Assignment - Foundations of HPC

Isac Pasianotto

Table of contents

1 Assignment 1

1.1 Introduction

In this assignment I was asked to submit a Conway's Game of Life implementation that is parallel with respect to both MPI and OpenMP. The full list of mandatory and optional tasks and a detailed description of the assignment requirements can be found in the given assignment description pdf file.

In this report I will present my implementation of the program implemented considering:

- the **ordered iteration method**: which means that the cells are updated in a sequential order (in particular starting from the top left corner of the matrix and going row by row) and its update is done before starting the next one:

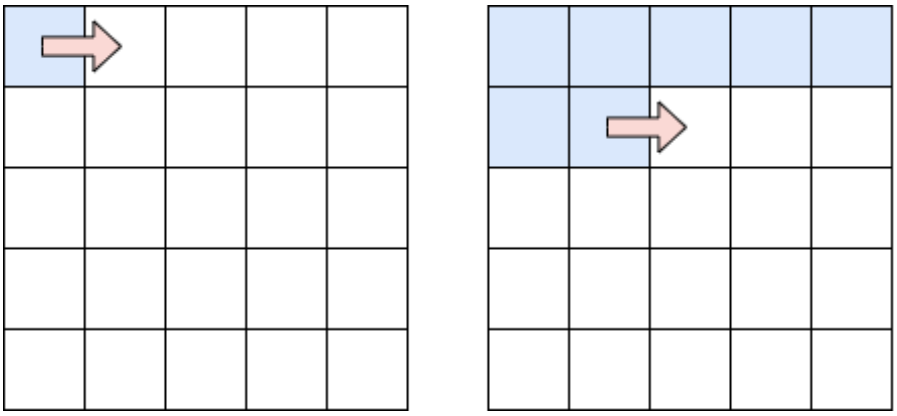


Figure 1: ordered iteration method scheme

- the **static iteration method**: which means that the playground is freezed and the program evaluates the next state every cell at the same time and then updates all of them only at the end of the iteration.

And I will also report the results obtained by running the program on the given cluster considering:

- the **strong MPI scalability**;
- the **weak MPI scalability**;
- the **OpenMP scalability**.

1.2 Methodology

The program is written in `C` programming language, using the `MPI` and `OpenMP` libraries for the parallelization. In the committed `Makefile` the compilation is done with the *gnu compiler* `gcc` and the `mpicc` wrapper for the MPI library.

The rappresentation of the playig field is done with a vector of `unsigned char` elements `k*k` length, where `k` is the number of cells in a row or column. The array mimc the square matrix of the Game of Life indexing the cells in the following way:

$$\begin{array}{ccccc} 0 & 1 & 2 & ... & \mathtt{k}-1 \\ \mathtt{k} & \mathtt{k}+1 & \mathtt{k}+2 & ... & 2\mathtt{k}-1 \\ 2\mathtt{k} & 2\mathtt{k}+1 & 2\mathtt{k}+2 & ... & 3\mathtt{k}-1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (\mathtt{k}-1)\mathtt{k} & \mathtt{k}^2-1 & \mathtt{k}^2 & ... & \mathtt{k}^2-1 \end{array}$$

Each cell can be in one of the two states: `alive` or `dead`, which are represented with the values `255` and `0` respectively.

The possibility of loading an initial configuration from a file and saving the final configuration and possibly the intermediate ones into a file is implemented using the `.pbm` format. In these files, the alive cell is going to be represented by a white square and the dead one by a black one.

The program can be run in two different ways:

- `-i`: which stands for **initialization**. In this case it will be randomly generated a matrix of `k*k` elements, where the size is settable with the `-k` option (default value is `100`) and saved into a file with the name settable with the `-f` option (default value is `game_of_life.pbm`).
- `-r`: which stands for **run**. In this case the program will load the initial configuration from a file with the name settable with the `-f` and will run the game:
  - for `-n` iterations (no default value, it must be set by the user);
  - saving the intermediate configurations every `-s` iterations (default value is `0`) where 0 means that the intermediate configurations are not saved. This intermediate file are saved in the `snaps` directory with the name `snapshot_nnnnn.pbm` where `nnnnn` is the number of the iteration.

The final result is written into a file with the named `game_of_life_END.pbm`.

The run mode can be done considering the two different iteration methods,settable with the `-e` option (default value is `-e 0`):

- `-e 0`: ordered iteration method;
- `-e 1`: static iteration method.

In regards to the measure of scalability required, I ran the program (in the *"run mode"*) on the Thin[1] nodes of the Orfeo cluster changing proprely the number of MPI tasks and the number of OpenMP threads. In order to have more reliable results, I measured each configuration attemped 5 times and I took the average of the results.

1.3 Implementation

In this section I'm going to discuss the technical details of the code I produced and the choices I made.

1.3.1 The `main.c` file & reading/writing files

The `main.c` file contains the main function of the program. It does two tasks:

1. It parses the command line given arguments and sets the variables accordingly. This part is done adapting the `get_args.c` code given in the assignment description. The only addition I made is the `-t` option, which makes the program print the time elapsed to run in the standard output. This option, which was not requested, was useful to me in the scalability assessment phase.
2. In case there is an mandatory argument missing, it prints the usage of the program and exits. Otherwise, it calls the appropriate function to continue the execution of the program and passes the arguments to it.

For what concerns dealing with the files, I used the code given in the assignment description `read_write_pgm_image.c`. Technically, the code was able to handle `.pgm` files, but since my need was to rappresent the alive and dead cells with white and black squares, the `.pbm` format enought for me. The reading and writing functions adopted are implemented only in a serial way, in the parallel contexts they are called by the master process only.

1.3.2 Initialization

When the program is run with the `-i` option, it generates a matrix of `k*k` elements randomly, with a probability of 0.15 for each cell to be alive. The initlization can be performed in serial and in this case it's quite straightforward: we just need to allocate a vector of `unsigned char` with the `malloc()` function, fill it with random values (using the `rand()` function and checking if `rand()%100` is less or greater than 15) and saving the result into a file with the `write_pbm` function.

[1]The choise of the Thin nodes is due to the fact that on days when I was measuring how the program scales, the Epyc nodes were particularly busy and the waiting time to get resources was too long. I suppose that conducting the measurements on the Epyc nodes would have give similar results.

The parallel initialization is more or less the same, the only difference is that in this case the work is divided among the MPI processes. In this case the vector is divided into `n` chunks, where `n` is the number of MPI processes. If the number of cells is not divisible by `n`, the master process takes care of the remaining cells[2]. When every process has done it's business, the master process collects all the chunks with the `MPI_Gather()` function and writes the result into the file.

1.3.3 Ordered iteration method

Since in the ordered method the evaluation of the next state of the n-th cell depends on the result of the evaluation of the $(n-1)$-th one, the domain decomposition approach to parallelization will not improve the execution time. For this reason the submitted implementation it's executed with a single MPI task.

The phases of the program when it runs with the ordered iteration method can be schematized as follows:

1. The program allocates enought space to store the starting configuration and fills it with the values it reads from the file.
2. It starts the extern `for` loop, which will be executed `n` times (number of iterations set by the user). In each iteration it will:
    - start the inner `for` loop, which will be executed `k*k` times (size of the row/column of the square matrix). In each iteration it will:
        - evaluate the next state of the i-th cell with the `should_live` function[3].
        - report immediately the result overwriting the previous value in the matrix that rappresents the current state of the game
    - check, according to the given parameter setted by the user with the `-s` option, if it's time to save the intermediate configuration. If it is the case, it saves the current state into the `./snaps` directory with the name `snapshot_nnnnn.pbm` using the `write_pbm` function.
3. When the execution of the `for` loop is finished, it saves the final configuration into the `game_of_life_END.pbm` file.

[2]There were other ways to divide the work among the processes, for example giving an extra cell to each process from the first to the `k%n`-th. Since the main focus of the project was on the run phase, I didn't spend too much time on investigating on the optmal choice.

[3]The `should_live` is a function used in both the ordered and the static iteration method. It takes as input: - the number of rows/columns that the playing field has;
- the index of the cell to be evaluated;
- the pointer to the matrix that rappresents the current state of the game.
- the pointer to the integer that rappresents the color depth of the image.
and returns a `unsigned char` value that could be either 0 or the value pointed by the last argument.
A cell still or becomes alive if it has exactly 3 or 2 alive neighbours, in all the other cases it dies.

4. Finally, it frees the memory allocated for the matrix and exits.

In this case the parallelization is done with OpenMP. In particular, the inner `for` loop is parallelized with the `#pragma omp parallel for` directive.

1.3.4 Static iteration method

In this type of evolution, the playgroud is "freezed" and firstly the program evaluates the next state of every cell and only then it all the matrix at the end. This means that it is possible to parallelize the code with MPI, giving to each process a chunk of the matrix to evaluate.

The implementation in serial is quite straightforward and it's the same as the one of the ordered method, with the only difference that the inner `for` loop which is executed `k*k` the result obtained by calling the `should_live` function is stored in a temporary matrix. At the end of the inner `for` loop, the temporary matrix become the new current state of the game swapping the pointers and the old matrix is freed.

The only relevant observation is that now the programs needs exactly the double of the space with respect to the ordered method, since it needs to store the current state of the game and the temporary one (two matrices of `k*k` elements).

The most intresting part is the parallel implementation. In this case workload is splitted among the MPI proccesses, where the i-th process should do the computation with respect to a chunk which size is:

$$\text{size of } i\text{-th chunk} = \begin{cases} \left\lfloor \frac{\mathtt{k}\cdot\mathtt{k}}{\#\ \mathrm{MPI}_{\mathrm{process}}} \right\rfloor + 1 & \text{if } \mathtt{k}\cdot\mathtt{k} \mod \#\ \mathrm{MPI}_{\mathrm{process}} \geq 0 \\ \left\lfloor \frac{\mathtt{k}\cdot\mathtt{k}}{\#\ \mathrm{MPI}_{\mathrm{process}}} \right\rfloor & \text{otherwise} \end{cases}$$

In this case the algorimth of the serial evolution method can be rappresented with the following phases:

1. Each process allocates enought space to store the starting configuration and fills it with the values that every process reads from the file.
2. There is a little needed overhead phase in which every process computes the size of portion of the global matrix that it has to evaluate and the starting index of the chunk. These informations are stored in the `lengths` and `offsets` arrays $\#\ \mathrm{MPI}_{\mathrm{process}}$ elements long.
3. Each MPI process allocates an additional array needed to store temporarily the result of the computation. This array has the same size of the chunk that the process has to evaluate.
4. The extern `for` loop starts, this cycle will be executed `n` times (number of iterations set by the user). In each iteration it will:

- wait for all the processes to finish the computation of the previous iteration (with the `MPI_Barrier` function).
- start the inner `for` loop, which will be executed size of i-th chunk times. In each iteration it will:
  - evaluate the next state of the i-th cell with the `should_live`.
  - store the result in the temporary array.
- Then every process will send its chunk of result to all the others (with the `MPI_Allgatherv` function). This is needed because in the next iterations each MPI process will need to have a copy of the whole matrix to call the `should_live` function.
- The root checks, according to the given parameter setted by the user with the `-s` option, if it's time to save the intermediate configuration. If it is the case, it saves the current state into the `./snaps` directory .

5. When the execution of the `for` loop is finished, it saves the final configuration into the `game_of_life_END.pbm` file by the root process.
6. Finally, every process frees the memory allocated for the computations.

Note than when the code is executed with MPI, the space each process needs to store decrease and it's alway less than two times the size of the whole playground as it was in the serial implementation.

1.4 Results & discussion

1.4.1 Code validation

To assess the correctness of the code, I've runned the serial version of the code with a very small playground (5×5) for a few iterations and manually checked the result for a certain number of trials. To check the correctness of the parallel version, I've use the `diff -s` command to compare the file generated by the two different implementations to make sure that the results are exactly the same.

Here is reported an example of the output that I've obtained:

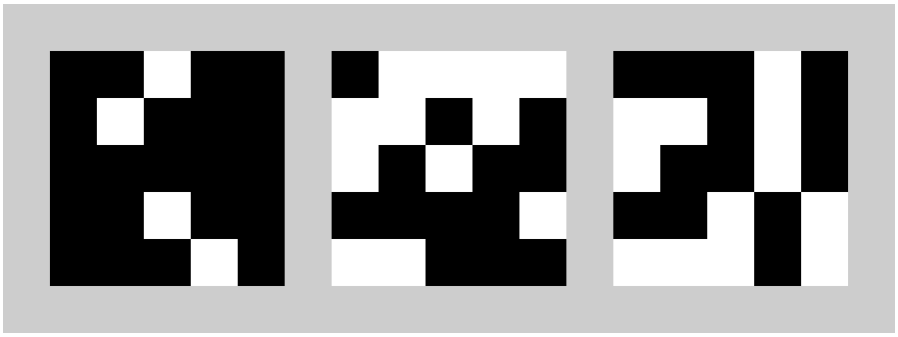


Figure 2: Ordered evolution execution

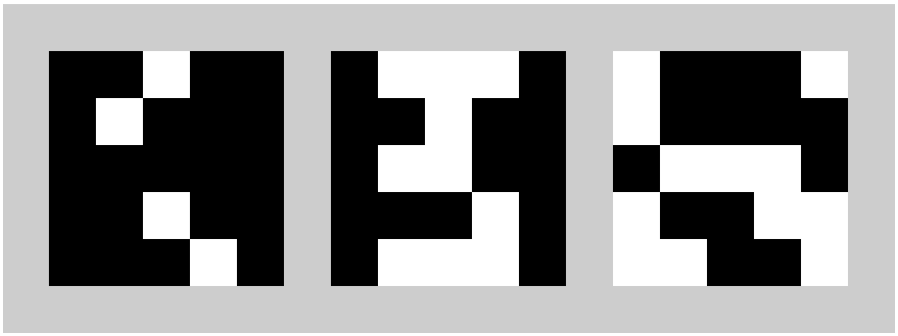


Figure 3: Static evolution execution

1.4.2 Scalability

In the following sections I'll present the results of the scaling tests that I've done in order to assess the scalability of the code. For each of the following tests, I've run the code multiple times (5) asking to the LRMS to have the exlusive use of the whole node given and I've computed the average time of the trials. First of all, there is a summary about I've computed the results that I've obtained…

1.4.2.1 OpenMP scalability

It was request to fix the number of MPI processes to 1 per socket and report the behaviour of the code with respect to the number of threads from 1 up to the number of cores present in the socket.

The Thin node I used has 2 sockets, each one with 12 cores. I runned the code both using 1 socket and all of them in the core, so the situation was the following:

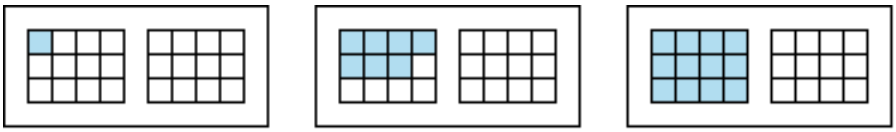


Figure 4: One socket: 1 MPI-task, from 1 up to 12 openMP threads

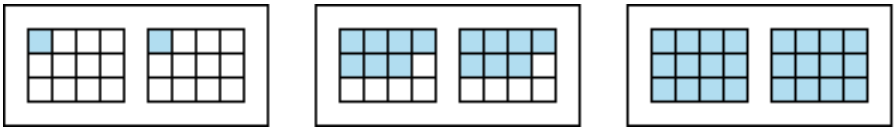


Figure 5: Two sockets: 2 MPI-tasks, from 1 up to 12 openMP threads for each task

In order to do that, I've used the `--map-by node --bind-to socket` options of the `mpirun` command and asked to the LRMS to reserve one node. I've tryed two different size of the side of the square playground.

I've measured the time the execution of the code took and use it to compute the **speedup** Sp and the **efficiency** Eff of the code with the following formulas:

$$Sp(p,n) = \frac{T_s(n)}{T_p(n)} \qquad Eff(p,n) = \frac{Sp(p,n)}{p}$$

where $T_s(n)$ is the time the serial version of the code took to solve a problem of size n and $T_p(n)$ is the time the parallel version of the code took to solve a problem of size n with p computing units.
Ideally, a perfect scalable code should have a speedup that is perfectly linear with the number of threads used and an efficiency that is equal to 1. What is auspicable for a good code is a speedup that is close to linear growth and an efficiency that is not too far from 1.

I also coomputed the **experimentally measured serial fraction of time** $e(n,\ p)$ with the following formula:

$$e(n,\ p) = \frac{\frac{1}{Sp(p,\ n)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

Ideally, if that fraction still constant as the number of threads involved in the computation increases, that means the lack of scaling w.r.t the ideal case is due to the fraction of serial workload that the code has to do. Otherwise, if that result encreases as the number of threads increases, that means that the lack of scaling is also due the hoveread of the parallelization.

1.4.2.2 Strong MPI scalability

For the strong scalability test, it was request to fix the size of the matrix which represents the playground and report the behaviour of the code with respect to the number of MPI processes.

I've ran the code in 2 Thin nodes, that having 2 sockets each one with 12 cores allowed me to test the code with up to 48 MPI processes. I've used the `--map-by core` option of the `mpirun`, leaving the default `--bind-to socket`: in this way, the LRMS will assign one MPI process as rappresented in the following figure:

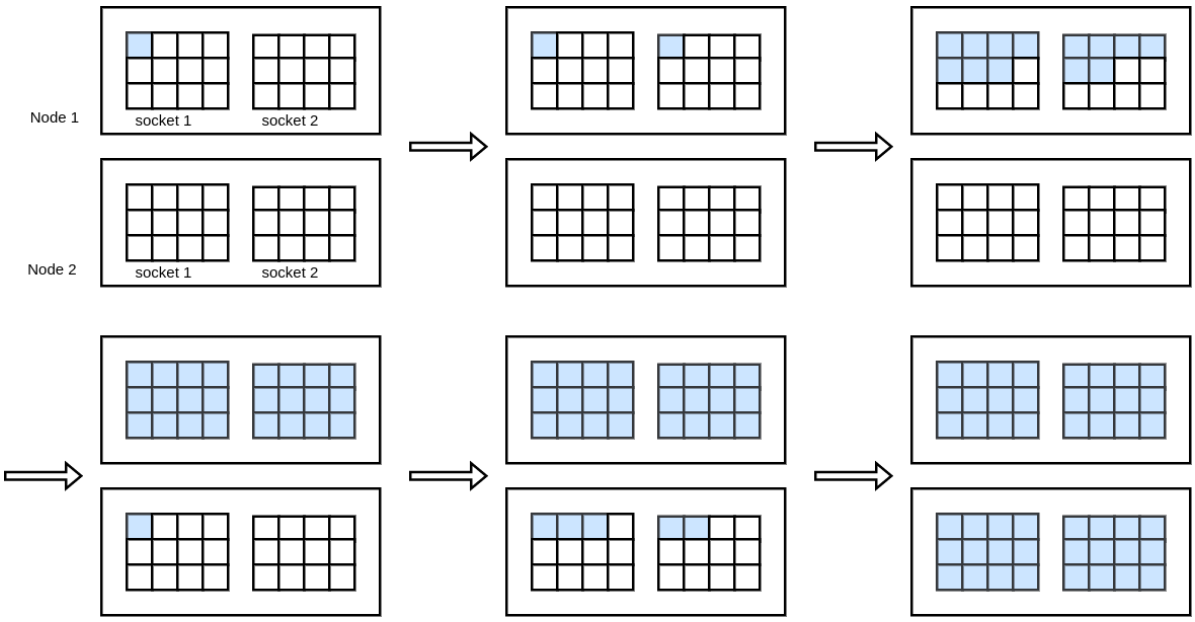


Figure 6: Strong MPI scalability, 2 Thin nodes: from 1 up to 48 MPI-tasks

I used this configuration because what I was more interested in was to check the scalability of the code in the static evolution method (since I didn't use MPI in the ordered evolution method) and using from the start box of socket a node has allowed the code to handle bigger playgrounds. The use of two distinct nodes was remanded to the range $\{25, \ldots, 48\}$ MPI processes, since the communication between the two nodes is not as fast as the communication between the cores of the same node.

To isolate the effect of the number of MPI processes from the effect of the number of threads, I've set the number of threads to 1 for each MPI process. Finally, I've repeated the measurements with three different size of the playground: $10,000^2$, $17,500^2$ and $25,000^2$ (always doing 10 iterations).

1.4.2.3 Weak MPI scalability

The idea of the weak scalability test is to run the code with a fixed number of MPI processes but keeping the workload each of them has to do constant. In the program, the workload is represented by the size of the playground, which is a square matrix with `k` rows and columns.

In other words, it must holds that:

$$\frac{(\texttt{k}(\#\text{MPI}_{process}))^2}{\#\text{MPI}_{process}} = \psi$$

where ψ is a constant defined as $\psi = \texttt{k}(1) \cdot \texttt{k}(1)$ and I put it equals to $10,000^2$.

With a simple algebraic manipulation, we can find the workload each MPI process has to do:

$$\texttt{k}(\#\text{MPI}_{process}) = \sqrt{\psi \cdot \#\text{MPI}_{process}}$$

Numerically, I've got:

# $\text{MPI}_{\text{process}}$	`k`(# $\text{MPI}_{\text{process}}$)
1	10,000
2	14,142
3	17,321
4	20,000
5	22,361
6	24,495
7	26,458
8	28,284
9	30,000
...	...

I've repeted the measurament using 1 and 16 openMP, expecting to see a constant time of execution in both cases.

1.4.3 Static evolution method scalability

In this section, I'll report the results of the scalability tests of the static evolution method, which is the one is parallelized with MPI and OpenMP.

1.4.3.1 OpenMP scalability (static evolution)

The results I've got are the following:

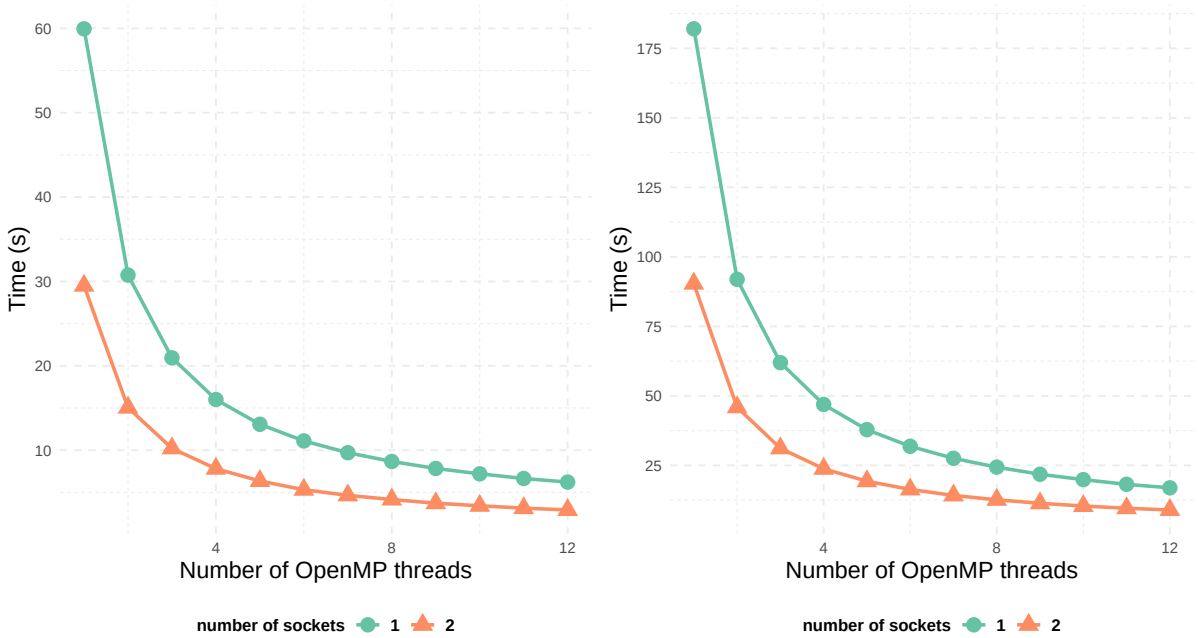


Figure 7: Time of static evolution: openMP scalability, size $10,000^2$ (on left) and $17,500^2$ (on right)

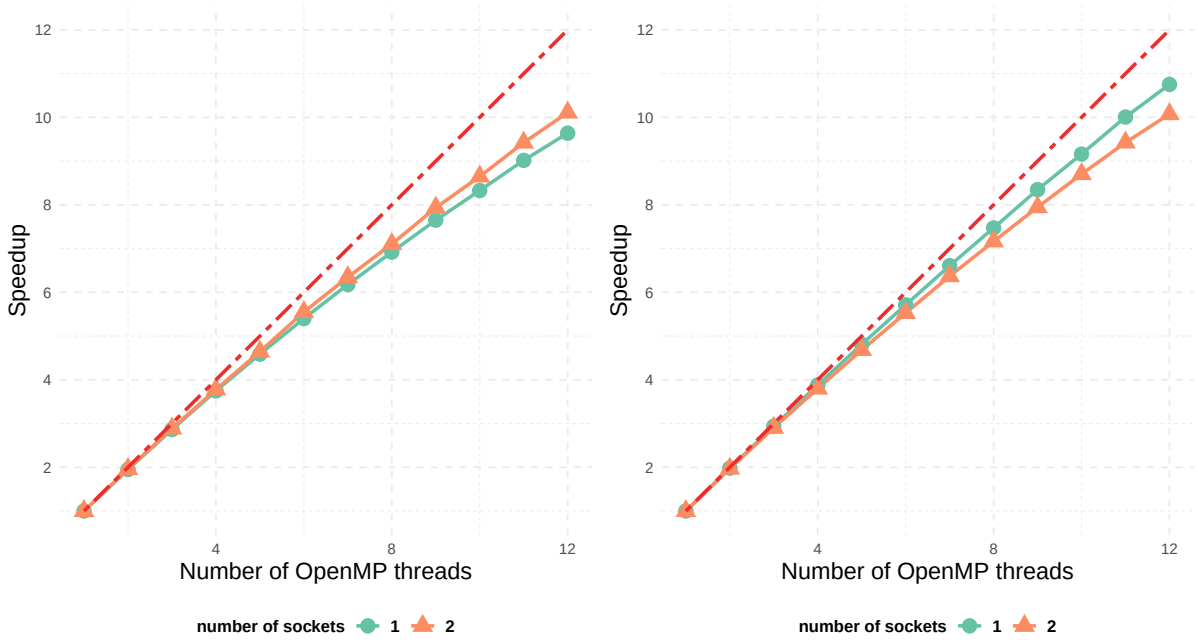


Figure 8: Speedup of static evolution: openMP scalability, size $10,000^2$ (on left) and $17,500^2$ (on right)

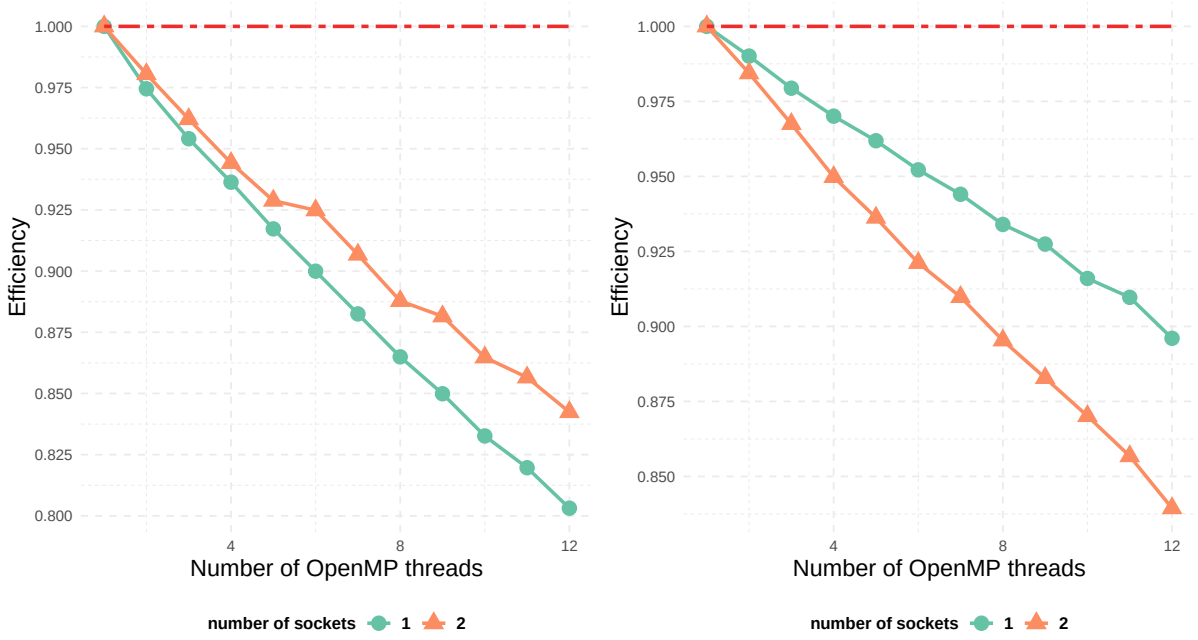


Figure 9: Efficiency of static evolution: openMP scalability, size $10,000^2$ (on left) and $17,500^2$ (on right)

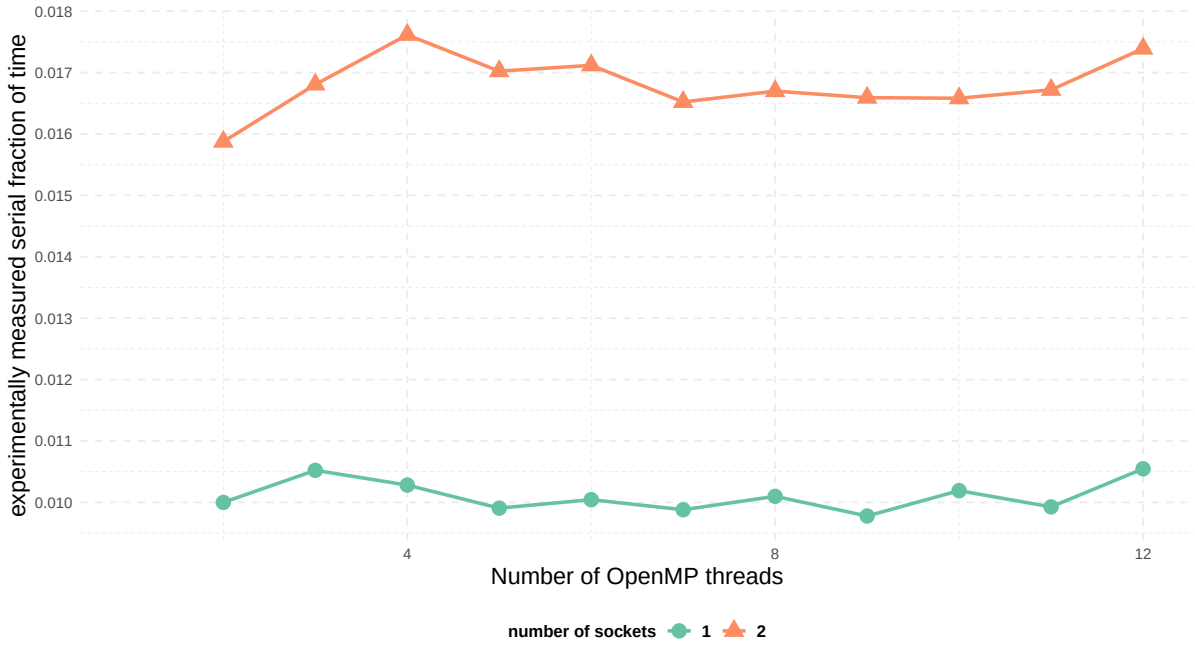


Figure 10: Experimentally measured serial fraction of time of static evolution: MPI scalability, size $17,500^2$

These plots show that:

- The code scales well with with respect to the number of openMP threads.
- Increasing the size of the problem (very) slightly increases the scalability of the code.
- The efficiency still be greater than 80%, which is a very satisfying result.
- The serial fraction of time is very low, and it is almost constant with respect to the number of openMP threads. That means there is only a constantn fraction of time that is spent in the serial part of the code (around 0.1% with one MPI task and 0.165% with 2).

1.4.3.2 Strong MPI scalability (static evolution)

In this section, I'll report the results of the scalability tests of the static evolution method, which is the one is parallelized with MPI. As I said before, for conducting these measurements, I set the number of openMP threads to 1, and repeated each experiment 5 times.

The results are reported in the following plots:

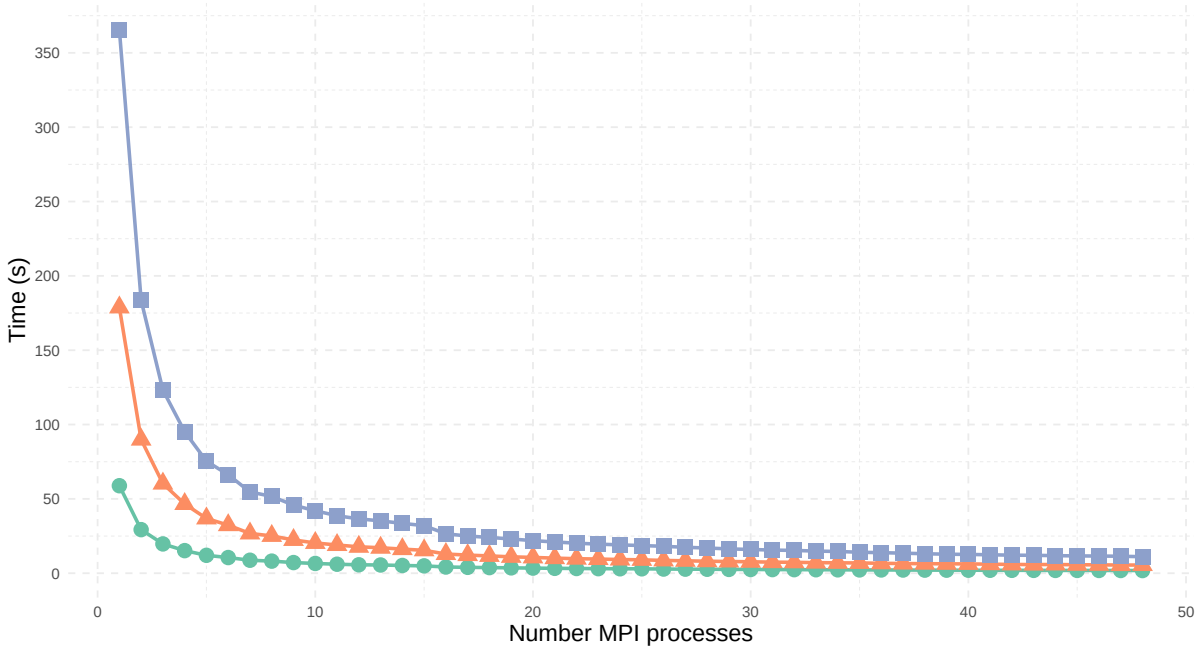


Figure 11: Time of static evolution: strong MPI scalability

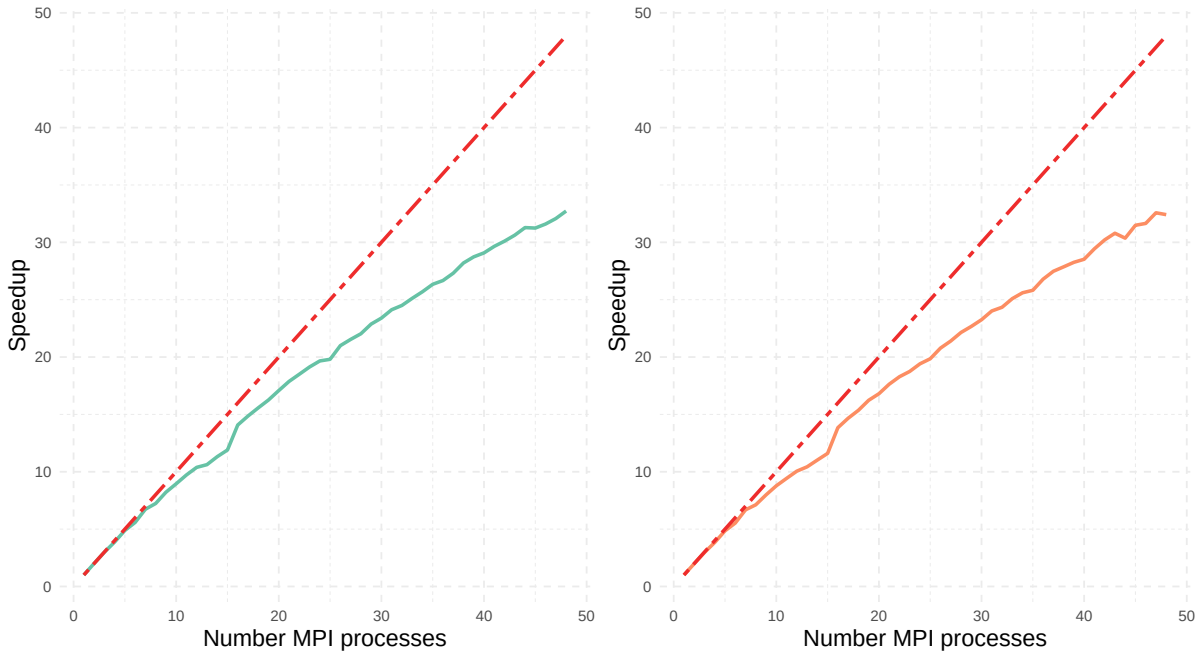


Figure 12: Speedup of static evolution: strong MPI scalability `k`=10,000 (left), `k`=17,500 (right)

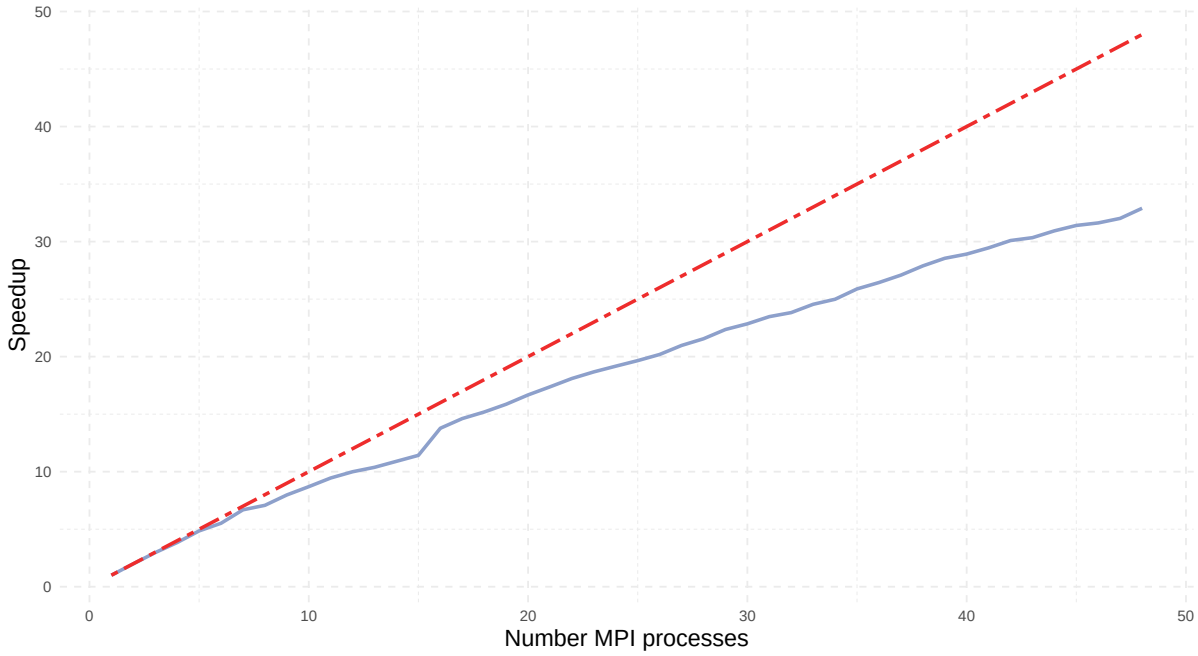


Figure 13: Speedup of static evolution: strong MPI scalability, k = 25,000

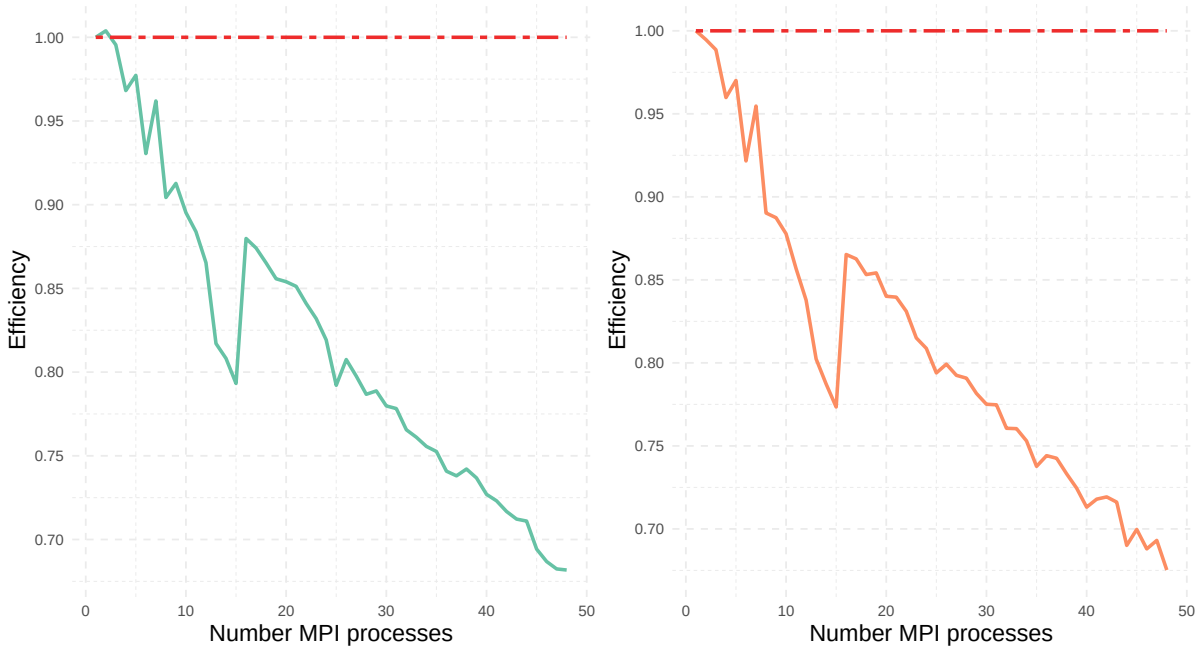


Figure 14: Efficiency of static evolution: strong MPI scalability k=10,000 (left), k=17,500 (right)

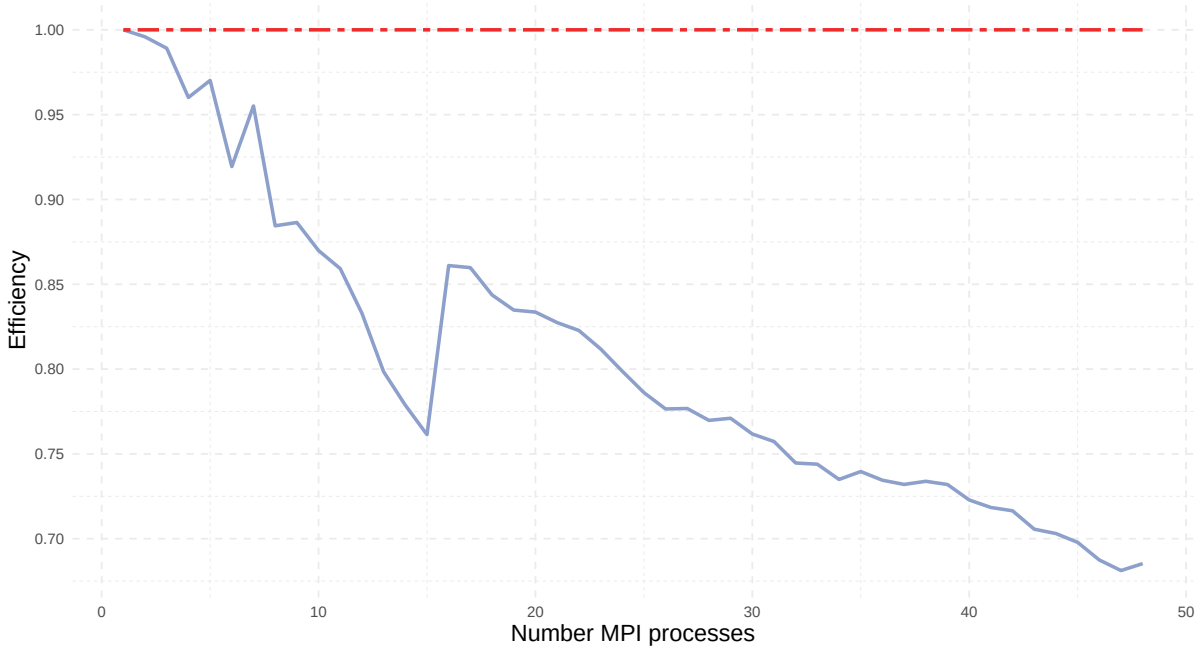


Figure 15: Efficiency of static evolution: strong MPI scalability k=25,000

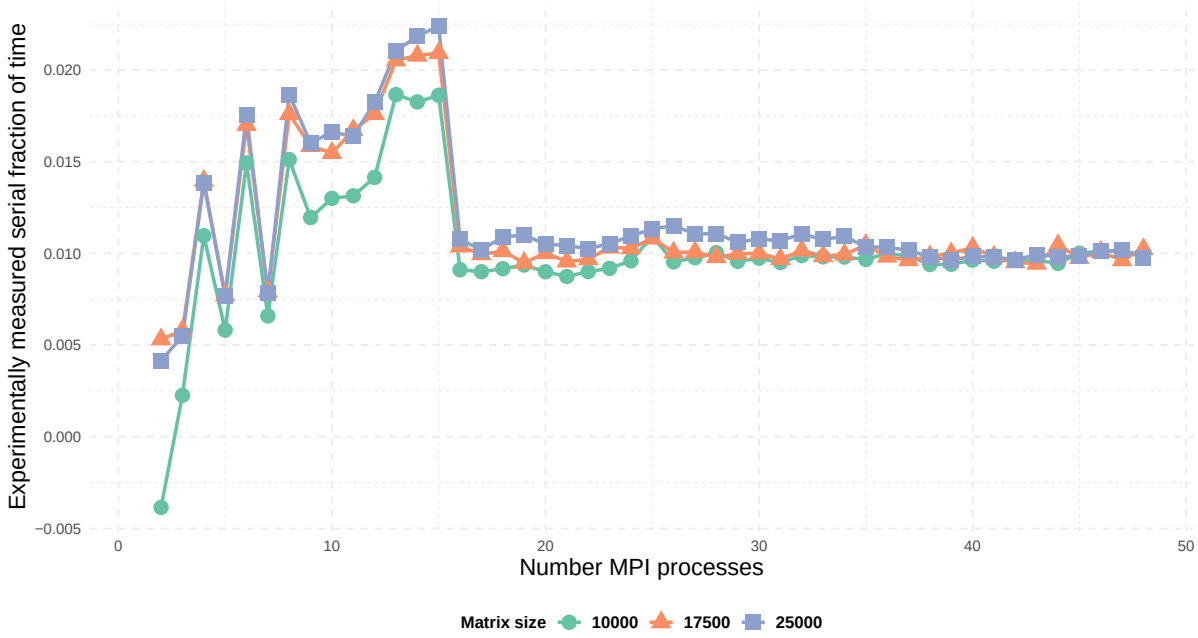


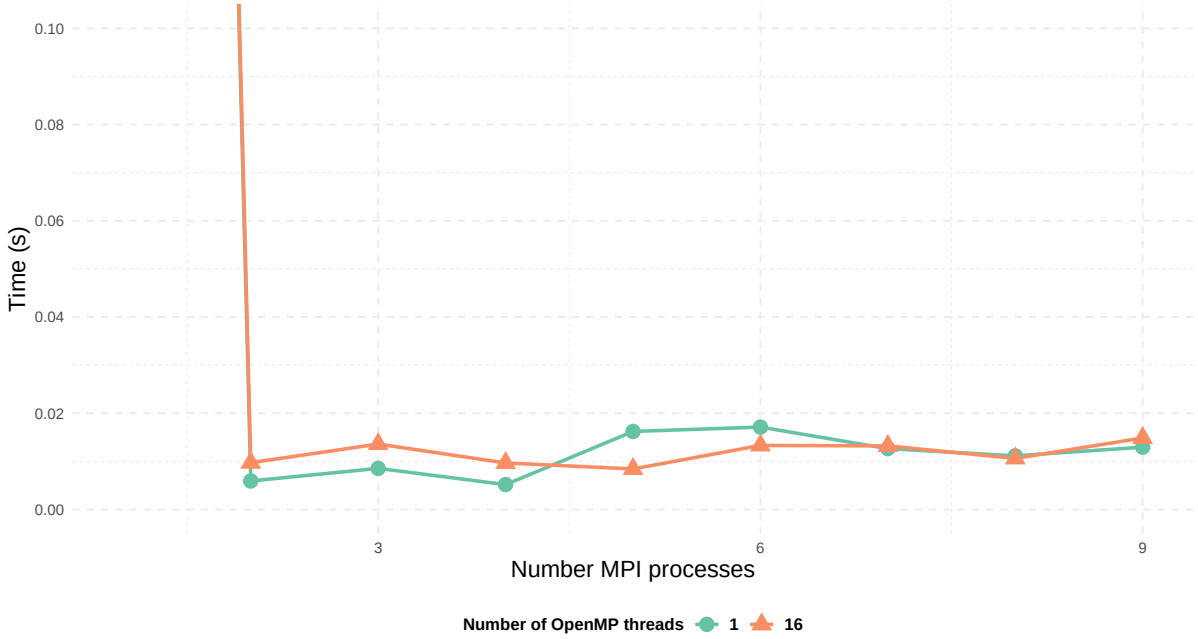
Figure 16: Experimentally measured serial fraction of time of static evolution: strong MPI scalability

These plots show that:

- The code I've committed has a satisfying strong scalability, as the speedup still increases with the number of MPI processes, even if it is not linear.
- The efficiency, which tends to 0 as the number of MPI processes increases, still decreasing with an acceptable rate: up to 48 MPI processes, the efficiency is still above the 67.5%.
- The most interesting result is the one aboyt the *experimentally measured serial fraction of time*: since the three lines present the same behavior and all of them are the result of an average of multiple runs, I can not imputate the irregularity to a causal factor. I can only conclude that for a low number of MPI processes, the lack of scaling is imputable to the parallelization overhead, which becomes negligible for a high number of MPI processes, where the missmatch between the performance measured and the theoretical perfect case is due to the fact that there is a fraction of code that is strictly serial and that can not be parallelized ($\approx 10\%$).

1.4.3.3 Weak MPI scalability (static evolution)

Finally, for what concerns the weak scalability, I've got the following results:



As expected, the time needed to complete the computation still constant because the workload for each MPI process is the same. The only exception is the case with 1 MPI, this is certainly due to the fact that the program when is executed with only one core, calls a different function that is not parallelized and which time is not comparable with the other ones.

1.4.4 Ordered evolution method scalability

In this section, I'll report the results of the scalability tests of the ordered evolution method. Since in this one the evolution of the n-th cell depends on the value of the $(n-1)$-th cell, the parallelization of the code with MPI would not be effective. The code is parallelized only with OpenMP, and the results are the following:

1.4.4.1 OpenMP scalability (ordered evolution)

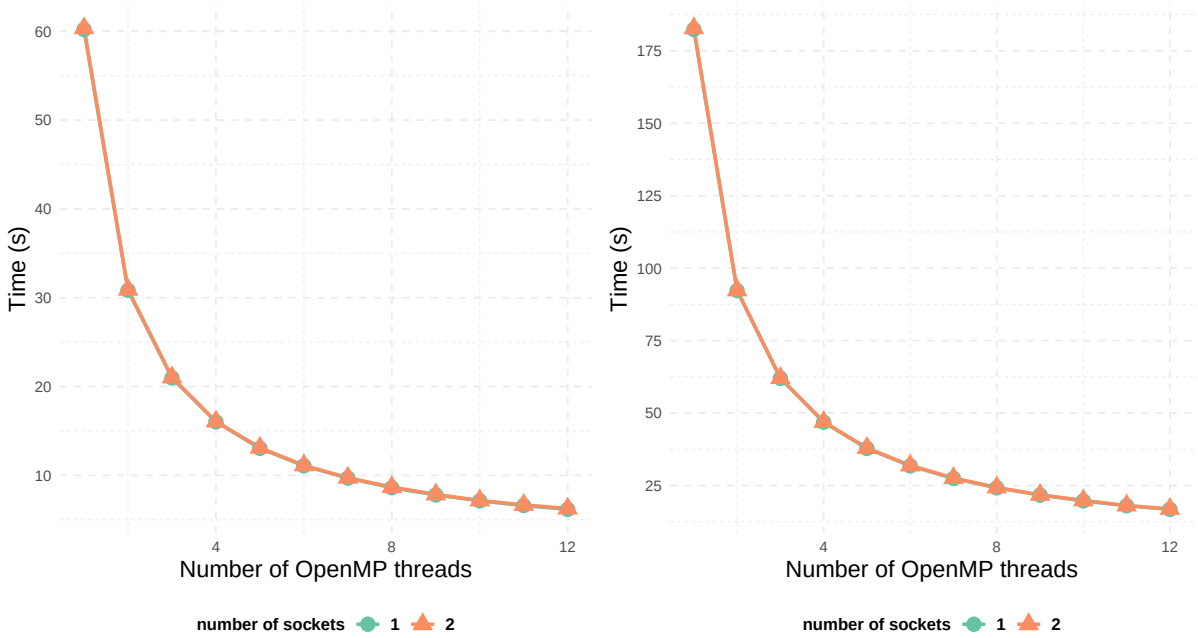


Figure 17: Time of ordered evolution: OpenMP scalability, size = $10,000^2$ (on left) and $17,500^2$ (on right)

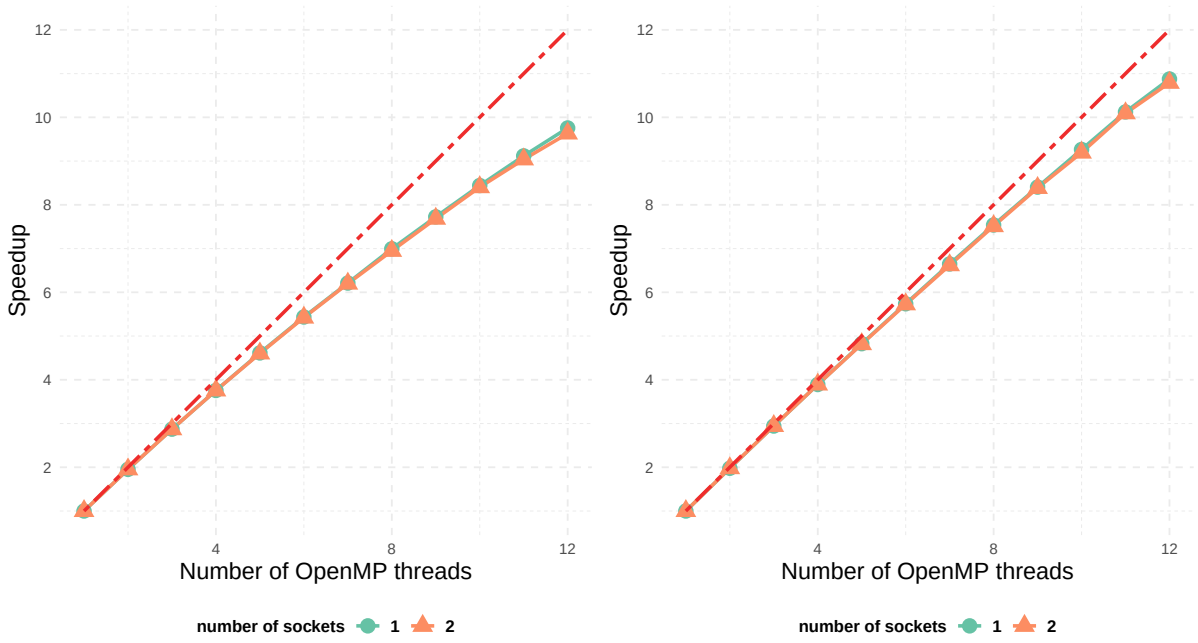


Figure 18: Speedup of ordered evolution: openMP scalability, size $10,000^2$ (on left) and $17,500^2$ (on right)

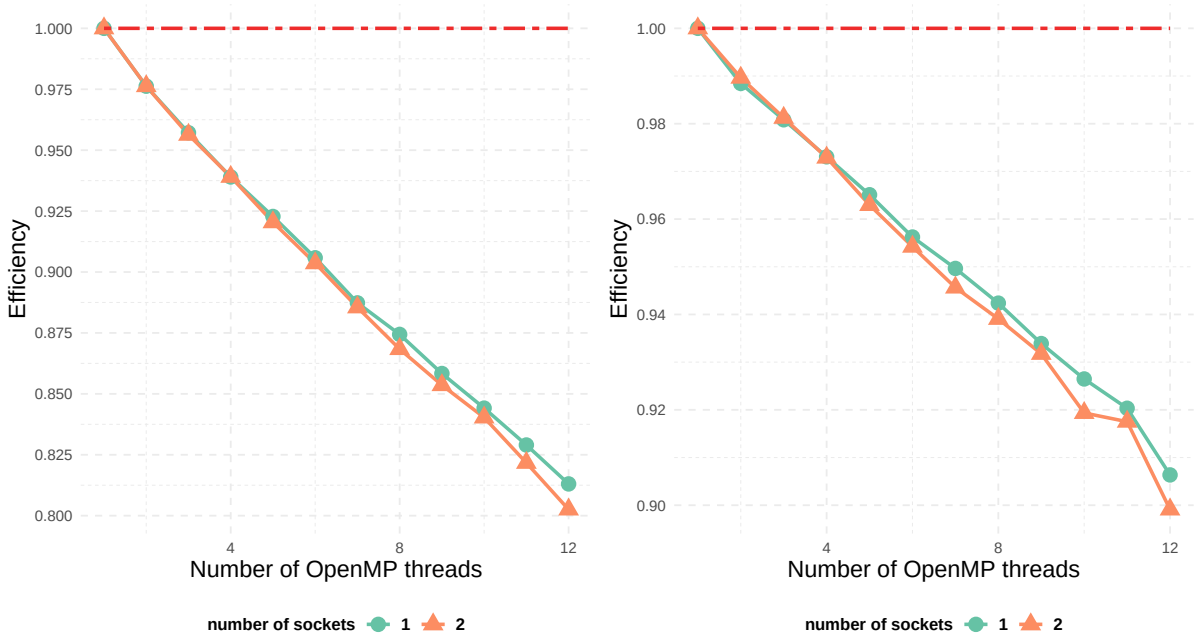


Figure 19: Efficiency of ordered evolution: openMP scalability, size $10,000^2$ (on left) and $17,500^2$ (on right)

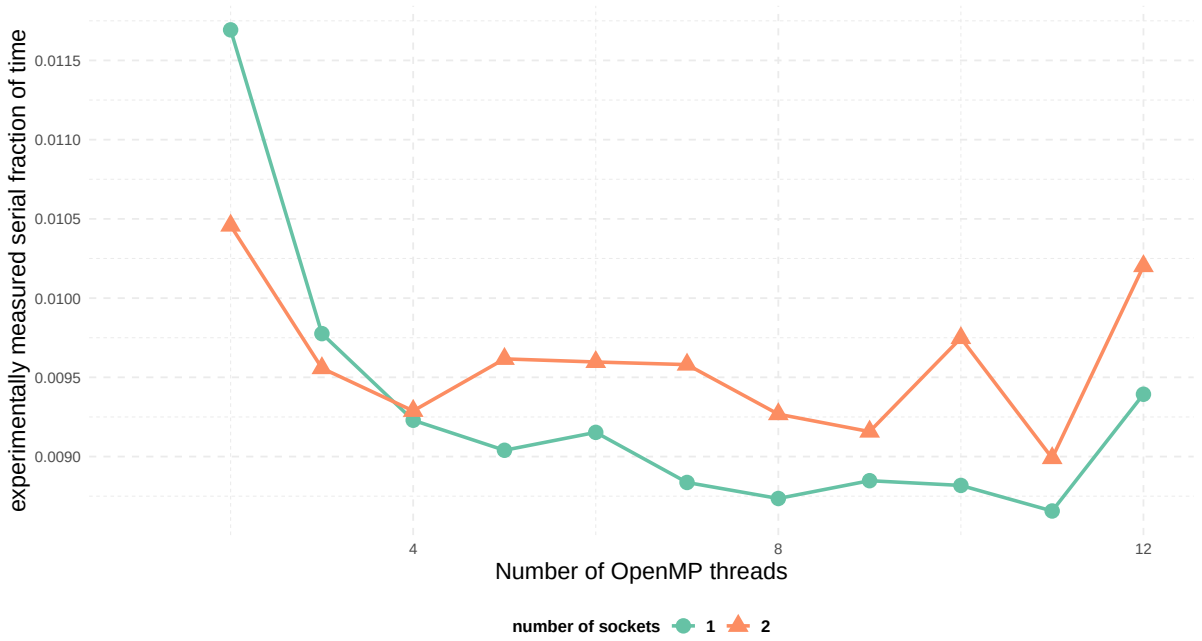


Figure 20: Experimentally measured serial fraction of time of ordered evolution: MPI scalability, size $17,500^2$

For what concerns the openMP scalability,the plot suggests that:

- The code scales well with respect to the number of openMP threads.
- For the bigger matrix, the code scales better (up to a certain limit obiously).
- The differences between running the code on a single socket and on two sockets becamo to be negligible (as expected).
- Up to 12 threads, the code still keeps a good efficiency (over 80%).
- starting from the 4th thread, the experimentally measured serial fraction of time is almost constant (around 0.925%).

1.4.4.2 Strong MPI scalability (ordered evolution)

As I've already said, due to the necessary implementation of the ordered evolution, the code will not scale with respect to the number of MPI tasks. Just to empirically verify this I've performed a measure of the time taken by the code with a matrix of size `-k 10000` and obtained:

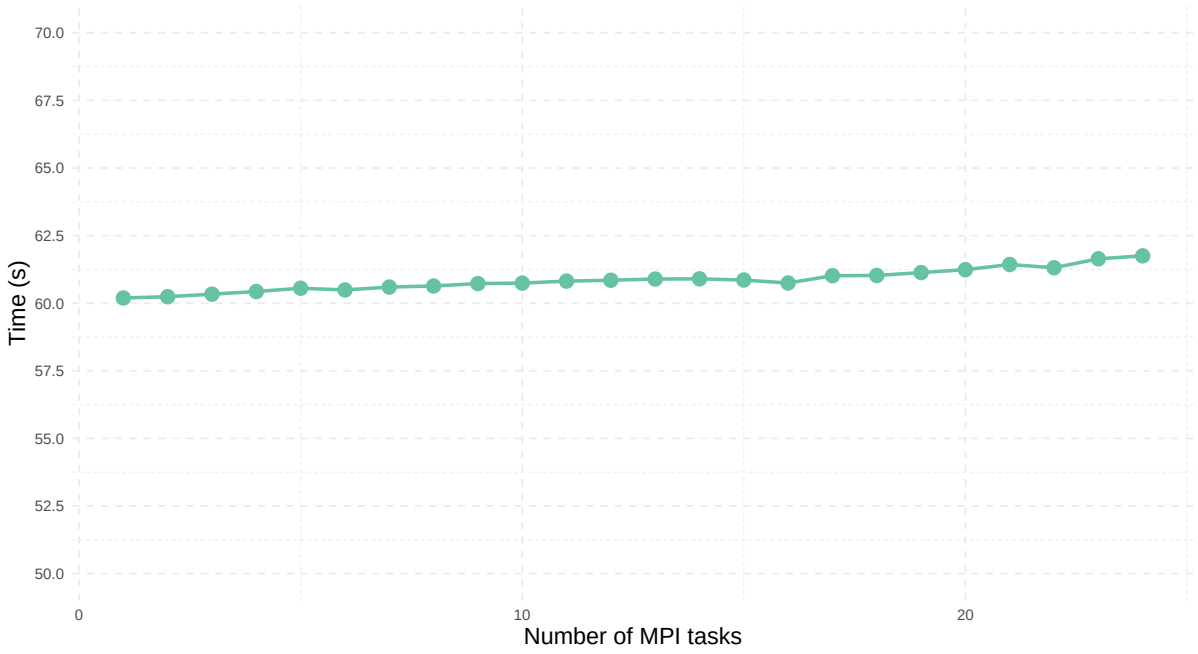
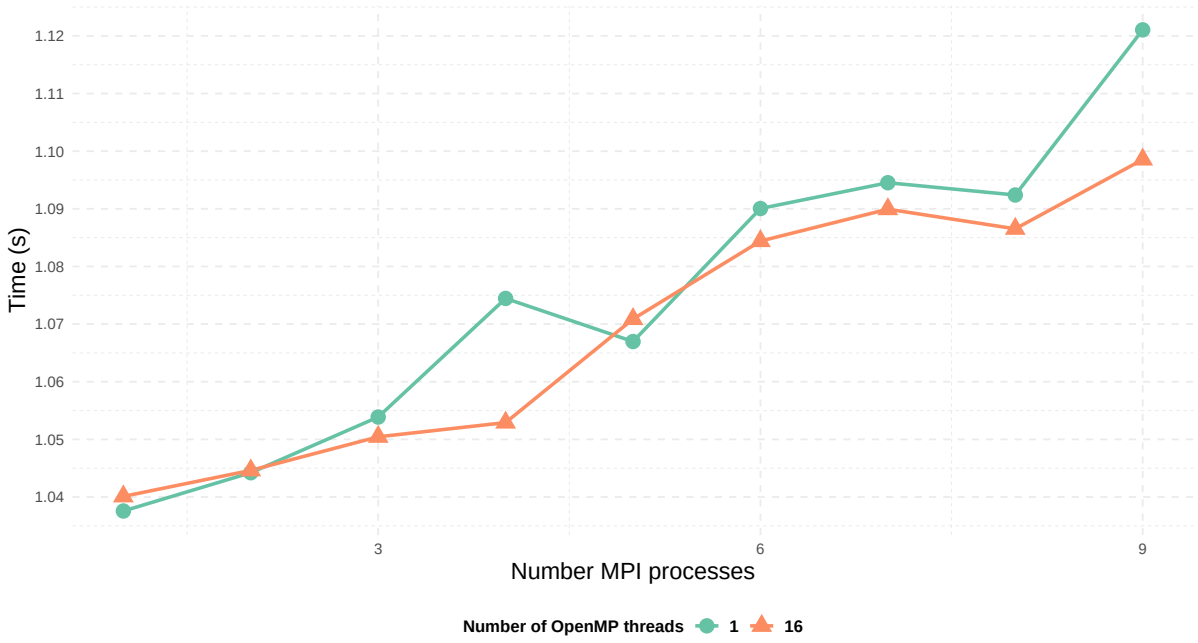


Figure 21: Strong MPI scalability of ordered evolution: MPI scalability, size $10,000^2$

As predicted, the time taken by the code is almost constant around 60 seconds, even if it seems to very slightly and imperceptibly slow down increasing the number of MPI tasks. When I run the code with more core than the number present in one node, the time start "jumped" around 100 seconds and then became more or less constant again (but more fluctuating). I guess that this is caused the physical allocation in run time of the memory used to store the two matrices.

1.4.4.3 Weak MPI scalability (ordered evolution)



As expected, the will not scale at all: since there is no difference running the code with one or more MPI tasks, the time taken by the code increases linearly with the size of the matrix it has to process.

1.5 Final considerations

At the end of the analysis, I can conclude that the code I've implemented has an acceptable scalability with respect to the number of MPI tasks and the number of openMP threads for what concerns the static evolution. The ordered evolution, instead, has a poor scalability but it's intrinsic to the nature of the problem.

1.5.1 Possible improvements

- The `read_pbm` and `write_pbm` functions are not parallelized. In particular the `write_pbm` is always called by the master process. It could be intresting try to parallelize them, for example using the `MPI-IO` and check where this approach leads to.
- The `ordered_evolution` is always called by only one proecss. It could be intresting trying to implement multi MPI-task version of the algorithm. I didn't expect to have an improvement in the time the algorimth takes to complete (due to the intrinsic serial nature of this kind of evolution), but If well implemented, it could allow to handle much

bigger matrices (since the memory needed to store the matrix could be divided among the MPI tasks in different sockets and nodes).

- In `parallel_static()` function each MPI task has its own copy of the whole matrix which rappresents the state of the system at a given time. This is a waste of memory, since to do the computation required, each MPI task needs only the information about the state of the system of his own chunk with two extra rows (one up and one down). It could be intresting to check if a version of the code where each MPI task has only the essential needed data, could lead to a better time performance even if there will be a probably higher overhead in the communication between the MPI tasks.

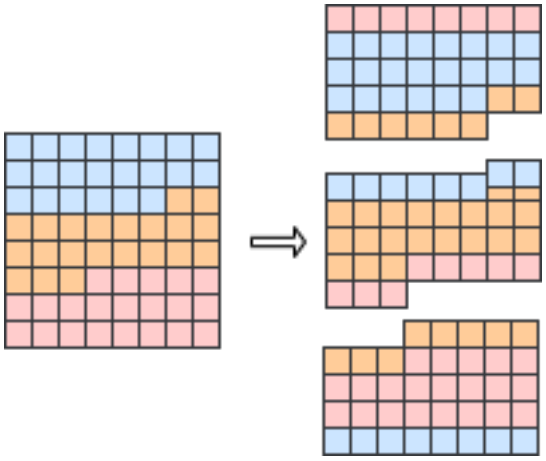


Figure 22: Example of the proposed domain decomposition: a matrix of size $\texttt{k}^2$=64, divided among 3 MPI tasks (the master process will handle 22 cells, the other two 21 cells each).

- Since I've tested the code only in the Thin nodes, it could be intresting how the code behaves in the Epic nodes too. Another interesting test could be check how the code act when it is compiled with other compilers like `icc`.

2 Assignment 2

The aim of this exercise is to show how three different math libraries (`OpenBLAS`, `MKL` and `BLIS`) perform on a matrix-matrix multiplication problem. To do that, I used a slightly modified version of the `dgemm.c` code provided by the professor, which I renamed `mygemm.c`. The only modification I made was on the `printf` statement to print the quantities of interest (time, GFlops) in a more convenient way for the further analysis.

According to the request, the matrix used for the test are all square matrices of the same size: $A,\ B,\ C \in \mathbb{R}^{\texttt{size}\times\texttt{size}}$, for this reason, in the following I will refer to the size of the matrix as the size of the problem (In general, performing a problem $A \times B = C$ with $A \in \mathbb{R}^{m\times n},\ B \in \mathbb{R}^{n\times k}, C \in \mathbb{R}^{m\times k}$ has a size problem of $m \times n \times k$).

It was requested to report the scalability of the code:

- fixing the number of cores (64 cores in the EPYC nodes, 12 in the THIN) and varying the size of the problem from 2,000 to 20,000 (I stepped by 1,000). This part is reported in the section named *"Size scaling"*.
- fixing the size of the problem at an intermediate value (I choose 12,000) and varying the number of cores from 1 to 64 (in the EPYC nodes) and from 1 to 12 (in the THIN nodes). This part is reported in the section named *"Core scaling"*.

Doing it in both `double` and `float` precision for floating point operations and for the different binding policies (I tested only `OMP_PROC_BIND=spread` and `OMP_PROC_BIND=close`).

In order to have a significative results, I performed the mesures multiple times, and considered the average of the results. In particular, i performed 15 runs for each combination of parameters in the size scalability tests and 10 runs for each combination of parameters in the core scalability tests. Always to have a significative results, I included the `#SBATCH --exclusive` directive in the `sbatch` command, to ensure that the cores are not shared with other jobs.

2.1 Remark:

After a first run of the code (node: Epyc, cores: 64(fixed), size: from 2,000 to 20,000, precision for f.p.o.: double, binding policy: spread), I noticed that the results unstable and fluctuating a lot. The results I got are rappresented in the following plot:

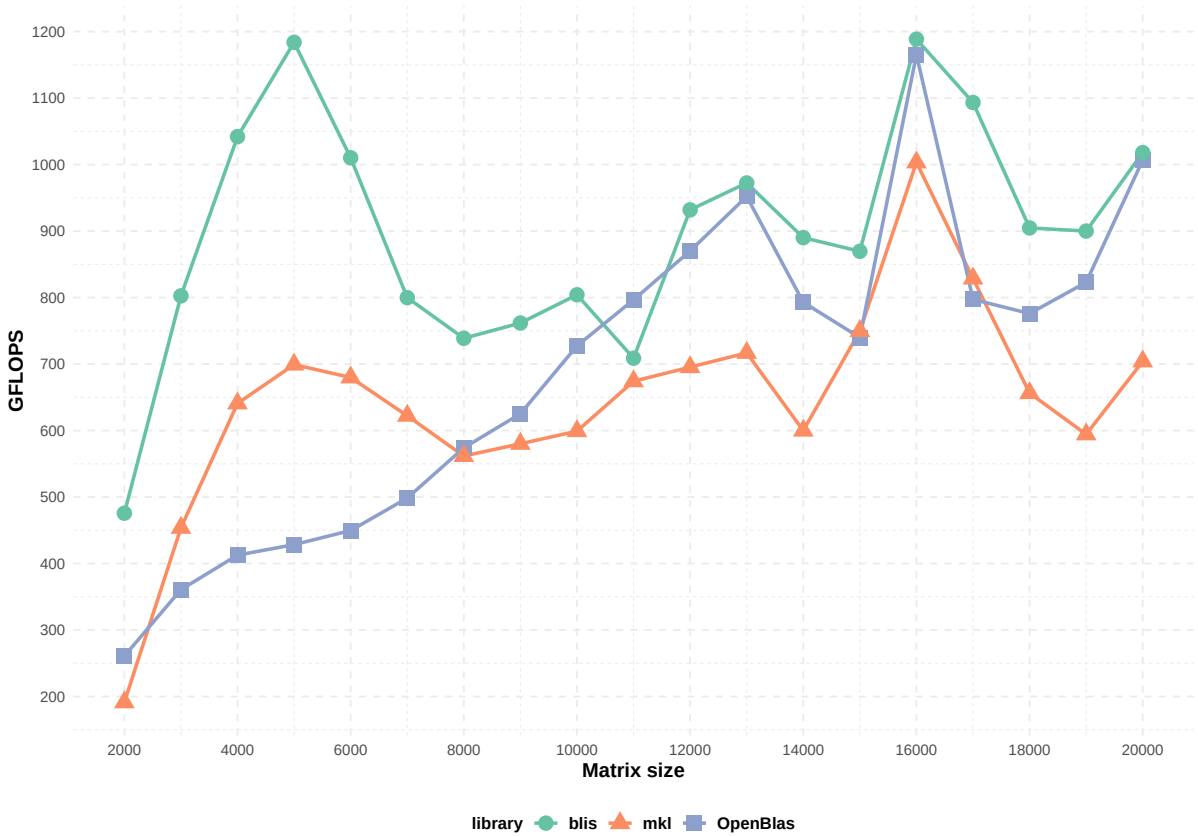


Figure 23: Size scaling in EPYC node, 64 cores fixed, double precision, spread binding policy

That result is caused by how the `mygemm.c` code is written. In particular, the initialization of the matrices (see lines 97-117) is done in serial [4] . This means that the matrices are initialized by a single core (using `OMP_PLACES=cores` as I did, and `OMP_NUM_THREADS`=64=number of cores fixed, each thread will be assigned to a single core) effectively introducing a "touch-first" policy even if the alloccation is done by `malloc` function.

Remembering that on AMD CPUs (like the ones used in the EPYC nodes) the cores are organized in NUMA reagions, the access to memory it's not uniform across the cores. In particular, the cores in the same NUMA region have a faster access to the memory than the cores in different NUMA regions.

To mitigate this problem[5], remembering that the CPUs we can find on the EPYC nodes are 2-socket CPUs, with 64 cores and 4 NUMA reagions each, I used the `numactl` command to

[4]That, as it seams rasonable, will increase reaching the the value we got in the size scaling section where the size of the matrix was 12,000.

[5]Since we are fixing the problem size, we can upper bound the speedup according to the Amdahl's law: $S(p,\ n) \leq \frac{1}{f+\frac{1-f}{p}} \leq 1/f$.

bind the cores to the NUMA regions. The policies I adopted are:

- for the **fxied number of cores**:
  - `numactl --interleave=0-7` when the process bingind policy is `spread` (force to use all the available memory regions);
  - `numactl --interleave=0-3` when the process bingind policy is `close`. (half of the memory regions, the one closest to the cores used by the process that are all grouped due to the `close` policy).
- for the **fixed size of the problem** things are a bit more complicated:
  - for the binding policy `spread`, I didn't use `numactl` because at each run the cores could be assigned differently.
  - for the binding policy `close`, I've done the same consideration as de "fixed number of cores" case, and do the following:
    * `numactl --interleave=0` if $n_{cores} \in \{1,\ \dots,\ 16\}$
    * `numactl --interleave=0-1` if $n_{cores} \in \{17,\ \dots,\ 32\}$
    * `numactl --interleave=0-2` if $n_{cores} \in \{33,\ \dots,\ 48\}$
    * `numactl --interleave=0-3` if $n_{cores} \in \{49,\ \dots,\ 64\}$

All this considerations are done for the EPYC nodes, for the THIN nodes I didn't use `numactl` because the cores are not organized in NUMA regions.

2.2 Size scaling

2.2.1 EPYC nodes

2.2.1.1 Double floating point operation precision

The theoretical peak performance can be computed as:

$$Flops_{peak} = \#\ cores \cdot \text{frequency} \cdot \frac{\text{FLOP}}{\text{cycle}}$$

The **AMD Epyc 7H12** can perform up to 16 double precision floating point operations per cycle, and has a maximum frequency of 2.6 GHz. This means that the theoretical peak performance is:

$$Flops_{peak}^{EPYC_{double}} = 64 \cdot 2.6\text{e}9 \cdot 16 = 2,662,400,000,000 = 2,662.4\text{GFlops}$$

The result I got are the following:

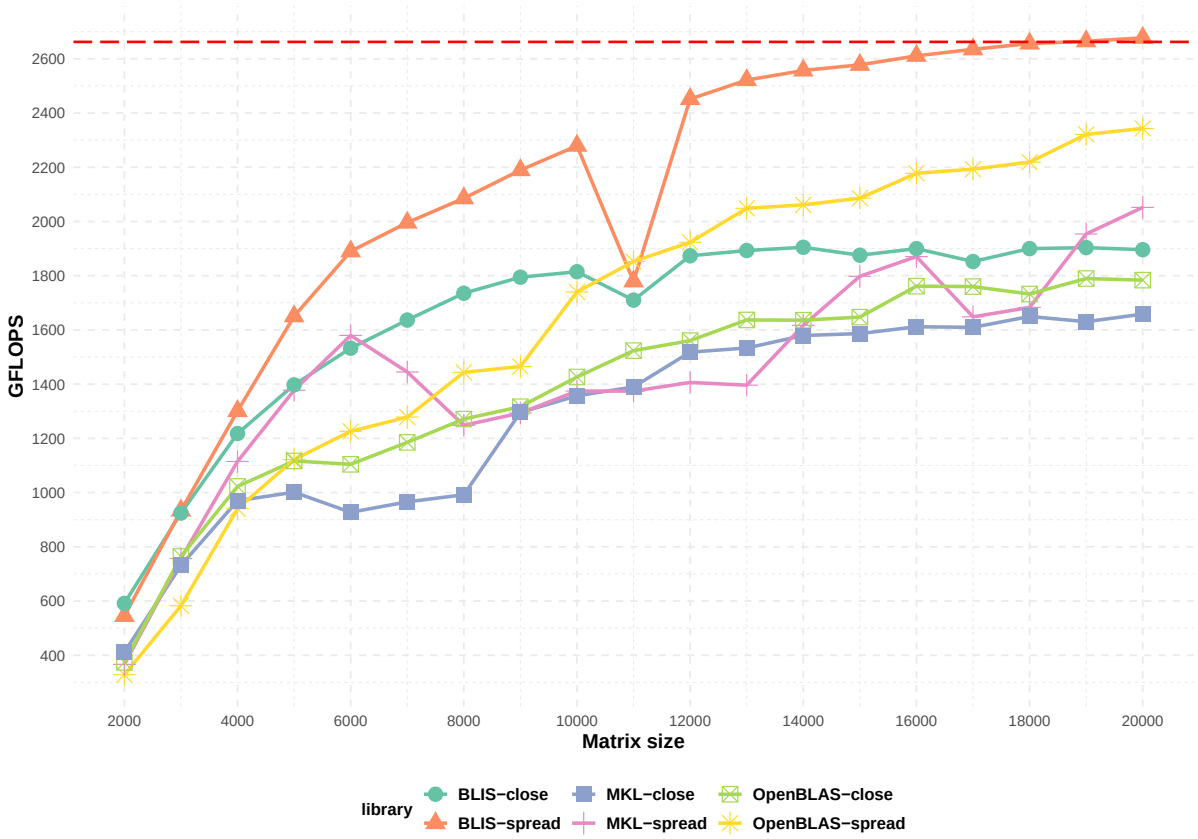


Figure 24: Size scaling: EPYC nodes, double precision

As we can see, the `spread` binding policy increase the number of floating point operations per seconds (the improvement is less noticeable while still appreciable for the `MKL` library). Moreover, we can say that `BLIS` library outperforms the others.

We can't ignore a big drop in the performance of the `BLIS` library when the size of the square matrix is 11,000 in both `spread` and `close` cases.

The size of 20,000 it's enought for the `BLIS` library to reach values very close to the $Flops_{peak}^{EPYC_{double}}$ threshold.

2.2.1.2 Single floating point operation precision

Regarding the floating point operation in single precision, the AMD Epyc 7H12 can reach at most 36 operations per seconds. Then the theoretical peak performance in this case is:

$$Flops_{peak}^{EPYC_{float}} = 64 \cdot 2.6\text{e}9 \cdot 32 = 5,324,800,000 = 5,324.8\text{GFlops}$$

Running the code, I got the following results:

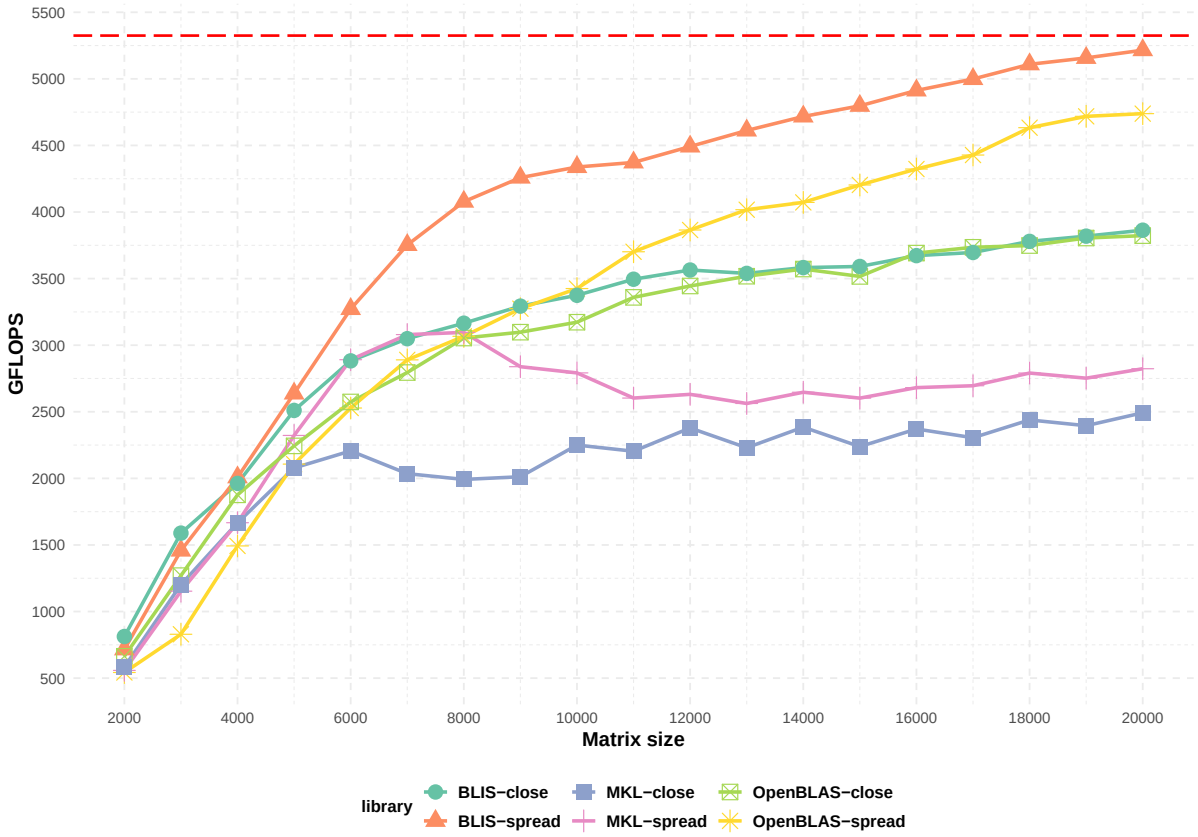


Figure 25: Size scaling: EPYC nodes, single precision

The `OpenBLAS` and `BLIS` libraries acted very similary to the double precision case. The `MKL` library, instead, performed significantly worse than the others. I suppose that this is due to the fact that the `MKL` library it's not as optimezed as the others for this kind of operations.

2.2.2 THIN nodes

Note that in order to perform the test on THIN nodes, I've re-compiled the `BLIS` library with loading the `architecture/Intel` module.

2.2.2.1 Double floating point operation precision

For the **Intel Xeon Gold 6126** processor, the one mouted in the THIN nodes, the theoretical peak performance we can gat for double precision Flops is: 998.5 GFlops[6].

What I got from my test is the following:

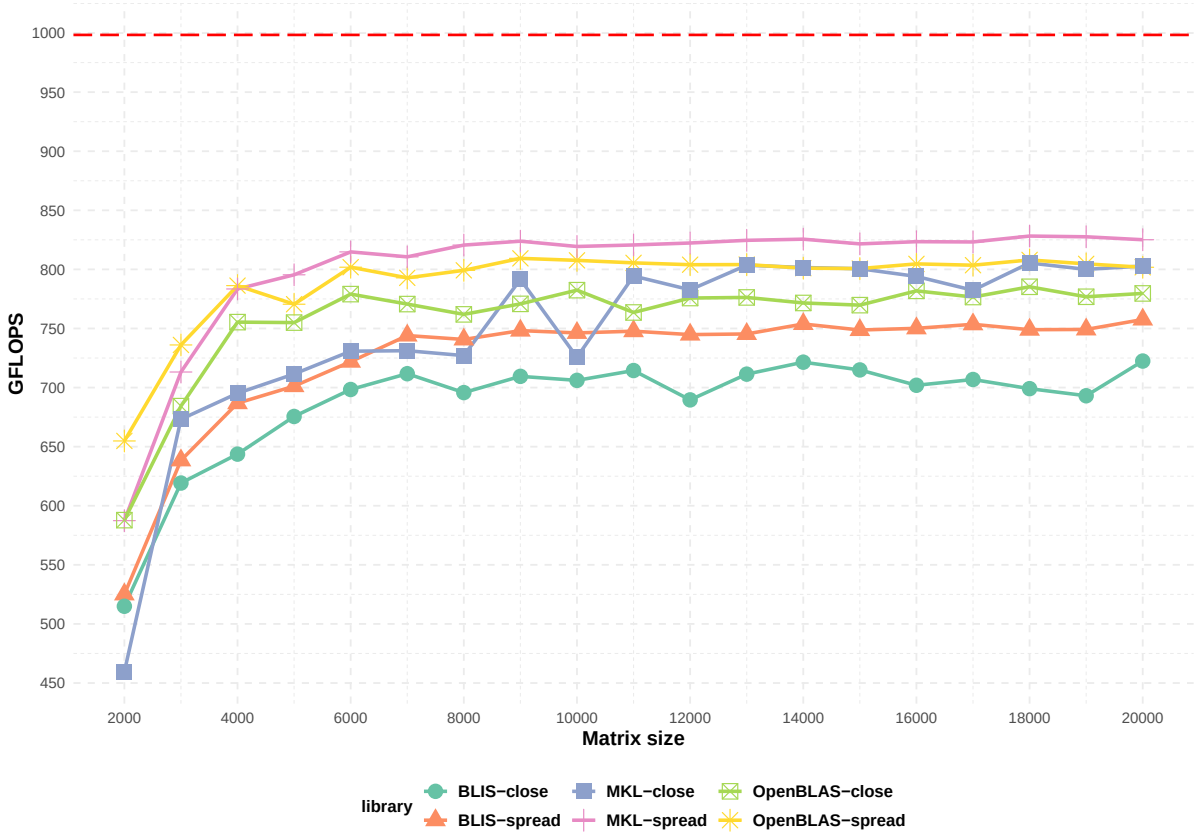


Figure 26: Size scaling: THIN nodes, double precision

Again, the `OMP_PROC_BIND=spread` policy outperforms the `close` one. Moreover the library that leads to the best results is `MKL` while `BLIS` performs worse than all the others. We have also to note that, unlike what happened in the EPYC nodes, in this case the size of the 20,000 square matrix was not sufficient to arrive at a number of GFLOPS close to the limit $Flops_{peak}^{THIN_{double}}$.

2.2.2.2 Single floating point operation precision

[6]1,997 TFlops is the the theoretical peak performance for an entire node (data founded in the given course material, at slide 63 of this file) and we are using 12 of the 24 available cores.

Halving from double to single the precision for floting point operations, the theoretical peak performance doubles, then $Flops_{peak}^{THIN_{float}} = 1,997$ GFlops. Performing the code I obtained the following results:

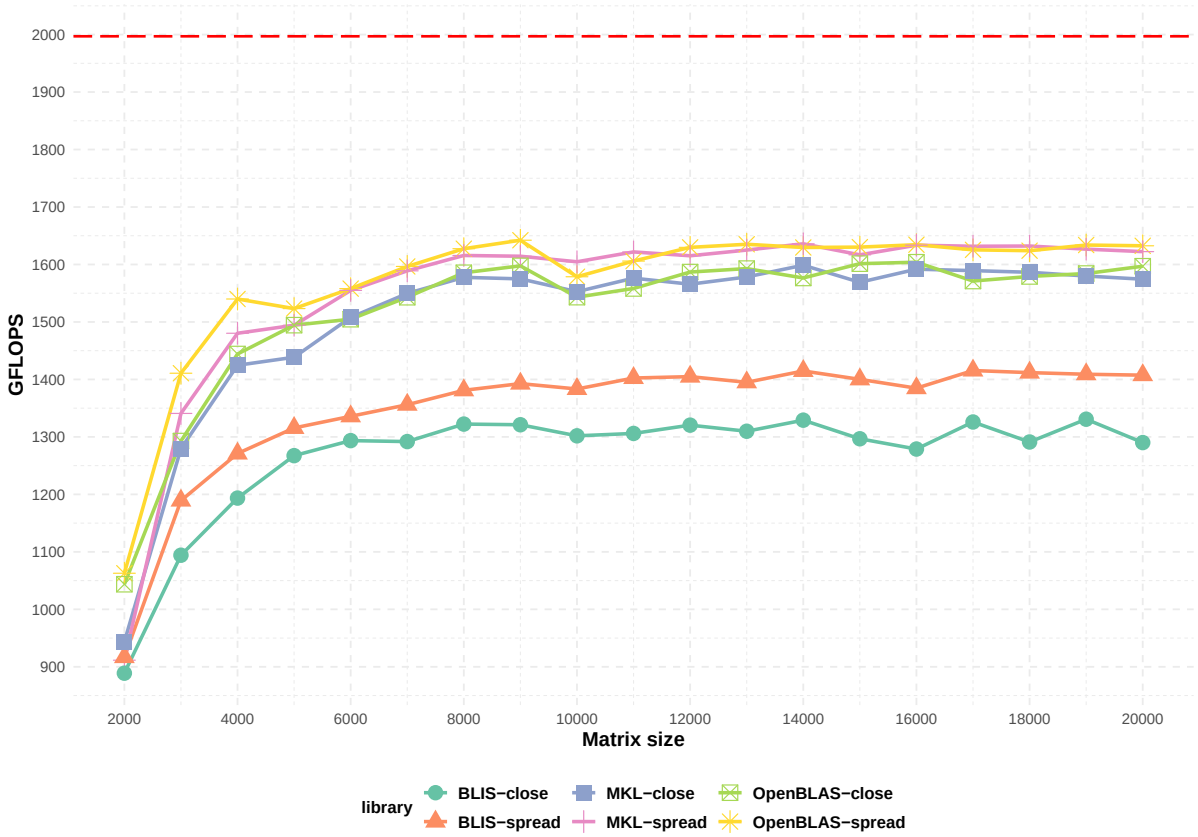


Figure 27: Size scaling: THIN nodes, single precision

Consistently with what happened in the previous case with double precision, the peak performance is not even remotely reached. As before, the `BLIS` library perform worste than the other. The first relevant thing to notice is that, according to that plot, `OpenBLAS` library is more or less as performant as the `MKL`, while in the previous case the last one seemed to work a little bit better.

Another relevant thig to notice is that in this case, unlike what happened with the EPYC nodes, the `MKL` library does not have a fall in performance with floating point precision.

2.3 Core scaling

In this section we will investigate the scaling of the code on the cores of the same node among the tree different libraries.

Since now what is changing is the number of the cores, it becomes less intresting the number of the GFlops the code is able to reach[7], what matters more is the time it takes to perform the operations. More in details, I considered the **speedup** and the corresponding **efficiency**:

$$S(n,\ t) = \frac{T_s(n)}{T_p(n)} \qquad E(n,\ t) = \frac{S(n,\ t)}{p}$$

where: n is the fixed problem size, T_s and T_p are respecrively the time needed to solve the problem in serial (on a single core) and in parallel (on p cores).

The ideal the speedup would be equal to the number of cores p, I reported that upper bound limit in the plots that follows rappresenting it with a red dashed line.

This limit is hardly reached because every code has a fraction f of it which is intrinsically sequential and not parallelizable. The more f is close to 1, the less the speedup will approach the upper bound[8].

For what concerns the efficiency, the auspicable behaviour is that it should be be as close as possible to 1, that is the case when the code is perfectly parallelized. Obiusly more the speedup walks away from the upper bound, more the efficiency will tend to 0.

Testing the code both with `OMP_PROC_BIND=spread` and `OMP_PROC_BIND=close` and increasing the number of cores with `OMP_NUM_THREADS` I obtained the results reported in the following sections.

2.3.1 EPYC nodes

2.3.1.1 Double floating point operation precision

[7] That, as it seams rasonable, will increase reaching the the value we got in the size scaling section where the size of the matrix was 12,000.

[8] Since we are fixing the problem size, we can upper bound the speedup according to the Amdahl's law: $S(p,\ n) \leq \frac{1}{f+\frac{1-f}{p}} \leq 1/f$.

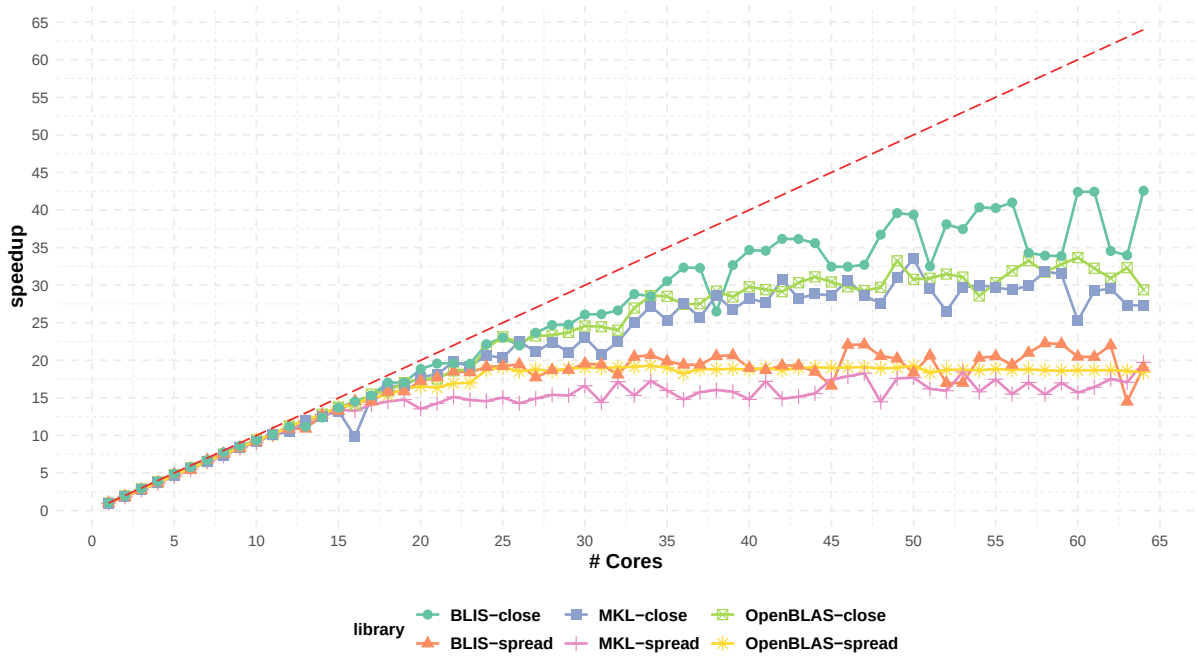


Figure 28: Core scaling: EPYC nodes, double precision: speedup

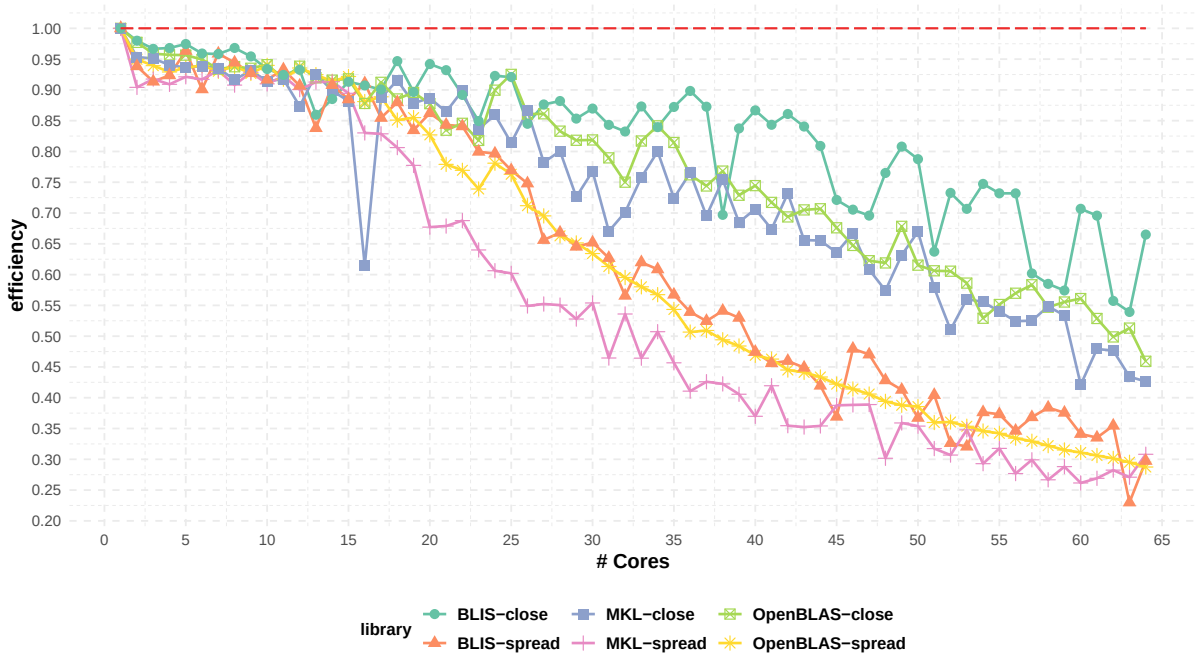


Figure 29: Core scaling: EPYC nodes, double precision: efficiency

For smaller number of cores, it seems that all the libraries with both binding policies perform almost the same, but the more the number of cores increases, the more the differences between the libraries become evident and the choosen policy matters.

Furthermore this plots confirms some results we already got in the previous section, in particular:

- the `spread` policy is always better than the `close` one
- on the EPYC nodes the `BLIS` library is the most performant one, while `OpenBLAS` and `MKL` are almost equivalent (maybe the first one is slightly better).

2.3.1.2 Single floating point operation precision

It's reasonable to expect that the results obtained with single precision will be similar to the ones obtained with double precision, due to the fact that we are considering the speedup which is a ratio between two times that should increase/decresase in the same way.

In fact the results I got are the following:

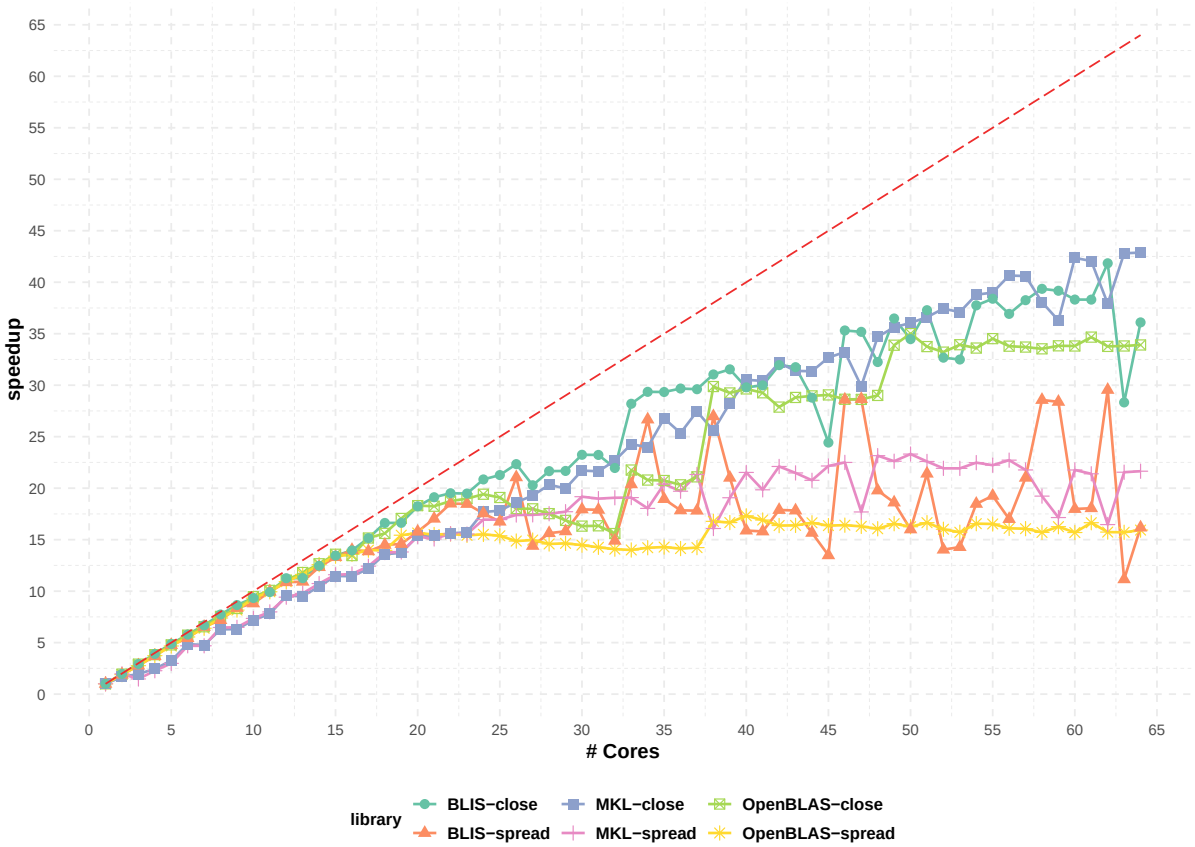


Figure 30: Core scaling: EPYC nodes, single precision: speedup

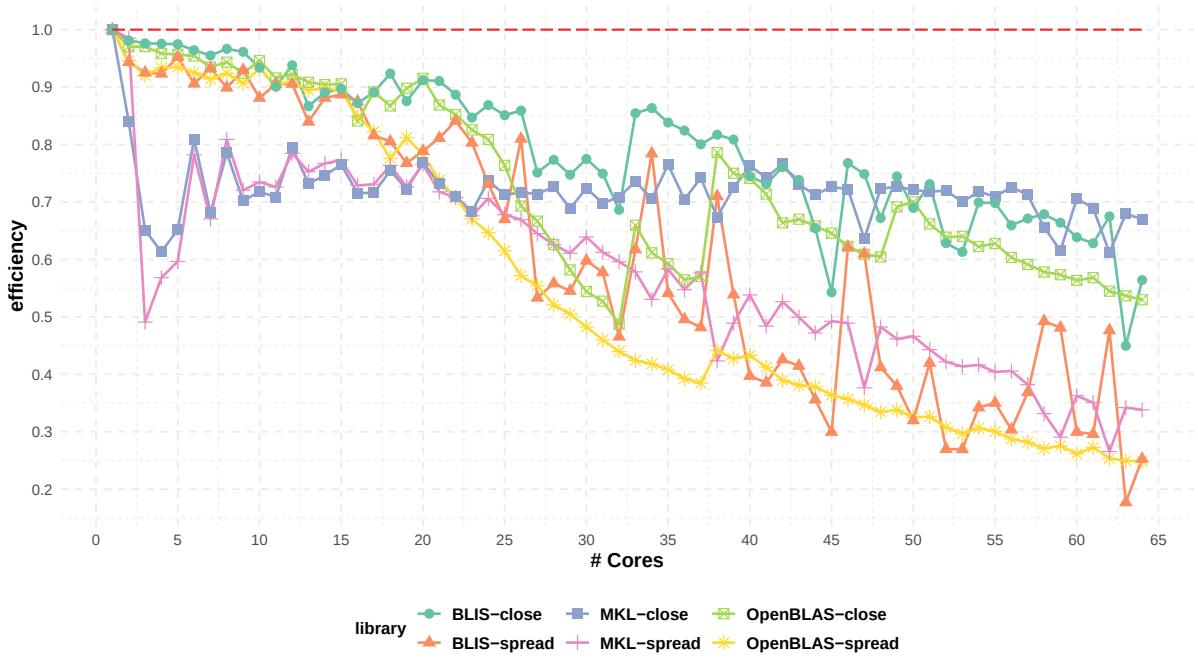


Figure 31: Core scaling: EPYC nodes, single precision: efficiency

The only notable difference is that now, for the `BLIS` library, the `close` results have become more fluctuating and instable.

2.3.2 THIN nodes

Let's now check the resluts obtained on the THIN nodes.

2.3.2.1 Double floating point operation precision

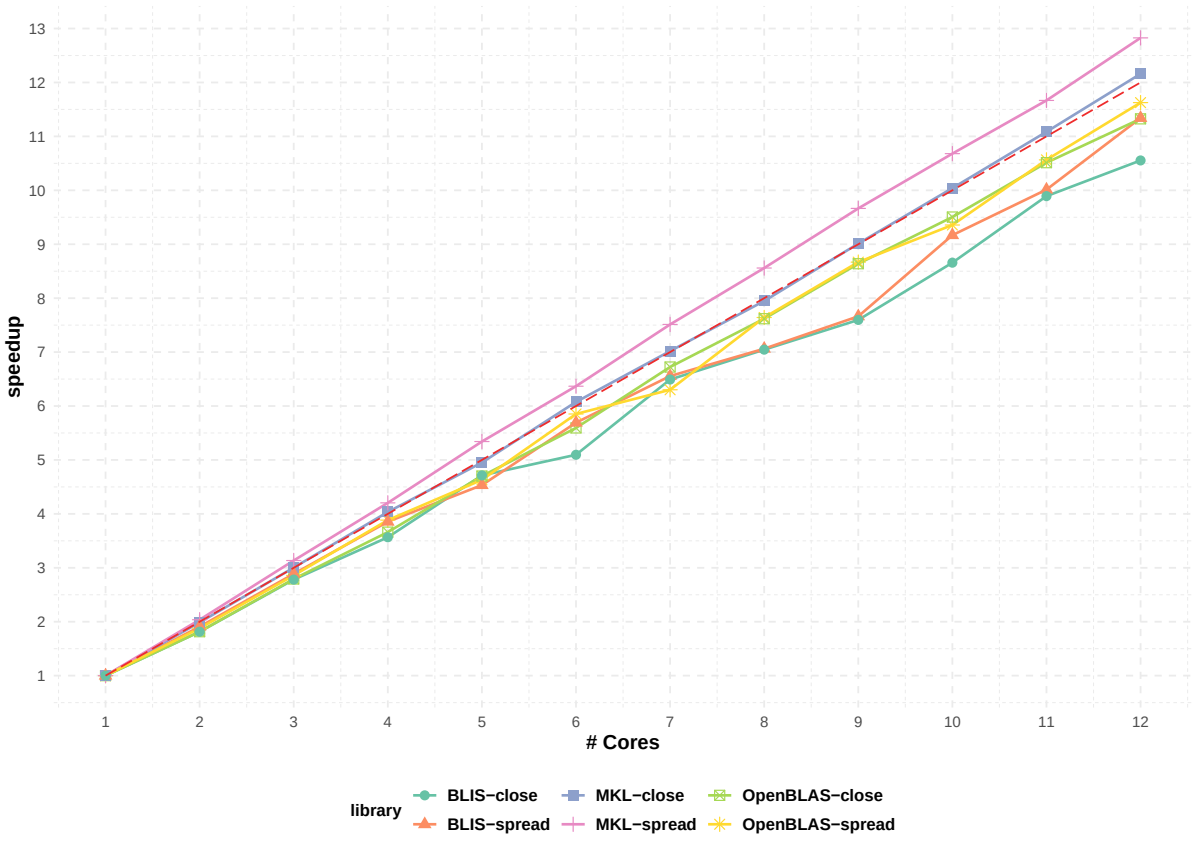


Figure 32: Core scaling: THIN nodes, double precision: speedup

As in the size scaling section, the `MKL` library is the most performant one, while `OpenBLAS` and `BLIS` are almost equivalent on the THIN nodes. The most surprising result is that this libraty reached and even exceeded the theoretical peak of speedup, which menas for example that doubling the number of core the time is more than halved.

2.3.2.2 Single floating point operation precision

Finally, let's check the results obtained with the single precision floating point operations:

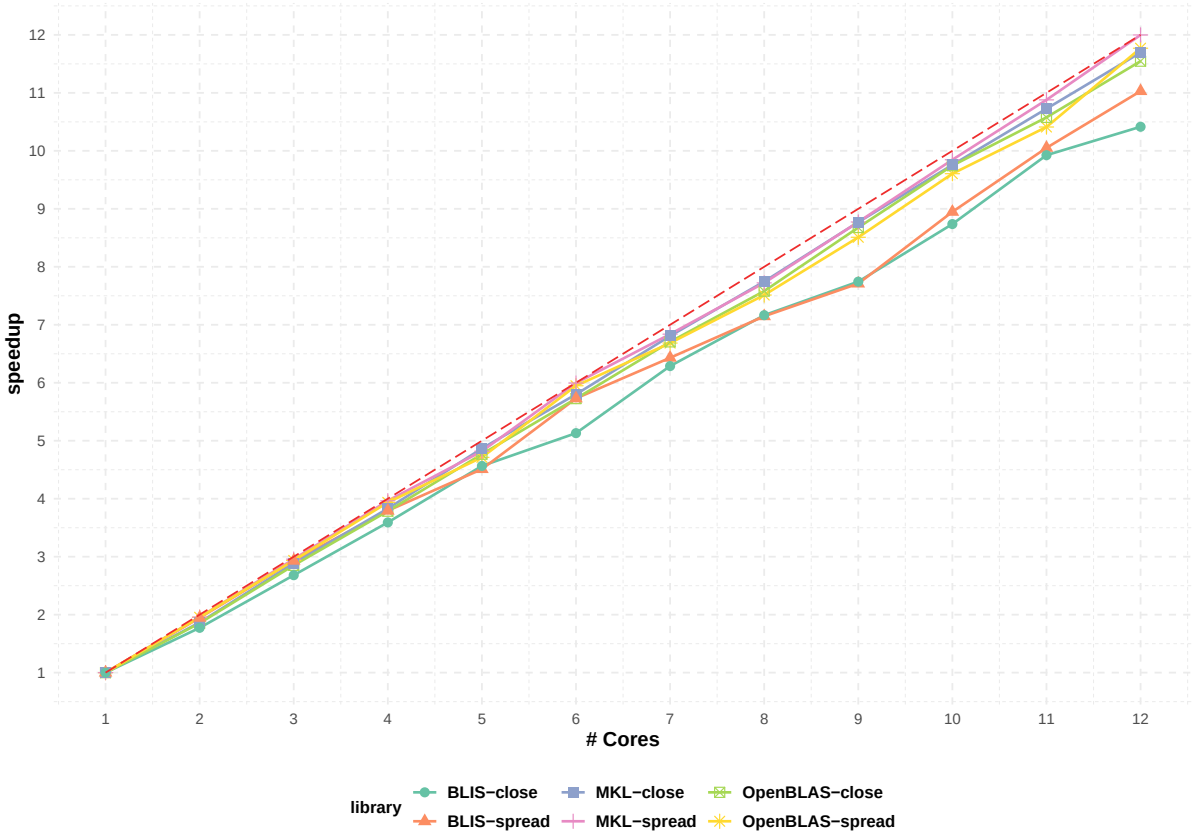


Figure 33: Core scaling: THIN nodes, single precision: speedup

In this case, all the libraries are very close to each other, but the `MKL` library is still the most performant one. None of them exceeded the theoretical peak of speedup.