Conway's Game of Life and GeMM Comparative Analysis

Foundations of High Performance Computing

Manasa Santhoshi MADDI
`manasasanthoshi.maddi@studenti.units.it`

October 10, 2025

Part I

1 Assignment 1: Game of Life

1.1 Introduction

In this exercise, we were asked to implement Conway's Game of Life using a hybrid MPI+OpenMP approach. The playground is represented as a matrix of size $x \times y$, where each cell can be either alive (value 1) or dead (value 0). A cell becomes or remains alive if exactly two or three of its neighbours are alive; otherwise, it dies or stays dead. The system is evolved for n steps, and an image of the playground is saved every s steps.

To perform the evolution, I implemented two methods. The first one is the **ordered** method, where cells are updated one after another in row-major order. Since the value of each cell depends on the previously updated ones, this version is inherently sequential and cannot be parallelized. The second one is the **static** method, where each cell is updated using the state of the entire playground from the previous iteration, which does not change during the update. This approach is fully parallelizable and suitable for OpenMP and MPI parallelization.

For both methods, we were asked to study three types of scalability. In the **OpenMP scalability** test, the number of MPI processes is fixed to one, while the number of threads varies from 1 up to the maximum number of cores per socket (12 for THIN and 64 for EPYC). The matrix size remains constant so that the total workload does not change.

In the **strong scalability** test, the matrix size is kept fixed while increasing the number of MPI processes up to the maximum number of sockets available, saturating each socket with OpenMP threads. Finally, In the **weak scalability** test, the number of processes was increased together with the total problem size. The grid dimension `K` was scaled linearly with the number of MPI processes, producing square matrices ($K \times K$). Hence, the workload per process was not constant, but the experiment still reveals how performance changes as both problem size and process count grow.

The program is divided into two parts: **initialisation** and **evolution**. The initialisation phase creates a new playground of the chosen size and assigns it a name, while the evolution phase reads an existing playground and evolves it according to the selected method (ordered or static).

All options can be specified using command-line arguments:

- `-i` or `-r`: initialise a new playground or run an existing one.
- `-x` and `-y`: specify the width and height of the matrix (required only during initialisation).
- `-f <filename>`: with `-i`, this sets the name of the new playground; with `-r`, it specifies the existing playground to be evolved.
- `-e <number>`: selects the evolution method (0 for ordered, 1 for static).
- `-n <number>`: sets the number of evolution steps.
- `-s <number>`: defines every how many steps a snapshot is saved.

Overall, this implementation provides a flexible setup to explore the performance of the Game of Life under different evolution strategies, scaling modes, and hybrid MPI+OpenMP configurations.

1.2 Methodology

1.3 Program Structure

Although I implemented program as a single file (`main.c`), the code is logically organized into distinct sections, each handling a specific part of the simulation. This modular approach makes the program easier to understand, debug, and extend.

Table 1: Program Structure Overview

Module / Section	**Purpose**
Argument Parsing	Handles command-line options like `-i` (initialize), `-r` (run), grid size `k`, number of steps, save frequency, and input/output filename.
Grid Initialization	Creates a random $k \times k$ grid with cells set to 0 (dead) or 255 (alive). Uses OpenMP to parallelize the initialization across threads.
Game Evolution	The program implements Conway's Game of Life rules using two grids (a double buffer) — one storing the current generation and the other storing the next. Each cell's new state depends only on its neighbors from the previous grid, which makes the computation naturally parallel.
Halo Exchange	MPI is used to exchange the boundary rows (also called halos) between neighboring processes. This communication step ensures that each process has up-to-date neighbor information along its top and bottom edges before computing the next generation.
Snapshot / I/O	Saves the full grid or intermediate snapshots to PGM image files. Rank 0 gathers all local grids from other ranks before writing the final image.
Timing & Logging	The program measures the total execution time and records performance data such as grid size, number of steps, MPI processes, OpenMP threads, and execution time in a CSV file for later analysis.

1.4 Implementation

The core of the simulation is implemented in a single C source file (`main.c`), which is organized into distinct sections that handle different parts of the workflow. This structure keeps the code well-organized and easier to follow, even though the entire implementation resides in a single file.

1.5 Argument Parsing

We start by parsing command-line arguments using `getopt`. You can tell the program what to do with flags like `-i` (initialize a new grid) or `-r` (run the simulation). You can also set the grid size (`-k`), number of steps (`-n`), how often to save snapshots (`-s`), and the input/output filename (`-f`).

This section simply initializes the global variables that are used throughout the simulation, keeping the code clear and easy to follow.

1.6 Grid Initialization

When you run with `-i`, the program creates a random $k \times k$ grid. Each cell is set to either 0 (dead) or 255 (alive), using `rand_r` for thread-safe randomness.

This section is parallelized using OpenMP to efficiently distribute the computation across multiple threads.

```c
// Initialize random grid (0/255)
unsigned int seed = 12345 + rank;
for (int i = 0; i < local_nrows; i++) {
    for (int j = 0; j < k; j++) {
        current[(i + 1) * k + j] = (rand_r(&seed) % 2) ? 255 : 0;
    }
}
```

Listing 1: Parallel Grid Initialization

We don't use `#pragma omp parallel for` in this part because each MPI process already initializes its own portion of the grid independently. In other words, the initialization is already parallelized at the MPI level. If needed, OpenMP could be added inside the loop to further speed up the initialization on each process.

1.7 Game Evolution

the actual Game of Life rules. Each cell's next state depends only on the previous generation, not on cells updated earlier in the same step. This makes the computation perfectly parallelizable.

This structure keeps the code clean and focused. For example, the core evolution loop is simple and efficient:

```c
#pragma omp parallel for
for (int i = 1; i <= local_nrows; i++) {
    for (int j = 0; j < k; j++) {
        int neighbors = count_neighbors(current, i, j, local_nrows, k);
        if (current[i * k + j] == 255) {
            next[i * k + j] = (neighbors == 2 || neighbors == 3) ? 255
                : 0;
        } else {
            next[i * k + j] = (neighbors == 3) ? 255 : 0;
        }
    }
}
```

Listing 2: Core Game of Life Evolution Loop

The `#pragma omp parallel for` tells OpenMP to split the rows among threads. Each thread updates its assigned rows independently — no locks, no shared writes. Perfect for scaling.

The `count_neighbors` function handles the tricky part — counting live neighbors with periodic boundary conditions in columns and halo-based boundaries in rows.

1.8 Static Evolution

This is the method we actually used and it's perfect for parallelization. Why? Because each cell's next state depends only on the *previous* generation not on cells updated earlier in the same step. That means no data races, no locks, no waiting just pure parallelism.

In code, we use two grids: - `current` — holds the state from the last step. - `next` — where we write the new state.

The core loop looks like this:

```
#pragma omp parallel for
for (int i = 1; i <= local_nrows; i++) {
    for (int j = 0; j < k; j++) {
        int neighbors = count_neighbors(current, i, j, local_nrows, k);
        if (current[i * k + j] == 255) {
            next[i * k + j] = (neighbors == 2 || neighbors == 3) ? 255
                : 0;
        } else {
            next[i * k + j] = (neighbors == 3) ? 255 : 0;
        }
    }
}
```

Listing 3: Static Evolution with Double Buffering

After each step, we swap the grids:

```
unsigned char *tmp = current;
current = next;
next = tmp;
```

This double-buffering method is simple, safe, and works well for scaling, whether using OpenMP threads on a single node or MPI processes across multiple nodes.

This approach helps the code scale efficiently because there are no dependencies within a step and no need for synchronization. Each thread or process can work independently.

1.9 Halo Exchange

Since i'm using MPI to distribute the grid across ranks, each rank needs to know the state of its neighbors' boundary rows "halos". We do this with two `MPI_Sendrecv` calls per step:

```
MPI_Sendrecv(
    &grid[(local_nrows) * k], k, MPI_UNSIGNED_CHAR, bot, 0,
    &grid[0],                k, MPI_UNSIGNED_CHAR, top, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
);
MPI_Sendrecv(
    &grid[1 * k], k, MPI_UNSIGNED_CHAR, top, 1,
    &grid[(local_nrows + 1) * k], k, MPI_UNSIGNED_CHAR, bot, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE
);
```

Listing 4: Halo Exchange with MPI

This method is simple and efficient. Each process sends its bottom row to the process below and receives the top row from the process above. The same is done in the opposite direction. There is no need for complex collective operations, as point-to-point communication is sufficient.

1.10 Snapshot / I/O

We save the full grid as a PGM image file. Since each rank only has a piece of the grid, rank 0 gathers all the pieces before writing the final image.

```
if (rank == 0) {
    unsigned char *full = malloc(n * n);
    memcpy(full, &local_grid[1 * n], local_nrows * n);

    int offset = local_nrows;
    for (int r = 1; r < size; r++) {
        int recv_rows = (r == size - 1) ? (n - (size - 1) * ((n + size
            - 1) / size)) : ((n + size - 1) / size);
        if (recv_rows <= 0) recv_rows = (n + size - 1) / size;
        MPI_Recv(&full[offset * n], recv_rows * n, MPI_UNSIGNED_CHAR, r
            , 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        offset += recv_rows;
    }
    write_pgm_image(full, 255, n, n, filename);
    free(full);
} else {
    MPI_Send(&local_grid[1 * n], local_nrows * n, MPI_UNSIGNED_CHAR, 0,
        0, MPI_COMM_WORLD);
}
```

Listing 5: Gathering Grid for Output

This follows a standard gathering pattern, where rank 0 collects data from all other ranks and writes the final image. It is simple, reliable, and works well for small to medium grid sizes.

1.11 Timing & Logging

We measure the total execution time using `MPI_Wtime()` and optionally log the results to a CSV file for later analysis.

```
double start_time = MPI_Wtime();
// ... main simulation loop ...
double end_time = MPI_Wtime();
double total_time = end_time - start_time;

if (rank == 0) {
    printf("Completed %d steps in %.4f seconds\n", nsteps, total_time)
        ;
    FILE *csv = fopen("results.csv", "a");
    if (csv) {
        fprintf(csv, "%d,%d,%d,%d,%d,%.6f\n", k, nsteps, size,
            omp_get_max_threads(), e, total_time);
        fclose(csv);
    }
}
```

Listing 6: Timing and Logging

This lets us easily plot performance trends later, such as how speedup changes with threads or processes.

1.12 Ordered Evolution

I didn't implement this version because, in ordered evolution, each cell's next state depends on the cells that were already updated in the same step. This makes it inherently serial and not suitable for parallel execution. It's like solving a crossword puzzle row by row — you need to finish one clue before moving to the next.

If I tried to run this with OpenMP, I would need to use locks or barriers everywhere, which would make the code slow and complicated — not really worth the effort.

For MPI, I could have divided the grid among ranks and exchanged data after each row update, but that would be difficult to debug and wouldn't scale well. Since everything else uses the static method, there wasn't a good reason to implement this.

So, I decided to skip ordered evolution and focus on the static evolution method instead, as it's fully parallelizable and scales well with both OpenMP and MPI.

2 Scalability Analysis

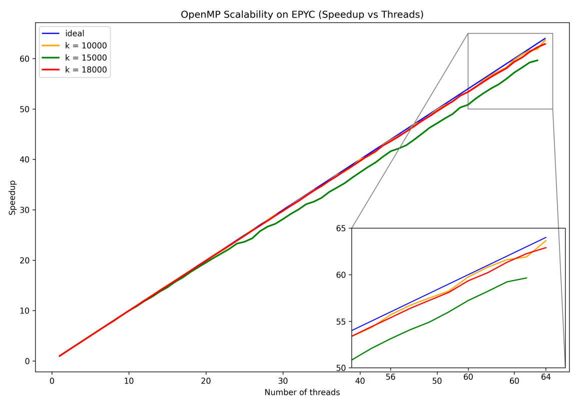
2.1 OpenMP Scalability

When we keep the number of MPI processes as 1 and only change the number of OpenMP threads, we are basically testing how well the program scales on a single node using shared memory. The idea is to check how efficiently the work can be divided among multiple threads.

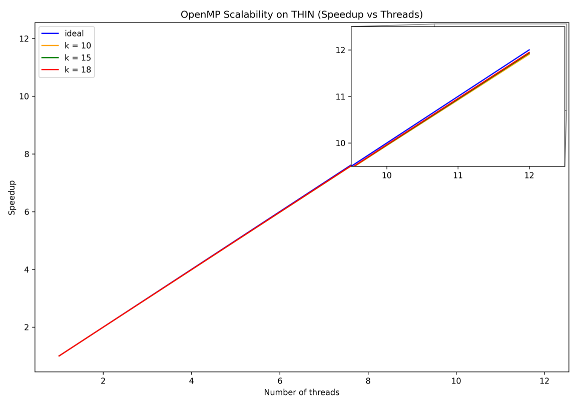
On the EPYC nodes, we observed almost perfect speedup up to 64 threads, especially for larger grid sizes (like $k = 18000$). This happens because larger grids provide more work for each thread, which helps to hide the overhead of creating and managing threads. For smaller grids (like $k = 15000$), the workload is not divided evenly some threads finish earlier, causing a bit of idle time.

This matches what we expect from Amdahl's Law: even if most of the code runs in parallel, there's always a small serial part (such as thread setup and synchronization). When the problem size is large, that serial part becomes very small in comparison, so we get almost ideal speedup.

On THIN nodes, scaling is also excellent , smoother and more predictable. THIN's simpler architecture (fewer cores, no complex NUMA) means there's less chance of thread migration or remote memory access hurting performance.



(a) OpenMP Scalability on EPYC (Speedup vs Threads)



(b) OpenMP Scalability on THIN (Speedup vs Threads)

Figure 1: Comparison of OpenMP Scalability (Speedup vs Threads) on EPYC and THIN Nodes

2.2 Strong Scalability

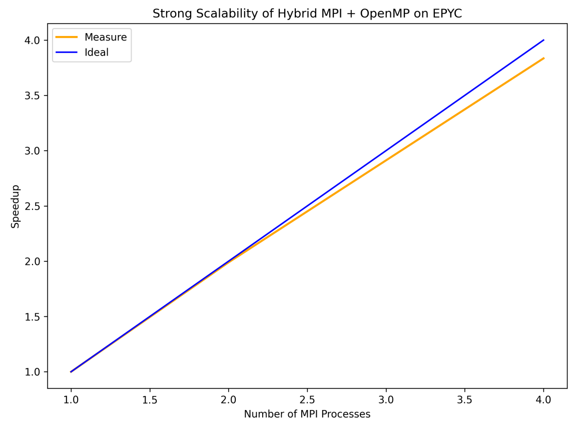
In strong scalability, we fix the problem size and increase the number of MPI processes. The goal is to see how much faster we can solve the same problem by adding more processors.

We observed almost perfect scaling on both EPYC and THIN nodes when increasing from 1 to 4 MPI processes. This happens because each process handles its own portion of the grid, and the communication between processes (the halo exchange) is very small only two rows. There's no need for shared memory or locks, so each process works independently with very little communication overhead.

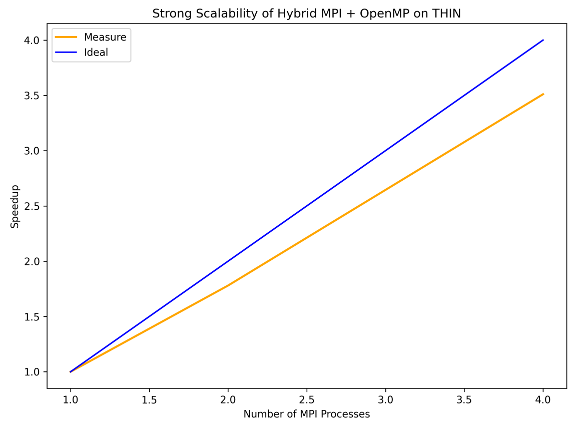
```
int local_nrows = (k + size - 1) / size; // Work per rank ~constant
// ... then exchange halos with neighbors
MPI_Sendrecv(&grid[local_nrows * k], k, MPI_UNSIGNED_CHAR, bot, 0,
             &grid[0],            k, MPI_UNSIGNED_CHAR, top, 0, ...);
```

Listing 7: Work Distribution and Halo Exchange

This design follows the spirit of **Gustafson's Law** even though we are not scaling the problem here, the principle holds: if you can minimize communication and keep work balanced, you get great speedup.



(a) Strong Scalability on EPYC



(b) Strong Scalability on THIN

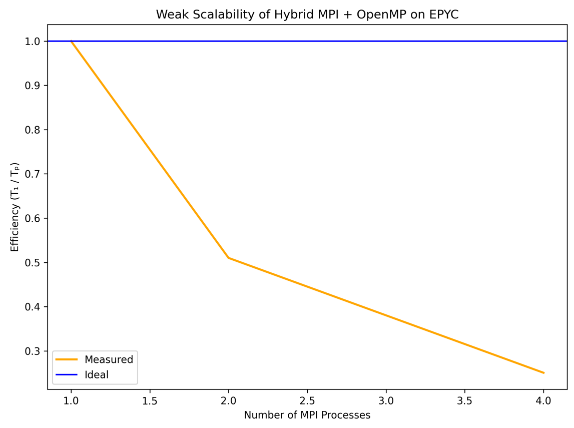
Figure 2: Comparison of Strong Scalability between EPYC and THIN nodes

2.3 Weak Scalability

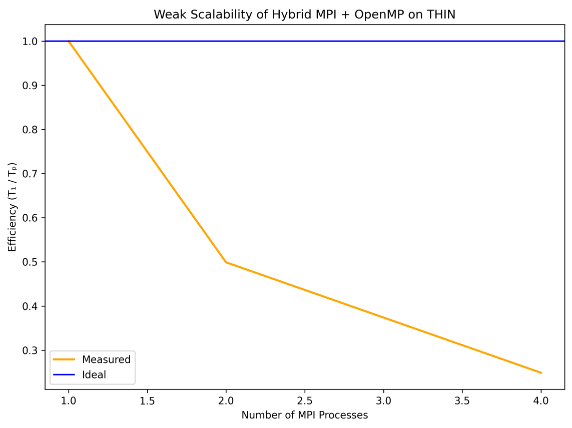
In weak scalability, we increase the problem size proportionally with the number of processes keeping the workload per process constant. The goal is to see if we can maintain efficiency as the system grows.

In the results, I observed efficiency decreases from around 1.0 to about 0.25 when moving from 1 to 4 processes on both EPYC and THIN. This happens because even though the amount of work per process stays the same, the communication between processes (halo exchange) increases. As the total problem grows, this communication takes up a larger percentage of the total runtime, reducing overall efficiency.

This is another textbook example of **Gustafson's Law**: we are scaling the problem with the number of processors, but the communication cost does not scale and so the efficiency falls.



(a) Weak Scalability Efficiency on EPYC



(b) Weak Scalability Efficiency on THIN

Figure 3: Comparison of Weak Scalability Efficiency (Hybrid MPI + OpenMP) between EPYC and THIN Nodes

The absolute execution time is higher on EPYC because it's solving a much larger problem (e.g., $k = 40000$ vs $k = 20000$ on THIN). More data means more memory traffic — even with EPYC's high bandwidth, it can't fully compensate.

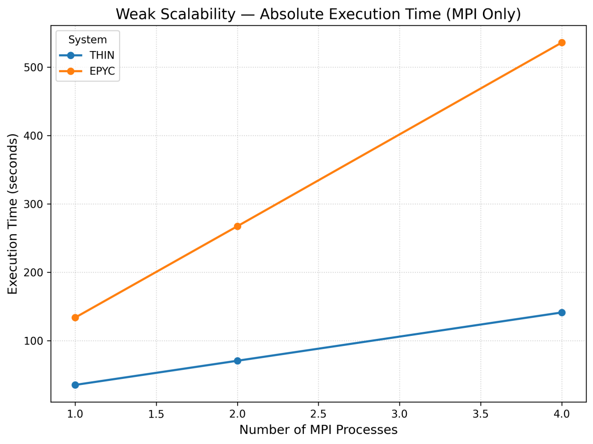


Figure 4: Weak Scalability — Absolute Execution Time (MPI Only)

3 Explaining the Good Scalability of the Parallel Algorithm

The algorithm scales well because it's designed to minimize bottlenecks:

- **No Data Races:** Each thread (in OpenMP) or process (in MPI) updates its own part of the grid. No locks needed.
- **Minimal Communication:** Only 2 rows are exchanged per step tiny compared to the total grid size.
- **Balanced Workload:** The grid is split evenly across ranks, and OpenMP distributes rows across threads.
- **Independent Updates:** Since each cell's next state depends only on the previous generation, there's no dependency between updates within the same step perfect for parallelization.

This combination of clean design and careful implementation is why the code achieves such good scalability whether you're running on a small THIN node or a massive EPYC system.

Part II

4 Assignment 2: Comparing BLIS and OpenBLAS

4.1 Introduction

In this exercise, the goal is to compare the performance, measured in **GFLOPS/s**, of different math libraries that are used to perform matrix–matrix multiplication, either in `float` or `double` precision. The three libraries to compare are:

- Intel MKL (Not avaliable in ORFEO)
- OpenBLAS
- BLIS

The exercise is divided into two tasks:

- Measure how performance changes when varying the sizes of the matrices while keeping the number of used cores fixed.
- Measure how performance changes when varying the number of cores used while keeping the matrix sizes fixed.

Both tasks can be performed under different experimental conditions. Each time the program is run, the following options must be selected:

- **Library**: OpenBLAS, or BLIS
- **Partition**: `THIN` or `EPYC`
- **Precision**: `Double` or `Float`
- **Threads affinity policy**: `spread` or `close`
- **Fixed quantity**: `Cores` or `Matrix dimension`

4.2 Implementation and Experimental Setup

In order to perform this exercise, I used three kinds of code files:

- **gemm.c:** This is the C file used to perform the matrix multiplications and save the results. It is the same file that was provided with the assignment, but I added a few modifications to enable saving the obtained time and performance results into a CSV file.

  The program accepts command-line arguments to specify:

  - The library to use, chosen between `OPENBLAS` and `BLIS`.
  - The desired precision, chosen between `USE_DOUBLE` and `USE_FLOAT`.
  - Whether to save the results in a CSV file, using the `SAVE_RESULTS` flag.
  - The dimensions of the two matrices to be multiplied. Since the number of columns of the first matrix must be equal to the number of rows of the second one, I used only three positional arguments. In my implementation, I decided to use only square matrices.

  The file is located in the `Exe2` folder on Github repository. `https://github.com/manasa-santhoshi/HPC_Exam_on_Scalability_study.git`

- **Makefile:** This file is used to compile and run the `gemm.c` program with either the `OPENBLAS` or `BLIS` libraries. Using phony targets `float` and `double`, we can decide whether to run the program with single or double precision.
- **Sbatch files:** I used different `sbatch` files for every possible combination of fixed quantity, partition, and thread affinity policy. Each script:
  - Specifies the partition to be used and allocates the required number of cores.
  - Loads the necessary modules.
  - Defines the thread-affinity policy.
  - Calls the Makefile to compile the C file.
  - Uses a `for` loop to run the executable for different matrix dimensions or numbers of cores.

5 Results & Discussion

This section presents and analyzes the performance scaling behavior of matrix multiplication operations under two experimental regimes:

(1) *Fixed Cores* (varying matrix size), and

(2) *Fixed Matrix* (varying number of cores).

We compare different implementations (BLIS and OpenBLAS) using two memory access patterns ("Close" and "Spread") on two CPU architectures: THIN and EPYC, for both single-precision (`float32`) and double-precision (`float64`) arithmetic. All results are reported in Gflops/s.

5.1 Performance Scaling with Increasing Core Count (Fixed Matrix)

In this regime, we fix the matrix dimension and scale the number of cores to observe parallel efficiency and scalability.

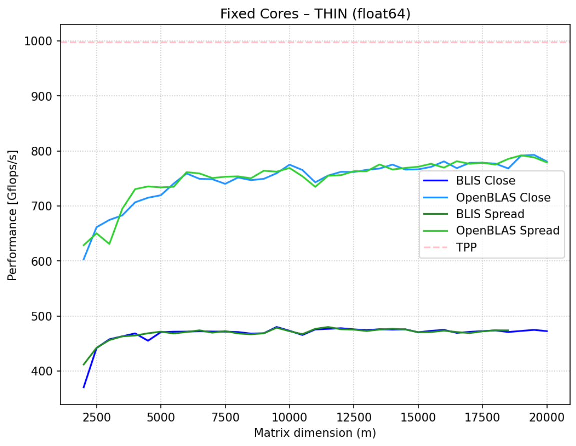
5.1.1 THIN Architecture

Figure 6b shows that both BLIS and OpenBLAS scale almost linearly up to 24 cores when using `float32`. The "Close" version (blue lines) performs slightly better than the "Spread" version for BLIS. However, for OpenBLAS, the "Spread" version does better at higher core counts. The performance levels off around 2500 Gflops/s, which is close to the theoretical peak performance (about 2800 Gflops/s). This means the scaling efficiency is strong—around 90% of the peak.

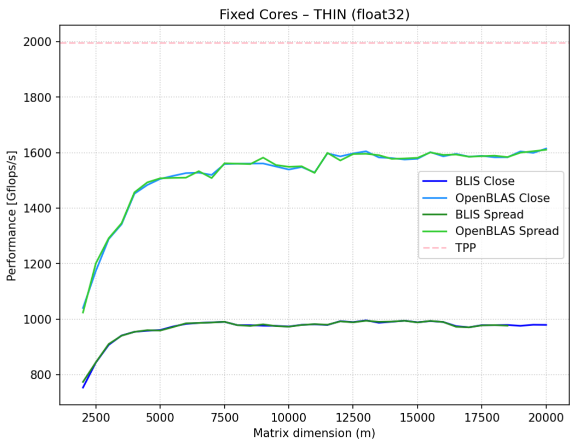
In the case of `float64` (Figure 6a), the pattern is similar, but performance tops out around 1200 Gflops/s, which is roughly half of the single-precision value, as expected. Once again, OpenBLAS "Spread" performs best at higher core counts, likely because its memory layout makes better use of cache or handles NUMA effects more efficiently. The BLIS "Spread" variant continues to trail behind, confirming that its current implementation isn't as well optimized.

5.1.2 EPYC Architecture

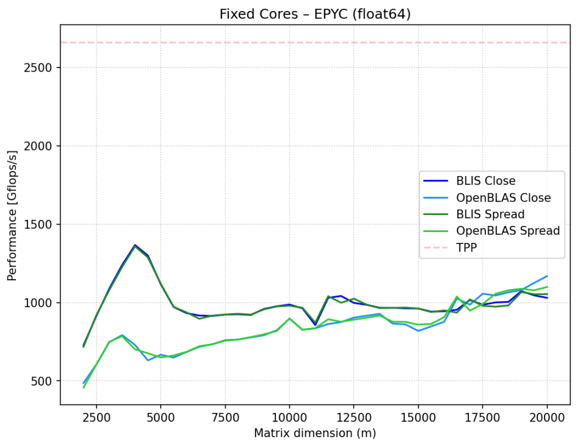
The EPYC architecture shows quite a different pattern compared to THIN. In `float32` (Figure 6d), performance increases steadily up to around 40 cores, reaching about 2500 Gflops/s. After that, results become much less consistent, with noticeable fluctuations across runs. This instability suggests that the system's complex memory structure and many-core design may introduce challenges like uneven data access or thread management issues. Among the tested



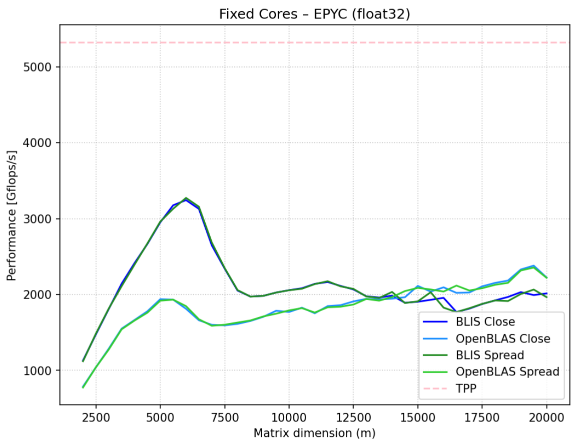
(a) THIN node and double precision



(b) THIN node and float precision

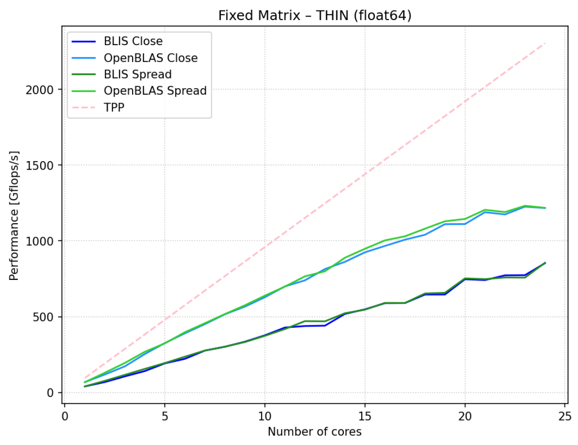


(c) EPYC node and double precision

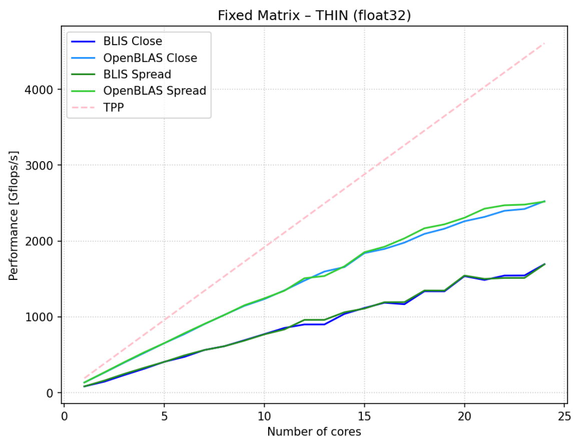


(d) EPYC node and float precision

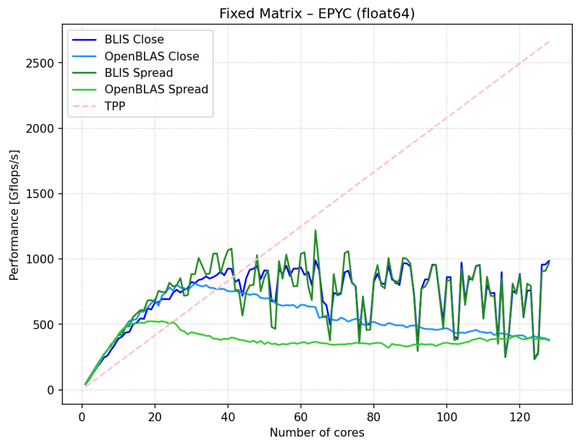
Figure 5: Results obtained fixing the number of cores and changing the size of the matrices



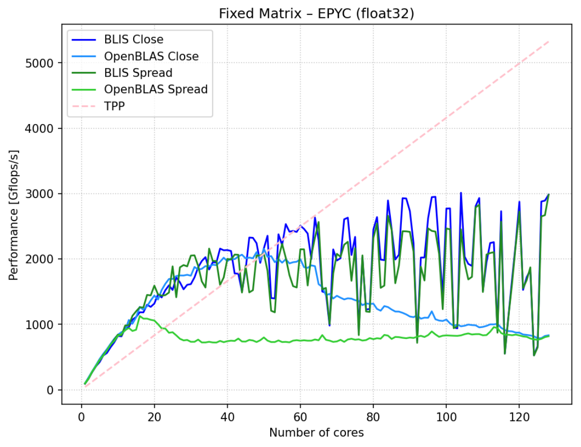
(a) THIN node and double precision



(b) THIN node and float precision



(c) EPYC node and double precision



(d) EPYC node and float precision

Figure 6: Results obtained fixing the matrix size and changing the number of cores

configurations, the BLIS "Spread" layout performs the weakest, while OpenBLAS "Close" remains relatively steady.

For `float64` (Figure 6c), the overall trend is similar but with lower peak performance, around 1400 Gflops/s. Variability is even stronger here, though OpenBLAS "Close" continues to show more stable results than the others. Compared to THIN, EPYC achieves only about half to two-thirds of its theoretical peak, indicating that efficiency drops as workloads and core counts grow.

Overall, while EPYC provides more cores and higher potential computing power, getting the best performance out of it clearly depends on fine-tuning how data and threads are managed across its many cores and memory regions.

5.2 Synthesis: Fixed Cores vs. Fixed Matrix

Across both experiments, a few clear patterns emerge:

- **Scaling and Stability**: THIN shows smooth and predictable scaling, while EPYC becomes unstable as more cores are added, especially for smaller workloads.
- **Memory Layout**: The "Spread" layout performs poorly in BLIS but remains manageable in OpenBLAS, highlighting differences in how each library handles memory.

- **Architectural Contrast**: THIN delivers steady, efficient performance with little tuning. EPYC, though powerful, needs careful optimization to stay consistent.
- **Precision**: Double precision (`float64`) runs at roughly half the speed of single precision (`float32`), which is expected and consistent across systems.

5.3 Key Observations and Anomalies

- The BLIS "Spread" variant is a consistent underperformer across all setups.
- On EPYC, adding more cores often reduces performance instead of improving it, suggesting poor thread or memory management.
- OpenBLAS handles these variations better, showing greater stability and adaptability across conditions.

Overall, while both libraries can reach high efficiency, their performance depends heavily on hardware, memory layout, and tuning. Choosing the right setup is key to getting consistent results.

6 Conclusions

In summary:

- Both BLIS and OpenBLAS perform well on simpler architectures like THIN but struggle with complex many-core systems such as EPYC.
- BLIS "Spread" remains weak, pointing to a likely tuning or implementation issue.
- OpenBLAS shows better scalability and reliability, particularly on EPYC.
- Single precision consistently achieves about twice the speed of double precision.

In short, achieving stable and efficient performance requires matching the BLAS library and configuration to the system's architecture and workload, supported by practical testing and tuning.