University of Trieste

Department of Mathematics, Computer Science and Geosciences

High Performance Computing

Poornima Devi Krishnasamy Karthikeyan

SM3500592

Submission Date: 8/11/2025

Contents

Chapter 1

Exercise 1: Game of Life

1.1 Introduction

This assignment focuses on implementing Conway's *Game of Life*, a cellular automaton that evolves on a 2D grid according to simple rules. Each cell in the grid can be either alive or dead, and its state in the next generation depends on the number of its eight neighboring cells that are alive. The evolution rules are as follows:

- A living cell survives if it has 2 or 3 living neighbors.
- A living cell dies if it has fewer than 2 neighbors (underpopulation) or more than 3 (overpopulation).
- A dead cell becomes alive if it has exactly 3 living neighbors.

The grid uses **periodic boundary conditions**, meaning that the edges wrap around to the opposite sides, forming a toroidal (donut-shaped) world. The program was designed to operate in two main modes:

- **Initialization:** generates a random playground of size $x \times y$ (with both dimensions ≥ 100) and saves it to `.pgm` file.
- **Evolution:** loads an existing playground from file and evolves it for a chosen number of generations. During evolution, snapshots of the system are saved every s steps.

The simulation supports different evolution modes, such as *ordered* and *static*: In the **ordered** mode, cells are updated one after another in row-major order, which is mostly a serial operation. In the **static** mode, each cell's next state is computed using the previous generation's grid, allowing the computation to be parallelized efficiently.

The main goal of this project was to develop an efficient **hybrid MPI + OpenMP** implementation capable of handling large grids and studying its scalability. The hybrid approach combines the strengths of both parallel programming models: MPI distributes the grid among multiple processes (in this work, using a 1D domain decomposition). OpenMP parallelizes the computation within each process, allowing multiple threads to update different parts of the grid simultaneously. This setup allows the program to exploit both distributed and shared memory parallelism, improving performance on modern multi-core, multi-node systems. To evaluate the implementation, three types of scalability tests were performed:

- **OpenMP scalability:** one MPI process per socket, increasing the number of threads per process to observe how performance improves with more CPU cores.
- **Strong MPI scalability:** keeping the total problem size fixed while increasing the number of MPI processes, to measure how efficiently the program speeds up.
- **Weak MPI scalability:** increasing both the number of MPI processes and the overall problem size so that each process handles the same workload, testing how well the code scales to larger systems.

Further details about the implementation choices, workload division, and scalability results are discussed in the following sections. For more information about the exercise specifications, refer to the original assignment document [?].

1.1.1 Methodology

Workload Division

To divide the workload efficiently among processes, a one-dimensional (1D) decomposition by rows was used. In this setup, the global grid is split horizontally, and each MPI process is assigned a group of consecutive rows. For example, with four processes and a grid of 10 rows, the first two processes may handle three rows each, while the remaining two handle two rows each.

This approach was choosen because it keeps the communication pattern simple and efficient. Exchanging halo rows between neighboring processes is straightforward in a row wise decomposition, as data only needs to move vertically between adjacent processes. The method also ensures a fairly even distribution of work, with only a small imbalance when the total number of rows is not perfectly divisible by the number of processes. Such differences are negligible for large grids and have little impact on performance. Overall, this strategy simplifies halo exchanges, reduces communication overhead, and scales well for large problem sizes.

Within each MPI process, OpenMP threads further parallelize the work. The grid assigned to each process is divided equally among threads, with any remaining rows distributed starting from the first thread. This ensures a balanced load and makes good use of all available CPU cores.

1.1.2 Initialization

Grid initialization follows a centralized approach where only the root MPI process (rank 0) generates the initial configuration. Using a fixed random seed (`srand(42)`), the implementation creates grids with approximately 30% live cells via `rand() % 100 < 30`. This density was chosen to promote interesting Game of Life dynamics while avoiding premature stabilization.

The resulting grid is saved to a PGM file, which is subsequently loaded and distributed to all MPI processes during the evolution phase. To prioritize the reproducibility and simplicity, I figured this initialization is the best choice plus using a fixed initialization across all MPI runs guarantees consistent starting conditions.

Evolution

This simulation uses a static evolution approach, where the grid updates are handled using two separate buffers: one for reading the current state and one for writing the next state. This setup prevents data conflicts during updates and makes it easier to parallelize the computation across MPI processes and OpenMP threads.

Each evolution step consists of three coordinated phases:

1. **Halo Exchange:** Each MPI process exchanges its boundary rows with neighboring processes using non-blocking communication. This step keeps the neighborhood information accurate, especially at the edges, where periodic boundary conditions are applied so that the grid "wraps around."

2. **Parallel Computation:** While halo data is being exchanged, the interior cells are updated in parallel using OpenMP threads. The computation uses both local and halo data to correctly apply the Game of Life rules. Overlapping communication with computation helps reduce waiting time and improves performance.

3. **State Synchronization:** Once all updates are complete, the new state replaces the old one by copying the next buffer into the current buffer. This ensures all processes start the next iteration with consistent data.

The static evolution method was chosen because it is simple, efficient, and easy to parallelize. Ordered evolution methods were avoided since they update cells one after another in a specific sequence, which creates dependencies between updates and limits parallel performance. In contrast, static evolution updates all cells independently, making it more suitable for distributed and multithreaded execution.

MPI and OpenMP Thread Support

Since the program combines MPI and OpenMP, it requires an MPI initialization that supports multithreading. For this reason, the function `MPI_Init_thread()` is used instead of the standard `MPI_Init()`. The program requests the `MPI_THREAD_FUNNELED` level, meaning that although multiple threads can run within each process, only the main thread performs MPI communication. This model matches the structure of the code, where OpenMP threads handle local computation, and MPI calls are made only by the master thread of each process.

This setup is simpler and more portable than using full multithreaded MPI communication (such as `MPI_THREAD_MULTIPLE`), while still allowing efficient hybrid parallel execution.

1.2 Implementation

1.2.1 Source Code Organization

The implementation follows a modular architecture that separates functionality into four primary source files located in the `programs/` directory.

File	Description
`main.c`	The entry point of the program. Handles MPI initialization, argument parsing, and coordinates between initialization (`-i`) and simulation (`-r`) modes.
`game_of_life.c`	Implements the core Game of Life logic, including cell updates, neighbor counting, halo exchange between MPI processes, and OpenMP parallelization.
`io.c`	Manages all file input/output operations in PGM format. Contains both serial and distributed MPI I/O routines (`load_pgm_mpi`, `save_pgm_mpi`, and `save_snapshot_mpi`).
`utils.c`	Contains utility routines for command-line parsing, configuration validation, and program setup.

Table 1.1: Source file organization of the implementation.

Each module exposes its functions via corresponding headers located in the `include/` directory, ensuring modularity and maintainability.

1.2.2 Build System and Compilation

The project uses a `Makefile`-based build system to automate compilation and linking. This setup ensures consistent builds across different systems and simplifies the use of compiler flags. The code is compiled using the MPI compiler wrapper `mpicc` with OpenMP support enabled through the `-fopenmp` flag and the `-O3` optimization level for better performance.

Listing 1.1: Simplified Makefile configuration.

```
CC = mpicc
CFLAGS = -O3 -fopenmp -std=c99 -Iinclude
LIBS = -lm

SOURCES = programs/main.c \
          programs/utils.c \
          programs/io.c \
          programs/game_of_life.c

TARGET = gol

$(TARGET): $(SOURCES)
	$(CC) $(CFLAGS) -o $@ $^ $(LIBS)

clean:
	rm -f $(TARGET) snapshot_*.pgm

.PHONY: clean
```

The Makefile builds a single executable, `gol`, supporting both distributed and shared-memory parallelism. MPI handles inter-process communication, while OpenMP provides thread-level parallelism within each process. This hybrid approach enables scalability across multi-core and multi-node environments.

1.2.3 Execution Workflow

The program supports two main execution modes:

- **Initialization mode (`-i`)** – generates the initial grid and saves it as a PGM file.
- **Run mode (`-r`)** – loads the grid, distributes it across MPI processes, and performs the simulation.

All executions are managed through shell (`.sh`) scripts submitted to the cluster. Each script specifies the number of MPI tasks, threads, and simulation parameters. This keeps the workflow reproducible and consistent with standard HPC job submission practices.

Listing 1.2: Example job submission script.

```
#!/bin/sh
#SBATCH --job-name=gol_run
#SBATCH --nodes=2
#SBATCH --ntasks=8
#SBATCH --cpus-per-task=4
#SBATCH --time=00:30:00
#SBATCH --output=output.log

export OMP_NUM_THREADS=4

srun ./gol -r input.pgm -t 500
```

This structure allows easy adjustment of scaling parameters by modifying the script rather than the source code. It also ensures that results are reproducible across different runs and configurations.

1.2.4 Performance Measurement

Execution times are measured using `MPI_Wtime()` and collected on the root process through `MPI_Reduce()`. This allows the computation of average, minimum, and maximum runtime across processes. The gathered data is later used for scaling and performance analysis.

Snapshots are generated at user-defined intervals and saved as PGM images to visualize the evolution of the system over time. This also allows comparing results from different configurations or hardware setups.

1.3 Results

1.3.1 Theoretical Speed-up Analysis

The parallel performance of the implemented Game of Life simulator is evaluated using the classical *speed-up* metric, defined as:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

where T_{serial} denotes the execution time of the sequential reference implementation, and T_{parallel} represents the corresponding runtime of the parallelized version. A higher value of S therefore indicates greater performance improvement due to parallelization.

1.3.2 OpenMP Strong Scaling

The Figures 1.1 and 1.2 compare OpenMP strong scaling performance on THIN and EPYC nodes. THIN nodes demonstrate superior scaling efficiency, achieving near-linear speedup up to 12 threads, while EPYC performance plateaus beyond 32 threads.

Parameters :	
Evolution :	Static
Partition:	THIN / EPYC
Grid size:	8000
Threads:	THIN: 1, 2, 4, 8, 12/ EPYC: 1, 2, 4, 8, 16, 32, 64

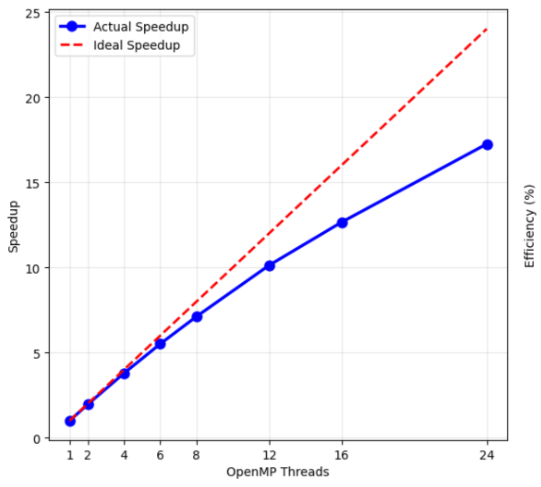

Figure 1.1: THIN OpenMP Strong Scaling

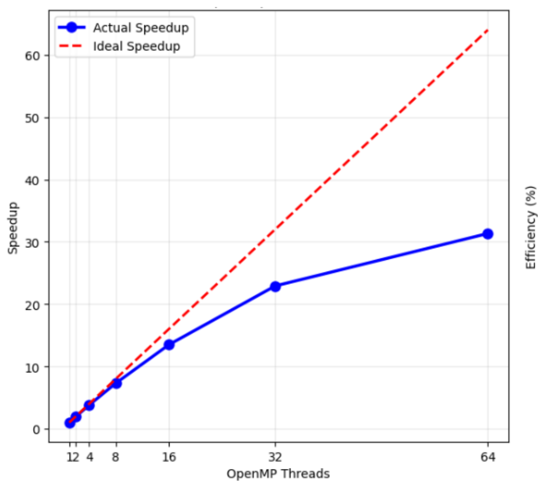

Figure 1.2: EPYC OpenMP Strong Scaling

- On **THIN nodes**, speedup is near-linear up to 12 threads (one socket), with good parallel efficiency within each socket. The 2-socket design creates less severe NUMA penalties than EPYC's architecture. Performance degrades beyond 12 threads due to cross-socket communication overhead.
- On **EPYC nodes**, speedup is near-linear up to 32 threads. Beyond 32 threads, scaling flattens dramatically due to severe NUMA effects across 8 NUMA domains, remote memory access latency, and increased false sharing from higher core counts triggering cache coherence overhead.

These results indicate that OpenMP parallelization is effective up to the point where hardware memory access limits dominate, emphasizing the importance of NUMA-aware allocation for large-scale grids.

1.3.3 MPI Strong Scaling

The Figures 1.3 and 1.4 compare MPI strong scaling performance on THIN and EPYC nodes. Both partitions show near-ideal speedup, with THIN maintaining slightly better efficiency due to lower inter-node communication overhead.

Parameters:	
Evolution:	Static
Partition:	THIN / EPYC
Nodes:	1,2,3
Grid size:	16000 (THIN), 24000 (EPYC)
MPI tasks:	2, 4, 6 $(THIN); 8, 16, 32$ (EPYC)
Threads :	THIN: 12; EPYC: 16

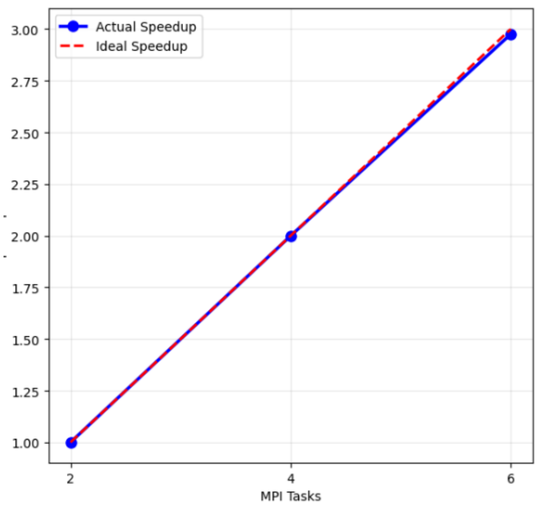

Figure 1.3: THIN MPI Strong Scaling

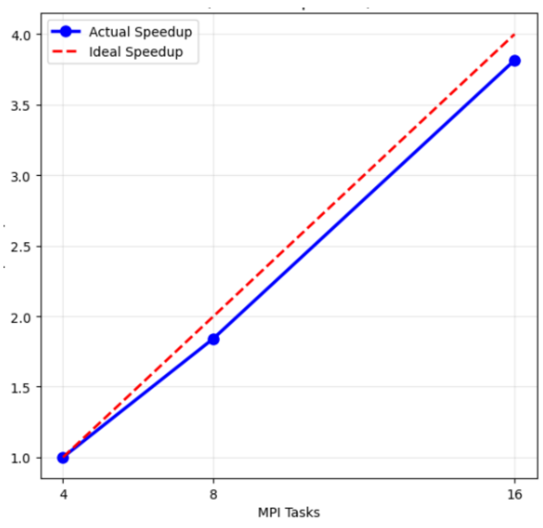

Figure 1.4: EPYC MPI Strong Scaling

- On **THIN nodes**, speedup is nearly ideal up to 6 MPI tasks, indicating minimal communication overhead and efficient load balancing across nodes.
- On **EPYC nodes**, speedup remains close to ideal up to 32 MPI tasks. The slight deviation from linearity is likely due to inter-node communication latency and intra-node NUMA effects at scale.

These results demonstrate that the hybrid MPI+OpenMP implementation scales efficiently in distributed-memory environments, with communication costs remaining small relative to computation time.

1.3.4 MPI Weak Scaling

The Figures 1.5 and 1.6 show MPI weak scaling performance on EPYC nodes. As the number of MPI tasks increases, the global grid size is scaled proportionally to keep work per task constant, in accordance with Gustafson's Law. Ideal behavior is a flat execution time and 100% efficiency.

Parameters:	
Evolution:	Static
Partition:	EPYC
Base local size:	4000
Nodes:	1, 2, 3
MPI tasks:	2, 4, 6
Threads per task:	4, 8, 12, 16, 32, 64
Grid scaling:	$k \propto \sqrt{\text{MPI tasks}}$

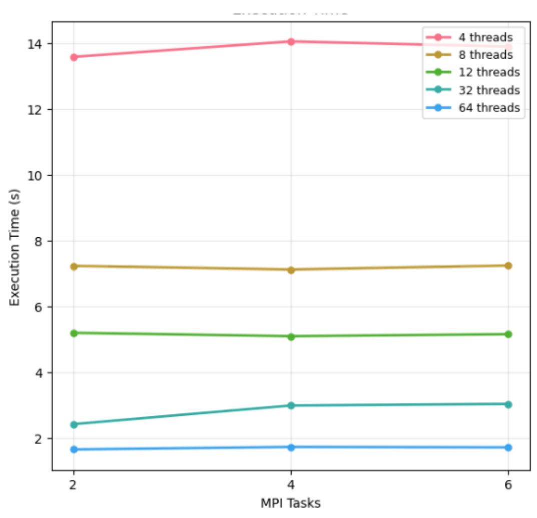

Figure 1.5: Execution time - MPI Weak Scaling

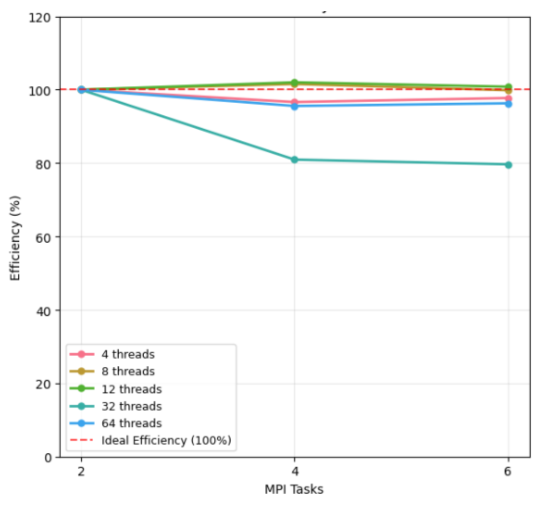

Figure 1.6: Efficiency - MPI Weak Scaling

- Execution time remains relatively constant across all thread counts and MPI task counts, confirming good weak scaling behavior. Minor increases at higher thread counts are likely due to increased communication overhead or NUMA effects.

- Parallel efficiency stays above 95% for most configurations, dropping slightly at 64 threads due to cache coherency costs and memory bandwidth saturation. This indicates the code scales well with problem size and resources.

These results demonstrate that the hybrid MPI+OpenMP implementation follows Gustafson's Law as more resources are added, the problem size grows proportionally, and runtime remains stable, enabling efficient use of large-scale systems.

Chapter 2

Exercise 2

2.1 Introduction

In this exercise, the objective was to evaluate and compare the computational performance expressed in Gflops of different Basic Linear Algebra Subprograms (BLAS) libraries when executing dense matrix-matrix multiplication (GEMM). The libraries under consideration were Intel's Math Kernel Library (MKL), OpenBLAS, and BLIS. The analysis was structured around two primary tasks:

1. Assessing how performance varies with increasing matrix dimensions while keeping the number of CPU cores fixed.

2. Evaluating performance scalability as the number of utilized cores increases, with matrix size held constant.

Each experiment was conducted under configurable conditions, including the choice of library, hardware partition (THIN or EPYC), floating-point precision (single or double), and thread affinity policy. Due to the unavailability of Intel MKL on the Orfeo HPC system, this study was limited to OpenBLAS (loaded via the system-provided module) and BLIS, which I compiled from source in my working directory to ensure compatibility and control over the build configuration. All experiments were executed exclusively on the THIN partition, as it was the only partition accessible for this exercise. I focused on both single-precision (`float`) and double-precision (`double`) arithmetic to observe how numerical precision impacts performance across the two libraries. The thread affinity policy was varied between two modes:

- `close`: threads are explicitly bound to specific cores to promote cache locality.

- `spread`: threads are spread across all cores to reduce contention and make the best use of cache and CPU resources.

This setup allowed for a controlled comparison of library efficiency under realistic HPC constraints, providing insight into how software implementation and runtime thread placement jointly affect GEMM performance.

Experimental Setting:

- **Library:** OpenBLAS or BLIS
- **Partition:** THIN
- **Precision:** Double (`double`) and Float (`float`)
- **Threads affinity policy:** `Close` or `Spread`
- **Fixed quantity:** Number of cores or matrix dimension

2.2 Implementation

In order to carry out this exercise, I used three main components:

- `gemm.c`: This is the core C program used to perform matrix-matrix multiplication and record performance results. It is a modified version of the provided `dgemm.c` example, extended to support both `float` and `double` precision and to optionally save timing and Gflops/s results to a CSV file. The program accepts command-line arguments to specify:
  - The BLAS library to use (`OPENBLAS` or `BLIS`),
  - The precision mode (`USE_DOUBLE` or `USE_FLOAT`),
  - Whether to enable result logging via the `SAVE_RESULTS` flag,
  - The matrix dimensions. Since I restricted all experiments to square matrices, only one dimension parameter is needed (i.e., $A, B, C \in \mathbb{R}^{N\times N}$).

  This single source file was used for both experimental tasks (varying matrix size at fixed core count, and varying core count at fixed matrix size), and it resides in the `ex2` folder of my project repository.

- `Makefile`: This file handles compilation and execution for both libraries and both precision types. It defines phony targets `float` and `double`, which automatically set the appropriate compiler flags and link against the correct library (OpenBLAS via module, BLIS via my local build).

- `sbatch` scripts: Since I only had access to the `THIN` partition, I created two separate job scripts:
  - One for the **fixed-core** experiments (strong scaling): matrix size varies while the number of OpenMP threads is held constant.
  - One for the **fixed-matrix** experiments (weak scaling): the matrix dimension is fixed, and the number of threads is increased.

  Each script requests the `THIN` partition, loads the necessary modules (for OpenBLASm BLIS), sets the OpenMP environment variables including `OMP_NUM_THREADS` and `OMP_PROC_BIND` to switch between `close` and `False` (i.e., unbound) thread affinity—and then invokes the `Makefile` to compile and run the executable across a range of parameters using a `for` loop.

2.3 Theoretical peak performance

The theoretical peak performance (`TPP`) represents the maximum floating-point throughput a processor core can theoretically achieve. It can be estimated in two different ways, depending on the information available:

1. Using processor specifications:

$$\text{TPP} = \text{cores} \times \text{frequency} \times \text{FLOPs per cycle}$$

2. Or by dividing the reported theoretical peak performance of the entire node by the number of cores:

$$\text{TPP per core} = \frac{\text{TPP}_{\text{node}}}{\text{number of cores}}$$

Since only the **THIN** partition was used for this exercise, and specific information about its processor frequency or FLOPs per cycle was not provided, I used the second approach. According to the course materials, the theoretical peak performance of an entire THIN node is reported as 1997 GFlops/s, with 24 cores per node. Thus, the theoretical peak performance per core is calculated as:

$$\text{TPP}_{\text{THIN}}^{\text{double}} = \frac{1997 \text{ GFlops/s}}{24 \text{ cores}} = 83.21 \text{ GFlops/s}$$

Assuming that single-precision arithmetic (`float`) achieves approximately twice the throughput of double-precision, the corresponding value for single precision is:

$$\text{TPP}_{\text{THIN}}^{\text{float}} = 2 \times \text{TPP}_{\text{THIN}}^{\text{double}} = 166.42 \text{ GFlops/s}$$

2.4 Results

2.4.1 Size Scalability

In this section, I examine how performance (measured in GFlops/s) scales with increasing matrix dimension N, while keeping the number of CPU cores fixed at 12 (the total number of cores available on a `THIN` node). The matrix sizes tested were:

$$\text{SIZES} = \{1000, 2000, 3000, 4000, 5000, 6000, 8000, 10000, 12000, 14000, 16000, 18000\}$$

For each matrix size, five runs were performed to ensure statistical reliability, and the median performance was used for plotting.

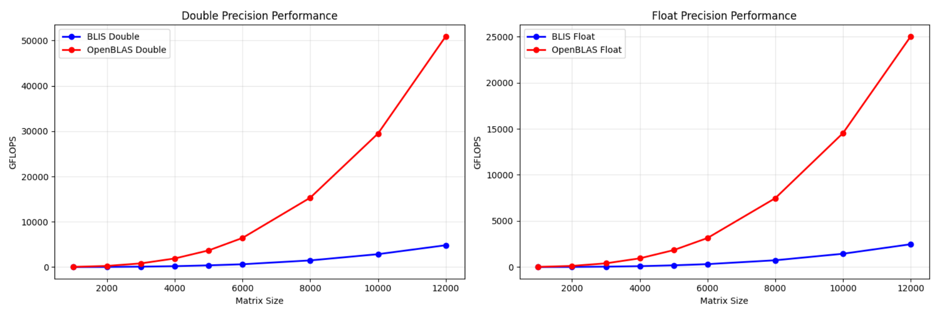

Figure 2.1: Performance (GFlops/s) vs. matrix size N under policy **close** ,comparing (`double` vs. `float`) of OpenBLAS and BLIS on the THIN partition.

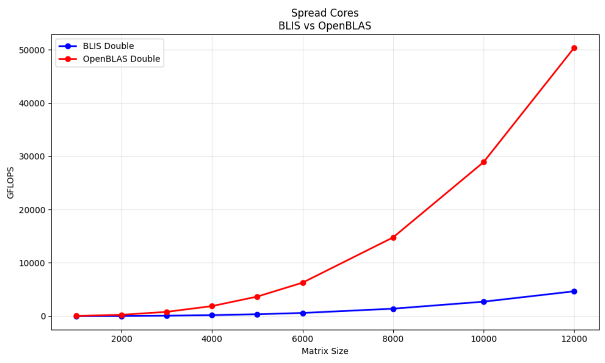

Figure 2.2: Performance (GFlops/s) vs. matrix size N under policy **Spread**, comparing (`double` vs. `Single`) of OpenBLAS and BLIS on the THIN partition.

Conclusion : Performance improves with increasing matrix size for both libraries. BLIS consistently outperforms OpenBLAS in single and double precision, with the advantage more pronounced for single precision due to better vectorization and lower memory bandwidth requirements. Both libraries reach performance saturation for $N > 12000$, indicating the 12-thread configuration approaches its computational limit.

Thread affinity has minimal impact on performance, as close and spread policies show similar trends. Overall, BLIS is better optimized for large matrix operations on the THIN node, benefiting from increased computational intensity and cache utilization at larger sizes.

2.4.2 Core Scaling

This section shows how performance (GFLOPS) changes as we increase the number of CPU cores for a fixed matrix size $N = 5000$. We tested the following thread counts:

$$\text{THREADS} = \{1, 2, 4, 6, 8, 12, 16, 24\}$$

For each case, we ran the computation five times and used the mean GFLOPS. Two thread placement strategies were tested: **close** (threads on the same cores) and **spread** (threads across different cores).

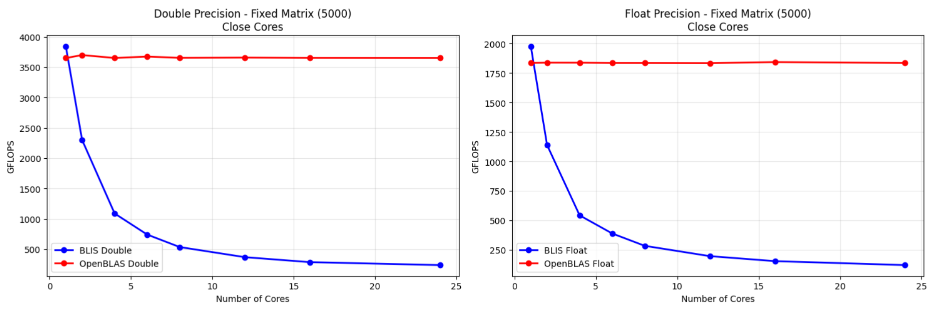

Figure 2.3: Performance (GFLOPS) vs. number of threads for $N = 5000$ under **close** affinity, comparing BLIS and OpenBLAS in single and double precision.

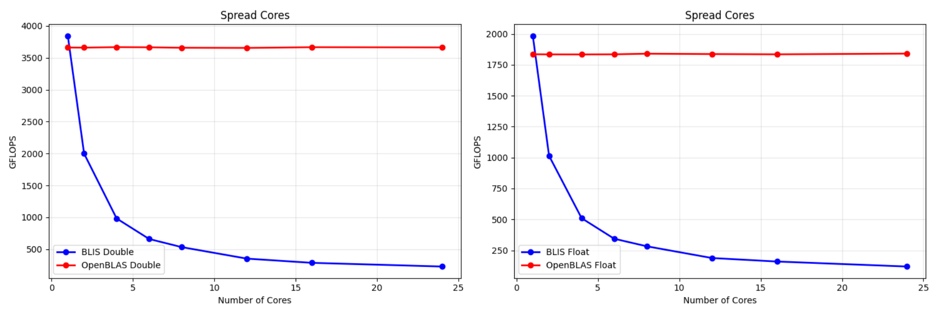

Figure 2.4: Performance (GFLOPS) vs. number of threads for $N = 5000$ under **spread** affinity, comparing BLIS and OpenBLAS in single and double precision.

Conclusion : From the performance curves, we can see that OpenBLAS maintains consistent performance across all core counts, demonstrating excellent multi-core scalability and efficient use of available computational resources. In contrast, BLIS performance drops significantly beyond 4 cores, decreasing from around 3800 GFLOPS in double precision to below 500 GFLOPS at 24 cores, likely due to cache inefficiency, memory bandwidth limitations, or lack of NUMA-aware optimizations. Thread placement (close vs spread) has minimal effect on either library, suggesting that these matrix multiplication operations are primarily CPU-bound rather than memory-bound. Interestingly, double precision achieves higher GFLOPS than single precision on this architecture, probably because of better vectorization or compiler optimizations for double precision. Overall, these results indicate that OpenBLAS is the better choice for scalable multi-core matrix computations, while BLIS would need further tuning to reach comparable performance.