University of Trieste

Foundations of High Performance Computing Course

Academic Year 2022–2023

Foundations of High Performance Computing Assignments

Davide Capone Sandro Junior Della Rovere

Contents

1 Assignment 1 **2**
1.1 Introduction . . . 2
1.2 Methodology . . . 2
1.2.1 Snapshot file format: the PBM format . . . 3
1.3 Implementation . . . 3
1.3.1 The `main.c` file . . . 3
1.3.2 The `initialize.c` file . . . 4
1.3.3 The `run.c` file . . . 4
1.3.4 The `check_cell_state.c` file . . . 4
1.3.5 The `ordered_evolution.c` file . . . 4
1.3.6 The `static_evolution.c` file . . . 5
1.3.7 The `omp_scalability.sh` file . . . 5
1.3.8 The `strong_MPI_scalability.sh` file . . . 6
1.4 Results & Discussions . . . 6
1.4.1 Code correctness . . . 6
1.4.2 OpenMP Scalability - Ordered Evolution . . . 6
1.4.3 OpenMP Scalability - Static Evolution . . . 10
1.4.4 Strong MPI Scalability - Ordered Evolution . . . 10
1.4.5 Strong MPI Scalability - Static Evolution . . . 10
1.5 Conclusions . . . 10

2 Assignment 2 **11**
2.1 Introduction . . . 11
2.2 Size scalability . . . 12
2.2.1 EPYC nodes . . . 13
2.2.2 THIN nodes . . . 15
2.3 Core scalability . . . 17
2.3.1 EPYC nodes . . . 18
2.3.2 THIN nodes . . . 21
2.4 Conclusions . . . 22

Last update on 2023–12–05T22:53:10Z.

1 Assignment 1

1.1 Introduction

This assignment consisted in implementing a parallel version of *Conway's Game of Life.* Then, two types of scalability tests have been performed:

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

The results, then, have been analyzed and some considerations are reported in this document.

1.2 Methodology

The *Conway's Game of Life* parallel program has been written in the `C` programming language, using an hybridization of the *OpenMP* and *MPI* libraries.

The game field is represented by an unsigned char array of size $k \times k$, where the first k elements correspond to the upmost row of the game field, the elements from k to $2k-1$ represent the row below it, and so on. Each game cell can contain either the value 255 (alive) or 0 (dead).

The program can be run with the following flags:

Argument	Description
`-i`	This flag is used to initialize the game on a $k \times k$ field, with random values, and then save it on a file.
`-r`	This flag is used to load the initial configuration and run the game from there.
`-k`	This flag is used to specify the size of the game field (default: 100).
`-e`	This flag is used to specify the evolution type, two options are possible: • 0 : with the ordered evolution method, the cells will be updated sequentially following row-major order; • 1 : with the static evolution method, the next state will be computed for all the cells at once, and only at the end of each iteration the update will be carried out.
`-f`	This flag is used to specify the name of the initial file: if it is used in combination with the -i file, it will specify the name of the file where the random initialization is going to be saved, otherwise (with the `-r` flag) it will specify the file where the initial configuration has to be retrieved (default: `game_of_life.pbm`).

`-n`	This flag is used to specify the number of iterations (default: 10).
`-s`	This flag is used to specify the frequency at which the snapshots of the game field will be taken, the value 0 means that only the final step will be dumped as a snapshot (default: 0).

The scalability tests have been performed on the EPYC nodes of the *Orfeo* cluster, each experiment has been repeated 10 times, measuring the execution time for each number of *MPI* tasks or *OpenMP* threads. Then the results have been grouped togheter and summarized using the mean in order to have significant results. Finally, in order to assess the scalability, two indexes have been considered:

1. **Speedup**, defined as:

$$S(n,t) = \frac{T_s(n)}{T_p(n)} \tag{1}$$

   - the speedup measures how much faster the parallel algorithm is compared to the sequential one;
   - p is the number of used cores;
   - $T_s(n)$ is the time taken by the sequential algorithm;
   - $T_p(n)$ is the time taken by the parallel algorithm.

2. **Efficiency**, defined as:

$$E(n,t) = \frac{S(n,t)}{p} \tag{2}$$

   - the efficiency measures the percentage of time for which a processor is utilized effectively.

The theorical maximum speedup is p, while the maximum efficiency is 1.

1.2.1 Snapshot file format: the PBM format

It has been decided to perform some minor modifications to the `read_write_pgm_image.c` program in order to change the file format of the snapshots from *PGM* to *PBM*. The *PBM* format uses 1 bit per pixel, which is perfectly adequate to our case because every cell (that is a pixel in the image) may just be dead or alive. The new program is called `read_write.c`

1.3 Implementation

1.3.1 The `main.c` file

The `main.c` file is a slightly modified version of the `get_args.c` file. It parses the command line arguments (already discussed in subsection 1.2) and sets the corresponding variables. The program also handles missing arguments by printing an error message and then stops the execution.

The main program also calls the `initialize()` function when the `-i` flag is specified, or alternatively the `run()` function when the `-r` flag is given.

1.3.2 The `initialize.c` file

This code is executed when the main.c program receives the `-i` argument. A $k \times k$ game field is initialized randomly (as discussed in subsection 1.2) and each cell has a 15% chance of being alive; the initialization can be of two types:

1. in the **serial version** an unsigned char array is allocated with the `malloc()` function, then each cell is filled with 255 if `rand()%100` is less than 15 or 0 otherwise, using a simple loop. Then, the game field is dumped on a *PBM* file named accordingly to the value of the `-f` flag;

2. the **parallel version** is more or less the same, apart from the fact that the array is divided in n chunks and the work is distributed among the n *MPI* processes. Then, the game field is dumped on a *PBM* file named accordingly to the value of the `-f` flag.

1.3.3 The `run.c` file

This code is executed when the `main.c` program receives the `-r` argument. It checks what kind of evolution is required by the user (specified with the `-e` flag) and calls the appropriate functions.

1.3.4 The `check_cell_state.c` file

This code implements a function to check if a cell should live or die. It basically sums the values in the neighborhood of the cell, then the sum is divided by the maximum value present in the game field, and if the result is 2 or 3, the cell should live, otherwise it should die.

1.3.5 The `ordered_evolution.c` file

This is the implementation of the ordered evolution method described in subsection 1.2. It starts by allocating the space needed to store the initial configuration and then reads it from the file (which has been generated by the `initialize.c` program).

Then it starts an external for loop, which is executed n times; for each iteration another internal loop, which is executed $k \times k$ times (i.e. the size of the game field) will evaluate the next cell's state with the `check_cell_state()` function, applying directly the updates in the matrix. At the end of the outer loop an if condition checks if it is required to take a snapshot, if so the current state of the matrix is saved in the `snaps/` directory (by calling the `write_pbm()` function).

In this case the parallelization with *MPI* would not be effective due to the nature of the procedure. The code is parallelized only with *OpenMP*.

1.3.6 The `static_evolution.c` file

This is the implementation of the static evolution method described in subsection 1.2. First of all it allocates the space needed to store the initial configuration and then reads it from the file that has been generated by the initialize.c program.

This procedure is clearly parallelizable, since it consits in performing some operations individually for each cell of the matrix, and then updating it all at once at the end.

The only difference between this implementation and the previous one is that the result obtained by calling the `check_cell_state()` function is stored in a temporary matrix, which becomes the new state of the game at the end of the inner for loop.

As for the parallelization procedure, each *MPI* process is given a chunk of the matrix of size $\frac{k \times k}{n^o \ MPI \text{ processes}}$ [1]. The main steps of the procedure are summarized as follows:

1. each process allocates enough space to store the initial game field and stores it;
2. every process computes the size and the starting index of the chunk on which it has to work;
3. each *MPI* process allocates an additional array of the same size as the chunk, to temporarily store the result of the computation;
4. the outer for loop iterates $n \times n$ times, at each step it:
   i. waits for all the other processes to finish the previous iteration (`MPI_Barrier`);
   ii. executes an inner for loop, which iterates $\frac{k \times k}{n^o \ MPI \text{ processes}}$ (size of the chunk) times, which in turn:
      - evaluates the next state of the cells with the `check_cell_state()` function,
      - stores the results in the temporary array;
   iii. collects the results of every process with the `MPI_Allgatherv()` function);
   iv. checks if it's time to save the snapshot, if so it saves a dump in the `snaps/` folder;
5. after the main loop the final snapshot is taken, and all the allocated memory is freed.

1.3.7 The `omp_scalability.sh` file

This slurm script has been created to perform the experiments relatively to the *OpenMP* scalability.

[1] $\frac{k \times k}{n^o \ MPI \text{ processes}} + 1$ if $k \times k \mod n^o \ MPI$ processes ≥ 0.

The request is to fix the number of *MPI* tasks to 1 per socket, and report the behaviour of the code when the number of threads per task is increased from 1 up to the number of cores present on the socket (64 in the case of EPYC nodes). In order to fix the number of *MPI* tasks to 1 per socket, the mpirun command is executed with the `-map-by node -bind-to socket` options. The number of used nodes is 1. The script is ran two times to perform two different tests, one with a game field of size 10000×10000, and the other with size 20000×20000.

1.3.8 The `strong_MPI_scalability.sh` file

This slurm script has been created to perform the experiments relatively to the strong *MPI* scalability.

The request is to show the run-time behaviour when the number of *MPI* tasks increases, fixing the size of the game field. In order to do that, the `mpirun` command is executed with the `-map-by core` option. Two EPYC nodes have been used, so the number of *MPI* processes goes from 1 to 128. To isolate the effect of the number of *MPI* processes from the effect of the number of threads, the number of threads has been set to 1 for each *MPI* process by calling `export OMP_NUM_THREADS=1`. Also in this case the script is ran two times to perform two different tests, one with a game field of size 10000×10000, and the other with size 20000×20000.

1.4 Results & Discussions

1.4.1 Code correctness

In order to test the correctness of *Game of Life*'s implementation, the program has been run with a small field size (5×5) and the results have been checked manually for a few steps.

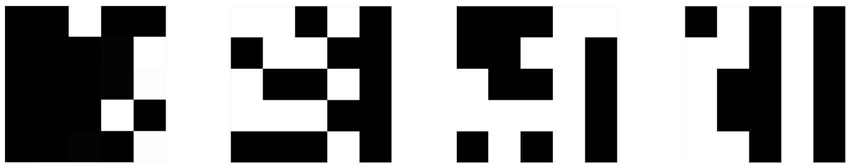

Figure 1: Ordered evolution on a 5×5 game field.

Since the code proved to work correctly, the following activity consists in performing the scalability tests.

1.4.2 OpenMP Scalability - Ordered Evolution

As expected, Figure 3 shows that the execution time decreases as the number of *OpenMP* threads increases. The execution time quickly approaches an almost constant

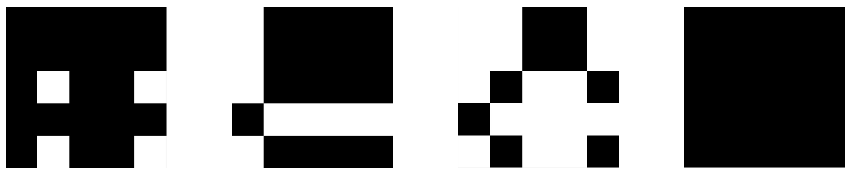

Figure 2: Static evolution on a 5 × 5 game field.

value, meaning that after a certain number of *OpenMP* threads (in particular 30-40), the benefit of multi-threading is not significant.

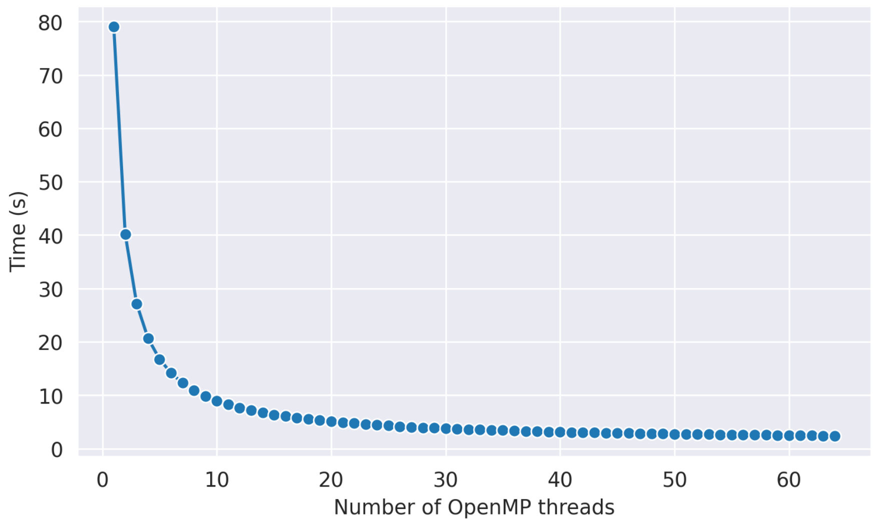

Figure 3: Execution time of the ordered evolution method, for each number of *OpenMP* threads, for different game field sizes.

Figure 4 shows a sublinear trend for the speedup, this is probably due to the communication and synchronization overhead introduced at higher core counts. Furthermore, *Amdahl's Law* states that the speedup of a program is limited by the proportion of the code that cannot be parallelized.

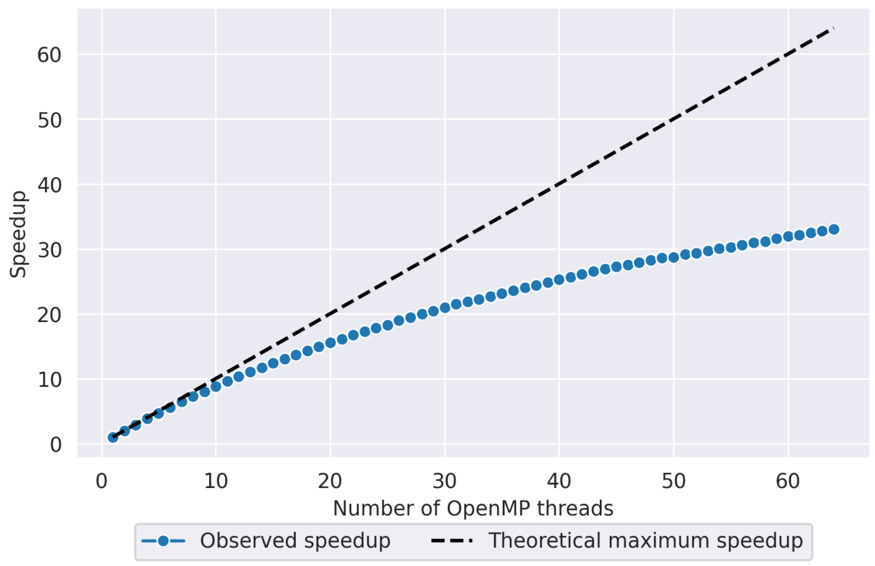

Figure 4: Speedup of the ordered evolution method, for each number of *OpenMP* threads, for different game field sizes.

Figure 5 suggests that the efficiency decreases quite fast as the number of *OpenMP* threads increases. This is probably due to the fact that the workload is not evenly distributed among the cores (`OMP_PLACES=cores`), some of them may finish their tasks early and wait for others to complete causing a decrease in efficiency. Beyond a certain point, adding more cores might not provide substantial benefits. Furthermore, as mentioned in subsection 1.2, the ordered evolution procedure has a nature that does not allow for great improvements with parallelization.

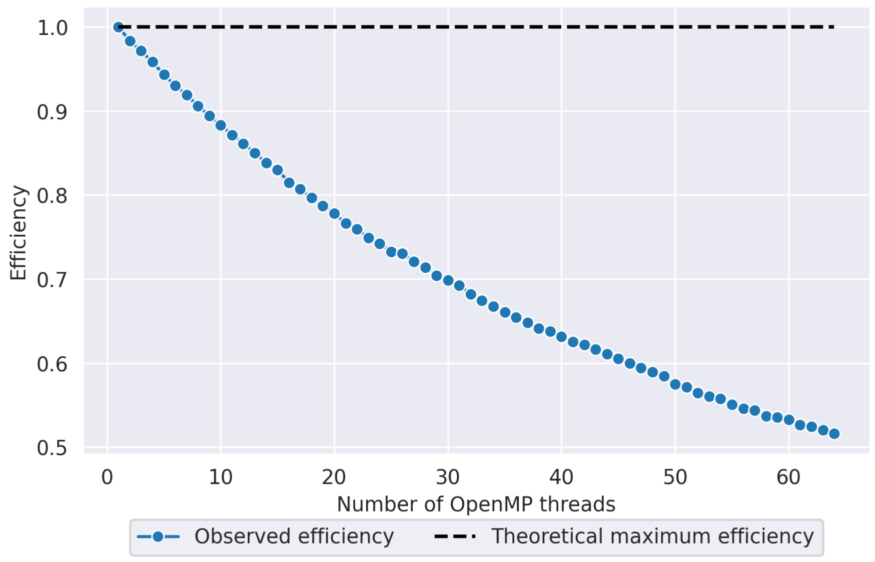

Figure 5: Efficiency of the ordered evolution method, for each number of *OpenMP* threads, for different game field sizes.

1.4.3 OpenMP Scalability - Static Evolution

1.4.4 Strong MPI Scalability - Ordered Evolution

1.4.5 Strong MPI Scalability - Static Evolution

1.5 Conclusions

The Orfeo cluster gave many problems in the last days before the delivery of the final report. The only test that has been run succesfully is the one reported. The same test had to be run with a different game field size for comparison, and also all the other tests have to be run with two different game field sizes, but since the date of the delivery has come and the connection to the Orfeo cluster doesn't seem to work properly anymore for all the group members, it has been decided to deliver the report as it is, incomplete but with promising results, because the code works correctly.

2 Assignment 2

2.1 Introduction

The goal of this assignment is to compare two math libraries: *Intel's Math Kernel Library* (*MKL*) and *OpenBLAS*. In order to perform this study, it is required to evaluate the performance of the aforementioned libraries on matrix multiplications using an already provided parallel program: `gemm.c`.

Two scalability tests have been performed on both EPYC and THIN nodes of *Orfeo*, using different floating point number precisions and core binding policies:

- **size scalability**: analyze the scaling of the *GEMM* calculations for an increasing matrix size, keeping a fixed number of cores.
- **core scalability**: analyze the scaling of the *GEMM* calculations for an increasing number of cores, keeping fixed the matrix size.

The `gemm.c` code has been modified in order to only retrieve the information needed for the analysis: in particular the *GFlops* (numbers of floating point operations executed in one second by the CPU) have been measured for the size scalability and, for the core scalability analysis, the execution time has been considered.

Each test has been executed 10 times for each configuration, then the results have been grouped togheter and summarized using the mean in order to have significant results.

2.2 Size scalability

All the results are compared to the theoretical peak performance of the *Orfeo* nodes. This quantity can be computed as:

$$Flops_{peak} = n_{cores} \times frequency \times (\frac{FLOP}{cycle}) \tag{3}$$

1. **AMD EPYC 7H12**
   - $n_{cores} = 64$
   - $frequency = 2.6$ GHz
   - AMD Epyc 7H12 can reach at most $\frac{FLOP}{cycle} = 32$ operations per second regarding the floating point (single precision) operations and $\frac{FLOP}{cycle} = 16$ operations per second for the double precision operations.
   - the theoretical peak performance is then $64 \times 2.6 \times 32 = 5324.8$ GFlops for single precision and $64 \times 2.6 \times 16 = 2662.4$ GFlops for double precision.
2. **Intel Xeon Gold 6126**
   - $n_{cores} = 12$
   - $frequency = 2.6$ GHz
   - Intel Xeon Gold 6126 can reach at most $\frac{FLOP}{cycle} = 32$ operations per second regarding the floating point (single precision) operations and $\frac{FLOP}{cycle} = 16$ operations per second for the double precision operations.
   - the theoretical peak performance is then $24 \times 2.6 \times 32 = 1996.8$ GFlops for single precision (this result is coherent with the data provided in the course material) and $24 \times 2.6 \times 16 = 998.4$ GFlops for double precision.
   - the theoretical peak performance with the *Max Turbo Frequency* (3.7 GHz, frequency at which the processor is capable of operating using *Intel Turbo Boost Technology*) is computed for analytical purposes: $24 \times 3.7 \times 32 = 2841.6$ GFlops for single precision and $24 \times 3.7 \times 16 = 1420.8$ GFlops for double precision.

2.2.1 EPYC nodes

Figure 6 shows the results achieved for the size scalability tests, on EPYC nodes, using single precision floating point numbers as members of the matrices employed in the multiplications. None of the libraries reach the theoretical peak performance, this may be due to several factors such as communication overhead and memory bandwidth. It might be beneficial to explore additional factors such as communication patterns, cache utilization, and memory bandwidth utilization in order to understand the reasons behind this and to improve the performance of the code. *OpenBLAS* shows better results with bigger matrices, while *MKL* seems better with small matrices: this could be attributed to the specific optimizations and parallelization strategies employed by each library. An improvement could be to use a different library depending on the size of the matrix.

Changing the binding policy doesn't seem to do much, the performance of the code is not sensitive to how the computational tasks are spread or kept close to each other in terms of physical core placement.

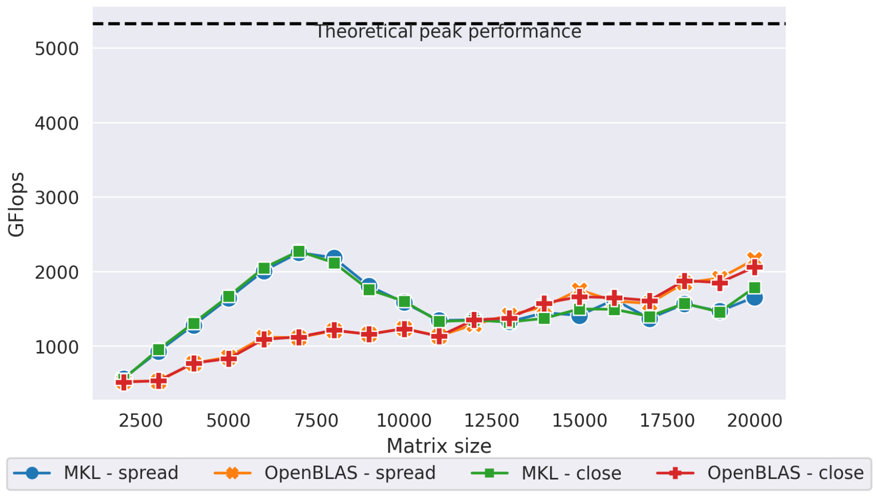

Figure 6: Size scalability on EPYC nodes with single precision.

Figure 7 shows the results relative to the use of double precision floating point numbers. The same considerations as before hold, but in this case *MKL* doesn't show a better performance with small matrices. The implementation of MKL may be optimized differently for single and double precision, for smaller and bigger matrices.

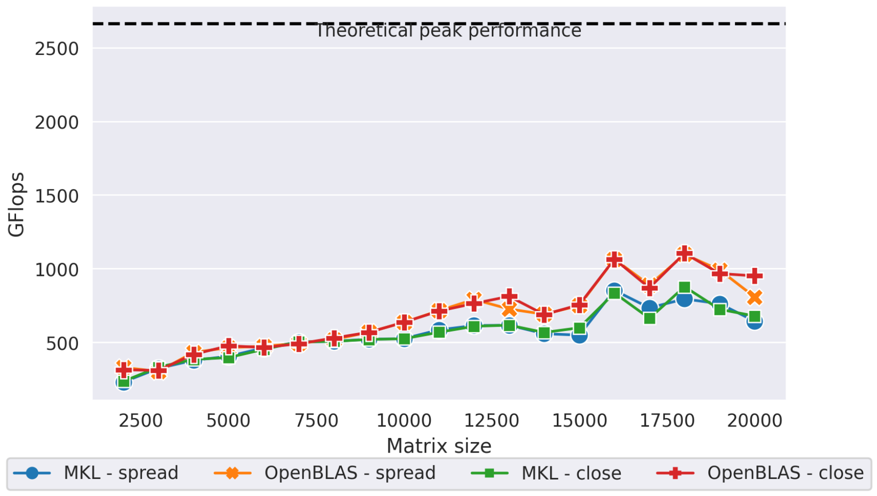

Figure 7: Size scalability on EPYC nodes with double precision.

2.2.2 THIN nodes

As an optional step in the assignment, also THIN nodes have been used to asses the scalability of the two libraries. Figure 8 displays the results for single precision. The performance here seems to go over the theoretical peak, this may be due to the fact that the processor went into turbo mode. The observed performance is even higher than the theoretical peak performance in the case of *Turbo Boost*, this may be due to caching effects. The binding policy seems to have a bigger impact on these nodes, for both libraries: the performance is better with the close policy, when the processes are assigned to phisically close cores.

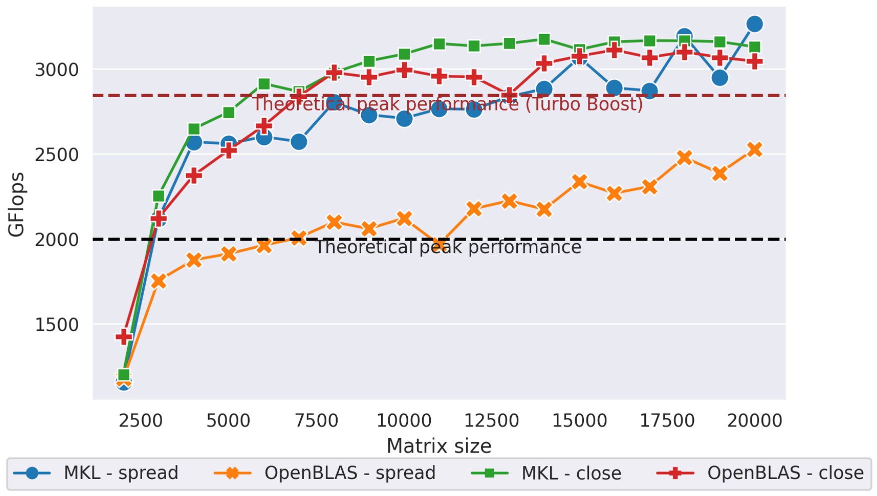

Figure 8: Size scalability on THIN nodes with single precision.

Figure 9 portraits the results in the case of double precision floating point numbers. The same considerations as before hold. On THIN nodes, the MKL library seems to perform better than *OpenBLAS*, independently on the matrix size and the number of used cores. This may be due to the fact that MKL is an Intel library and maybe it's optimized for their processors.

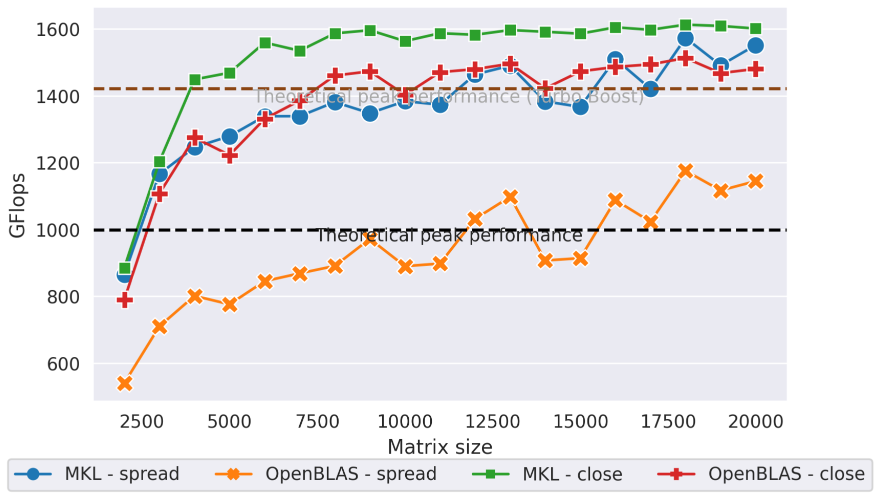

Figure 9: Size scalability on THIN nodes with double precision.

2.3 Core scalability

To analyze the results, two indexes are going to be considered:

1. **Speedup**, defined as:

$$S(n,t) = \frac{T_s(n)}{T_p(n)} \tag{4}$$

   - the speedup measures how much faster the parallel algorithm is compared to the sequential one;
   - p is the number of used cores;
   - $T_s(n)$ is the time taken by the sequential algorithm;
   - $T_p(n)$ is the time taken by the parallel algorithm.

2. **Efficiency**, defined as:

$$E(n,t) = \frac{S(n,t)}{p} \tag{5}$$

   - the efficiency measures the percentage of time for which a processor is utilized effectively.

The theorical maximum speedup is p, while the maximum efficiency is 1.

2.3.1 EPYC nodes

Figure 10 shows the speedup of the parallel program for an increasing number of cores, on EPYC nodes with single precision. The trend is clearly sublinear. By looking at the plot, it seems that after the 16th core the accelleration in the speedup starts decreasing, this is probably due to the communication and synchronization overhead introduced at higher core counts. Furthermore, *Amdahl's Law* states that the speedup of a program is limited by the proportion of the code that cannot be parallelized.

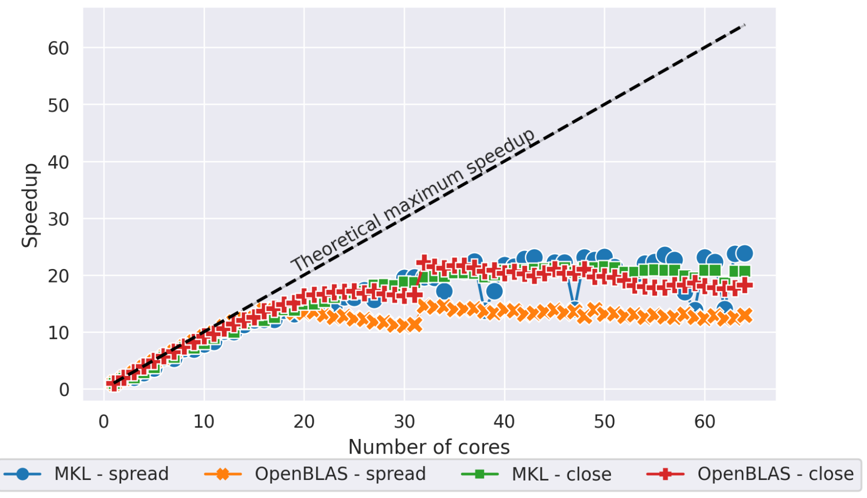

Figure 10: Speedup on EPYC nodes with single precision.

Figure 11 represents the efficiency measure of the EPYC CPU, with single floating point precision. The efficiency tends to decrease while the number of cores grows, this is probably due to the fact that the workload is not evenly distributed among the cores, some of them may finish their tasks early and wait for others to complete causing a decrease in efficiency. Beyond a certain point, adding more cores might not provide substantial benefits. The steeper decrease in efficiency at higher core counts suggests that there may be a critical point where the overhead becomes too significant or the workload distribution becomes less balanced. This could be due to factors like increased contention for shared resources or increased communication costs. Optimizing communication patterns, load balancing, and minimizing synchronization points may improve the scalability of this code. Based on the results it can be said that none of the libraries performs significantly better than the other in terms of speedup and efficiency, when it comes to the core scaling on EPYC nodes. The *OpenBLAS* library shows better performances with the close binding policy, when the tasks are assigned to physically close cores.

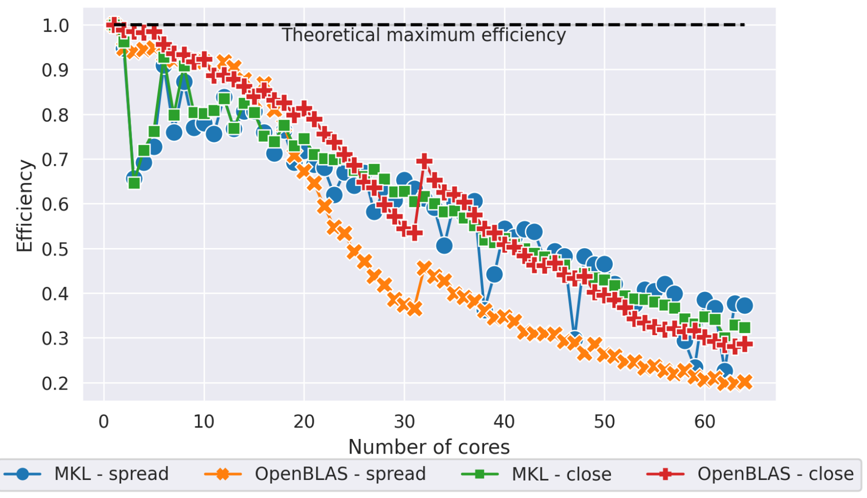

Figure 11: Efficiency on EPYC nodes with single precision.

Figure 12 represents the observed speedup on EPYC nodes with double precision.

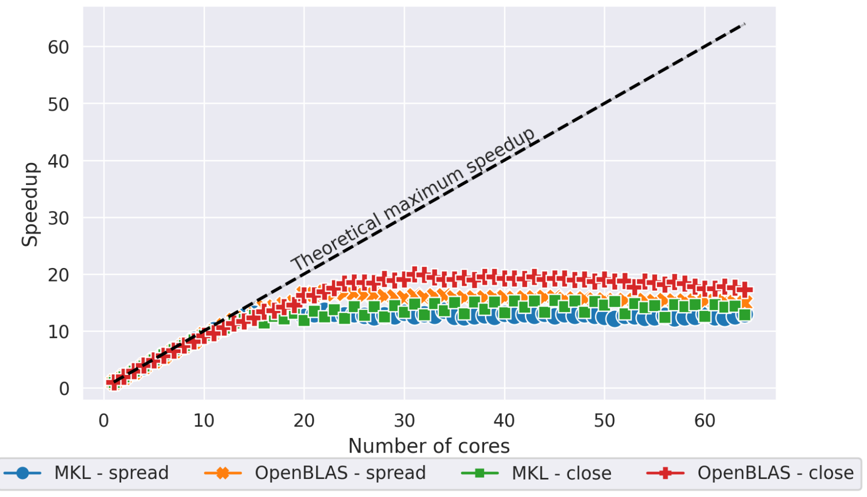

Figure 12: Speedup on EPYC nodes with double precision.

Figure 13 plots the observed efficiency on EPYC nodes with double precision. The same considerations as before apply also in the case of double precision. *OpenBLAS* with the close binding policy scales slightly better on double precision.

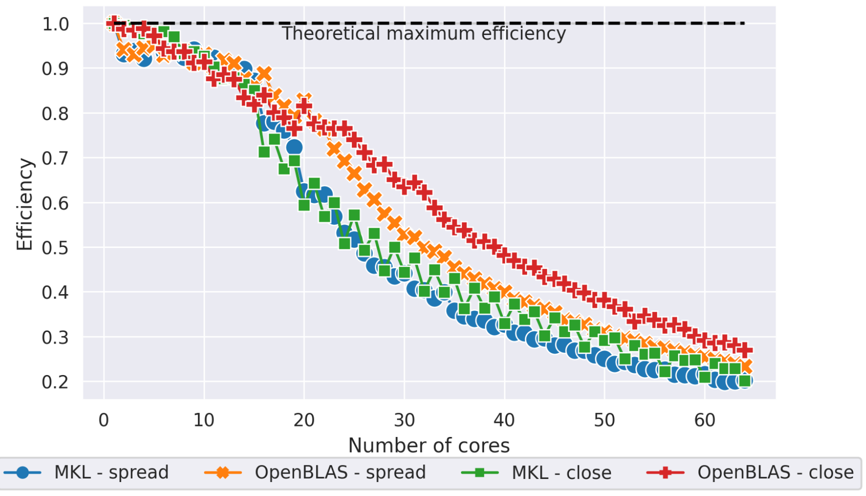

Figure 13: Efficiency on EPYC nodes with double precision.

2.3.2 THIN nodes

The two libraries scale better on THIN than on EPYC nodes when it comes to core scalability. Since the speedup is almost linear it has been decided to not proceed further with the analysis of efficiency.

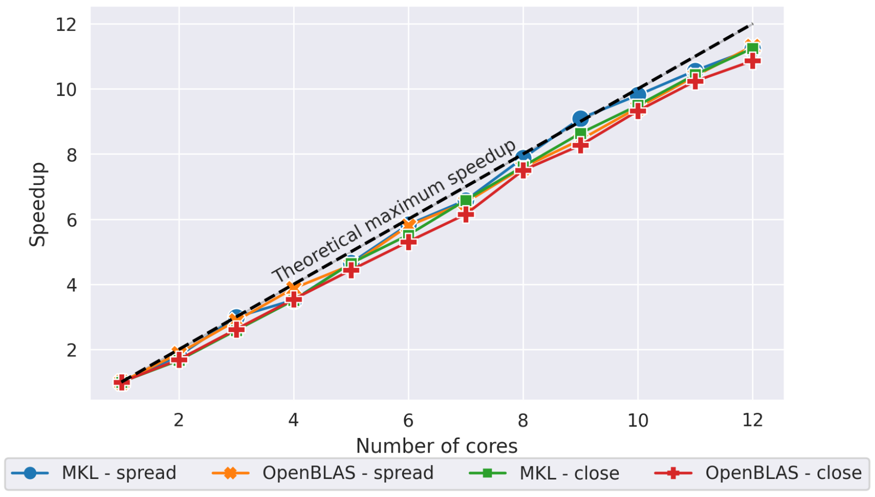

Figure 14: Speedup on THIN nodes with single precision.

For both single and double precision MKL shows the best results, coherently with what can be seen for the size scaling on THIN nodes, even going over the theoretical maximum speedup, this may be due to caching effects.

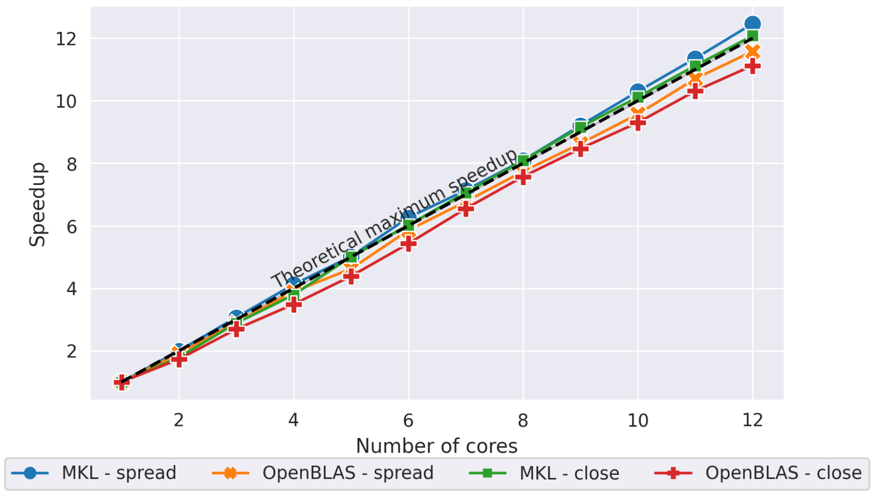

Figure 15: Speedup on THIN nodes with double precision.

2.4 Conclusions

To conclude, the assignment showed that MKL seems to scale better on THIN nodes, for both size and core scalability, with the close binding policy yielding the best outcomes. In any case, this processor seems to be the optimal choice for the *gemm.c* parallel code, although all of the tested libraries follow very similar trends in all the experiments for both kinds of Orfeo nodes.

None of the two libraries reaches the theoretical peak performance, nor follows a linear speedup, and neither of the two libraries substantially outperforms the other on EPYC nodes, with one being sometimes better and sometimes worse.