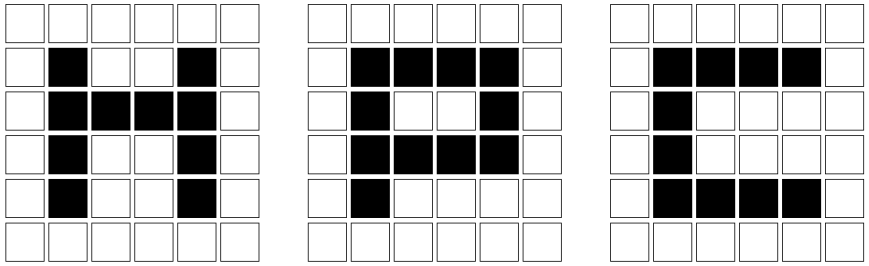
Foundations of High Performance Computing

Report - Data Science and Scientific Computing Course

G. Alessio

April 2024



Contents

1 The Game of Life (GoL)

1.1 Introduction

The Game of Life (GoL) is a simulation where cells on a grid are marked as either "alive" or "dead". The game starts with an initial arrangement of these cells and evolves over time according to specific rules that determine whether cells live or die. The game does not require player interaction after the initial setup (for more information about the game here the reference [1]). Accordingly with the exercise request I implemented an hybrid MPI + OpenMP code (reference[1]), using a domain decomposition to distribute the workload among the MPI tasks in order to perform the simulation. There are different evolutions method. I've mainly focused in:

1. **Ordered Evolution:** Cells are updated one at a time, in a sequential order from left to right, proceeding row by row.
2. **Static Evolution:** The entire grid is evaluated at once, and all cell updates are applied simultaneously after all cells have been assessed.

I'll conduct three types of scalability studies to understand how well our implementation performs under different conditions:

1. **OpenMP Scalability:** Here, we fix the number of MPI tasks (one per CPU socket) and increase the number of threads per task. This tests how well the system scales with more threads on the same number of sockets.
2. **Strong MPI Scalability:** In this study, we keep the size of the grid constant but increase the number of MPI tasks handling the grid. This helps us understand how adding more processors affects performance without changing the workload size.
3. **Weak MPI Scalability:** For this, we increase the size of the grid along with the number of processors, keeping the workload (number of cells) each MPI task handles constant. This tests the system's ability to handle larger problems efficiently as resources are added.

These scalability studies will help determine the effectiveness of the parallelization strategies used in the simulation. For detailed information, refer to the provided reference material [[2]].

1.2 Methodology

The program is written in C language and is capable of performing the two types of evolutions described in the previous section. The playground is read

[1] the github repository where you can find the code and plot https://github.com/gianmarcoalessio/FHPC-assignment

from a **PGM** file, which is converted into an **array of unsigned char**. The choice to use a matrix of unsigned char was made to reduce RAM usage. Within the program, alive cells are represented by the number 0, and dead cells by the number 255, corresponding to the black and white colors in the output grid, respectively. The program consists of mainly three modules:

1. **read_write_pgm.c**: A support module that reads from and writes to a PGM file, facilitating the distribution of data across the nodes.
2. **static_evolution.c**: Implements the static evolution as previously mentioned.
3. **ordered_evolution.c**: Implements the ordered evolution as previously mentioned. 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.

A key design choice for **the static evolution** was the distribution method of the playground among MPI tasks. Distributed parallelism allows storing segments of the playground across multiple nodes, which is particularly advantageous when handling extensive playgrounds.

Conway's Game of Life operates as a cellular automaton where each cell's state is determined by its adjacent neighbors. This spatial dependency makes **domain decomposition an effective strategy** for parallelization of this simulation. In contrast, functional decomposition does not suit this context well. Since the game consistently applies the same rule across all grid cells, dividing the work based on functions does not efficiently segment the problem into independent tasks, as there is essentially one primary function in operation.

The rule of evolution in Conway's Game of Life is influenced by each cell's immediate neighboring cells:

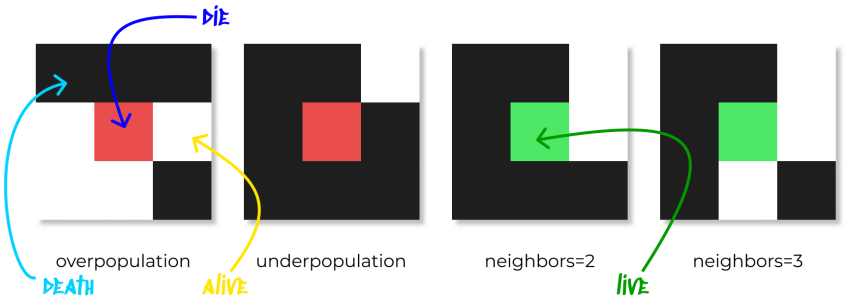


Figure 1: Classical Rules for the GoL

Typically those within the same row (if we think that cells in the same row or columns are closer than the ones in the diagonals). Thus, by allocating rows of cells to different processes, the need for inter-process communication is

minimized. This setup allows each process to access most of the required data directly. Consequently, I chose a **row-wise domain decomposition** approach, where each process is assigned sequential rows of the grid.

Each cell's state in the next timestep is influenced not only by its current state but also by the states of its eight adjacent cells. This group includes cells directly to the left, right, above, below, and on the diagonals, forming a 3x3 neighborhood around the cell.

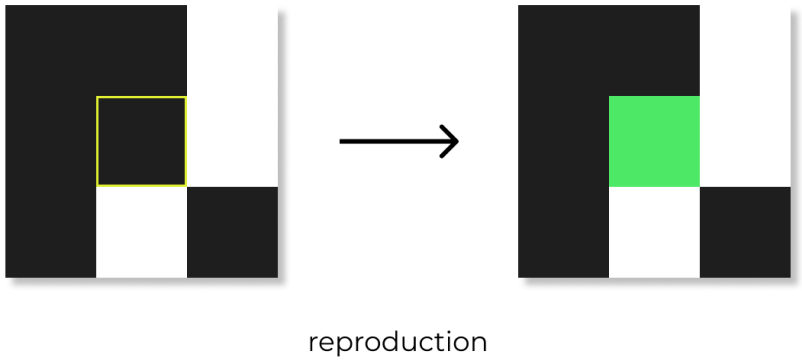


Figure 2: Classical condition of cell reproduction

For instance, a cell on the top row of a segment managed by a process requires the state of the cells in the row above to determine its next state. However, this row is controlled by the adjacent process above, making it inaccessible directly. This situation is similar for cells on the bottom row, which depend on the row below managed by another process.

To address this, processes create "adjacent" rows that mirror the top and bottom rows of the neighboring segments. Each timestep, processes exchange their top and bottom rows with their neighboring processes. These rows are stored as adjacent rows, ensuring that every process has the necessary data to update all its cells, including those on the segment edges [reference at the figure 3].

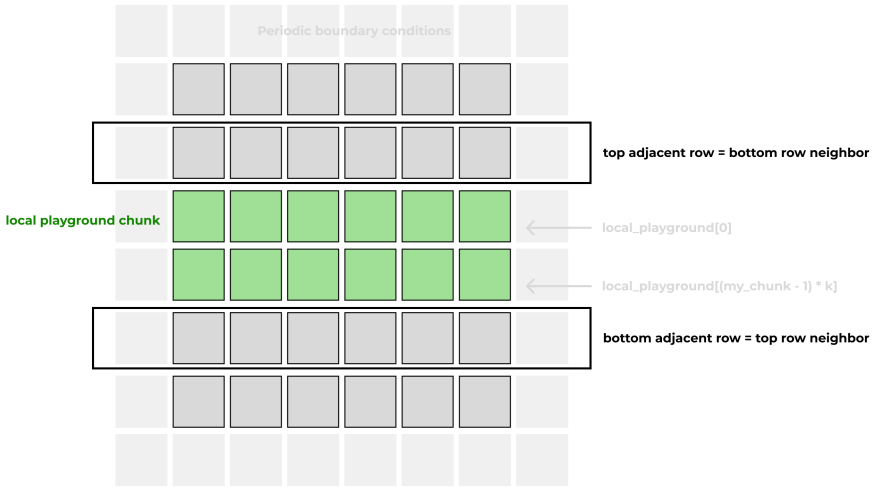


Figure 3: The parallelization approach

Performance was evaluated by running multiple tests on the Thin nodes of the Orfeo cluster, adjusting the number of MPI processes and OpenMP threads appropriately. In this section, I am primarily interested in the speedup described by the formula below:

$$Speedup = \frac{T_1}{T_n}$$

where T_1 is the time taken by a single process, and T_n is the time taken by n processes. Note that the ideal speedup is given by $Speedup = n$, where n is the number of processes.

1.3 Implementation

In this section I discuss some technical aspects of the code, in particular I focus on the parallelization methods.

1.3.1 Initialization

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].

```
unsigned char *parallel_init_playground(int xsize, int ysize)
{
    // Allocate memory for the image
    unsigned char *image = (unsigned char *)malloc(xsize * ysize *
    sizeof(unsigned char));
```

[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

```
    // Use the current time to randomize the seed
    unsigned int time_seed = time(NULL);

// Generate random black-and-white pixel values
#pragma omp parallel
    {
        // Each thread has a unique seed
        unsigned int seed = time_seed ^ omp_get_thread_num();

#pragma omp for schedule(static, 1)
        for (int i = 0; i < xsize * ysize; i++){
            if (rand_r(&seed) % 2){ image[i] = MAXVAL;
            } else { image[i] = 0; }
        }
    }
    // Return the pointer to the allocated memory
    return image;
}
```

Listing 1: Parallel playground Initialization

Each thread uses a unique seed derived from the current time and its thread number (`time_seed ^omp_get_thread_num()`). The directive `#pragma omp for schedule(static, 1)` specifies that each thread should process the grid initialization in chunks of one element at a time, but distributed in a round-robin fashion.

When the `-r` command is executed, the game initiates, leveraging the MPI component of the hybrid parallelism. The playground is segmented into blocks, each with a specified size and uniquely ordered by rank. Each block is allocated to a different MPI process responsible for the computations required for its segment of the grid.

The operation within each MPI process begins by allocating memory for a section of a PGM file and then reading that specific segment using the `parallel_read_pgm_image` function:

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

This function enables concurrent reading of different portions of the playground PGM file by multiple MPI tasks. Each process reads its assigned portion. Following this, all processes synchronize using `MPI_Barrier` before the file is closed. After the file is read, the computational phase commences, which is handled by the OpenMP portion of our hybrid code.

1.3.2 Serial and output validation

To verify the accuracy of the code, a small PGM image (4x4 pixels) was initially created and evolved for one generation using a serial version of the Game of Life. The result was manually validated through visualization. After having assessed that the serial program works properly for a single iteration I have tried to do

two iterations and also in this case I manually verified the output. Below the used example:

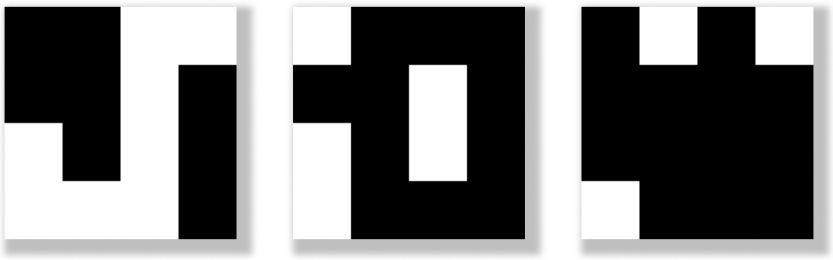


Figure 4: Test Static Evolution

The command line command used for comparison is provided below:

```
compare -metric AE media_serial/static/0_test_dump_static.pgm thin/
    testing02static/ssnapshot_00001.pgm difference.pgm
```

This command compares two PGM images to assess if there are any differences post-evolution, if any, are saved in `difference.pgm`.

The specifics of the computation are dictated by the `-e` command argument, determining the type of evolution—either ordered or static—that will be implemented.

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

1.3.3 Static Evolution Implementation

The `static_evolution` function is organized as follows:

1. **Initialization**: it begins by determining the MPI rank and the total number of MPI processes. It then calculates the size of the grid portion each process is responsible for.

2. **Adjacent Rows**: Memory is allocated to store the top and bottom rows from the neighboring processes, which are also identified in this step.

3. **Main Loop**:

   - **MPI Communication**: Non-blocking sends (`MPI_Isend`) and blocking receives (`MPI_Recv`) are used. This allows processes to continue

operations without waiting for the sends to complete, enhancing parallelism and reducing wait times. Blocking receives ensure data integrity and synchronization before computation begins.

- **OpenMP Parallelization**: A nested loop OpenMP parallel region is initiated, collapsing the two loops into a single parallel loop. This distributes the iterations across multiple threads, optimizing parallelism. Each thread computes the new state of its assigned cells while considering neighboring cell states and boundary conditions imposed by the adjacent rows.

```
#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_adjacent_row : &
    local_playground[(y - 1) * xsize]),
        (y == my_chunk - 1 ? bottom_adjacent_row : &
    local_playground[(y + 1) * xsize]),
        local_playground, updated_playground, xsize,
    my_chunk, x, y);
    }
}

```

- **Grid Swap and Snapshot Writing**: After updating, pointers between `local_playground` and `updated_playground` are swapped to update the local grid state. If the current step number is a multiple of `s`, a snapshot of the grid is captured using the `write_snapshot` function.
- **Memory Deallocation**: Memory allocated for the adjacent rows is freed, ensuring efficient resource management throughout the simulation.

4. **Final Snapshot**: After completing all steps, an additional snapshot is captured if the number of steps differs from `s`.

1.3.4 Ordered Evolution Implementation

Similar to the static evolution method, the ordered evolution method utilizes a domain decomposition approach. As in the previous case, each process reads a portion of the playground (chunk). Before starting the iteration process, the last MPI Task sends its last row to process 0, and each process sends its first row to the preceding process. At the end of the elaboration, each process sends its last row to the preceding process. Notably, in this method, an MPI task begins computation upon receiving the last row from the subsequent process.

In this configuration, I expect that parallelization will not improve computational time; however, it is important to consider that with this type of domain decomposition, memory scales effectively. Since each MPI Task reads its submatrix directly from the file, the overall size of the matrix can be larger than

the RAM of a single node. In contrast, with the serial approach, the entire playground must fit within the RAM of a single node.

1.4 Results and Performance

I have performed several test to evaluate the computation time and the speedup of the different type of evolution, with different size and with different OpenMP and MPI options.

The performance of the program was measured by recording the wall clock time before and after the execution of the `static_evolution` function. The total time elapsed is computed and then averaged over the number of iterations. This average time is recorded in a CSV file by the master process (rank 0) to ensure data consistency and avoid file access conflicts. Wall clock time was chosen for its accuracy in reflecting the real execution time in a parallel computing environment.

```
gettimeofday(&start_time, NULL);
static_evolution(playground, 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);
}
```

Listing 2: Performance Measurement Code

This section presents the measurements made for the static evolution method used, specifically focusing on:

- OpenMP Scalability
- MPI Strong Scalability
- MPI Weak Scalability

All measurements were conducted on Thin nodes of the High-Performance Computing cluster Orfeo.

OpenMP Scalability

The goal was to assess how the program's performance scales with an increasing number of OpenMP threads. These tests were conducted using the OpenMP environment variable `OMP_PROC_BIND=close`. A single MPI Task was assigned

to each socket using the command line option `--map-by socket` when utilizing 1 or 2 sockets on the same machine. For configurations involving more than three sockets (up to four sockets for Thin nodes), the options `--map-by node --bind-to socket` were employed.

For the Thin nodes, scalability tests were conducted using a matrix size of 25000×25000 with 30 iterations across configurations with 1, 2, 3, and 4 sockets. The results are graphically represented below to illustrate the OpenMP scalability on the Thin nodes.

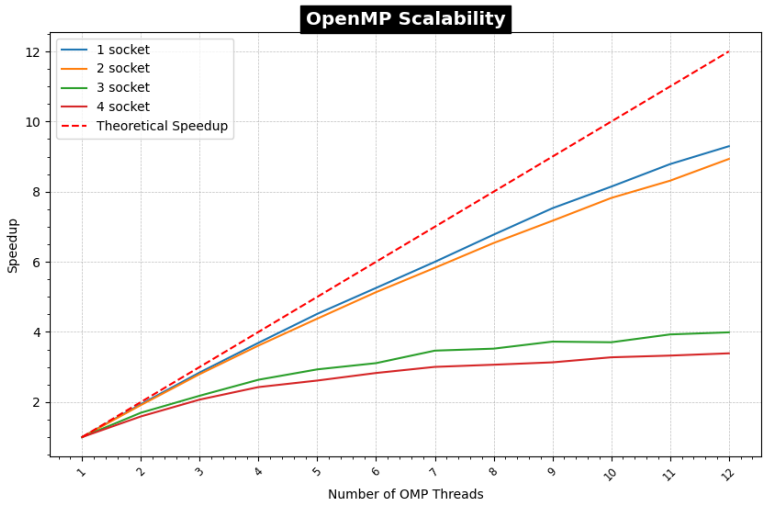


Figure 5: For Static Evolution with different number of sockets on THIN

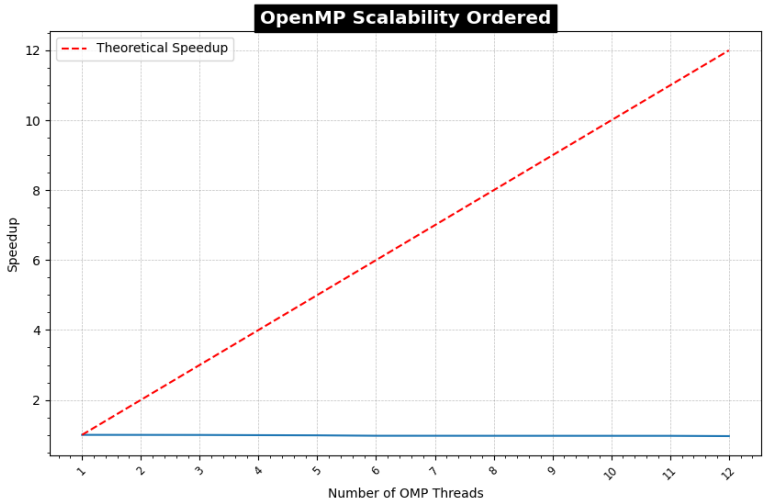


Figure 6: For Ordered Evolution with 1 sockets on THIN

MPI Strong Scalability

To assess the scalability of the MPI implementation, the number of OpenMP threads was set to one, focusing on the MPI scalability by mapping MPI tasks directly onto cores. The experiments were conducted using varying playground sizes to evaluate how scalability is influenced by the size of the computational domain. Here's a snippet from the batch file used for running the experiments:

```
export OMP_NUM_THREADS=1  # Use one thread per core
for size in 15000 25000 35000 45000  # Different playground sizes
do
    mpirun -np 4 -N 2 --map-by socket main.x -i -f "playground_${size}.pgm" -k $size
    for procs in 1 $(seq 2 2 48)  # Varying number of MPI processes
    do
        echo -n "${size}," >> $datafile
        echo -n "${procs},">> $datafile
        mpirun -np $procs -N 2 --map-by core main.x -r -f "playground_${size}.pgm" -e 1 -n 10 -s 0 -k $size
        echo >> $datafile  # Append a newline to the data file
    done
done
```

Listing 3: Batch file snippet

The scalability of the code improves as the size of the playground increases. This enhancement is attributed to the reduction in synchronization overhead relative to the increased computational workload provided by larger image sizes. Initial tests on playgrounds with dimensions up to 45000×45000 demonstrate

significant scalability improvements, particularly with a higher number of MPI processes. Optimal scaling is achieved when the number of processes and the size of the playground are balanced to maximize the use of available computing resources.

This structured approach allows us to observe the impact of varying the number of MPI tasks and the size of the computational domain on the performance of our MPI implementation, providing valuable insights into how to optimize for different hardware setups and problem sizes. The results are graphically represented below:

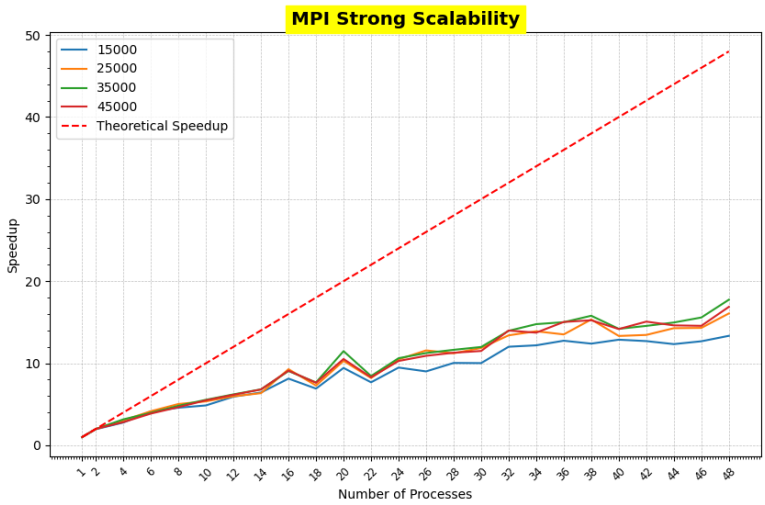


Figure 7: Static evolution represented with different playground size on THIN

MPI Weak Scalability

The primary goal of assessing weak scalability is to examine whether the execution time can remain constant when both the number of processes and the workload per process increase proportionally. Weak scalability is an important metric in parallel computing, as it indicates how well a system can handle increased loads by proportionally increasing resources.

The experiment began with a single process handling a 10000×10000 grid, equivalent to 10^8 bytes of data. As the number of processes increased, the size of the playground for each process also increased according to the formula $size = \sqrt{n} \cdot 10^4$, where n is the number of MPI processes. This scaling ensures that each process manages a similar amount of data despite the increasing total data size and number of processes.

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, the number of OpenMP threads was fixed at 12 per socket. MPI processes were mapped by socket, utilizing 3 nodes for Thin configurations. Graph [8] show the scalability results.

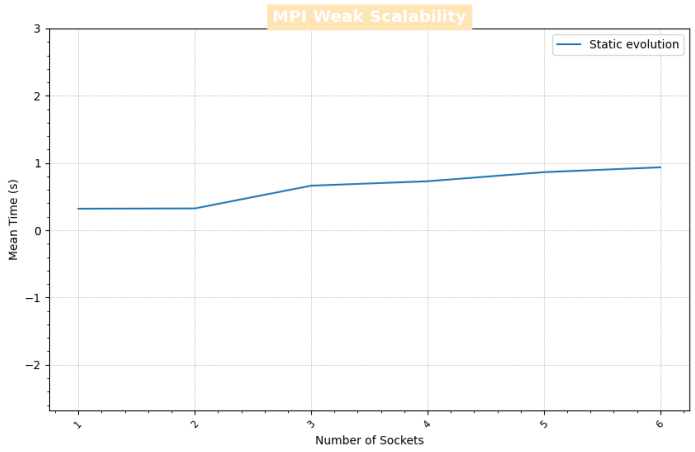


Figure 8: MPI Weak Scalability on Thin for static evolution

As the number of MPI processes increases, so does the inter-process communication overhead, particularly when the communication spans across different nodes. This additional overhead can significantly impact performance, demonstrating the challenges in maintaining execution time as the system scales. The results, illustrated on the graph [8], highlight the efficiency and limits of weak scalability under varying loads and system configurations.

1.5 Conclusions

As it concerns the ordered evolution is stricly serial and the only possible parallelization is the one presented in the section above which give as memory scalability.

Some possible improvements could be:

- Exploration of Non-Standard Domain Decomposition:
  - **Current Limitation:** The game is limited to square matrices.
  - **Improvement:** Adapt the code to handle rectangular matrices, which might allow better utilization of computational resources and match real-world usage scenarios better.
- Investigation into Different MPI Communication Methods:
  - **Current Observation:** Communication overhead significantly impacts performance.
  - **Improvement:** Implement and compare different MPI communication methods, such as 'ready send', to possibly reduce synchronization delays and improve performance.
- Enhanced Testing with Diverse Initial Conditions (Incorporating the "Finger of God" scenario):
  - **New Feature:** Introduce random perturbations in the initial conditions as described by the "Finger of God", where cells may randomly die or generate due to cosmic interactions.
  - **Expected Outcome:** This could lead to discovering new dynamic behaviors and patterns, potentially affecting scalability and efficiency insights.
- Hybrid MPI+OpenMP Optimization:
  - **Current Observation:** Hybrid models show limitations especially in ordered evolution scenarios.
  - **Improvement:** Conduct more extensive hybrid MPI+OpenMP tests, possibly adjusting the ratio of MPI tasks to OpenMP threads to find optimal configurations for different types of load and node characteristics.
- Further Increase in the Number of Iterations and Task Sizes:
  - **Current Observation:** Increasing iterations makes parallelism more effective.
  - **Improvement:** Test higher iterations and larger task sizes to fully evaluate the potential of parallelism, especially to see if the gains can outweigh the communication overhead.
- Performance Testing Across Different Node Types:
  - **Current Observation:** Performed only in THIN nodes.
  - **Improvement:** Conduct targeted performance comparisons between different node types, focusing on specific configurations that leverage their respective strengths.

- Detailed Analysis of Thread and Core Utilization:
  - **Current Observation:** Optimal thread usage varies significantly between node types.
  - **Improvement:** Detailed experiments to determine the most efficient use of threads per MPI task on different hardware, adjusting for both small and large problem sizes.
- Incorporation of Variable Computational Workloads:
  - **Optional Rule Incorporation:** Introduce variable computational challenges within the game to test scalability under dynamic load conditions.
  - **Expected Outcome:** This would help in understanding how different MPI and OpenMP configurations handle sudden changes in computational demand.

2 Comparing MKL, OpenBLAS and BLIS

The goal of this exercise is to compare performance of three math libraries available on HPC: MKL, OpenBLAS and BLIS. In order to do it has been selected the Matrix Multiplications level 3 BLAS (Basic Linear Algebra Subprograms) that the general definition state:

$$C \leftarrow \alpha op(A) op(B) + \beta C \tag{1}$$

In our case we consider simply for $\alpha = 1$, $op(X) = X$ and $\beta = 0$. The 3 BLAS function used is called *gemm*. Defined as:

```
GEMMCPU(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k, alpha,
    A, m, B, k, beta, C, m);
```

Listing 4: 3 BLAS function

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).

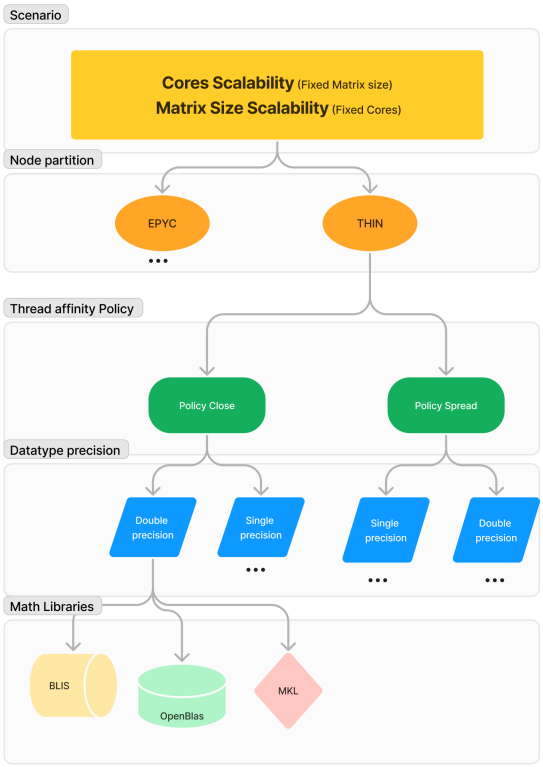


Figure 9: All the scenarios considered to compare performance

The two scenarios to be analyzed were:

- **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), threads allocation policy (close or spread), precision of the *gemm* function (double and single).

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.

2.1 Details Set-Up

To enhance the analysis process, the gemm.c file has been adjusted to calculate the performance metrics such as GFLOPS (Floating Point Operations Per Second) and the mean and standard deviation of the execution time across **ten consecutive runs**:

```
#ifdef MULTIPLE_ITERATIONS

    ...

    int num_trials = 10;
    for (int trial=0; trial < num_trials; trial++){
    clock_gettime(CLOCK_MONOTONIC, &begin);
        GEMMCPU(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
     alpha, A, m, B, k, beta, C, m);
    clock_gettime(CLOCK_MONOTONIC, &end);
    elapsed = (double)diff(begin,end).tv_sec +
                (double)diff(begin,end).tv_nsec / 1000000000.0;
    double gflops = 2.0 * m *n*k;
    gflops = gflops/elapsed*1.0e-9;
  elapsed_sum += elapsed;
        elapsed_sq_sum += elapsed * elapsed;
        gflops_sum += gflops;
        gflops_sq_sum += gflops * gflops;
    }
    double elapsed_mean = elapsed_sum / num_trials;
    double elapsed_sd = sqrt((elapsed_sq_sum / num_trials) - (
    elapsed_mean * elapsed_mean));
    double gflops_mean = gflops_sum / num_trials;
    double gflops_sd = sqrt((gflops_sq_sum / num_trials) - (
    gflops_mean * gflops_mean));
```

Maintaining consistent parameters such as the number of threads or matrix size throughout each iteration. The resulting metrics are then stored in **.csv** files for further examination.

Theoretical Peak Performance - TTP

The theoretical peak performance represents the maximum computational power a system can achieve under ideal conditions. It is calculated based on factors such as the number of CPU cores, clock speed, and instruction set capabilities. I used it in order to set a reference for the experiment. The formula I used is:

$$\text{TTP (GFLOPS)} = \#\text{cores} \times \text{clock(GHz)} \times \frac{\text{FLOPs}}{\text{Cycle}} \tag{2}$$

Node Type	Float(GFLOPS)	Double (GFLOPS)
EPYC	$64 \times 2.6 \times 32 = 5324.8$	$64 \times 2.6 \times 16 = 2662.4$
THIN	1997	998.5

Table 2: Theoretical Peak Performance (TPP) for EPYC and THIN Nodes

Now that all necessary configurations are in place, I can shift my focus to data gathering

Implementation

Table 2 provides a glimpse into the output obtained from this process.

Matrix Size	Time Mean (s)	Time SD	GFLOPS Mean	GFLOPS SD
2000	0.023569	0.004371	695.148240	88.133904
3000	0.060036	0.007911	913.557360	108.085388
4000	0.173409	0.013910	742.564468	55.458824
5000	0.321705	0.090602	851.388269	265.637915
6000	0.498166	0.165280	956.715361	275.247653
7000	0.742405	0.238388	998.082636	235.387349
8000	1.051124	0.306676	1037.142612	220.586396
...	...	...	...	...

Table 3: Performance metrics obtained from running the Intel Math Kernel Library (MKL) on an EPYC platform. The computations were conducted using double precision and employing the "close" thread affinity policy.

To facilitate the computation of these metrics, I devised a batch job script named "job.sh". Within this script, I orchestrated the allocation of essential resources, loaded requisite modules, configured the CSV files, and orchestrated the execution of programs while systematically varying the parameters. An example follows (studying matrix size scalability on THIN nodes):

```
#!/bin/bash
#SBATCH --no-requeue
#SBATCH --job-name="ex2"
#SBATCH -n 12
```

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

The lines prefaced with "`#SBATCH`" are directives specifically tailored for the SLURM job scheduler, this particular script requests exclusive access to 12 CPU cores on a single node in the "THIN" partition for a maximum runtime of 2 hours, with a specific job name and environment configuration.

```
module load architecture/Intel
module load mkl
module load openBLAS/0.3.23-omp
export LD_LIBRARY_PATH=/u/dssc/galess00/final_assignment_FHPC/
    exercise2/myblis_thin/lib:$LD_LIBRARY_PATH
```

Then this script loads specific software modules tailored for the Intel architecture, the Intel Math Kernel Library (MKL), and the OpenBLAS library with OpenMP support. The `LD_LIBRARY_PATH` environment variable is updated to include a custom library path, ensuring that the necessary BLIS library is accessible to the system at runtime.

```
export OMP_PLACES=cores
export OMP_PROC_BIND=close // or spread
export OMP_NUM_THREADS=12 // 64 for EPYC
export BLIS_NUM_THREADS=12
```

These commands configure parallel execution settings: binding OpenMP threads close to where they are created and allocating 12 threads each for OpenMP and BLIS operations.

```
for lib in openblas mkl blis; do
  for prec in float double; do
    file="${lib}_${prec}.csv"
    if [ ! -f $file ]; then
      echo "matrix_size,time_mean(s),time_sd,GFLOPS_mean,GFLOPS_sd"
     > $file
    fi
  done
done
```

The initialization loop is implemented to generate the .csv files. The conditional statement prevents data loss by ensuring that files are not overwritten when executing multiple jobs, resulting in a total of 48 distinct .csv files.

```
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
```

Finally the last part of the script perform a nested loop to iterate over different matrix sizes, libraries, and precision types.

Things to keep in mind

- MKL is developed and maintained by Intel Corporation, primarily targeting Intel architectures. However, its performance on non-Intel processors may not be as good.
- BLIS development may not be as rapid or well-supported as other libraries, potentially leading to longer update cycles or slower adoption of new features.
- OpenBLAS is optimized for various CPU architectures, its performance may vary depending on the specific hardware configuration.

2.2 Size scaling - Fixed Cores

For the size scaling I have increased the size of the matrix from 2000×2000 to 20000×20000 by steps of 1000 (at each step I have increased both the number of rows and the number of columns in order to have always square matrices). For each size and math library I have taken 10 different measurements.

Below are the tested policies (default options: `OMP_NUM_THREADS=12` for THIN (64 for EPYC) and `OMP_PLACES=cores`):

- `OMP_PROC_BIND=close`,
- `OMP_PROC_BIND=spread`

2.2.1 THIN performance

Let's consider firstly the case of THIN nodes, showing the difference between float and double precision data types when opting for close or spread thread allocation policy.

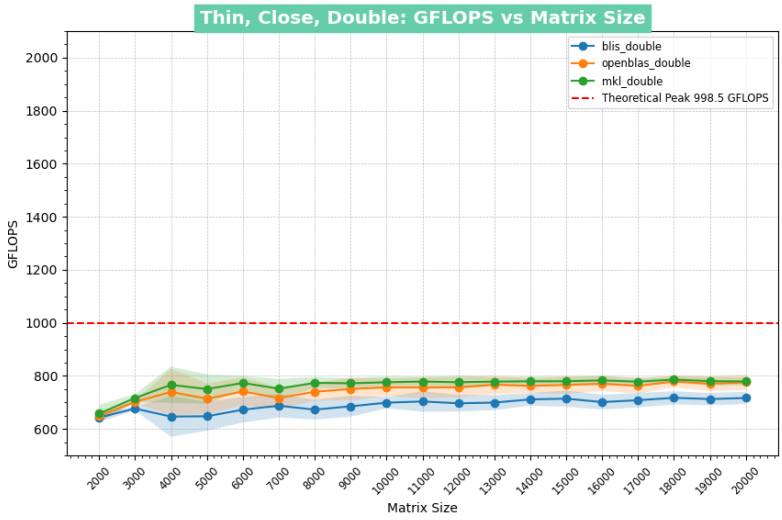


Figure 10: Graph for data type comparison with the graph below

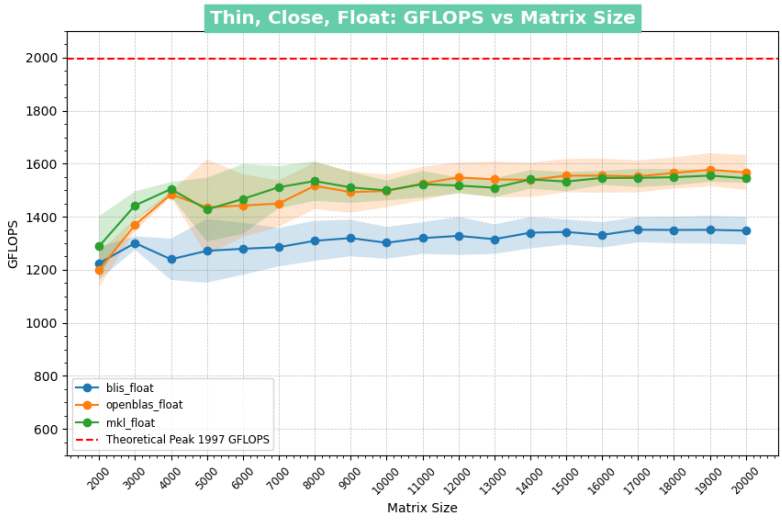


Figure 11: Graph for data type comparison with the graph above

I can see in from the graphs [Figure 10] and [Figure 11] that for both data types, the performance increases with matrix size, peaking around a certain point before leveling off or slightly decreasing. This suggests that larger matrices **initially benefit** from more parallelism (utilizing more threads efficiently)

until memory bandwidth or other resource limitations start to restrict performance gains.

Both graphs include a theoretical peak line (998.5 GFLOPS for double and 1997 GFLOPS for float). None of the libraries achieve these peaks, which is common in practical scenarios due to overheads and inefficiencies not captured in theoretical models.

Performance across libraries:

- **Intel MKL**: Generally shows the highest or near-highest performance across all matrix sizes and data types. This is consistent with MKL's reputation for being highly optimized for Intel architectures.
- **OpenBLAS**: Performs competitively but slightly below MKL in most cases. It shows particular strength in smaller matrix sizes.
- **BLIS**: Tends to lag behind the other two libraries, especially in the double precision graph. This might be indicative of lesser optimizations for the operations or hardware used in this test.

Impact of Data Type (Double vs Float)

The float operations (single precision) are capable of achieving higher GFLOPS compared to double precision, particularly noticeable in larger matrix sizes. This is expected as single precision computations typically require **less memory bandwidth** and computational resources, thus allowing higher throughput. In the float graph, there is a peak performance observed around mid-range matrix sizes (around 5000 to 7000), which then stabilizes. This might be due to better utilization of cache memory with these sizes.

The comparison of double precision and single precision performance under the same operational conditions (matrix size, number of threads, memory policy) provides valuable insights into the efficiency of these libraries in different computational contexts. It also underscores the importance of choosing the right library and precision for specific computational needs, balancing between computational speed and precision requirements.

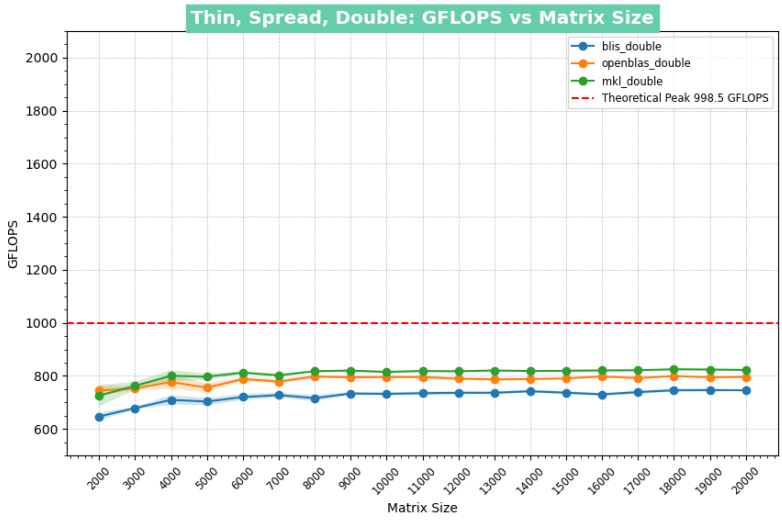


Figure 12: Graph for data type comparison with the graph below

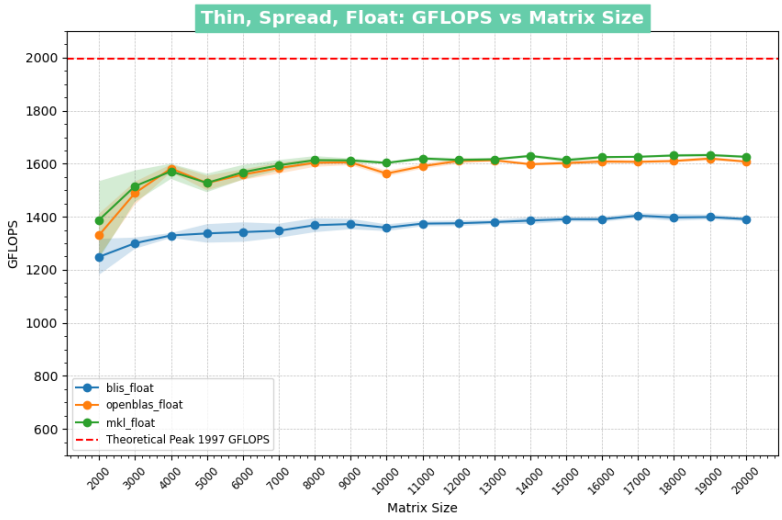


Figure 13: Graph for data type comparison with the graph above

Performance Drop in "Spread" policy ([Figure: 12] and [Figure: 13]), for both data types compared to the "Close" setup. This could suggest that the "Spread" policy, which likely distributes data more widely across memory, may not be as effective in utilizing CPU cache or local memory optimally, leading to

increased data retrieval times.

Concluding Comparisons

- **Effect of Data Type**: Both memory policies affirm that single precision computations are more efficient, achieving higher GFLOPS, particularly under conditions that can utilize the hardware efficiently.
- **Memory Management Policy**: The "Close" policy appears to be better for achieving higher computational performance in matrix operations, suggesting it is better suited for environments where high throughput is critical and data locality can be maximally exploited.
- **Library Performance Consistency**: MKL consistently shows high performance, followed closely by OpenBLAS, with BLIS slightly behind in both policies and data types. This underscores the importance of selecting the right library based on the specific performance characteristics and hardware compatibility.

2.2.2 EPYC performance

Now let's consider the case of THIN nodes in the Matrix Size Scalability Scenario, showing the difference between float and double precision data types as we did for thin nodes when opting for close or spread thread allocation policy.

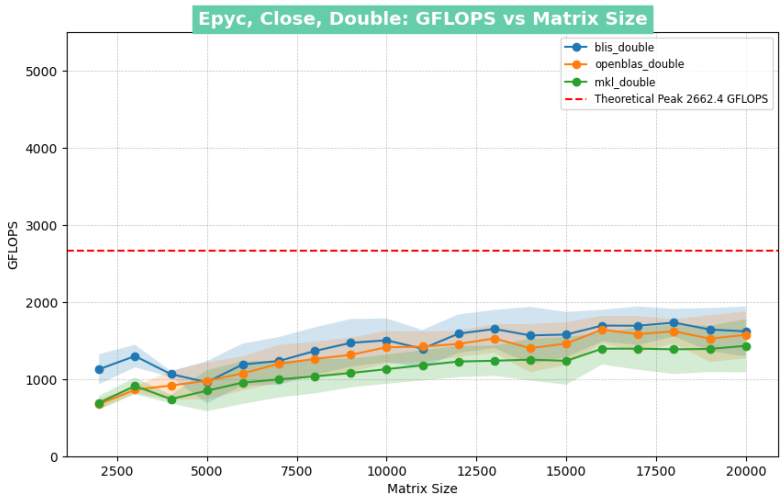


Figure 14: Graph for data type comparison with the graph below

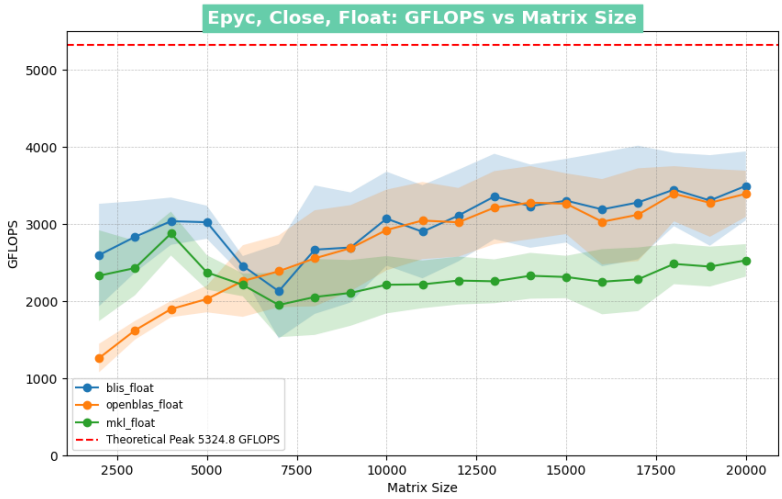


Figure 15: Graph for data type comparison with the graph above

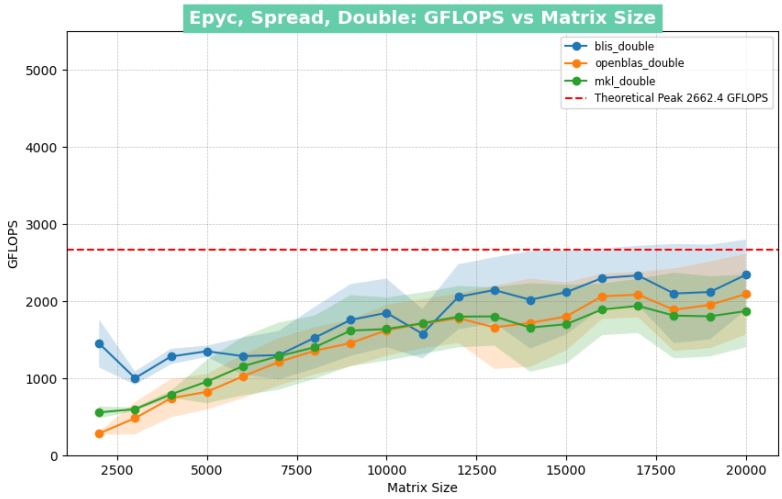


Figure 16: Graph for data type comparison with the graph below

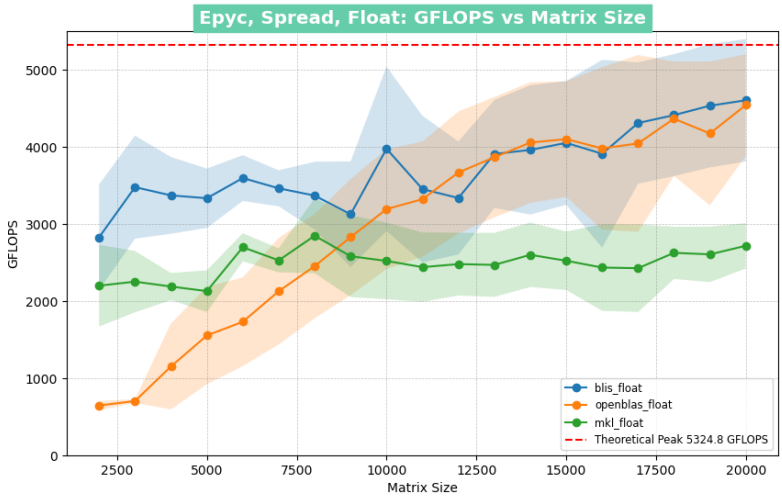


Figure 17: Graph for data type comparison with the graph above

There is a notable variability and a general upward trend in performance for float datatype as matrix size increases. This graph [Figure 11] shows particularly high GFLOPS readings for **BLIS** and **OpenBLAS** (in contrast compare it to the THIN node where MKL library for close policy in both precision had the lead), surpassing the other EPYC graphs. The variability is less pronounced for double precision, but there is a gradual increase in performance as matrix sizes increase. The performance peaks are less dramatic than in the float graph, and generally stay closer to each other across different libraries.

Both graphs under the spread policy show greater variance compared to the close policy. This could suggest that the spread memory management allows for a less consistent but potentially higher peak performance under certain conditions (especially in larger matrix sizes).

EPYC processors may benefit from the spread policy by reducing bottlenecks associated with local memory accesses, particularly in scenarios where multiple threads are reading and writing to memory simultaneously.

The higher performance and variance seen in the spread policy graphs, especially in float precision, suggest that while this policy may introduce less predictability in performance due to varied data placement and access patterns, it also provides **opportunities for higher peak performance**. This might be particularly advantageous in workloads where data can be efficiently pre-fetched and managed across multiple caches and memory channels. Selecting the right memory policy thus depends significantly on the specific characteristics of the

workload and the computational resources of the hardware platform.

2.3 Cores scaling - Fixed Matrix Size

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

In order to measure the performance I've used the *speedup* introduced in the **exercise 1**, that I'm going to recall it now, described by the formula below:

$$Speedup = \frac{T_1}{T_n}$$

Where T_1 is the time taken by a single process, and T_n is the time taken by n processes. Note that the ideal speedup is given by $Speedup = n$, where n is the number of processes. Below is the code used to compute the speedup and the standard deviation of the speedup, which is derived from the error propagation of the standard deviation of the execution time.

```
...

y = [y[0]/y_i for y_i in y] # speedup
sd = [float(line.split(',')[2]) for line in data[1:]]
sd = [np.abs(y[0]/y_i)*np.sqrt((sd[0]/y[0])**2+(sd_i/y_i)**2) for
    sd_i,y_i in zip(sd,y)]

...
```

Like in previous section I start with the THIN partition.

2.3.1 THIN performance

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.

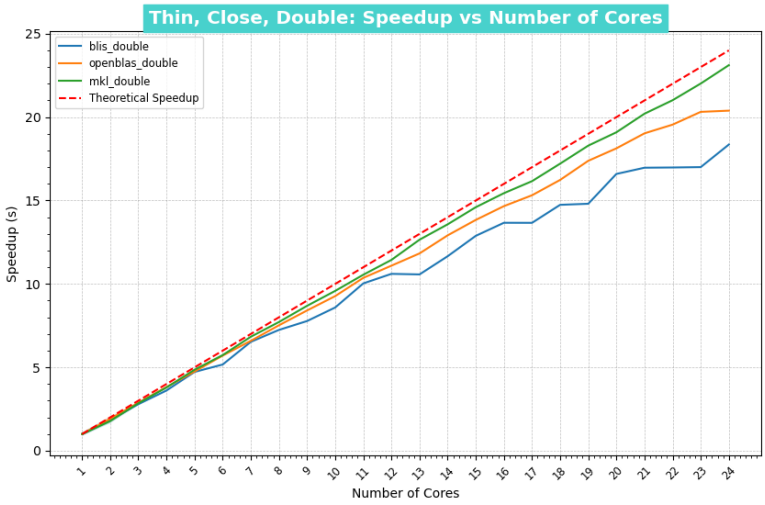


Figure 18: Graph for data type comparison with the graph below

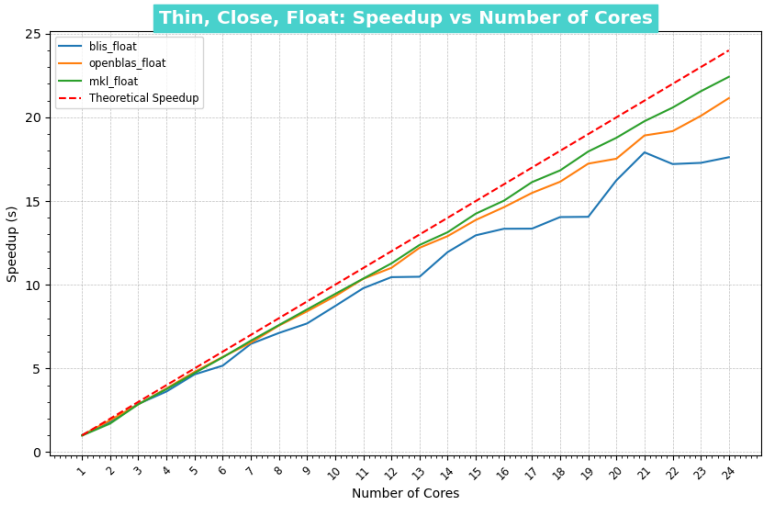


Figure 19: Graph for data type comparison with the graph above

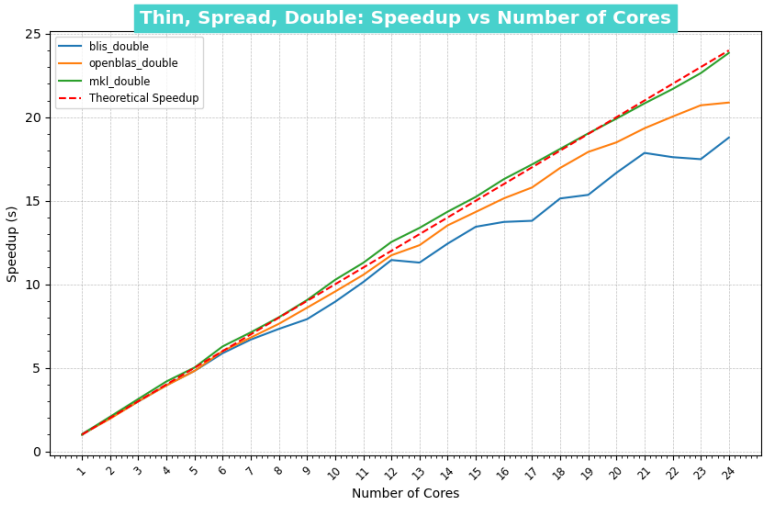


Figure 20: Graph for data type comparison with the graph below

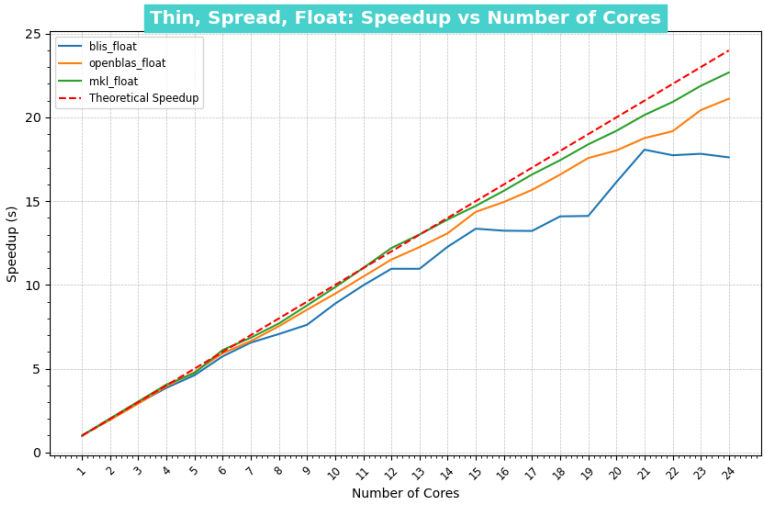


Figure 21: Graph for data type comparison with the graph above

2.3.2 EPYC performance

Across all configurations, the speedup factor quickly reaches its peak and then stabilizes. This performance plateau can primarily be attributed to false shar-

ing and the increased overhead associated with maintaining cache coherence. As the number of cores increases, each core is assigned a smaller segment of the matrices. This leads to more frequent instances of false sharing, where a core accesses data that is already cached by another core. Furthermore, the overhead associated with ensuring cache coherence escalates with the number of threads, contributing significantly to the leveling off of performance improvements. Compared to other libraries, the BLIS library exhibits less consistency and more irregular speedup patterns.

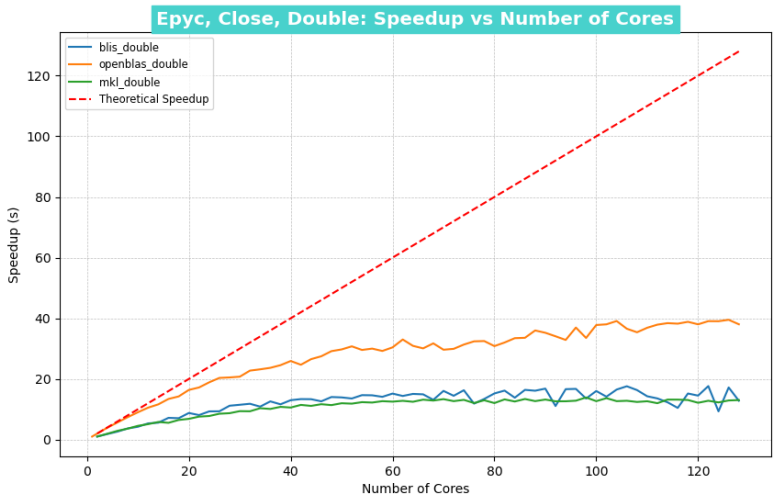


Figure 22: Graph for data type comparison with the graph below

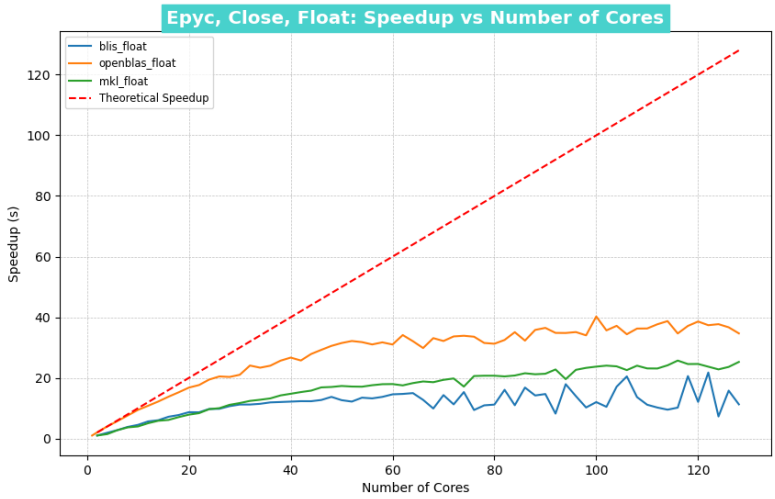


Figure 23: Graph for data type comparison with the graph above

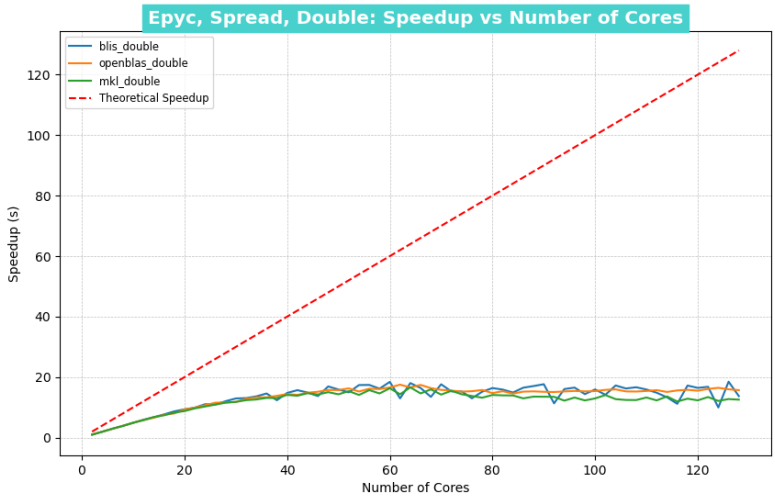


Figure 24: Graph for data type comparison with the graph below

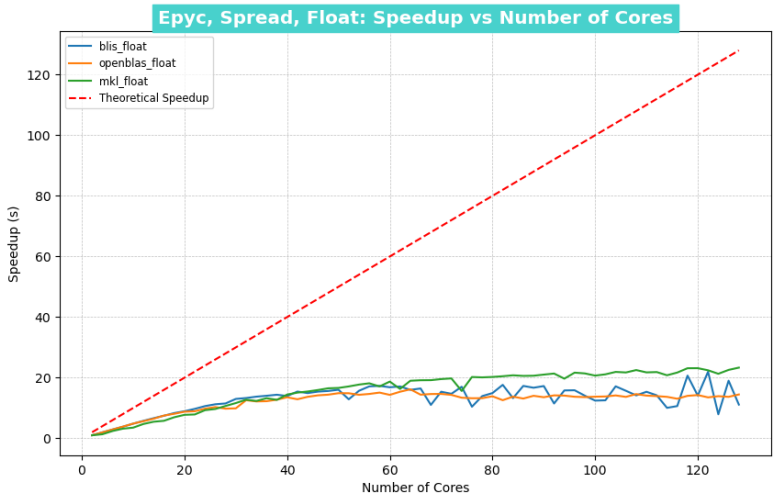


Figure 25: Graph for data type comparison with the graph above

References

[1] John Horton Conway. *the Game of Life*. 1970. URL: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[2] Foundations of HPC. *Final assigments FHPC course 2022/2023*. 2022. URL: https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/blob/main/Assignment/exercise1/Assignment_exercise1.pdf.