Foundations of High Performance Computing - Final Project

Safa Yassin

October 7, 2025

Contents

1. Exercise 1: Game of life

1.1. Introduction

This exercise required us to implement Conway's Game of Life using a hybrid MPI+OpenMP approach. Conway's Game of Life is a cellular automaton devised by British mathematician John Horton Conway in 1970[3]. It's a zero-player game, meaning its evolution is determined by its initial state without requiring any further input.

RULES

- The playground consists of a matrix of size $x \times y$
- Each cell can be either alive (value 1) or dead (value 0)
- Any live cell with fewer than two live neighbors dies, think of it as **underpopulation**.
- Any live cell with 2 or 3 live neighbors lives on to the next generation.
- Any live cell with more than 3 live neighbors dies, think of it as **overpopulation**.
- Any dead cell with exactly 3 live neighbors becomes a live cell, think of it as by **reproduction**.

So to conclude: a cell becomes or remains alive if exactly 2 or 3 of its neighbors are alive; it dies or stays dead otherwise.
The system must be evolved for n steps, with an image saved every s steps Two evolution modes were implemented:

- **Ordered evolution:** Cells evolve sequentially in row-major order, starting from $(0, 0)$ and proceeding to $(0, 1)$, and so on. Since each cell's status depends on previously updated cells, changes take effect immediately and this method is inherently serial and not parallelizable.
- **Static evolution:** At each state s_i, the playground is frozen and all cells evolve to state s_{i+1} based exclusively on the previous state's configuration. Since cells don't affect each other during the same evolution step, this mode is fully parallelizable.

1.2. Tasks

I need to conduct three distinct scalability studies:

1. *OpenMP scalability:* using a single MPI process with a fixed matrix size, vary the number of OpenMP threads from 1 to the maximum cores available per socket (12 for THIN nodes, 64 for EPYC nodes). This measures the effectiveness of shared-memory parallelization within a single node.
2. *Strong scalability:* fix the total matrix size and increase the number of MPI processes from 1 to the maximum available sockets, fully saturating each socket with OpenMP threads. This would help us evaluate how well the algorithm accelerates a fixed problem as we add more resources.
3. *Weak scalability:* increase the number of MPI processes while keeping each process's workload constant by scaling the matrix dimensions as $10000 \times (10000 \cdot N_{\text{processes}})$. This would help us understand if proportionally larger problems can be solved in constant time with proportionally more resources.

1.3. Some Insights

In order to parallelize Conway's Game of Life we need to distribute the computational workload across multiple processes while maintaining the dependency requirements of the evolution rules (as we would need to inspect the neighbors' states!).

We employ a **domain decomposition** approach, where the playground matrix/grid is partitioned among MPI processes, with each process responsible for updating a subset of rows. This 1D row-wise decomposition was chosen for its simplicity and minimal communication overhead, as each process only needs to exchange boundary information with two neighbors (above and below).

To address boundary dependencies, we implement an **exchange** mechanism, where each process maintains additional **ghost rows** containing copies of boundary rows from adjacent processes. Before each evolution step, processes communicate via `MPI_Sendrecv` to update these ghost rows, ensuring every cell has access to its complete neighborhood (each cell inspects 8 cells in the 3x3 grid environment, so it needs to exchange rows with the process "above" and "below"). This communication represents the primary overhead in the parallel implementation—as we increase the number of processes, each handles fewer rows but the ratio of communication to computation increases, however this should scale well for large playgrounds.
Within each process's local domain, we further parallelize the computation using OpenMP threads.

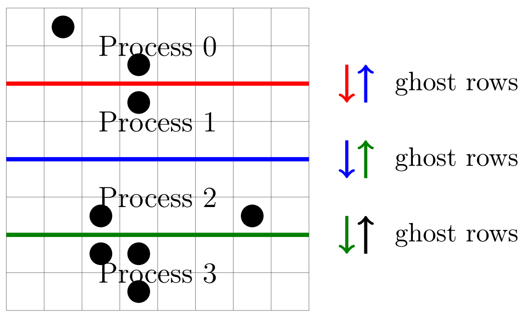


Figure 1: simplistic graph to have an intuition over row-wise domain decomposition with ghost row exchange between adjacent MPI processes (alive cells in black)

Notice that I will be connecting the boundaries of the playground with each other.

1.4. Evaluation metric

The parallel performance is evaluated using the **speedup** metric, calculated as:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{1}$$

In this equation, T_{serial} represents the wall clock execution time for the evolution when the number of tasks (which corresponds to the number of threads in OpenMP or processes in MPI) is set to 1. On the other hand, T_{parallel} denotes the execution time when the number of tasks is increased to n.
Hence the speedup metric provides a measure of the improvement in performance due to parallelization, also I expect it to vary according the evolution method and the scalability

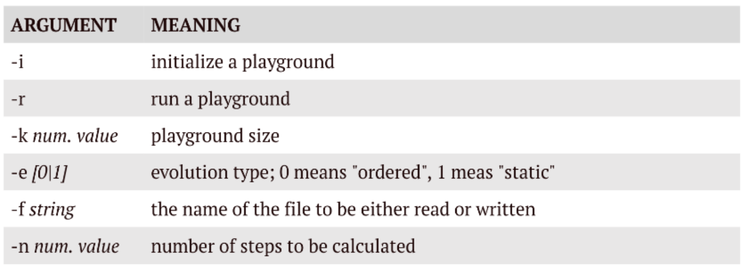
test I am conducting:
For the **ordered evolution** The theoretical speedup is $S = 1$ as it is inherently sequential, but in practice, we expect $S < 1$ due to communication and synchronization overhead.
For the **static evolution**, in the case of *OpenMP and Strong Scalability* theoretically we expect a linear speedup ($S = n$), however we know that practical speedup will be reduced by communication overhead, load imbalance, and Amdahl's Law limitations (there will be always a serial part).
For the *weak scalability* we expect constant execution time ($S = 1$), with slight degradation caused by the inter-process communication overhead.
I will be measuring and visualizing the speedup metric across the different types of scalability using available nodes on the cluster (THIN nodes), the "theoretical speedup" will be drawn as a dashed line for comparison.
I will be also mentioning the **efficiency** term during the analysis, which quantifies how effectively the parallel resources are utilized:

$$E(p) = \frac{S(p)}{p} = \frac{T_1}{p \cdot T_p} \tag{2}$$

where $S(p)$ is the speedup achieved with p processors, T_1 is the execution time on a single processor, and T_p is the execution time on p processors. Ideal efficiency is $E = 1.0$ (or 100%), indicating perfect linear scaling. Decreasing efficiency can be a good indication of how in practice we don't have the theoretical scaling we expect but rather a scaling that is affected by communication and memory contention.

1.5. Initialization and Evolution commands

Following the instruction in the document of the exercise [1] we were asked to include the following commands for easier manipulation of the initialization and evolution of the grid.



ARGUMENT	MEANING
-i	initialize a playground
-r	run a playground
-k *num. value*	playground size
-e *[0|1]*	evolution type; 0 means "ordered", 1 meas "static"
-f *string*	the name of the file to be either read or written
-n *num. value*	number of steps to be calculated

I added that, and indeed it makes it easier to initialize the grid and make some fast tests [code in appendix A.1].
Here are some execution examples:

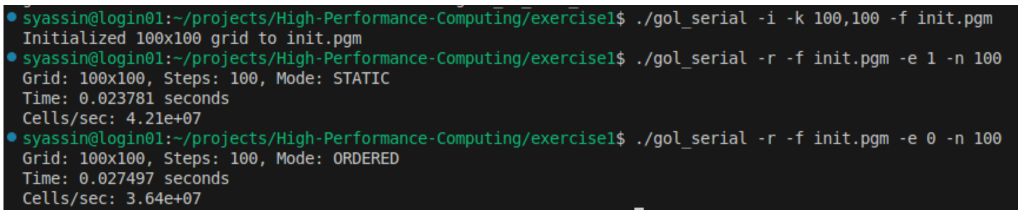
```
# Initialize grid
./gol_serial -i -k 10000,10000 -f grid.pgm

# OpenMP with 12 threads
export OMP_NUM_THREADS=12
./gol_omp -r -f grid.pgm -e 1 -n 100
```

```
# MPI with 4 processes, 12 threads each
export OMP_NUM_THREADS=12
mpirun -np 4 ./gol_mpi -r -f grid.pgm -e 1 -n 100
```

1.6. Initialization and baseline calculations

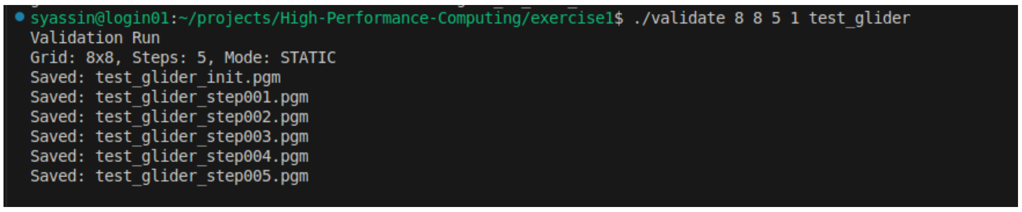
I start by creating the *game_of_life.c* code in which the baseline serial computation of the problem was implemented. the ordered and static modes were both added, I run the code to test the correctness of the implementation



```
syassin@login01:~/projects/High-Performance-Computing/exercise1$ ./gol_serial -i -k 100,100 -f init.pgm
Initialized 100x100 grid to init.pgm
syassin@login01:~/projects/High-Performance-Computing/exercise1$ ./gol_serial -r -f init.pgm -e 1 -n 100
Grid: 100x100, Steps: 100, Mode: STATIC
Time: 0.023781 seconds
Cells/sec: 4.21e+07
syassin@login01:~/projects/High-Performance-Computing/exercise1$ ./gol_serial -r -f init.pgm -e 0 -n 100
Grid: 100x100, Steps: 100, Mode: ORDERED
Time: 0.027497 seconds
Cells/sec: 3.64e+07
```

Figure 2: serial implementation using the commands on both ORDERED and STATIC modes

In addition, I opted to create a validation framework that exports grid states as PGM images and then convert them to PNG images, for example the famous glider example was visualized. This allows us to visually inspect the correctness of the code. another verification was implemented using ImageMagick's **compare** tool to detect pixel differences between generations.



```
syassin@login01:~/projects/High-Performance-Computing/exercise1$ ./validate 8 8 5 1 test_glider
Validation Run
Grid: 8x8, Steps: 5, Mode: STATIC
Saved: test_glider_init.pgm
Saved: test_glider_step001.pgm
Saved: test_glider_step002.pgm
Saved: test_glider_step003.pgm
Saved: test_glider_step004.pgm
Saved: test_glider_step005.pgm
```

Figure 3: run validate code to produce images

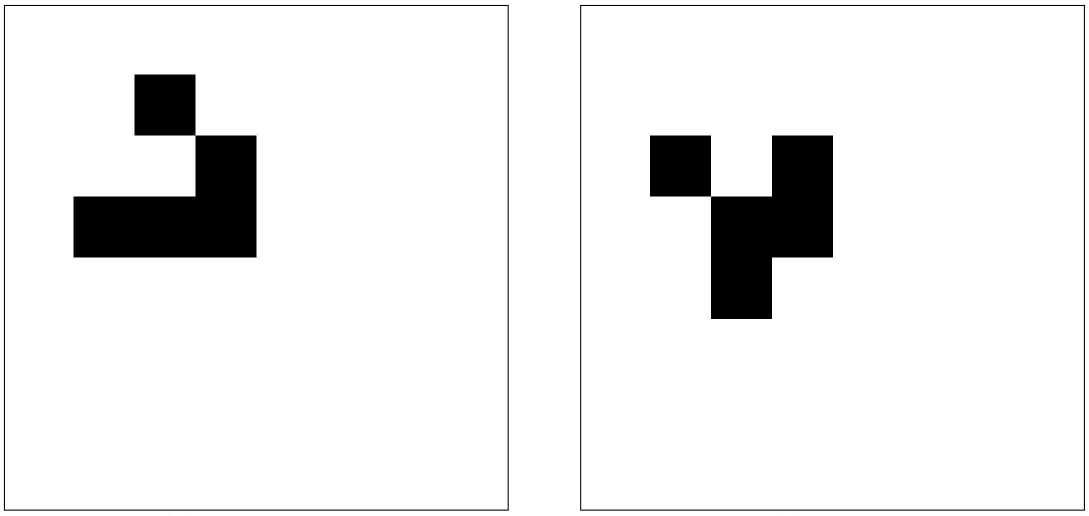

(a) Initial state (b) Step 1

Figure 4: Glider evolution

I opted to compute explicitly the serial implementation because it established the performance baseline T_serial for calculating speedup in subsequent parallelization steps, we know that the ORDERED mode will remain serial throughout the project, while STATIC mode will be parallelized using OpenMP and MPI.
In both modes I implement Conway's rules (survival on 2-3 neighbors, birth on 3 neighbors) but there is something to take in consideration, the playground is theoretically "infinite" but the way we implement it is that we connect the boundaries with each other (having in mind the topology of a torus) and hence I added the periodic boundary conditions. I note that in The STATIC mode I maintain two grids/matrices to ensure all cells evolve based on a frozen snapshot. [check Appendix A.2]

1.7. let's go parallel!

1.7.1. OpenMP scalability

In this section the idea is to inspect how does the speedup change when increasing the number of threads within a single node.

```
    --nodes=1
    --ntasks=1
    --cpus-per-task=12
    OMP_NUM_THREADS=[1,2,4,6,8,10,12]
```

notice that I used the static mode only with 50 steps

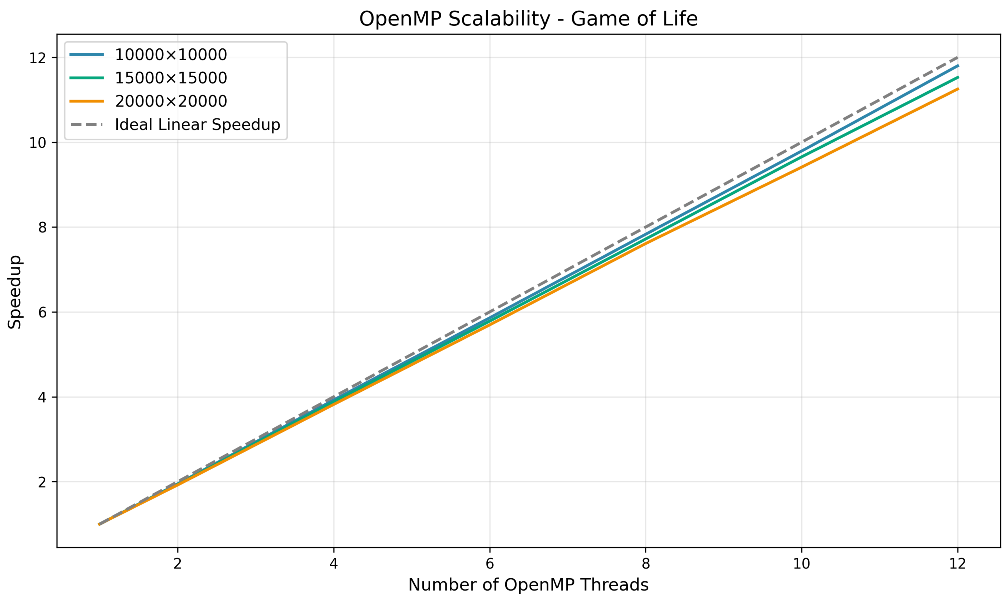


Figure 5: OpenMP scalability using THIN node with 50 steps

As seen in figure 5, the shape of the measured speed-up is almost linear and very close to the theoretical speed-up. We notice however that there is a modest degradation of scalability as the matrix size grows: 93.8% efficiency for 20000 size matrix respect to 98.3% efficiency for the 10000 size matrix. This reduction in parallel efficiency for larger matrices can be attributed to increased memory bandwidth and cache hierarchy effects.
I repeat the same analysis for the **EPYC** node

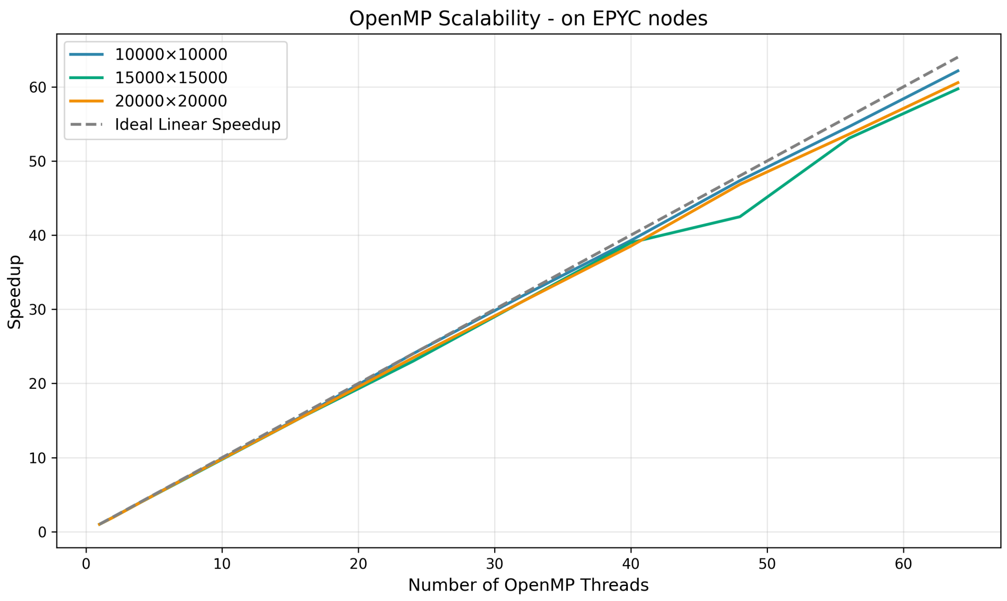


Figure 6: OpenMP scalability using EPYC node with 50 steps

the speedup is almost linear here too with some perturbation for the matrix with 15000x15000 size.

1.7.2. Strong scalability

Here I evaluate how a fixed-size problem accelerates as more computational resources are added. I keep the grid size constant with 50 evolution steps, while we test the speedup while the number of MPI processes increases. I repeated the test across growing number of matrix size.
here I opt to change the processes while keeping the $OMP\_NUM\_THREADS = 1$ meaning that is not leveraging the parallelization across threads.

```
    --nodes=2
    --ntasks=24
    --cpus-per-task=1
    OMP_NUM_THREADS=1
    MPI_processes=[1 2 4 6 8 12 16 20 24]
```

I used 2 THIN nodes with 12 cores each, distributing all 24 cores as individual MPI processes. let's take for example The 20000×20000 grid, here it undergoes decomposition, starting with each process handling between 20000 rows (1 process) down to 833 rows (24 processes). As I explained before each process exchanges only 2 ghost rows representing 0.24% of its local domain, which is a good computation-to-communication ratio.
I calculate this scalability across three different "fixed" matrix sizes.

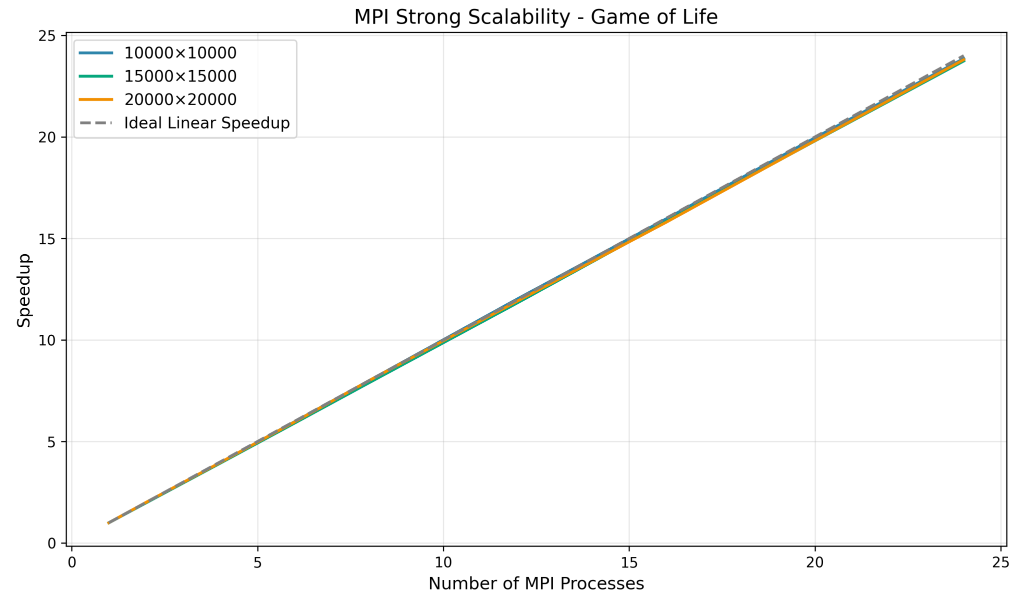


Figure 7: strong scalability with MPI only, using THIN nodes with 50 steps

Figure 7 demonstrates near-ideal strong scaling for all matrix sizes (although we do know that for larger matrix sizes the problem won't be resolved that easy). The larger problem size (20000x20000) shows slightly better scaling with 99% efficiency suggesting that as the matrix size grows the communication overhead becomes less significant relative to computation load.
I believe here that nevertheless, as the number of processes will increase further this scaling will be hindered more by the communication overhead (I could not allocate more resources to check this argument).

1.7.3. Weak scalability

Here I want to assess whether proportionally larger problems can be solved in constant time with proportionally more resources. So I keep the workload per process fixed, increase the total problem size and number of MPI processes scale together and check if the speedup stays constant. As I was limited with access to resources, I increased the processes from 1 to 4 and the grid sizes proportionally.

```
    --nodes=2
    --ntasks-per-node=2
    --cpus-per-task=6
    OMP_NUM_THREADS=6
    MPI_processes: 1, 2, 4
    Grid_sizes: 10000x10000, 10000x20000, 10000x40000
```

so here I keep the hybrid configuration of MPI and OpenMP, thus maintaining 6 OpenMP threads per MPI process across all tests. The grid dimensions scale in the column direction: 1 process handles 10000×10000 cells (100M cells), 2 processes handle 10000×20000 (200M cells total, 100M per process- same workload), and 4 processes handle 10000×40000 (400M cells total, 100M per process- same workload again), and we plot the speedup:

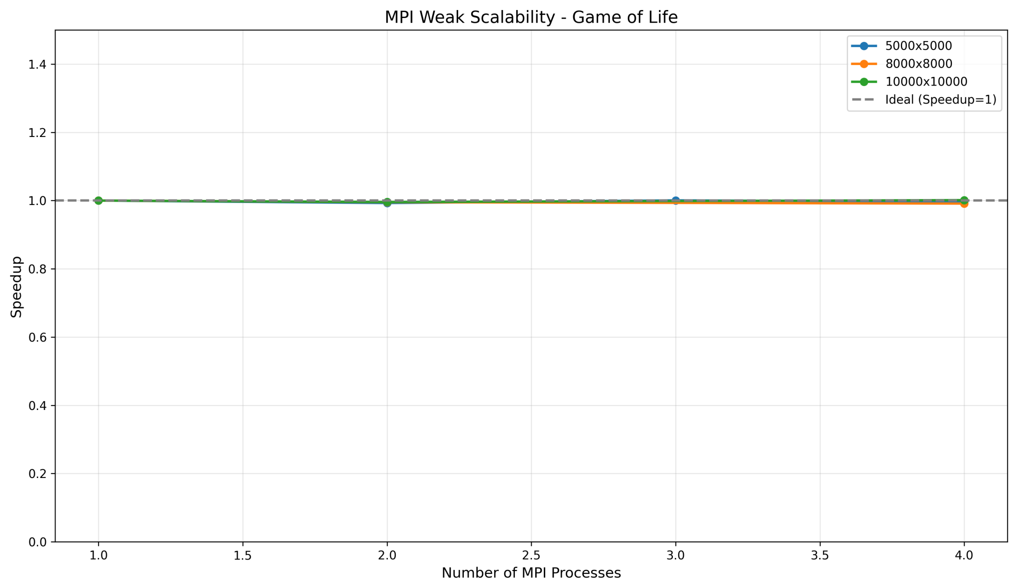


Figure 8: Weak scalability using THIN nodes with 50 steps

Figure 8 shows good weak scaling across the 4 processes, however I choose to inspect further across more processes by fixing the $OMP\_NUM\_THREADS = 1$ and run 24 processes across 2 THIN nodes (as each node has two sockets, and each socket has 12 cores), we get those results:

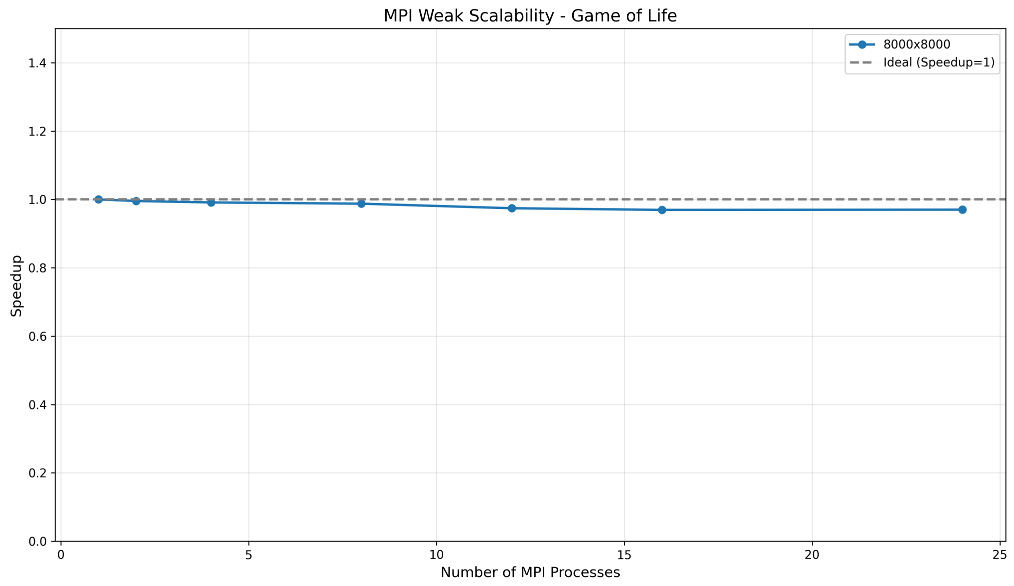


Figure 9: Weak scalability using THIN nodes with 50 steps- scaling over more processes

We can already notice that as we add more processes the speedup is not perfectly equal to 1, because although the workload is equal yet as we add more processes we pay kind of "parallel tax" which is some time and overhead cost that we notice as we scale up the system.

2. Exercise 2: Math libraries comparison

2.1. Introduction

The goal of this exercise is to evaluate and compare mathematical libraries used in high-performance computing: OpenBLAS, and BLIS. The focus is specifically on the `gemm` function (GEMM stands for General Matrix Multiplication), which is part of the level 3 BLAS (Basic Linear Algebra Subprograms) routines. I'll test both the double precision variant (`dgemm`) and single precision variant (`sgemm`) and the two different thread binding (`close`) and (`spread`) to see how each library performs and scales with increasing matrix size or increasing number of cores.

2.2. Tasks

In order to test the scalability over the number of cores and matrix size I consider two situations in which I add to the different combinations of the configuration parameters:

- **First test: Fixed cores** The number of CPU cores is kept fixed (64 for Epyc and 12 for Thin) while the matrix size is varied (from 200 to 2000 with steps of 500). This reveals how performance changes as the matrix size grows.
- **Second test: Fixed matrix** The approach is reversed: the matrix size is kept constant (1000) while the number of cores is scaled. Both node types are tested using 1, 2, 4, 6, etc. cores (step of 2), up to 128 for Epyc nodes and 24 for Thin nodes. This gives us insight on how different libraries take advantage of additional processing power.

2.3. Configuration Parameters

Every time I run the test, I would need to pick:

- Which **library** to use (OpenBLAS, or BLIS)
- Which **type of hardware** (THIN or EPYC partition)
- What **precision level** (single or double)
- How **threads are distributed** (spread out across cores or kept close together)
- What we're keeping **fixed** (number of cores or matrix size)

2.4. Definition of the problem

GEMM (General Matrix Multiplication) operation computes this fundamental linear algebra operation:

$$C = \alpha \cdot A \times B + \beta \cdot C$$

where $A \in \mathbb{R}^{m\times k}$, $B \in \mathbb{R}^{k\times n}$, and $C \in \mathbb{R}^{m\times n}$ are matrices, while α and β are scalar coefficients.
In the following analysis I will take all matrices to be square of size n, and $\alpha = 1$ and $\beta = 0$.
The code *gemm.c* was provided in which 3 matrices A, B, C are allocated, A and B are filled and the BLAS routine calculates the matrix-matrix product C=A*B
I note that the provided code already utilizes conditional compilation directives to support computations in both single and double precision. By defining USE FLOAT or USE DOUBLE flags.
With respect to **Parallelization**, it has been implemented in the code using OpenMP. By inserting `#pragma omp for` directives within the key computational loops, following the creation of a parallel region with `#pragma omp parallel`. Thus, parallelization was exclusively utilized through OpenMP threading across the available cores.
Hence in this part I will be focusing on **thread-level parallelism in shared-memory context** and scrutinize how each library uses multiple cores and memory, disregarding the communication overhead in distributed memory context.

2.5. Architecture and Peak Performance

The benchmark was performed on two node types from the ORFEO Data Center [2]: THIN (Intel Xeon Gold 6126) and EPYC (AMD EPYC 7H12). A node's Theoretical Peak Performance (TPP) is calculated as:

$$\text{FLOPS} = \text{cores} \times \frac{\text{cycles}}{\text{second}} \times \frac{\text{FLOPs}}{\text{cycle}} \tag{3}$$

The TPP for single (FP32) and double (FP64) precision for a single node of each type is summarized below. For FP32, the peak performance is double that of FP64, as the hardware can process twice the number of 32-bit operations per cycle.

	THIN	**EPYC**
single (FP32)	1996.8 GFLOPS	5324.8 GFLOPS
double (FP64)	998.4 GFLOPS	2662.4 GFLOPS

The THIN node's performance is based on its documented peak of 1.997 TFLOPS for FP64. The EPYC node's performance is derived from its architecture: 128 cores at 2.6 GHz, each capable of 16 FP64 operations per cycle (128×2.6 GHz$\times16 = 5.3248$ TFLOPS).

2.5.1. Thread binding

OpenMP provides two primary strategies: **close** (compact) and **spread** (scattered) distribution. The `close` strategy places threads on adjacent cores, typically within the same processor socket or NUMA node, optimizing for data locality and shared cache utilization. Conversely, the `spread` strategy distributes threads evenly across all available cores and sockets, maximizing aggregate memory bandwidth and reducing resource contention. The variable *OMP_PLACES* defines a series of places to which the threads are assigned, and *OMP_PROC_BIND* describes how threads are tied to those places.

2.5.2. libraries description

- **OpenBLAS:** An open-source implementation of the Basic Linear Algebra Subprograms (BLAS) library, optimized for high performance on a wide variety of CPU architectures.
- **BLIS (BLAS-like Library Instantiation Software Framework):** A modern framework for instantiating BLAS-compatible libraries. Its design offers greater flexibility and adaptability than traditional BLAS implementations, which can result in better performance across diverse systems and problem sizes. Being a newer project, it may sometimes lack the specific optimizations found in more established libraries like OpenBLAS or MKL.

2.6. Implementation

I follow the exercise documentation for the implementation of the comparison of the different libraries across different conditions
Set up: The Blis library was set up following the instructions here in which 128 cores were used to compile the file in a parallel way, and save the artifact in a local path. I load the openBlas library as well and include it later in the sbatch file.

```
### OpenBLAS library
module load openBLAS/0.3.29-omp
### BLIS library
BLISROOT=/u/dssc/syassin/myblis
```

I test scalability over matrix size and core number by preparing three different types of code that help us set the variables in a trackable way (although in the exercise description it was advised to use the makefile, however I preferred to practice better the use of shell files and the allocation of resources via sbatch file:

- **gemm.c**: A slightly modified version of the dgemm.c code that was provided, in which the matrix multiplication is implemented; I deleted the parts involving the MKL libraries as the module was not available on the cluster, I added the possibility to save the results in .csv files and some error handling messages.
- **Shell scripts: benchmark.sh**: I created several bash scripts in which I implement the benchmarking process by systematically iterating through various configuration parameters (we have four files, two for fixed cores (THIN and EPYC) and for fixed

matrix (THIN and EPYC). These scripts, I load the appropriate library modules (or set the installation path for the Blis library), I compile gemm.c with the correct compiler flags for different libraries and precision types, I set OpenMP environment variables for the parallelization part, and lastly execute benchmarks across the different conditions we sat before (spread, closed, float or double).

- **SBATCH submission scripts**: SLURM job submission scripts that allocate the necessary cluster resources and execute the corresponding benchmark shell scripts.

2.7. Matrix size scalability: fixing the cores

for the first task in which I keep the number of cores fixed (12 for THIN and 64 for EPYC) and scale over the matrix size, I get the following results:

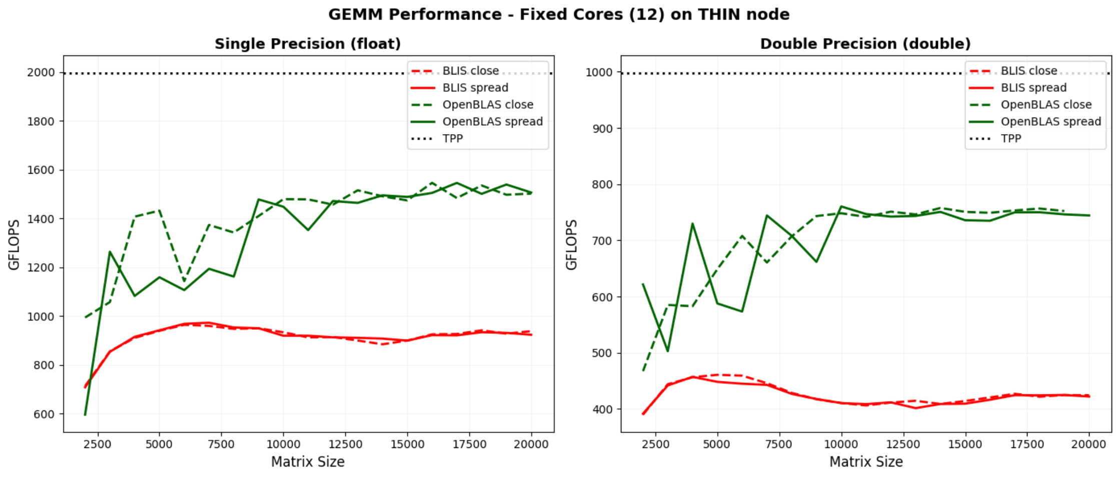


Figure 10: matrix scalability using openBLAS and BLIS across different conditions on THIN node

There is a noticeable difference between the libraries with the OpenBLAS performing better, since the GMM problem requires a fixed number of operations, approximately $2n^3$ FLOPs, more Gflops indicate that the library completes the operation **faster** hence we assume here that openBLAS optimizes better the use of cpus, the victorization and the memory bandwidth for this task.
Another thing we notice is how the Gflops increase as the matrix size grows to then arrive to a plateau, and the trend is the same for both float and double precision, thread binding and across libraries. What I assume is that for small matrices memory allocation and overhead dominates over the actual "benefit" of parallelization and so for medium/large enough size matrices, the cache hierarchy is better utilized and the overhead becomes negligible to the actual work and it shows the potential of the hardware- getting closer to the TPP Notably, achieving $60\% - 70\%$ of TPP is considered excellent performance, as TPP represents an idealized scenario that assumes perfect conditions (zero cache misses, infinite memory bandwidth, 100% core utilization) that cannot be achieved in practice.) It is important to note that for larger and larger matrices we might see a drop in the Gflops as we will be facing memory problems related to cache misses.

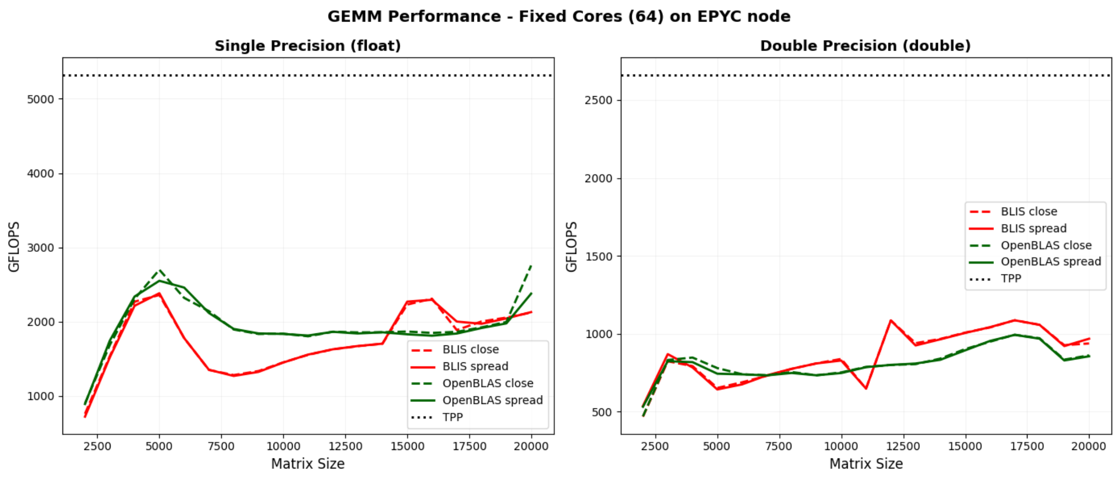


Figure 11: matrix scalability using opebBLAS and BLIS across different conditions on EPYC node

Here the EPYC node has a different scaling behavior compared to THIN, as it achieves lower efficiency relative to its TPP despite significantly higher absolute performance (higher Gflops for both float and double). I assume what happens here is related to the complexity of the EPYC node respect to the THIN node, it has 128 cores (two sockets with 64 cores each), and has 8 NUMA domains, which we should take in consideration: In fact if we observe the code provided dgemm.c, I note that all matrices were initialized sequentially on the main thread before the parallel computation begins. This might have causes that all memory pages are allocated on the NUMA domain of the initializing thread. and what I guess happened here is that when OpenMP threads subsequently execute across multiple NUMA domains during the GEMM computation, most threads must access remote memory, which causes some latency penalties. In addition we notice there is a sharp drop in Gflops, especially for the float point precision around $n = 5000$-7000 matrix size, this is likely cthe point where the working set size exceeds the aggregate L3 cache capacity and forces sustained remote memory access across NUMA boundaries, taking much more time to execute the operations.
My guess for the sharper drop when it comes to float respect to double is that float should be x2 faster computationally, so it becomes more memory bound When NUMA imbalance forces threads to access remote memory, but generally they both precision points exhibit similar behavior.
Just to test further my observations I repeat the calculations over the EPYC node with different initialization (check A.4) and I get these results

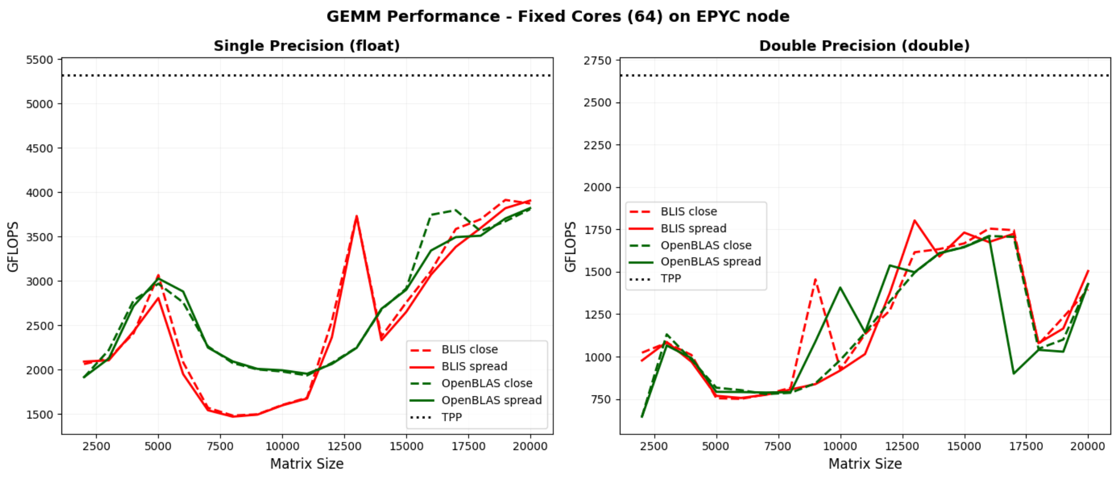


Figure 12: matrix scalability on EPYC node- using NUMA aware initialization

We can see a similar trend for the EPYC node (due to the architecture) but we can notice how we get higher GFlops than before, meaning that here is wasting less time reaching remote memory, it is interesting to scrutinize further the weird behavior at some matrix size ranges.

2.8. Number of cores scalability: fixing the matrix size

for the second task in which I increase the number of cores while keeping the size of the matrix fixed I get the following results:

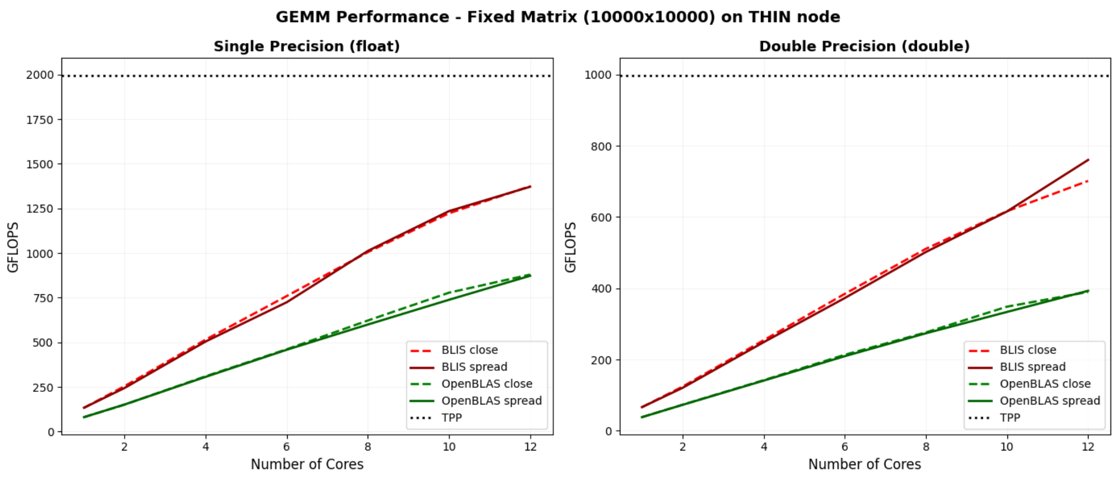


Figure 13: Enter Caption

For this part, since we are increasing the number of cores, the "ideal" scaling would be a **linear** one!, in fact with the thin nodes we get a similar trend, with BLIS library performing better. we expect also that this linear scaling is affected by the number of cores, because as the number of cores increase the law of Amdahl kicks in, this is visible in the EPYC node:

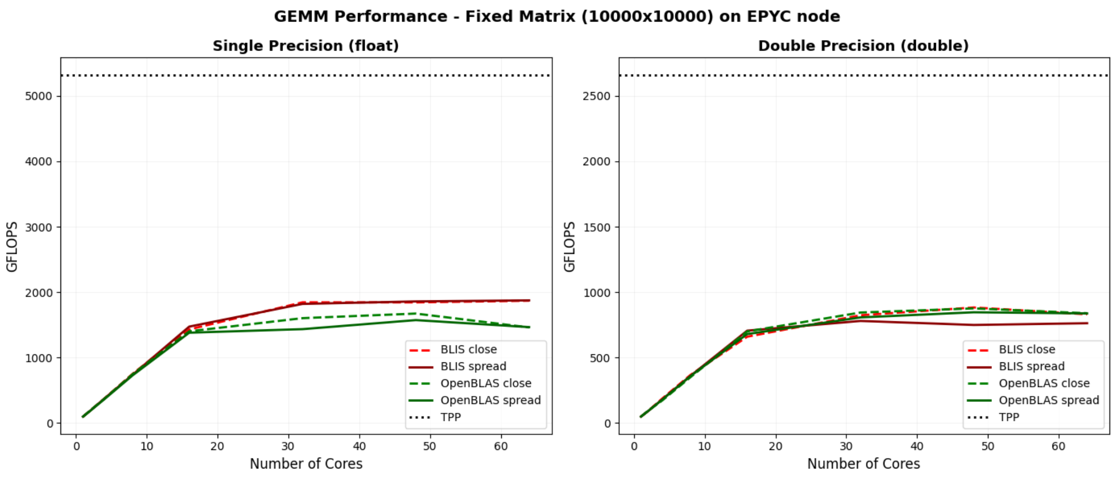


Figure 14: Enter Caption

here we see the linear trend when we have few cores, but once we increase the number of the cores the cross-socket communication slows things down.

2.9. Conclusions:

Both libraries, OpenBLAS and BLIS, are optimized in different ways to handle GEMM operations. OpenBLAS performed generally better in the first task with fixed cores and varying matrix size, while BLIS showed better performance in the second task with fixed matrix and increasing number of cores. The THIN node shows a more expected trend and is easier to interpret, as the node architecture is simpler with only 2 NUMA domains. The EPYC node architecture, on the other hand, exhibits the challenges we face with memory handling when dealing with more cores distributed across 8 NUMA domains. These results highlight the awareness we need to have regarding scalability when using large architectures with complex memory topologies, demonstrating that even to study the performance of a well-studied operations like matrix multiplication, we need to carefully consider the underlying hardware characteristics and memory subsystem limitations.

A. Code snippets

A.1. command-line parsing

```c
while ((opt = getopt(argc, argv, "irk:e:f:n:")) != -1) {
    switch (opt) {
    case 'i': init_mode = 1; break;
    case 'r': init_mode = 0; break;
    case 'k':
        if (sscanf(optarg, "%d,%d", &rows, &cols) == 1) {
            cols = rows;
        } else if (sscanf(optarg, "%d,%d", &rows, &cols) != 2) {
            if (rank == 0) fprintf(stderr, "Error:␣Invalid␣size\n");
            MPI_Finalize();
            return 1;
        }
        break;
    case 'e': evolution = atoi(optarg); break;
    case 'f': filename = optarg; break;
    case 'n': steps = atoi(optarg); break;
    default:
        if (rank == 0) print_usage(argv[0]);
        MPI_Finalize();
        return 1;
    }
}
```

A.2. Serial baseline

The code of the implementation of the static mode in which the function reads from the current grid and writes updates to a temporary grid based on the **survival rules**. Creating a temporary grid ensures all cells update simultaneously from the same generation.

```c
void evolve_static(Grid *grid, int steps) {
    Grid *temp = create_grid(grid->rows, grid->cols);

    for (int step = 0; step < steps; step++) {
        for (int i = 0; i < grid->rows; i++) {
            for (int j = 0; j < grid->cols; j++) {
                int neighbors = count_neighbors(grid, i, j);
                int cell = get_cell(grid, i, j);

                if (cell == 1) {
                    set_cell(temp, i, j,
                        (neighbors == 2 || neighbors == 3) ? 1 : 0);
                } else {
                    set_cell(temp, i, j, (neighbors == 3) ? 1 : 0);
                }
            }
        }

        int *swap = grid->data;
        grid->data = temp->data;
        temp->data = swap;
    }

    free_grid(temp);
}
```

A.3. OpenMP Parallelization

The OpenMP version parallelizes the STATIC mode by distributing cell updates across threads. I use the `collapse(2)` clause to merge the nested loops for better load balancing:

```c
void evolve_static_omp(Grid *grid, int steps) {
    Grid *temp = create_grid(grid->rows, grid->cols);

    for (int step = 0; step < steps; step++) {
        #pragma omp parallel for collapse(2) schedule(static)
        for (int i = 0; i < grid->rows; i++) {
            for (int j = 0; j < grid->cols; j++) {
                int neighbors = count_neighbors(grid, i, j);
                int cell = get_cell(grid, i, j);

                if (cell == 1) {
                    set_cell(temp, i, j,
                        (neighbors == 2 || neighbors == 3) ? 1 : 0);
                } else {
                    set_cell(temp, i, j, (neighbors == 3) ? 1 : 0);
                }
            }
        }

        int *swap = grid->data;
        grid->data = temp->data;
        temp->data = swap;
    }

    free_grid(temp);
}
```

Thread count is controlled via the `OMP_NUM_THREADS` environment variable. The ORDERED mode remains serial as it is inherently non-parallelizable.

A.4. NUMA aware initialization

```c
// NUMA-aware parallel initialization
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < (m*k); i++) {
        A[i] = (MYFLOAT)(i+1);
    }

    #pragma omp for
    for (int i = 0; i < (k*n); i++) {
        B[i] = (MYFLOAT)(-i-1);
    }

    #pragma omp for
    for (int i = 0; i < (m*n); i++) {
        C[i] = 0.0;
    }
}
```

References

[1] Foundations of HPC Course. Assignment exercise 1. https://github.com/Foundations-of-HPC/Foundations_of_HPC_2022/blob/main/Assignment/exercise1/Assignment_exercise1.pdf, 2022. Accessed: October 7, 2025.

[2] ORFEO Data Center. Orfeo computational resources. https://orfeo-doc.areasciencepark.it/HPC/computational-resources/, 2024. Accessed: October 7, 2025.

[3] Wikipedia contributors. Conway's game of life. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life, 2024. Accessed: October 7, 2025.