Foundations of High-Performance Computing
Final Project

Costanzo Gabriel, Imeneo Silvia

Course of AA 2022-2023 - Data Science and Scientific Computing

Contents

1 Exercise 1

1.1 Introduction

The exercise is focused on the implementation of a parallel version of Conway's "Game of Life". In this section, we are going to briefly describe the setup and the rationale of our implementation. For further information please check the reference material [1].
The Game of Life (GOL) is a game that requires no input from the player, but just an initial state from which the game will evolve, based on specific rules about the "alive" or "dead" status of the cells composing the playground.
There are several ways in which this evolution can take place and we consider two of them:

- The ordered evolution: the cells are updated one by one, in sequential order, from left to right and one row at a time.
- The static evolution: the system is frozen, the status of each cell is evaluated and then all the cells are updated.

The exercise requests an implementation of these two approaches of the game, using a hybrid MPI (distributed memory parallelism)+ OpenMP (shared memory parallelism) code, and three scalability studies:

- **OpenMP scalability**: we have a fixed number of MPI tasks (1 per socket) and an increasing number of threads per task (up to the number of cores present on the socket)
- **Strong MPI scalability**: we have a fixed size of the playground and an increasing number of MPI tasks (across which the playground is divided).
- **Weak MPI scalability**: we have an increasing size of the playground, a corresponding increasing number of sockets (saturated with OpenMP threads) and a fixed workload per MPI task.

1.2 Methodology

We start with the initialization of our playground: a random black-and-white image represented as a 2D array of unsigned characters.
The initiated playground is then divided into multiple blocks, which are individually assigned to different MPI tasks in order to have a distribution of the overall workload. This is where the MPI section of our hybrid approach takes part. This step is about domain (not functional) decomposition since we have to apply the same rules to all the cells in the grid.
The workload division that we opt for is by row, meaning that each MPI task works in a given order on an evenly distributed number of rows. Each task reads just a sub-portion of the playground and then, after having updated its cells, it

[1] Assignment_exercise1 on the GitHub of the course

writes the output of its own part of the grid.

When focusing on the cell updating step, so on the evolution of the playground, we have to consider two rules of the Game of Life (see Figure 1):

- the next state of each cell depends on its 8 neighboring cells,
- the borders of the playground are periodic.

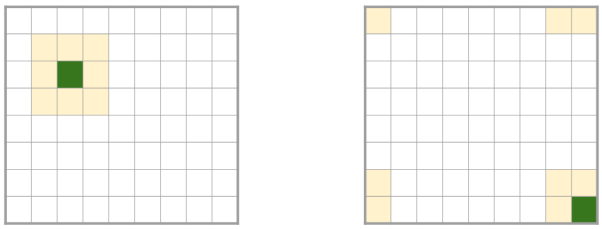

Figure 1: Two examples of a cell (green) and its 8 neighbors (yellow)

This means that, when the playground is divided into multiple blocks, in order to be able to update the first and the last row of each block, we need to know the last and the first row of the preceding and of the following blocks respectively(Figure 2).

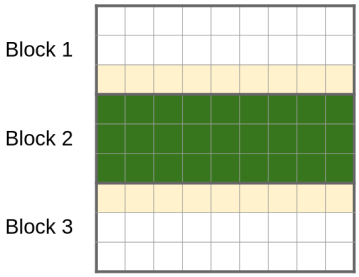

Figure 2: To update Block 2, we need to know the last row of Block 1 and the first row of Block 3

To comply with these rules, we use two auxiliary rows, called ghost rows: at each step of the evolution, each MPI task sends its first and last row to its neighboring tasks, which store this information in their own ghost rows. In this way, each block is always provided with the information needed to update its own part of the playground.

For both the ordered and the static approach, the "update cell" function updates the state of a cell based on the rules of Conway's Game of Life. It works

by counting the number of alive neighbors for a given cell (considering both the local playground and the top and bottom ghost rows) and then it updates the cell's state accordingly.

Let's now quickly see the peculiarities of the ordered and the static evolution approaches.

Ordered evolution

Due to the intrinsic serial structure of the ordered evolution, synchronization here is very important. This means that each block of the playground (hence each MPI task) needs to wait for its turn to update its cells. If the order is not respected, the results are compromised.
For this reason, after having read its portion of the local playground and having allocated the memory for the two ghost rows, each process checks if its turn has arrived. If it has not, the MPI task simply receives the updated MPI order and the top ghost row from the preceding block.
When its turn arrives, the MPI process performs the following operations n times:

- It divides the local region into threads using OpenMP parallelization and each thread, in order, updates its cells using the "update cell" function.
- If it's not the last process, it increases the MPI counter variable and communicates it to the next process. It also sends top and bottom ghost rows to its neighbors using MPI communication and barrier operations.
- Additionally, at regular intervals (each s steps), it writes a snapshot of the local region to an output file.
- Eventually, at the last step, it writes the final state of the GOL grid and frees the memory used for its two ghost rows.

Static evolution

As in the ordered evolution case, also in the static one each MPI task reads its portion of the local playground and allocates memory for the ghost rows. Also here synchronization is important and respected, but the MPI tasks are allowed for some degree of overlap between computation and communication. This means that, after initiating the communication with the other processes, a process can continue with its computation without waiting to complete the communication. However, as we will see in the next section, synchronization points are still present to ensure correctness.
As in the ordered evolution algorithm, also in the static one the OpenMP part is concentrated in a n iteration loop:

- The MPI process enters the loop and exchanges the ghost rows as already mentioned.
- A nested loop OpenMP region starts and the workload is distributed across multiple threads. Each one of them takes care of updating the status of its cells based on the "update cell" function.
- Once the threads are done with their work, the pointers to the playground are exchanged so that the currently updated playground can become the initial playground of the next iteration.
- If we are in a step multiple of s, a snapshot is taken.
- Eventually, the memory for the two ghost rows is released and a final snapshot is written.

1.3 Implementation

Command-line arguments

The exercise requested our code to be able to accept different command-line arguments, so we made sure to include them.
The Game starts when the `-i` command is received and a playground, with a given size `-k`, is initialized and written to a `pgm` file[2].

```
void * initialize_playground(int xsize, int ysize)
{
    srand(time(NULL));

    // Allocate memory for the image
    unsigned char *image = (unsigned char *)malloc(xsize * ysize *
    sizeof(unsigned char));

    // Generate random black-and-white pixel values
    for (int y = 0; y < ysize; y++)
    {
        for (int x = 0; x < xsize; x++)
        {
            image[y * xsize + x] = (rand() % 2) * MAXVAL;
        }
    }
    // Return pointer to the allocated memory
    return (void *)image;
}
```

The `initialize_playground` function uses `srand(time(NULL))` to ensure that, each time the program is run, the sequence of random numbers generated will be different. It then allocates dynamic memory to create an image of unsigned char. We use unsigned char since it's a common choice in image processing due to its memory efficiency: it uses only one byte of memory per pixel and that

[2]The code used is the one provided in the read_write_pgm_image.c file in the GitHub assignment directory of the FHPC course

helps when dealing with large images.
The function then generates random black-and-white pixel values and stores them in the allocated memory. The values are either 0 (black) or `MAXVAL` (white). Eventually, the function returns a void pointer pointing to the allocated memory containing the initialized image data.

When the command `-r` is called, the game starts running and the MPI part of the hybrid parallelism is used. The playground is divided into blocks, with a specific size each and a specific order in a rank. Each block is assigned to a different MPI process that will take care of performing the computations needed for its specific portion of the whole grid.
The work in each MPI process starts by allocating the memory for a portion of a `pgm` file and then reading that single portion using the `parallel_read_pgm_image` function.

```
unsigned char * playground_o = (unsigned char *)malloc(portion_size
    * sizeof(unsigned char));

parallel_read_pgm_image((void **)&playground_o, fname, my_offset,
   portion_size);
```

This function allows for multiple MPI tasks to collaboratively read different portions of the playground `pgm` file. Each process reads its designated portion and then, after all processes have completed reading, they synchronize using `MPI_Barrier` before the file is closed.

Once the file has been read, the computational part (the OpenMP section of our hybrid code) can take place. The development of the computation depends on the argument given to the `-e` command, which indicates the chosen evolution: ordered or static.

Let's now see more in detail these two approaches that we already introduced in the Methodology section.

Ordered evolution

The `ordered_evolution` function is structured as follows:

1. MPI Setup: MPI rank and size are retrieved to make sure that the function knows when each task's turn is. The `mpi_order` variable is initialized and it will be used as a counter variable so that each MPI process can know when its turn has arrived.

```
void ordered_evolution(unsigned char *local_playground, int
    xsize, int my_chunk, int offset,  int n, int s) {
int rank, size, nthreads;
MPI_Comm_rank(MPI_COMM_WORLD, &rank); //rank of the current
    process
MPI_Comm_size(MPI_COMM_WORLD, &size); //total number of
    processes
```

```
5
6 int mpi_order = 0; // counter variable
7
```

2. Ghost Rows Setup and Exchange: the memory for the two ghost rows is allocated and the neighboring processes are identified. After that, blocking MPI send and receive operations are used: (i)to send the top row of the local grid to the process above and (ii) to receive the bottom ghost row from the process below. The use of blocking operations ensures correct synchronization and ordered processing.

```
1 // MPI send and receive operations for initial exchange of
      ghost rows
2 MPI_Send(&local_playground[0], xsize, MPI_UNSIGNED_CHAR,
      top_neighbor, 1, MPI_COMM_WORLD);
3
4 MPI_Recv(bottom_ghost_row, xsize, MPI_UNSIGNED_CHAR,
      bottom_neighbor, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
5
6 MPI_Send(&local_playground[(my_chunk - 1) * xsize], xsize,
      MPI_UNSIGNED_CHAR, bottom_neighbor, 0, MPI_COMM_WORLD);
7
8 MPI_Recv(top_ghost_row, xsize, MPI_UNSIGNED_CHAR, top_neighbor
      , 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
9
```

3. n loop: inside the loop, the processes wait for their turn by repeatedly receiving the `mpi_order` value from their top neighbor. Once a process's turn arrives (so when `mpi_order` matches the process's rank), it proceeds to perform computation.

    - At this stage, OpenMP parallelism is used to distribute the computation among multiple threads within the same process. Each thread is responsible for updating a portion of the rows in the local grid. The threads are synchronized using a critical section that contains an `order` variable. A critical section is a specific section of a program's code that is designed to be executed by only one thread at a time, and the `order` variable is here used to keep track of which thread should perform the computation. After a thread has completed its work and incremented the `order` variable it exits the critical section.

```
1  int nthreads;
2  int th_chunk;
3  int th_mod;
4  int order;
5
6  #pragma omp parallel
7  {
8      #pragma omp single
9      {
10         nthreads = omp_get_num_threads();
11         th_chunk = my_chunk / nthreads;
12         th_mod = my_chunk % nthreads;
```

```
    }
}

order = 0;
#pragma omp parallel
{
    int me = omp_get_thread_num();
    int done = 0;
    int th_my_first = th_chunk * me + ((me < th_mod) ? me
    : th_mod);
    int th_my_chunk = th_chunk + (th_mod > 0) * (me <
    th_mod);

    while (!done) {
        #pragma omp critical
        {
            if (order == me) {
                for (int y = th_my_first; y < th_my_first
    + th_my_chunk; y++) {
                    for (int x = 0; x < xsize; x++) {
                        update_cell_ordered(top_ghost_row,
     bottom_ghost_row, local_playground, xsize, my_chunk,
    x, y);
                                    }
                                }
                order++;
                done = 1;
                        }
                    }
                }
            }
        }

```

- If the process is not the last one, it updates the `mpi_order` and sends it to the process below together with its bottom row, which will be used as the top ghost row in the next step. If the process is either the first one or not the last one, it also sends its top row as the bottom ghost row to the process above.
- After each iteration, all processes synchronize using `MPI_Barrier` to ensure that they have all completed their current step before proceeding to the next one.
- If the current step number is a multiple of s (the command-line argument which indicates every how many steps a dump of the system is saved on a file), a snapshot of the grid is written using the `write_snapshot` function.

4. Final Snapshot: after completing all (n) steps, an additional snapshot is written.

5. Memory Cleanup: dynamically allocated memory for the ghost rows is freed before the function exits.

```
    free(top_ghost_row);
    free(bottom_ghost_row);
}
```

Static evolution

The `static_evolution` function is structured as follows:

1. Initialization: as the `ordered_evolution` function does, the static one starts by retrieving the MPI rank, the total number of MPI processes, and by calculating the size of the portion of the grid that each process is responsible of.

2. Ghost Rows: memory is allocated to store the top and bottom rows of the neighboring processes, which are also identified in this step.

3. n loop:

    - MPI Communication: each MPI process sends its top and bottom rows to its neighbors and receives the bottom and the top rows of its preceding and following processes. Differently than in the ordered evolution, here the MPI communication is a mix of non-blocking send and blocking receive operations. The non-blocking operations (which indicate that a process can continue its execution without waiting for the communication to be completed) are used to improve parallelism and reduce wait time. The blocking operations instead are used to ensure correctness and synchronization, to make sure that each process doesn't start the computation until it has the required data.
    - OpenMP: a nested loop OpenMP parallel region is started. The parallel region divides the work among multiple threads, and the two nested loops are collapsed into a single parallel loop. By collapsing these two loops, the OpenMP `parallel for` loop will distribute the combined iterations of both loops among the available threads in a way that optimizes parallelism.
      The outer loop iterates over the rows of the local portion of the grid, while the inner loop iterates over the columns, and, within the loop, the `update_cell_static` function is called: each thread computes the new state of its assigned cell while taking into account the state of neighboring cells and the boundary conditions represented by the ghost rows.

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < my_chunk; y++)
    {
        for (int x = 0; x < xsize; x++)
        {
```

```
6             update_cell_static((y == 0 ? top_ghost_row : &
    local_playground[(y - 1) * xsize]),
7                 (y == my_chunk - 1 ? bottom_ghost_row : &
    local_playground[(y + 1) * xsize]),
8                 local_playground, updated_playground,
    xsize, my_chunk, x, y);
9         }
10    }
11
```

- Grid Swap: once each thread has done its job and updated its cells, pointers between `local_playground` and `updated_playground` are swapped. This updates the local grid state for the next iteration.
- Snapshot Writing: as for the `ordered_evolution` function, if the current step number is a multiple of s, a snapshot of the grid is written using the `write_snapshot` function.
- Memory deallocation: the memory allocated for the two ghost rows is freed to release memory resources and ensure that memory is efficiently managed throughout the simulation.

4. Final Snapshot: after completing all the n steps, an additional snapshot is written.

Measuring run-time and performance

The exercise requests to report the change that we see in the run-time behaviour when varying the different settings in the scalability studies. To measure this run-time, we use the `gettimeofday` function, which computes the wall-clock time. We decided to use this function because we wanted to measure the time passed between the beginning and the end of the cell evolution function run for all the MPI processes. For this reason, we call `gettimeofday` before the ordered/static evolution function and after the `MPI_Finalize` (to make sure that all the processes have finished their work). The difference between the obtained start and end values indicates the total elapsed time needed to perform the cell evolution n times. This elapsed time is then divided by the number of iterations and the mean value is printed to a csv file.

For some scenarios, we also evaluated the performance using the speedup, which measures how faster a parallelized application runs compared to the same application running in a non-parallelized setting (so, for example, with a single thread or a single core, depending on which scaling we are considering). Ideally, the speedup has a linear shape, where doubling the number of processing elements leads to doubling the performance and, in our case, halving the run time. In practice, achieving perfect linear speedup is often not possible due to overhead and communication costs in parallel computing.

The Speedup is calculated using the formula:

$$Speedup_n = \frac{T_{serial}}{T_{par\_n}}$$

Where T_{serial} is the time needed when the computation happens using 1 MPI process or 1 OpenMP thread, and T_{par_n} is the time needed when using n processes or n threads running in parallel.

Job files

We prepared different batch script files to automate the repetitive commands and to indicate the specific arguments for each different scalability study that we performed.
The files have some similarities, like:

- SLURM commands (SLURM is ORFEO's resource manager software):

```
#!/bin/bash
#SBATCH --no-requeue
#SBATCH --job-name="static_strong_MPI"
#SBATCH --partition=THIN
#SBATCH -N 2
#SBATCH -n 48
#SBATCH --exclusive
#SBATCH --time=02:00:00

```

  Where: `#!/bin/bash` tells the system to use bash shell to execute the script; `#SBATCH --no-requeue` avoids the job from being re-queued automatically; `#SBATCH --partition` indicates the architecture that we want to run on; `-N` and `-n` indicate the number of requested nodes and cores respectively; `#SBATCH --exclusive` requests exclusive access to the allocated node; and `#SBATCH --time` indicates the maximum run-time allowed for the job.

- Module loaded and threads affinity policy:

```
module load openMPI/4.1.5/gnu/12.2.1
policy=close
export OMP_PLACES=cores
export OMP_PROC_BIND=$policy

```

  After loading the module for OpenMPI with the GNU compiler version, we use `OMP_PLACES=cores` to specify that OpenMP threads should run on CPU cores, and use `OMP_PROC_BIND=$policy` to indicate that the OpenMP threads should be bound to specific CPU cores in a close manner, as indicated by `policy=close`.

- Compilation of the executables with the use of the Makefile:

```
loc=$(pwd)
cd ../../..
make all location=$loc
cd $loc

```

- Final cleaning step to remove generated files, object files, executables, and all currently loaded modules in the current environment.

```
cd ../../..
make clean
module purge
cd $loc

```

The commands provided before are equal for all shell scripts. Following we describe the parts of the scripts for the different scalability implementations. Let's see one example for each scalability study:

- OpenMP scalability:

```
(...)
#SBATCH --partition=THIN
#SBATCH -N 1
#SBATCH -n 24
(...)

tasks=2
size=25000

data=$loc/time.csv
#echo "threads_per_socket, ordered_mean, static_mean" > $data

mpirun main.x -i -k $size -f "playground.pgm"

for thread in $(seq 1 1 12)
do
  export OMP_NUM_THREADS=$thread
  echo -n "${thread}," >> $data
  mpirun -np $tasks --map-by socket main.x -r -f "playground.
    pgm" -e 0 -n 5 -s 0 -k $size
  mpirun -np $tasks --map-by socket main.x -r -f "playground.
    pgm" -e 1 -n 50 -s 0 -k $size
done

```

We use 1 node (`-N 1`) and set the number of MPI tasks equal to 2 since we have 2 sockets in a node and we want to have one MPI task per socket. This distribution is ensured by the use of `--map-by socket`, which tells `mpirun` that the 2 MPI tasks should be distributed such that each one runs on a different CPU socket.
We fix the size of the playground to 25,000 and then, with the for loop, we generate data for the increasing number of threads, from 1 up to 12 (the number of cores present in one socket of a THIN node). On the CSV

file, we print the mean value of 5 iterations for the ordered evolution and 50 iterations for the static one.

- Strong MPI scalability:

```
(...)
#SBATCH --partition=THIN
#SBATCH -N 2
#SBATCH -n 48
(...)

data=$loc/time.csv
echo "size, processes, ordered_mean, static_mean" > $data

export OMP_NUM_THREADS=1
size=20000

(...)
for processes in $(seq 1 1 48)
do
      echo -n "${size},"  >> $data
      echo -n "${procs},">> $data
      mpirun -np $processes -N 2 --map-by core main.x -r -f "
    playground_${size}.pgm" -e 0 -n 50 -s 0 -k $size
      mpirun -np $processes -N 2 --map-by core main.x -r -f "
    playground_${size}.pgm" -e 1 -n 100 -s 50 -k $size
done
```

We fix the size of the playground to 20,000 and assign 1 thread per MPI task. The use of the command `--map-by core` places each MPI task on a different core. The number of MPI tasks is what changes, going from 1 up the the maximum number of cores that we have in the selected nodes, so 48 for two THIN nodes. The exercise indicates to use as many nodes as possible, but we limited it to 2 nodes due to the high request for resources on ORFEO and the long waiting time. The CSV file collects an average of 50 iterations for the ordered evolution and 100 for the static one.

- Weak MPI scalability:

```
(...)
#SBATCH --partition=THIN
#SBATCH -N 3
#SBATCH -n 72
(...)


data=$loc/time.csv
echo "size, procs, ordered_mean, static_mean" > $data

export OMP_NUM_THREADS=12
for size in 10000 14143 17321 20000 22361 24495
do
    mpirun -np 1 -N 1 --map-by socket main.x -i -f "
    playground_${size}.pgm" -k $size
done

```

```
17
18 echo "10000, 1," >> $data
19 mpirun -np 1 -N 3 --map-by socket main.x -r -n 5 -e 0 -s 0 -f
       "playground_10000.pgm" -k 10000
20 mpirun -np 1 -N 3 --map-by socket main.x -r -n 50 -e 1 -s 0 -f
       "playground_10000.pgm" -k 10000
21
22 echo "14143, 2," >> $data
23 mpirun -np 2 -N 3 --map-by socket main.x -r -n 5 -e 0 -s 0 -f
       "playground_14143.pgm" -k 14143
24 mpirun -np 2 -N 3 --map-by socket main.x -r -n 50 -e 1 -s 0 -f
       "playground_14143.pgm" -k 14143
25
26 #(continues with sizes 17321, 20000, 22361 and 24495 and an
       additional MPI task each time)
```

We use 6 sockets (by reserving 3 nodes), so our script computes the data for 6 different sizes of the playground and the corresponding number of MPI tasks.
By setting `OMP_NUM_THREADS=12` (for THIN nodes, 64 in the case of EPYC) we are essentially saturating each socket with OpenMP threads. This means that we are utilizing all available physical cores within each socket. We use `--map-by socket` to ensures that the MPI tasks are distributed evenly across the available sockets, avoiding over-subscription of CPU cores within a single socket.

Calculation of the playground size for weak MPI scalability

For Weak MPI scalability, to calculate the size of the problem (i.e., the dimensions of the playground) needed in order to maintain the workload per MPI process constant as we increase the number of MPI processes, we can use the following approach:

1. **Initial Size**: Start with an initial size of $10,000 \times 10,000$ for one MPI process. This means that the first MPI process will handle a matrix of size $10,000 \times 10,000$, which amounts to 10^8 cells.
2. **Scaling Size with MPI Processes**: As we scale up the number of MPI processes, we want to adjust the size of the matrix (k) so that the workload per MPI process remains constant. To do this, we use the formula:

$$\text{size} = \sqrt{n \times 10^8}$$

   - n represents the number of MPI processes we plan to use.
   - *size* is one side of the matrix for each MPI process.

Example Calculation: Let's say we want to test with 4 MPI processes:

$$\text{size} = \sqrt{4 \times 10^8} = \sqrt{4 \times 100,000,000} = \sqrt{400,000,000}$$

By calculating the square root, we find that for 4 MPI processes, each process will work with a matrix of size approximately $20,000 \times 20,000$.

In this way, applying the formula to the processes from 1 to 6 we get the following playground sizes:

MPI tasks	Matrix Size (k)
1	10,000 x 10,000
2	14,142 x 14,142
3	17,321 x 17,321
4	20,000 x 20,000
5	22,361 x 22,361
6	24,495 x 24,495

1.4 Results & Discussion

What could we expect in our results?

- Ordered evolution: as described before, ordered evolution is inherently serial due to the dependencies between cells. This means that there is limited scope for parallelism in the computation of cell status updates and, as a result, we may not expect a rising speedup in this case.
- Static evolution: here the limitations of the serial parts of the code are smaller, so in this case we can expect to see the real benefits of parallelization.

Let's now see the details for each scalability study.

1.4.1 OpenMP scalability

For the OpenMP study, we fix the number of MPI tasks to 1 per socket and check the behaviour of the code when the number of threads per task is raised from 1 up to the number of cores present on the socket. The purpose is to evaluate how well a parallel program utilizing OpenMP scales when increasing the number of threads per task on a multicore processor.

Ordered evolution

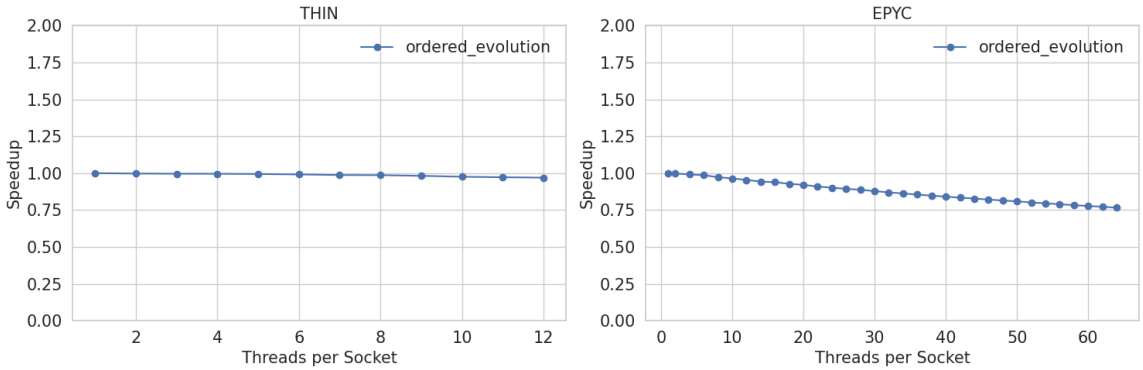

Figure 3: OpenMP scalability, ordered evolution

As we expected, Figure 3 shows no speedup increase for both EPYC and THIN nodes. The performance until 12 cores is similar for both types of nodes. However, for EPYC nodes, the effect of increasing the thread count clearly leads to performance degradation, which could be associated with synchronization.

Static evolution

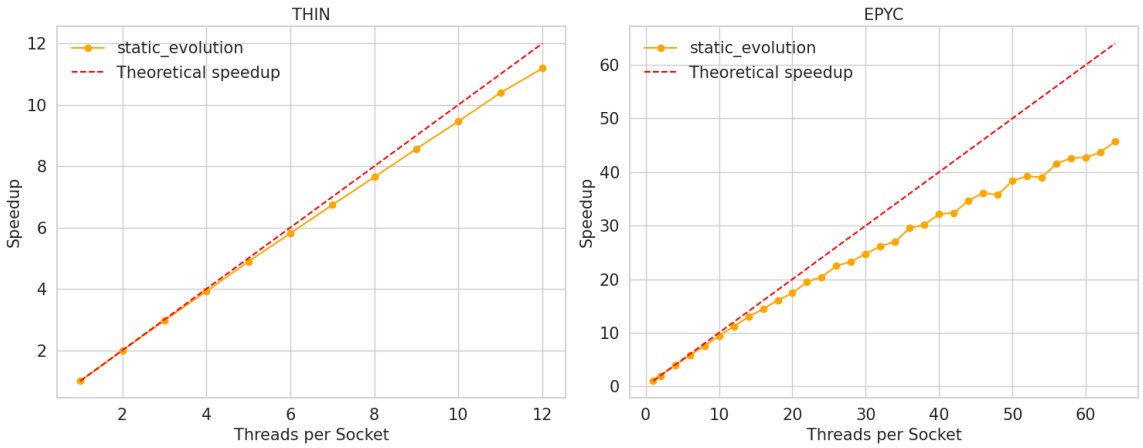

Figure 4: OpenMP scalability, static evolution

Figure 4 shows a rising speedup when we increase the number of threads for both EPYC and THIN nodes. However, we see that, after 12 threads

per socket, the speedup distances itself from the Theoretical speedup. This could be associated with contention, thread management and synchronization.

1.4.2 Strong MPI scalability

For the Strong MPI scalability study, we fix the size of the playground and the number of OpenMP threads per each MPI task, and we want to see how the run-time changes when we increase the number of tasks. In this scenario, the OpenMP part is not considered (it is actually absent since we have just 1 thread per process), and we investigate how the run-time changes when we divide the playground into smaller and smaller blocks, each one assigned to an MPI task. We repeated the test using two different fixed sizes (`k`) of the playground: 10000x10000 and 20000x20000.

Ordered evolution

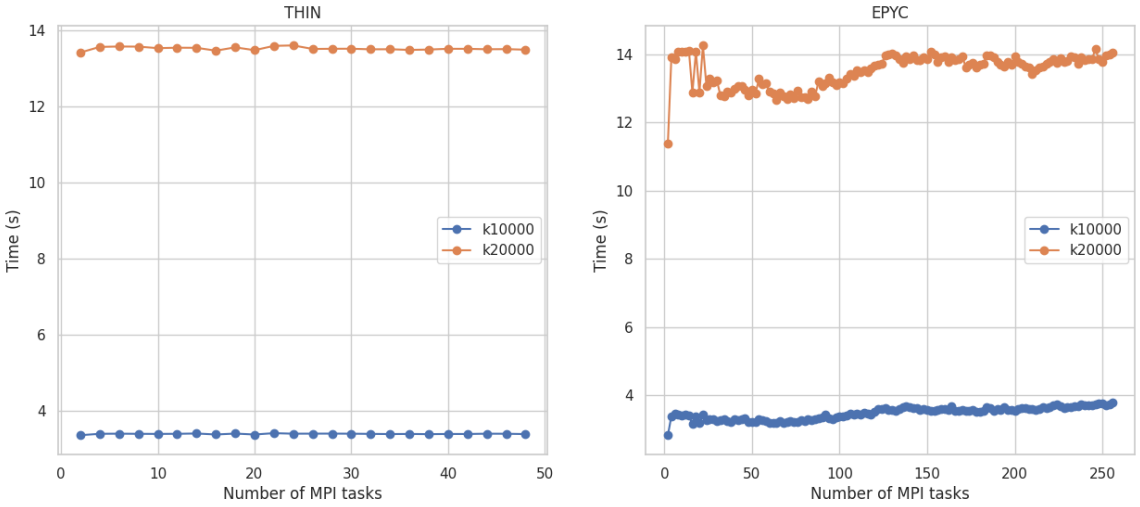

Figure 5: MPI strong scalability, ordered evolution

As we expected, Figure 5 shows a constant speedup for both the 10000x10000 and the 20000x20000 playgrounds and for both, EPYC (256 cores) and THIN (48 cores) nodes. However, for EPYC nodes, we observe a more unstable behaviour.

Static evolution

When analyzing the scalability for the static evolution, we considered two different sizes of the playground (10000x10000 and 20000x20000) and two different numbers of iterations: 10 and 100.

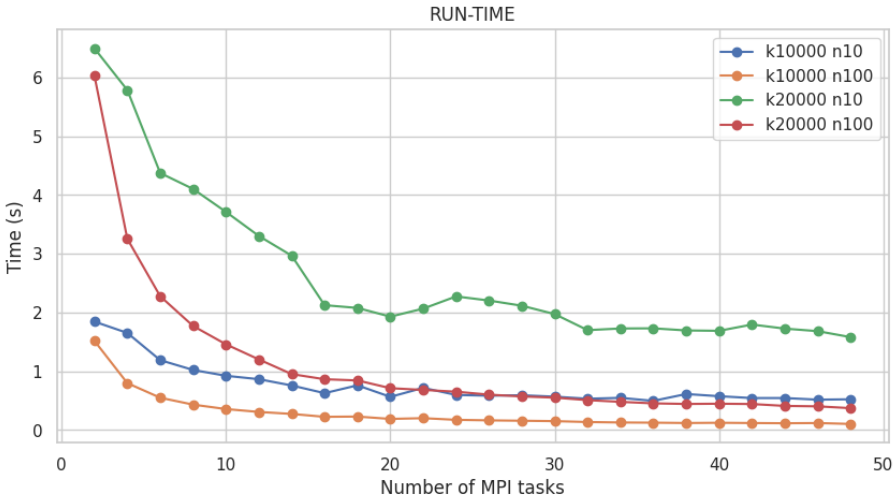

Figure 6: MPI strong scalability, THIN nodes, static evolution

By observing Figure 6, we can see that for any size and number of iterations, the run-time significantly decreases when we compute the operations using multiple MPI tasks. Interestingly, the best performance is obtained with a size of the playground 20000 and 100 iterations. As you can see in the plot for this setting, the run time decreases in an exponential decay way when the number of processors is increased.

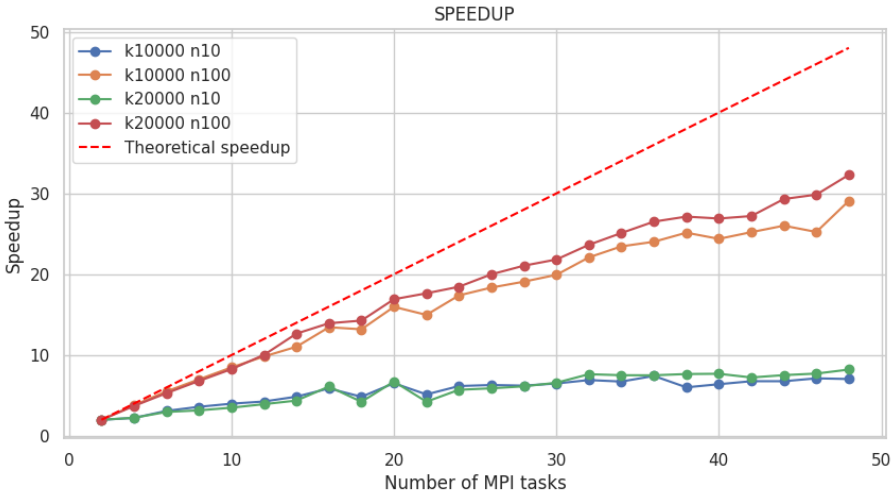

Figure 7: MPI strong scalability, THIN nodes, static evolution

When considering the Speedup (Fig. 7), we notice that the benefit of the parallelization is significant when we pass from 10 to 100 iterations and almost not noticeable when we double the size of the playground but have the same number of iterations. This might be due to the fact that, when we increase the number of MPI tasks, we add a communication overhead. When we perform fewer iterations per MPI task, the communication overhead becomes relatively more significant because the ratio of communication to computation is higher. With 100 iterations per MPI task, the communication overhead is amortized over a more substantial amount of computation, making it less significant in comparison.

1.4.3 Weak MPI scalability

The purpose of the Weak MPI Scalability is to evaluate the scalability of a parallel program that utilizes both MPI for distributed memory parallelism and OpenMP for shared memory parallelism. In this case, we start with an initial problem size (playground) and gradually increase the number of MPI processes while keeping the workload per MPI task fixed. The term "weak scalability" implies that we are increasing the computational resources (in this case, the number of sockets) in proportion to the problem size, aiming to maintain a constant workload per MPI task. We always have 1 MPI task per socket and, in this setting, we are considering 6 sockets (3 nodes).

Static evolution

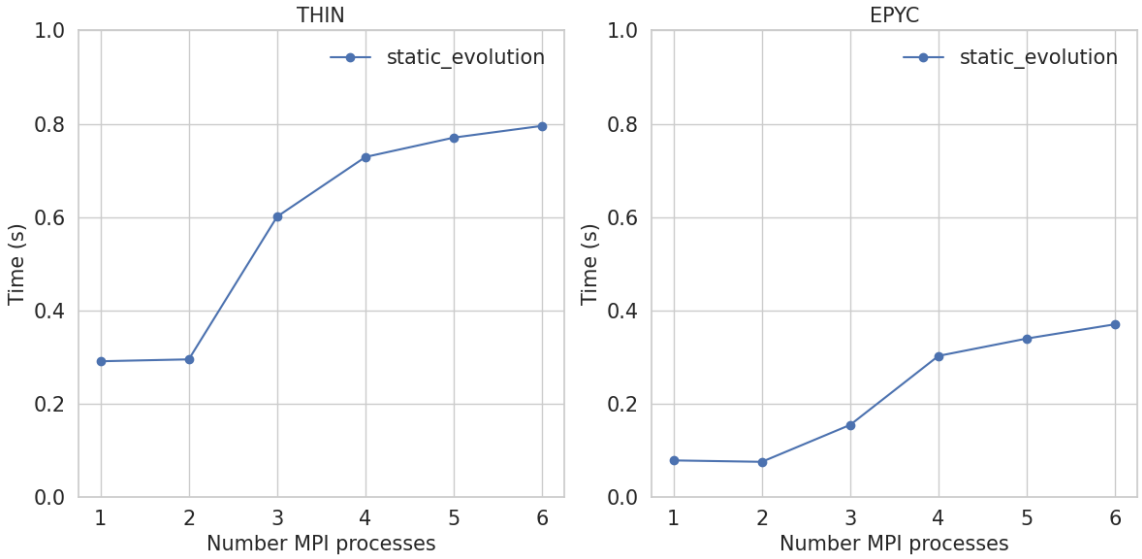

Figure 8: Weak MPI scalability, static evolution

Even though we set the workload to be constant for each MPI task, as we can see from Figure 8, the run-time increases when adding more tasks. We see that the run-time is constant for the first two tasks (first node) and then increases when moving to the second node (from process 2 to 3), probably due to communication overhead. We notice a further explainable increase when adding a third node (from process 4 to 5), but we haven't clearly understood why the increase in the run-time is so steep inside the second node, for both EPYC and THIN nodes.

1.5 Conclusions

1.5.1 OMP scalability

In an ideal scenario, we would expect to see close to linear speedup as we increase the number of threads. This indicates that our code efficiently utilizes the available CPU cores within each socket. However, as we see in Figure 4, after certain number of threads the increase in the speedup diminishes and, in Figure 3, the performance actually gets even worse. This behaviour could be due to increased synchronization overhead or contention for shared resources.

We identify the point of diminishing returns as the optimal ratio of number of threads over employed resources. Based on this, in the static scenario, we would say that: for a problem size of 25000x25000, using one MPI tasks per socket, the resources are optimized when we have 6 threads per THIN nodes and 16 threads per EPYC nodes.

1.5.2 Strong MPI scalability

An ideal scenario is characterized by linear speedup as we add more MPI tasks. This suggests that the code efficiently distributes the workload across multiple nodes without significant communication overhead. However, as we see in Figure 7, after a certain number of MPI tasks the speedup distances itself from the Theoretical line. This behaviour could be due to communication overhead from data exchanges or synchronization points that become bottlenecks.
Nevertheless, despite the burden of communication, since we noticed that increasing the number of iterations makes parallelism more valuable, we could investigate whether a further increase in n would make this gain even more substantial or if communication overhead would still remain a significant limit.

From Figure 6, we can also see that a local minimum in the run-time is reached when using 20 MPI tasks. Although it is true that the run-time continues decreasing until the 48 MPI tasks, choosing 20 cores ensures a maximal decay in the run-time. Based on this last observation, about our results on the static evolution we would say that the setting that benefits the most from parallelism is the one with two THIN nodes, a problem size of 20000x20000, 100 iterations, and 20 cores.

1.5.3 Weak MPI scalability

In an ideal scenario, we would observe that the execution time remains constant as we scale up. This indicates that our code can efficiently utilize additional computational resources.
As we see in Figure 8, the workload remains constant in the sockets of the same node but, when the MPI processes change the node, the run-time increases, showing an inter-node communication delay.

In terms of performance, the EPYC nodes perform better than the THIN nodes. The EPYC nodes take about half of the time to complete the MPI processes compared with the THIN nodes. This could be due to either the higher number of OMP threads by socket or/and a smaller communication overhead.

1.5.4 Ordered and Static Evolution

What we expected when starting this study is quite confirmed by the results that we just presented.
As Amdahl's Law states, the speedup of a parallel program is limited by the fraction of the code that cannot be parallelized, and this is evident when observing the ordered evolution case. We clearly see that our hybrid MPI+OpenMP code has strong limitations in these settings and there's not much that we can do to improve the performance. One thing that could be interesting to try is to compare the hybrid code with a serial one, to see how the absence of communication and synchronization overhead impacts the run-time of the code.

For the static evolution instead, we see that increasing the number of threads per MPI task leads to good results (OpenMP scalability). In the same way, increasing the number of MPI tasks seems to help, especially when dealing with a high number of iterations (Strong MPI scalability). What we haven't done in this study, but we could try to do, is to increase both the number of threads per task and the overall number of MPI tasks. While this might not be beneficial for small playgrounds or in the case of few iterations (due to the communication overhead), it could be worth trying it when working on very large grids or with many iterations.

2 Exercise 2

2.1 Introduction

The goal of the second exercise was to evaluate the performance of the math libraries MKL (Intel Math Kernel Library), OpenBLAS (an open-source BLAS library) and BLIS (BLAS-like Library Instantiation Software Framework) when performing matrix-matrix multiplication, and to compare it with the Theoretical Peak Performance.
The matrix multiplication was carried on with the use of the *gemm* function - a level 3 BLAS function - which is available both for single and double precision.

BLAS (Basic Linear Algebra Subprograms) provides a set of routines for common linear algebra operations [3]

Level 3 includes a "general matrix multiplication", where A, B and C are allocated matrices. Once A and B are filled, the *gemm* function calculates the matrix product $C = A * B$

The exercise requested us to test the scalability in two macro scenarios:

- **When increasing the size of the matrices** while keeping the number of cores fixed (12 for THIN and 64 for EPYC).
- **When increasing the number of cores** while keeping the size of the matrices fixed (we considered a size of 10000x10000).

For each scenario, we considered 8 different settings, based on different combinations of the 3 following options:

- **Node partition**: EPYC or THIN.
  EPYC (AMD EPYC 7H12) have 128 cores, while THIN (Intel Xeon Gold 6126) have 24. An EPYC node generally supports more memory channels and capacity, which can be advantageous for memory-intensive workloads.
- **Threads allocation policy**: close or spread.
  In the close allocation policy, SLURM aims to assign tasks to cores that are physically close to each other in the same socket. This minimizes communication overhead and improves cache utilization. In the spread allocation policy, SLURM aims to distribute the allocated cores as evenly as possible across the available sockets. This helps balancing the load across the CPU sockets, achieving better resource utilization.[4]

[3] Wikipedia: Basic Linear Algebra Subprograms
[4] SLURM documentation: salloc

– **Precision of the *gemm* function**: double (*dgemm*) and single (*sgemm*)

The double datatype used in *dgemm* has a total size of 64 bits, so a higher number of decimal digits is reserved for precision compared to those of *sgemm*, whose datatype has a total size of 32 bits.

The multiple options that we considered led to a total of 16 scenarios (see table below). In each of them, we compared the performance of the three math libraries and the theoretical peak performance.

Scalability over cores (fixed matrix size)	EPYC nodes	Threads allocation policy: close	precision: dgemm
			precision: sgemm
		Threads allocation policy: spread	precision: dgemm
			precision: sgemm
	THIN nodes	Threads allocation policy: close	precision: dgemm
			precision: sgemm
		Threads allocation policy: spread	precision: dgemm
			precision: sgemm
Scalability over size of matrix (fixed number of cores)	EPYC nodes	Threads allocation policy: close	precision: dgemm
			precision: sgemm
		Threads allocation policy: spread	precision: dgemm
			precision: sgemm
	THIN nodes	Threads allocation policy: close	precision: dgemm
			precision: sgemm
		Threads allocation policy: spread	precision: dgemm
			precision: sgemm

Figure 9: All the scenarios considered to compare performance

2.2 Implementation

2.2.1 BLIS library

We started by downloading and compiling the BLIS library as suggested by the instructions and updated the path to BLIS in the Makefile.

```
srun --partition=EPYC -n1 ./configure --enable-cblas --
 enable-threading=openmp --prefix=/u/dssc/gcosta00/
 assignment/exercise2/blis auto
srun  -n 1 --cpus-per-task=128 make -j 128
make install
```

Since in the development of this exercise we assessed the performance of BLIS library on both EPYC and THIN nodes, to get the best out of BLIS we compiled the library on both architectures.

2.2.2 *gemm.c* function

We slightly edited the *gemm.c* file provided by the exercise so that, in the compilation step, we could save the output data in CSV files, using the option `WRITE_CSV`. In the CSVs, we collected the elapsed time and the GFLOPS, and calculated the mean and the standard deviation for both. This was possible thanks to another option we included in the *gemm.c* file, `MULTIPLE_ITERATIONS`, which repeats the trials 30 times.

2.2.3 Makefile

We made changes to the file provided by the assignment, enabling the compilation of *gemm.c* and the use of `WRITE_CSV` and `MULTIPLE_ITERATIONS`. We also allowed for the compilation in both single and double precision and indicated in which directory the executables had to be generated.

2.2.4 Shell script files

We have written several shell scripts to run the matrices multiplication within the time limit set for the cluster manager.
For some of the 16 scenarios considered, our *gemm.c* program would take longer than the allowed two hours, so, for those cases, we edited the scripts to work with a single library at the time.
For some configurations of fixed size and fixed cores, we also spread the computation over multiple .sh files. For example, for Fixed Cores, we used one file to compute GFLOPS of matrices going from a size of 2000 to a size of 11000, and another file for matrices going from a size of 12000 to a size of 20000.
We assume that this long time needed has to be attributed to the fact that we iterated the matrix-matrix multiplication thirty times.

Below you can find examples of shell scripts for performing the benchmark on a fixed number of cores or fixed size of the matrices.

Fixed number of cores

The following is a script example for the three libraries together, with close policy, and run in EPYC architecture:

```
1    #!/bin/bash
2    #SBATCH --no-requeue
3    #SBATCH --job-name="Epyc_FixedCores_close"
4    #SBATCH -n 64
5    #SBATCH -N 1
6    #SBATCH --get-user-env
7    #SBATCH --partition=EPYC
```

```
    #SBATCH --exclusive
    #SBATCH --time=02:00:00
    #SBATCH --output="summary.out"

    # Load required modules
    module load architecture/AMD
    module load mkl
    module load openBLAS/0.3.23-omp

    # Add BLIS library path to LD_LIBRARY_PATH
    export LD_LIBRARY_PATH=/u/dssc/gcosta00/assignment/
    exercise2/blis/lib:$LD_LIBRARY_PATH

    # Store the current working directory
    directory=$(pwd)

    # Set OpenMP policy, threads, and binding
    policy=close
    arch=EPYC #architecture

    # Function to clean and compile
    clean_and_compile() {
      cd "$1"
      make clean loc="$2"
      make all loc="$2"
      cd "$2"
    }

    # Function to run benchmarks
    run_benchmarks() {
        for lib in openblas mkl blis; do
            for prec in float double; do
                file="${lib}_${prec}.csv"
                echo "matrix_size,time_mean(s),time_sd,
    GFLOPS_mean,GFLOPS_sd" > "$file"
    done
  done

  for i in {0..18}; do
    let size=$((2000+1000*$i))
    for lib in openblas mkl blis; do
      for prec in float double; do
        echo -n "${size}," >> "${lib}_${prec}.csv"
        "./${lib}_${prec}.x" "$size" "$size" "$size"
      done
    done
  done
}

    # Run tasks
    clean_and_compile ../../.. "$directory"
    cd "$directory"

    export OMP_PLACES=cores
    export OMP_PROC_BIND="$policy"
    export OMP_NUM_THREADS=64
```

```
8
9       run_benchmarks
10
11      clean_and_compile ../../.. "$directory"
12      module purge
```

Fixed size of the matrices

In this case, we wrote an individual shell script for each library.
The following is a script example for the MKL library, with spread policy, and run in THIN architecture:

```
1       #!/bin/bash
2       #SBATCH --no-requeue
3       #SBATCH --job-name="THIN_mkl_FixedSize_spread"
4       #SBATCH -n 24
5       #SBATCH -N 1
6       #SBATCH --get-user-env
7       #SBATCH --partition=THIN
8       #SBATCH --exclusive
9       #SBATCH --time=02:00:00
10      #SBATCH --output="summary_1.out"
```

```
1       # Load required modules
2       module load architecture/AMD
3       module load mkl
4       module load openBLAS/0.3.23-omp
```

```
1       # Add BLIS library path to LD_LIBRARY_PATH
2       export LD_LIBRARY_PATH=/u/dssc/gcosta00/assignment/
        exercise2/blis/lib:$LD_LIBRARY_PATH
```

```
1       # Store the current working directory
2       directory=$(pwd)
```

```
1       # Set OpenMP policy, threads, and binding
2       policy=spread
3       arch=THIN #architecture
```

```
1       # Function to clean and compile
2       clean_and_compile() {
3         cd "$1"
4         make clean loc="$2"
5         make all loc="$2"
6         cd "$2"
7       }
```

```
1       # Function to run benchmarks
2       run_benchmarks() {
3           for lib in mkl; do
4               for prec in float double; do
5                   file="${lib}_${prec}.csv"
6                   echo "matrix_size,time_mean(s),time_sd,
        GFLOPS_mean,GFLOPS_sd" > "$file"
```

```
    done
  done

  for i in {0..18}; do
    let size=$((2000+1000*$i))
    for lib in mkl; do
      for prec in float double; do
        echo -n "${size}," >> "${lib}_${prec}.csv"
        export OMP_PLACES=cores
        export OMP_PROC_BIND="$policy"
        export OMP_NUM_THREADS="$cores"
        "./${lib}_${prec}.x" "$size" "$size" "$size"
      done
    done
  done
}
```

```
    # Run tasks
    clean_and_compile ../../.. "$directory"
    cd "$directory"

    run_benchmarks

    clean_and_compile ../../.. "$directory"
    module purge
```

To see the other shell scripts, please refer to our GitHub repository[5].

2.2.5 Theoretical Peak Performance

A request of the assignment was to compare the performance of the three libraries with the Theoretical Peak Performance (TPP). The TPP represents the maximum computational capability, so the maximum number of floating point operations possible on a given hardware platform under ideal conditions. It can be computed as:

$$TPP = \#cores \cdot clock\ rate \cdot \frac{FLOPS}{cycle}$$

Where:

- #cores indicates the number of cores contained in a socket of the EPYC or of the THIN node,
- Clock rate is the number of cycles that a CPU performs per second,
- $\frac{FLOPS}{cycle}$ indicates the number of single or double-precision FLOPS that each core can deliver per cycle.

EPYC nodes have 64 cores per socket, a clock rate of 2.6 GHz, and can perform 32 single precision FLOPS per cycle and 16 double precision FLOPS per cycle. The Theoretical Peak Performance hence is:

[5] GitHub repository

- double: $TPP = 64 \cdot 2.6GHz \cdot 16 = 2662.4\ GFLOP$
- float: $TPP = 64 \cdot 2.6GHz \cdot 32 = 5324.8\ GFLOPS$

THIN nodes have 12 cores per socket and the Theoretical Peak Performance[6] is:

- double: $TPP = 1997\ GFLOP$
- float: $TPP = 3994\ GFLOPS$

2.3 Results

2.3.1 Scalability over matrix size

Let's start by comparing the performance of the three libraries and the TPP when keeping the number of cores fixed and increasing the size of the matrices (in this case the TPP is constant since the number of cores doesn't change).

EPYC NODES

In the next four graphs, we are in the context of EPYC nodes and we show, side to side, the results obtained when using the *gemm.c* function with single and double precision, introducing the case of close thread allocation policy first and then the spread policy one.

[6] Based on slide 63 of Lecture2: HPC-Hardware here

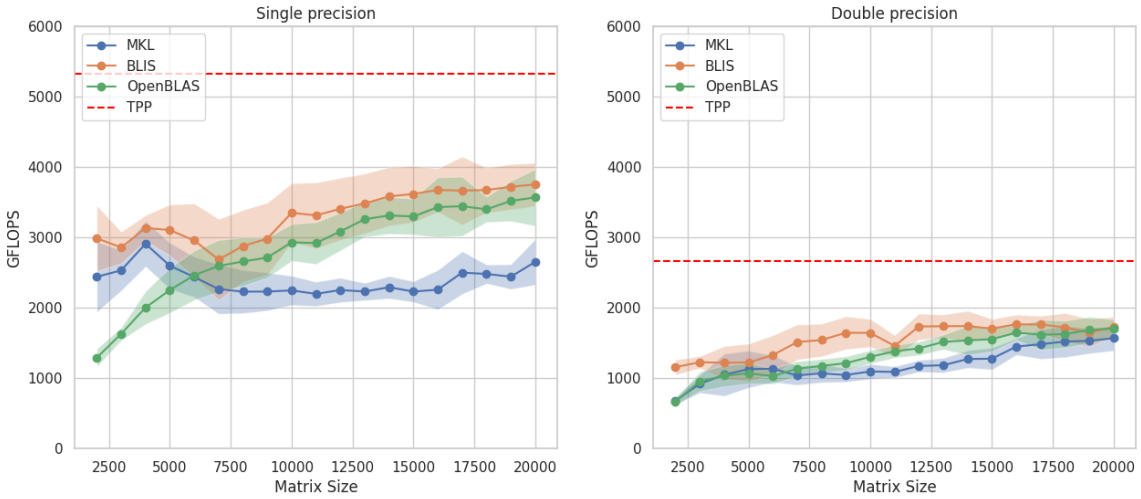

Figure 10: Fixed cores, EPYC nodes, **close** policy, single and double precision

In the case of close policy, we can see that all three libraries perform below the TPP, both for single and double data types.
In the case of single precision, MKL appears to scale badly, since its GFLOPS are almost constant (except for some moderate peaks) despite the increasing size of the matrices. The performance of BLIS is always higher than the one of the other two libraries, nevertheless, OpenBLAS has the fastest growing rate of GFLOPS, having the most accentuated slope shown in the graph. In the case of double precision though, this better scalability seems to be absent: BLIS remains the library with the highest number of giga floating point operations per second, but none of the three libraries scales significantly better than the others.

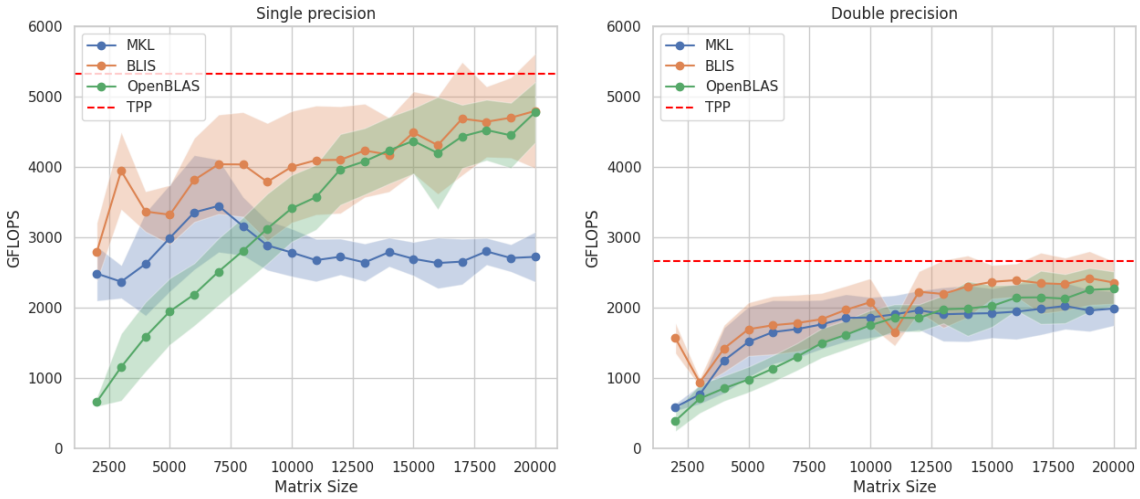

Figure 11: Fixed cores, EPYC nodes, **spread** policy, single and double precision

What we have seen in the graphs representing the close thread allocation policy seems to be partially repeated in the case of spread allocation when comparing the three libraries against each other: MKL doesn't show any particular scaling ability, while OpenBLAS is the one that has the highest increasing rate of GFLOPS, and BLIS keeps being the one with the highest performance at almost any size of the matrices. Nevertheless, when comparing the three libraries against the TPP, the performance in the spread case is much higher than the one in the close case, with BLIS reaching the theoretical peak performance in some of the iterations and OpenBLAS getting very close to it.
The same is evident in the case of double precision: none of the three libraries seems to perform significantly better than the others, but all of them have higher GFLOPS (hence get closer to the TPP) compared to the close case, and the scaling seems to grow faster (at least when the matrix size is up to 5000x5000 for MKL and BLIS, and up to 10000x1000 for OpenBLAS).

THIN NODES

Remaining in the setting of scalability over matrix size, let's now consider the case of THIN nodes, showing again the difference between float and double precision data types when opting for close or spread thread allocation policy.

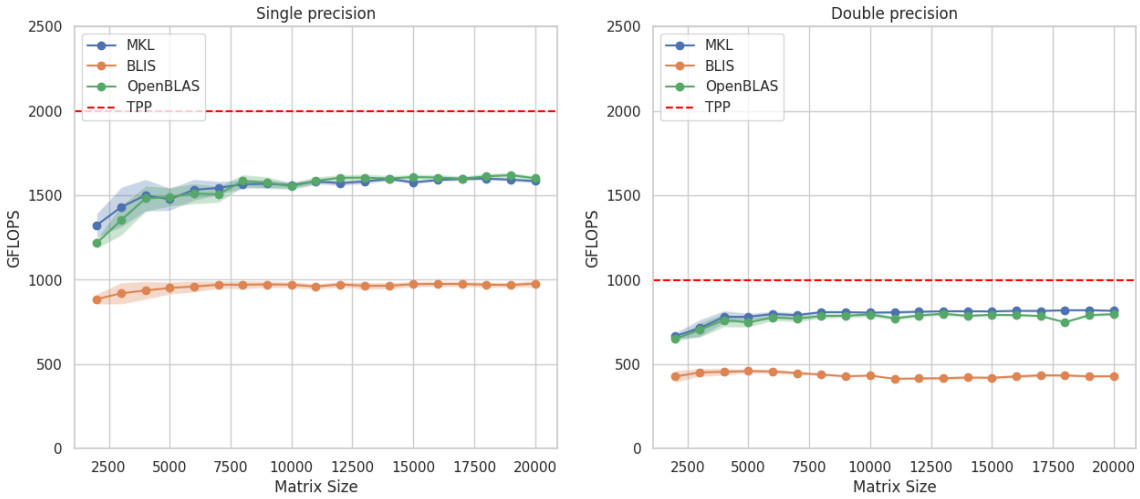

Figure 12: Fixed cores, THIN nodes, **close** policy, single and double precision

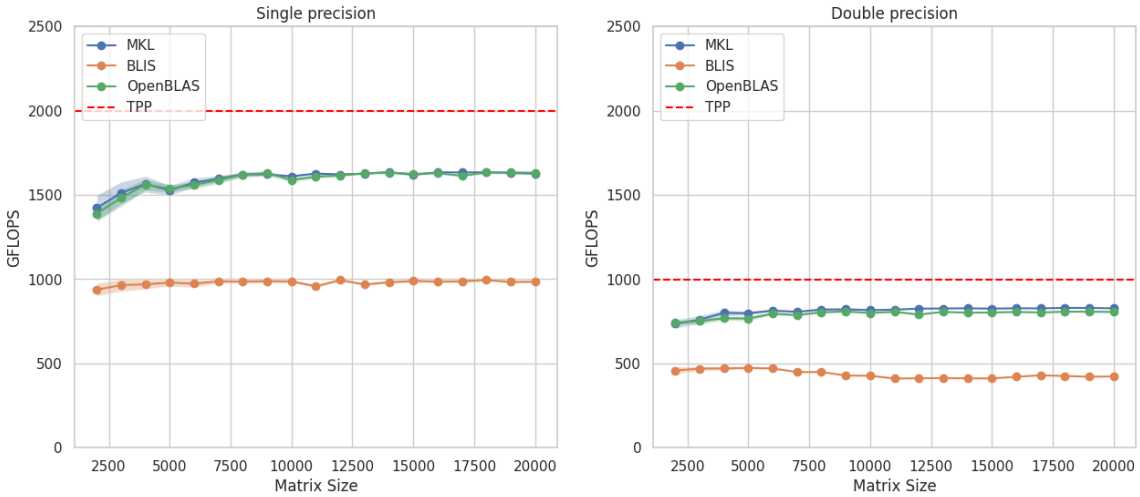

Figure 13: Fixed cores, THIN nodes, **spread** policy, single and double precision

The two sets of graphs shown above are almost identical, showing that there is almost no difference between close and spread thread allocation. For both single and double precision, we can see that the BLIS library performs bellow the TPP and worse than MKL and OpenBLAS, which have an almost identical performance instead.
The only (quite moderate) sign of good scalability can be noticed in MKL

and OpenBlas when the size of the matrices is very small, from 2000x2000 to around 4000x4000. After that, the performance is very flat, with the GFLOPS not increasing with bigger matrices.

2.3.2 Scalability over the number of cores

We considered the case where we fixed the matrix size to 10000x10000 and increased the number of cores, using up to all the 128 in the EPYC nodes and up to 24 in the THIN ones.

For this case, we decided to measure the performance of the three libraries using both the GFLOPs and the Speedup.

- **The GFLOPs** quantifies the computational efficiency of each library, allowing us to compare how well each of them utilizes the available CPU cores for performing matrix multiplications. We calculated the GFLOPs starting from the overall Theoretical Peak Performance calculated at paragraph 2.2.5, and deriving the ideal GFLOPS for each individual core:
  - EPYC
    - float: TPP per core = 83.2 GFLOPS
    - double: TPP per core = 41.6 GFLOPS
  - THIN
    - float: TPP per core = 166.4GFLOPS
    - double: TPP per core = 83.2GFLOPS

  We then multiplied the values above by the number of cores that we had at each step of the scaling test, so that we could have a TPP to be used as a benchmark.
- **The speedup** measures how faster an application that uses multiple cores runs compared to the same application running on a single core.
  - It was calculated following the definition:

$$Speedup_n = \frac{T_{serial}}{T_{parallel\_n}}$$

EPYC NODES

GFLOPs

Coherently with what done so far, we propose below a comparison of the results obtained with float and double precision data types, both in the case of close and spread thread allocation.

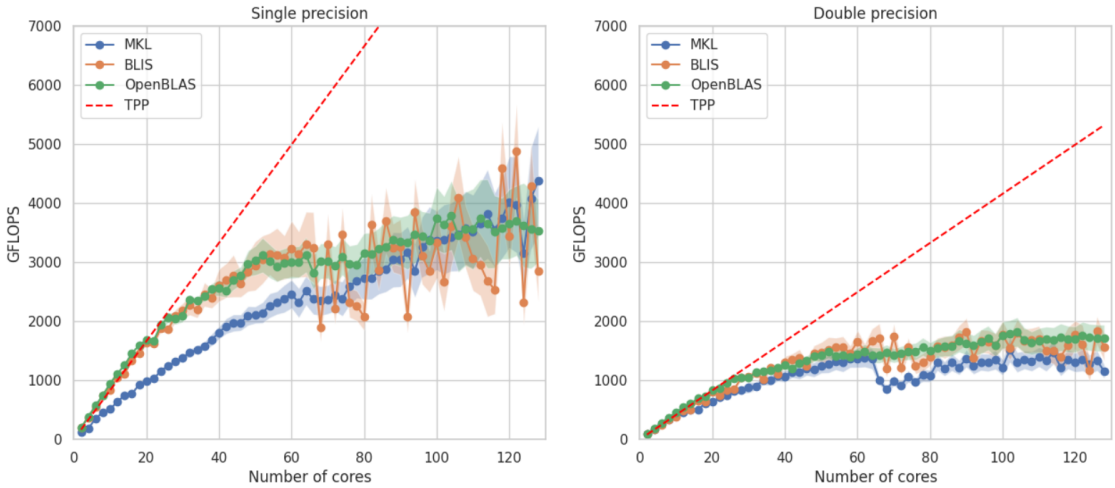

Figure 14: Fixed matrix size, EPYC nodes, **close** policy, single and double precision

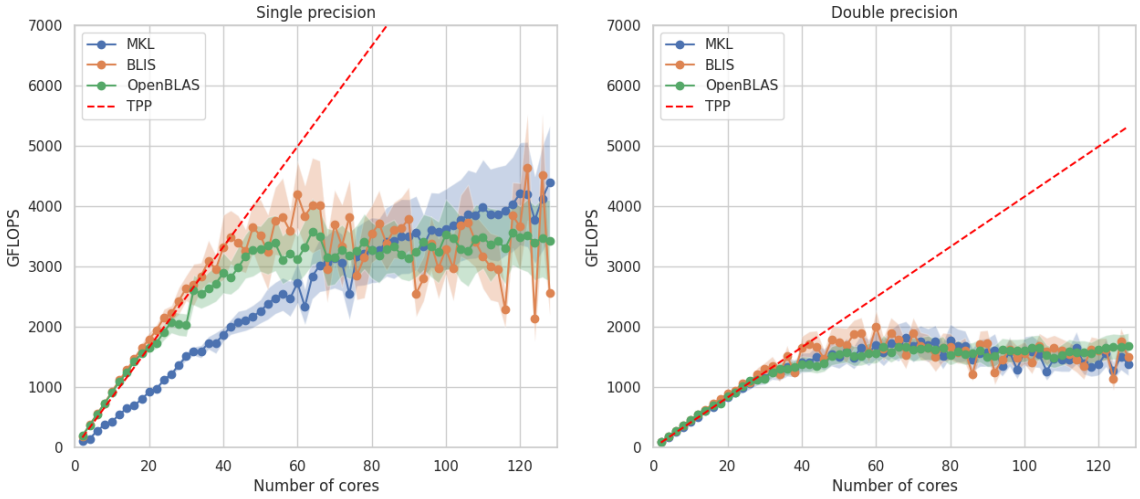

Figure 15: Fixed matrix size, EPYC nodes, **spread** policy, single and double precision

Focusing on comparing the close and the spread cases, we notice no major difference when evaluating the two upper graphs (close case) against the two lower ones (spread case).
In the case of spread policy, when using single precision (left graphs), we

have a higher variability in the data of OpenBLAS and BLIS compared to the close case, but, when we consider how each library performs against each other, the situation is quite similar. In fact, for both the close and the spread case we see that MKL performs worse than BLIS and OpenBLAS up to a certain number of cores (around 100 in the close case and around 80 in the spread one), but then outstands the other two when the number of cores increases. Nevertheless, except for the very initial stage, MKL always stands below the TPP, while BLIS and OpenBLAS show optimal performance when using up to around 20 nodes in the close setting, and up to around 40 in the spread one.

When focusing on the scenario of double data types, we see that MKL remains the overall least-performing library in the close case, while the differences among the three libraries are null in the spread one.
We can see that, for both thread allocation cases, MKL, OpenBLAS and BLIS have an optimal performance (so are on the same line as the TPP) up to a certain number of nodes (again, around 20 nodes for close and around 40 nodes for spread), but then all the three reach a plateau after that.

Speedup

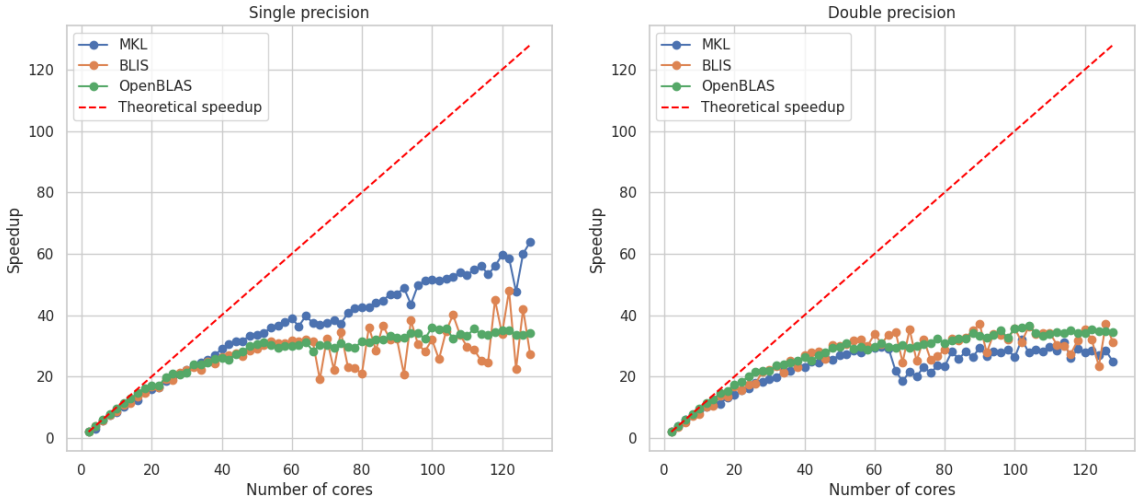

Figure 16: Fixed matrix size, EPYC nodes, **close** policy, single and double precision

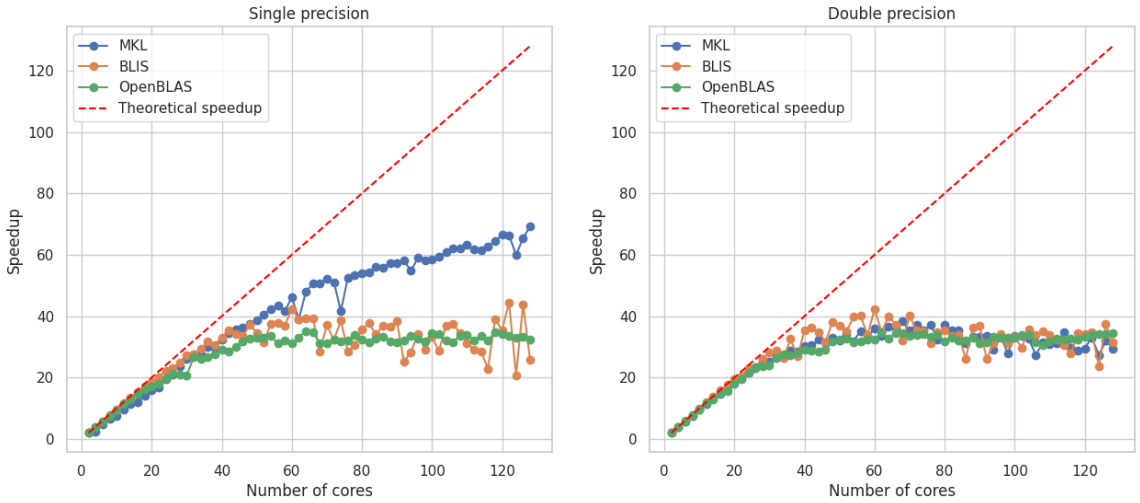

Figure 17: Fixed matrix size, EPYC nodes, **spread** policy, single and double precision

When considering the speedup, we can see that, for both close and spread cases, in the single precision scenario, MKL is the library that seems to benefit the most from the use of parallelism compared with BLIS and OpenBLAS, which have a very similar scaling and reach a plateau at around 40 cores.
For the double precision scenario, no library significantly outperforms the others. All three of them scale almost perfectly up to a little less than 20 cores in the close case, and up to a bit more than 20 cores in the spread one. After that point, their trend is almost the same and the use of additional cores doesn't reduce the runtime.

THIN NODES

GFLOPs

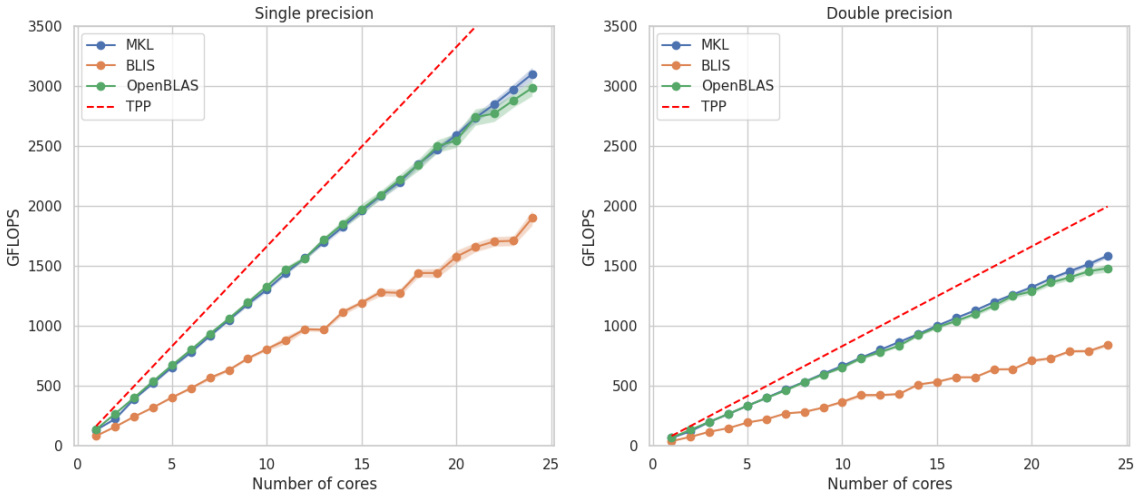

Figure 18: Fixed matrix size, THIN nodes, **close** policy, single and double precision

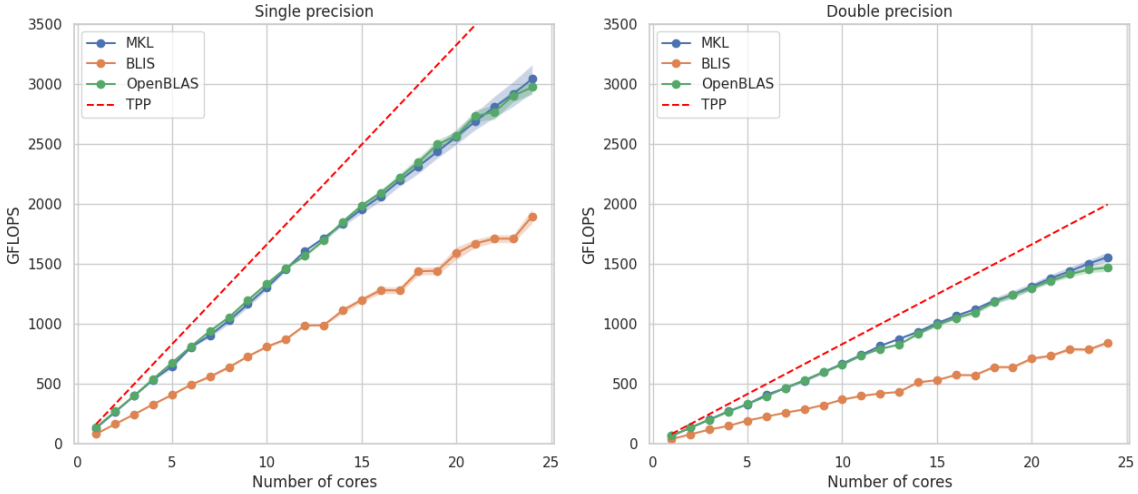

Figure 19: Fixed matrix size, THIN nodes, **spread** policy, single and double precision

For the case of THIN nodes, again we appreciate no relevant difference between the close and the spread thread allocation system. The behaviour of the three libraries seems to be quite similar when comparing single and double precision as well. All three have an optimal performance when using just one core, but then, as the number of cores increases, all of them get more and more far away from the Theoretical Peak Performance. BLIS

distances itself quite soon from TPP and more sharply, while OpenBLAS and MKL do it at a slower pace and have an almost identical performance up to the maximum number of cores used.

Speedup

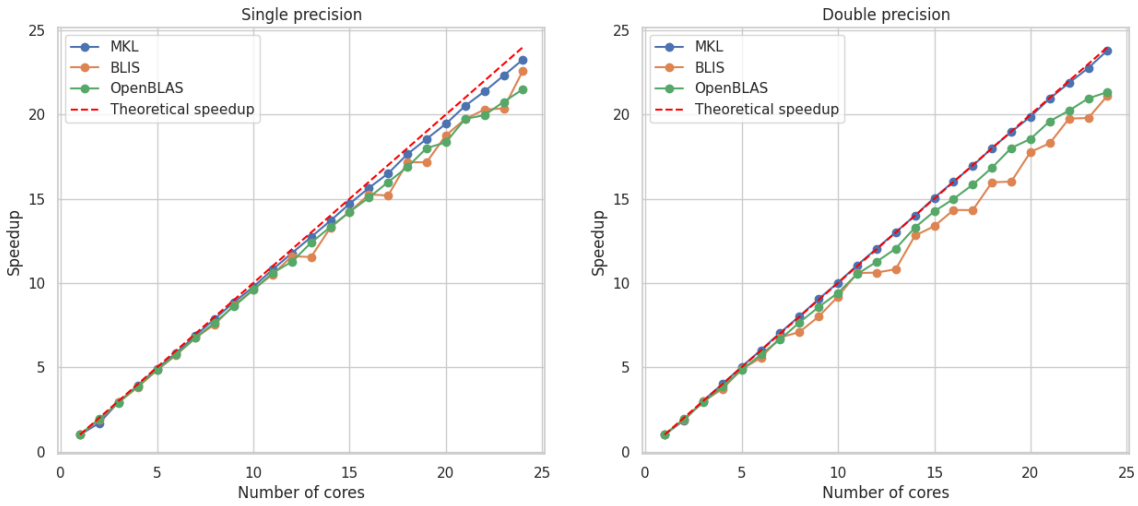

Figure 20: Fixed matrix size, THIN nodes, **close** policy, single and double precision

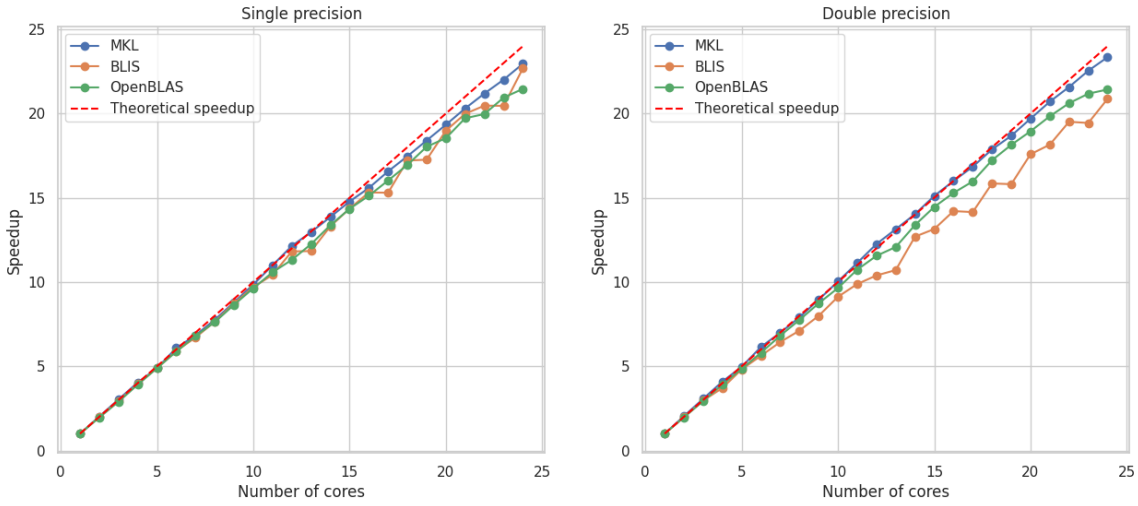

Figure 21: Fixed matrix size, THIN nodes, **spread** policy, single and double precision

When considering the speedup, for both the close and the spread case, we see that MKL seems to scale perfectly, especially in the double precision scenario. OpenBLAS and BLIS have the same performance when looking at the single precision, while, in the double precision case, BLIS scales worse than OpenBLAS, but they are nevertheless both always very close to the speedup line.

2.4 Conclusions

We can conclude that no library is always better than the others, and actually each one of them might have been designed to give its best in a specific setting.
For example, we can see that MKL tends to perform better when used with THIN nodes and this might be because it is optimized for Intel.

When scaling over the size of the matrices, in the case of EPYC nodes, a possible explanation of the significantly better performance of the spread over the close binding policy could be that, when we fix the number of cores to 64, with close policy, the threads are probably using the cores from one single socket. But when we use spread policy instead, the threads can work on both sockets, so we have better memory usage, with both the L3 caches available.
When considering the THIN nodes, we would expect an increasing number of GFLOPS when the size of the matrices increases, but this is not

reflected in the graphs. A reason for this could be that the maximum advantage of parallelism is reached fast and then, after that, the performance stays constant.

When scaling over the number of cores, again we see that the performance seems to increase initially and then reaches a plateau quite soon. This might be due to the fact that the matrix size (here 10000x10000) could be too small and consequently, the benefit of parallelization can be observed just for the very initial stage and then loses its value.

We could edit these tests trying to "stress out" the system more, by significantly increasing the size of the matrices or by reducing the number of cores used. This should better exploit the processing power and might show a more performing scaling.