Foundations of High Performance Computing

Final assignment

Marco Sicklinger

September 2023

1 Exercise 1

1.1 Introduction

The purpose of this exercise is to implement a parallel version of the Conway's Game of Life exploiting an hybrid MPI/OpenMP approach.

The Conway's Game of Life is a zero-player game that consists of an infinite, two-dimensional orthogonal grid of square cells, each of which can be in one of two states (dead or alive) and interacts with its eight neighboring cells according to a specific set of rules. There exist many variants of this cellular automation: the rules followed here can be found at the Life's Wikipedia [1] page.

1.2 Methodology

1.2.1 Initialization

The initial Life's grid is generated in such a way that each cell can be in the dead or alive state with equal probability.

In this work, the grid initialization is performed in a serial way. Successively, it is stored as a `pgm` file for further use.

1.2.2 Padding

As stated above, the two-dimensional Life's grid is unbounded. This feature is addressed through the implementation of periodic boundary conditions. The approach chosen to deal with this problem is to model the grid as a torus [2] by padding the original grid with two more rows and columns, acting as copies of those at the opposite edges of the grid.

In this way, it is possible to avoid the modulo operation or conditional blocks that would require evaluation for each position of the grid at each iteration of the game. The price to pay for padding is an extra memory consumption that amounts to $2(k_1 + k_2 + 2)S_{\text{type}}$, with S_{type} being the size of the type used to represent cell' state and k_1 and k_2 are the number of rows and columns, respectively.

1.2.3 Parallel Approach

The total workload is divided by rows: each MPI process is assigned a portion of the grid with number of rows equal to the integer division between the total number of rows and the number of MPI tasks. If the remainder of this operation is not zero, the additional rows are assigned to the MPI process with highest rank. Notice that the padding mechanism described above must be exploited for each of the grid portions assigned to the MPI tasks since adjacent sub-grids are not independent: while there is no conceptual difference regarding the purpose of the halo columns, the halo rows serves as copies

of the first and last rows of the bordering sub-grids. Only the top and bottom rows of the first and last MPI processes, respectively, actually simulate the PBC.

Additionally, each MPI task spawns a number of threads responsible for operating on the individual cells by performing the necessary computations required to follow the evolution's rules. Specifically, each thread will count the number of alive neighbors for each cell in the portion of the corresponding MPI process' sub-grid assigned to it.

1.2.4 Message Passing Approach

In order to implement the parallel approach described above, the message passing strategy adopted for the evolution is a combined send and receive. Each process exchanges its top row with the "previous"[1] one and its bottom row with the "successive" one, creating a double exchange of information between processes. The received data, that are the bordering rows of the neighboring processes, are stored in the halo rows coming from padding the sub-grid. Moreover, The first and last MPI processes are suitably defined as the successive and previous of one another, respectively.

Additionally, MPI functions are also used to scatter the chunks (local grids) from the root process to the others and to gather the local sub-grids into the master process when it is requested to save the life's snapshot.

1.3 Implementation

1.3.1 Framework

This exercise is implemented in the C++ programming language. In order to stay in line with the course lecture, the MPI C functions are used instead of the C++ bindings to the library.

The core of the game is represented by the `Life` class. It contains all the attributes and methods for running the game's evolution. The `utils.cpp` source file contains the methods for the grid random generation and saving; `game.cpp` contains the `get_args` function (for parsing the command line arguments) and `main` function, which calls the generation of the initial grid and the evolution function.

1.3.2 Domain Decomposition

In this section, the implementation through domain decomposition of the parallel approach described above (1.2.3) will be presented.

First, in the `Life`'s constructor, the MPI processes initialize the suitable attributes with the values provided for the MPI initiliazation, namely their own rank, the number of processes and the ranks of the previous and successive processes; with this information they compute the local number of rows (the number of rows they are going to operate on) and the corresponding quantities for the padded grid. Then, the master process (`rank=0`) requests the grid from memory, saved as a PMG image in the `snapshot` directory, and sends the local grids (non-padded) to the other processes, keeping the first chuck of rows for itself. The sending is performed via `MPI_Scatterv` function. The variable-message-size version of the `MPI_Scatter` collective subroutine was chosen due to the fact that the MPI task with highest rank may be assigned to a local grid with a different number of elements from the others. Finally, all the processes initialize their own local padded grids by calling the `initializeObs` function.

The code snippet below reports the implementation of the scattering of the chucks to the the corresponding MPI processes.

[1] By "previous" it is meant, the process with immediately lower rank.

```
        // master process computes information
        // needed to send the right chunk of grid
        // to the right process
        if (rank == 0) {
            globalState = new int[rows*cols];
            read_state(filename, globalState, lifeSize);

            send_counts = new int[nTasks];
            displs = new int[nTasks];

            int offset = (rows%nTasks)*cols;
            for (int r = 0; r < nTasks; r++) {
                int add_offset = r == nTasks - 1 ? 1 : 0;
                send_counts[r] = localSize + offset*add_offset;
                displs[r] = localSize*r;
            }
        }

        // scatter the sub-grids to corresponding processes
        MPI_Scatterv(globalState, send_counts, displs,
                    MPI_INT,
                    localState, localSize,
                    MPI_INT,
                    0,
                    MPI_COMM_WORLD);
```

1.3.3 Implementation of the Parallel Approach: Message Passing

During the evolution, each process must compute the halo columns and exchange the rows bordering with other processes' sub-grids. The first task is simply done by a `for` loop iterating through the row index, while the second is performed exploiting the MPI function `MPI_Sendrecv` as shown below.

```
    void Life::haloExchange(){
    MPI_Sendrecv(
        localObs + rows*localColsHalo, localColsHalo, MPI_INT, upRank, 0,
        localObs, localColsHalo, MPI_INT, loRank, 0,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
    MPI_Sendrecv(
        localObs + localColsHalo, localColsHalo, MPI_INT, loRank, 1,
        localObs + (localRowsHalo - 1)*localColsHalo, localColsHalo, MPI_INT, upRank
            ↪ , 1,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE
    );
}
```

Each process sends the first and last rows of the actual local grid (second and second to last in the padded one, respectively) to the previous and next processes, respectively, and it receives from them the last and first rows of the actual grid.

1.3.4 Implementation of the Parallel Approach: Multi-Threading

In the `staticStep` function, OpenMP threads are spawned to work simultaneously on the loop iterating over the whole local grid in order to count the alive neighbors of each cell. Since each thread performs the same computational work, the static assignment is used:

```
#pragma omp parallel for schedule(static)
for (int x = 1; x <= localRows; x++) {
    for (int y = 1; y <= cols; y++){
        // local population count is computed
        int local_population = census(x, y);
        // game's rules evaluation
        localObsNext[x*localColsHalo + y] = !((localObs[x*localColsHalo + y] ==
            ↪ ALIVE && local_population == 2) || (local_population == 3));
    }
}
```

In the code snippet above, the `census` function just counts the number of dead cells and subtracts this number to the total number of neighbors (that is 8).

1.3.5 Implementation of the Parallel Approach: Snapshot Saving

Life's snapshot are taken with a frequency chosen by the user: when the saving condition is met (compilation is done with the appropriate flag, `SSAVE`), the sub-grids are gathered back into the master process (using the `MPI_Gatherv` collective routine) which then writes the whole grid onto a PGM image.

```
// master process sets up the quantities
// and the necessary to receive the local grids
// in the right place in globalState
if (rank == 0) {
    globalState = new int[rows*cols];
    recv_counts = new int[nTasks];
    displs = new int[nTasks];

    int offset = (rows%nTasks)*cols;
    for (int r = 0; r < nTasks; r++) {
        int add_offset = r == nTasks - 1 ? 1 : 0;
        recv_counts[r] = localSize + offset*add_offset;
        displs[r] = localSize*r;
    }
}

// gather local grids in global buffer on master process
MPI_Gatherv(
    localState, localSize,
    MPI_INT,
    globalState, recv_counts, displs,
    MPI_INT,
    0,
    MPI_COMM_WORLD);
```

1.4 Results

In the case of the static evolution, scalability measurements were conducted to assess the performance of the hyprid MPI/OpenMP approach:

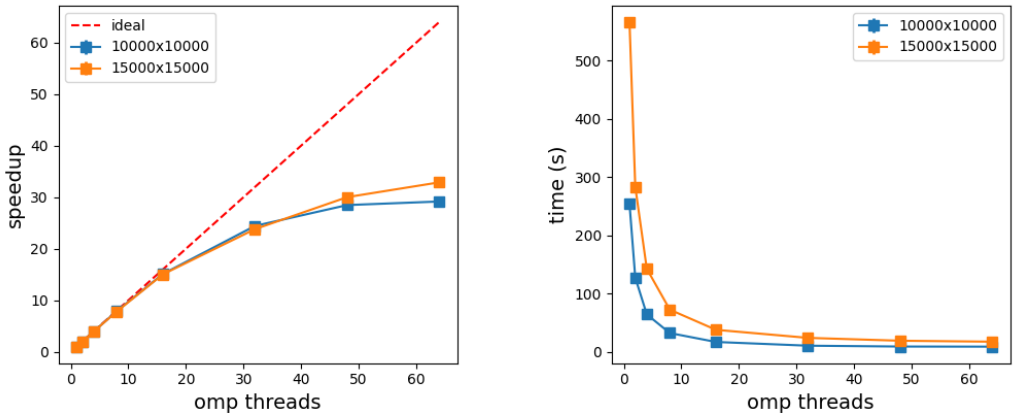
- OpenMP scalability;
- Strong MPI scalability;
- Weak MPI scalability.

The performance comparison is executed based on the speedup, a quantity computed as the ratio between the execution time of a single worker and that of multiple workers, and the run-time. Each simulation is 500 iterations long. For the two MPI studies, two kinds of experiments were conducted: one with no intermediate system dumps and one with a snapshot saving frequency of 1 snapshot every 25 iterations.

1.4.1 OpenMP scalability

The OpenMP scalability study was conducted allocating a single task on a node and increasing, at each run, the number of threads from 1 to 64. They were run setting `OMP_PROC_BIND=close` and mpirun option `--map-by socket`. Two grid sizes were tested: $k_1 = 10000$ and $k_2 = 15000$. The speed-up and run-time results are shown below.

From figure 1a, it is noticeable that the speed-up keeps increasing up to 64 threads. However, for numbers of threads greater than 16, the speed-up behavior stops scaling almost linearly, making evident the presence of an overhead for larger numbers of threads. The results are consistent across the different tested sizes.



(a) Speed-up behavior.

(b) Run-time behavior.

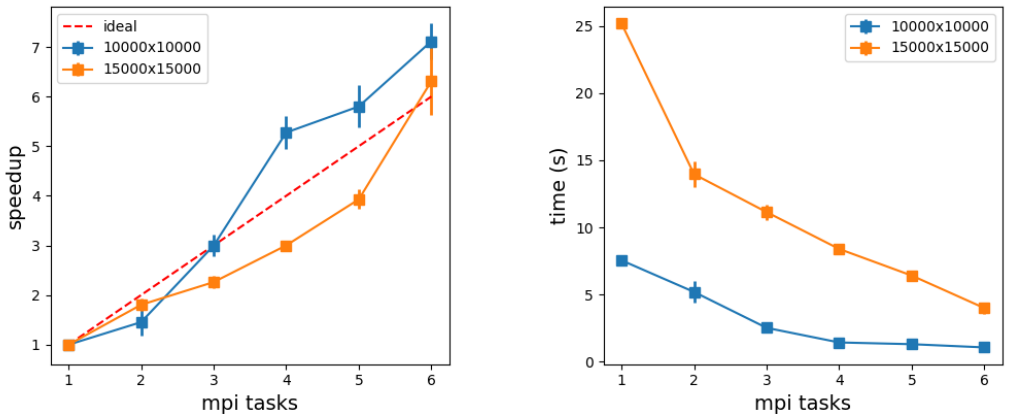
Figure 1: OpenMP scalability curves.

1.4.2 Strong MPI scalability

The MPI strong scalability study is executed by keeping the size of the grid constant across various runs while increasing the number of mpi processes. Again, sizes $k_1 = 10000$ and $k_2 = 15000$ where chosen.

First, the results corresponding to a single system dump at the end of the evolution will be shown.

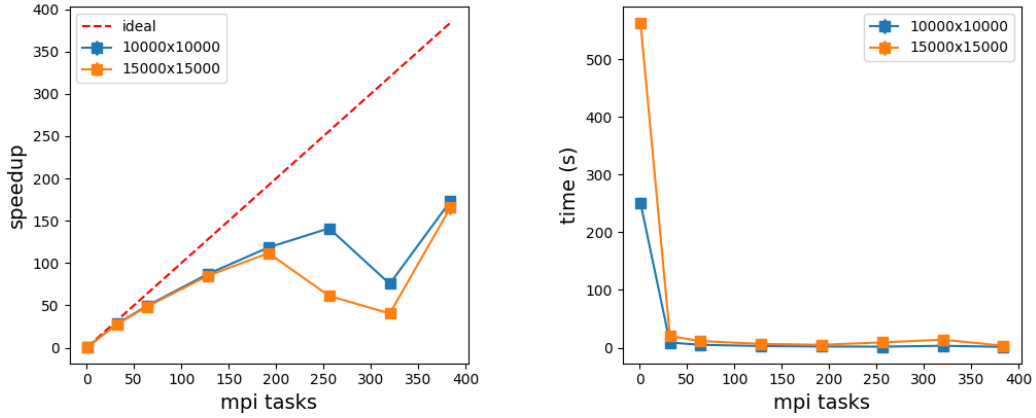
In figure 2, it is possible to observe the results obtained using `OMP_NUM_THREADS=64` and mpirun option `--map-by socket`. In this case, there is an evident difference between the tested sizes speed-up and run-times behaviors. This may be due to the message passing overhead that increases when increasing the length of the rows to be exchange.



(a) Speed-up behavior. **(b)** Run-time behavior.

Figure 2: Strong MPI scalability curves using 64 threads per MPI process with 1 system dump at the end.

Figure 3 shows the results for the runs with `OMP_NUM_THREADS=1` and `--map-by core`. It is possible to notice that, as the number of MPI processes grows, the scalability shows a pronounced sub-linear behavior. This behavior is probably caused by the message passing approach chosen for the exchange of the halo rows between processes: the `MPI_Sendrecv` is a blocking routine and therefore, it holds off the rest of the computation, becoming very expensive when a large number of MPI tasks is used.



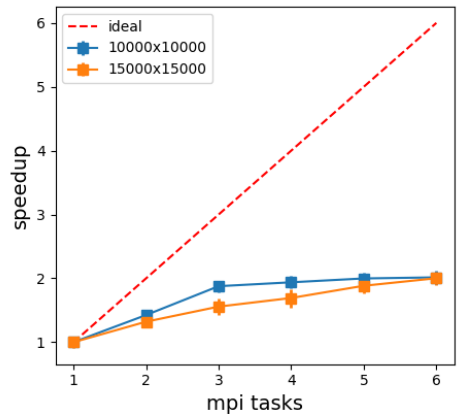
(a) Speed-up behavior. **(b)** Run-time behavior.

Figure 3: Strong MPI scalability curves using a single thread per MPI process with 1 system dump at the end.

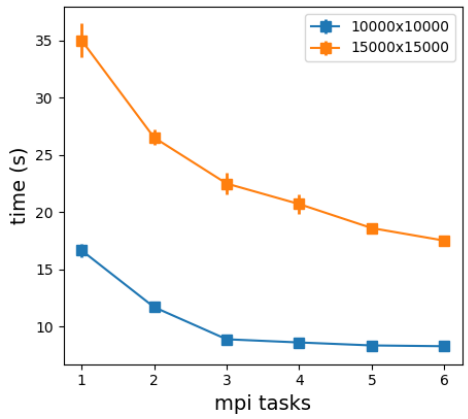
The results corresponding to the runs with 1 snapshot saving every 25 evolution steps are discussed below in figure 4.

It is possible to observe a significant drop in the performance when we increase the frequency at which the system is saved. This is one of the main limitation of this work, showing that gathering the

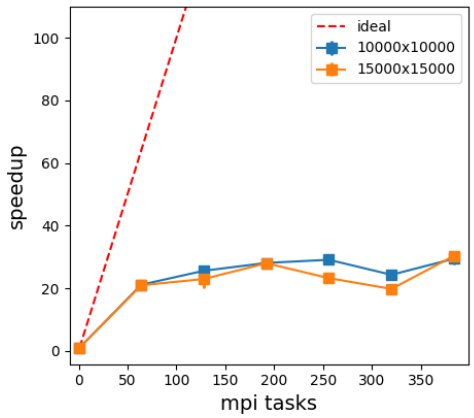
data from all the processes multiple times during a simulation is not be suitable for certain applications, e.g. when it is needed to observe the complete evolution of a system.



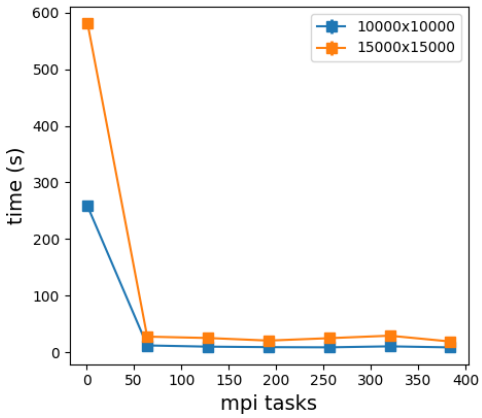
(a) Speed-up behavior for strong MPI scalability using 64 threads per MPI process and `--map-by socket`.



(b) Run-time behavior for strong MPI scalability using 64 threads per MPI process and `--map-by socket`.



(c) Speed-up behavior for strong MPI scalability using a single threads per MPI process and `--map-by core`.



(d) Run-time behavior for strong MPI scalability using a single threads per MPI process and `--map-by core`.

Figure 4: Strong MPI scalability curves with 1 system dump every 25 iterations.

1.4.3 Weak MPI scalability

In this kind of study, the ratio between the grid size and the number of MPI processes is kept approximately constant: starting from an initial grid size $k_1 = 12500$ for a single MPI process, it was further increased using the following rule:

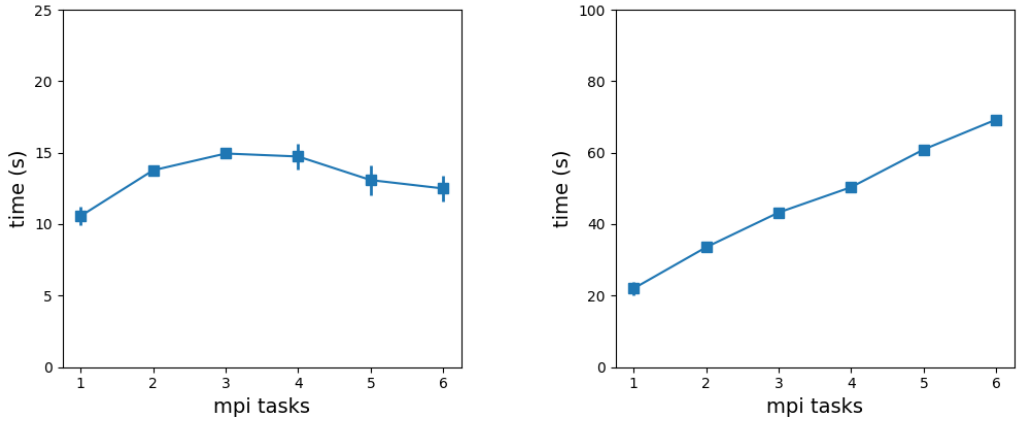
$$k_n = \lfloor \sqrt{n} k_1 \rfloor,$$

where n is the number of MPI tasks. The tested sizes are grouped in the table below. The runs were executed setting again `OMP_NUM_THREADS=64` and using the `--map-by socket` binding option.

n	1	2	3	4	5	6
k_n	12500	17677	21650	25000	27950	30618

Looking at the graph displayed in figure 5a, it is possibile to notice an unexpected behavior: ideally, the run-time should remain constant with the number of MPI processes. In this case, even if the difference in terms of absolute time measurements amounts to a few seconds, it is possible to observe a variation in the performance as the number of MPI tasks (and system size) increases.

When increasing the snapshot saving frequency (figure 5b), the run-time behavior worsen noticeably: the reason behind this behavior is the gathering of all the sub-grids into one single MPI process, which becomes more expensive as more processes are required to send the corresponding sub-grids.



(a) Run-time behavior for weak MPI scalability for 1 system dump at the end.

(b) Run-time behavior for weak MPI scalability for 1 system dump every 25 steps.

Figure 5: MPI scalability curves in the case of no snapshot writing.

1.5 Conclusions

The scalability studies shows that the chosen implementation allows to improve the run-time performance when using multiple OpenMP threads and MPI processes. Nonetheless, from the obtained results, it is possible to notice some problems which present opportunities for improvement.

Two main factors may play an important role:

- The message passing approach chosen to exchange the halo data between processes;
- The gathering of the local grids into one single process (when intermediate snapshot are taken).

Performing the halo exchange using the `MPI_Sendrecv` may be the cause of the poor scalability performance obtained. When using a high number of MPI tasks with a single thread per task, the local grids are much smaller and therefore much less computation on them can be done between two halo exchanges. A better suited approach could be to use strategically the non-blocking routines `MPI_Isend` and `MPI_Irecv`:

1. initiate the communication process before the double `for` loop on the rows and columns (for the computation of the rules of the game);
2. proceed with the computation of the Conway's game of life in a double `for` loop that excludes the first and last two rows (of the padded grids). The first and the last are the ones to be exchanged, while the second and the second to last cannot be updated if the exchange has not completed yet;
3. after the exchange has safely completed, compute the rules on the second and second to last rows.

In this way, it may be possible to perform the majority of the necessary computations on the grid cells without having to wait for the communication to happen.

The MPI gathering process exploited to collect the local grids in a single one play crucial role when intermediate system snapshots are taken. This assumption is supported by the results obtained using a higher sampling frequency for dumping the system. Both speed-up and run-time scale better when the gathering mechanism is not triggered, indicating that efforts should be focusing on improving the implementation of the snapshots' writing procedure. A possible solution could be to parallelize the writing procedure using MPI parallel I/O tools. This approach could improve both the performance related to the snapshot writing itself and the management of I/O processes especially for very large grid sizes. The workflow could be organized in the following way:

1. Declare a file using the file handle `MPI_File` and open it with the `MPI_File_open` routine;
2. After setting up the proper header for a PGM file (by the master process only), use the `MPI_File_write_at_all` routine to adjust each process' view of the file: `MPI_File_set_view` takes an offset parameter `disp` which allows each process to access the data in the file from a particular position: in this case the displacement should be defined based on the size of the local sub-grid and proportional to the process' rank.
3. Use the `MPI_File_write` routine to write the local grids into the file declared at step 1;

2 Exercise 2

In this section, the performances of MKL and OpenBLAS math libraries were compared. The focus is on the general matrix multiplication, implemented through the level 3 BLAS function *gemm*.

The analysis is divided into *size scaling* and *core scaling*, measuring floating point operations per second and speed-up, respectively. For each of the aforementioned studies, both single and double precision were considered. For this study, squared matrices only were considered.

The nodes used to run the jobs belong to the EPYC partition.

2.1 Implementation

The source code file was slightly modified in order to print on file only the necessary quantities: the number of processes, m, n, k (the linear dimensions of the matrices), the elapsed time and the floating point operations per second. This allowed an easier post-processing and display of the data. In addition to the saved measurements, the relative difference between MKL and OpenBLAS performance was computed in order to better understand the relationship between the two libraries' performances when scaling either the size of the matrices or the number of cores. The utilized formula for the relative difference is the following:

$$\frac{P_{\text{MKL}} - P_{\text{OpenBLAS}}}{P_{\text{OpenBLAS}}},$$

where P is the performance measured in terms of floating point operation per second.

The jobs were submitted using the `sbatch` command and two different bash scripts, one for the size scaling analysis and the other for the core scaling analysis.

The affinity study was limited to three places (sockets, cores and threads) and two binding policies (spread, close).

2.2 Theoretical Peak Performance

Given the clock rate value ν, the number of floating point operations per cycle F and the number of cores N, the theoretical peak performance TPP can be computed as:

$$TPP = \nu F N.$$

Regarding the EPYC nodes, the aforementioned quantities have the following values:

- $\nu_{\text{EPYC}} = 2.6\,\text{GHz}$ (from the official AMD website [3]);
- $F_{\text{EPYC}}^{\text{float}} = 32$, $F_{\text{EPYC}}^{\text{double}} = 16$ as stated at the linkedwebsite [4];
- $N = 64$;

thus,

$$TPP_{\text{EPYC}}^{\text{float}} \simeq 5324.8 \text{ GFLOPS} \text{ and } TPP_{\text{EPYC}}^{\text{double}} \simeq 2662.4 \text{ GFLOPS}.$$

It follows that, for a single core,

$$TPP_{\text{EPYC,core}}^{\text{float}} \simeq 83.2 \text{ GFLOPS} \text{ and } TPP_{\text{EPYC,core}}^{\text{double}} \simeq 41.6 \text{ GFLOPS}.$$

The value of $TPP_{\text{EPYC}}^{\text{float}}$ resembles what indicated in the tutorial `hpl-on-epyc.md` available in the github repository of the course.

2.3 Size Scaling on Epyc

This section presents the analysis of the affinity study conducted while varying the size of the matrices from 2000×2000 to 20000×20000 (with steps of 2000).

2.3.1 Single Precision

The performance of the number of floating point operations per second when using single precision can be observed from figures 6, 7.

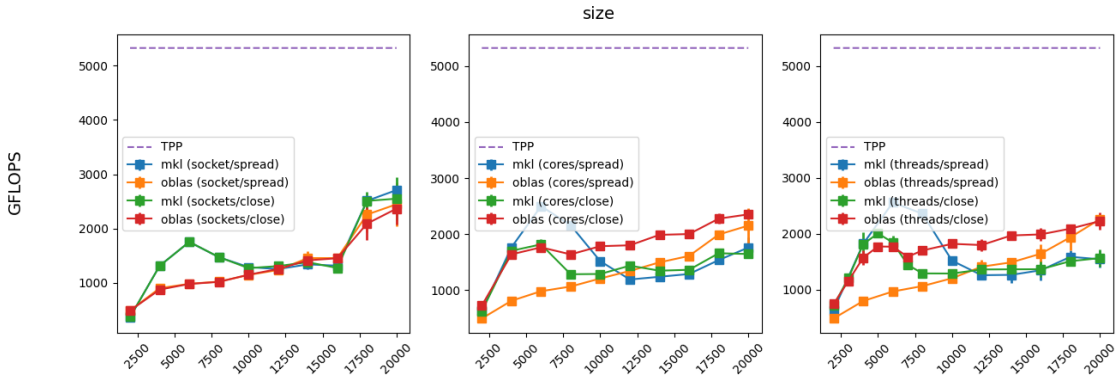


Figure 6: Single precision speed-up scaling w.r.t. the size of the matrices

In the case of `OMP_PLACES=sockets`, the obtained trends are similar for both MKL and OpenBLAS. It is noticeable that the difference between the corresponding bindings is negligible.

In the other cases, when using the Math Kernel Library, setting `OMP_PROC_BIND=spread` leads to better performance in the small-medium size range while, with the exception of the `OMP_PLACES=sockets` case both the bindings leads to poor performances for larger sizes.

As far as the OpenBLAS library is concerned, `OMP_PROC_BIND=close` leads to better results basically everywhere in the studied size range. However, the spread policy catches up for large matrices, indicating that it might be better when exceeding the upper bound of this range.

Overall, the performances remain below 50% of the theoretical peak performance.

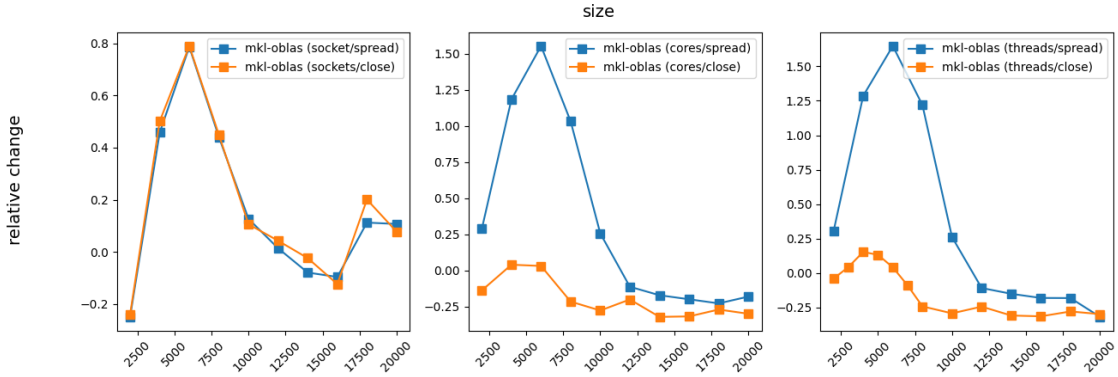


Figure 7: Relative change scaling w.r.t. the size of the matrices in single precision

The trend of the relative difference between MKL and OpenBLAS shows similar features across places and bindings. In all the cases, MKL performance surpasses that of OpenBLAS for small matrices, reaching a maximum performance percentage increase in the $[5000, 7500]$ size range, after which it rapidly decreases for medium-sized matrices. Apart from some irregularity in the trend for `OMP_PLACES=sockets`, OpenBLAS generally performs better when dealing with large matrices, showing a slow increasing performance relative difference with respect to MKL for growing sizes.

2.3.2 Double Precision

Looking at figures 8, 9, it is possible to observe the performance of the number of floating point operations per second when using double precision.

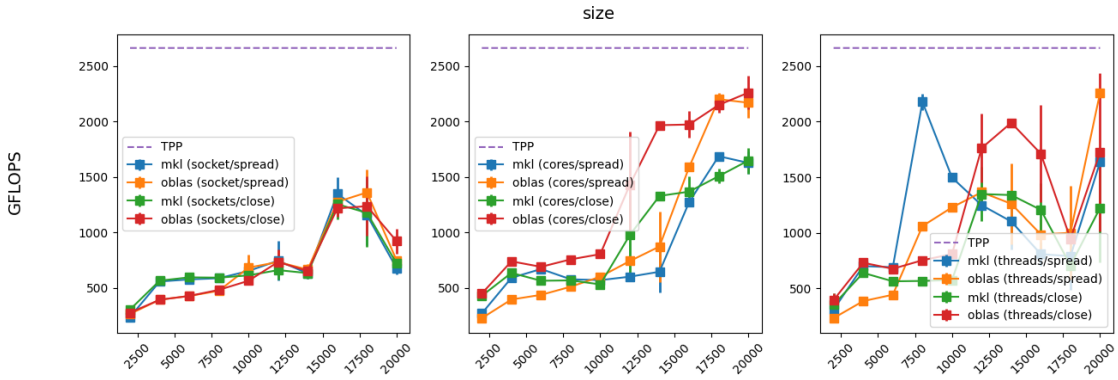


Figure 8: Double precision speed-up scaling w.r.t. the size of the matrices

In the case of double precision, the behavior for large sizes presents more irregularities than the previous case and, generally lower performance in terms of GFLOPS, as expected.

The above graphs show a behavior similar to the one obtained for the single precision when using `OMP_PLACES=sockets`. In this case, with `OMP_PLACES=sockets`, the choice of the binding makes little difference. When `OMP_PLACES=cores` or `OMP_PLACES=threads`, OpenBLAS outperforms MKL for large matrices.

In this case, it is possible to notice that the performance reach an high ration with respect to $TPP^{\text{double}}_{\text{EPYC}}$, especially for the case of OpenBLAS used with `OMP_PLACES=cores`.

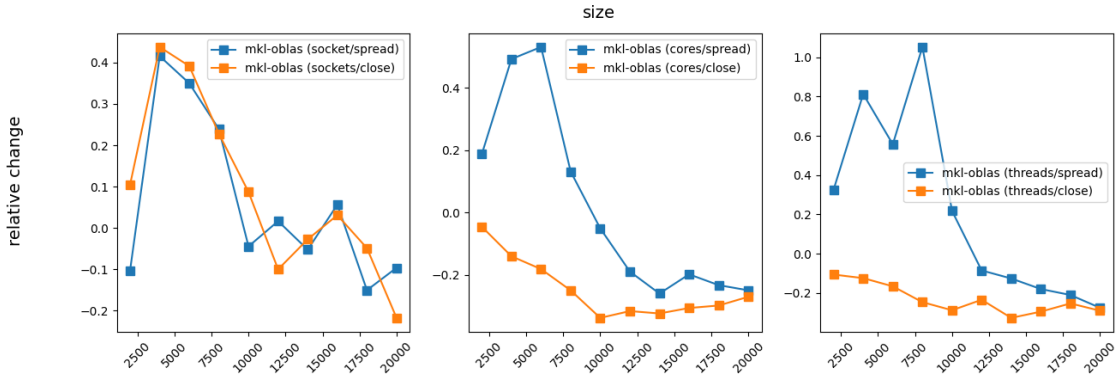


Figure 9: Relative change scaling w.r.t. the size of the matrices in double precision

As for the single precision case, the performance relative difference of MKL with respect to OpenBLAS reaches a maximum value for small matrices and successively decreases with growing matrix size. Again, OpenBLAS performance surpass that of MKL for large matrices.

2.4 Core Scaling

The analysis reported in this section was conducted keeping the matrix size fixed at 11000×11000, while gradually increasing the number of cores.

2.4.1 Single Precision

Looking at figure 10 it can be noticed that OpenBLAS generally achieve better performances than MKL for `OMP_PLACES=cores` and `OMP_PLACES=threads` (with the same binding policy) and that, in these cases, same bindings replicate similar behavior for different places. In almost all the cases, both libraries scale well up to 24 cores. For `OMP_PLACES=cores` and `OMP_PLACES=threads`, the close binding leads to better results for both the libraries, with the exception for `OMP_PLACES=sockets`. In this case, setting `OMP_PROC_BIND=spread` leads to better results for both OpenBLAS and MKL.

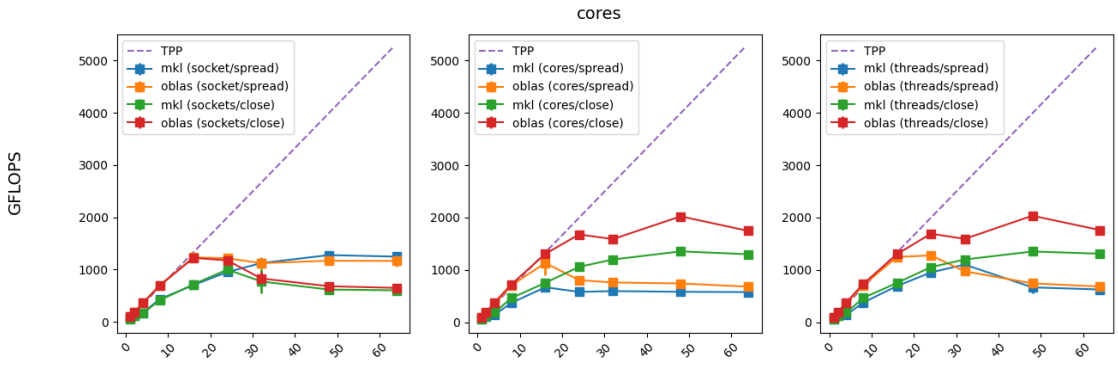


Figure 10: Single precision speed-up scaling w.r.t. the number of cores

From figure 11 it is possible to notice that OpenBLAS outperforms MKL in almost all the cases, except for `OMP_PLACES=sockets` with the spread binding policy.

It is possible to observe that the GFLOPS behavior scales well up to 20 cores, but then it either remains approximately constant or starts decreasing. With the exception of `OMP_PLACES=sockets`, the decreasing behavior is characteristic of the `OMP_PROC_BIND=spread` binding policy.

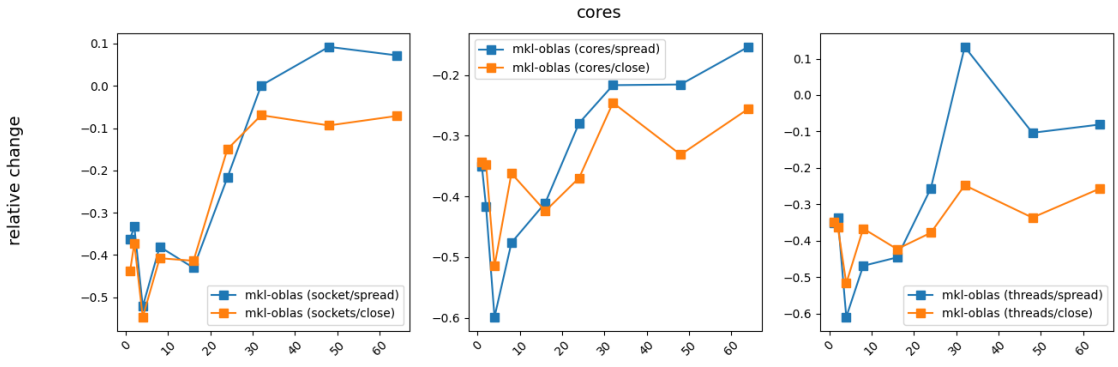


Figure 11: Relative change scaling w.r.t. the number of cores in single precision

2.4.2 Double Precision

Overall, OpenBLAS scales better when performing double-precision floating point operations. Only when using sockets as places it is possible to notice a performance increase for MKL for large number of cores. OpenBLAS scales well up to 24 cores while MKL up to 16 cores.

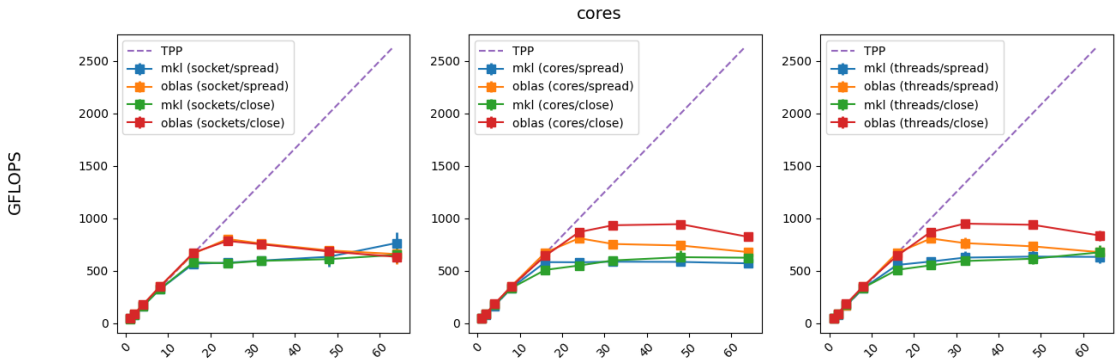


Figure 12: Double precision speed-up scaling w.r.t. the number of cores

The trend of the performance relative different is similar across all places and bindings and somehow resemble the one found for the single precision scaling: the minimum of the relative difference is shifted towards the medium range of number of cores, meaning that OpenBLAS scales better than MKL in a larger number of cores range. When further increasing this quantity MKL achieves better scaling (the slope of the curve is positive) but it is already out of the strong scaling region. Both in this case and in the single precision case, OpenBLAS's speed-up shows a decreasing trend for large number of cores, suggesting the overhead associated with this library has growing influence when increasing the number of processes.

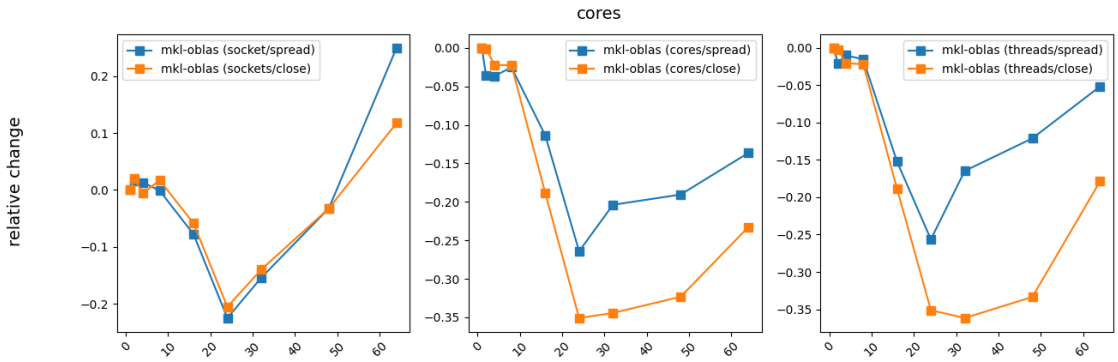


Figure 13: Relative change scaling w.r.t. the number of cores in double precision

3 Conclusions

As expected, the sustained performance for both single and double precision cases is lower than the theoretical one.

Moreover, form the results reported above it is possible to conclude that MKL library generally performs worse than OpenBLAS on an EPYC node. This is due to the fact that the EPYC nodes are characterized by non-Intel architecture. Therefore, using a THIN node might lead to much better results when using the Math Kernel Library.

References

[1] Conway's Game of Life. Wikipedia.
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[2] J. Pitt-Francis and J. Whiteley. *Guide to Scientific Computing in C++*, 2nd edition, chapter 11, section 4.2.2. Springer, 2017.

[3] Advanced Micro Devices. *AMD EPYC™ 7H12*. https://www.amd.com/en/product/9131.

[4] Microway. *Detailed Specifications of the AMD EPYC "Rome" CPUs*.
https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-amd-epyc-rome-cpus.

[5] William Gropp. *Lecture 32: Introduction to MPI I/O*.
https://wgropp.cs.illinois.edu/courses/cs598- s16/lectures/lecture32.pdf.