# Foundations of High Performance Computing - Final Project Report

Simone Cappiello

July 18, 2023

## Contents

# 1 Game of Life

## 1.1 Introduction

The assignment required us to implement a parallel version of Conway's Game of life. Conway's Game of Life is a cellular automaton devised by British mathematician John Horton Conway. It's a zero-player game, meaning its evolution is determined by its initial state and doesn't require any further input. Each cell on an infinite grid lives, dies, or is born based on rules related to its adjacent cells: a cell is born if it has exactly three neighbors, survives if it has two or three living neighbors, and dies otherwise. I was prompted to implement two evolution modes for the game of life:

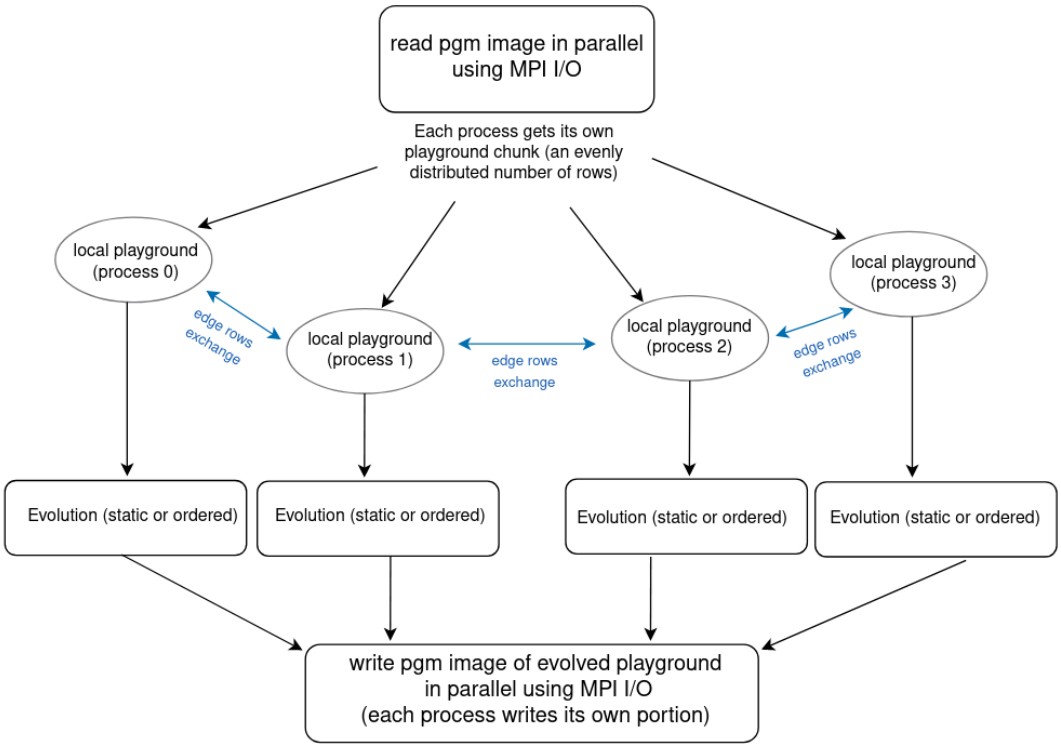
- static evolution: at each state $s_i$ the system is frozen and the evolution of cells to state $s_{i+1}$ is based only on the conditions of the previous state.
- ordered evolution: it's completely serial, the evolution starts from cell (0,0) and moves forward by lines in order (next cell to evolve would be (0,1) and so on...). Changes are immediate and affect the evolution of the next cells.

For more information see the references [1] [2].

.

## 1.2 Methodology

In this section I present the algorithms I designed for my implementation of the game of life. First of all, I chose to take care of how the playground is distributed among MPI tasks. In fact, one of the advantages of distributed parallelism in this case is the possibility to store chunks of the playground on different nodes instead of on a single one, which is significant especially when we are dealing with very large playgrounds. Conway's Game of Life is a cellular automaton simulation where the state of each cell in a grid is dependent on its immediate neighboring cells. Therefore, the simulation inherently has a spatial nature to it, which makes domain decomposition a more natural choice for parallelizing this problem. On the other hand, functional decomposition is less suitable for this problem. This is because the game involves repetitive application of the same rule to all cells in the grid. Functional decomposition would therefore not help in breaking the problem into independent parts, as there's essentially only one function that needs to be applied. The evolution rule in Conway's Game of Life depends on each cell's immediate neighbors, which are primarily in the same row. By dividing the domain into rows, we ensure that each process has immediate access to most of the data it needs, minimizing the need for inter-process communication. Therefore the domain decomposition strategy I adopted is row-wise: each process gets (in order) a chunk of rows. The following diagram illustrates the strategy I employed to distribute the workload and generate the PGM output of the evolved playground:




In this way, I ensured that no single process had to store the entire playground. The capacity to distribute the storage of the playground across multiple processes is the sole advantage of using MPI for ordered evolution, a problem that is inherently sequential and not suitable for parallelization.
The state of each cell at the next time step depends not just on its own current state, but also on the states of its eight neighboring cells. This includes the cells directly to the left and right, above and below, and on the diagonals. In other words, to compute the next state of a cell, you need to know the current states of the cells in a 3x3 neighborhood around it.
Now, consider a cell on the top row of a band managed by a process. To calculate its next state, this process needs to know the current states of the cells in the row above. But this row is part of the band managed by the process above it, so it's not directly available. The same goes for cells in the bottom row of the band, which need information about cells in the band below.
The solution is to create "ghost" rows that represent the top and bottom rows of the neighboring bands. At each time step, each process sends its top and bottom rows to its neighbors, which store these in their ghost rows. This way, each process has access to the information it needs to update all the cells in its band, including those on the top and bottom edges.
Following this discussion, I present the algorithms I developed for implementing static and ordered evolution.
**Note:** "n" indicates the number of generations to evolve the playground and "s" is the snapshot frequency, read [1] carefully to understand the notation used in this report.

**Algorithm 1 Static Evolution**

After the parallel reading of the playground, each process does the following:

1: Determine the ranks of the top and bottom neighbor processes in the ring topology.
2: Allocate bottom ghost row and top ghost row.
3: **for** $step = 1$ to $n$ **do**

- Allocate updated playground, it will contain the status of the local playground at state $s_{step+1}$.
- Initiate a non-blocking send of the top row of the local playground to the top neighbor and the bottom row to the bottom neighbor.
- Receive (blocking call) the bottom ghost row from the bottom neighbor and the top ghost row from the top neighbor.
- Use OpenMP (parallel for) to update the state of each cell in the local playground. For each cell in the current process's chunk of the playground:
  - **If** the cell is in the top edge row, **then** use the top ghost row as the top neighbor row. Otherwise, use the row above in the local playground.
  - **If** the cell is in the bottom edge row, **then** use the bottom ghost row as the bottom neighbor row. Otherwise, use the row below in the local playground.
  - Update the cell's state based on its own state and the states of its neighbors.
- **If** the current step is a multiple of $s$, **then** write a snapshot of the current state of the system to a file.
- Exchange pointers between updated playground and local playground, free updated playground.

4: Free the memory allocated for top ghost row and bottom ghost row.
5: **If** the last iteration status has not been written, **then** write its snapshot.

**Algorithm 2 Ordered Evolution**

After the parallel reading of the playground, each process does the following:

1: Determine the ranks of the top and bottom neighbor processes in the ring topology.
2: Allocate memory for bottom ghost row and top ghost row.
3: Perform initial exchange of ghost rows using blocking send/receive operations.
4: Initialize mpi_order to 0.
5: **for** $step = 1$ to $n$ **do**

- **While** the rank does not match mpi_order (integer variable used to correctly synchronize processes):
  - Receive (blocking) the updated mpi_order from the preceding process.
  - Receive (blocking) the top ghost row from the preceding process.
- **If** the rank matches the updated mpi_order, **then** perform the following steps:
  - **If** it is not the first step (step != 0) **or** the process is the last one (rank == size-1), **then**:
    - Initiate a non-blocking receive operation for the bottom ghost row from the next process.
    - Wait for the non-blocking receive operation to complete.
  - Determine the number of threads and distribute work among them.
  - For each thread, compute the portion of the local playground that it will handle.
  - For each cell in the designated portion, update its state.
- **If** the process is not the last one (rank != size-1), **then**:
  - Increment mpi_order and send it to the following process.
  - Send (blocking) the bottom row of the local playground as the top ghost row for the next process.
  - **If** it is not the last step (step != n) **or** the process is the first one (rank == 0), **then** initiate a non-blocking send of the top row of the local playground as the bottom ghost row for the previous process.
- Synchronize all processes using a barrier operation.
- **If** the current step is a multiple of $s$, **then** write a snapshot of the current state of the playground to a file.
- **If** the process is the last one (rank==size-1) **and** the current step is not the last step (step != n), **then**:
  - Reset mpi_order to 0 and send it to the first process.
  - Send the bottom row of the local playground as the top ghost row for the first process.
  - Initiate a non-blocking send of the top row of the local playground as the bottom ghost row for the previous process.

6: **If** the final state has not been written **and** $s$ does not equal to $n$, **then** write a snapshot of the final state of the system to a file.
7: Free the memory allocated for top ghost row and bottom ghost row.

## 1.3 Implementation

Here I delve deeper on the actual C implementation of the algorithms I briefly described above. The parallel region is instantiated inside the main function and the distribution of the workload takes place before running either static or ordered evolution.

```
MPI_Init(NULL, NULL);
int rank;
int size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int color_depth = 1 + (MAXVAL > 255);
int chunk = k / size;  //k is the size of the playground
int mod = k % size;
int my_chunk = chunk + (rank < mod);
int my_first = rank*chunk + (rank < mod ? rank : mod);
int portion_size = my_chunk * k * color_depth;
const int header_size = 23;
int my_offset = header_size + my_first * k * color_depth;
```

After computing these parameters, parallel MPI reading is performed:

```
unsigned char *playground_o = (unsigned char *)malloc(portion_size
                                * sizeof(unsigned char));
parallel_read_pgm_image((void **)&playground_o, fname,
                        my_offset, portion_size);
```

Where "parallel_read_pgm_image()" function allows each process to read its own portion of playground from the pgm image to evolve and store it in its version of "playground_o" using the offset each one of them computed independently.

```
void parallel_read_pgm_image(void **image, const char *image_name,
                             int offset, int portion_size) {
    MPI_File fh;
    MPI_Status status;
    MPI_File_open(MPI_COMM_WORLD, image_name, MPI_MODE_RDONLY,
                  MPI_INFO_NULL, &fh);
    MPI_File_seek(fh, offset, MPI_SEEK_SET);
    MPI_File_read(fh, *image, portion_size, MPI_UNSIGNED_CHAR, &status);
    MPI_Barrier(MPI_COMM_WORLD); // making sure all processes
    // read their portion before closing the file
    MPI_File_close(&fh);

}
```

**Static Evolution**

Since each process directly reads its own local playground beforehand, the only necessary MPI communication between processes after that is the exchange of ghost rows. To minimize communication overhead I opted for two non blocking sends matched with two blocking receives:

```
MPI_Request request[2];
```

```
// Each process sends its top row to its top neighbor
MPI_Isend(&local_playground[0], xsize, MPI_UNSIGNED_CHAR, top_neighbor, 0,
      MPI_COMM_WORLD, &request[0]);

// Each process sends its bottom row to its bottom neighbor
MPI_Isend(&local_playground[(my_chunk - 1) * xsize], xsize, MPI_UNSIGNED_CHAR,
          bottom_neighbor, 1, MPI_COMM_WORLD, &request[1]);

// Each process receives its bottom ghost row from its bottom neighbor
MPI_Recv(bottom_ghost_row, xsize, MPI_UNSIGNED_CHAR, bottom_neighbor, 0,
     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Each process receives its top ghost row from its top neighbor
MPI_Recv(top_ghost_row, xsize, MPI_UNSIGNED_CHAR, top_neighbor,
     1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

The blocking receive operations (MPI_Recv) ensure that the receiving process waits until the data has been fully received before continuing. This is important to ensure that the process doesn't start using the ghost rows before they have been fully updated.

After the exchange took place, each process uses OpenMP to further parallelize the work:

```
  #pragma omp parallel for collapse(2)
       for (int y = 0; y < my_chunk; y++)
         {
            for (int x = 0; x < xsize; x++)
            {
                update_cell_static((y == 0 ? top_ghost_row
                            &local_playground[(y - 1) * xsize]),
                            (y == my_chunk - 1 ? bottom_ghost_row :
                            &local_playground[(y + 1) * xsize]),
                            local_playground, updated_playground,
                            xsize, my_chunk, x, y);
            }
         }
```

Each iteration roughly brings the same computational work, therefore OpenMP default static schedule is more efficient since the direct and predictable assignment has a smaller overhead (the iteration is divided in chunks of homogeneous size and distributed to threads in circular order).

### Ordered evolution

Ordered evolution is a synchronization challenge that requires ensuring each computing unit performs its computations in the correct sequence. To manage this synchronization, I utilized the variable "mpi_order". This variable allows each process to recognize when it's their turn to compute, ensuring that only one process enters the computation region at any given time.

```
// Where all the processes wait for their turn.
while(rank != mpi_order) {
            MPI_Recv(&mpi_order, 1, MPI_INT, top_neighbor,
```

```
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            //THE (BLOCKING) CALL TO RECEIVE THE TOP GHOST ROW
            //FROM ITS PRECEDING PROCESS
            MPI_Recv(top_ghost_row, xsize, MPI_UNSIGNED_CHAR,
                    top_neighbor, 4, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
        }
```

When its turn arrives the process enters the computation region, updates its chunk, sends updated mpi_order to the process that follows and its edge rows to its neighbors. The OpenMP region follows a similar logic of the MPI one.

```
if (rank == mpi_order) {
            if(step != 1 || rank == size-1){
            // non blocking receive followed by a wait.
            // Only the last process (of rank=size-1) needs the
            // bottom ghost row from process 0 at first iteration
                MPI_Irecv(bottom_ghost_row, xsize,
                       MPI_UNSIGNED_CHAR, bottom_neighbor,
                       5, MPI_COMM_WORLD, &reqs[0]);

                MPI_Status status;
                MPI_Wait(&reqs[0], &status);

            }
        }
            // Perform computation using OpenMP...(omitted)

            if(rank != size-1) {
                mpi_order++;
                MPI_Send(&mpi_order, 1, MPI_INT, bottom_neighbor,
                     0, MPI_COMM_WORLD);
                // Send the bottom row as the top ghost row
                // for the next process
                MPI_Send(&local_playground[(my_chunk - 1) * xsize], xsize,
                                        MPI_UNSIGNED_CHAR, bottom_neighbor,
                                         4, MPI_COMM_WORLD);

                if(rank == 0 || step != n) {  // process of rank==size-1
                // needs its bottom ghost row
                // from process 0 even at the last iteration,
             // so in last iteration no process sends
               // its top row except for process 0

                  MPI_Isend(&local_playground[0], xsize,
                             MPI_UNSIGNED_CHAR, top_neighbor, 5,
                            MPI_COMM_WORLD, &reqs[1]);

                }
            }
```

```
        MPI_Barrier(MPI_COMM_WORLD);
        // Gather the local playgrounds back to the master process
        if(step % s == 0) {
           par_write_snapshot(local_playground, 255, xsize,
                          my_chunk, "osnapshot", step, offset);
       }

      if(rank == size-1 && step != n) {
          mpi_order = 0;
          MPI_Send(&mpi_order, 1, MPI_INT,
                   bottom_neighbor, 0, MPI_COMM_WORLD);
          // last process sends to the first one
          // Send the bottom row as the top ghost
          // row for the next process (process 0 in this case)
          MPI_Send(&local_playground[(my_chunk - 1) * xsize],
                                     xsize, MPI_UNSIGNED_CHAR,
                                     bottom_neighbor, 4, MPI_COMM_WORLD);

          MPI_Isend(&local_playground[0], xsize,
                    MPI_UNSIGNED_CHAR, top_neighbor, 5,
                    MPI_COMM_WORLD, &reqs[2]);

      }
```

### Output validation

To verify the accuracy of the code, I initially created a small PGM image (8x8 pixels) and evolved it for one generation using a serial version of the Game of Life, manually validating the result through visualization. Subsequently, I tested the parallel versions with a greater number of generations and larger grids. The command line command I used was as follows:

`compare -metric AE image1.pgm image2.pgm difference.pgm`

In this command:

- `compare` is the command that compares the images.
- `-metric AE` is an option that tells `compare` to count the number of differing pixels (absolute error count) and print it to the standard error output.
- `image1.pgm` and `image2.pgm` are the images you want to compare.
- `difference.pgm` is the output image that will show where the images differ. Differing pixels will be highlighted.

If the images are the same, `compare` will output `"0"` (meaning there are no differing pixels). If they're different, `compare` will output the number of differing pixels and produce an image that shows where the differences are.

**Measuring performance**

```
gettimeofday(&start_time, NULL);
static_evolution(playground_s, k, my_chunk, my_offset, n, s);
MPI_Finalize();
gettimeofday(&end_time, NULL);
time_elapsed = (end_time.tv_sec - start_time.tv_sec) +
(end_time.tv_usec - start_time.tv_usec) / 1e6;

mean_time = time_elapsed / n;
if (rank == 0) {
FILE *fp = fopen("timing.csv", "a");
fprintf(fp, "%f\n", mean_time);
fclose(fp);
free(playground_s);
}
```

The above code is how I measured the performance of my program. The process begins by capturing the current wall clock time before the execution of the parallel evolution function using the `gettimeofday` function. After the function execution and finalizing the MPI environment, the end time is captured again. The difference between the end time and the start time gives the total time elapsed during the execution of the parallel function.

The mean time is then calculated by dividing the total time elapsed by the number of iterations. This mean time is written to a CSV file for further analysis. This operation is performed only by the process with rank 0 to avoid conflicts between different processes trying to write to the same file simultaneously.

I chose to measure wall clock time instead of CPU time because wall clock time provides a more accurate measure of the real time that a program takes to execute in a parallel computing environment. CPU time, on the other hand, measures the total time that the CPU spends executing the program, which can be misleading in a parallel environment where multiple processes or threads may be executing simultaneously on different CPUs.

## 1.4 Results and Discussion

Performance is quantified in terms of speedup, which is defined as follows:

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

In this equation, $T_{serial}$ represents the wall clock execution time for the evolution when the number of tasks (which corresponds to the number of threads in OpenMP or processes in MPI) is set to 1. On the other hand, $T_{parallel\_n}$ denotes the execution time when the number of tasks is increased to $n$.

The speedup metric provides a measure of the improvement in performance due to parallelization. A speedup of $n$ implies that the parallel version of the program is $n$ times faster than the serial version. In an ideal scenario, the speedup would be linear, i.e., doubling the number of tasks would halve the execution time (Theoretical speedup, which I always represented as a dashed line in the plots).

### 1.4.1 OpenMP scalability

Here I fixed the size of the playground to 25000x25000 and the number of generations n=50 (and n=5 for ordered evolution to accelerate the execution). In the batch file I set one MPI process per socket and gradually increased the number of threads (each thread runs on a core) up to the maximum number (64 cores per socket for Epyc nodes and 12 cores per socket for Thin nodes). The batch file for Thin nodes I used is the following, the one for Epyc is very similar:

```
#SBATCH --no-requeue
#SBATCH --job-name="openMP_scal"
#SBATCH --partition=THIN
#SBATCH -N 1
#SBATCH -n 24
#SBATCH --exclusive
#SBATCH --time=02:00:00

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

loc=$(pwd)
cd ../..
make par location=$loc
cd $loc

processes=2
size=25000

datafile=$loc/timing.csv
echo "threads_per_socket, ordered_mean, static_mean" > $datafile

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


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


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

On Epyc nodes the speedup slightly decreases with the increase of cores. This is probably caused by the increase in probability of false sharing and cache coherence issues. The performance would probably improve if we increase the size of the playground.

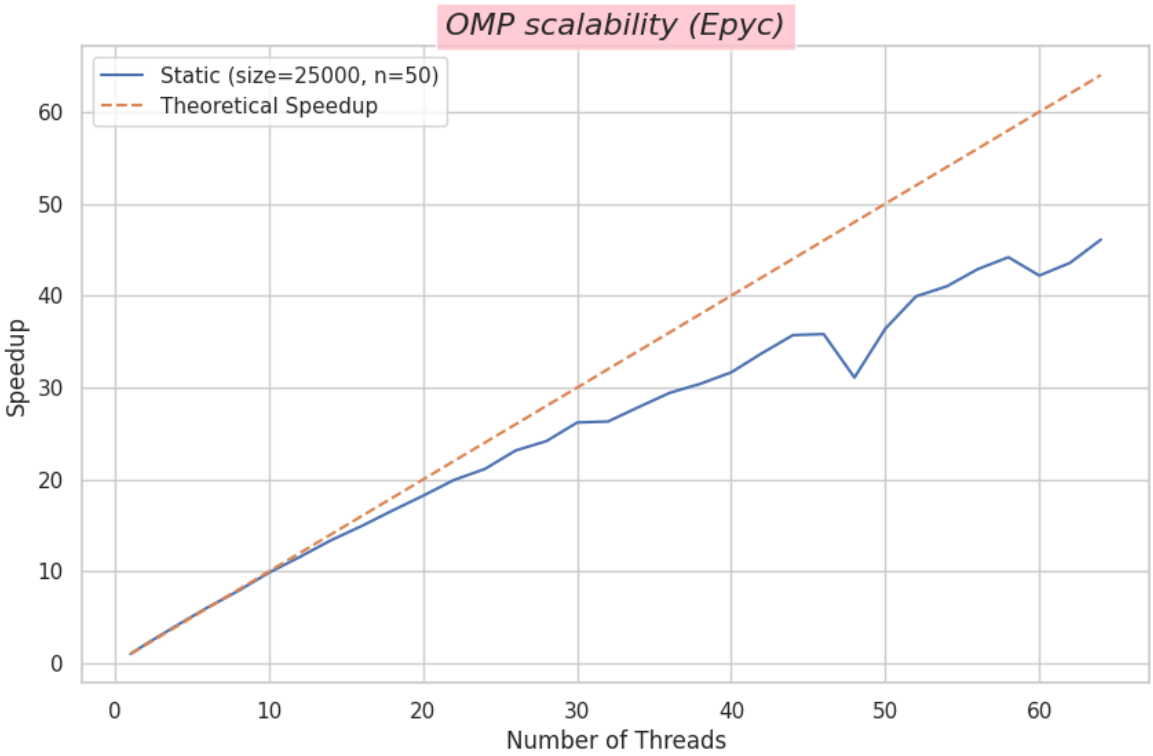



Figure 1: OpenMP scalability for static evolution, Epyc

The speedup of ordered evolution is almost constant with a little drop as the number of threads increases due to added synchronization overhead.

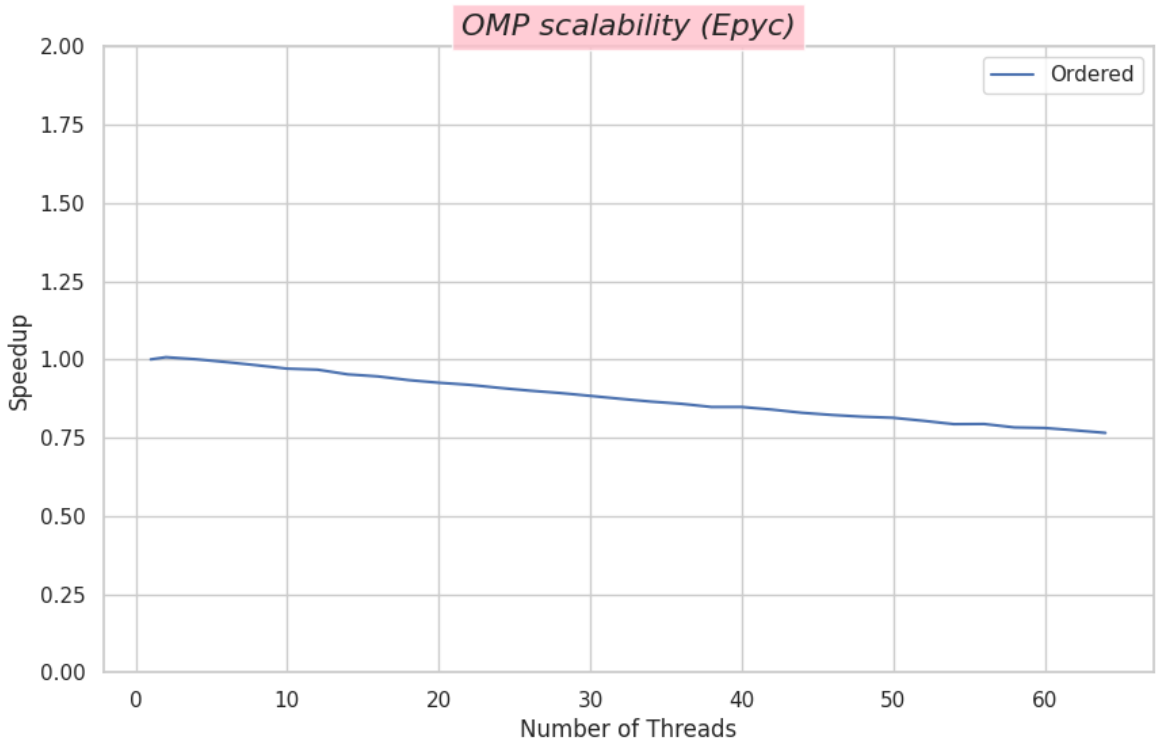



Figure 2: OpenMP scalability for ordered evolution, Epyc

On the Thin nodes we observe near-perfect speedup; the number of threads scales appropriately with the size of the playground.

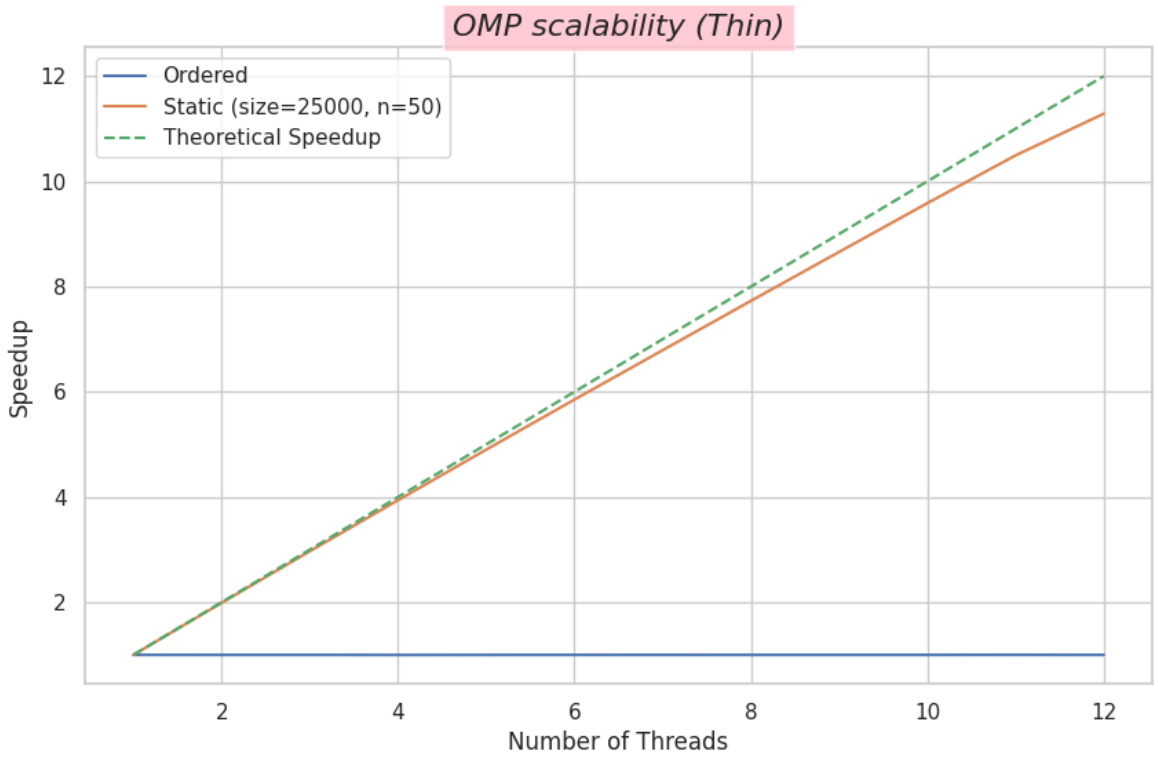



Figure 3: OpenMP scalability, Thin

### 1.4.2 MPI strong scalability

I set the number of OMP threads to one and mapped MPI tasks on cores, increasing them from 1 to the maximum possible on the node. I allocated two nodes. Here's a snippet of the batch file I wrote:

```
export OMP_NUM_THREADS=1
size=25000


mpirun -np 4 -N 2 --map-by socket main.x -i -f
        "playground_${size}.pgm" -k $size
for procs in 1 $(seq 8 8 256)
do
          echo -n "${size}," >> $datafile
          echo -n "${procs},">> $datafile

   mpirun -np $procs -N 2 --map-by core main.x -r
              -f "playground_${size}.pgm" -e 0 -n 3 -s -k $size
```

```
        mpirun -np $procs -N 2 --map-by core main.x -r -f
            "playground_${size}.pgm" -e 1 -n 100 -s 0 -k $size
done
```

The observed scalability of the code improves with the increase in the size of the playground and the number of iterations. This is primarily because the overhead associated with synchronization becomes less significant compared to the benefits gained from the introduction of additional parallel processing units as the size of the PGM image expands. Thin nodes show a better scalability for this playground size (25000x25000). To achieve optimal scaling with Epyc nodes, it's necessary to use larger playground sizes. As the number of processes increases, the playground size should also be enlarged to draw nearer to optimal speedup.

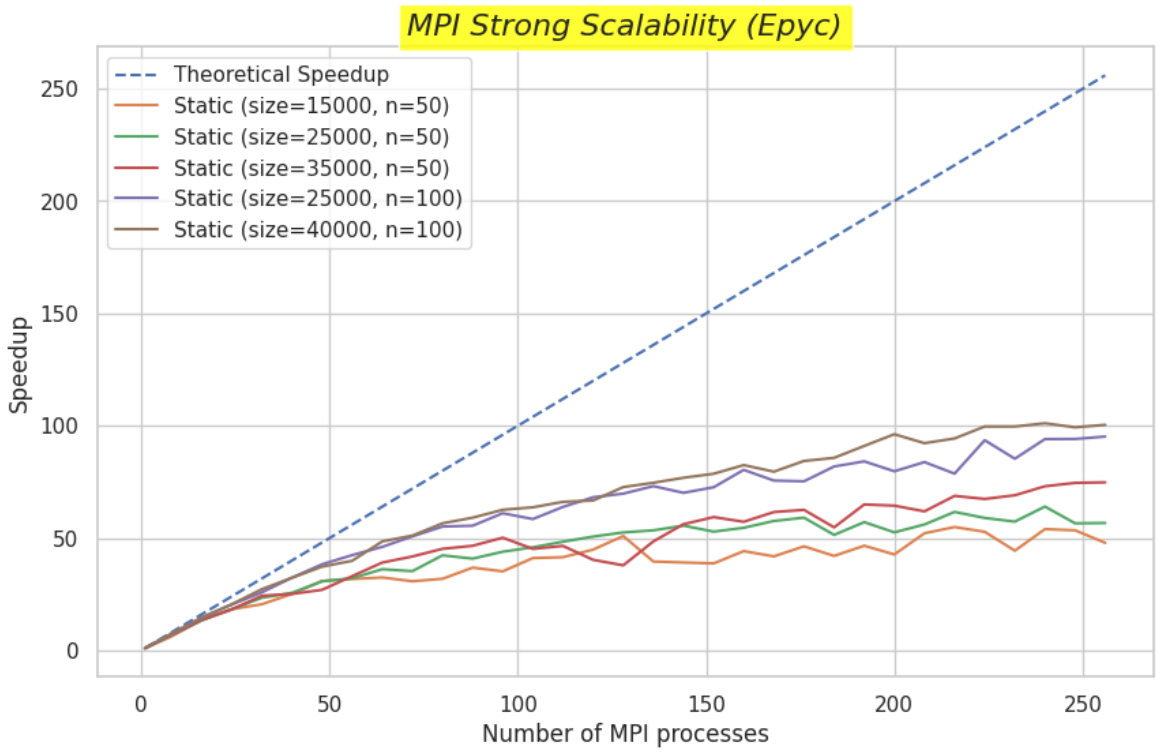



Figure 4: MPI strong scalability, Epyc

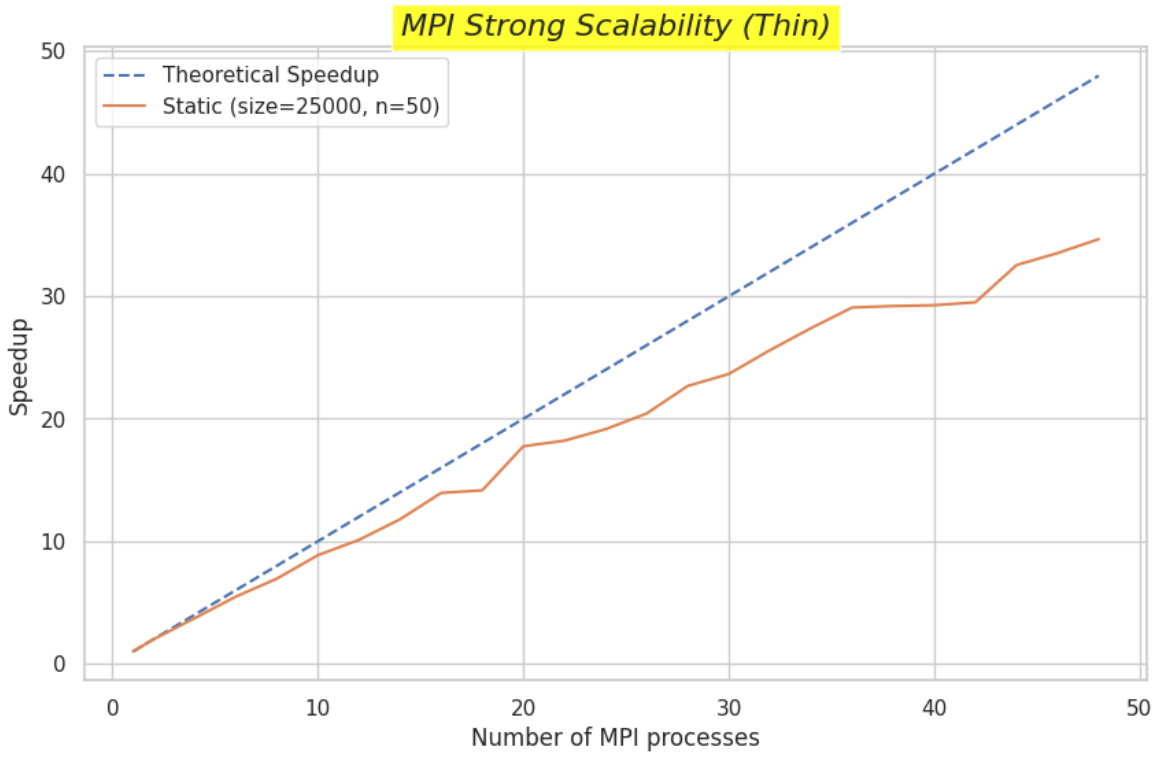



Figure 5: MPI strong scalability, Thin

### 1.4.3 MPI weak scalability

The purpose of weak scalability is to maintain the execution time constant as we keep the workload per task constant by increasing the size of the playground along with the number of processes. We start with a 10000x10000 size for 1 process. $10000x10000 = 10^8$ *bytes* is the size each process has to deal with. Applying the formula $size = \sqrt{n \cdot 10^8}$, where $n$ is the number of processes used, we obtain the following sizes:

| MPI Processes | Size |
|---|---|
| 1 | 10000 |
| 2 | 14143 |
| 3 | 17321 |
| 4 | 20000 |
| 5 | 22361 |
| 6 | 24495 |

Table 1: MPI Processes and Corresponding Sizes

For this study I fixed the number of OpenMP threads to 20 per socket and mapped the MPI processes by socket, I used 3 nodes for both Epyc and Thin.

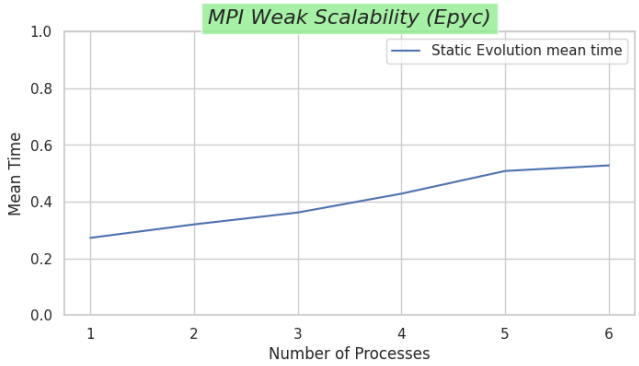



Figure 6: MPI weak scalability, Epyc

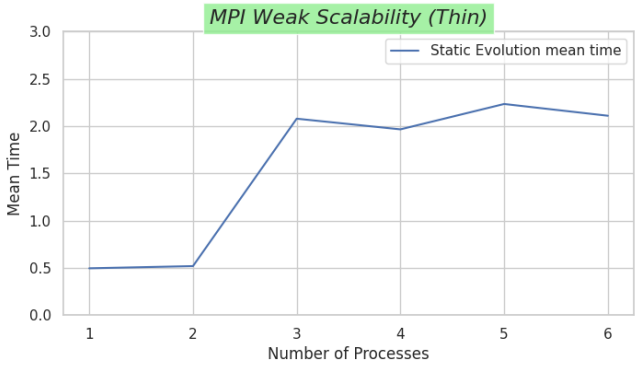



Figure 7: MPI weak scalability, Thin

As we increase the number of MPI processes, we also introduce additional communication overhead. This effect becomes particularly pronounced and significantly impacts performance when communication is required between different nodes.

### 1.5 Conclusions

The obtained results are generally satisfactory. Further investigations could be made into the strong scalability of MPI, particularly with larger sizes. Given the time constraints associated with node usage, the most feasible approach would be to conduct tests on rectangular matrices that have more rows than columns. However, the current version of the code is limited to implementing the Game of Life for square matrices. Other interesting experiments would using the 'funneled' mode for hybrid OpenMP/MPI code, and exploring different MPI communication methods, such as using 'ready send' instead of the standard send. These tests could provide valuable insights into performance differences.

## 2 Comparing MKL, OpenBLAS and BLIS on matrix-matrix multiplication

### 2.1 Overview

The second exercise of the assignment consisted in comparing the performance of three HPC math libraries: MKL, openBLAS and BLIS. It was required to perform a level 3 BLAS operation, which is a (square in our case) matrix-matrix multiplication of this form (in our case $\alpha = 1$ and $\beta = 0$):

$$C \leftarrow \alpha AB + \beta C$$

. I had to study (and compare) the scalability of this operation in two different contexts on Epyc and Thin nodes, using both double and single floating point precision, with different thread affinity policies (I chose close and spread). The two scenarios to be analyzed were:

- Fix the number of cores (64 for Epyc and 12 for Thin) and **scale over the matrix size** of A, B and C. I opted to collect data of matrices going from size 2000 to size 20000 with 1000 as interval between one data point and the other (2000, 2100, ..., 19000, 20000).

- Fix the matrix size to 10000 and **scale over the number of cores**. For Epyc nodes I gathered data using cores from 1 to 128 with step 2 (1, 2, 4, 6, ..., 126, 128), for Thin nodes I did the same but of course stopped at 24, which is the maximum number of cores per node.

## 2.2 Procedure

I have modified the gemm.c file in order to compute the mean and standard deviation of the required measures over 5 runs with fixed parameters (same number of threads or same matrix size for all 5 iterations) and to store the results into .csv files. An example of the obtained output is in Table 2.

| matrix size | time mean (s) | time sd | GFLOPS mean | GFLOPS sd |
|---|---|---|---|---|
| 2000 | 0.026054 | 0.013363 | 671.433262 | 113.179618 |
| 3000 | 0.060946 | 0.005347 | 892.087555 | 69.591691 |
| 4000 | 0.172274 | 0.032275 | 767.737189 | 134.296256 |
| 5000 | 0.368069 | 0.032896 | 684.343216 | 57.408408 |
| 6000 | 0.591384 | 0.07526 | 740.894678 | 82.02083 |
| 7000 | 0.874059 | 0.134862 | 799.761448 | 97.05355 |
| 8000 | 1.295007 | 0.144449 | 798.624859 | 71.255828 |
| 9000 | 1.847016 | 0.196494 | 796.511747 | 67.300123 |
| 10000 | 2.416993 | 0.174065 | 831.040951 | 49.618134 |
| 11000 | 3.25681 | 0.207115 | 820.141351 | 43.740717 |
| 12000 | 4.188604 | 0.354623 | 829.704896 | 54.629586 |
| 13000 | 5.326629 | 0.434216 | 829.202909 | 52.772138 |
| 14000 | 7.128977 | 0.653726 | 775.168786 | 58.454522 |
| 15000 | 8.073415 | 0.448802 | 838.283124 | 39.730695 |
| 16000 | 9.694578 | 0.518489 | 847.084908 | 38.870534 |
| 17000 | 12.364251 | 1.092453 | 800.095669 | 61.201615 |
| 18000 | 13.66215 | 0.694641 | 855.699881 | 38.473638 |
| 19000 | 16.966753 | 1.537044 | 814.006856 | 60.843782 |
| 20000 | 10.940399 | 3.134184 | 1522.223718 | 212.179985 |

Table 2: Example of output (this is MKL on EPYC using double precision and close as thread affinity policy). The mean and standard deviation are computed over 5 iterations for each matrix size.

In order to compute these quantities I had to write a batch job file I named "my_job.sh". Inside it I asked for the allocation of the necessary resources, loaded the modules, setup the csv files and run the programs by varying the parameters. An example follows (studying matrix size scalability on EPYC nodes):

```
#!/bin/bash
#SBATCH --no-requeue
#SBATCH --job-name="ex2"
```

```
#SBATCH -n 64
#SBATCH -N 1
#SBATCH --get-user-env
#SBATCH --partition=EPYC
#SBATCH --exclusive
#SBATCH --time=02:00:00

module load architecture/AMD
module load mkl
module load openBLAS/0.3.21-omp
export LD_LIBRARY_PATH=/u/dssc/scappi00/myblis/lib:$LD_LIBRARY_PATH

location=$(pwd)

cd ../../..
make clean loc=$location
make cpu loc=$location

cd $location
policy=close
arch=EPYC

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

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

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

I wrote a job file for each combination described above.

## 2.3 Theoretical Peak Performance

I had to compare the performance of the nodes with their theoretical peak performance (expressed in GFLOPS). I calculated the TPP using the following formula:

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

**EPYC**

Epyc nodes have up to 64 cores per socket and a frequency of 2.6 G Hz. Each core can deliver 32 single precision (float) FLOPS per cycle and 16 double precision (double) FLOPS per cycle.

- float: $TPP = 64 \cdot 2.6GHz \cdot 32\frac{FLOPS}{cycle} = 5324.8\ GFLOPS$
- double: $TPP = 64 \cdot 2.6GHz \cdot 16\frac{FLOPS}{cycle} = 2662.4\ GFLOPS$

**THIN**

Thin nodes have 12 cores per socket and we know that the theoretical peak performance of each node is 1997 TFLOPS in double point precision. Let's calculate the TPP per socket:

- float: $TPP = 1.997\ TFLOPS = 1997\ GFLOPS$
- double: $TPP = \frac{1997\ GFLOPS}{2} = 998.5\ GFLOPS$

## 2.4 Size Scalability

In this case I kept the number of used OpenMP threads fixed, trying to scale over the matrix size. Each thread is associated with a core (OMP_PLACES = cores), so we allocated 64 cores for EPYC and 12 for THIN nodes. I was tasked with examining matrix sizes ranging from 2,000 to 20,000, and I opted to increment the size in steps of 1,000 for each subsequent matrix size under consideration. Things to keep in mind when analyzing performance of the libraries:

- **OpenBLAS**: an open-source implementation of the BLAS (Basic Linear Algebra Subprograms) library that is optimized for speed on many types of CPUs.
- **MKL (Intel Math Kernel Library)**: a proprietary library developed by Intel. It's highly optimized for Intel processors, and it often outperforms other libraries on systems with these processors, especially for large matrices. However, its performance on non-Intel processors may not be as good.
- **BLIS (BLAS-like Library Instantiation Software Framework)**: a recent framework for instantiating BLAS-like libraries. It's designed to be more flexible and adaptable than traditional BLAS implementations, which can lead to better performance on a wider range of systems and problem sizes. However, as a newer library, it may not be as optimized as OpenBLAS or MKL.

### 2.4.1 EPYC

Let's compare the performance of the libraries with different thread binding policies (close vs spread).

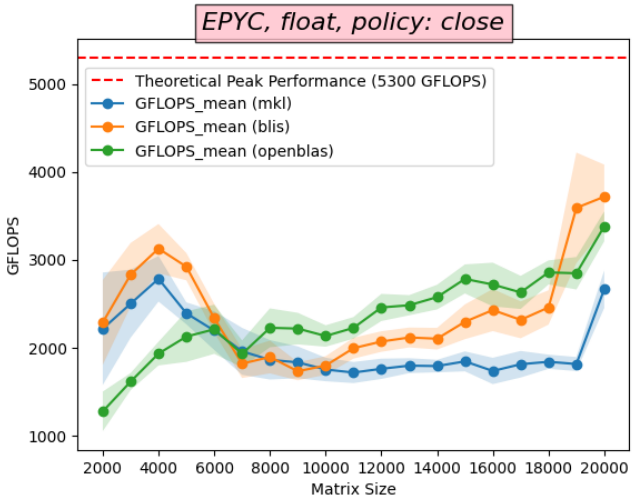



Figure 8: Close threads, single precision

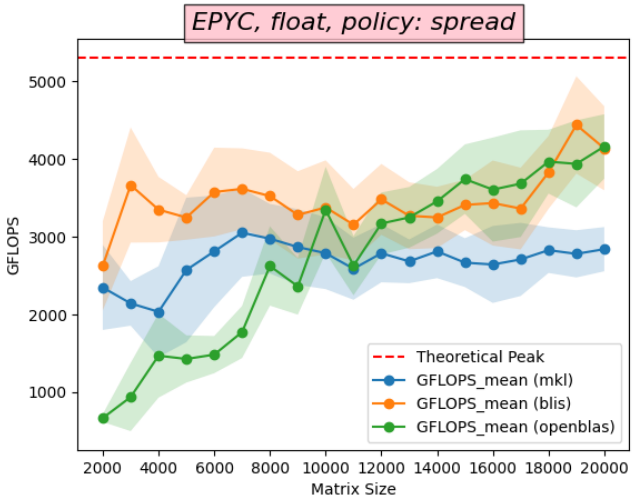



Figure 9: Spread threads, single precision

For **close** core policy OpenBLAS demonstrates an overall ascending trend in GFLOPS. This increasing trajectory is only punctuated by a small, yet discernable, decline around the 7000 GFLOPS mark (probably caused by the cache size limit overcome and the subsequent necessity to fetch data from the RAM).
In contrast, the MKL library's performance records the least GFLOPS among all the libraries under review. Despite having a trend line similar to that of the BLIS library, MKL persistently performs lower GFLOPS.
Following a marked decrease in GFLOPS at the matrix size of 4000, the performance of the BLIS library flattens, remaining relatively constant. A notable surge in performance is only observed when handling larger matrix sizes.
When instead we use **spread** cores thread affinity policy, a significantly larger variability in the measures is present (larger standard deviation). When close policy is adopted, the threads executing the benchmark code are placed as close to each other as possible in the system's topology (always the same predictable positions), leading to greater consistency in the results. This is obviously not the case for spread policy, where at each call of GEMMCPU (where the parallel region is initialized and closed) the threads are bound to different cores.

We get very similar results with double precision.

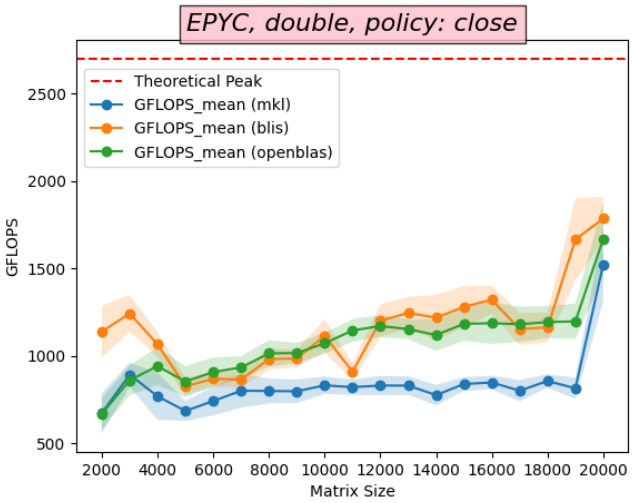



Figure 10: Close threads, double precision

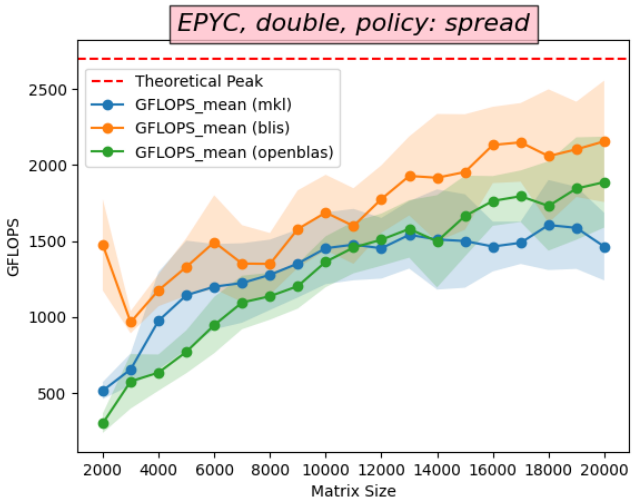



Figure 11: Spread threads, double precision

### 2.4.2 THIN

Thin nodes show a very different behavior. For all plots we observe an increase from sizes 2000 to sizes 4000, after which "saturation" seems to be reached: the performance stays constant. This is probably due to the smaller number of threads (only 12) used, the maximum advantage of parallelism is reached fast and the maximum number of FLOPS that can be performed by each thread is reached sooner. It's clear (especially with double precision) that the best performing library on Thin nodes is MKL, this result was expected because MKL is developed and optimized by Intel for Intel processors.

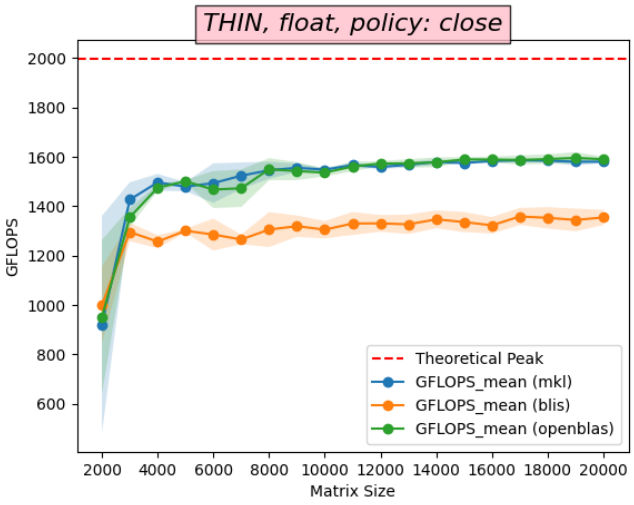



Figure 12: Close threads, single precision

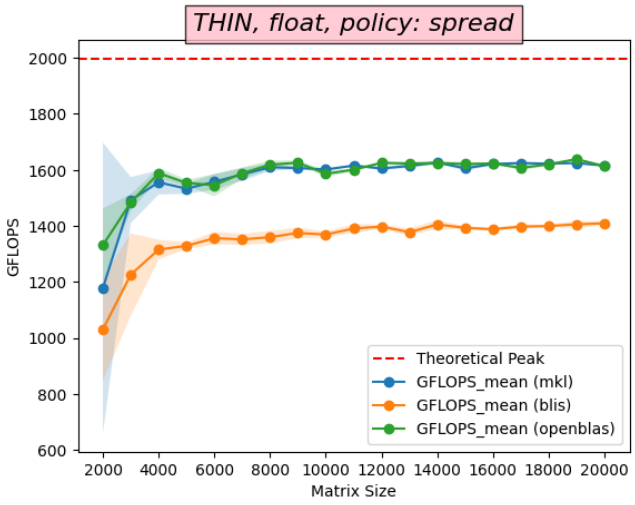



Figure 13: Spread threads, single precision

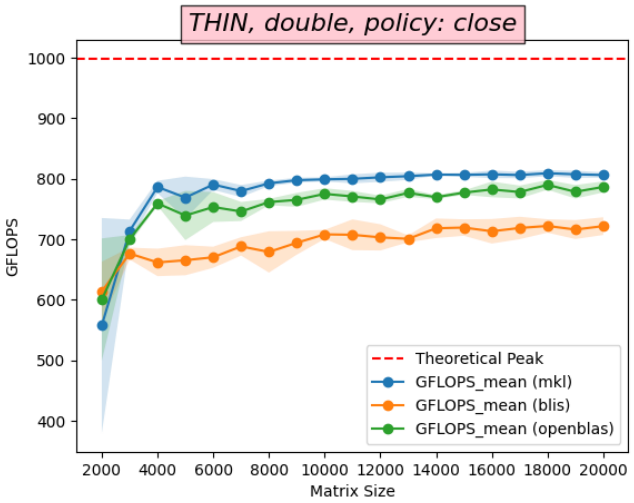



Figure 14: Close threads, double precision

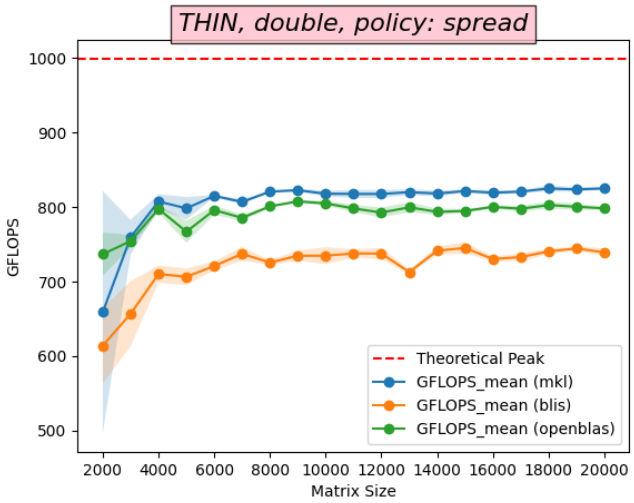



Figure 15: Spread threads, double precision

## 2.5 Core Scalability

Here I fixed the size of the matrices to 10000x10000 and studied the performance in terms of speedup (w.r.t. serial execution time) brought by the increase in the number of used cores.

The "theoretical speedup" is simply the naive ideal assumption that as we double the parallel computing units the execution time halves. I calculated the speedup exactly like I did in Exercise 1.

### 2.5.1 EPYC

For all configurations, the speedup factor rapidly peaks and subsequently remains steady. This performance stagnation can be attributed to false sharing and the increased overhead of maintaining cache coherence. As the number of cores escalates, each core is allocated a smaller segment of the matrices, leading to more frequent instances where a core accesses data already cached by another core, known as false sharing. In other words, the overhead of ensuring cache coherence, which increases with thread count, contributes to the flattening of performance. The BLIS library is less consistent and has more irregular speedup than the other two libraries.

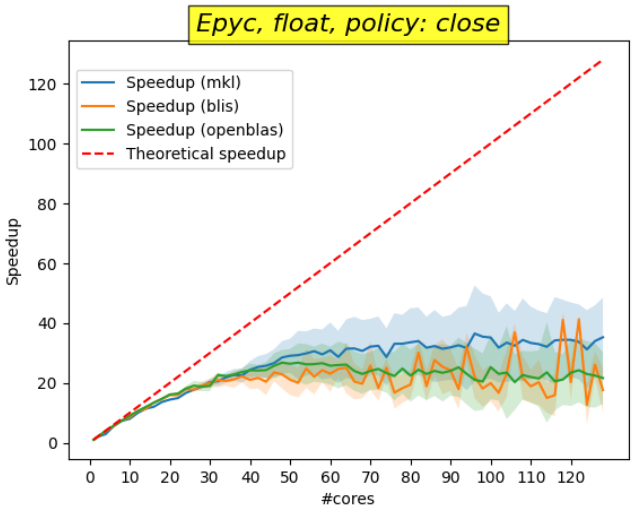



Figure 16: Close threads, single precision

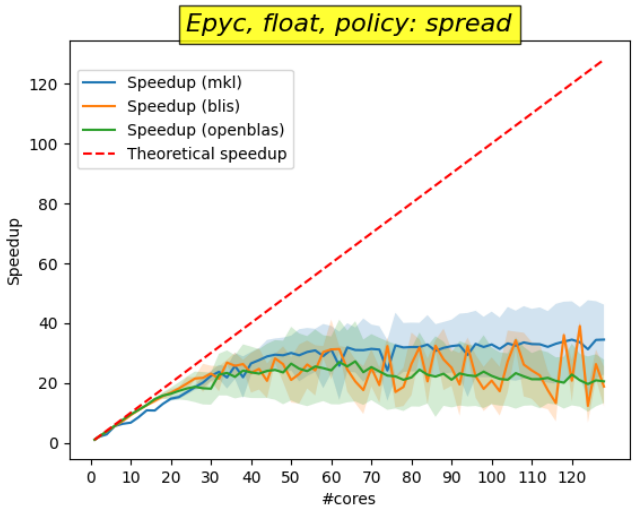



Figure 17: Spread threads, single precision

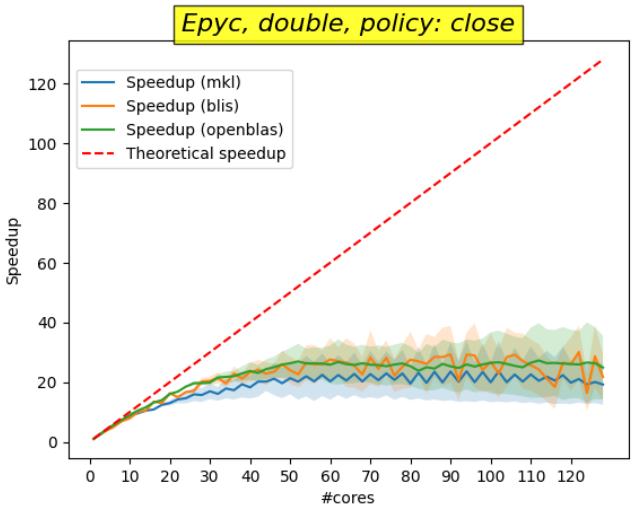



Figure 18: Close threads, double precision

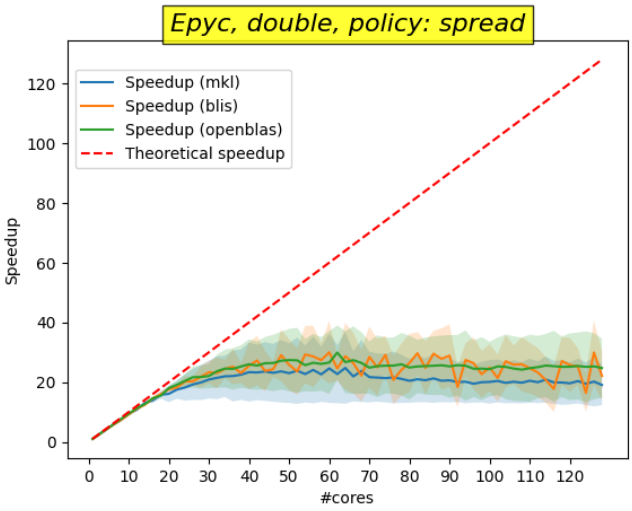



Figure 19: Spread threads, double precision

### 2.5.2 THIN

Once again, as expected, the MKL library displays an almost perfect speedup on Intel nodes, follows OPENBLAS with BLIS as the least efficient library of the three.

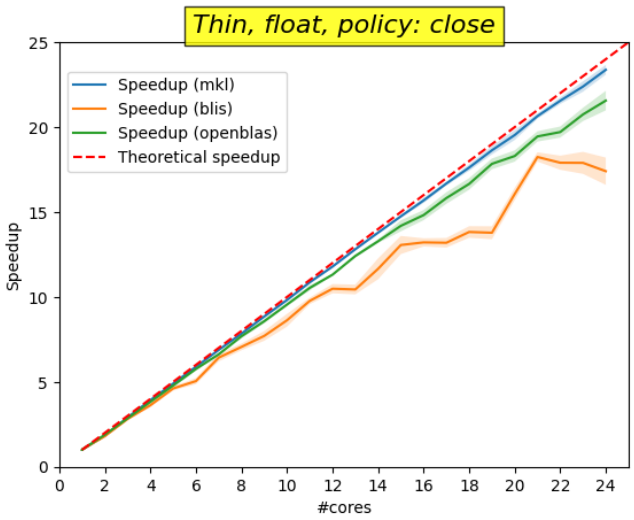



Figure 20: Close threads, single precision

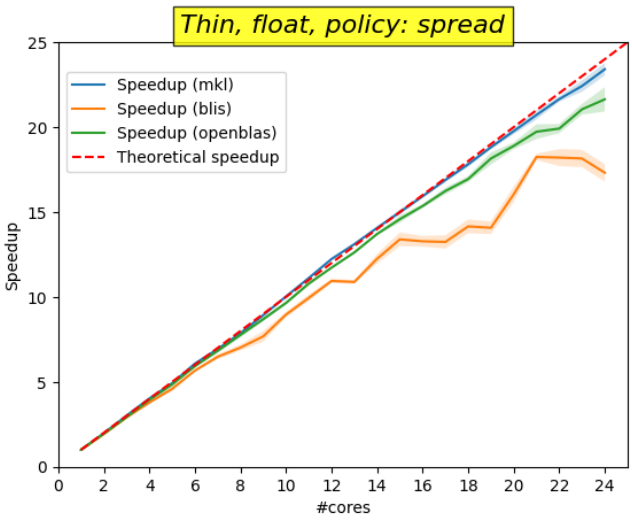



Figure 21: Spread threads, single precision

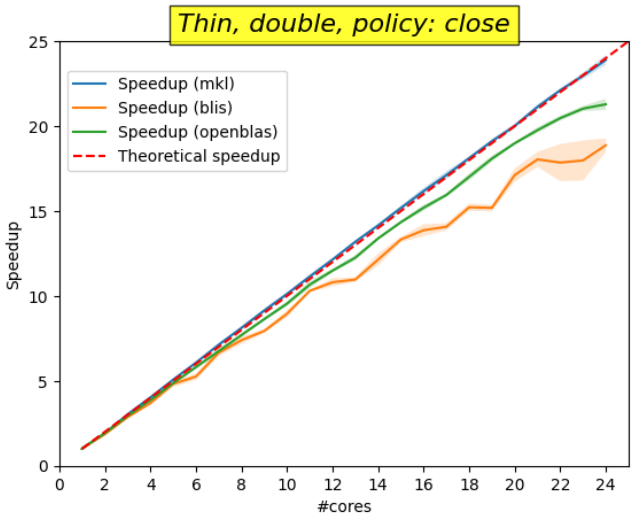



Figure 22: Close threads, double precision

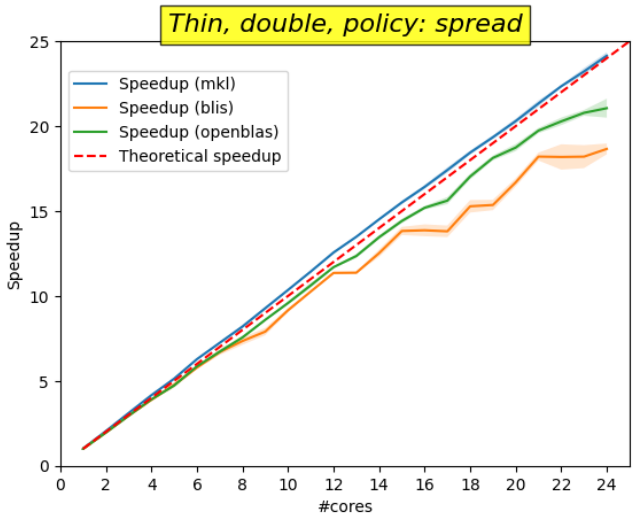



Figure 23: Spread threads, double precision

# References

[1] Luca Tornatore. Exercise 1 parallel programming. Link, 2022.

[2] Wikipedia. Conway's game of life — wikipedia, the free encyclopedia, 2022.