FOUNDATIONS OF HIGH PERFORMANCE COMPUTING FINAL ASSIGNMENT

Msc in Data Science and Scientific Computing
University of Trieste

Author
Tommaso Tarchi
Trieste
19/04/2023

Contents

1 Exercise 1

1.1 Introduction

In this first part of the assignment we were asked to implement a hybrid MPI+openMP parallel version of Conway's game of life (abbreviated to GOL in this document) in which each MPI process would spawn a number of openMP threads to further parallelize its work. We were also asked to test its scalability from three points of view:

- **OpenMP scalability**: with one MPI process per socket, increasing number of openMP threads (each thread on a different core) from one per process up to saturation (so 64 for EPYC nodes and 12 for THIN nodes)
- **Strong MPI scalability**: increasing number of MPI processes from 1 up to one per socket (we saturated each socket with openMP threads) with constant total workload
- **Weak MPI scalability**: increasing number of MPI processes from 1 up to one per socket (again sockets are saturated with threads) with constant workload per process

GOL takes place on a rectangular playground (*grid*) filled with square cells that can be in two alternative states: *dead* or *alive*. At each generation, each cell of the grid is updated according to the state of the neighbouring cells: if among the 8 neighbours there are 2 or 3 alive cells, the cell becomes (or stays) alive, otherwise it becomes (or stays) dead. Edges of the grid are considered to be bordering to each other, i.e. the "world" has the shape of a torus. For a more detailed description of GOL's rules see the related Wikipedia page (Wikipedia, 2001).

In our case evolution had to be of two kinds:

- **Ordered**: cells are updated in "real time", meaning that the program iterates over all cells computing their state and updating it immediately. Notice that this kind of evolution is "intrinsically" serial, therefore parallelization here is a mere exercise of style (a part from the case of large playgrounds that do not fit in a single node's memory, in which case MPI allows to use more than only one node's memory - more on that in section 1.3.2)
- **Static**: the program iterates over all cells computing their state and storing it somewhere; only after all new states are computed the grid is updated

The program had to be implement in such a way that it could be run in two modes:

- *Initialisation*: initialise a random grid of any rectangular size such that $x\_size \geq 100, y\_size \geq 100$ (where x_size and y_size are the number of columns and rows respectively) and store it in a PGM file with a name assigned by the user (otherwise the default `game_of_life.pgm` has to be used)
- *Evolution*: run GOL starting from any initial playground (otherwise `game_of_life.pgm` is the default) and for any number of generations chosen by the user; along the run a dump of the system's state has to be taken with a given frequency and stored in a file called `snapshot_nnnnn.pgm`, with `nnnnn` generation number (padded with zeros)

For a more detailed description of the aim of the exercise and the way the program had to take parameters in input see the original pdf of the exercise (Tornatore Luca, 2022). **Notice** that, regarding input arguments, the only difference between our implementation and the one described in that document is that in our code we can pass any number of rows and columns using, respectively, the two arguments `m` and `k`; and also that evolution can be chosen to be ordered, static or static in place setting the `-e` option respectively to 0, 1 or 2.

1.2 Methodology

1.2.1 Workload division

We chose to divide the workload by rows, meaning that we assign to all processes an equal number of rows of the grid, assigning the remainder rows one to each process in sequence starting from the 0-th process; for instance, if we have 4 processes and a playground with `x_size`=10 and `y_size`=3, the four processes will have (in order) grids with sizes 3x3, 3x3, 2x3, 2x3. We are aware of the fact that in this way we are likely assigning to some of the processes a larger number of cells to work on. However this approach makes the code simpler and, supposing large grids are being used, the penalization in performance is probably not much relevant.

Regarding the division of work within each process, we decided to equally distribute the number of cells among threads (assigning the remainder cells one to each thread in sequence starting from the master).

1.2.2 Initialisation

During initialisation we randomly set each cell's state either to dead or alive, with equal probability. We do that in parallel, letting each MPI process operate at the same time and each one of them letting the spawned threads operate in parallel.

A fundamental part is to make sure that all threads use a different seed to generate random numbers. This is important to avoid recurring patterns in the grid's state. For more on this subject see the related appendix A.

1.2.3 Evolution

We implemented three kinds of evolution; in any case, at each generation all processes have to communicate their upper and lower edges' state to the two neighbour processes (in case of a single process, this will communicate with itself), so they can use them to evolve their bordering cells:

- **Ordered**: we use just one grid and we update all cells sequentially: starting from the 0-th process, each process's threads (in sequence as well) update their cells
- **Static**: we use two grids, one to store the old system state, used to determine the new system state, and one to store the new system state itself
- **Static in place**: we are able to use one single grid to store both old and new system states. The purpose of this kind of evolution is solely to save half of the memory used in the previous one (it does not improve the performance)

1.3 Implementation of GOL

The parallel GOL code, called `parallel_gol.c`, relies on functions defined in `gol_lib.c` for all kinds of evolution and PGM header reading. Actually, we also presented a version of the code that has all evolution functions "embedded", called `parallel_gol_unique.c`. We did that because we are aware of the fact that function calls comport a certain overhead. Since this functions are called potentially a large number of times (as many as generations are), we preferred to give the possibility of avoiding this overhead.

In addition to that, we gave a modified version of parallel GOL called `parallel_gol_mod.c`, in which we do a "warm up" of the grid used to store the initial system state. For the reason for this see 1.5.5.

However, in the following document we will always speak about the `parallel_gol.c` version.

1.3.1 Hybrid parallelization

To mix openMP and MPI we used the so called **funneled mode**, in which MPI calls can be done from within the openMP parallel region, but only by the master thread. This approach allowed us to perform communications among processes and parallel I/O without getting out of the parallel region, and therefore avoiding superfluous synchronizations among threads and the parallel region "management" overhead.

1.3.2 Grid storage

Ideally, we would use only one bit to represent the state of each cell (actually two in *static in place* evolution, as we will see), but in C the smallest standard type available is `char`. Therefore we chose to represent the grid as a one-dimensional array of `char`, which we renamed `BOOL` in the code, and to use the last bit (the last two in static in place evolution) of the variable to store the cell's state: if the bit is 1 the cell is alive, otherwise it is dead. We chose to use a one-dimensional array to make operations over it simpler an faster.

Actually, we never use a single grid to store all the system. What we do in evolution is to (dynamically) allocate a smaller grid (called `my_grid`) for each process according to the number of rows assigned to it, with two more rows for the neighbours' bordering cells' states. The initial cells' state is then read in parallel by all processes and stored in their own grids. After the computation (and, if asked, when dumping a certain system state) all processes write in parallel their grids' state. In a similar way, in initialisation each process allocates a grid only for its cells, initialises it and then all processes write the generated playground in parallel. In this way, at no point of the program we need to allocate the complete system on a single process, saving memory and allowing the user to evolve systems occupying more memory than the one available on one node.

For the static (non in place) evolution we used an auxiliary grid to store the updated grid state, called `my_grid_aux`.

Finally, it is worth noting that we made all cells of a thread contiguous, so that the number of cache misses is reduced. However, we still have a number of cache misses in evolution, since, while evolving a certain cell, the thread has to access all cell's neighbours. We can't exclude a more complicated distribution of the grid among threads to be more

efficient (for instance dividing the grid in rectangular "regions"), but we were not able to come up with it, especially considering that both grid size and threads number are variable.

1.3.3 Parallel I/O

A part from PGM files header writing and reading (see the related appendix B), all input/output operations to or from PGM files are done in parallel by processes. For that we used the specific MPI I/O functions.

MPI I/O operations to PGM files occur four times in the code: when writing the randomly initialised playground in initialisation mode, when reading the initial playground in running mode, when writing the system dump and the final system state (in running mode as well).

Let's look at an example of PGM writing, in particular that of randomly initialised playground:

```
MPI_File f_handle;   // pointer to file
int check = 0;   // error checker

/* formatting the PGM file */
const int color_maxval = 1;
/* the third number added to header_size must be equal to
 * the total length in bytes of the header, spaces and
 * new-line characters included and length of m and k
 * excluded */
const int header_size = m_length+k_length+7;

if (my_id == 0) {

    /* setting the header */
    char header[header_size];
    sprintf(header, "P5 %d %d\n%d\n", k, m, color_maxval);

    /* writing the header */
    int access_mode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
    check += MPI_File_open(MPI_COMM_SELF, fname, access_mode,  /
                           MPI_INFO_NULL, &f_handle);
    check += MPI_File_write_at(f_handle, 0, header, header_size,  /
                               MPI_CHAR, &status);
    check += MPI_File_close(&f_handle);
}

/* needed to make sure that all processes are actually
 * wrtiting on an already formatted PGM file */
MPI_Barrier(MPI_COMM_WORLD);

/* opening file in parallel */
```

```
int access_mode = MPI_MODE_WRONLY;
check += MPI_File_open(MPI_COMM_WORLD, fname, access_mode,  /
                       MPI_INFO_NULL, &f_handle);

/* computing offsets */
int offset = header_size;
if (my_id < m_rmd) {
    offset += my_id*my_n_cells;
} else {
    offset += m_rmd*k + my_id*my_n_cells;
}

/* writing in parallel */
check += MPI_File_write_at_all(f_handle, offset, my_grid,  /
                               my_m*k, MPI_CHAR, &status);

check += MPI_Barrier(MPI_COMM_WORLD);

check += MPI_File_close(&f_handle);
```

The part executed inside the first if statement is the PGM file formatting. This is done the 0-th process (`my_id` stores the process's id) only and it is a very quick operation (it only consists in writing a short string); for more details on formatting see the already cited appendix B. After the barrier (needed to make sure that all processes are writing on a formatted PGM), all processes open the file, whose name is stored in `fname`, in parallel in write only mode; then they compute the offset from which they will have to start writing (`my_m` is the number of rows assigned to the process), and finally write in parallel the initialised playground.

We won't show the complete PGM reading here, since it is in general similar to writing.

1.3.4 MPI communications

In running mode, at each generation each process has to communicate the bordering cells' states to its two neighbour processes, and has to receive the same information from them. This step has to be performed in a different way depending on the kind of evolution chosen by the user.

For all communications we used the functions `MPI_Send` and `MPI_Recv`, and set tags, destinations and sources in the following way (notice that the previous process of the 0-th is the (n-1)-th and that the successor of the (n-1)-th is the 0-th):

```
const int prev = (my_id != 0) * (my_id-1) + (my_id == 0) * (n_procs-1);
const int succ = (my_id != n_procs-1) * (my_id+1);
const int tag_send = my_id*10;
const int tag_recv_p = prev*10;
const int tag_recv_s = succ*10;
```

As we already said, in ordered evolution processes and threads work serially. Communications are performed serially themselves: for each generation, at the beginning of each

process's "turn" the two neighbour processes send their bordering rows' state to it. The process stores them in its two supplementary rows and then performs evolution. What follows is the implementation of this communication mechanism (`check` is just an error checker for MPI communications - we removed the chunks of code to check these errors, being not essential to the present dissertation):

```
if (n_procs > 1) {

    if (proc == 0) {

        if (my_id == n_procs-1)
            check += MPI_Send(my_grid+my_n_cells, x_size, MPI_CHAR,  /
                              succ, tag_send, MPI_COMM_WORLD);

        if (my_id == 1)
            check += MPI_Send(my_grid+x_size, x_size, MPI_CHAR,  /
                              prev, tag_send, MPI_COMM_WORLD);

        if (my_id == 0) {
             check += MPI_Recv(my_grid, x_size, MPI_CHAR, prev,  /
                              tag_recv_p, MPI_COMM_WORLD, &status);
            check += MPI_Recv(my_grid+x_size+my_n_cells, x_size, MPI_CHAR,  /
                              succ, tag_recv_s, MPI_COMM_WORLD, &status);
        }

    } else if (proc == n_procs-1) {

        if (my_id == n_procs-2)
            check += MPI_Send(my_grid+my_n_cells, x_size, MPI_CHAR,  /
                              succ, tag_send, MPI_COMM_WORLD);

        if (my_id == 0)
            check += MPI_Send(my_grid+x_size, x_size, MPI_CHAR, prev,  /
                              tag_send, MPI_COMM_WORLD);

        if (my_id == n_procs-1) {
            check += MPI_Recv(my_grid, x_size, MPI_CHAR, prev,  /
                              tag_recv_p, MPI_COMM_WORLD, &status);
            check += MPI_Recv(my_grid+x_size+my_n_cells, x_size, MPI_CHAR,  /
                              succ, tag_recv_s, MPI_COMM_WORLD, &status);
        }

    } else if (n_procs > 2) {

        if (my_id == proc-1) {
            check += MPI_Send(my_grid+my_n_cells, x_size, MPI_CHAR,  /
```

```
                                succ, tag_send, MPI_COMM_WORLD);

            } else if (my_id == proc+1) {
                check += MPI_Send(my_grid+x_size, x_size, MPI_CHAR, prev,  /
                                tag_send, MPI_COMM_WORLD);

            } else if (my_id == proc) {
                check += MPI_Recv(my_grid, x_size, MPI_CHAR, prev, tag_recv_p,  /
                                MPI_COMM_WORLD, &status);
                check += MPI_Recv(my_grid+x_size+my_n_cells, x_size, MPI_CHAR,  /
                                succ, tag_recv_s, MPI_COMM_WORLD, &status);
            }

        }

    /* case of single MPI process */
    } else {

        check += MPI_Send(my_grid+my_n_cells, x_size, MPI_CHAR, succ,  /
                          tag_send, MPI_COMM_WORLD);
        check += MPI_Recv(my_grid, x_size, MPI_CHAR, prev,  /
                          tag_recv_p, MPI_COMM_WORLD, &status);
    }
```

In case of a single process, the communication (which happens between the process and itself) of the first row to the bottom has to be performed in the middle of the evolution. That because the last row of the grid has to be updated considering the new state of the first one. That is done by adding the following piece of code inside the function for ordered evolution (not present in functions for static and static in place evolution):

```
    if (n_procs == 1) {

        if (my_thread_id == 0) {
            check += MPI_Send(my_grid+x_size, x_size, MPI_CHAR, prev,  /
                              tag_send, MPI_COMM_WORLD);
            check += MPI_Recv(my_grid+x_size+my_n_cells, x_size, MPI_CHAR,  /
                              succ, tag_recv_s, MPI_COMM_WORLD, &status);
        }
    }
```

Things are much simpler in case of static and static in place evolution. In them all processes can communicate in parallel their borders' state at the beginning of the generation: even processes first send and then receive, while odd processes first receive and then send. This (but the inverse would be equivalent) is, in our opinion, the best way to carry out the communications, since it minimizes the time the "slowest" process waits for data, especially if the number of processes is even. Let's see the code:

```
    if (my_id % 2 == 0) {
```

```
        check += MPI_Send(my_grid+x_size, x_size, MPI_CHAR, prev,  /
                          tag_send, MPI_COMM_WORLD);
        check += MPI_Recv(my_grid+x_size+my_n_cells, x_size, MPI_CHAR,  /
                          succ, tag_recv_s, MPI_COMM_WORLD, &status);
        check += MPI_Send(my_grid+my_n_cells, x_size, MPI_CHAR, succ,  /
                          tag_send, MPI_COMM_WORLD);
        check += MPI_Recv(my_grid, x_size, MPI_CHAR, prev, tag_recv_p,  /
                          MPI_COMM_WORLD, &status);

    } else {

        check += MPI_Recv(my_grid+x_size+my_n_cells, x_size, MPI_CHAR,  /
                          succ, tag_recv_s, MPI_COMM_WORLD, &status);
        check += MPI_Send(my_grid+x_size, x_size, MPI_CHAR, prev, tag_send,  /
                          MPI_COMM_WORLD);
        check += MPI_Recv(my_grid, x_size, MPI_CHAR, prev, tag_recv_p,  /
                          MPI_COMM_WORLD, &status);
        check += MPI_Send(my_grid+my_n_cells, x_size, MPI_CHAR, succ,  /
                          tag_send, MPI_COMM_WORLD);

    }
```

1.3.5 Evolution

As we said, for each process, each thread evolves "its own" cells. To do that, it iterates over cells, reading for each one of them its eight neighbour's states and computing the current cell's new state. Before exploring how the thread uses this information, let's take a look at how the iteration itself is carried out.

The key point to observe is that **edge cells** (meaning cells that are on the edge of the grid) are in a relative position w.r.t. their neighbours that is different from the one of **internal cells**. In fact, if `i` is the number of the row and `j` is the number of the column of an internal cell (therefore `i*x_size+j` is its position), the neighbours' positions are: `(i-1)*x_size+j-1`, `(i-1)*x_size+j`, `(i-1)*x_size+j+1`, `i*x_size+j-1`, `i*x_size+j+1`, `(i+1)*x_size+j-1`, `(i+1)*x_size+j`, `(i+1)*x_size+j+1`. On the other hand, for edge cells these positions change; for instance, in case of a left edge cell (`j=0`) we have the neighbours in: `(i-1)*x_size+j`, `(i-1)*x_size+j+1`, `i*x_size+j-1`, `i*x_size+j+1`, `(i+1)*x_size+j-1`, `(i+1)*x_size+j`, `(i+1)*x_size+j+1`, `(i+2)*x_size+j-1`.

Since each process's grid has one more row on the top and one more on the bottom storing the neighbour bordering states, evolution of first and last rows' cells is in practice not different from that of internal ones. But still, left and right edge cells need a different updating process than internal ones.

To deal with that, we are presented with many possible options. The most trivial strategy would be to iterate over all cells and to place, at each iteration, an if statement that would evolve the cell's state according to whether it is a left edge, a right edge or an internal cell. This approach is very simple to implement but it creates a lot of conditional branching,

probably slowing down the program a lot.

Another, less naive, way would be to use one more column on the left to store the right edge and one more on the right to store the left one. This would not require any if statement, but it would comport additional usage of memory and computation; not a dramatic one, but still avoidable.

A more complicated way would be to let some threads evolve the edge cells and others evolve the internal ones. This strategy would avoid if statements and memory waste, but it would increase the number of cache misses and the workload would be pretty hard to be kept balanced.

All previous methods have problems regarding speed and/or ease of implementation. Therefore, we chose to go with a different strategy: previously to evolution, each thread computes the position of the first edge met during evolution and the first and the last rows it will have to evolved completely (i.e. from left edge to right edge). In this way, cells preceding the first edge met can be evolved using a simple for loop, and for loops can be used to evolve cells internal to "complete" rows and those remained at the end as well. With this approach no additional memory is used, the number of if statements is reduced to only one per thread per generation (needed for the case in which the first cell to be updated is a left edge cell) and the workload is perfectly balanced across threads.

As an example to better explain our approach, we present here the function for ordered evolution, called `ordered_evo`:

```
void static_evo(BOOL* my_grid, BOOL* my_grid_aux, const int x_size,  /
                int my_thread_start, int first_edge,  /
                const int first_row, const int last_row,  /
                const int my_thread_stop) {


    char count;   // counter of alive neighbor cells
    int position = my_thread_start;   // position of the cell to update


    /* updating cells preceding the first edge */

    for ( ; position<first_edge; position++) {

        count = 0;
        for (int b=position-x_size-1; b<position-x_size+2; b++) {
            count += my_grid[b];
        }
        count += my_grid[position-1];
        count += my_grid[position+1];
        for (int b=position+x_size-1; b<position+x_size+2; b++) {
            count += my_grid[b];
        }

        my_grid_aux[position] = (count == 2 || count == 3);
```

```
}


/* updating first edge encountered */

if (position == first_edge) {

    count = 0;
    count += my_grid[position-2*x_size+1];
    for (int b=position-x_size-1; b<position-x_size+2; b++) {
        count += my_grid[b];
    }
    count += my_grid[position-1];
    count += my_grid[position+1];
    for (int b=position+x_size-1; b<position+x_size+1; b++) {
        count += my_grid[b];
    }

    my_grid_aux[position] = (count == 2 || count == 3);
}


/* iteration on 'complete' rows */

for (int i=first_row; i<last_row; i++) {

    /* updating first element of the row */
    position = i*x_size;
    count = 0;
    for (int b=position-x_size; b<position-x_size+2; b++) {
        count += my_grid[b];
    }
    count += my_grid[position-1];
    count += my_grid[position+1];
    for (int b=position+x_size-1; b<position+x_size+2; b++) {
        count += my_grid[b];
    }
    count += my_grid[position+2*x_size-1];

    my_grid_aux[position] = (count == 2 || count == 3);

    /* iteration on internal columns (updating internal elements
    * of te row) */
    for (int j=1; j<x_size-1; j++) {
```

```
        count = 0;
        position = i*x_size+j;
        for (int b=position-x_size-1; b<position-x_size+2; b++) {
            count += my_grid[b];
        }
        count += my_grid[position-1];
        count += my_grid[position+1];
        for (int b=position+x_size-1; b<position+x_size+2; b++) {
            count += my_grid[b];
        }

        my_grid_aux[position] = (count == 2 || count == 3);
    }

    /* updating last element of the row */
    count = 0;
    position = (i+1)*x_size-1;
    count += my_grid[position-2*x_size+1];
    for (int b=position-x_size-1; b<position-x_size+2; b++) {
        count += my_grid[b];
    }
    count += my_grid[position-1];
    count += my_grid[position+1];
    for (int b=position+x_size-1; b<position+x_size+1; b++) {
        count += my_grid[b];
    }

    my_grid_aux[position] = (count == 2 || count == 3);
}

position++;


/* checking if all cells have been already updated */
if (position < my_thread_stop) {

    /* updating first element after last row */

    count = 0;
    for (int b=position-x_size; b<position-x_size+2; b++) {
        count += my_grid[b];
    }
    count += my_grid[position-1];
    count += my_grid[position+1];
    for (int b=position+x_size-1; b<position+x_size+2; b++) {
```

```
            count += my_grid[b];
        }
        count += my_grid[position+2*x_size-1];

        my_grid_aux[position] = (count == 2 || count == 3);

        position++;


        /* updating remaining elements */

        for ( ; position<my_thread_stop; position++) {

            count = 0;
            for (int b=position-x_size-1; b<position-x_size+2; b++) {
                count += my_grid[b];
            }
            count += my_grid[position-1];
            count += my_grid[position+1];
            for (int b=position+x_size-1; b<position+x_size+2; b++) {
                count += my_grid[b];
            }

            my_grid_aux[position] = (count == 2 || count == 3);
        }
    }

    return;
}
```

The structure of the other two functions for evolution `static_evo` and `static_evo_in_place` is roughly the same, with some differences.

In ordered evolution, a part from MPI communications in case of a single process (as we saw in section 1.3.4), the only difference is that, instead of updating `my_gird_aux`, we directly update `my_grid`.

For what concerns static in place evolution, the structure of the iteration over cells is still the same, but the updating mechanism is a little more complicated. First of all, we define an `int` variable (shared among threads) called `bit_control`, which is used in evolution and I/O to signal which, among the last and the second last of `BOOL`, is the bit storing the current cells' state. Then we define the following variables:

```
char alive = 2 - bit_control % 2;
char dead = 1 + bit_control % 2;
char shift = bit_control % 2;
```

where `shift` signals of how many bits `BOOL` has to be shifted to the right if we want to have the last bit as the old state signaling one; in other words: if `shift` is equal to 0, then we

read the last bit and update the second last, while if it is equal to 1 we read the second last and update the last. Here it is how this mechanism is implemented (recall that `count` is the alive neighbours counter):

```
count += (my_grid[relative_position] >> shift) & 1;
```

`alive` and `dead` are, instead, used for updating in the following way:

```
if (count == 2 || count == 3) {
    my_grid[position] |= alive;
} else {
    my_grid[position] &= dead;
}
```

At the end of each generation `bit_control` has to be increased by the master threads, so that in the following generation the right state signaling bit will be read.

1.3.6 Testing

While building the program each newly added part was tested individually.

In particular, it is worth briefly explaining how we checked the three kinds of evolution to work properly. First, we wrote a serial version of GOL that could be applied to playgrounds of any size. This version was checked "manually" on many different grids of different sizes and shapes (i.e. different `x_size-y_size` ratios). Once we made sure that both ordered and static evolution worked in this version, we tested the parallel version on large grids checking the coherence of the results w.r.t. the serial version using the shell command `diff`.

1.4 Measures

We measured performance on both EPYC and THIN nodes, using *close cores* thread affinity policy. To do that, we organized the directory in the following way:

```
.
|-- EPYC/
|    |-- openMP_scal/
|    |-- openMP_scal_mod/
|    |-- strong_MPI_scal/
|    |-- weak_MPI_scal/
|
|-- src/
|
|-- images/
|    |-- snapshots/
|
|-- Makefile
|
|-- THIN/
     |-- openMP_scal/
     |-- strong_MPI_scal/
```

```
|-- weak_MPI_scal/
```

`EPYC/` and `THIN/` contain the results obtained on the related partitions, organized in the three displayed directories containing the related kinds of scalability (`EPYC/` contains an additional folder for openMP scalability using `parallel_gol_mod.c`). `src/` contains the GOL source codes (both serial and parallel versions). `images/` contains all PGM files produced by the programs, in particular system's snapshots and final state are stored in `snapshots/`; notice that default read or written images (e.g. `game_of_life.pgm` initialised if no name is passed) are placed in this directory, however, the program is perfectly capable of initialising and reading images from elsewhere; in any case, the name has to be passed with the complete relative path.

1.4.1 Speedup

To measure the scaling we used, depending on the kind of scaling, the **speedup** or the inverse of it (i.e. the time in unit of elapsed serial time). By serial time we do not mean the serial version of GOL's time, but the time measured for the configuration with only one thread or one process depending on the kind of scaling; for instance, in strong MPI scalability, we speak about serial time for the configuration with only one process and saturated threads, even if it is not serial at all.

First, we had to choose the kind of time to measure. The often used kind is the CPU time. However, measuring the process CPU time would have somehow, in our opinion, distorted the measure. That because this kind of time is simply the sum of the times each thread used the CPU, and therefore we could only compute from it a kind of average of the time that each thread spent working. In our opinion, this is not a good measure of performance for our case, since it doesn't consider the capability of the program to keep parallel threads synchronized. What we want is, instead, a measure of the time passed between the moment in which the program started and the one in which it stopped. To do that we used the openMP function `omp_get_wtime`, which returns the elapsed wall clock time in seconds, called only by the master thread of the 0-th process and always after an `MPI_Barrier` to make sure that all processes have finished their job.

We chose to take each time measure on a (small) number of generations, obviously the same for all measures of a certain kind of scalability. We made this choice, instead of taking a single generation's time, for two reasons: one, that in this way we are already performing some kind of statistics from inside the code, and the other, more important, that the iteration of generations could introduce some kind of overhead itself, and in this way we are including it.

Another decision to make was which part of the code to benchmark. This is a little bit tricky. Our final decision was to measure the time passed between the beginning of the first generation and the end of the last (without taking any snapshots of the system). We made this choice because we thought that in a program like this, in which the number of generations is variable, the key thing to know is how well the repeated part scales. In fact in benchmarking, given our time limitations, we had to run the program for a very small number of generations; now, supposing that the user wants to run the program for a larger number of generations (which is likely the case), the time spent by the non-repeated part of code is not interesting for him, since it becomes more and more irrelevant as the

number of generations increases. At that point, the user will likely be interested only in how well the repeated part scales, and adding the non-repeated part to the time measured would introduce some kind of "noise" to that information. Of course, having more time and resources, the ideal thing would be to measure also the non repeated part of program separately and to give that information as well.

The elapsed time was saved to a file called `data.csv`.

Finally, it is worth noting that speedup and inverse speedup are not "absolute" measures of performance, since they are rescaled using the serial time. But this kind of measure is suitable for our aim, since we are only interested in studying scalability.

1.4.2 Makefile

To speedup the measurements we wrote a simple Makefile that can be used to compile both serial and parallel versions of GOL in a target directory, passed to the file by a variable called `$data_folder`. Parallel GOL can be optionally compiled in the "unique" version (i.e. with evolution functions embedded) and in the version with grid warm up.

The Makefile also contains two commands to clean the directory from executables and PGM files.

1.4.3 Job files

SLURM (ORFEO's resource manager software) allows the user to submit jobs in both interactive and batch modes. In interactive mode the user has to "manually" load modules, request resource allocation and wait for it, after that he can run the code. In batch mode, instead, the user can include module loading, resources requests and commands for the shell in a bash script (the *job file*); he can then submit the job by simply calling

```
batch job_file_name.sh
```

In this way he does not need to wait for free resources, and all the work will be done "autonomously". For our aim the batch mode was much more suitable.

In each one of the three directories `openMP_scal/`, `strong_MPI_scal/` and `weak_MPI_scal/` (`openMP_scal_mod/` in `EPYC/` contains the exact same stuff as `openMP_scal/` but with different make commands in the job file to compile the modified version of GOL) in each one of `EPYC/` and `THIN/` we placed a `job.sh` file with instructions to perform the requested measures. All these files have a similar structure, which can be described in the following way:

1. SLURM commands: a list of commands addressed directly to SLURM (preceded by `#` and therefore ignored by the shell) aimed to request the needed resources, in particular: the partition (`--partition`), the number of nodes (`-N`), the number of cores (`-n`) and the maximum amount of time (`--time`; the resources will be occupied only for the time needed to complete the task, but with the specified limit as an upper bound). The following is an example from the job file written for openMP scalability on EPYC partition:

   ```
   #!/bin/bash
   #SBATCH --no-requeue
   ```

```
#SBATCH --job-name="openMP_scal"
#SBATCH --partition=EPYC
#SBATCH -N 1
#SBATCH -n 128
#SBATCH --exclusive
#SBATCH --time=02:00:00
#SBATCH --output="summary.out"
```

For what concerns the other commands we have: `--no-requeue` used to tell SLURM to never requeue the job (for example after a node failure), `--job-name` used to simply change the job's name in the queue and `--exclusive` that asks for an exclusive use of the requested node. We also have the `--output` command, which renames the file to which the standard output will be readdressed; because of the way in which we wrote `job.sh` and `parallel_gol.c`, it is possible to look at this file (called `summary.out`) to check whether the task has been completed successfully or not and whether the right number of processes and threads has been used. For strong MPI and weak MPI scalabilities the chunk of code is identical a part from the resources allocation, which is of two entire nodes (`-N 2`, `-N 256`). On THIN nodes the code chunk is identical a part from `--partition=THIN` and `-n 24` (or `-n 48`). Finally, the job's name was set to `openMP_scal`, `strong_MPI_scal` or `weak_MPI_scal` depending on the kind of scalability, but any name would be fine (since it is only used to identify your job in the queue).

2. Module loading, threads affinity policy setting and compiling: the module system is addressed to load the needed architecture and the openMPI library, the threads affinity policy is set and the executables are compiled in the current directory. Here you can see an example from a job file written for EPYC:

```
echo LOADING MODULES...
echo
module load architecture/AMD
module load openMPI/4.1.4/gnu/12.2.1

echo SETTING THREADS AFFINITY POLICY...
echo
alloc=close
export OMP_PLACES=cores
export OMP_PROC_BIND=$alloc

echo COMPILING EXECUTABLES...
echo
datafolder=$(pwd)
cd ../..
make parallel_gol data_folder=$datafolder
cd $datafolder
```

In this chunk the only thing that changes for different job files is the loaded architecture: THIN nodes have Intel instead of AMD. The messages printed via `echo` are there just to make `summary.out` more readable.

3. Variables setting: in this part we set all fixed variables that we will use in the code, for instance in case of openMP scalability on EPYC we have:

```
node=EPYC
scal=openMP
mat_x_size=25000
mat_y_size=25000
n_gen=5
n_procs=2
```

while for strong and weak MPI scalability we have respectively:

```
node=EPYC
scal=strong_MPI
n_gen=5
n_threads=64
```

and

```
node=EPYC
scal=weak_MPI
unit_mat_size=10000
n_gen=5
n_threads=64
```

As you can see, we set the number of generations to 5 and we saturate sockets in both strong and weak MPI scalabilities.

4. Data file initialisation: the CSV files containing data (called `data.csv`) are created or overwritten, and a small "header" containing a summary of the conditions in which measurements were carried out is written. Again, for instance, openMP scalability on EPYC:

```
echo CREATING/OVERWRITING CSV FILE...
echo
datafile=$datafolder/data.csv
echo "#,,," > ${datafile}
echo "#node:,${node},," >> $datafile
echo "#scalability:,${scal},," >> $datafile
```

```
echo "#performance_measure:,time(s),," >> $datafile
echo "#threads_affinity_policy:,${alloc},," >> $datafile
echo "#playground_x_size:,${mat_x_size},," >> $datafile
echo "#playground_y_size:,${mat_y_size},," >> $datafile
echo "#generations:,${n_gen},," >> $datafile
echo "#sockets:,2,," >> $datafile
echo "#,,," >> $datafile
echo "#,,," >> $datafile
echo "threads_per_socket,ordered,static,static_in_place" >> $datafile
```

5. Proper computation: this part is different for different kinds of scalability. However there are some things in common. For all measurements we used the `mpirun` option `--map-by socket` to make sure that each process is placed on a different socket spawning threads on the cores of that socket. Also, we used a newly generated initial playground for each different measure, so that the measures are somehow more representative (less "biased"). For openMP scalability we wrote:

```
### generating random playground
export OMP_NUM_THREADS=64
mpirun -np 2 --map-by socket parallel_gol.x -i -m $mat_x_size  /
                                            -k $mat_y_size


for n_threads in $(seq 1 1 64)
do

    ### running the evolution
    export OMP_NUM_THREADS=$n_threads
    echo -n "${n_threads}" >> $datafile
    mpirun -np $n_procs --map-by socket parallel_gol.x -r -e 0  /
                                            -n $n_gen -s 0
    mpirun -np $n_procs --map-by socket parallel_gol.x -r -e 1  /
                                            -n $n_gen -s 0
    mpirun -np $n_procs --map-by socket parallel_gol.x -r -e 2  /
                                            -n $n_gen -s 0
    echo >> $datafile

    echo
    echo -----------
    echo
done
```

For strong MPI scalability we had less computation to do, so we had time to perform some statistics (we repeated each measure 5 times) and to repeat the scalig for three different matrix sizes (10000x10000, 20000x20000, 30000x30000):

```
export OMP_NUM_THREADS=$n_threads

for mat_size in $(seq 10000 10000 30000)
do
    for count in $(seq 1 1 5)
    do
        ### generating random playground
        mpirun -np 4 -N 2 --map-by socket parallel_gol.x -i -m  /
                                    $mat_size -k $mat_size

        for n_procs in $(seq 1 1 4)
        do

            ### running the evolution
            echo -n "${mat_size}x${mat_size}," >> $datafile
            echo -n "${n_procs}" >> $datafile
            mpirun -np $n_procs -N 2 --map-by socket parallel_gol.x  /
                                       -r -e 0 -n $n_gen -s 0
            mpirun -np $n_procs -N 2 --map-by socket parallel_gol.x  /
                                       -r -e 1 -n $n_gen -s 0
            mpirun -np $n_procs -N 2 --map-by socket parallel_gol.x  /
                                       -r -e 2 -n $n_gen -s 0
            echo >> $datafile

            echo
            echo -----------
            echo
        done
    done
done
```

Finally, for weak MPI scalability we did (again, we had time to do some statistics):

```
export OMP_NUM_THREADS=$n_threads

for count in $(seq 1 1 5)
do
    for n_procs in $(seq 1 1 4)
    do
        ### generating random playground
        mat_x_size=$((unit_mat_size*n_procs))
        mpirun -np 4 -N 2 --map-by socket parallel_gol.x -i -m  /
                                $mat_x_size -k $unit_mat_size

        ### running the evolution
```

```
            echo -n "${mat_x_size}x${unit_mat_size}," >> $datafile
            echo -n "${n_procs}" >> $datafile
            mpirun -np $n_procs -N 2 --map-by socket parallel_gol.x  /
                                        -r -e 0 -n $n_gen -s 0
            mpirun -np $n_procs -N 2 --map-by socket parallel_gol.x  /
                                        -r -e 1 -n $n_gen -s 0
            mpirun -np $n_procs -N 2 --map-by socket parallel_gol.x  /
                                        -r -e 2 -n $n_gen -s 0
            echo >> $datafile

            echo
            echo -----------
            echo
        done
    done
```

6. Module releasing, executables cleaning and PGM files cleaning:

```
echo RELEASING LOADED MODULES AND CLEANING FROM EXECUTABLES AND IMAGES...
cd ../..
make clean_all data_folder=$datafolder
module purge
cd $datafolder
```

1.5 Results

The following charts report speedup (or inverse speedup) computed on data gathered for all kinds of scalability, on both node partitions and for all kinds of evolution ("static ip" indicates static in place evolution). The expected speedup (or inverse speedup) is also plotted as a dashed line.

Notice that ordered evolution is **not** expected to scale, since its execution is intrinsically serial. In this case the best we can hope is to maintain the serial time when going parallel, meaning that communications and parallel regions are made so efficient that their time is not relevant w.r.t. the final one.

1.5.1 OpenMP scalability

As you can see from figures 1 and 2, we got better results on THIN nodes. This is probably caused, in part, by the fact that THIN nodes have less cores per socket, since EPYC nodes present good results up to a certain number of cores as well. Two possible reasons why this could be the cause are the following:

1. **Distance from RAM**: since we allocate the grid outside the parallel region, i.e. using the master thread, all data are placed in those banks of memory that are the nearest to the core hosting the master thread. Since we use close cores threads affinity

policy, the spawned threads will be placed on cores the nearest possible to the master, and therefore to memory banks storing the grid. However, increasing the number of threads causes new threads to be placed further and further from those memory banks, slowing the average time to get data.

2. **False sharing**: increasing the number of threads means having each thread working on smaller portions of data. Since data are stored in a contiguous way and since caches work "by lines", the more cores you have working on the grid the more likely it will be to have the same piece of data in two different cores' cache at the same time. Therefore, an overhead is introduced to keep all caches coherent.

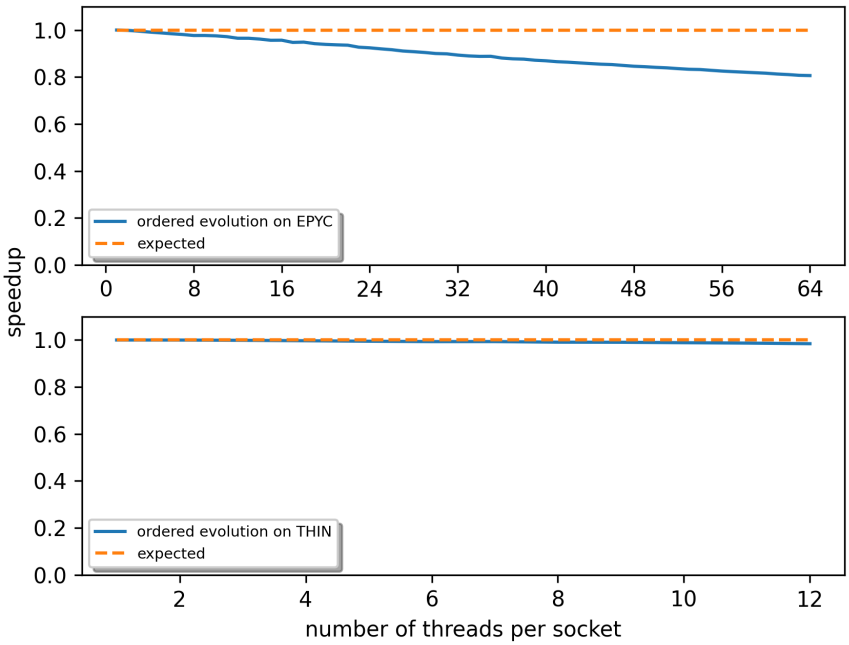

Figure 1: Scalability: openMP, evolution: ordered, matrix size: 25000x25000, number of sockets: 2

1.5.2 Strong MPI scalability

Here we present data obtained with matrix size 20000x20000, but those obtained with 10000x10000 and 30000x30000 are not much different.

From figures 3 and 4 we see again that results are much better on THIN w.r.t. EPYC, especially for ordered evolution. In this case distance from RAM and false sharing should not be a problem, since each process allocates its own grid. The problem, instead, is probably in communications among processes which are, for some reason, less efficient on EPYC nodes. The exact reason for this is hard to tell, since it probably resides in hardware details. However, we can make the hypothesis that, having EPYC sockets more cores, they probably have larger sizes as well, making the communications between them slower.

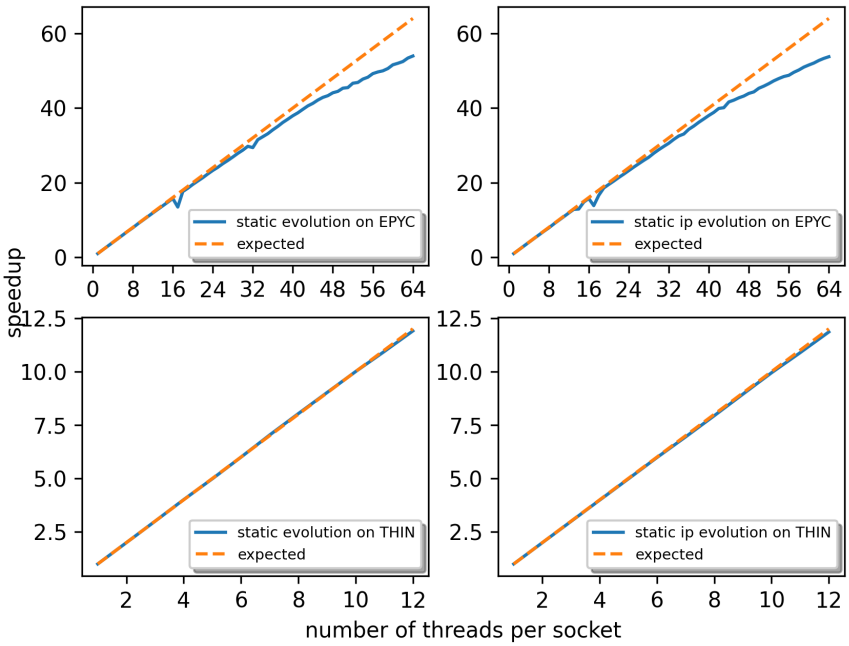

Figure 2: Scalability: openMP, evolution: static and static in place, matrix size: 25000x25000, number of sockets: 2

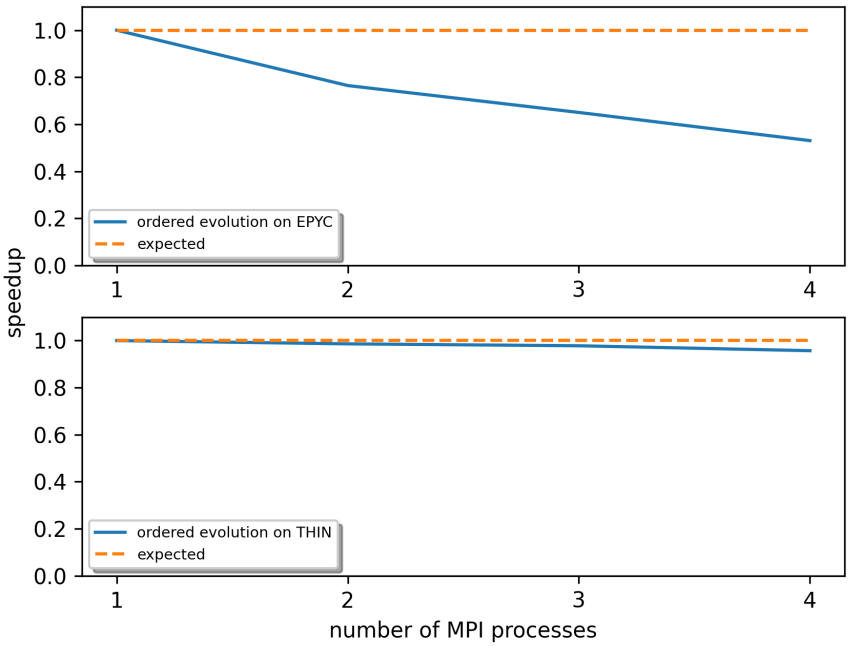

Figure 3: Scalability: strong MPI, evolution: ordered, matrix size: 20000x20000, number of threads per socket: maximum (64 for EPYC and 12 for THIN)

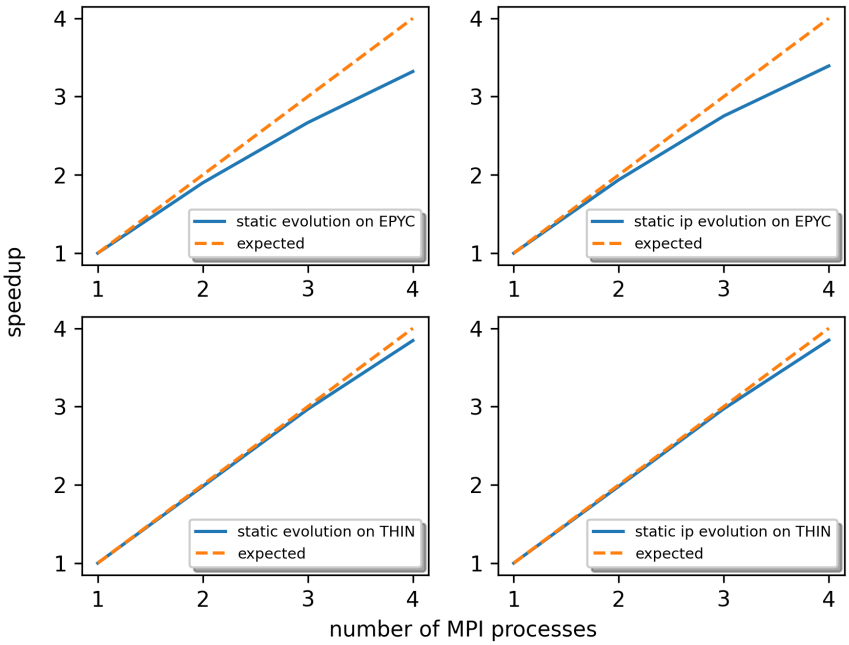

Figure 4: Scalability: strong MPI, evolution: static and static in place, matrix size: 20000x20000, number of threads per socket: maximum (64 for EPYC and 12 for THIN)

1.5.3 Weak MPI scalability

The analysis of these data (see figures 5 and 6) seems to confirm the better efficiency of MPI communications on THIN nodes w.r.t. EPYC.

1.5.4 Static Vs static in place

Since static in place evolution requires some additional computation w.r.t. static non in place, we expect to have a slightly worse performance with the first one. Here we present a very quick comparison of this two kinds of evolution's performances. In figures 7 we plotted data collected on EPYC nodes, but those collected on THIN nodes give a very similar picture.

As it is apparent from the plots, the difference is almost negligible. This makes the choice of static in place evolution almost obvious, unless even a minimum improvement in performance is important or unless we have a very large availability of memory w.r.t. the size of the problem.

1.5.5 Adding grid warm up

A simple attempt to fix false sharing and distance from RAM is that of doing a previous "warm up" of the grids, meaning that before reading the initial playground each thread accesses its own part of the grid. Doing so, each portion of grid is stored in the nearest RAM banks to the core that will work on it. We tried to modify the code in this way, but we did not gain much improvement: as you can see from figure 8, the results we got are similar

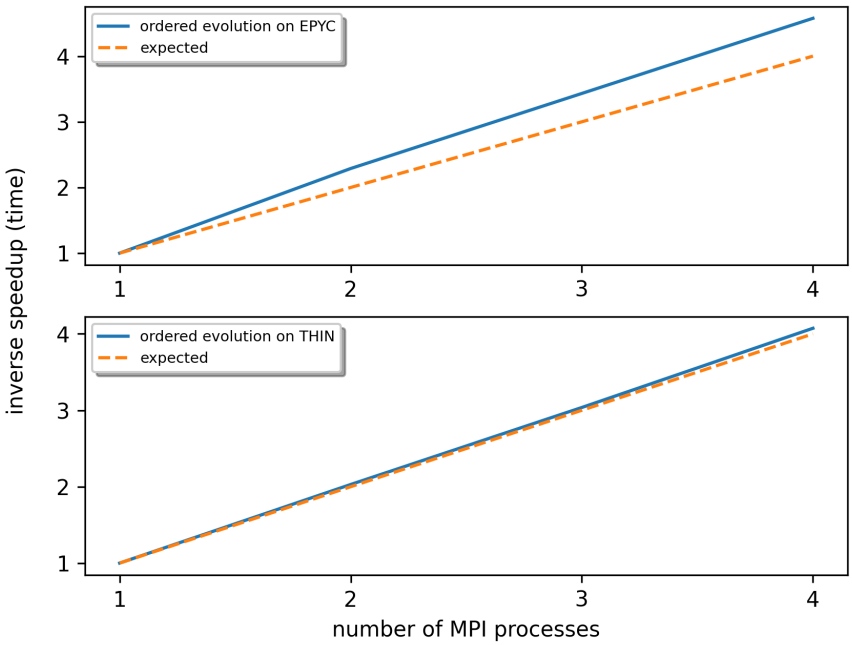

Figure 5: Scalability: weak MPI, evolution: ordered, starting matrix size: 10000x10000, number of threads per socket: maximum (64 for EPYC and 12 for THIN)

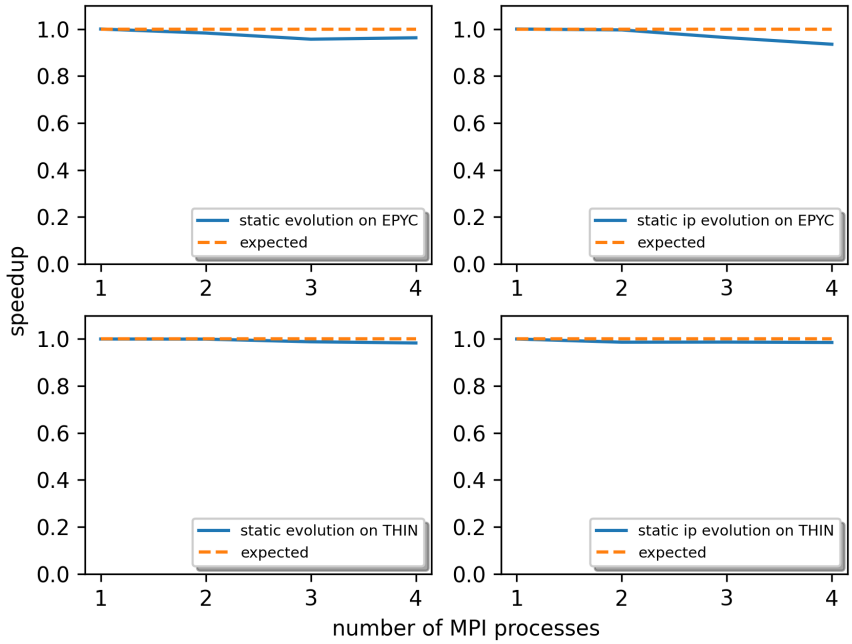

Figure 6: Scalability: weak MPI, evolution: static and static in place, starting matrix size: 10000x10000, number of threads per socket: maximum (64 for EPYC and 12 for THIN)

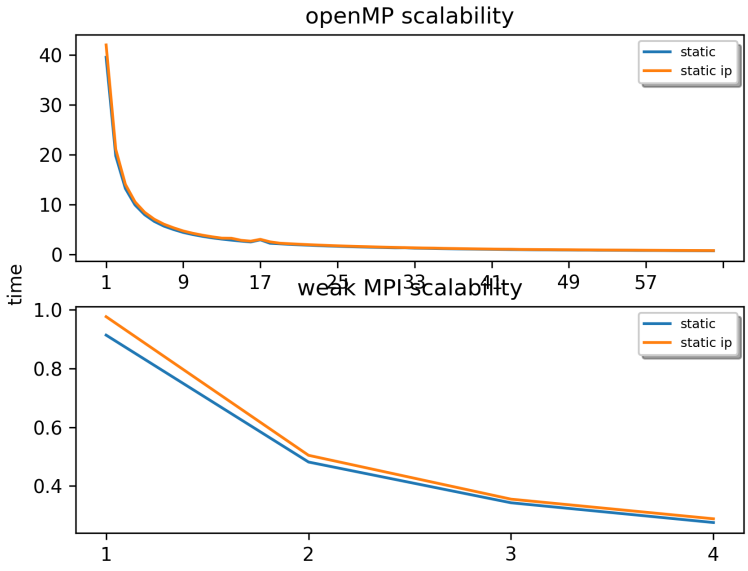

Figure 7: Comparison by absolute running time between static and static in place evolution for openMP scalablity and strong MPI scalability; on the x axis the number of MPI processes or threads per socket

to those obtained without warming up, in particular we still have the loss of performance for large numbers of cores.

It is worth pointing out that adding warm up will likely make I/O operations slower, since these operations are performed by master threads alone, and now grids are "scattered" among all RAM banks of the socket, instead of being entirely placed in the closest to the cores on which master threads are running. In many cases (especially for large grid sizes), it is likely that the little gain in performance given by warm up is not worth losing I/O speed.

1.6 Conclusions

We have seen that in all kinds of scaling and for all kinds of evolution, the code scales better on THIN nodes w.r.t. EPYC.

We individuated three possible sources limiting scaling: false sharing, distance from RAM and speed of communications among sockets.

We tried to fix false sharing and distance from RAM on EPYC by doing a warm up of the grid, but the results were still poor for large numbers of cores. In our opinion, the proposed solution does not work because false sharing and distance from RAM are a problem mostly while computing the number of alive neighbours. Warming up only partially solves this problem, since part of some cells' neighbours will be far from the core anyway (since they may "belong" to another threads' space), and, in fact, the larger the number of threads the more this is likely to happen.

The only way we can think of to solve these issues is to distribute cells among threads in a different way. For instance, we could use as the "unit block" to build each threads' portion of grid a square group of cells fitting the cache. In this way the "perimeter" of each

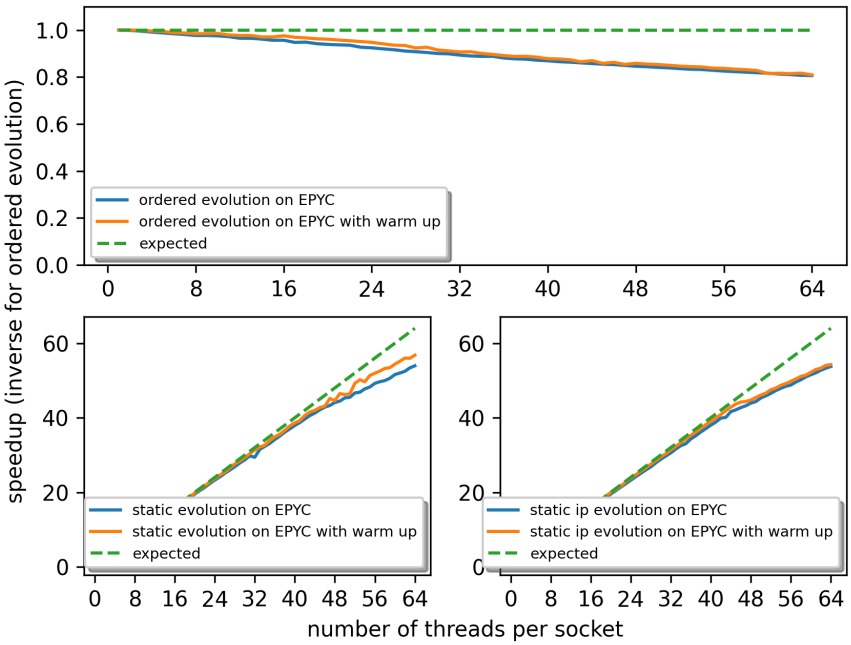

Figure 8: Comparison by speedup (for static and static in place evolution) and inverse speedup (for ordered evolution) among GOL with and without warming up the grid

thread's grid portion would likely decrease (especially for large numbers of threads), leading to a smaller number of cache misses.

In any case, the effect of false sharing and distance from RAM should decrease for increasing grid sizes. So the program will probably scale up to a higher number of cores for larger playgrounds.

For what concerns speed of communications, we do not have any non-hardware solutions for that. However, this problem should be less relevant when using grid sizes with a larger `y_size` / `x_size` ratio, since in that case data to be communicated among processes are less heavy in proportion.

Finally, we point out the fact that we used a close cores threads affinity policy, which we considered to be more appropriate to reduce the slowing effect of distance from RAM (and, somehow, even false sharing since L3 cache is shared in groups of four close cores). However, we cannot exclude different policies to give better results. Having more space and time, we could have experimented with them as well.

2 Exercise 2

2.1 Aim of the exercise

In this second part of the assignment we were asked to compare the performance of three high performance math libraries, namely MKL, openBLAS and BLIS, and to compare it with the theoretical peak performance. In particular we had to perform measures on a level

3 BLAS (*Basic Linear Algebra Subprograms*) function called `gemm`, available in both single and double point precision.

The **level 3 BLAS** category contains matrix-matrix operations including our `gemm` (*GEneral Matrix Multiplication*), in the form:

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

(in our case we will set $\alpha = 1, \beta = 0$ and we will consider only square matrices). The nature of matrix-matrix operations is such that, when the size of the problem grows, the number of floating point operations to be performed grows faster than the number of memory transfers needed (to understand that it is pretty helpful to think, for instance, about the way we usually compute row-column matrix multiplications on paper); in this way we can exploit our computational resources in the best way possible. This kind of operations are said to be **not memory bounded**. For details on BLAS operations we invite you to read the related wikipedia page (Wikipedia, 2004).

The measures had to be done in different conditions, i.e. choosing one and only one option for each of the following parameters at a time:

- Node partition: EPYC or THIN
- Precision of the `gemm` function: float (i.e. single point) or double (i.e. double point)
- Threads affinity policy (the options were of our choice): close cores or spread cores
- Math library: MKL, openBLAS or BLIS
- Varying parameter: number of cores (at fixed matrix size) or matrix size (at fixed number of cores)

For varying number of cores we chose to fix the matrix size to 10000 and to take the number of cores ranging from 1 to the maximum available on the node, while for varying matrix size we fixed the number of cores to half of the maximum available on the node (so 64 for EPYC and 12 for THIN) and took matrix sizes from 2000 to 20000.

We will call **configuration** a possible set of parameters, for example: "EPYC node, float precision, close cores threads affinity policy, MKL library, varying matrix size".

To accomplish the task we were already given two pieces of code:

- `gemm.c` (actually called `dgemm.c` in the oroginal course repository): a code that can be used to call `gemm` either on single or double point precision and either from MKL, openBLAS or BLIS library
- `Makefile`: a Makefile that can be used to compile `gemm.c` in double point precision and with any of the three libraries

Performance was measured in both elapsed time and *GFLOPS* (*giga-floating point operations per second*).

The BLIS library had to be compiled by the student on the target machine, and the right path to it had to be assigned to the `$BLISROOT` variable in the Makefile.

2.1.1 Theoretical peak performance

The **theoretical peak performance** can be computed as:

$$TPP = clock\_rate \times \#flops\_per\_cycle \times \#cores.$$

In the case of EPYC nodes we have a base clock rate of $2.6GHz$[1] and a double precision floating point operation capability of up to 16 operations per clock cycle[2]. Therefore, for each core we have a contribute of $TPP^{double}_{EPYC} = 41.6GFLOPS$ in double point precision and, consequently, $TPP^{float}_{EPYC} = 83.2GFLOPS$ in single point precision.

For what concerns THIN instead, we have for an entire node a TPP of $1.997TFLOPS$ in double point precision (as reported in the course slides here), therefore for a single core:

$$TPP^{double}_{THIN} = \frac{1.997TFLOPS}{24} = 83.2GFLOPS,$$

and so $TPP^{float}_{THIN} = 166.4GFLOPS$.

2.2 Implementation

To carry out the measurements we used SLURM's sbatch mode (see section 1.4.3 for details about how sbatch mode works). We created eight different folders, one for each combination of node partition, varying parameter and threads affinity policy. In each of these folders we placed a job file called `job.sh` to run the computations on the cluster. All of these job files have a very similar structure and can be set to a specific configuration (i.e. set the last two parameters left: library and precision) by simply uncommenting the corresponding lines. Each time we run the job, all the six possible executables obtained using the combination of library and precision are compiled in the folder containing `job.sh`, but only the one (or ones) chosen by uncommenting lines is used to collect data, for each configuration in a different CSV file. As a result, data are saved in 48 CSV files, one for each possible configuration.

To better explain the files organization here is the directory's tree (the eight folders containing data are grouped in two larger parent folders, one for each node partition):

```
.
|-- EPYC/
|    |-- cores_w_close_cores/
|    |-- cores_w_spread_cores/
|    |-- size_w_close_cores/
|    |-- size_w_spread_cores/
|
|-- gemm.c
|
|-- Makefile
|
|-- THIN/
```

[1] This information can be found on the official AMD website

[2] We were not able to find this information on the official AMD website, but most web sources agree on it, for example see here

```
|-- cores_w_close_cores/
|-- cores_w_spread_cores/
|-- size_w_close_cores/
|-- size_w_spread_cores/
```

Each subdirectory of `EPYC` and `THIN` is related to a different combination of varying parameter and threads affinity policy. And each one of them has the same content:

```
.
|-- blis_d.csv
|-- blis_f.csv
|-- job.sh
|-- mkl_d.csv
|-- mkl_f.csv
|-- oblas_d.csv
|-- oblas_f.csv
|-- summary.out
```

Let's now look more in detail at all the files we used.

2.2.1 gemm.c

We slightly modified the original code to save results in the right CSV file. More precisely, we added an option we called `SAVE_RESULTS`, such that if passed during compilation it causes the program to write elapsed time and GFLOPS to the CSV file `DATAFILE` (created in the folder containing the executable), whose name is defined depending on library and precision using preprocessor directives. Messages printed to standard output are kept anyway and used as a way to check whether the computation was run successfully until the end (see 2.2.3 for details).

2.2.2 Makefile

We modified the file so that it is possible to compile `gemm.c` in both single and double point precision, to save results to file and to compile the executables in the right folder. The folder in which the executables have to be produced is passed to `Makefile` through the variable `$data`.

2.2.3 job.sh

As we previously said, these files are all very similar and each one of them can be set for any combination of library and precision. Each one of our job files can be divided into several blocks of instructions:

1. SLURM commands: we already addressed the meaning of this commands in section 1.4.3 of exercise 1, so we will not repeat it. The following is an example from a job file written for the EPYC partition:

   ```
   #!/bin/bash
   #SBATCH --no-requeue
   ```

```
#SBATCH --job-name="cores_scal"
#SBATCH --partition=EPYC
#SBATCH -N 1
#SBATCH -n 128
#SBATCH --exclusive
#SBATCH --time=02:00:00
#SBATCH --output="summary.out"
```

Because of the way in which we wrote `job.sh` and `gemm.c`, it is possible to look at this file (called `summary.out`) to check whether the task has been completed successfully or not and whether the right matrix size has been used. For THIN nodes the code chunk is identical a part from `--partition=THIN` and `-n 24`.

2. Module loading and compiling: the module system is addressed to load the needed architecture and libraries, the path to "manually built" BLIS library is exported and the executables are compiled in the current directory. Here you can see an example from a job file written for EPYC:

```
echo LOADING NEEDED MODULES...
echo
module load architecture/AMD
module load mkl
module load openBLAS/0.3.21-omp
export LD_LIBRARY_PATH=/u/dssc/ttarch00/myblis/lib:$LD_LIBRARY_PATH
echo
echo COMPILING PROGRAMS FOR OPENBLAS, MKL AND BLIS ON TARGET MACHINE...
echo
datafolder=$(pwd)
cd ../..
make float data=$datafolder
make double data=$datafolder
cd $datafolder
```

In this chunk the only thing that changes for different configurations is the loaded architecture: THIN nodes have `Intel` instead of `AMD`. The messages printed via `echo` are there just to make `summary.out` more readable.

3. Threads affinity policy and other variables setting: this part slightly changes depending on the configuration. Here is an example from the job file with parameters "EPYC node, close cores threads affinity policy, varying number of cores":

```
size=10000
node=EPYC
alloc=close
```

```
### setting threads allocation policy
export OMP_PLACES=cores
export OMP_PROC_BIND=$alloc
```

`OMP_PLACES` and `OMP_PROC_BIND` are environment variables used to set the threads affinity policy, the variable `$size` will be later passed to the executables and `$node` will be used for printing. In case of varying matrix size another line is added to this block to set the number of processes:

```
export OMP_NUM_THREADS=$ncores
```

where `OMP_NUM_THREADS` is the environment variable controlling the number of openMP threads. Each library has its own environment variable to control the number of processes, but if they are not specified this one "decides".

4. CSV initialisation: file creation (or overwriting if it already exists) and parameters' values writing (for details on the structure of CSV files see 2.2.4). This part is exactly the same for all job files:

```
### overwriting old datafiles and setting up new ones
#echo "#,,," > mkl_f.csv
#echo "#node:,${node},," >> mkl_f.csv
#echo "#library:,MKL,," >> mkl_f.csv
#echo "#precision:,float,," >> mkl_f.csv
#echo "#allocation:,${alloc},," >> mkl_f.csv
#echo "#,,," >> mkl_f.csv
#echo "cores,mat_size,time(s),GFLOPS" >> mkl_f.csv

#echo "#,,," > oblas_f.csv
#echo "#node:,${node},," >> oblas_f.csv
#echo "#library:,openBLAS,," >> oblas_f.csv
#echo "#precision:,float,," >> oblas_f.csv
#echo "#allocation:,${alloc},," >> oblas_f.csv
#echo "#,,," >> oblas_f.csv
#echo "cores,mat_size,time(s),GFLOPS" >> oblas_f.csv

#echo "#,,," > blis_f.csv
#echo "#node:,${node},," >> blis_f.csv
#echo "#library:,BLIS,," >> blis_f.csv
#echo "#precision:,float,," >> blis_f.csv
#echo "#allocation:,${alloc},," >> blis_f.csv
#echo "#,,," >> blis_f.csv
#echo "cores,mat_size,time(s),GFLOPS" >> blis_f.csv

#echo "#,,," > mkl_d.csv
```

```
#echo "#node:,${node},," >> mkl_d.csv
#echo "#library:,MKL,," >> mkl_d.csv
#echo "#precision:,double,," >> mkl_d.csv
#echo "#allocation:,${alloc},," >> mkl_d.csv
#echo "#,,," >> mkl_d.csv
#echo "cores,mat_size,time(s),GFLOPS" >> mkl_d.csv

#echo "#,,," > oblas_d.csv
#echo "#node:,${node},," >> oblas_d.csv
#echo "#library:,openBLAS,," >> oblas_d.csv
#echo "#precision:,double,," >> oblas_d.csv
#echo "#allocation:,${alloc},," >> oblas_d.csv
#echo "#,,," >> oblas_d.csv
#echo "cores,mat_size,time(s),GFLOPS" >> oblas_d.csv

#echo "#,,," > blis_d.csv
#echo "#node:,${node},," >> blis_d.csv
#echo "#library:,BLIS,," >> blis_d.csv
#echo "#precision:,double,," >> blis_d.csv
#echo "#allocation:,${alloc},," >> blis_d.csv
#echo "#,,," >> blis_d.csv
#echo "cores,mat_size,time(s),GFLOPS" >> blis_d.csv
```

As you can see all lines are commented. While comments beginning by `###` are thought to be "true comments", the ones beginning by a single `#` are those that have to be uncommented depending on which configuration we want to run: each subblock of six lines corresponds to a different configuration. For instance, if we want to run the configuration with openBLAS library and double point precision, we should uncomment the fifth block, in the following way:

```
### overwriting old datafiles and setting up new ones
#echo "#,,," > mkl_f.csv
#echo "#node:,${node},," >> mkl_f.csv
#echo "#library:,MKL,," >> mkl_f.csv
#echo "#precision:,float,," >> mkl_f.csv
#echo "#allocation:,${alloc},," >> mkl_f.csv
#echo "#,,," >> mkl_f.csv
#echo "cores,mat_size,time(s),GFLOPS" >> mkl_f.csv

#echo "#,,," > oblas_f.csv
#echo "#node:,${node},," >> oblas_f.csv
#echo "#library:,openBLAS,," >> oblas_f.csv
#echo "#precision:,float,," >> oblas_f.csv
#echo "#allocation:,${alloc},," >> oblas_f.csv
#echo "#,,," >> oblas_f.csv
```

```
#echo "cores,mat_size,time(s),GFLOPS" >> oblas_f.csv

#echo "#,,," > blis_f.csv
#echo "#node:,${node},," >> blis_f.csv
#echo "#library:,BLIS,," >> blis_f.csv
#echo "#precision:,float,," >> blis_f.csv
#echo "#allocation:,${alloc},," >> blis_f.csv
#echo "#,,," >> blis_f.csv
#echo "cores,mat_size,time(s),GFLOPS" >> blis_f.csv

#echo "#,,," > mkl_d.csv
#echo "#node:,${node},," >> mkl_d.csv
#echo "#library:,MKL,," >> mkl_d.csv
#echo "#precision:,double,," >> mkl_d.csv
#echo "#allocation:,${alloc},," >> mkl_d.csv
#echo "#,,," >> mkl_d.csv
#echo "cores,mat_size,time(s),GFLOPS" >> mkl_d.csv

echo "#,,," > oblas_d.csv
echo "#node:,${node},," >> oblas_d.csv
echo "#library:,openBLAS,," >> oblas_d.csv
echo "#precision:,double,," >> oblas_d.csv
echo "#allocation:,${alloc},," >> oblas_d.csv
echo "#,,," >> oblas_d.csv
echo "cores,mat_size,time(s),GFLOPS" >> oblas_d.csv

#echo "#,,," > blis_d.csv
#echo "#node:,${node},," >> blis_d.csv
#echo "#library:,BLIS,," >> blis_d.csv
#echo "#precision:,double,," >> blis_d.csv
#echo "#allocation:,${alloc},," >> blis_d.csv
#echo "#,,," >> blis_d.csv
#echo "cores,mat_size,time(s),GFLOPS" >> blis_d.csv
```

Considering that each job file belongs to a different folder specifying a combination of partition, threads affinity policy and varying parameter, by uncommenting the file in this way we are specifying a complete configuration.

5. Actual measure performing: data collecting and saving to the previously initialised CSV file. This part differs in that the iteration is over the number of cores or the matrix size depending on the varying parameter. For completeness we report both the case of varying number of cores:

```
### performing measures
for ncores in $(seq 1 1 128)
```

```
do
    export OMP_NUM_THREADS=$ncores

    for count in $(seq 1 1 5)
    do
        #echo -n "${ncores}," >> mkl_f.csv
    #./gemm_mkl_f.x $size $size $size
    #echo -n "${ncores},"  >> oblas_f.csv
    #./gemm_oblas_f.x $size $size $size
    #echo -n "${ncores},"  >> blis_f.csv
    #./gemm_blis_f.x $size $size $size
    #echo -n "${ncores}," >> mkl_d.csv
      #./gemm_mkl_d.x $size $size $size
    #echo -n "${ncores},"  >> oblas_d.csv
    #./gemm_oblas_d.x $size $size $size
    #echo -n "${ncores},"  >> blis_d.csv
    #./gemm_blis_d.x $size $size $size
    echo
    echo -----------
    echo
    done
done
```

and that of varying matrix size:

```
### performing measures
for size in $(seq 2000 250 20000)
do
for count in $(seq 1 1 5)
do
#echo -n "${ncores}," >> mkl_f.csv
#./gemm_mkl_f.x $size $size $size
#echo -n "${ncores},"  >> oblas_f.csv
#./gemm_oblas_f.x $size $size $size
        #echo -n "${ncores},"  >> blis_f.csv
#./gemm_blis_f.x $size $size $size
#echo -n "${ncores}," >> mkl_d.csv
#./gemm_mkl_d.x $size $size $size
#echo -n "${ncores},"  >> oblas_d.csv
#./gemm_oblas_d.x $size $size $size
#echo -n "${ncores},"  >> blis_d.csv
#./gemm_blis_d.x $size $size $size
echo
echo -----------
echo
```

```
done
done
```

To run a specific configuration we need to uncomment the corresponding two lines inside the inner loop; for example the third and the fourth lines after `do` in the inner loop if we want the configuration with openBLAS and single point precision to be run. The function of the inner loop is to repeat the measure with the aim of doing a little bit of statistics. Note that in the first case we have to set the number of openMP processes, while in the second case we have already set it at point 3.

6. Module releasing and executables cleaning:

```
echo REMOVING COMPILED PROGRAMS AND UNLOADING MODULES...
echo
cd ../..
make clean data=$datafolder
module purge
```

This is the same for all job files.

2.2.4 CSV files

The name of all these files is in the form `$lib_$prec.csv`, in which `$lib` is `mkl`, `oblas` or `blis` depending on the library and `$prec` is `f` for single point precision and `d` for double point precision (for example `blis_f.csv`). They all have the same structure: some lines (commented with `#`) to summarize the configuration followed by data displayed in four columns: `cores` (number of cores), `mat_size`, `time(s)` (elapsed time) and `GFLOPS`. As an example we show the first lines of (the "raw version" of) one file:

```
#,,,
#node:,THIN,,
#library:,BLIS,,
#precision:,float,,
#allocation:,close,,
#,,,
cores,mat_size,time(s),GFLOPS
12,2000,0.030444,525.558432
12,2000,0.024279,658.996279
12,2000,0.030727,520.707869
12,2000,0.030399,526.338765
12,2000,0.023769,673.142635
12,2250,0.039930,570.529020
12,2250,0.039107,582.531921
12,2250,0.032089,709.935450
12,2250,0.039008,584.021615
```

```
12,2250,0.040056,568.730392
12,2500,0.051372,608.304783
12,2500,0.040489,771.818951
12,2500,0.041447,753.970263
12,2500,0.064145,487.174716
```

2.3 Results and conclusions

In this section we report a (mainly graphical) analysis of the results obtained in the different configurations. Each graph will show data related to the six possible combinations of library and threads affinity policy for a different combination of node partition, varying parameter and precision. Performances will be compared among different configurations and all of them will be compared to the theoretical peak performance (TPP - dashed line in the plots).

2.3.1 Cores scalability

In the following graphs we examine the behaviour with the number of cores as the varying parameter. Ideally, the performance should increase proportionally to the number of cores and be roughly equal to the theoretical peak one.

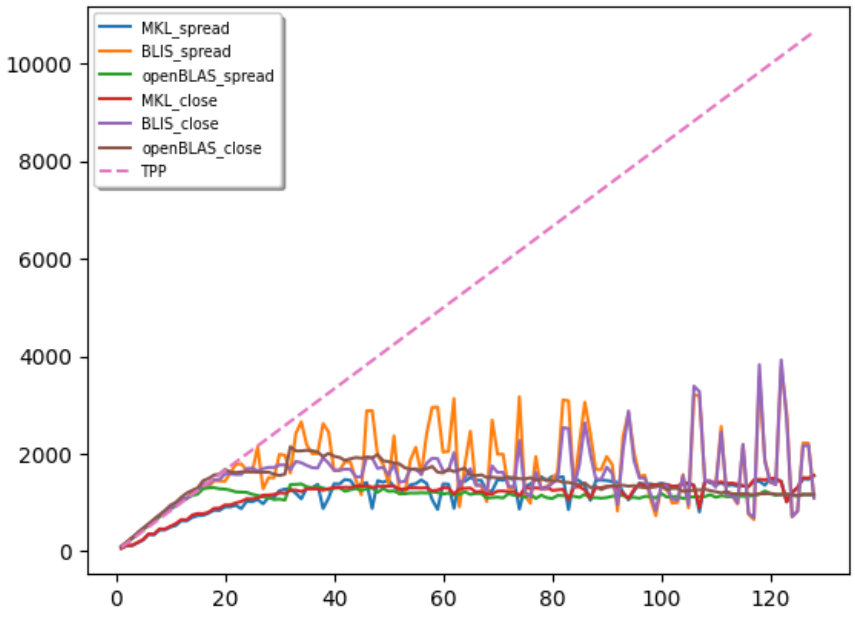

Figure 9: Varying parameter: number of cores, node partition: EPYC, precision: float, matrix size: 10000

EPYC As you can see from figures 9 and 10, as long as we have a number of cores smaller than 20 openBLAS and BLIS perform at the top of the machine's capabilities. Despite giving worse results sometimes, MKL seems to scale and to have an averall decent performance.

Going beyond 20-25 cores the gap between theoretical and actual performances grows more and more. Not only libraries' performances don't grow, they even decrease. The

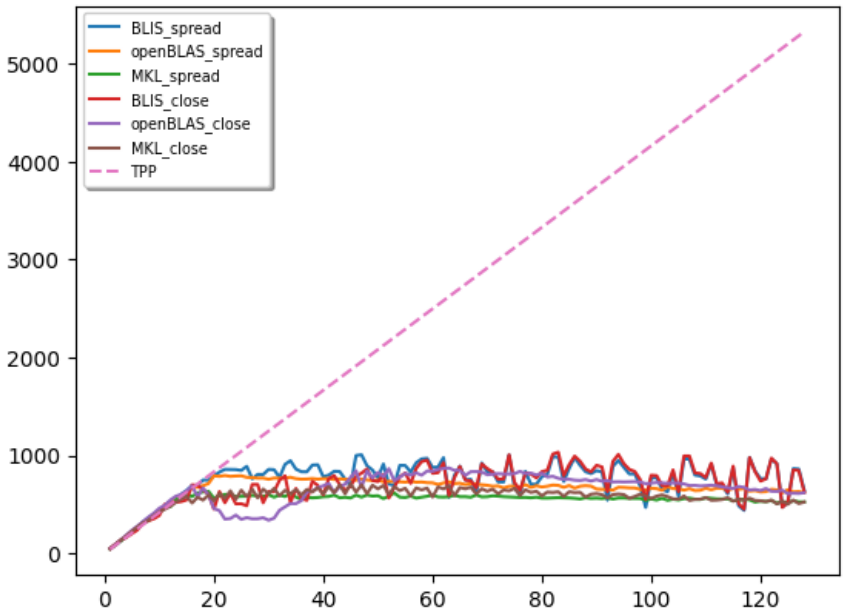

Figure 10: Varying parameter: number of cores, node partition: EPYC, precision: double, matrix size: 10000

strangest behaviour is the BLIS' one, whose performance seems to decrease on average (as the other two) but with an extremely variable ("swinging") trend.

Our (speculative) explanation for this general decrease is that the chosen matrix size is too small for the operation to be "efficient" in the use of caches. In fact, we must take into account two facts: first, each core has its own cache (apart from the level 3 which is shared in groups of four) and, second, in `gemm.c` matrices are initialised by the master thread alone, meaning that the memory used for each matrix is allocated contiguously. As a consequence, we can have what is usually called **false sharing**: each time a thread accesses some data, not only those data but also the neighbour ones are loaded to the related cache, and since those data could have already been accessed by some other thread, a certain effort is required to keep cache coherence among threads. Clearly, the larger the number of cores, the smaller the portions of matrices on which each core works and so the more frequently a core will access some data already loaded to some other core's cache. As a result, the portion of time spent to keep cache coherence will increase when increasing the number of cores. In other words, there is some kind of trade-off between workload distribution among processes and efficiency of the cache coherence keeping system. Therefore, it is our speculation that with a larger matrix size (and, consequently, larger portions of work for each core) "cores scaling" would result in being respected up to a number of cores larger than 20.

Another reason for this drop in performance could be that, since memory access is not uniform across all the node, the fact that matrices are initialised by a single thread should cause cores that are far from the master to access memory more slowly. This effect, even if probably present, seems to be not very large, otherwise close cores affinity policy would outperform spread cores one (at least for small numbers of cores). But this is not the case, as you can see from figures 11 and 12 (same graphs as the previous two, but without TPP to better confront the configurations).

Finally, it is worth noting how the (already irrelevant) difference among the two policies disappears for large numbers of cores. The reason for this is obvious: when the number of cores saturates the available resources on the node, the order in which you choose them becomes indifferent.

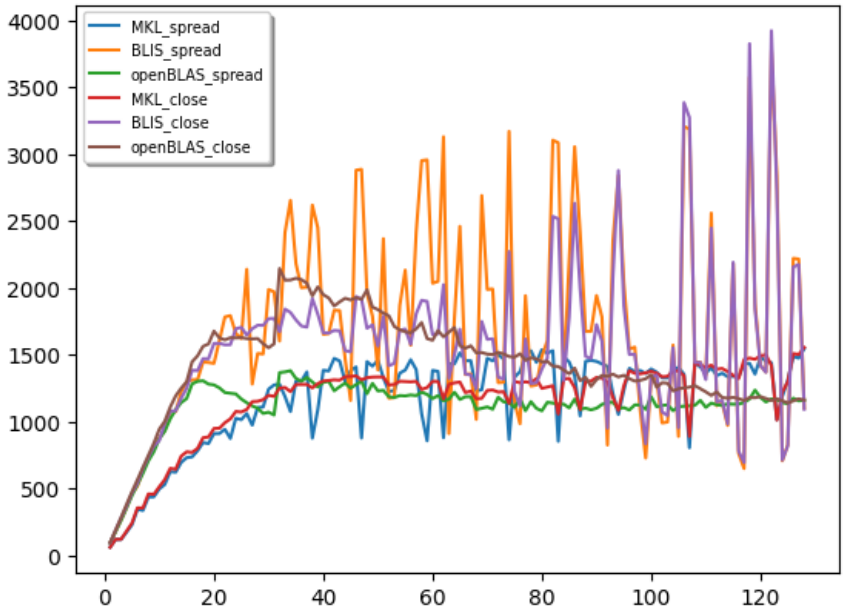

Figure 11: Varying parameter: number of cores, node partition: EPYC, precision: float, matrix size: 10000

THIN Differently from the EPYC case, from figures 13 and 14 we can see that on THIN nodes the measured performances are close to TPP until saturation of the available cores. This is somehow coherent with our previous explanation of the performance drop on EPYC, since here the number of cores is much smaller and therefore the matrix size is (apparently) big enough for the trade-off between workload distribution and cache coherence keeping to be optimal.

We can also observe that this time the worst performing library is BLIS and that, again, the choice of threads affinity policy does not really make any difference.

2.3.2 Size scalability

In this part we take into exam the behaviour with matrix size as the varying parameter. Ideally, the performance should always be equal to the theoretical peak one (which is constant, since we are always using the same number of cores).

EPYC As it is apparent from the graphs in figure 15 and 16, the libraries' performances are pretty poor w.r.t. to TPP. For small matrix sizes BLIS seems to be better performing than the others, but as the matrix size increases this gap disappears completely and the three libraries perform more or less equally.

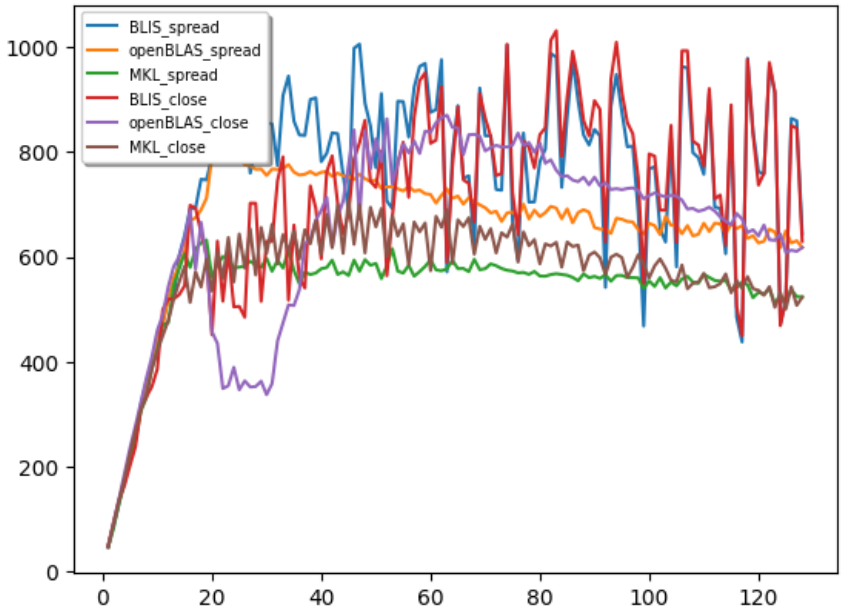

Figure 12: Varying parameter: number of cores, node partition: EPYC, precision: float, matrix size: 10000

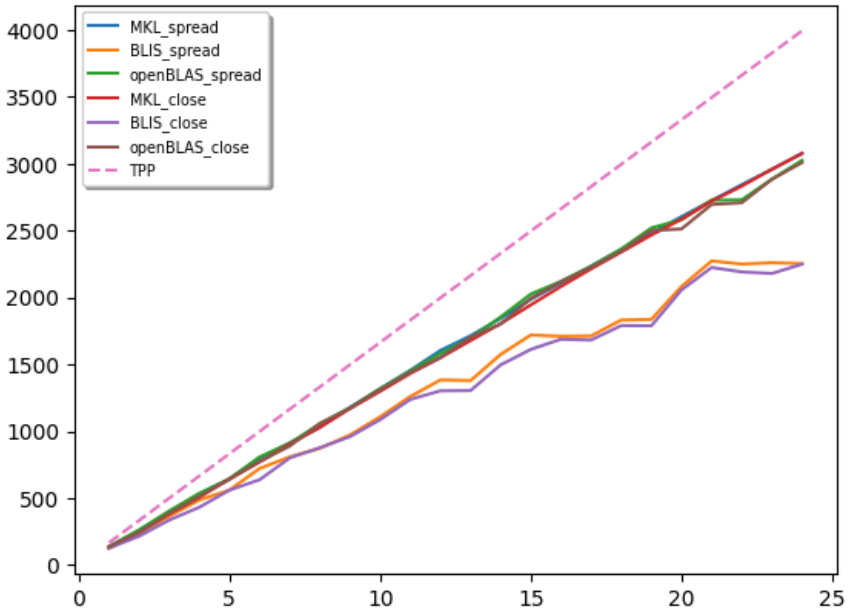

Figure 13: Varying parameter: number of cores, node partition: THIN, precision: float, matrix size: 10000

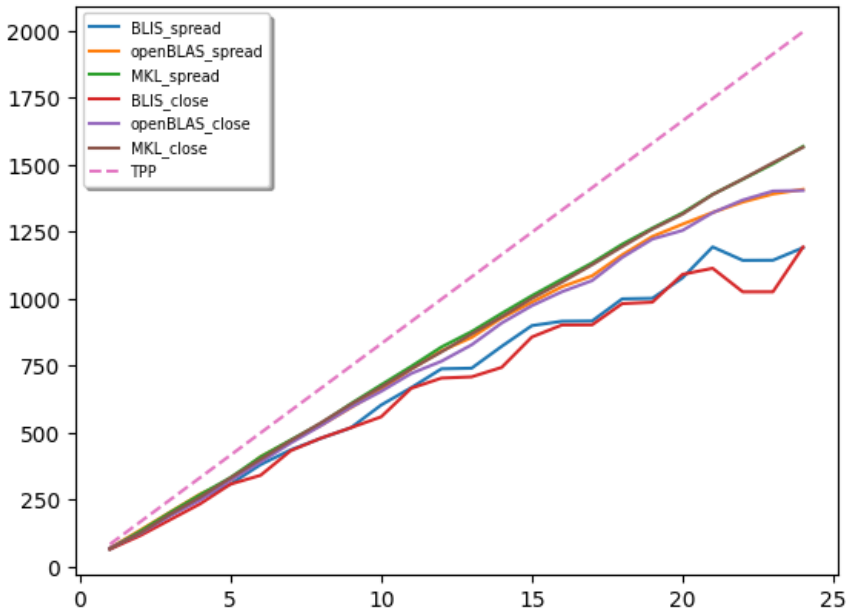

Figure 14: Varying parameter: number of cores, node partition: THIN, precision: double, matrix size: 10000

For what concerns the threads affinity policy, spread cores seems to be slightly better (at least for MKL and BLIS) but we cannot really state anything.

Besides libraries and threads affinity comparison, more in general we can observe that the performance increases quickly for small matrices size and then slows down. However, the general trend seems to be slightly increasing, suggesting that (probably) larger matrix sizes would give better performances. This is somehow coherent with our previous explanation of the bad performance on EPYC nodes in cores scaling: if 10000 was a too small matrix size for the workload division to be efficient when the number of cores was above 20, then we surely need a much larger matrix size for it to be efficient with 64 cores.

THIN In this case (see figures 17 and 18) measured performances are closer to TPP, in particular MKL is the best one with both threads affinity policies. Also, we can observe that the performance is for all libraries much more stable than it is on EPYC, and that the growing trend clearly stops more or less between 7500 and 10000, that being coherent with the results obtained in cores scalability: a matrix size of 10000 is sufficient to make the cache coherence keeping mechanism efficient on 20-25 cores.

Differently from the EPYC case, here we can state for sure that `spread cores` is better performing than `close cores`. This is particularly true for openBLAS, for which `spread cores`'s performance is almost three times better than `close cores`.

A Random numbers generation

In initialisation, for each cell of its own portion of grid each thread has to generate a random number among 0 and 1. This is achieved by simply generating a random floating

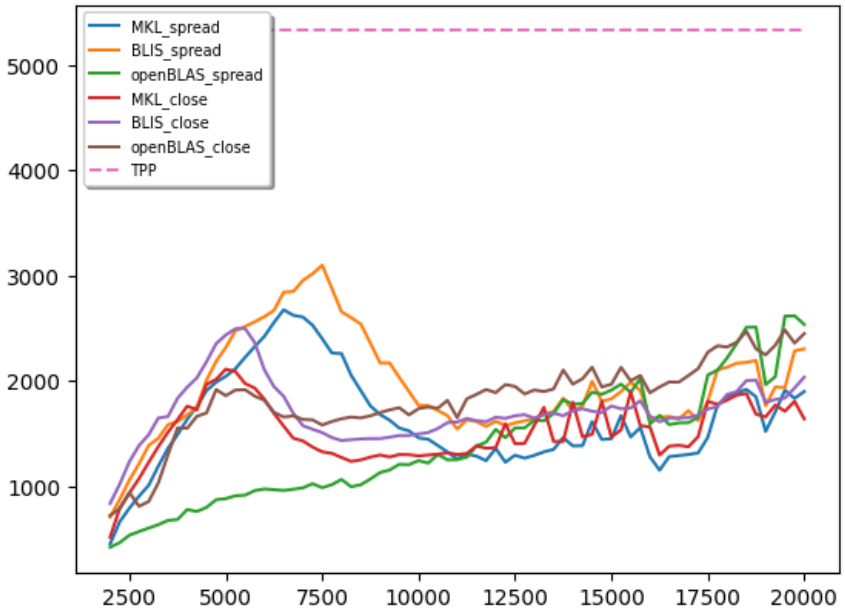

Figure 15: Varying parameter: matrix size, node partition: EPYC, precision: float, number of cores: 64

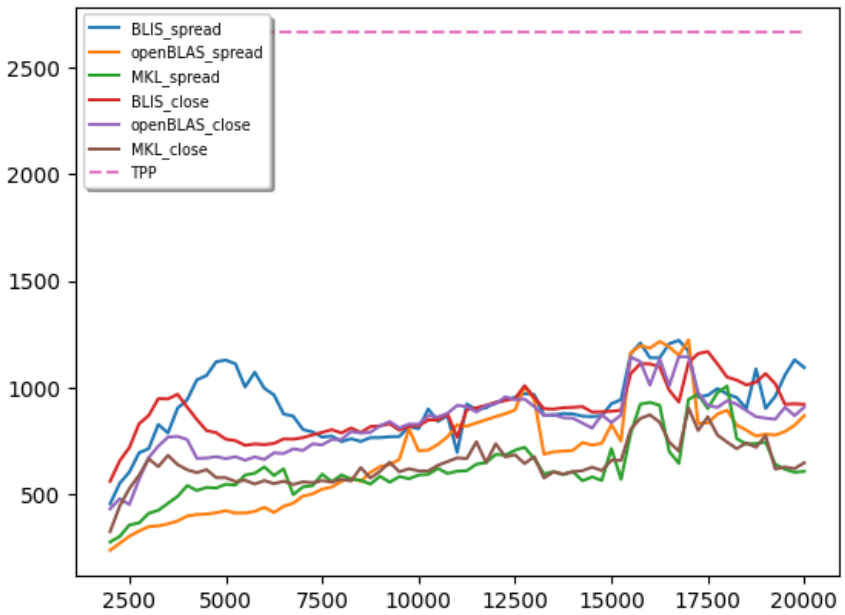

Figure 16: Varying parameter: matrix size, node partition: EPYC, precision: double, number of cores: 64

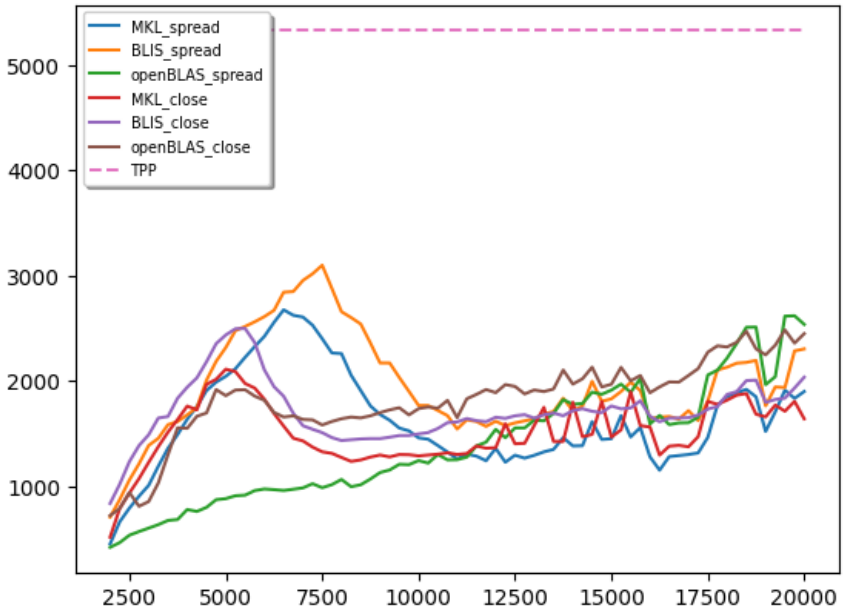

Figure 17: Varying parameter: matrix size, node partition: THIN, precision: float, number of cores: 12

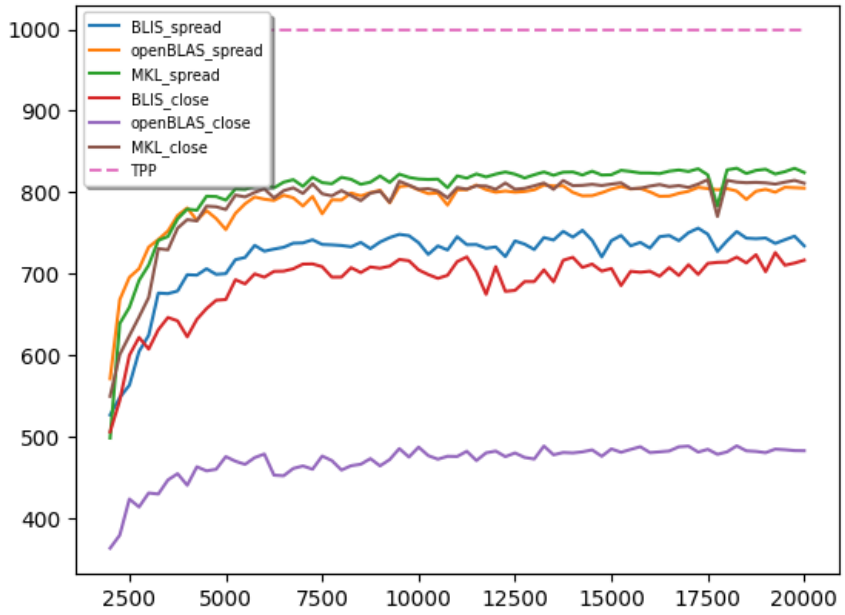

Figure 18: Varying parameter: matrix size, node partition: THIN, precision: double, number of cores: 12

point number between 0.5 and 1.5, using the function `drand48_r`, and typecasting it to `int`. The obtained integer is then converted to `BOOL`.

To be able to generate each time a different grid, we must set a different seed at each initialisation. To do that we use, as it is usually done, `time(NULL)`, which returns the time in second passed since a fixed date in the past. However, we also want to have a different seed for each thread of each process, since we would not like to have recurring patterns of `dead`/`alive` states in the grid.

To achieve that, we initialise the 0-th process's master thread's seed using `time(NULL)`. Then we broadcast it to all processes. At this point, each master thread sets its seed to the received one plus its process's rank multiplied by the number of threads on each process. Finally, each thread sums its id to its master's seed. Here is the complete chunk of code for random grid initialisation (`my_id` is the process's rank, while `my_thread_id` is the thread's id, and `my_grid` is the grid to initialise):

```
/* setting the seed */

struct drand48_data rand_gen;

/* generating a unique seed for each process */
long int seed;
if (my_id == 0)   // in this way we are sure that the seed
                  // will be different for all processes
    seed = time(NULL);

check += MPI_Bcast(&seed, 1, MPI_LONG, 0, MPI_COMM_WORLD);

seed += my_id*n_threads;

#pragma omp parallel
 {
    /* setting a different seed for each thread */
    int my_thread_id = omp_get_thread_num();
    srand48_r(seed+my_thread_id, &rand_gen);

   #pragma omp for schedule(static)
    for (int i=0; i<my_n_cells; i++) {

        /* producing a random integer among 0 and 1 */
        double random_number;
        drand48_r(&rand_gen, &random_number);
        short int rand_bool = (short int) (random_number+0.5);

        /* converting random number to BOOL */
        my_grid[i] = (BOOL) rand_bool;
    }
}
```

B PGM headers

To be able to write to a PGM file, we must first format it (i.e. write the header). In the same way, to be able to read it we need to first get the content of its header. For details on PGM files' header we invite you to read the already cited original pdf of the exercise (Tornatore Luca, 2022).

In our case both file formatting and header reading were done by the 0-th process. Here we just briefly expose how we did that.

For instance, the following is the piece of code used to format system snapshots (`check` is just an error checker for MPI I/O communications):

```
if (my_id == 0) {

    /* setting the header */
    char header[header_size];
    sprintf(header, "P5 %d %d\n%d\n", x_size, y_size, color_maxval);

    /* writing the header */
    access_mode = MPI_MODE_CREATE | MPI_MODE_WRONLY;
    check += MPI_File_open(MPI_COMM_SELF, snap_name, access_mode, MPI_INFO_NULL, &f_handle);
    check += MPI_File_write_at(f_handle, 0, header, header_size, MPI_CHAR, &status);
    check += MPI_File_close(&f_handle);
}
```

In it, `color_maxval` (the integer representing the darkest color in the PGM) was previously set to 1 (since we just need black and white) and `header_size` is set to the total length in bytes of the header, spaces and new-line characters included. The array `header` is a string storing the complete header to be written.

For what concerns reading, things are a little more complicated. To do that, we used the following function:

```
/* function to get content and length of the header of a pgm file (modified
 * version of prof. Tornatore's function to read from pgm) */
int read_pgm_header(unsigned int* head, const char* fname) {

    FILE* image_file;
    image_file = fopen(fname, "r");

    head[0] = head[1] = head[2] = head[3] = 0;

    char    MagicN[3];
    char   *line = NULL;
    size_t  k, n = 0;

    /* getting the Magic Number */
    k = fscanf(image_file, "%2s%*c", MagicN);
```

```
    /* skipping all the comments */
    k = getline(&line, &n, image_file);
    while (k > 0 && line[0]=='#')
        k = getline(&line, &n, image_file);

    /* getting the parameters */
    if (k > 0) {

        k = sscanf(line, "%d%*c%d%*c%d%*c", &head[1], &head[2], &head[0]);
        if (k < 3)
            fscanf(image_file, "%d%*c", &head[0]);

    } else {

        fclose(image_file);
        free(line);
        return 1;   /* error in reading the header */
    }

    /* getting header size */
    int size = 0;
    for (int i=0; i<3; i++) {
        int cipher = 9;
        int power = 10;
        size++;
        while (head[i] > cipher) {
            size++;
            cipher += 9*power;
            power *= 10;
        }
    }
    head[3] = 6 + size;

    fclose(image_file);
    free(line);
    return 0;
}
```

The passed pointer `fname` is a string storing the name of the file whose header is to be read, while `head` is an array to store the header's content, in particular its elements are (in order): `color_maxval`, `x_size`, `y_size` and `header_size` (which we remind being measured in bytes).

Once the function has been called by the 0-th process, the content of `head` is spread to all processes, and used to distribute the work and to write snapshot files.

References

Tornatore Luca. (2022). *Exercise 1 - parallel programming.* Retrieved from https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/blob/main/Assignment/exercise1/Assignment_exercise1.pdf

Wikipedia. (2001). *Conway's game of life — Wikipedia, the free encyclopedia.* Retrieved from https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Wikipedia. (2004). *Basic linear algebra subprograms — Wikipedia, the free encyclopedia.* Retrieved from https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms