Foundations of High Performance Computing Exam Assignments Report

Leonardo Musini

February 2025

Contents

1 Parallel Programming on Game of Life

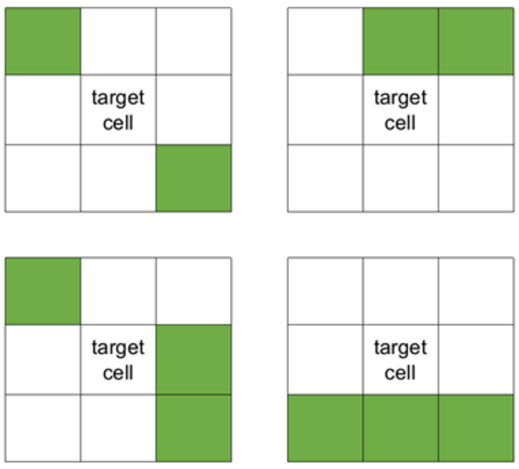
1.1 Introduction

The goal of this exercise is to implement a parallelized version of the **Game of Life (GoL)** using a hybrid approach that combines **MPI** and **OpenMP** to exploit both distributed and shared memory parallelism.

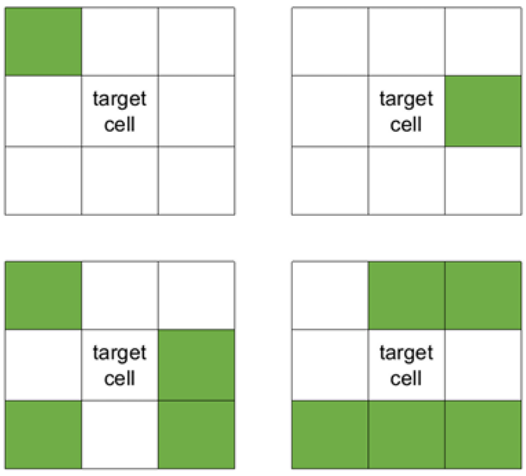
The **Game of Life**[1][2] is a cellular automaton simulation that operates on an infinite two-dimensional grid. The concept of an "infinite grid" is approximated by a $k \times k$ matrix, referred to as the playground, with periodic boundary conditions applied at the edges. The smallest unit in the playground is a single cell, which can exist in one of two states: "alive" or "dead".

The evolution of the system is governed by rules, based on the states of the eight immediate neighbors of each cell:

- a cell becomes, or remains, alive if 2 to 3 cells in its neighborhood are alive (Fig.1(a));
- a cell dies, or do not generate new life, if either less than 2 cells or more than 3 cells in its neighborhood are alive (under-population or over-population) (Fig.1(b)).



(a) Alive cell cases



(b) Dead cell cases

Figure 1: Updating rules for cell reproduction

To update a cell's status according to these fundamental rules, two distinct evolution strategies are employed in this project:

- **Ordered Evolution:** cells are upgraded sequentially in a row-major order, this inserts a "spurious" signal in the system evolution.
- **Static Evolution:** the status evaluation and status update of each cell is disentangled. All cells in the grid are evaluated simultaneously, and their updates are applied collectively at the end of each timestep.

Both type of evolutions are implemented in a **serial** and **parallel** (hybrid MPI + OpenMP) way, depending on the resources available. The simulation employs domain decomposition to distribute computational workloads among multiple MPI tasks and utilizes OpenMP threads within each task to enhance performance.

To evaluate the efficiency and scalability of the parallel implementation, three types of scalability studies are performed:

- **OpenMP Scalability:** the number of MPI tasks is fixed (one per socket), while the number of threads per task is increased up to the number of cores present in the socket.
- **Strong MPI Scalability:** the size of the matrix is kept fixed, while the number of MPI tasks is increased on multiple nodes.
- **Weak MPI Scalability:** the size of the matrix increase proportionally with the number of MPI tasks, one per socket (saturated with OpenMP threads).

1.2 Methodology

The code is written in C language. The program begins by initializing a $k \times k$ grid of **unsigned char**, chosen to reduce RAM usage since it occupies only 1 byte of memory per cell. The grid is then saved as a binary **PGM file**. The playground is generated with a random distribution of cells, where alive cells are represented by white dots and dead cells by black dots as in the images (Fig.2).

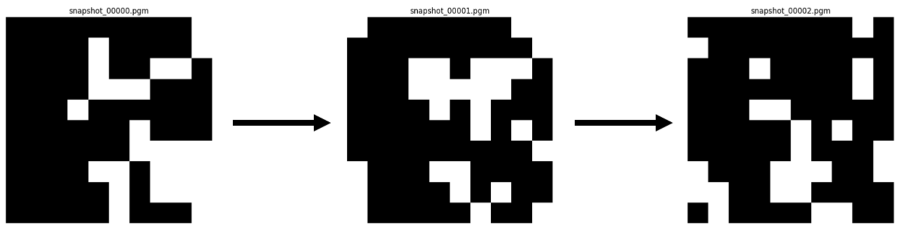


Figure 2: Example of static evolution

The PGM playground is then read to start the simulation phase, during which each cell state is updated according to the two rules provided in the previous section, that are based on the statuses of their eight neighbors.

The system evolves using two distinct evolution strategies: **ordered** and **static**. In the **ordered evolution**, cells statuses are updated in a strict row-major order, progressing from left to right and row by row. In the **static evolution** the evaluation and update of cell statuses is divided in two steps: the system is "frozen" to evaluate the state of each cell and of its neighbors, then is updated all at once.

Both methods are implemented in two ways: **serial** and **parallel**. The **serial** implementation is not entirely sequential, as it utilizes OpenMP parallelism within a single process to enhance the performance. This option is used when only a single MPI process is involved, avoiding unnecessary overhead from MPI function calls and resource allocation.

The **parallel** versions instead are implemented in a fully hybrid MPI+OpenMP approach in order to exploit both distributed memory parallelism (across nodes using MPI) and shared memory parallelism (within each node using OpenMP). This is done using **domain decomposition**, where the grid is divided into submatrices and distributed among MPI tasks. Each MPI task is responsible for updating a specific portion of the grid and then they need to communicate just to exchange information of boundary rows of each grid block. Since the update of each cell depends on its neighbors, the cells at the border of the grid of one process need to receive information of the statuses of cells from the border of another process.

Domain decomposition was chosen over functional decomposition because it is better suited for the Game of Life, where the computational workload is tightly coupled to the grid's spatial layout. The update rules of each cell status are not very complex or difficult to manage, while the main problem is handling computations and updates on very large matrices. Dividing function execution between MPI processes does not fit well for a problem of this kind. Instead by dividing the grid spatially, each MPI task processes a contiguous block of rows, where communications are needed only for information exchange of halo regions (boundary rows) between neighboring tasks. In this way each process handles a smaller grid, exploiting effectively the advantages of parallelism. This approach also reduces synchronization overhead, as each task operates independently on its subdomain, with minimal dependencies beyond the halo exchanges.

For what concerns the **ordered evolution**, since it is intrinsically serial, it does not take full advantage of parallelism. This is because the ordered evolution updates the cell statuses in a strict row-major order, introducing dependencies between iterations. Each cell update depends on the completion of previous updates within the same row and column, making it difficult to divide the workload effectively among multiple threads or processes. MPI tasks and thread division must respect the grid order of update. This means that even if we divide the workload in more processes, each should wait the end of the previous one.

In contrast, the **static evolution** method is inherently more suitable for parallelization. By separating the evaluation and update phases, we can allow different blocks of rows to be processed simultaneously without interdependencies. In the former step, the system can be considered "frozen", so we can evaluate the statuses of each cell in more processes without any interference, since there are no update order to be respected. The division enables an even distribution of work among MPI tasks and OpenMP threads that can be executed in paralell. This makes static evolution much more efficient and scalable for exploiting parallelism in both shared and distributed memory systems.

In the final part of the algorithm, the program can save snapshots of the system at specified intervals for both evolution strategies. If multiple MPI processes are involved, the global grid must be reconstructed before saving. To achieve this, all subgrids managed by the MPI processes are gathered and unified into a single global grid, which is then saved as a complete snapshot file. This approach ensures that the snapshots accurately represent the state of the entire simulation at each recorded step, regardless of the number of processes involved.

1.3 Implementation

The full program is divided in five scripts:

- `main.c`: central script that will become executable;
- `init.c`: function to initialize the playground;
- `static.c`: serial and parallel functions of static evolution;
- `ordered.c`: serial and parallel functions of ordered evolution;
- `read_write_pgm_image.c`: functions to read and write PGM files.

The `read_write_pgm_image.c` script is a slightly modified version of the code provided by the professors[3].

1.3.1 main.c

The main.c file serves as the entry point for the program, handling command-line argument parsing (if no argument is provided there are some default values), initialization, and simulation phase. It also manages MPI initialization, task distribution, and timing for scalability study.

The program uses **getopt** (List.1) to parse command-line arguments. These arguments specify the action to be performed (INIT for initialization or RUN for the evolution), grid size (k), evolution strategy (e, 0 for ordered and 1 for static), number of steps (n), snapshot interval (s, if the value is 0 only the last one will be saved), and file name (fname). This part of the code was provided by the professors[3].

```c
    while ((c = getopt(argc, argv, optstring)) != -1) {
        switch (c) {
        case 'i':
            action = INIT;
            break;
        case 'r':
            action = RUN;
            break;
```

Listing 1: Command-line argument parsing using getopt

The program begins by initializing the MPI environment (MPI_Init), determining the rank of the current process (MPI_Comm_rank) and calculating the total number of processes (MPI_Comm_size).

A conditional statement determines the selected action: the program will either initialize the playground or start the simulation, depending on the provided input.

In the initialization phase, the program checks if the current process is rank 0, as all input/output operations are handled by this rank to ensure consistency. The grid is initialized (initialize_playground()) and saved as a PGM image for visualization and further use (List.2).

```c
    if (action == INIT) {
        unsigned char *playground = NULL;
        if (rank == 0) {
            playground = (unsigned char *)malloc(k * k * sizeof(unsigned char));
            initialize_playground(playground, k, k);
            write_pgm_image(playground, 255, k, k, fname);
            printf("Playground initialized and saved to %s\n", fname);
            free(playground);
        }
```

Listing 2: Initialization phase

In the simulation phase, the program further distinguishes between the number of MPI processes: if **size > 1**, it executes the MPI version of the evolution functions, otherwise, it defaults to the serial versions.

Focusing on the MPI execution, the PGM image is read only by rank 0 to not waste memory. The parameters, such as grid dimensions, evolution strategy, and the number of iterations, are broadcast to all MPI processes (MPI_Bcast()), ensuring a consistent distributed environment.

The grid is then partitioned by rows using domain decomposition. Each MPI task is assigned a contiguous block of rows, dynamically allocated by each process to account for uneven distributions. If the **height** is not even divisible, some processes need one extra row. If the **rank < remainder** that process get it.(List.3).

```c
    int base_height = height / size;
    int remainder = height % size;
    int block_height = height / size + (rank < remainder);
    unsigned char *local_playground = (unsigned char *)malloc(block_height * width
     * sizeof(unsigned char));
    int *proc_counts = NULL, *displs = NULL;

    if (rank == 0) {
        proc_counts = (int *)malloc(size * sizeof(int));
        displs = (int *)malloc(size * sizeof(int));
        for (int i = 0; i < size; i++) {
            proc_counts[i] = (i < remainder ? base_height + 1 : base_height) *
    width;
            displs[i] = (i == 0 ? 0 : displs[i - 1] + proc_counts[i - 1]);
        }
    }
```

Listing 3: Domain decomposition

The displacements and block sizes are calculated and scattered to the respective processes using `MPI_Scatterv()`, enabling each task to receive its portion of the grid.

Depending on the evolution strategy (`ORDERED` or `STATIC`), the corresponding evolution function `mpi_ordered_evolution` or `mpi_static_evolution` is invoked.

To measure the execution time, the program uses `clock_gettime(CLOCK_MONOTONIC)` for its high precision and independence from system clock adjustments. The timing data is used to compute performance metrics, such as the total elapsed time and the average time per iteration.

1.3.2 `init.c`

The `initialize_playground()` (List.4) function initializes the grid by assigning each cell a random value: 255 for alive or 0 for dead cells.

To efficiently handle large grids, the initialization process is parallelized using OpenMP. The directive `#pragma omp parallel` creates a parallel region, spawning multiple threads, each with its own thread ID. This ID is used to generate a thread-specific random seed, ensuring that each thread produces an independent sequence of random numbers. This avoids conflicts or duplication of cell states across the grid.

```c
void initialize_playground(unsigned char *playground, int width, int height) {
    // Seed the random number generator
    unsigned int seed = (unsigned int)time(NULL);

    #pragma omp parallel
    {
        unsigned int thread_seed = seed + omp_get_thread_num();
        #pragma omp for collapse(2) schedule(static)
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                playground[y * width + x] = (rand_r(&thread_seed) % 2 == 0) ?
    ALIVE : DEAD;
            }
        }
    }
}
```

Listing 4: init.c

Within the parallel region, the directive `#pragma omp for collapse(2)` is applied to the nested loops. The `collapse(2)` clause treats the two loops as a single iteration space, enabling

OpenMP to distribute the workload more evenly among threads. The `schedule(static)` clause ensures that iterations are divided into equal-sized chunks, assigning an even workload to each thread, since the grid size does not change.

1.3.3 `static.c`

In the `static.c` script two versions of the static evolution are implemented: `mpi_static_evolution()`, for execution across multiple MPI tasks, and `static_evolution()`, which runs only when there is a single MPI task. The former takes several variables from `main.c`, including the playground, the grid's dimensions, and parameters related to the MPI processes.

It dynamically allocates (using `malloc()`) several arrays such as: `global_playground`, allocated in rank 0 to reconstruct the full grid when saving snapshots, the two halo rows (or border rows), allocated for each MPI task, allowing them to exchange boundary information with neighboring processes, and `next_playground` array, used to implement the "freezing" phase of static evolution, ensuring that the original grid `playground` remains unchanged while the new state is computed.

It then computes the neighbors of each process on the border regions to determine at which rank each MPI task needs to send and receive halo rows information during communication. It takes in account the periodic boundary condition, selecting the correct `rank` process. It also precomputes the relative positions of neighboring cells to optimize performance, avoiding redundant calculations (List.5).

```c
    // Determine neighbors of MPI processes
    int neighbor_top = (rank == 0) ? size - 1 : rank - 1;
    int neighbor_bottom = (rank == size - 1) ? 0 : rank + 1;

    // Precompute neighbor offsets for performance
    int neighbor_offsets[8][2] = {
        {-1, -1}, // Top-left
        {-1,  0}, // Top
        {-1,  1}, // Top-right
        { 0, -1}, // Left
        { 0,  1}, // Right
        { 1, -1}, // Bottom-left
        { 1,  0}, // Bottom
        { 1,  1}  // Bottom-right
    };
```

Listing 5: Neighbors rank and precomputed offsets

At each step, non-blocking communication is used between MPI processes to enable faster data exchange and prevent deadlocks (List.6).

```c
    MPI_Request requests[4];
    for (int step = 0; step < tot_steps; step++) {
        MPI_Irecv(halo_top, width, MPI_UNSIGNED_CHAR, neighbor_top, 0,
    MPI_COMM_WORLD, &requests[0]);
        MPI_Irecv(halo_bottom, width, MPI_UNSIGNED_CHAR, neighbor_bottom, 1,
    MPI_COMM_WORLD, &requests[1]);
        MPI_Isend(playground, width, MPI_UNSIGNED_CHAR, neighbor_top, 1,
    MPI_COMM_WORLD, &requests[2]);
        MPI_Isend(playground + (block_height - 1) * width, width,
    MPI_UNSIGNED_CHAR, neighbor_bottom, 0, MPI_COMM_WORLD, &requests[3]);
```

Listing 6: Communication between MPI processes

One key advantage of non-blocking communication is that while messages are being transmitted, the code can compute in the meanwhile. Since the exchange of halo rows is only required for evaluating the statuses of border row cells, the central part of the grid can be processed independently without waiting for communication to complete. To take advantage of this, two separate loop updating sections are implemented: one for updating the inner region of the grid, which can be computed immediately, and another for updating the top and bottom halo rows after communication has finished. These loop sections are separated by a call to `MPI_Waitall()`, ensuring that all data exchanges are completed before proceeding with the computation of border rows.

Both updating sections utilize `#pragma omp parallel for collapse(2) schedule(static)` to implement multi-threading, taking advantage of OpenMP parallelism within each MPI process. By collapsing the two nested loops, the workload is distributed evenly across available threads. Within these loops, the update rules for each cell, as seen in previous sections, are applied based on the statuses of its neighboring cells, considering the periodic boundary condition (List.7).

```
#pragma omp parallel for collapse(2) schedule(static)
for (int row = 1; row < block_height - 1; row++) {
    for (int col = 0; col < width; col++) {
        int alive_neighbors = 0;
         // Calculate alive neighbors for inner rows
        for (int i = 0; i < 8; i++) {
            int offset_x = neighbor_offsets[i][0];
            int offset_y = neighbor_offsets[i][1];
            int neighbor_x = (col + offset_x + width) % width;
            int neighbor_y = row + offset_y;
            alive_neighbors += (playground[neighbor_y * width + neighbor_x]
                                == ALIVE);
        }
```

Listing 7: Cell status updating

The update loop for halo row is very similar: a conditional statement is added to check if the neighbor row of the current cell is in the local grid or in the grid of another process(List.8).

```
    unsigned char neighbor_value;
    if (neighbor_y < 0) {
         neighbor_value = halo_top[neighbor_x];
    } else if (neighbor_y >= block_height) {
        neighbor_value = halo_bottom[neighbor_x];
    } else {
        neighbor_value = playground[neighbor_y * width + neighbor_x];
    }
```

Listing 8: Conditional statement for halo rows

After that, the program swaps the pointers `playground` and `next_playground` to implement the "update at once" phase of static evolution.

In a conditional statement that checks the number of required snapshots, the `MPI_Gatherv()` function is invoked to gather the distributed subgrids from all MPI processes into the root process, reconstructing the full grid. The complete grid is written to a PGM file using `write_pgm_image()` (List.9).

```c
    unsigned char *temp = playground;
    playground = next_playground;
    next_playground = temp;

    // Save snapshots
    if (snap_step > 0 && step % snap_step == 0) {

        MPI_Gatherv(playground, block_height * width, MPI_UNSIGNED_CHAR, global_playground, proc_counts, displs, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

        if (rank == 0) {
            char snapshot_filename[256];
            snprintf(snapshot_filename, sizeof(snapshot_filename), "snapshot_%05d.pgm", step);
            write_pgm_image(global_playground, 255, width, height, snapshot_filename);
            printf("Global snapshot saved: %s\n", snapshot_filename);
```

Listing 9: Updating phase and snapshot saving

At the end of the function, all dynamically allocated arrays are freed to ensure proper memory management and prevent memory leaks.

The serial `static_evolution()` function follows a similar structure but without any functionality related to MPI features, since there is no need for inter-process communication. Notable modifications are:

- halo row exchanges are unnecessary and they are not allocated;
- only one updating loop section is needed;
- no MPI communication is implemented;
- `MPI_Gatherv()` is not required for snapshot saving, since there is just one single process.

1.3.4 `ordered.c`

Also in the `order.c` script, two versions of the ordered evolution are implemented, depending on the number of MPI processes involved. Both versions share a similar structure with the functions in `static.c`, but due to the inherently serial nature of this evolution strategy, which follows a row-major order, several significant changes are introduced:

- no additional subgrid is allocated since there is no evaluation phase;
- non-blocking communication are still implemented, but to mantain the right updating order, no computation are performed during MPI communications, moving `MPI_Waitall()` directly under them with a consequent single loop section for updating;
- an additional loop is added to ensure that each MPI task is processed respecting the correct sequence of grid blocks;
- OpenMP parallelization is adjusted using the `#pragma ordered` directive, enforcing the required row-major processing order.

These modifications significantly reduce the performance benefits of MPI and OpenMP parallelization compared to the static evolution approach. However, they are necessary to respect the strict serial definition of ordered evolution.

1.4 Results & Discussion

Scalability studies are conducted on the THIN partition of ORFEO[4], as the EPYC nodes were unavailable at the time of the experiment. For all the experiments the number of evolution steps is 50 and the OpenMP threads are mapped to cores with a close affinity policy in order to exploit shared resources among close physical cores. Each scenario is repeated five times to ensure more reliable performance measurement and the results are averaged.

The performance evaluation is based on two metrics: **runtime** and **speedup**, as scalability is analyzed with respect to an increasing number of processes.

The speedup is computed as:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

where:

- S_p is the speedup when using p processes,
- T_1 is the execution time using a single process,
- T_p is the execution time using p processes.

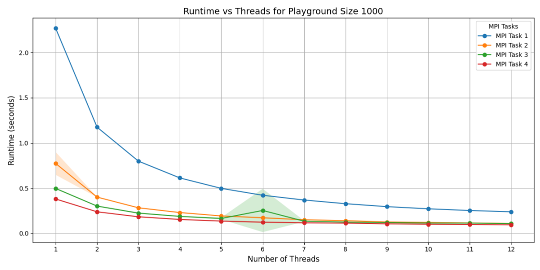
This metric evaluates the efficiency of parallel execution, indicating how much faster the computation becomes as the number of tasks increase. It is then compared to the **ideal speedup**, where the values have a linear correspondence to the number of processes involved.

The elapsed time values are saved in a CSV file and then analyzed in Python in the `analysis.ipynb` notebook.

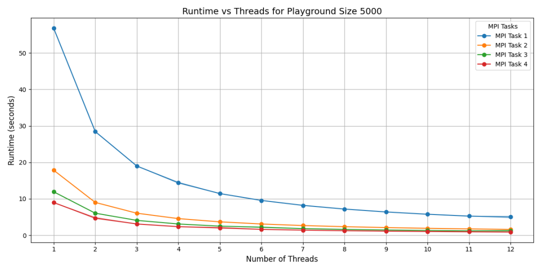
1.4.1 OpenMP Scalability

The **OpenMP scalability** study was conducted on two nodes increasing the number of threads per task up to 12, each thread mapped to a single core. The experiment is repeated for 4 MPI tasks, mapped to one per socket, and for three playground sizes: 1000, 5000, 10000.

In the **static evolution** in Fig.3 and Fig.4 , for all three sizes, the runtime decreases as the number of OpenMP threads increases, which is expected due to the parallelism provided by them. Similarly, as the number of MPI tasks increases, there is a decrease in runtime, benefiting from the division of the workload. Between the sizes we can see how the time increases for greater grids due to more computation involved. These trends show a good scaling as the number of threads increase, with a well management of thread resources.



(a) Runtime for playground size 1000



(b) Runtime for playground size 5000

Figure 3

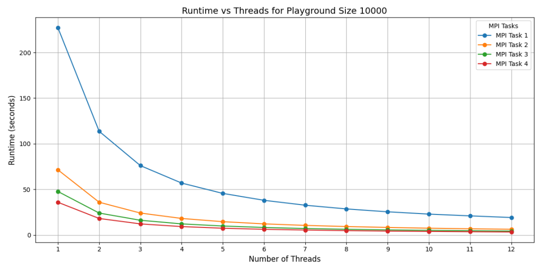
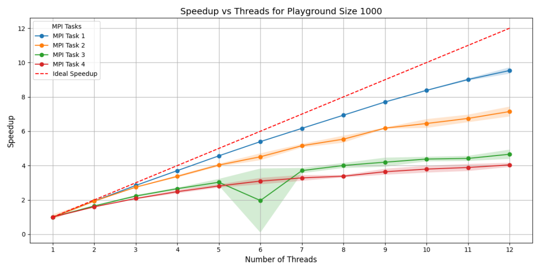
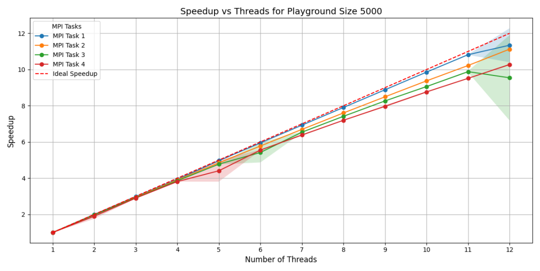


Figure 4: Runtime for playground size 10000

Regarding the speedup, in Fig.5 and Fig.6 we can observe that for grid sizes 5000 and 10000, the values generally follow the ideal trend, especially for a single MPI task. As the number of MPI tasks increases, there is a slight degradation in performance, likely due to communication overhead. This pattern is also reflected in the grid size 1000. However, the speedup values deviate more significantly from the ideal trend, indicating that communication and thread management overheads have a greater impact relative to the smaller workload.



(a) Speedup for playground size 1000



(b) Speedup for playground size 5000

Figure 5

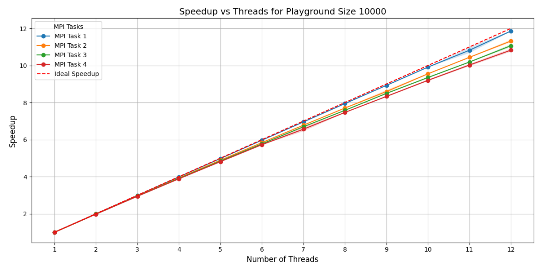
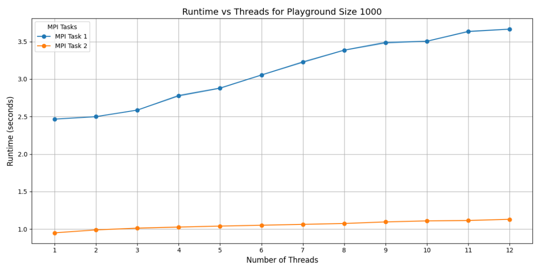
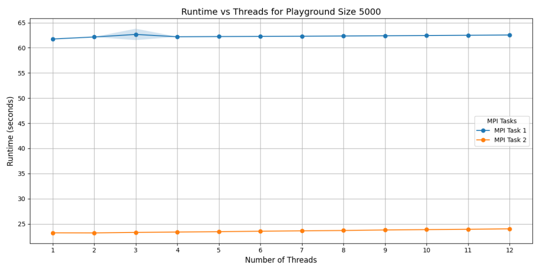


Figure 6: Speedup for playground size 10000

For the **ordered evolution**, the study is conducted using grid sizes of 1000 and 5000, with two MPI tasks. In Fig.7 we observe that the results align with expectations for a strictly serial evolution. The OpenMP scaling remains flat, indicating no significant dependency on the number of threads involved. However, the division of workload between MPI tasks provides a performance benefit, in particular for grid size of 5000. For a grid size of 1000, we can see a slight increase in runtime, likely due to the overhead of thread parallelization.



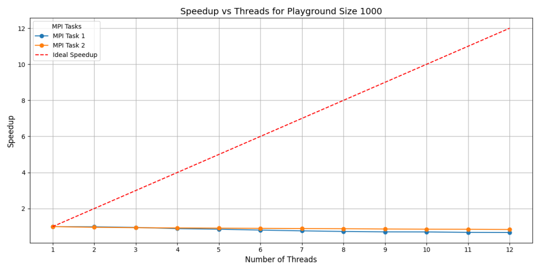
(a) Runtime for playground size 1000



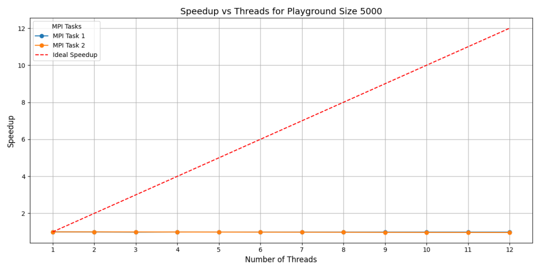
(b) Runtime for playground size 5000

Figure 7: Scaling for ordered evolution

Fig.8 reflects the previous conclusions of the scaling results: the speedups flatten and do not align to ideal values. It shows no exploitation of thread parallelization, as expected.



(a) Speedup for playground size 1000



(b) Speedup for playground size 5000

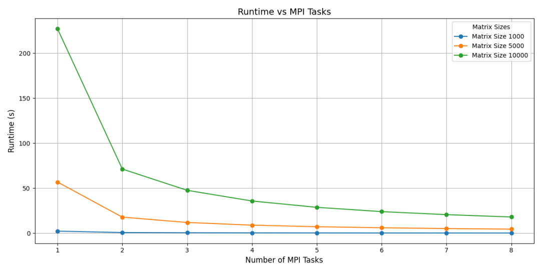
Figure 8: Speedup for ordered evolution

1.4.2 Strong MPI Scalability

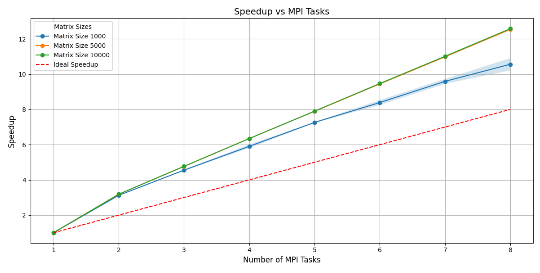
The **strong MPI scalability** study is conducted on 4 nodes, varying the number of MPI tasks up to 8, mapping them to sockets. The number of thread is fixed to 1 per process in order to highlight the behavior of pure MPI processes in this study. The experiment is repeated for playground sizes of 1000, 5000 and 10000.

In Fig.9(a), the **static evolution** shows good scaling, indicating efficient utilization of MPI parallelism. The surprising results appear in Fig.9(b), where the speedup exceeds the ideal values and improves as the size increases. This suggests good cache usage as dividing the grid between tasks, the workload per process may fit better in caches, reducing memory access times expecially for very large grids.

We can expect similar results as in OpenMP scalability study for the **ordered evolution**. In Fig.10, it can be observed that the parallelization provided by MPI tasks give an improvement dividing the computations per processes, but since they need to be executed in order, the time and consequently the speedup, flatten.

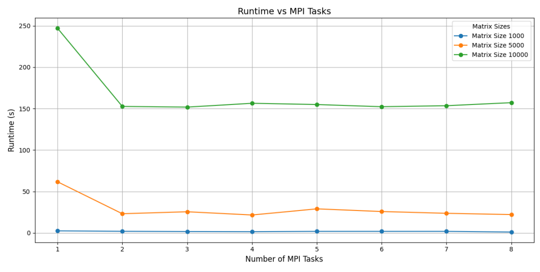


(a) Runtime for static evolution

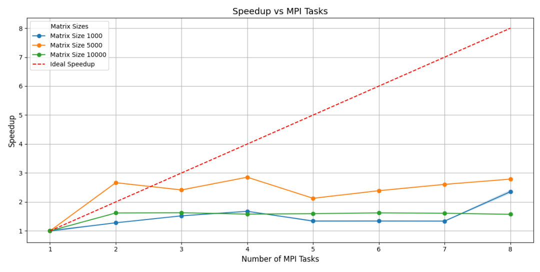


(b) Speedup for static evolution

Figure 9: Strong scalability for static evolution



(a) Runtime for ordered evolution



(b) Speedup for ordered evolution

Figure 10: Strong scalability for ordered evolution

1.4.3 Weak MPI Scalability

The **weak MPI scalability** study is conducted on 4 nodes by fixing the workload for 8 MPI tasks, each mapped on one socket saturated with threads (`OMP_NUM_THREADS=12` for THIN nodes). This means that the grid sizes must vary proportionally with the number of MPI tasks. Specifically, the sizes follows the formula:

$$k_j = k_{\text{init}} \sqrt{\text{MPI}_j} \tag{2}$$

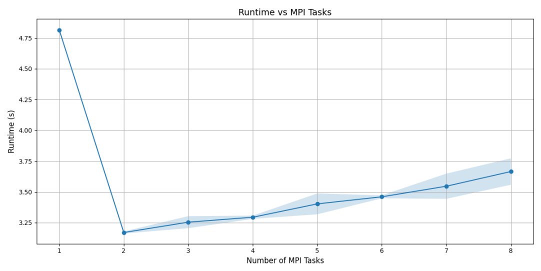
where:

- k_j is the grid size for the j-th MPI task
- k_{init} is the initial grid size
- MPI_j is the number of MPI tasks

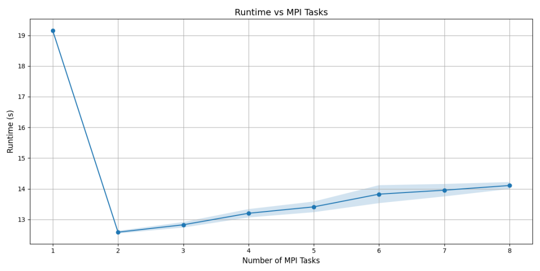
The first experiment was conducted with an initial playground size of 5000. In Fig.11(a), it is shown the execution time: it decreases when using two MPI tasks and then it slightly increases in next tasks. To confirm this behavior, a second study was repeated with a starting playground size of 10000.

Both experiments show an initial decrease in runtime when two MPI tasks are used, followed by a slight increase. This sudden drop, followed by a near-flattening, highlights the computational advantages when multiple processes are involved. But it also can suggest that the serial implementation is not optimized as it should.

The increase in runtime can be attributed to the overhead introduced by communication between MPI tasks. This behavior suggests that the program does not scale perfectly with a proportional workload, but overall, the results indicate a good level of scalability.



(a) Initial size of 5000



(b) Initial size of 10000

Figure 11: Weak scalability for static evolution

1.5 Conclusions

In this study, we conducted three types of scalability tests (OpenMP, strong MPI, and weak MPI) on two different evolution strategies for the Game of Life.

The **ordered evolution** showed its inherently serial nature the results are the ones expected: increasing the number of processes and distributing the workload did not significantly improve performance. While some advantage was observed with additional MPI tasks, the speedup results indicated that there is no real gain in performance.

One potential way to improve scalability is to relax the strict row-major update order and allow some level of parallelization. However, this would diverge from the original definition of the evolution process.

In contrast, the **static evolution** demonstrated strong performance in OpenMP scalability, with efficient thread management and parallelization. Even more impressive results were observed in the strong scalability tests, where it achieved speedup values exceeding the ideal ones, benefiting significantly from the division into MPI tasks and managing the memory optimally.

In the weak scalability test we can observe good results, even though they are not perfect. The big drop in elapsed time is a signal of how parallelization allows better computation management, but it also may suggest that there is room of improvement for what concern the serial implementation.

In any case, this evolution strategy is well-suited for parallelization. By breaking the process into two steps, evaluation and update, it becomes easier to implement in parallel. The evaluation phase, in particular, benefits significantly from parallel execution since there is no dependencies between updates of cell statuses, maximizing the advantages of processes division with gains in performance improvements.

Future works could focus on improving the handling of larger grids by scaling resources proportionally to enhance weak scalability. In addition, the serial implementation should be checked in several more scenarios, to fully understand its criticality and performance.

The program can also be tested under different MPI and OpenMP configurations or different architectures, as the Epyc nodes, to better understand how it manages workload distribution and scalability.

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

2.1 Introduction

The objective of this exercise is to evaluate and compare the performance of three widely used HPC math libraries: **Intel MKL[5], OpenBLAS[6], and BLIS[7]**. The focus of this comparison is on **matrix-matrix multiplication**, specifically the `GEMM` (General Matrix Multiply) function from **Level 3 BLAS (Basic Linear Algebra Subprograms)**. The operation performed follows the standard equation:

$$C = \alpha AB + \beta C \tag{3}$$

where α and β are scalar coefficients, and A, B, and C are matrices. In this study, the computation is simplified by considering only square matrices and setting $\alpha = 1$ and $\beta = 0$, reducing the equation to:

$$C = AB \tag{4}$$

To assess the scalability and efficiency of these libraries, experiments are conducted under two different tests:

- **Size scalability:** increasing the size of the matrices while keeping the number of cores fixed.
- **Core scalability:** increasing the number of cores while keeping the matrix size constant.

The experiments were carried out on two distinct architectures: **THIN** and **EPYC** nodes, using both **single** (SGEMM) and **double precision** (DGEMM) floating-point computations. Additionally, two thread allocation policies, **close** and **spread**, were applied to observe their impact on performance.

This results in a total of 16 scenarios, each of which will be compared against the theoretical peak performance of the corresponding processors.

2.2 Implementation

The main file `gemm.c` and the corresponding `Makefile` are provided by the professors[3]. Minor changes are made to the latter to handle the correct computation and creation of executables. The main script implements matrix-matrix multiplication for all three libraries, supporting both single and double precision floating-point arithmetic. Additionally, it measures execution time and computes GFLOPs for performance comparison. Multi-threading is managed through the `GEMMCPU` function, as the BLAS implementations in all three libraries handle parallelization internally.

In **ORFEO**[4], the OpenBLAS library is pre-installed, whereas MKL and BLIS need to be manually downloaded and compiled. After installation, the correct library paths are set using the `export LD_LIBRARY_PATH` command to ensure proper location.

The only implementation aspect to consider is managing job submission in the **SLURM** environment (List.10). For both scalability studies, the parameters of each scenario are predefined, and the executables are generated using the command `make cpu` within each partition. The execution process iterates over different BLAS libraries and precision settings. The only difference lies in the final loop:

- for **core scalability**, the loop iterates over the number of threads, ensuring the correct OpenMP and BLIS thread settings are exported;
- for **size scalability**, the loop iterates over the dimensions of the square matrices, adjusting their size accordingly.

```bash
for prec in "${precisions[@]}"; do

    for impl in "${implementations[@]}"; do

        csv="$output_dir/${prec}_${impl}_core_spread_results.csv"
        echo "threads,runtime,GFLOPS" > $csv

        for threads in $(seq 1 24); do

            export OMP_NUM_THREADS=$threads
            export BLIS_NUM_THREADS=$threads

            for run in $(seq 1 5); do

                output=$(srun -n1 --cpus-per-task=$threads $exec_dir/gemm_${prec}
    _${impl}.x $size $size $size)

                runtime=$(echo "$output" | grep -oP 'Elapsed time \K[0-9.]+' |
    head -n1)
                gflops=$(echo "$output" | grep -oP '[0-9.]+ GFLOPS' | grep -oP '
    [0-9.]+')

                echo "$threads,$runtime,$gflops" >> $csv

            done
        done
    done
done
```

Listing 10: Example of shell script

The elapsed time and GFLOPs values are collected through shell operations and regular expressions, saved in a CSV file and then analyzed in Python in the `analysis.ipynb` notebook.

2.3 Results & Discussion

For each scenario, the experiments are executed five times to ensure more reliable performance measurements, mitigating the impact of transient system fluctuations.

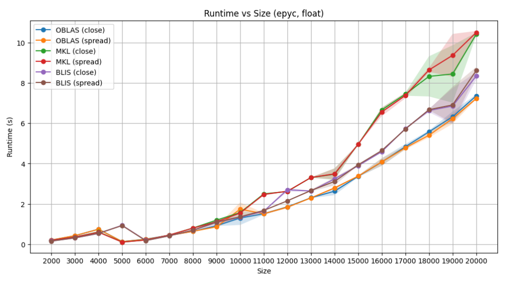
In the case of **size scalability**, the matrix size varies from 2000×2000 to 20000×20000 in increments of 1000. The primary metrics considered are the **runtime** and the **GFLOPs**, which will be compared against the theoretical peak performance of each partition (see Appendix).

For **core scalability**, the number of cores varies from a single one to the maximum possible on a node for each partition. The chosen metrics are **runtime** and **speedup**, to evaluate how well the program scales as the number of threads increases. As seen in the previous exercise, the speedup is computed as:

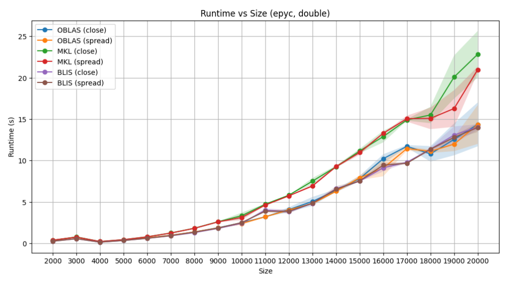
$$S_p = \frac{T_1}{T_p} \tag{5}$$

2.3.1 EPYC

For the **size scalability** study on the **EPYC partition**, a single node with 64 threads (the maximum per socket) is used.



(a) Runtime with single precision



(b) Runtime with double precision

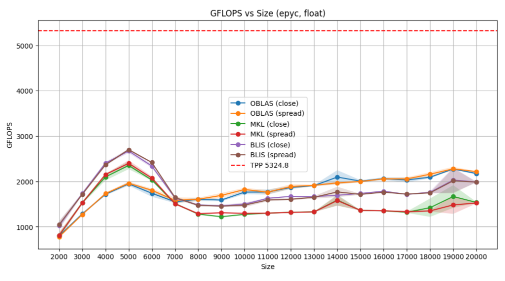
Figure 12: Runtime vs matrix sizes for EPYC partition.

It can be observed in Fig.12 that the runtime increases with the matrix size, which is expected due to the higher computational workload required for larger matrices. In addition between precisions, we can see that the single one achieves lower elapsed times compared to the double as the matrix size increases. The same, but opposite, behavior can be seen in Fig.13, where GFLOPs of float precision have higher values than double ones. This results are expected since single precision computations typically demand less memory bandwidth and consume fewer computational resources than double precision operations.

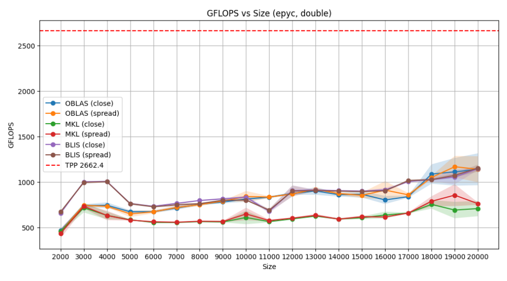
In Fig.13, the GFLOPs exhibit a generally flat behavior, remaining below the theoretical peak performance. This is reasonable, as the number of processes is not increasing. An interesting observation in Fig.13(a) is the maximum GFLOPs achieved around a matrix size of 5000, indicating that the computational resources are most effectively utilized at this size. In addition, around the peak region the OpenBLAS library is the one with less GFLOPs among them, while in the flat region it has the highest values.

In both comparisons there is no significant difference in performance between the thread affinity policies across all three libraries. However, the MKL library exhibits a shift in behavior between single and double precision after a matrix size of 17000.

Among the libraries, the Intel MKL one performs the worst. Since it is optimized for Intel CPUs, its performance may be suboptimal on an EPYC node with an AMD CPU.



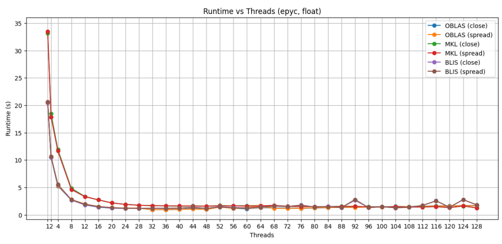
(a) GFLOPs with single precision



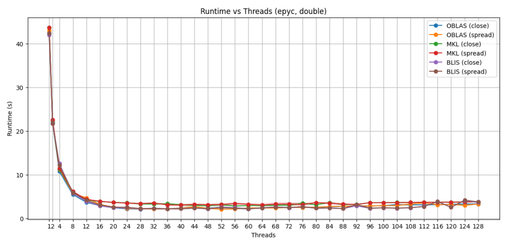
(b) GFLOPs with double precision

Figure 13: GFLOPs vs matrix sizes for EPYC partition.

For the **core scalability** study on the **EPYC partition**, a matrix size of 10000 is used, while the cores varies from 1 to 128 (the maximum per node).



(a) Runtime with single precision



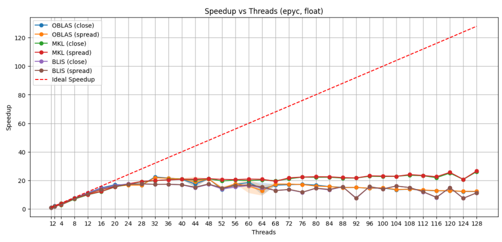
(b) Runtime with double precision

Figure 14: Runtime vs number of cores for EPYC partition.

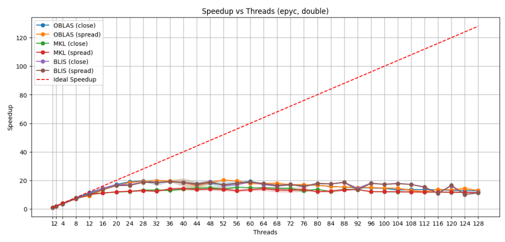
In Fig.14, we can see that the runtime decreases with the increasing number of cores for all three libraries until it stabilizes at a plateau around 24 cores. This behavior is expected due to resource saturation and thread synchronization overhead.

In Fig.15, the speedup of the three libraries follows the ideal values up to 16 cores, after which it begins to flatten, indicating scaling inefficiency. A surprising result is that the MKL library, although it flattens like the others, achieves higher speedup values in single precision, possibly indicating better thread scaling.

Similarly to the size scalability study, there is no significant difference between the thread affinity policies.

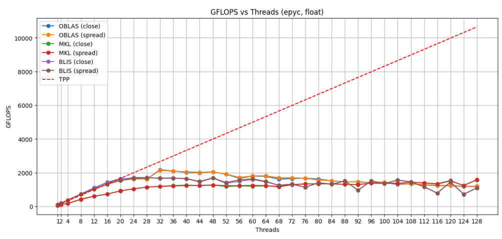


(a) Speedup with single precision

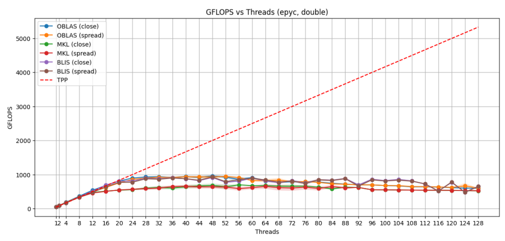


(b) Speedup with double precision

Figure 15: Speedup vs number of cores for EPYC partition.



(a) GFLOPs with single precision

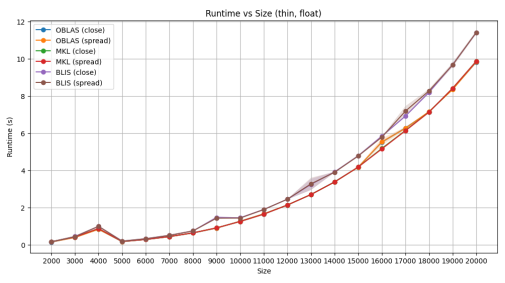


(b) GFLOPs with double precision

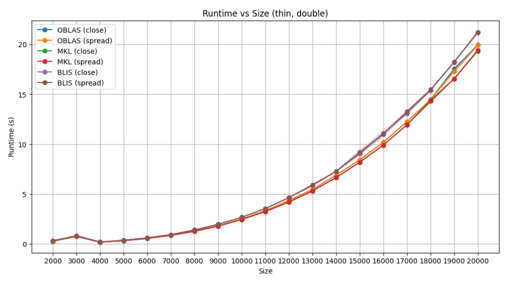
Figure 16: GFLOPs vs number of cores for EPYC partition.

2.3.2 THIN

For the **size scalability** study on the **THIN partition**, it is used a single node with 12 threads (the maximum per socket).

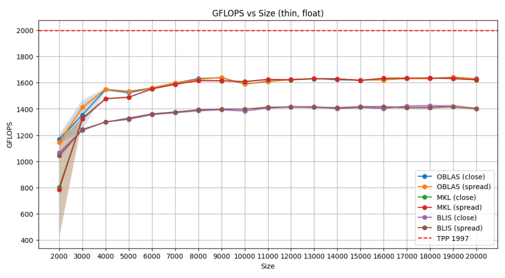


(a) Runtime with single precision

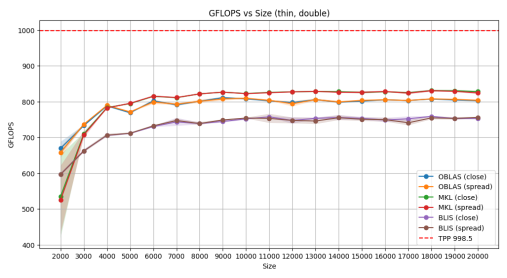


(b) Runtime with double precision

Figure 17: Runtime vs matrix sizes for THIN partition.



(a) GFLOPs with single precision



(b) GFLOPs with double precision

Figure 18: GFLOPs vs matrix sizes for THIN partition.

Similar observations can be made as in the study conducted on the EPYC system: in Fig.17 the runtime increases with the matrix size, while the GFLOPs exhibit a generally flat behavior below the theoretical peak performance.

In constrast with the previous analysis, the GFLOPs in Fig.18 show an initial increase followed by stabilization, without any noticeable peak or decline in computational performance.

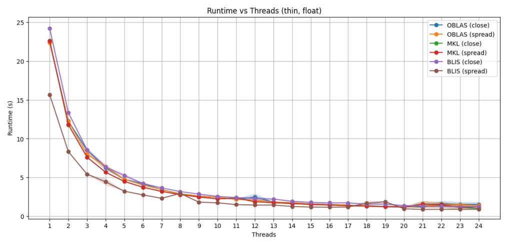
On the THIN node with an Intel CPU, the best performance is achieved by the MKL library, closely followed by the OpenBLAS library. This is reasonable, as MKL is specifically optimized for Intel CPUs, while OpenBLAS can adapt well to the hardware architecture. The BLIS instead shows more than 200 GFLOPs less compared to the others.

For the **core scalability** study on the **THIN partition**, a matrix size of 10000 is used, while the cores varies from 1 to 24 (the maximum per node).

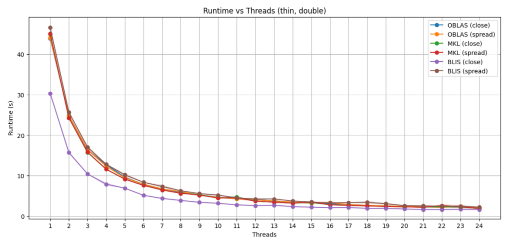
The runtime decreases as the number of cores increases for all libraries, which is an expected result consistent with previous observations. However, the BLIS library exhibits different behavior between single and double precision for fewer than 12 cores: in single precision, the spread policy results in shorter execution times, while in double precision, the close policy performs better.

Regarding the speedup, all three libraries initially follow the ideal speedup curve, with the MKL library showing slightly better results. This behavior holds up to 20 cores for MKL and

OpenBLAS, and up to 17 cores for BLIS. After these values, the speedup of the libraries deviates from the ideal, showing varying performances.

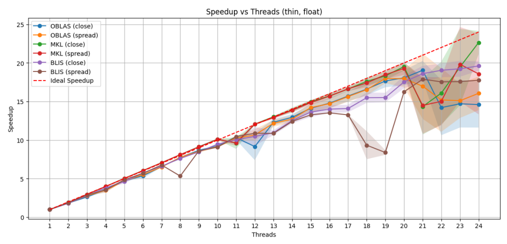


(a) Runtime with single precision

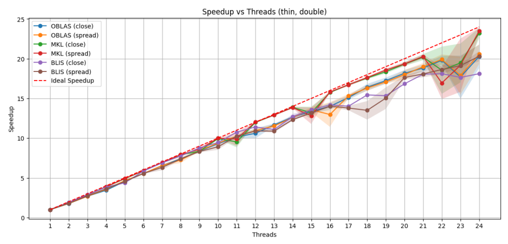


(b) Runtime with double precision

Figure 19: Runtime vs number of cores for THIN partition

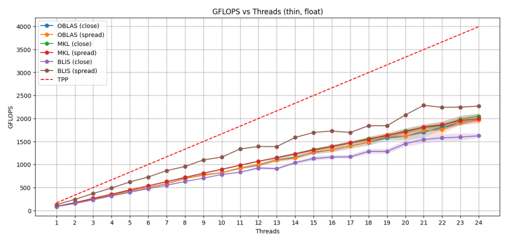


(a) Speedup with single precision

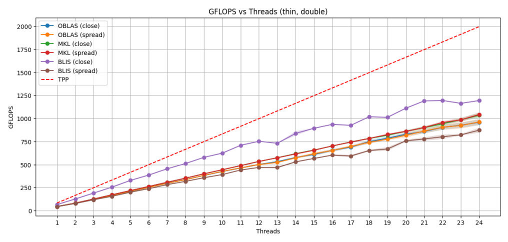


(b) Speedup with double precision

Figure 20: Speedup vs number of cores for THIN partition



(a) GFLOPs with single precision



(b) GFLOPs with double precision

Figure 21: GFLOPs vs number of cores for THIN partition

The peculiar behavior of BLIS library is repeated in Fig.21. It achieves the best GFLOPs over all in spread affinity policy for float operations, while with close one for double operations. These results may be justified due to the fact that in double precision operations, data locality becomes more critical and a close affinity policy might help mitigate cache latency effects.

2.4 Conclusions

This study measured and compared the performance of three BLAS libraries (OpenBLAS, Intel MKL, and BLIS) through size and core scalability tests on two distinct computational architectures: the EPYC node with AMD CPUs and the THIN node with Intel CPUs.

For **size scalability**, runtime increased with matrix size, as expected due to the higher computational workload. The GFLOPs remained below the theoretical peak performance on both systems, likely constrained by resource limitations such as memory bandwidth and L3 cache efficiency. On the EPYC partition, OpenBLAS demonstrated the best performance, while MKL, which is optimized for Intel CPUs, outperformed on the THIN partition. In single precision, a notable peak in GFLOPs was observed around matrix size 5000, indicating optimal resource utilization, likely due to cache reuse and efficient memory access patterns. BLIS consistently underperformed on both architectures, possibly due to less aggressive optimizations for high-core-count architectures.

For **core scalability**, runtime decreased as the number of cores increased until it plateaued, reflecting resource saturation. On both architectures, ideal speedup was observed up to 16 cores, beyond which performance divergence occurred, likely due to NUMA effects, memory contention, and diminishing parallel performances. On the EPYC partition, MKL achieved slightly higher speedup values in single precision, despite running on AMD CPUs. On the THIN partition, MKL performed the best, with OpenBLAS achieving similar performance at higher core counts. The BLIS library seems to have great benefits and drawbacks in this node, since in GFLOPs analysis it showed the best and worst performances between the two thread affinity policies.

Overall, these results underscore the importance of CPU architecture in library selection. MKL remains the best choice for THIN nodes, thanks to its deep integration for Intel CPU and threading optimizations, while OpenBLAS demonstrates strong adaptability across architectures, particularly on AMD EPYC processors. BLIS was the most inconsistent performer, suggesting that further optimizations are needed to fully utilize these two architectures.

References

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

[2] "Conway's Life." https://conwaylife.com/.

[3] Stefano Cozzini, Luca Tornatore. https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/tree/main/Assignment.

[4] Orfeo, Area Science Park. https://orfeo-doc.areasciencepark.it/.

[5] Intel, "Math Kernel Library." https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[6] Kazushige Goto, "OpenBLAS." http://www.openmathlib.org/OpenBLAS/.

[7] Wikipedia contributors, "BLIS." https://en.wikipedia.org/wiki/BLIS_(software).

Appendix

The **Theoretical Peak Performance (TPP)** is computed using the following formula:

$$TPP \text{ (GFLOPS)} = \text{Clock Rate (GHz)} \times \text{Number of Cores} \times \text{FLOPs per cycle} \tag{6}$$

This implies that each partition has a well-defined Theoretical Peak Performance (TPP) for both single and double precision computations.

Each EPYC node consists of two **AMD EPYC 7H12** processors, with a total of 64 cores and a clock speed of 2.6 GHz. The architecture allows for 32 floating-point operations per cycle in single precision and 16 in double precision, leading to:

$$TPP_{\text{single}} = 64 \times 2.6 \times 32 = 5324.8 \text{ GFLOPS} \tag{7}$$

$$TPP_{\text{double}} = 64 \times 2.6 \times 16 = 2662.4 \text{ GFLOPS} \tag{8}$$

Each THIN node consists of two **Intel Xeon Gold 6126** processors, with an established peak performance of **1997 GFLOPS** in single precision for the entire node. The theoretical peak performance for the entire node is then:

$$TPP_{\text{single}} = 1997 \text{ GFLOPS} \tag{9}$$

$$TPP_{\text{double}} = 998.5 \text{ GFLOPS} \tag{10}$$