Foundations of HPC
Final assignment

Sara Cocomello, Francesco Maria Speciale

September 2023

Contents

Exercise 1

1.1 Introduction

The purpose of the exercise is running a hybrid MPI + openMP parallel code which executes the evolution of a bidimensional grid of cells using Conway's game of life rules. Basically a cell is considered to be alive (white coloured) if among its neighbours there are 2 or 3 alive cells, or dead (black coloured) otherwise. By neighbours of a cell, we mean the cells located no more than one row and one column above or below the considered cell. The playground (i.e. the grid) was required to be capable to contain an unlimited number of cells (not less than 100 per dimension in any case) and to have periodic boundary conditions at the edges. This means that cells at an edge have to be considered neighbours of the cells at the opposite edge along the same axis.
Various evolutions arise from the same grid depending on the moments of the evaluation and the update of each cell. For this exercise, as requested we performed:

- ordered evolution: cells are evaluated and updated sequentially, one after the other. This means that at each evolution step all the cells but the first one have as neighbours some cell whose value has already been updated. So, the choice of the first cell to be evolved influences the evolution of the entire grid in cascade.
- static evolution: cell values are updated only after the new values for all the cells in the grid have been evaluated using last completed evolution status. Therefore, the order in which the cells are evaluated does not matter.

It was also requested to test three different performance scalability aspects.

- OpenMP scalability: setting one MPI process per socket, evaluate performance increasing the number of OpenMP threads per process from 1 up to saturation.
- Strong MPI scalabaility: keeping grid size fixed, test runtime behaviour increasing the number of MPI processes.

- Weak MPI scalability: given an initial size, show the run-time behaviour scaling up from 1 socket (saturated with OpenMP threads) up to as many sockets as possible keeping fixed the workload per MPI task (so, increasing grid size proportionally).

1.2 Methodology

The grid was represented as a PGM file of black and white pixels. Black pixels are intepreted as dead cells and white pixels as alive cells. The initialization of the matrix is pseudo-casual. Reading and writing the grid is parallelised by using independent MPI processes which read and write the file at different byte offsets.
Static evolution of the grid is parallelised too. The grid is split in a number of blocks of adjacent rows equal to the number of MPI processes and for each block a process is dedicated. The association between each process and its block is determined by the rank of the process (process 0 takes the first block, process 1 the second one, etc.). No blocks can contain one row more than any other block. We made this choice for domain decomposition because we thought it was a good compromise between simplicity and load balance. Moreover, since to evolve each cell requires the "neighbours" from its previous and next row, this method ensures that each MPI process communicates just with two other processes (the one which deals with the rows-block above and that which deals with the one below).
Other methods, like splitting the grid in bidimensional subgrids (like squares) or distributing cells to processes one by one would need size-dependent communication schemas. Moreover, this type of decomposition does not need message passing for neighbors that are on the same row and makes the problem of periodic conditions at the margin for the left and right sides of the grid easily manageable.
The evolution of the cells inside the blocks is handled differently according to the kind of evolution. For static evolution we have a further decomposition of the domain: every process spawns OpenMP threads and the update of the block rows is divided among. Each thread deals in parallel and independently with a different subset of the rows-block.
Ordered evolution, instead is inherently serial, because cells are evolved one by one and the result of their evolution is immediately used to determine the update of the cells that come immediately after in order. Therefore OpenMP threads are avoided in this case and the evolution of each block is completely serial. MPI processes are retained only to avoid the memory overload that might occur if a single process was deputized to evolve the entire grid. Actually their execution, as we will see in next section, is sequential, such that at each evolution step every process starts only when the previous one has ultimated the serial evolution of its cells and sent their updated value as a message to the processes that need the new values.
Finally, the program has the option to save periodically in a file the partial

evolution (or just the final one, at the end of course) of the grid every *period* evolution steps. This option is deactivated when we study the scalability because writing the file takes a significant fraction of runtime and it would affect an appropriate understanding of the performance.

1.3 Implementation

This section presents a detailed description of all the source files and the functions they contain inside. Code snippets are provided where they were considered helpful to better understand the logic of the program.

1.3.1 main.c

The main of the program catches the arguments inserted by the user by command line. Two separate flags are received as input to set grid initialization and perform the evolution. The available options are the grid size (horizontal and vertical can be set separately), the kind of evolution, the number of evolution steps, and the frequency interval, in evolution steps, with which to save the grid evolution. Depending on the flags and the evolution option, three different functions can be executed: `initialize_image` to create the grid, `evolve_static` and `evolve_ordered` for the two possible evolutions.

1.3.2 init.c

This file contains the two functions used for setting up the parallel environment and the domain decomposition with MPI.

- `initialize_procs`: creates the MPI communicator used in the program. In case the number of rows is less than the number of processes, it creates an MPI Group of processes starting from `MPI_COMM_WORLD` and includes in the group just the processes whose rank is less than the number of rows. Then, a new communicator from this group is created, processes size and ranks are reassigned and extra processes are set with `MPI_UNDEFINED` rank. In this way we work just with `rows` active MPI processes with rank from 0 to `rows` - 1. This is made to prevent any kind of wrong behaviour from extra processes, which would easily lead to segmentation faults or wrong message passing. We could also have excluded extra processes with a simple `if-else` construct at the top of most of the instructions blocks, but we discarded this option because it would have caused numerous branching, which would have slowed performance.
  In order to manage communication between processes elegantly and homogeneously, we used some smart MPI APIs to create a framework for a topological correspondence between the playground split in rows-blocks and the processes.
  `MPI_Cart_create` creates a communicator to which Cartesian topology information has been attached and `MPI_Cart_coords` endows processes with

"Cartesian coordinates" given their rank. Basically, each process is given a multi-dimensional index of integer numbers which maps the processes in a "multidimensional and bound Cartesian space" of processes. It is straightforward then thinking to build a "Cartesian communicator" which maps the rank of each process in bi-dimensional matrix of one column and *#rows-blocks* rows and to create a correspondence between processes and rows-blocks. Adjacent processes take care of adjacent row blocks in the playground and processes are mapped in the same order of the rows-blocks they operate on.
Furthermore `MPI_Cart_create` allows to set periodic boundary conditions at the edges and thanks to the API `MPI_Cart_shift` we identify for each process the ranks of the processes which are "shifted" by one row above or below its position in the "processes matrix" (i.e. the ranks of the processes that handle rows-blocks which are adjacent to its rows-block).
Thanks to these APIs we skip any control on the rank of the processes, which would have been necessary to handle correctly the difference between the communication among the processes dedicated to the inner blocks of the playground and those which deal with the edges (which haven't consecutive ranks). All processes recognize "autonomously" their neighbours.

- `set_parameters`: determines the subset size of the playground handled by each process and the byte offset after which the process must start reading and writing the pgm file (the end can be derived adding the size of the subset to the initial offset).

```
*rest=rows%procs;
*buffer_size=cols*(rows/procs+(rank < *rest));
unsigned int partial=*buffer_size*rank;
*disp=rank < *rest ? partial:partial+cols*(*rest);
```

The first `*rest` processes (ranked from 0 to `*rest` -1) handle `1+rows/procs` rows, whereas the following just one less. So, `*buffer_size` is initialized with the corresponding number of cells (computed just doing the product of the rows with the columns). This will be used to determine the size of the dynamic memory to allocate for reading the grid. Byte offset for file reading and writing follows directly. If the process is one the first `*rest` we are sure that all the processes that read before it read its same number of cells (`*buffer_size`). Otherwise, if the process `*buffer_size` is `cols` smaller (because `rank` $\geq$ `*rest`) we need to add to `cols *rest` times to `*buffer_size` because all the processes with bigger load read previous parts of the file. Ternary operator is used for these instructions to avoid branching.

1.3.3 read_write.c

This file contains all the functions used to read and write the pgm file, initialize the playground and save runtimes in a csv file.

- `write_header`: writes the header of the pgm file. This function is executed just by the process of rank 0.
- `read_header`: reads the header of the pgm file to retrieve matrix dimensions and the size of the header, which is needed by all the processes for the offset after which starting reading/writing.
- `write_image_pgm`: writes in parallel the pgm file. It uses MPI APIs `MPI_File_open`, `MPI_File_set_view` and `MPI_File_write` to open, set the bytes offset in the file for each process, and write the buffer of cells in the file. Cell values are written as `unsigned char` (1 byte) to minimize memory occupation. Buffer size and offset are set by calling function `set_parameters` (see1.3.2).
- `read_pgm_image`: reads in parallel the pgm file. It opens the file and sets the offsets for each process exactly as in `write_image_pgm` and executes reading with `MPI_File_read`. Of course also here cell values are treated as `unsigned char`.
  The dynamic memory allocated for the read buffer, actually, is larger than the number of read cells, and the difference amounts to the double of the number of columns of the grid. This extra space is necessary to host the values of the neighbour cells required for the evolution which are read by the "upper" and "lower" processes. The fact that these values are always equal to the number of columns is ensured by the "contiguity" of the "illusory" grid margins and the choice of the domain decomposition. The only cells which need their neighbours to be communicated by other processes are those on the first row and the last row of each node: half of their neighbours lie on the closest inner row, and half on the closest outer row.
  Thus, in order to arrange the best placement for the neighbours in terms of performance and code readability, read cells are saved in the memory buffer with a displacement of `columns` bytes from the first slot referenced by the pointer to the buffer.
- `initialize_image`: it's the function that is called when the program is asked to create the playground. It initializes an MPI parallel region and calls `initialize_procs` to create the cartesian communicator. Then, each process fills randomly by means of an OpenMP parallel `for` loop a dynamic memory buffer whose size has been decided with the same instructions performed in `set_parameters` (see 1.3.2 for both functions). Threads write on different slots of the shared memory, so filling the buffer in parallel represents a really convenient choice. The memory buffer must contain the cell values that will be written in the block of the playground assigned

to each process. 0 and 255 are the minimum and maximum numerical values that can be represented by means of an `unsigned char` and they represent black and white cells respectively. Then the playground can be considered as a matrix of 0 and 255 values. Finally, once the loop ha finished, `write_file_pgm` is called by each process to save the cells in the file and then memory for it is freed.

- `write csv`: writes number of processes, threads, grid size, number of evolutions and runtime on a csv file.

1.3.4 static.c

Static evolution is handled in this file.

- `evolve_static`: evolves the playground in a static mode with a parallel approach.
  First of all, it reads the header of the pgm file where the playground to be evolved is saved. This is done to get the playground size which is fundamental for the domain decomposition.
  Then, a hybrid MPI+OpenMP parallel region is initiated and the cartesian communicator is returned by `initialize_procs` (see 1.3.2). All the processes read a piece of the image with the correct displacement by executing `read_pgm_image` (see 1.3.3) and dynamic memory is allocated to host the evolved version of the loaded subset of the grid, reserving 2*`columns` extra bytes for the same reasons explained in `read_pgm_image`. Thus, evolution is not in place.
  At this point two different behaviours are performed depending on the number of the processes. If we have a single to evolve the whole matrix, no message passing is needed but a for each evolution just an OpenMP parallel `for` loop is executed to fill the extra-memory slots dedicated to the neighbors. Since we have just a single process, we really don't need this operation in this case, because the neighbours are already in the memory buffer of the process. However, we think it is convenient as it allows us to use a function for finding the neighbours (see `compute_neighbors` below) which is completely indifferent to the number of processes and the position of the cell, and it is performed without any branching operation. The second parallel loop executes `compute_neighbors` on all the cells of the buffer. This function returns the updated value for each cell, which is assigned to the correspondent memory slot referenced by the pointer to the evolved copy of the playground. Pay attention that the cells reserved to receive the neighbors are not evolved.
  Schedule for this loops is static because the workload is exactly the same for each iteration (transfer the neighbours, which are always the same amount, and update every cell, without any branch). Again, paralleling is completely safe because each thread writes on different bytes of the shared memory. At the end of each evolution step a check on the number of step

performed is done to evaluate whether it is time to save the partial evolution in a file and if this is the case `write_pgm_image` is called. Of course, evolved playground is saved without its first and last `columns` bytes.
Finally, before starting a new iteration, pointers to the previous playground (before the last evolution) and its evolved version are swapped. Dynamic memory referenced by this last pointer is going to be freed at the end of the loop. When evolution is finished, the runtime from the initialization of the MPI region is saved in a file. Time is computed by means of the API `MPI_Wtime`.

```
ptr=read_pgm_image(&cart_comm,ptr,filename,
    initial_offset,maxval,&rows, &cols,&rest,&
    buffer_size,procs,cart_rank);

unsigned char * evo = malloc(2*cols+buffer_size);

for(int i=0; i < evolutions; i++){
    #pragma omp parallel for schedule(static)
    for (int k=0; k<cols; k++){
        ptr[k]=ptr[buffer_size+k];
        ptr[cols+buffer_size+k]=ptr[cols+k];
      }

    #pragma omp parallel for schedule(static)
    for(int j=cols; j < cols+buffer_size; j++)
        evo[j]=compute_neighbor(j,ptr,cols,
        *maxval);

    if(steps > 0)
        if (i%steps==0){
            sprintf(evo_title,
            "./evos/evo_%d_of_%u.pgm", i+1,
               evolutions);
            write_pgm_image(&cart_comm,evo_title,
               &(evo[cols]), procs, cart_rank,
               rows, cols,*maxval);
          }
        unsigned char * temp = evo;
        evo = ptr;
        ptr = temp;
    }

free(evo);
global=MPI_Wtime()-t_start;
```

A similar process is defined if there is more than a process. The only difference is that the parallel loop which sets the upper neighbours for

the first row and the lower neighbours for the last one is substituted by message passing among every process and the one which the cartesian communicator has mapped immediately above and below it.
Every process executes two non-blocking sends of the first and last row of its block and two blocking receives of the outer neighbours of the cells lying on the first and last row of its block.
We recall that by using cartesian communicator with periodic boundaries we can skip any control on row indexes (see 1.3.2. Neighbours are saved, again in the first and last `columns` positions referenced by the pointer to the playground. The rest of the instructions for the evolution does not change. It's important to understand that, in this case, message passing is necessary, whereas the parallel for loop for the single process is just an implementation choice among the possible ones.
Finally, since in this case we have multiple processes, the longest one is considered for the computation of the runtime. `MPI_Reduce` gathers all runtimes from running processes, selects the maximum and returns its value to process 0, which is in charge of recording it on the csv file.

```
for(int i=0; i < evolutions; i++){
    MPI_Isend(&(ptr[cols]), cols,
       MPI_UNSIGNED_CHAR, proc_above, 1, cart_comm
       , &req);
    MPI_Isend(&(ptr[buffer_size]), cols,
       MPI_UNSIGNED_CHAR, proc_below, 0, cart_comm
       , &req);
    MPI_Recv(&(ptr[cols + buffer_size]), cols,
       MPI_UNSIGNED_CHAR, proc_below, 1 ,
       cart_comm, &status);
    MPI_Recv(ptr, cols, MPI_UNSIGNED_CHAR,
       proc_above, 0 , cart_comm, &status);

    #pragma omp parallel for schedule(static)
    for(int j=cols; j < cols+buffer_size; j++)
        evo[j]=compute_neighbor(j, ptr, cols, *
           maxval);

    if(steps>0)
        if(i%steps==0){
            sprintf(evo_title, "./evos/evo_%d_of_%
               u.pgm", i+1, evolutions);
            write_pgm_image(&cart_comm,evo_title,
               &(evo[cols]), procs, cart_rank,
               rows, cols, *maxval);
          }
    unsigned char * temp = evo;
    evo = ptr;
```

```
    ptr = temp;
    }
    free(evo);
    double duration=MPI_Wtime()-t_start;
    MPI_Reduce(&duration, &global, 1, MPI_DOUBLE,
       MPI_MAX, 0, cart_comm);
}
```

- `compute_neighbors`: it is the routine called to evaluate the neighbors and the evolved value for each cell. It just sums up the values of the neighbours and returns 255 if the sum of the neighbours is between $255*2$ and $255*3$, meaning that there are just two or three neighbors alive, otherwise 0. The position of the neighbors is determined computing a byte shift with respect to the position of the cell to be evolved and accessing the corresponding positions in the memory buffer.
  In order to understand how the function works, consider that each memory buffer is be indexed from 0 up to `buffer_size + 2 columns -1`. The first `columns` positions contain the neighbours coming from the process above in the cartesian map of the processes, and the last `columns` positions contain the neighbours coming from the process below. So, the values of the indexes of the cells to evolve for each buffer go from `columns` to `buffer_size+columns`.
  We can access neighbours from the row above or the row below on the same column of the current cell by adding or substracting a shift of `columns` positions (or bytes, since `unsigned char` size is just 1 byte) to the current position. Also, we can move to the right or to the left and add this horizontal shift to the vertical simply by adding or subctracting one. The only exceptions occur when we want to move from the first column to the left or from the last column case to the right. Given the periodic boundaries, in the former case we need to add `columns-1` (left neighbour of a cell on the first column is the last cell on the same row) and `-columns+1` in the latter (right neighbour of a cell on the last column is the first cell on the same row). Again, we used ternary operator to deal with this cases.

```
unsigned char compute_neighbor(unsigned int
    matrix_pos, unsigned char* this_batch, unsigned
     int cols, unsigned int maxval){

    unsigned int left = matrix_pos % cols>0?
    -1 : cols -1;
    unsigned int right = matrix_pos%cols < cols-1
       ?
    1:-cols+1;

    unsigned int sum =
    this_batch[matrix_pos-cols+left]+
```

```
    this_batch[matrix_pos-cols] +
    this_batch[matrix_pos-cols+right]+
    this_batch[matrix_pos+left]+
    this_batch[matrix_pos+right]+
    this_batch[matrix_pos+cols+left]+
    this_batch[matrix_pos+cols]+
    this_batch[matrix_pos+cols+right];
    return sum >= maxval*2 && sum <= maxval * 3?
    (unsigned char) maxval : (unsigned char)0;
}
```

1.3.5 ordered.c

This file manages ordered evolution and contains just the single function `evolve_ordered`, which calls `compute_neighbors` from `static.c`. The grid is evolved from the highest to lowest row going from the leftmost column to the rightmost one.

- `evolve_ordered`: it updates the playground using ordered evolution criteria. Even though it is inherently serial, we decided to implement it using MPI to prevent memory overloads. Nevertheless, there's noTHINg parallel and each processes starts evolving the playground only when the previous has finished its part. OpenMP threads are completely absent because of the nature of the problem. We need to wait for the completion of the evolution of each single cell before moving to the next one. On the other hand we implemented this evolution in place.
  As for static evolution, we distinguished the case with single process from the case with multiple processes.
  After the usual reading of the header, initialization of the parallel region, and reading of the image, if there is just a process, we need to initialize the first and last `columns` slots of the memory buffer with the elements from the last and first row of the playground respectively, as it happens for one process in `evolve_static` (see 1.3.4 for more details). This time we need to do it with two separate serial loops, because of the ordered nature of the evolution (we can pass each cell only after it has evolved). After this step, `compute_neighbors` is called, partial evolutions can be saved and runtimes are recorded as in the static evolution.

```
ptr = read_pgm_image(&cart_comm, ptr, filename,
    initial_offset, maxval, &rows, &cols, &rest,
     &buffer_size, procs, cart_rank);

for(int i=0; i < evolutions; i++){
    for(int j=cols; j < cols+buffer_size; j++){
        if(j==cols){
            for (int k=0; k<cols; k++){
                ptr[k]=ptr[buffer_size+k];
```

```
            }
        }
        if(j==buffer_size){
            for (int k=0; k<cols; k++){
                ptr[cols+buffer_size+k]=
                ptr[cols+k];
            }
        }
        ptr[j]=compute_neighbor(j, ptr, cols, *
            maxval);
    }
```

Things are trickier when dealing with more than a process. We need a loop across the evolutions which lets processes work in a queue. A new evolution step for all the processes can start only if each of them has received the cells updated by the process below in the previous evolution step. Furthermore, each process can start only if it has received the values of the neighbors from the process above it, which have been updated before but in the same evolution step.
There are two exceptions to manage: the first process at the first evolution step must enter without waiting for any message and the last process needs also the message from the first process in order to start its evolution (even though the first process is "below" the last one, the values of its cells have already been updated at the beginning of the evolution step). The implementation of these rules can be found in the following snippet.

```
MPI_Isend(&(ptr[cols]), cols, MPI_UNSIGNED_CHAR,
    proc_above, 1, cart_comm, &req);
MPI_Recv(&(ptr[cols + buffer_size]), cols,
    MPI_UNSIGNED_CHAR, proc_below, 1, cart_comm,
     &status);
MPI_Isend(&(ptr[buffer_size]), cols,
    MPI_UNSIGNED_CHAR, proc_below, 0, cart_comm,
     &req);
MPI_Recv(ptr, cols, MPI_UNSIGNED_CHAR, proc_above,
     0, cart_comm, &status);

for(int i=0; i < evolutions; i++){

    if(i!=0 || cart_rank!=0)
        MPI_Recv(ptr, cols, MPI_UNSIGNED_CHAR,
            proc_above, 0 , cart_comm, &status);

    if(cart_rank==procs-1)
        MPI_Recv(&(ptr[cols+buffer_size]), cols,
            MPI_UNSIGNED_CHAR, proc_below, 1,
            cart_comm, &status);
```

```
    for(int j=cols; j < cols+buffer_size; j++)
        ptr[j]=compute_neighbor(j, ptr, cols, *
            maxval);

    if(i < evolutions -1 || cart_rank == 0)
        MPI_Isend(&(ptr[cols]), cols,
            MPI_UNSIGNED_CHAR, proc_above, 1,
            cart_comm, &req);

    if(i < evolutions - 1 || cart_rank < procs -1)
        MPI_Send(&(ptr[buffer_size]), cols,
            MPI_UNSIGNED_CHAR, proc_below, 0,
            cart_comm);

    if(cart_rank!=procs-1 && i < evolutions -1)
        MPI_Recv(&(ptr[cols+buffer_size]), cols,
            MPI_UNSIGNED_CHAR, proc_below, 1,
            cart_comm, &status);

    if(steps > 0)
        if(i%steps==0){
            sprintf(evo_title, "./evos/evo_%d_of_%
                u.pgm\n", i+1, evolutions);
            write_pgm_image(&cart_comm,evo_title,
                &(ptr[cols]), procs, cart_rank,
                rows, cols, *maxval);
          }
      }
double duration=MPI_Wtime()-t_start;
MPI_Reduce(&duration,&global,1,MPI_DOUBLE,MPI_MAX
    ,0,cart_comm);
```

Memory buffer is prepared before starting the evolution to allow process 0 to work without messages during the first evolution. We didn't condition the send on the rank because we thought that branching could have made performance worse. Since processes with rank greater than 0 have to wait anyway for the previous processes to complete the evolution an extra message is redundant but should not affetc the performance. In the end, it is worth noticing that during the last evolution step we avoid sending the messages that should be received in the next one (otherwise they would remain pending). Saving the grid and recording the runtime works like for the static evolution.

1.4 Results

In this section the main results obtained from the scalability study on our implementation are presented. All the results are obtained from 4 repeated executions of the program. Mean and standard deviation are reported in the plots.

1.4.1 OpenMP scalability

OpenMP scaling was tested setting cores as `OMP_PLACES` and close thread affinity policy to optimize memory access. Processes are mapped by socket.
From Figure **??** we can compare the Speedup in the OpenMP scalability for a different number of sockets, on EPYC nodes. As the number of tasks increases, the Speedup derived by increasing the number of threads per task is further from the Theoretical Speedup. We may explain this behavior by considering that if we increase the number of processes, keeping the size of the problem fixed, each thread will work on a smaller number of cells. This means that a greater fraction of the cells handled by each thread is saved in a memory region which is shared with other threads and mapped in their cache. This causes false sharing and an overhead is introduced to keep all caches coherent. Moreover, since the buffer was allocated before the initialization of the OpenMP parallel region, threads are placed further and further from the closest memory banks of the master thread, where all the data for the single process are saved. So it's reasonable not to expect a perfect scaling. In the end, the reduction in time derived by parallelizing the tasks might be less significant with a great number of threads with small workload considering the overhead generated by the creation of the parallel environment.

On THIN nodes the Speedup is close to the theoretical scaling. A possible explanation could be that we are limiting the number of OpenMP threads to 12 per socket.

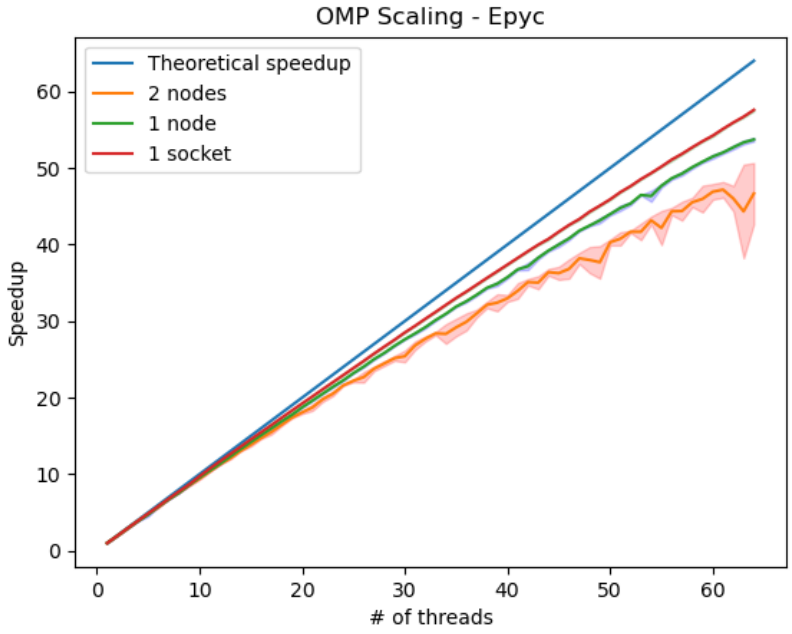

Figure 1.1: Comparison of the Speedup for 1 socket, 2 socket, 4 socket, on EPYC nodes, size of matrix: 25000x25000, evolution iterations: 50

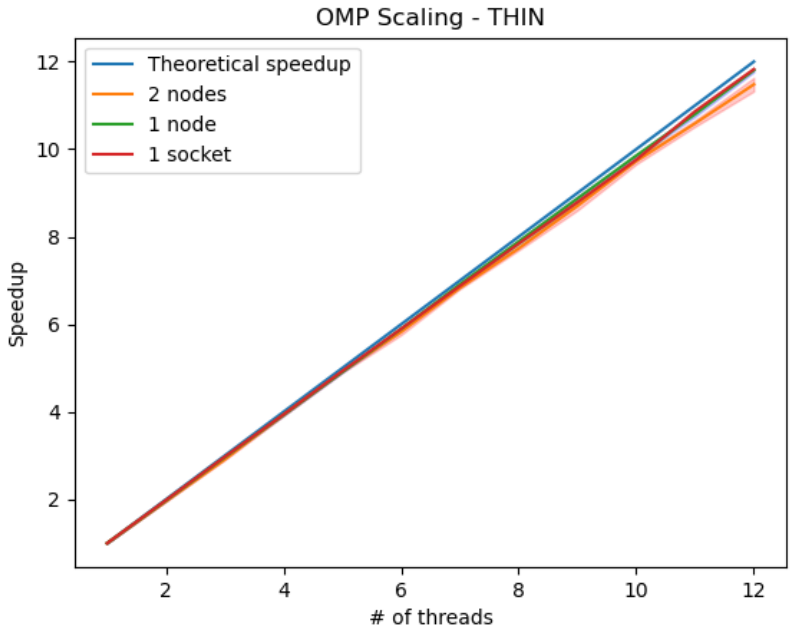

Figure 1.2: Comparison of the Speedup for 1 socket, 2 socket, 4 socket, size of matrix: 25000x25000, evolution iterations: 50

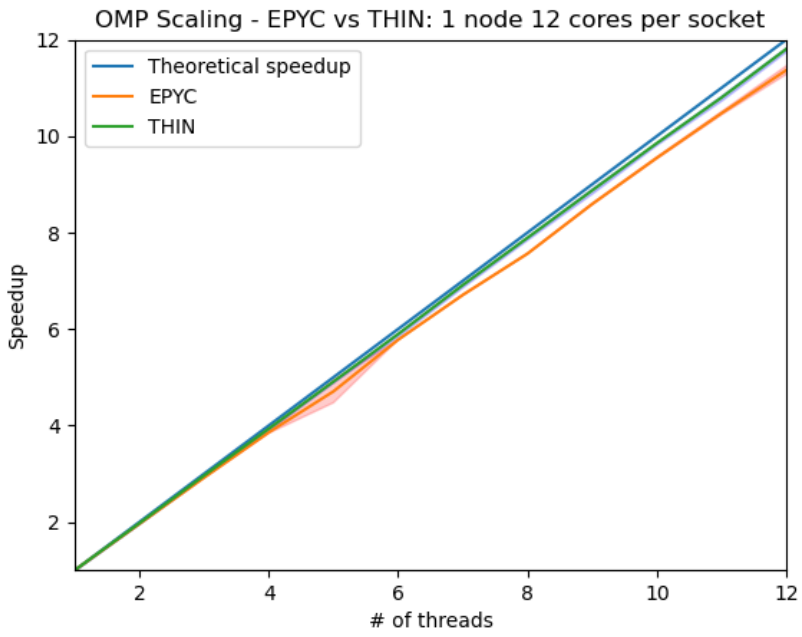

Figure 1.3: Speedup on EPYC nodes and THIN nodes, size of matrix: 25000x25000, evolution iterations: 50

Comparing the two partition with the same amount of resources, we can state that the Speedup on THIN nodes is slightly better that the one on EPYC nodes.

1.4.2 Strong scalability

We decided to test strong scaling with two different mapping approaches. In the first one, we mapped and bound processes by core. Therefore we didn't exploit OpenMP threading. We did this choice in order to evaluate the impact on performance of pure MPI parallelization and in particular of the overhead that is generated by message passing.
In Figure 1.4 and 1.5 we observe the results for three EPYC and THIN nodes, respectively. It is clear that the program performs much better, close to perfect scaling, on THIN nodes, with some oscillations for an increasing number of core.
On EPYC partition, instead, there is an evident downgrade in the speedup when a new node starts to be populated by MPI processes. Variability in the measures increases accordingly. We imagine this trend is related to some problems with communications between processes.

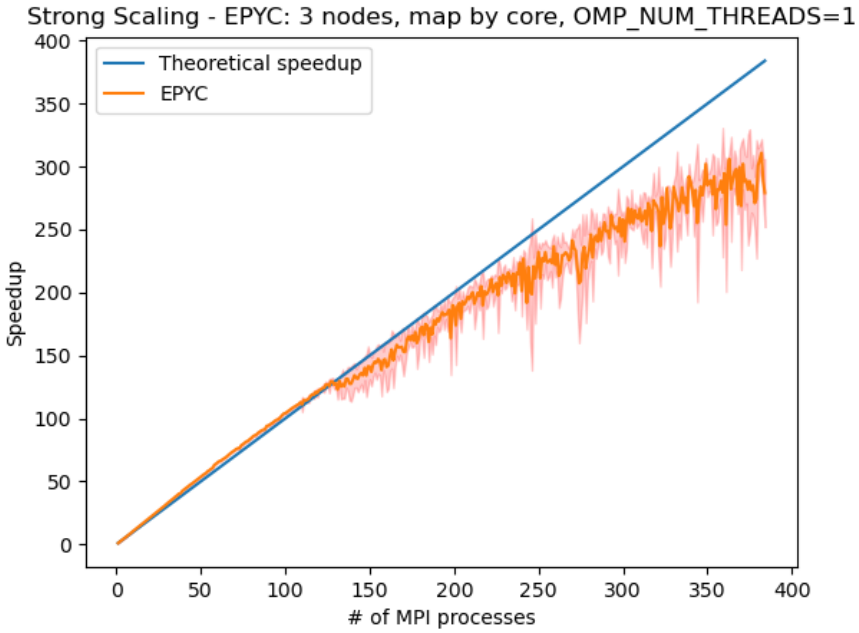

Figure 1.4: Speedup on EPYC nodes, size of matrix: 25000x25000, evolution iterations: 50

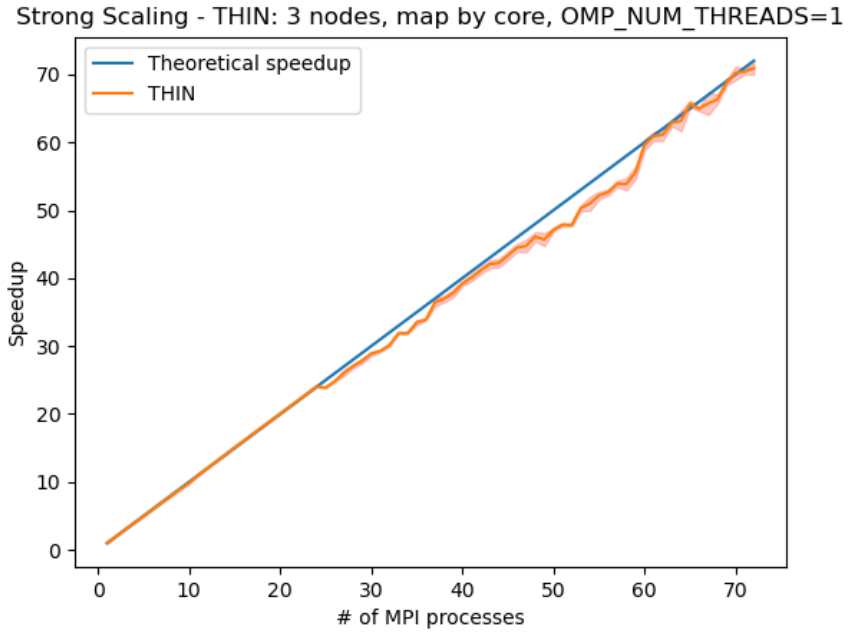

Figure 1.5: Speedup on THIN nodes, size of matrix: 25000x25000, evolution iterations: 50

The second strategy we tried is mapping processes by node and binding them to socket, considering then the impact of threads on the performance. Threads were set for both nodes at the maximum value available per socket and placed on cores with close affinity policy.
In Figure 1.6 we plotted the results for the strong scalability on 4 nodes on both EPYC and THIN partitions. The trend on THIN partition is closer to the theoretical speedup, while on EPYC partition the scalability is further from the ideal speedup and it looks more irregular. Since we have bound processes to sockets, that means that the number of OpenMP threads for each process on the two partitions is different. The number of OpenMP threads grows faster with the number of processes on EPYC partition, probably generating a higher rate of false sharing with respect to THIN partition. This, together with the communication issues we have already discussed above, may be one of the possible explanations for the different scaling performance.

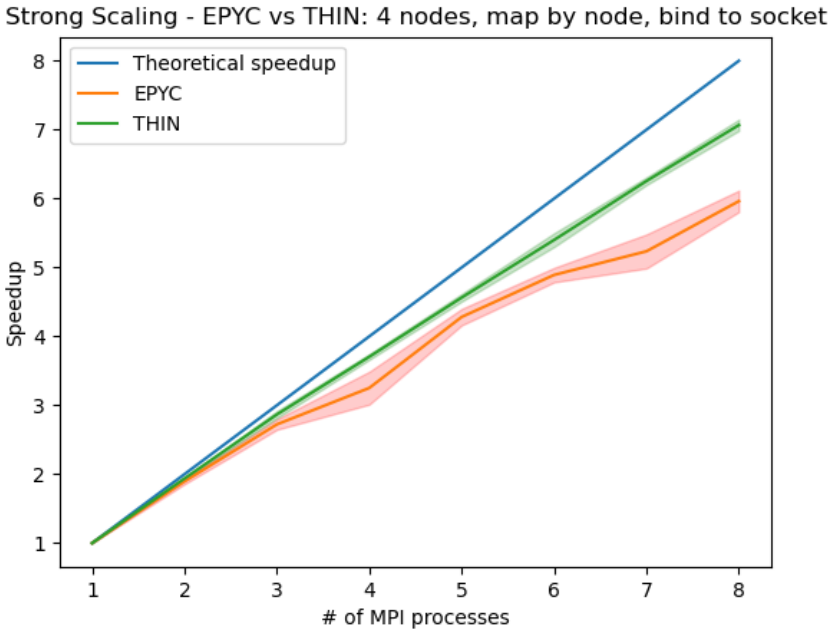

Figure 1.6: Speedup on EPYC nodes and THIN nodes, size of matrix: 25000x25000, evolution iterations: 50

1.4.3 Strong scalability for ordered evolution

For a matter of completeness we also tested MPI strong scaling on ordered evolution, mapping and binding processes by cores. The difference between the other sections is in the number of evolution steps that we decided to set to 5 instead of 50.
Although this kind of evolution cannot be parallelized, since we have implemented it in MPI to limit process memory usage (1.3.5), we still decided to test strong scaling to check if message passing introduces a consistent time overhead with respect to a inherently serial code (executed when process size is 1). Observing the results (Figure 1.7), we see that the execution time for EPYC stays approximately constant and actually it is slightly lower at least until 128 cores, probably because of good cache usage. Then runtime has more variability, maybe because the overhead introduced by communication between the cores of the first and the second node is less negligible with respect to the effective runtime of each process.
Regarding THIN nodes, it holds approximately the same. Again we observe at first a decrease in runtime and a slightly increase when the cores

occupy more than a single node.

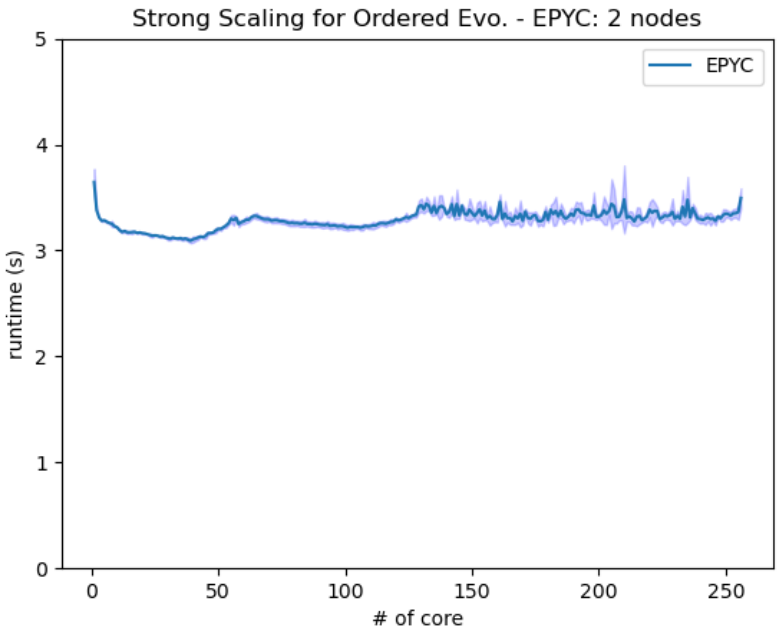

Figure 1.7: Runtime on EPYC nodes, size of matrix: 10000x10000, evolution iterations: 50

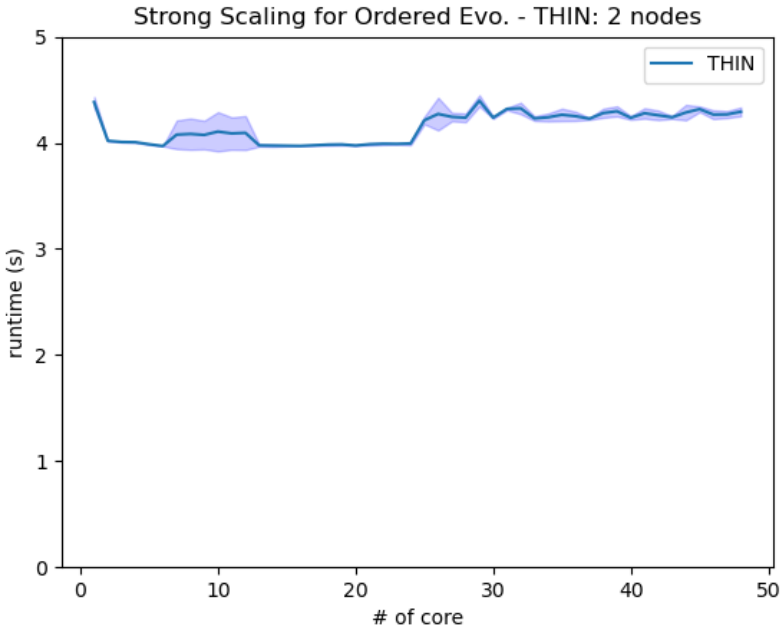

Figure 1.8: Runtime on THIN nodes, size of matrix: 15000x15000, evolution iterations: 50

1.5 Weak scalability

We tested weak scaling on 4 nodes on both EPYC and THIN partitions, mapping the processes by node and binding them to socket, in order to saturate each socket with OpenMP threads. To perform this analysis we kept the number of columns fixed and we increased the number of rows proportionally to the number of processes used to execute the program. Specifically we started from a 15000x15000 matrix up to 120000x120000. We recorded a certain variability in runtimes that was not theoretically expected and a general increase in runtime as the number of MPI processes grows. From Figure1.9 the comparison between the speedup on the two partitions it is evident that on THIN nodes the program scales better. The obtained results appear coherent with the hypotheses we made about strong scalability and OpenMP scalabilty (see 1.4.1 and 1.4.2): THIN nodes look like to manage MPI communication among processes better than EPYC ones and the smaller number of cores they can host probably causes less false sharing.

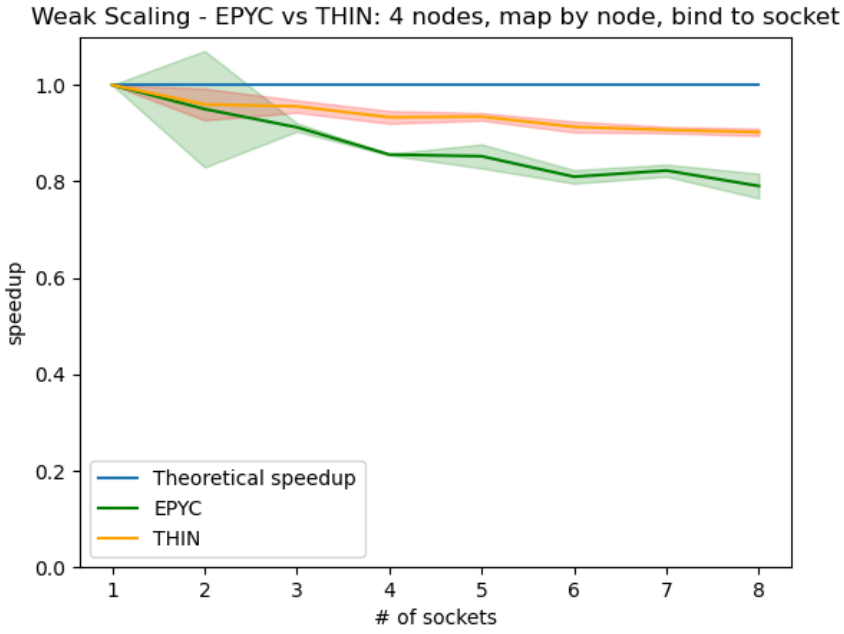

Figure 1.9: Speedup on 4 THIN nodes, EPYC vs THIN, starting size of matrix: 15000x15000, evolution iterations: 50

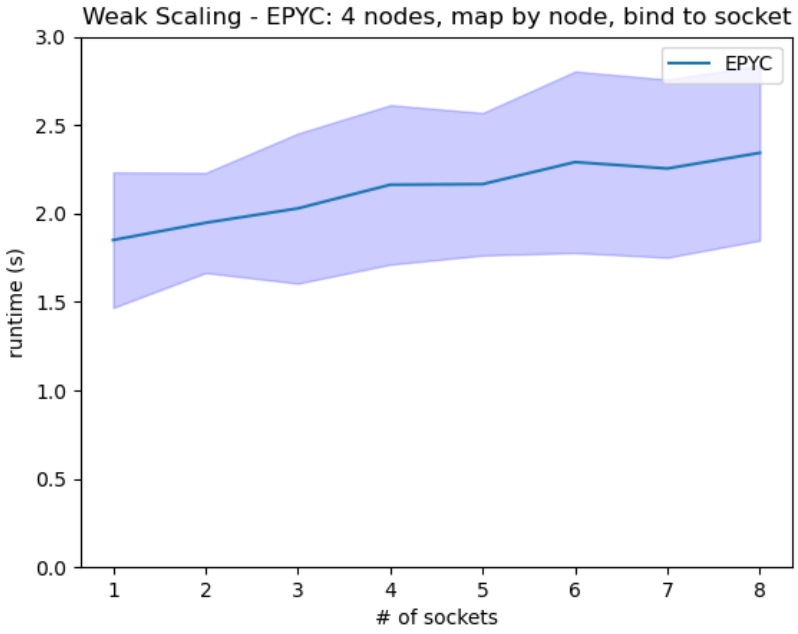

Figure 1.10: Speedup on 4 EPYC nodes, starting size of matrix: 15000x15000, evolution iterations: 50

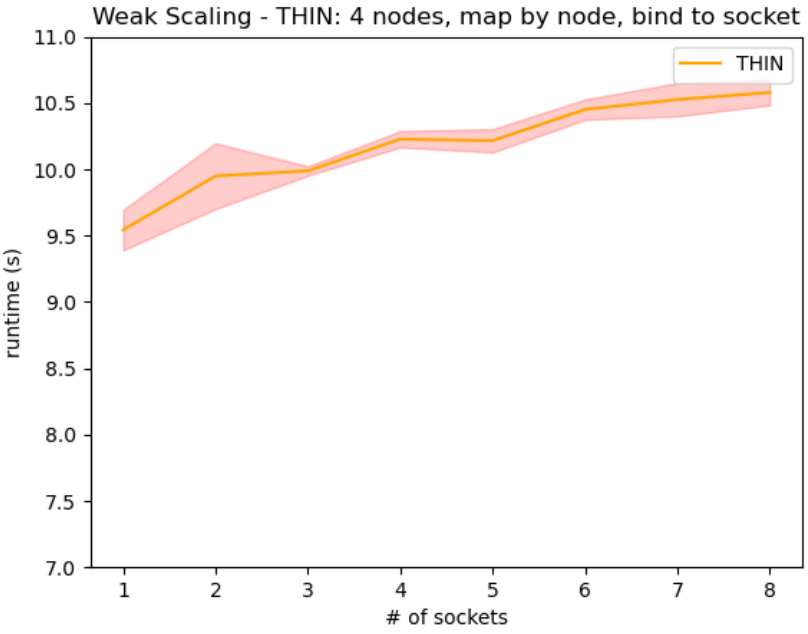

Figure 1.11: Speedup on 4 THIN nodes, size of matrix: 15000x15000, evolution iterations: 50

1.6 Conclusions

To conclude, we think that even if the theoretical peak performance was not reached in all the scalability studies with our implementation of Game of life, we largely analysed each situation in order to motivate the results. Some further improvement could be considered. The main pitfalls of our program may be found in the fact that the static evolution is not in place, moreover the workload is not perfectly balanced as we have processes which are in charge of an additional row. We would suggest to implement an in place version of it and it would be interesting to use a different domain decomposition maybe exploiting the cartesian communicator that we used. Moreover, the ordered evolution may be improved by reducing the message passing among different processes.

Exercise 2

2.1 Introduction

The goal of this exercise is to compare performance of three HPC math libraries: MKL, OpenBLAS and BLIS. The latter library had to be downloaded and compiled as optional phase.
The comparison is performed by focusing on the level 3 BLAS function called gemm, available both for double precision (dgemm), and single precision (sgemm).
In the code provided, "gemm.c", 3 matrices A,B,C are allocated, A and B are filled and the BLAS routine calculates the matrix-matrix product C. This example computes real matrix $C = \alpha \times A \times B + \beta \times C$ using BLAS function dgemm, where A, B, and C are matrices and alpha and beta are scalars. Specifically we set $\alpha = 1.0; \beta = 0.0$. Moreover we choose the matrices to be square and of the same dimension.
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. This kind of operations are said to be CPU bounded instead of memory bounded.

For what concern the Blis library, it was compiled both on EPYC and on THIN nodes, thus the two libraries are saved in different directory that are specified in each sbatch file.

2.2 Methodology

To compare the three HPC libraries we run the gemm function on both EPYC and THIN nodes. First of all we can distinguish two scenarios.

In the first one, in order to measure the scalability over the size of the matrix, We fix the number of cores and we let the dimension of the matrix

growing incrementally from 2000 to 200000.
The same analysis is repeated using both single and double precision. Another parameter to be set is the different threads allocation policy, allowing the cores to be allocated close or spread.

In the second scenario we instead measure scalability over the number of cores that run the OpenMP threads by fixing an intermediate size for the matrix (we fix the size to 11000). Again, we repeat the analysis using single and double precision for the gemm function and by setting different threads allocation policies.
The measurements of the performance is repeated under different settings and compared to the theoretical peak performance value.

Performance has been measured in both elapsed time and the number of single or double precision floating point operations per second, measured in GFLOPS, but only the latter has been actually used to perform the comparisons.

The collected data were then analysed in Jupyter notebooks.
In order to have more reliable results and to reduce the noise, each execution was repeated 5 times with the same parameters set, then the mean are considered in the plots together with the standard deviation.

In the following paragraphs more details about each step are offered.

2.3 Implementation

Before presenting the main results, we describe how we performed the necessary measurements.

2.3.1 Makefile and gemm.c

We have slightly modified the gemm.c file and the Makefile that were already provided.

The gemm.c was modified in order to print the results in a CSV file, provided with the information about the parameters of the environment set at each execution through the test.sh file (section 2.3.2). In the Makefile, the option to compile both the executables for the single and the double precision were added. Moreover the path of the folder were to save the executables was passed to the Makefile through the test.sh file:

```
    ...
    folder_save=$(pwd)
    cd ../../..

    make float folder=$folder_save
    make double folder=$folder_save

    cd $folder_save
    ...
```

2.3.2 Slurm sbatch script test.sh

Moreover we added a sbatch script in order to compile the executable files and run them in the correct environment by specifying the precise parameters for each library and partition settings.

We implemented a practical way to run with multiple combinations, between the different architectures (EPYC and THIN), floating point operations (single or double precision), size of the data, number of threads, processor binding policies, choice of memory allocation.

First of all we can prepare the environment:

```
#!/bin/bash
#SBATCH --job-name="TEST"
#SBATCH --no-requeue
#SBATCH --nodes=1
#SBATCH --cpus-per-task=64
#SBATCH --time=02:00:00
#SBATCH --exclusive
#SBATCH --partition=EPYC
#SBATCH --error=job.%J.err
###SBATCH --output=job.%J.out
```

In particular, by specifying the number of nodes, the partition, the kind of allocation (exclusive), the number of cores. The output and error files are collected respectively in the files job.%J.err and job.%J.out, while the computation was redirected on the CSV files form inside the batch file.
The results were collected in CSV files (nameofthelibrary.csv / dnameofthelibrary.csv). By submitting the test.sh file it was possible to carry out the measurements for one library at a time using both single and double precision.
The test.sh file contains a general setting that can be modified to fix the desired paramenters.

To change the parameters we can set:

```
places = {socket | cores | threads} //(only
    cores was considered in this report)
partition = {EPYC | THIN}
allocation_policy = {close | spread}
library = set by uncommenting the
    corresponding lines
```

```
50  #echo -n "${cores},"  >> mkl.csv
51  #./gemm_mkl.x $size $size $size >> mkl.csv
52  #echo -n "${cores},"  >> oblas.csv
53  #./gemm_oblas.x $size $size $size >> oblas.csv
54  #echo -n "${cores},"  >> blis.csv
55  ./gemm_blis.x $size $size $size >> blis.csv
```

The PLACES and the BINDING policies are passed to the corresponding environmental variables:

```
34  export OMP_PLACES=$places
35  export OMP_PROC_BIND=$allocation_policy
```

For what concern the first scenario, the number of cores are modified in a for loop with index "cores" by setting:

```
43  export OMP_NUM_THREADS=$cores
44  export BLIS_NUM_THREADS=$cores
```

The files and their precise organization can be found at the GitHub repository, together with the data that we used to obtain the final results.

2.4 Theoretical peak performance

The last thing to tackle is the evaluation of the Theoretical peak performance. The single node theoretical peak performance measured in FLOP per second is:

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

If we consider the EPYC node partition, when we are referring to double precision, a single core can deliver up to 16 floating point operations per cycle, since we have **128** cores and the maximum frequency is 2.6 GHz, the theoretical peak performance is **5.3 TFLOPs**.

Therefore, for a single core, we have:

$$TPP_{double} = \frac{5.3}{128} = 41.6Gflops$$

If we consider the Theoretical peak performance for using single precision, we obtain the double of the performance, that is:

$$TPP_{float} = 83.2Gflops$$

For what concerns the THIN nodes, we can carry out the same considerations. A node has a Theoretical peak performance of 1.997 TFLOPS, in floating point precision, therefore for a single core:

$$TPP_{double} = \frac{1.997}{24} = 83.2Gflops, TPP_{float} = 166.4Gflops$$

2.5 Results

WiTHIN this section, we will delve into the analysis of our results, based on a graphical representation. Data are gathered by comparing the MKL, OpenBLAS and BLIS library with respect to different thread affinity policies . As we progress, we evaluate the performance across the diverse setups, while also assessing them against the theoretical peak performance.

2.5.1 Size scalability

In this section we analyse the results obtained by fixing the number of cores while the size of the matrices grows form 2000x2000 to 20000x20000 by steps of 1000.

On EPYC nodes

For the measurements on EPYC partition, we consider half of the available cores on one node and we set

```
OMP_NUM_THREADS = 64
OMP_PLACES = cores
```

and used as policies for threads binding:

```
OMP_PROC_BIND={close|spread}
```

From Figure 2.1 and Figure 2.2, we can notice that the performances are well below the Theoretical peak performance when using 64 cores. Among the three libraries Blis is the one which is pertorming better, for a smaller

size, while at an increasing size the difference between the libraries become less pronounced.

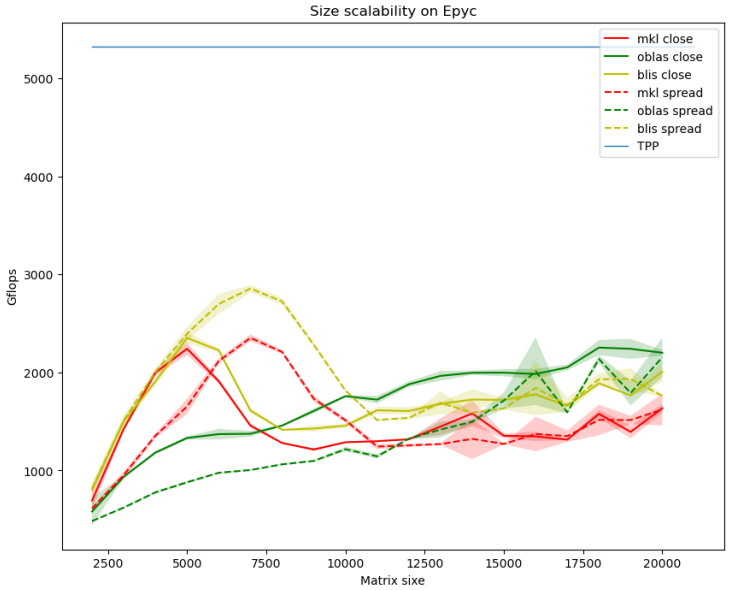

Figure 2.1: Matrix size scalability on EPYC node partition, number of cores: 64, precision: float

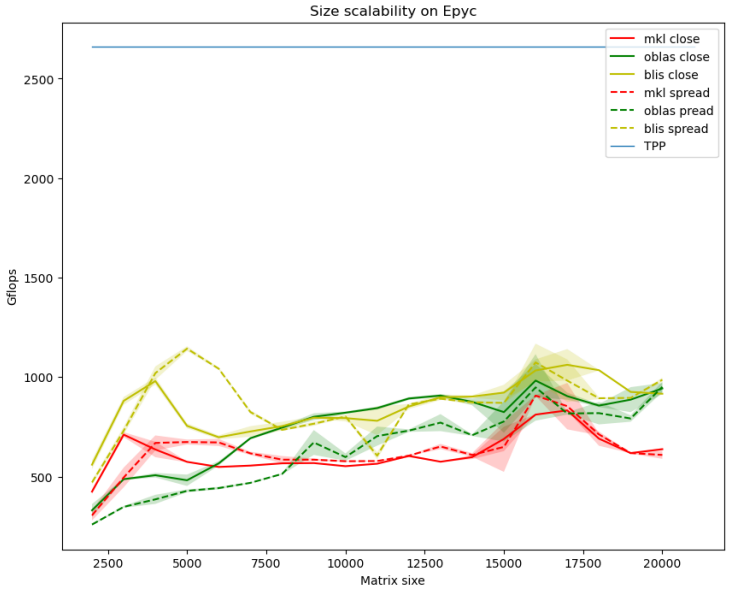

Figure 2.2: Matrix size scalability on EPYC node partition, number of cores: 64, precision: double

It is worth noticing that for both MKL and BLIS libraries, up to a certain matrix size, the trend of performance grows consistently. This is possibly because of an optimal use of cache, while the performances significantly fall until a corresponding size of 11000x11000, where the trend appears to reach an approximately steady growth that anyway does not reach the same performance of the first peak.

For what concerns the threads binding policies, it is not clear whether the allocation policy spread affects positively the performance, but we can say that for OBLAS library the close policy clearly achieves better results for smaller size values, wile for BLIS library it is the spread policy to perform better.
By considering the plot for double precision in Figure 2.2 we can observe a similar trend.

By comparing the three libraries using single precision, we can observe that the one that achieves the best performance is BLIS library, reaching a maximum value of 2890 Gflops (size = 7000) with spread cores. But moving our attention to bigger matrix sizes, OBLAS seems to better perform in terms of Gflops. MKL, instead, follows a trend similar to those of BLIS for both close and spread cores, but reaching smaller values (max Gflops = 2370, size = 7000, sparse cores). It is important to point out that, while the OBLAS library clearly outperforms the BLIS performance for

highest size values for single point precision, when using double precision BLIS library keep scoring highest values of Gflops (Figure 2.2) .

Numactl

As the memory is allocated by the first thread in gemm.c, we thought it would be interesting to change the memory placement. Down below we report the results obtained for the size scalability study when we change the memory placement policy with numactl function on EPYC node, using single precision (Figure 2.3, Figure 2.4). By specifying the interleave policy for numactl we modify the behavior of the operating system page allocation when a first touch on the page is done. With - interleave the memory pages are allocated across nodes specified by a nodeset. The option is passed as follows:

```
numactl --interleave=0,1,2,3 ./gemm_blis.x $size
    $size $size >> stat_blis.csv
```

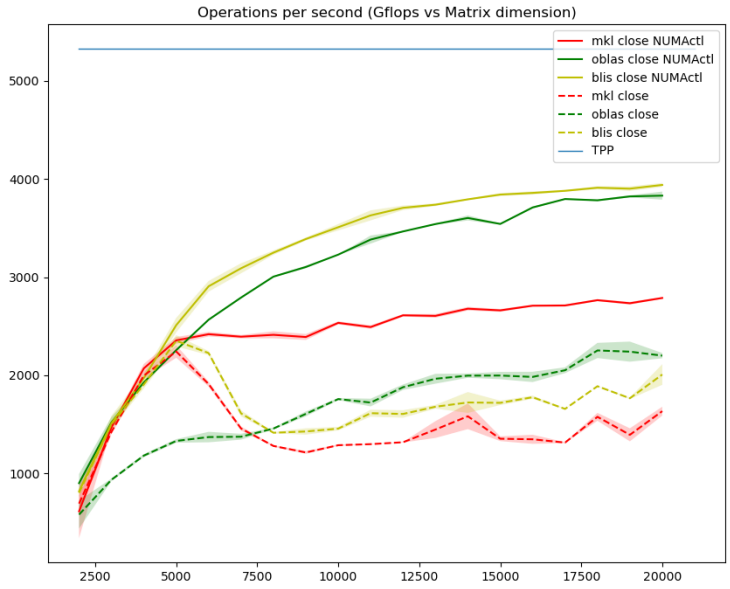

Figure 2.3: Matrix size scalability on EPYC node, number of cores: 64, precision: float, numactl – interleaved policy, close cores

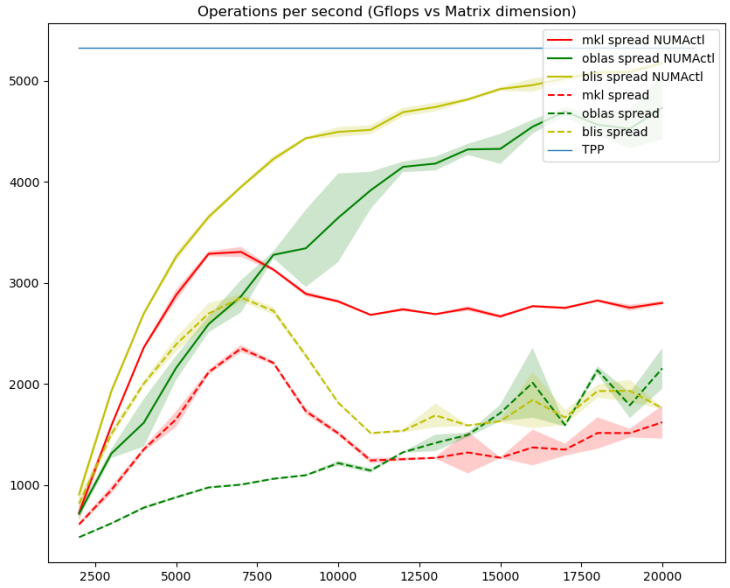

Figure 2.4: Matrix size scalability on EPYC node, number of cores: 64, precision: float, numactl – interleaved policy, spread cores

By using this option at run time we are asking for a specific allocation of the memory across the NUMA regions, that can be exploited by the threads allocation policy. It greatly increases the performance of the program, which it appears to be even close to the Theoretical peak performances for the spread allocation policy (Figure 2.4 .

Here we can easily see the difference between the two allocation policies close and spread in the performances.

On THIN nodes

We performed the same analysis on a THIN node, by using half of the available cores (12), in order to compare the two threads binding policies. By observing Figure 2.5 and Figure 2.6 we can immediately notice that the performances are closer to the Theoretical peak performances than in the previous section, and the trend looks more stable.

First of all we can see that the spread policy is strongly better than the close one for both single and double precision, as it is always scoring higher values for all libraries.
Comparing the libraries, MKL seems to perform better than the others

libraries overall, even if it reaches similar results than OBLAS library for single precision. BLIS library on THIN nodes is clearly performing worse than the others, but keeping a relatively small gap.

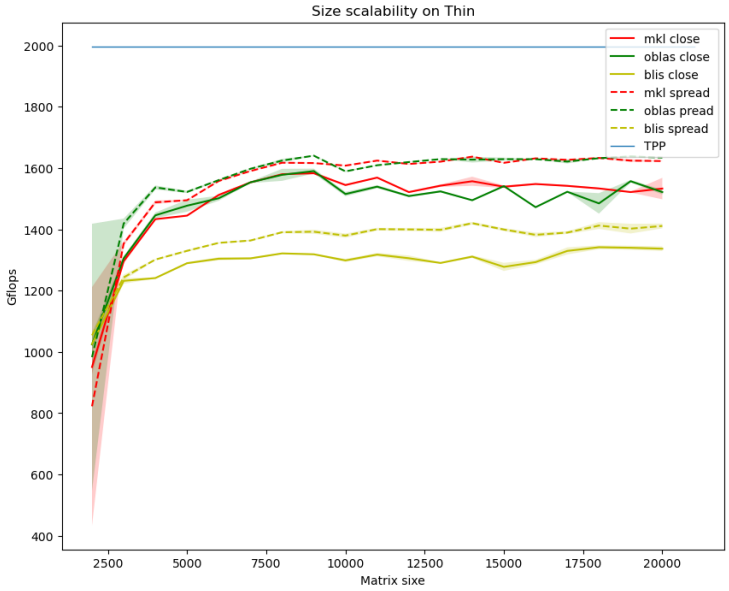

Figure 2.5: Matrix size scalability on THIN node partition, number of cores: 12, precision: float

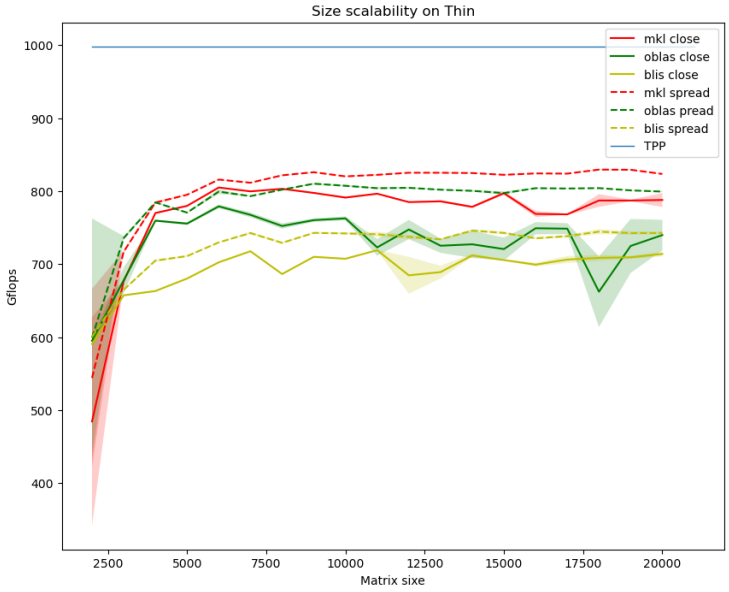

Figure 2.6: Matrix size scalability on THIN node partition, number of cores: 12, precision: double

2.5.2 Core scalability (strong scalability)

In this section we measure the scalability over the number of cores at fixed size, in our case we consider a matrix size of 11000x11000. Theoretically, we expect the performance to increase proportionally to the number of cores.

On EPYC nodes

From Figure 2.7 and Figure 2.8 we can notice that on EPYC node, for a small number of cores (less than 20), the performance of two of the three libraries, OBLAS and BLIS are very close to the peak performance that we would like to reach; while the MKL library has worst performance for a small number of cores. As the number of cores increases, the gap between the results and the TPP increases, while the difference between the three libraries decreases. We can notice that the trend of the BLIS library has a lot of variance compared to the trend of the other two.

An hypothesis to for the overall decrease in performances, on EPYC nodes, may be found by making some considerations on the memory allocation in the gemm.c program. Before computing the matrix product using the

gemm function (via CBLAS interface), memory is allocated before initiating the parallel region, that means it is contiguous.
As a consequence, it may happen that multiple threads work on data that reside in the same cache line. The data loaded in their caches also include data assigned to other threads, that means that more work has to be done to keep the cache accesses coherent. The effort to treat this issue is apparently more that the benefits of having a bigger number of threads working on the same data.

Moreover, the difference between the allocation policies close and spread seems not to affect the performances, and this is even more apparent for a large number of cores, indeed as the resources are saturated, the way we occupied them becomes irrelevant. The only library in which the policies differs in performance is OpenBlas for which in both single and double precision the close policy scores better results.

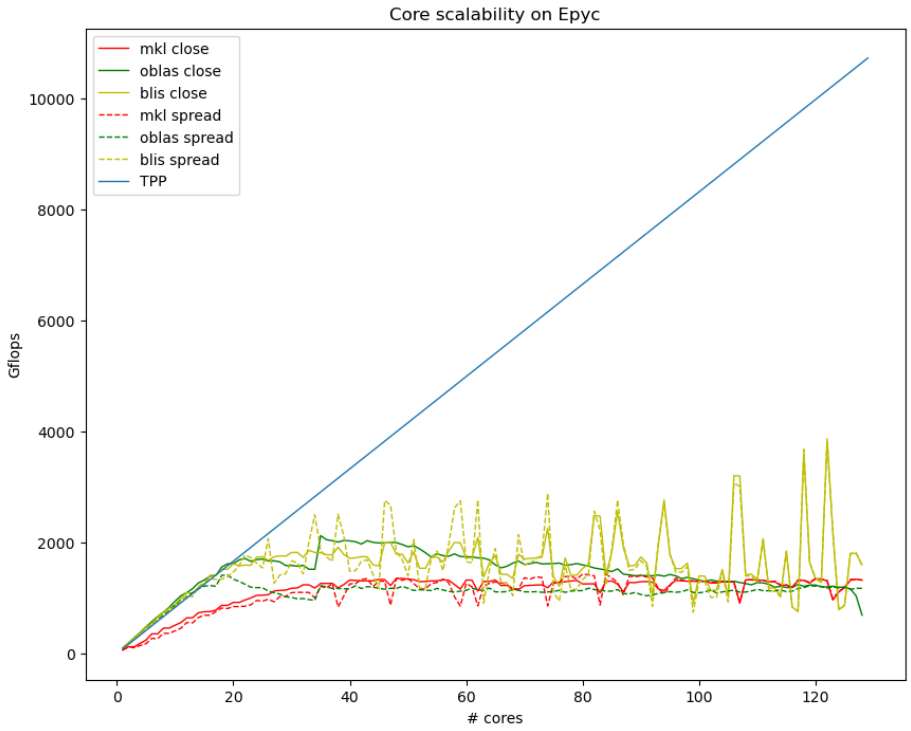

Figure 2.7: Scalability at different number of cores on EPYC node partition, size of matrices: 11000, precision: float

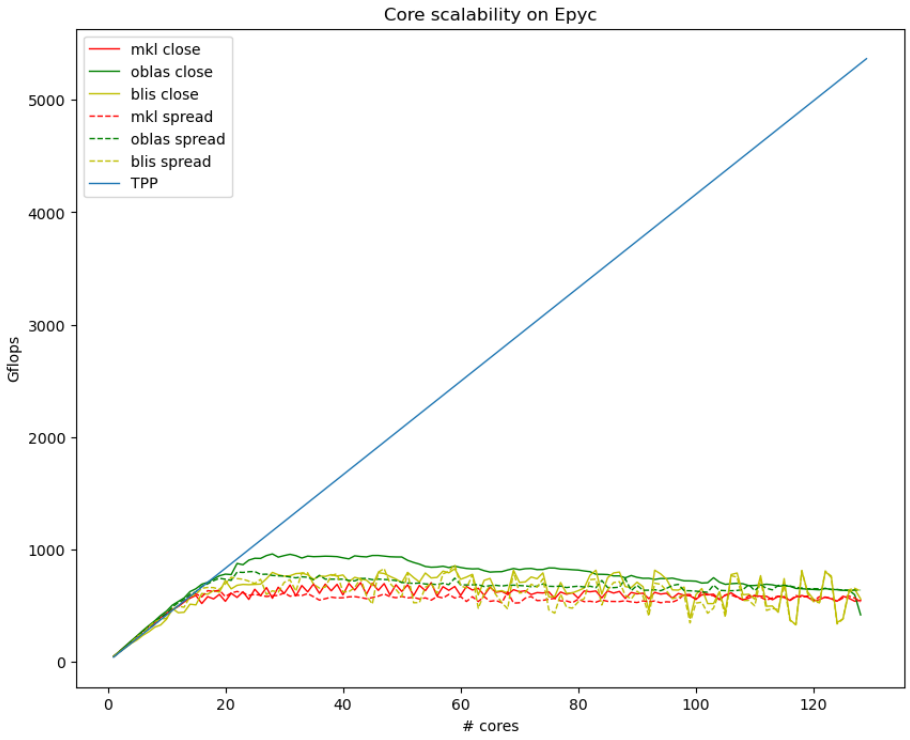

Figure 2.8: Scalability at different number of cores on EPYC node partition, size of matrices: 11000, precision: double

Numactl

As we think that a possible explanation for the decreasing trend could be found in the memory allocation, we decide to test the function numactl also here, with a different nodelist with respect to the number of cores used.

The policy that we tested is :

```
OMP_PROC_BIND=close and the numactl policy:
 From 1 to 16 OpenMP threads: --interleave=0
 From 17 to 32 OpenMP threads: --interleave=0,1
                    [...]
 From 113 to 128 OpenMP threads: --interleave
    =0,1,2,3,4,5,6,7
```

The numactl policy with close cores greatly improves the performances (Figure 2.9) and we can now observe a positive trend as the number of cores

increases. Since we can observe an improvement given from a different memory allocation, another reason that could affect the distance from the theoretical peak performance may be found in the distances between threads that struggle to reach the memory allocated by thread zero.

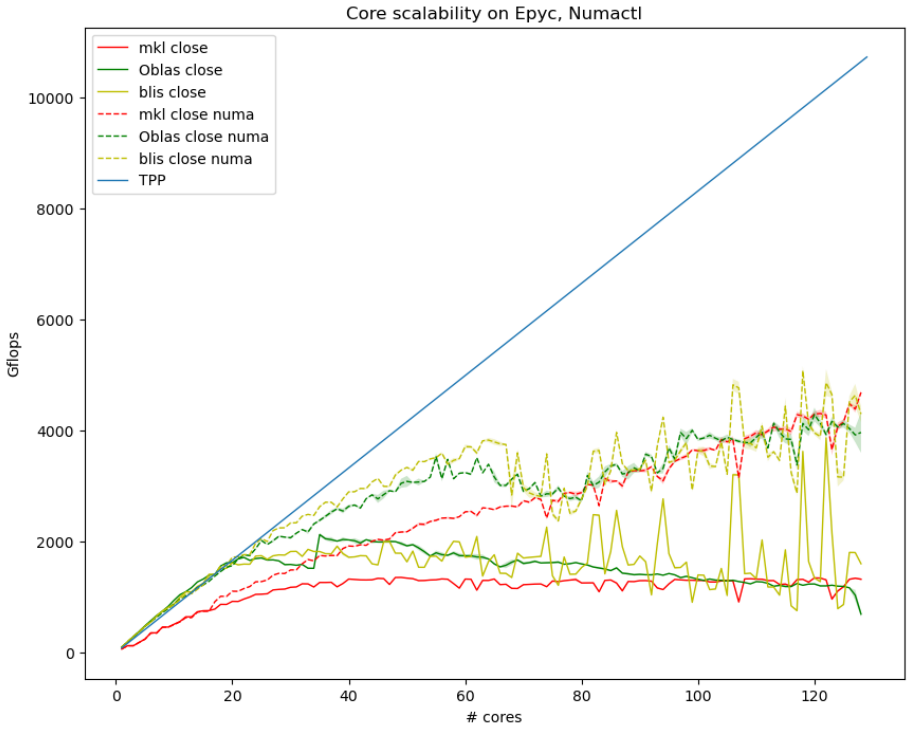

Figure 2.9: Core scalability on EPYC node, number of cores: 1-128, precision: float, numactl – interleaved policy, close cores

On THIN nodes

If we consider the core scalability study on THIN nodes, in Figure 2.10 and Figure 2.11, the gap between the actual measurements and the theoretical expected results consistently decreases, and we can observe a positive trend. Here the maximum number of threads is 24, thus we obtain a growing trend as we may imagine that the size is such that the data on which the threads are working do not interfere with other threads work.

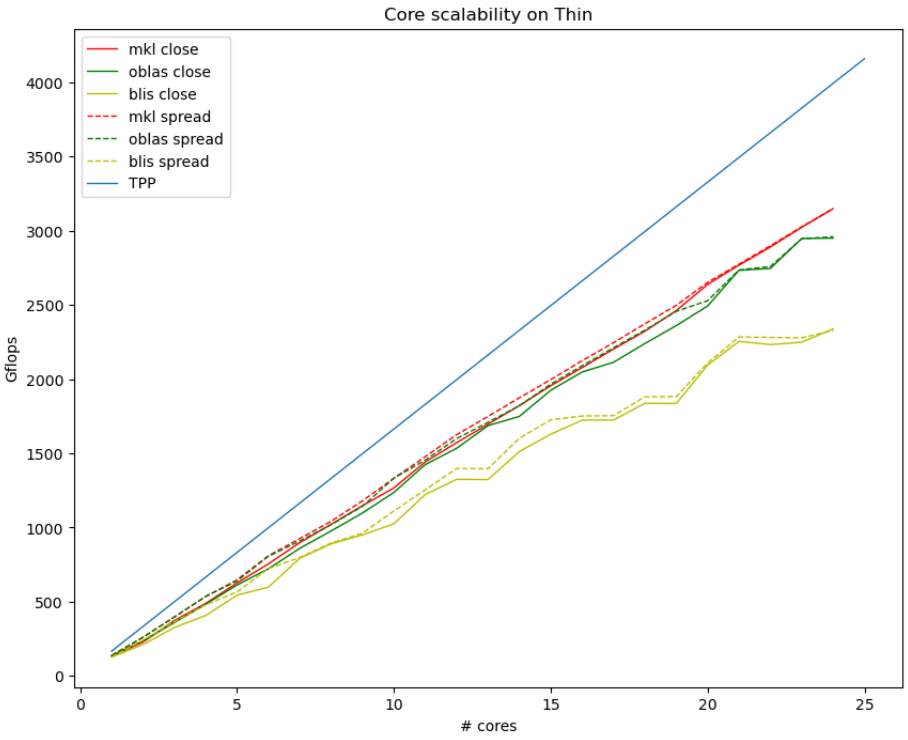

Figure 2.10: Scalability at different number of cores on THIN node partition, size of matrices: 11000, precision: float

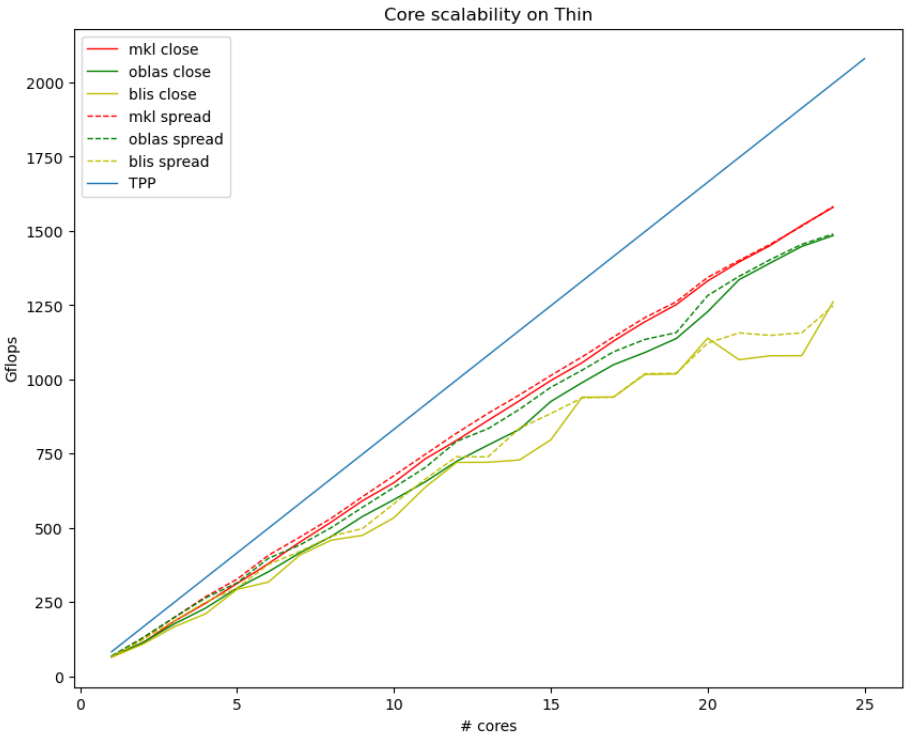

Figure 2.11: Scalability at different number of cores on THIN node partition, size of matrices: 11000, precision: double