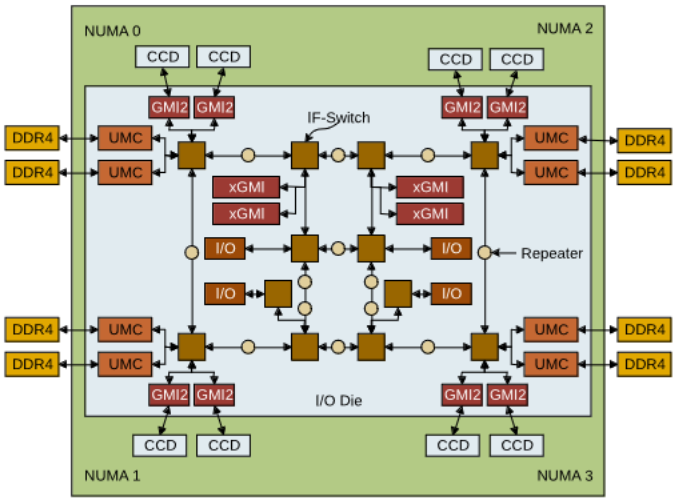
created for this purpose. We wrote some simple 3×3 initial set-ups and checked by hand their evolutions. For example, the set-up shown in Figure 6 is stable with our update rules and indeed our program produced a stable evolution. Since we needed to check the evolution on very low dimension set-ups we also implemented the scale of the image in the Game of Life program passed with the option `-c <cell_size>`.

5.2 Scalability studies

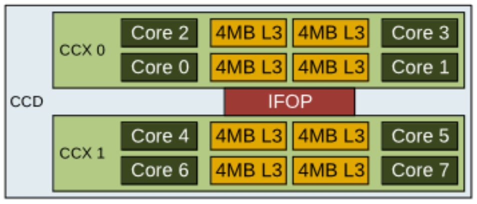
The scalabilitiy studies were made on the EPYC nodes of the ORFEO infrastructure. Each node is provided with two `AMD Rome` processors. Each processor has 64 cores with simultaneous-multithread disabled. The 64 cores are distributed in 4 NUMA regions as shown in Figure 7. Each NUMA region is formed by 2 Core Complex Dice (CCD), and each of them is composed by 2 Core Complex (CCX) by 4 core each.
The studies were performed by executing a batch script which allocates the required resources, creates the initial set-up, launches the program with an increasing number of OpenMP threads or MPI processes with the correct binding policy. The OpenMP and the weak scalability for MPI studies were done with 100 steps of the static evolution since the ordered one is intrinsically serial. For the strong MPI scalability the number of step evolutions were fixed to 10.

5.2.1 OpenMP scalability

The OpenMP scalability was performed by fixing two MPI processes into 2 sockets of a node and increasing for each process the number of OpenMP threads up to the number of cores in the socket. In order to produce the data, we used a batch script in which we require an entire node, create an initial set-



(a)



(b)

Figure 7: Structure of the AMD Rome CPU (a) and each Core Complex Dice (b)

up, set the `OMP_NUM_THREADS` environmental variable making it change from 1 to 64 and run the program with the `mpirun` command with the options `-np 2 map--by socket`. In this way two processes are generated into 2 different sockets and each thread will be spawned in the socket of the process it belongs to. The results were finally plotted using `gnuplot` obtaining the results in Figure 8.

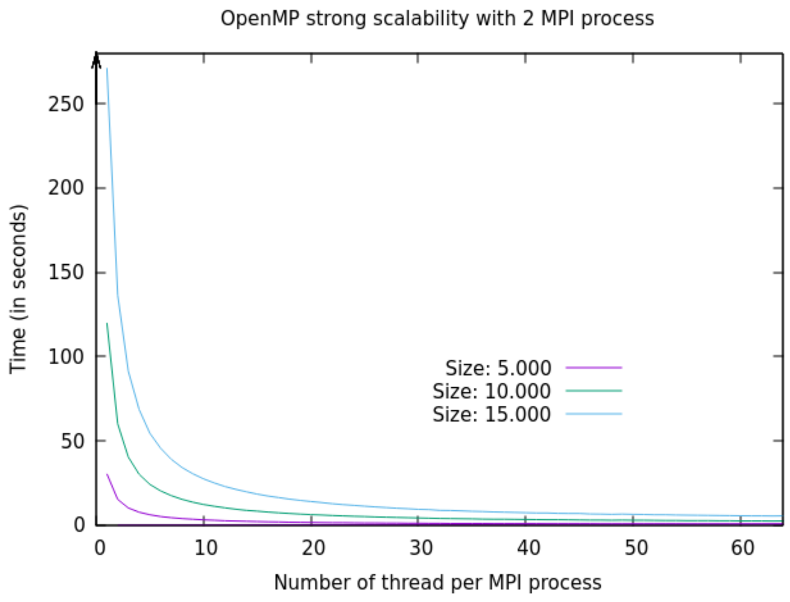


Figure 8: Time of 100 steps of the static evolution for an increasing number of OpenMP threads

The speedup was then analysed by measuring the time taken by the serial code (one MPI process and one OpenMP thread) and dividing it by the times obtained achieving the results in Figure 9.
We also decided to investigate the behaviour of the program for low dimensions and as expected the performance was poorer for too low dimensions. This behaviour starts to be mitigated by reaching a dimension of 500 as shown in Figure 10.

5.2.2 MPI strong scalability

The MPI strong scalability study was done by setting the number of OpenMP to 1 per process and increasing the number of MPI processes. In order to produce the data, we used a batch script in which we require two nodes, create the initial set-up, set the `OMP_NUM_THREADS` environmental variable to 1 and run the program with the `mpirun` command with the options `-np $i map--by core` with `$i` increasing from 1 to 256. In this way the processes are spawned one after

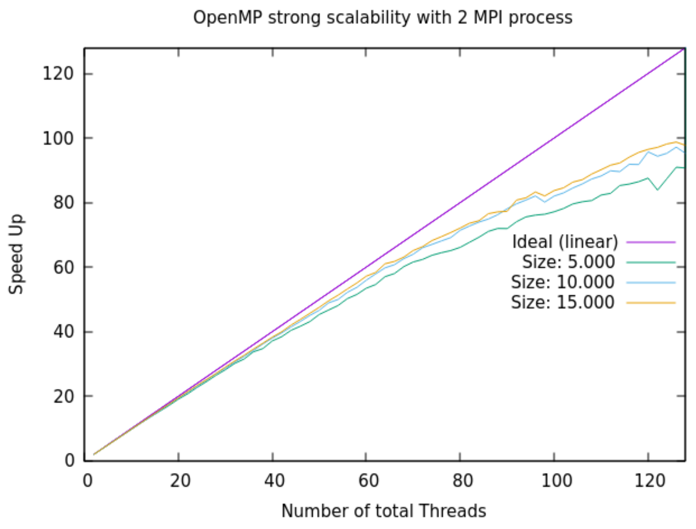


Figure 9: Speedup of 100 steps of the static evolution for an increasing number of OpenMP threads

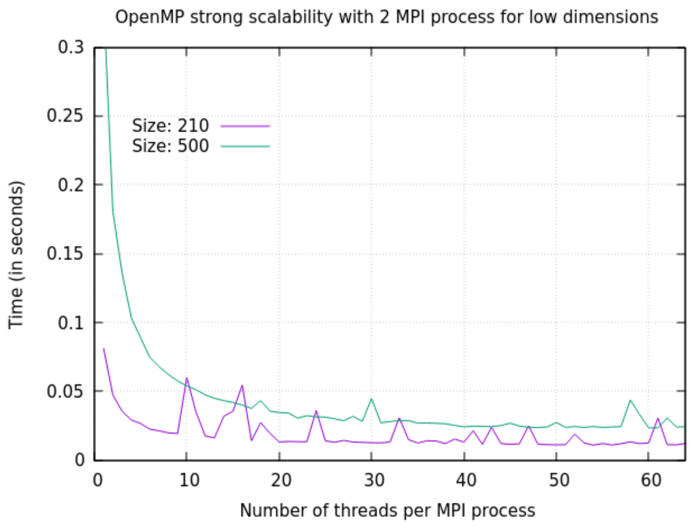


Figure 10: Time of 100 steps of the static evolution for an increasing number of OpenMP threads

the other following the numeration of the cores in the socket until saturating all the socket and after that all the node.
Finally the results were plotted using `gnuplot` obtaining the plot shown in

Figure 11. We also evaluated the speed-up obtaining the result in Figure 12. As we can see the performance in terms of MPI scalability are poorer than those obtained increasing the number of threads.

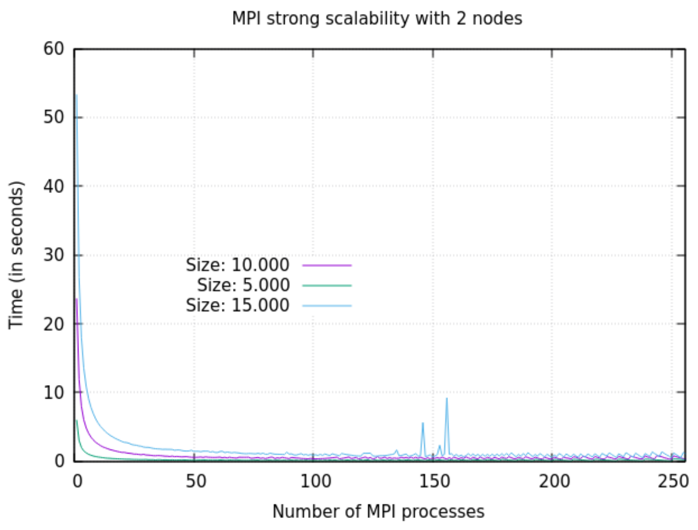


Figure 11: Time of 10 steps of the static evolution for an increasing number of MPI processes

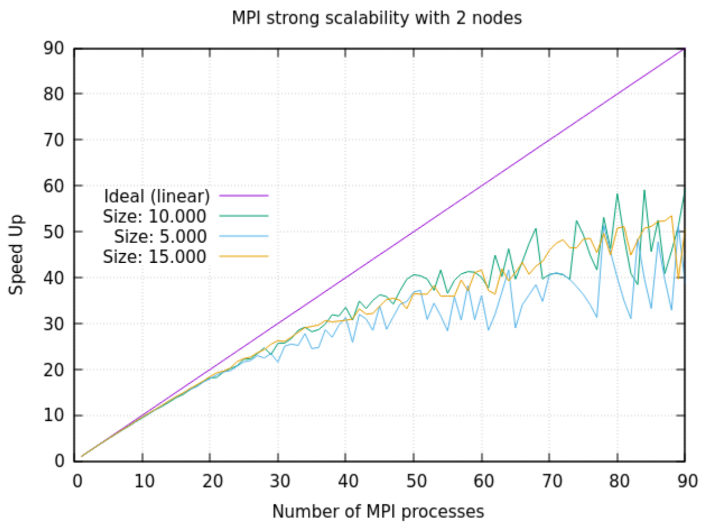


Figure 12: Speedup of 10 steps of the static evolution for an increasing number of MPI processes (cut at 90 processes)

5.2.3 MPI weak scalability

The MPI weak scalability was done maintaining the workload for each MPI process constant while increasing the number of MPI processes. Since each process works on an image of dimension $\frac{k}{n_{MPI}} \times k$, where n_{MPI} is the number of MPI processes, fixing this dimension as $\frac{k}{n_{MPI}} \times k = 10000$ we obtain that the size must be given by $k = \sqrt{n_{MPI}} \cdot 5000$. With this equation we previously evaluated the sizes for each value of n_{MPI} and then used a batch script to allocate the resources, initialize the image with the correct dimensions (the k previously evalueted with the given formula) and run the program with the `mpirun` command and the options `-np $n_MPI --map-by core` making the `$n_MPI` value increases by 1 after each iteration. Finally, the results were plotted using `gnuplot` obtaining the graph in Figure 13.

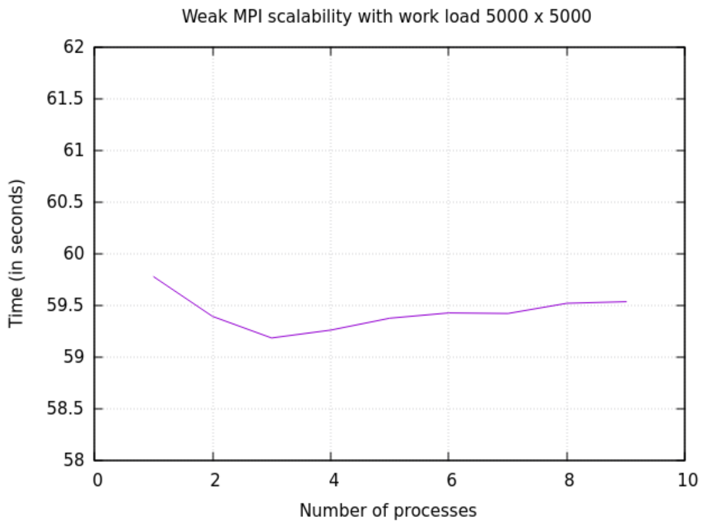


Figure 13: Weak MPI scalability starting with size 5000 and 1 process and ending with size 15000 and 9 processes

6 Conclusion

As shown by the outcomes, we achieved a poorer result for the MPI scalability than that obtained with the OpenMP scalability. The irregular periodic behaviour could be due to the memory organization of the MPI processes. In fact each process will load the total initial image and then take its portion of the image to work with. This procedure was done because the functioning of the program was meant to use an MPI process per socket from each node and then populating the sockets with OpenMP threads. In fact with this procedure the program performance is satisfactory as shown in the plot in Figure 14 realized with `mpirun -np 8 --map-by ppr:1:socket:pe=64` on 4 nodes and increasing the `OMP_NUM_THREADS` environmental variable from 1 to 64.

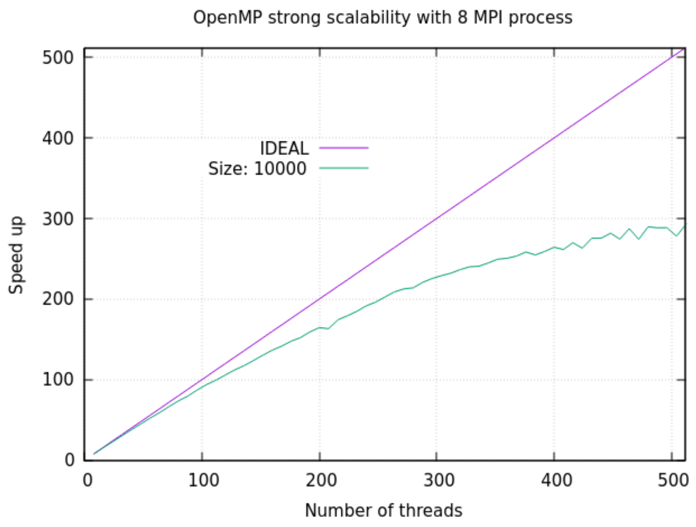


Figure 14: Speedup of 100 steps of the static evolution for an increasing number of OpenMP threads on 8 MPI processes (4 nodes 1 per socket)

Part II

Assignment 2

7 Introduction

BLAS stands for "Basic Linear Algebra Subprograms" and defines a set of fundamental operations on vectors and matrices useful to perform linear algebra computations. There are three level of BLAS operations: level 1 (vector operations), level 2 (matrix-vector operations) and level 3 (matrix-matrix operations). Different implementations of BLAS are available, in this analysis we took into account:

- Intel MKL: Intel Math Kernel Library, a library of optimized math routines and including, among others, BLAS, LAPACK and ScaLAPACK
- OpenBLAS: an optimized BLAS library based on GotoBLAS2
- BLIS: BLAS-like Library Instantiation Software, where, virtually, all computation within level-2 (matrix-vector) and level-3 (matrix-matrix) BLAS

operations can be expressed and optimized in terms of very simple kernels

The aim of this work is to compare the performances of the three implementations on a THIN node hosted by the ORFEO cluster using up to 12 cores. The common task will be to run a provided code that focuses on the level 3 BLAS function called `gemm`. This program receives as input the values M, K and N representing the dimensions of two matrices: A is a $M \times K$ matrix and B is $K \times N$. The output is the matrix $C = AB$ of dimension $M \times N$.

We were given two files: `gemm.c` and a `Makefile`. We slightly modified the first one to get on the screen the set of the useful results already separated with commas, obtaining as last line something in this format:
`value_of_M,value_of_N,value_of_K,elapsed_time_in_seconds,GFLOPS`
The file was renamed as `gemm_MOD.c`.
We modified the `Makefile` after installing the BLIS library in order to get, together with `gemm_mkl.x` and `gemm_oblas.x`, also the executable `gemm_blis.x`.
We prepared two versions of the `Makefile`: `Makefile_single` (with flag `-DUSE_FLOAT` for single precision) and `Makefile_double` (with flag `-DUSE_DOUBLE` for double precision).

8 Implementation

We wrote four bash scripts to automate the process (two programs to study the scalability over the matrix size and the other two to study the scalability over the number of cores):

- `bash_THIN_single_point1.sh`: it creates a folder beginning with `results_THIN_single` (the complete name is also given by the day and the time of creation of the folder, so if one wants to run the bash script multiple times the previous results will not be overwritten). The executables `gemm_mkl.x`, `gemm_oblas.x` and `gemm_blis.x` previously mentioned are executed with the command `srun -n1 --cpus-per-task=12` and a value of M, N and K such that $M = N = K$. The first value is 2000, the last one is 20000 and the step is 1000 (so 19 iterations). This entire increase in the matrix size is repeated for 7 tests. At the end of the process we end up with 21 files: 7 tests for each library, and each test contains 19 rows. A row stores the values of the last line printed on the screen by the executable (i.e., it stores the results we need in the format: `value_of_M,value_of_N,value_of_K,elapsed_time_in_seconds, GFLOPs`)
- `bash_THIN_double_point1.sh`: it is equal to the previous bash script, the only difference is in the name given to the folder containing the results: in this case, it will begin with `results_THIN_double`

- `bash_THIN_single_point2.sh`: it creates a folder beginning with `results_THIN_pt2_single` (the complete name is also given by the day and the time of creation of the folder, so if one wants to run the bash script multiple times the previous results will not be overwritten). In this case, the size of $M = N = K$ is set to 1200. The increasing quantity is the number of available cores to perform the computations, ranging from 1 to 12 with step 1 (so 12 iterations). The executables `gemm_mkl.x`, `gemm_oblas.x` and `gemm_blis.x` are executed with the command `srun -n1 --cpus-per-task=$i` where `$i` is the integer equal to the number of available cores. This entire increase in the number of available cores is repeated for 7 tests. At the end of the process one ends up with 21 files: 7 tests for each library, and each test contains 12 rows. A row stores the values of the last line printed on the screen by the executable, but prepending the number of cores used (i.e., it stores the results we need in the format: `num_cores,value_of_M,value_of_N,value_of_K, elapsed_time_in_seconds,GFLOPs`)

- `bash_THIN_double_point2.sh`: it is equal to the previous bash script, the only difference is in the name given to the folder containing the results: in this case, it will begin with `results_THIN_pt2_double`

To run these scripts and therefore the programs one needs to have computational resources and some options set up. So before executing one of the scripts we had to run a few commands:

- `salloc --exclusive -n 12 -N1 -p THIN --time=2:0:0`: to reserve the computational resources: an entire node `THIN` will be reserved and, here, 12 cores will be used

- `module load architecture/Intel`, `module load mkl`, `module load openBLAS/0.3.23-omp`: to load the required modules

- `export OMP_PLACES=cores`: to specify that threads should be assigned one per core (and not, for example, one per socket)

- `export OMP_PROC_BIND={spread, close}`: we will investigate what happens if we set `export OMP_PROC_BIND=spread` (the software threads are placed onto places as evenly as possible) and what happens if we set `export OMP_PROC_BIND=close` (the software threads are placed onto places as close as possible to each other)

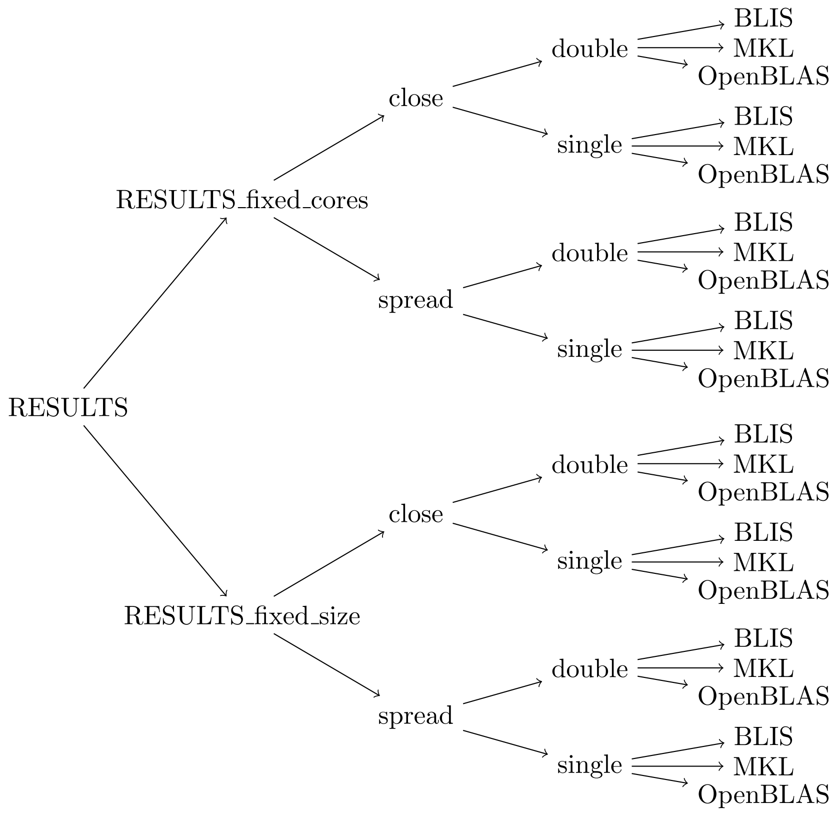
- `export BLIS_NUM_THREADS=12`: to set the maximum number of threads that the BLIS library will be allowed to use

- `export LD_LIBRARY_PATH={path_for_blis}:$LD_LIBRARY_PATH`: replace the `{path_for_blis}` with the path to the `lib` folder of BLIS, so the system will search for libraries also in that directory

Now with `srun -n1 make cpu` we can run the `Makefile`, one between the two available, that we want to execute (after renaming it `Makefile`). With `chmod +x {bash_script}` (replacing `{bash_script}` with the name of the target script) we make the bash script executable. Running now this last one, we will get the folder with the results saved in `.csv` files.

9 Results and discussion

At the end of data gathering we organized the results in a structure of folders like this:



where `RESULTS_fixed_cores` stores the results obtained using 12 cores and variable input and `RESULTS_fixed_size` the results with input 1200 and variable number of cores.
The `.csv` files are contained in the folders that constitute the leaves of the tree: each of these directories has 7 files, one per test. We wrote a program in Python, `function_mean_matrix.py`, that contains the function `mean_matrix(csv_directory)`. With this last one, we can get the mean values from the 7 tests belonging to the same folder: the input is the parameter `csv_directory`, that is the path of the folder containing the files to average,

while the output is a NumPy matrix. These NumPy matrices (one per leaf) will be used to plot the summary graphs.

9.1 Scalability over the input size

We will compare the results with the theoretical peak performance, whose formula is:

$$FLOPs = cores \times frequency \times FLOP/cycle$$

and the result is given in FLOP per second. During lectures we saw that the peak performance for a `THIN` node taking into account all the 24 cores is 1.997 TFLOPs. Considering the formula and 12 cores, the amount we used in the programs, we get that the theoretical peak performance in our case is 998.5 GFLOPs if we deal with double precision. Instead, if we consider the single precision, that number should be doubled and therefore the theoretical peak performance in this situation is 1997 GFLOPs.

To study the scalability over the matrix size we used the bash scripts `bash_THIN_single_point1.sh` and `bash_THIN_double_point1.sh`, paying attention to have in the same folder respectively the executables coming from the `Makefile_single` and from the `Makefile_double`. The bash scripts were run twice, once having set `export OMP_PROC_BIND=spread` and the other time with `export OMP_PROC_BIND=close`.

In Figure 15 we can observe that real performances are not so close to the theoretical one. BLIS clearly stands out from the other two libraries and does not show a significant difference between the options `spread` and `close` of the `OMP_BIND_PROC` environmental variable. The two lines of `spread` and `close` of MKL seem almost to overlap. On the other hand, the difference between the two lines of OpenBLAS is recognizable and would lead to pick the OpenBLAS library with `OMP_BIND_PROC=close` as the best option. With MKL and OpenBLAS we reach more or less the 80% of the value of the theoretical peak performance.

In Figure 16 the curves more or less retain their arrangement and their profile. However, one has to remark that the scale has changed and, compared to the previous image, now the values of GFLOPs have decreased. The two lines of MKL seem to overlap less, slightly indicating that the `close` option is a better choice; `close` as a better choice is confirmed for OpenBLAS too. Also in this case we reach more or less the 80% of the theoretical peak performance with MKL and OpenBLAS.

In Figure 17 and in Figure 18 we reported the elapsed time as a function of the input size. One can notice that, reasonably, the curves have the trend of a

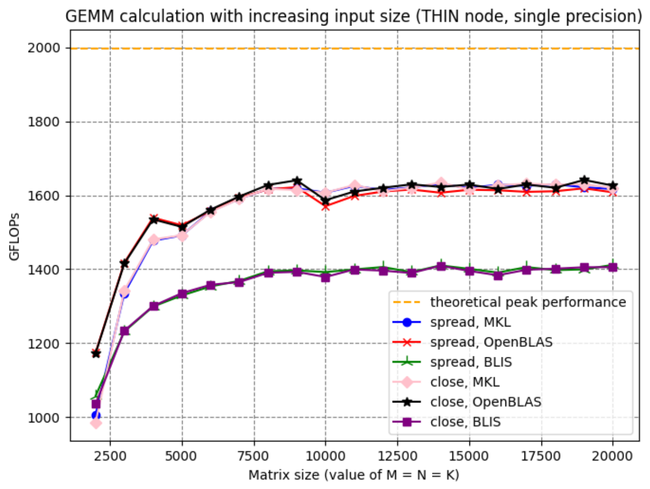


Figure 15: 12 cores on a THIN node, single precision: GFLOPs as a function of the input size

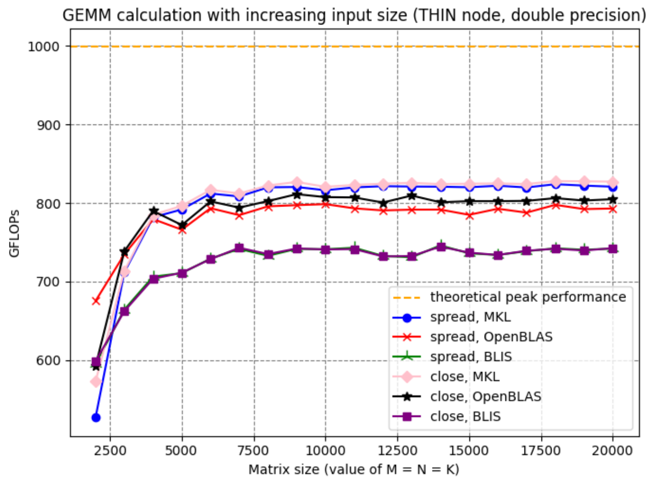


Figure 16: 12 cores on a THIN node, double precision: GFLOPs as a function of the input size

third-degree polynomial in the form of:

$$elapsed\_time\_in\_seconds = constant \cdot (input\_size)^3$$

with $input\_size$ equal to the value of $M = N = K$.

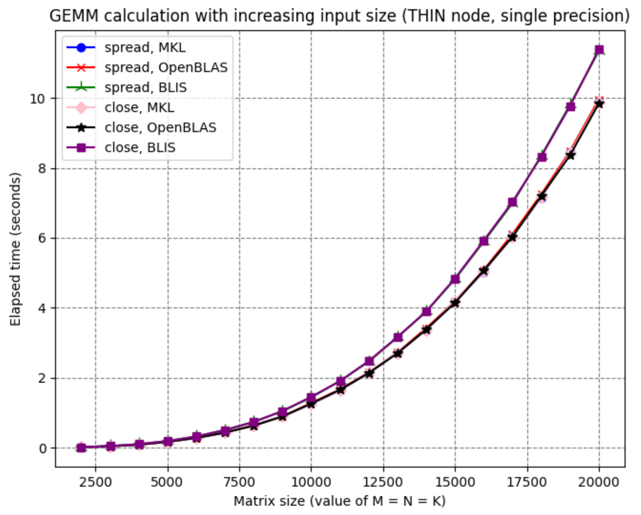


Figure 17: 12 cores on a THIN node, single precision: elapsed time as a function of the input size

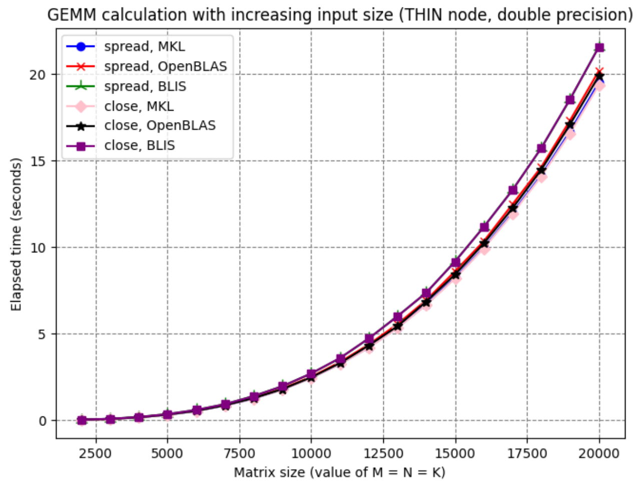


Figure 18: 12 cores on a THIN node, double precision: elapsed time as a function of the input size

9.2 Scalability over the number of cores

In this section we will investigate the scalability over the number of cores: we decided to fix as input size the value 1200 (so $M = N = K = 1200$) and we collected the results increasing the number of available cores from 1 to 12. We gathered the data through the use of the bash scripts `bash_THIN_single_point2.sh` and `bash_THIN_double_point2.sh`, paying attention to have in the same folder respectively the executables coming from the `Makefile_single` and from the `Makefile_double`. Also in this case the bash scripts were run twice, once having set `export OMP_PROC_BIND=spread` and the other time with `export OMP_PROC_BIND=close`.
We report the graphs of GFLOPs and elapsed time in this case too, but we also plotted the graph regarding the speedup $S(P)$ (where P is the amount of parallel units at our disposal). The speedup describes somehow the gain in performing the computations in parallel rather than in serial. It is computed as:

$$S(P) = \frac{T(1)}{T(P)}$$

where $T(1)$ is the time needed for the serial computation and $T(P)$ for the one in parallel with P computational units. The ideal trend would be a graph with a 45-degree inclined line (so that $S(P) = P$). This line is represented as an orange dashed line in the plots. In our analysis, we took $T(1)$ as the amount of time needed to run the same code but using just a core.
One may also consider the efficiency $\epsilon(P)$ to have an idea on how efficiently, in average, the computational units are exploited. It is computed as:

$$\epsilon(P) = \frac{S(P)}{P}$$

and, basically, it quantitatively expresses the proximity of the real results with the ideal ones.

In Figure 19 and in Figure 20 we reported the amout of GFLOPs as a function of the number of available cores, in the first image with single precision and in the second one with double precision. In both cases the BLIS lines behave quite differently compared to those of MKL and OpenBLAS, which, instead, seem almost to overlap in particular in Figure 19.

The graphs representing elapsed time as a function of the amount of available cores, in Figure 21 and in Figure 22, appear very curious as well: also here the MKL and the OpenBLAS libraries follow a similar trend, while the two lines of BLIS show a completely different tendency. The peculiarities that stand out in the BLIS lines are mainly three:

- the high value of the elapsed time if we use only one core, and the rapid decrease with two cores (the elapsed time dropped by more than the 50% with the single precision)

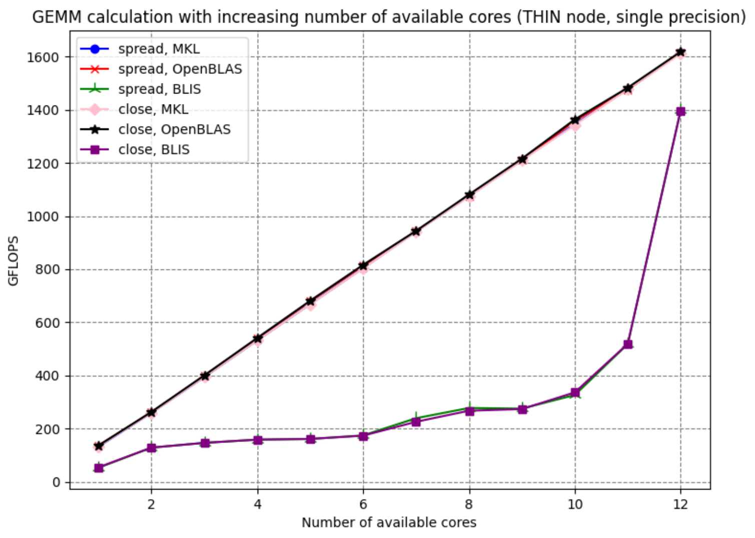


Figure 19: Input size of $M = N = K = 1200$ on a THIN node, single precision: GFLOPs as a function of the number of available cores

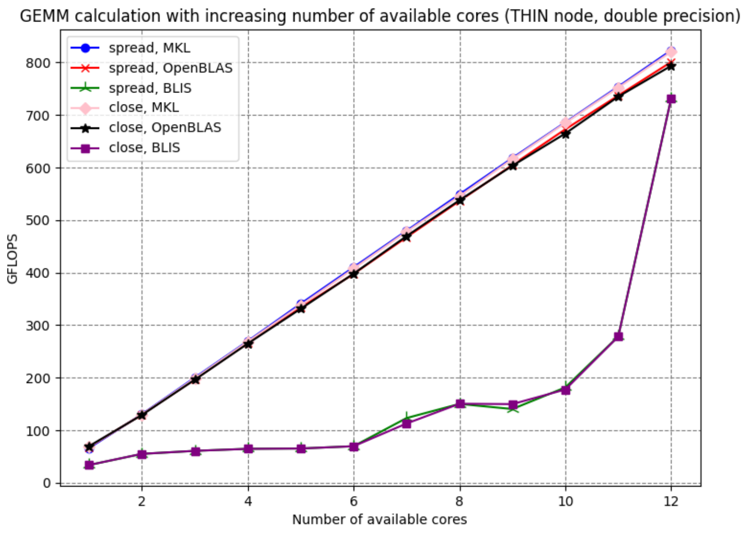


Figure 20: Input size of $M = N = K = 1200$ on a THIN node, double precision: GFLOPs as a function of the number of available cores

- the undulatory trend in the middle region
- the elapsed time if we use 12 cores, that appears to be the same of MKL and OpenBLAS

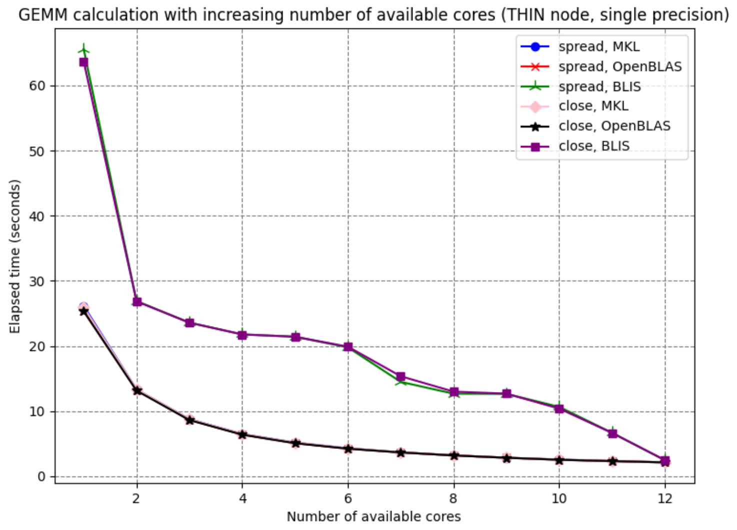


Figure 21: Input size of $M = N = K = 1200$ on a THIN node, single precision: elapsed time as a function of the number of available cores

Finally, we plotted the graphs of the speedup, computed as suggested by the formula previously indicated: the results can be seen in Figure 23 and in Figure 24. As already mentioned, the orange dashed line represents the ideal trend (it would suggest that the total workload is always the same and appears to be equally split among the cores). MKL and OpenBLAS give close performances and, additionally, we can observe that some lines outperform the ideal trend. This may happen because the baseline we chose as the value for $T(1)$ probably was not the fastest and the more optimised choice, so its value is higher than what it should be and therefore the speedup appears to be greater.
This problem is particularly relevant in the case of BLIS: as one might expect after having seen the other graphs, the trend is undulatory and towards the end rises steeply, reaching a value much higher than the ideal case; this applies to both the Figure 23 and the Figure 24. Also in this case the trend is strongly influenced by the value of $T(1)$ we chose as the baseline (as a matter of fact, remember that in Figure 21 and in Figure 22 the elapsed time obtained using one core was oddly high). We presume that this strange result, especially regarding the speedup having 12 cores, may be caused by a suboptimal choice of $T(1)$

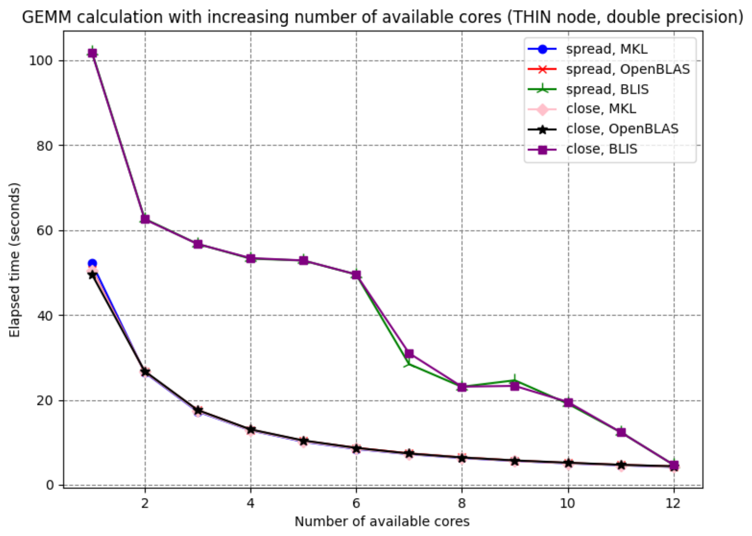


Figure 22: Input size of $M = N = K = 1200$ on a THIN node, double precision: elapsed time as a function of the number of available cores

and by some kind of optimisation that the BLIS library could have made, for example relying on the environmental variables (as a matter of fact, the only thing that we modified to get the results was just the amount of cores made available for the computations).

10 Hints on a possible study on an EPYC node

We focused our attention on the `THIN` nodes of the ORFEO cluster, but we also have access to a set of `EPYC` nodes. Some differences that we should have considered if we had carried on the analysis also here would be:

- reserving 64 cores with `salloc` (as asked by the text of the assignment) instead of 12
- use `module load architecture/AMD` instead of `module load architecture/Intel`
- compile the BLIS library on an `EPYC` node and not on a `THIN` node
- considering the architecture of the `EPYC` node, organised in NUMA regions, one could think about using the `numactl` command to better manage the thread placement

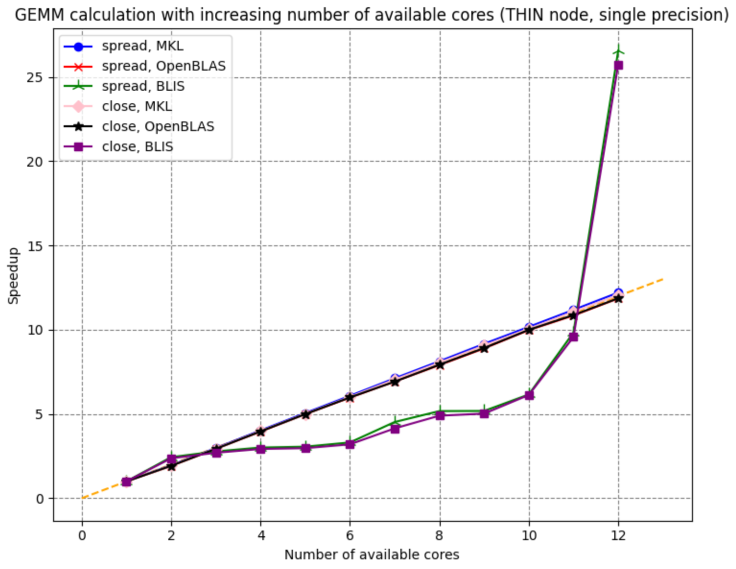


Figure 23: Input size of $M = N = K = 1200$ on a THIN node, single precision: speedup as a function of the number of available cores

11 Conclusion

In these sections we investigated the scalability over the input size and over the number of cores, taking into account MKL, OpenBLAS and BLIS and using as common task to execute the `gemm` function on a `THIN` node.
Regarding the variation on the input, we observed that the BLIS library has lower values in the graphs representing GFLOPs, while MKL and OpenBLAS are quite similar and reach higher performances. The elapsed time is consistent to what we might have expected. The differences between `OMP_PROC_BIND=spread` and `OMP_PROC_BIND=close` are not so evident.
Changing the number of cores, instead, gave curious results. MKL and OpenBLAS also in this case behave in a similar way, and seem not to be so bad (still, we have to remember about the specific choice we did for $T(1)$). Instead, BLIS showed a strange trend, letting us presume that this outcome depended on some kind of optimization and on some environmental variables. In any case the differences between `OMP_PROC_BIND=spread` and `OMP_PROC_BIND=close` seem not to be so pronounced.

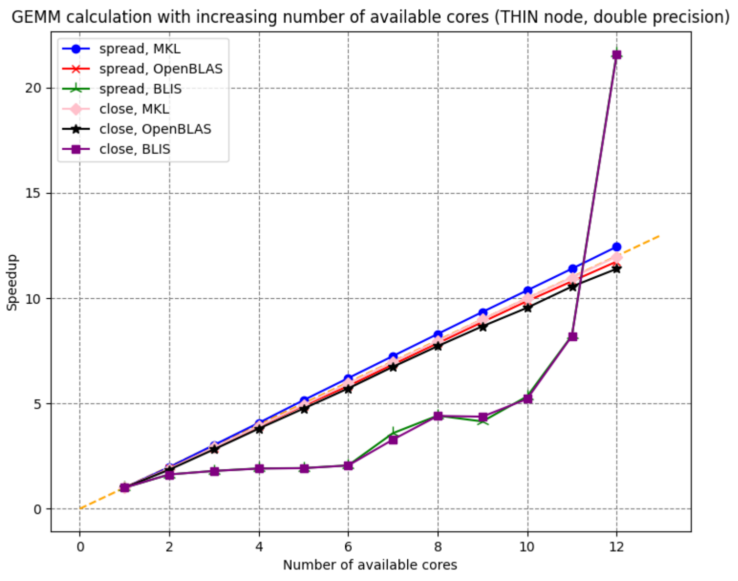


Figure 24: Input size of $M = N = K = 1200$ on a THIN node, double precision: speedup as a function of the number of available cores