FHPC Assignment Report

Linda Broglio, Matteo Fadelli, Nicola Perin

February 16, 2023

Contents

1 Exercise 1

1.1 Introduction

Conway's Game of Life[1] is a cellular automaton that is played on a 2D square grid. Each square (or "cell") on the grid can be either alive or dead, and they evolve according to the following rules:

- Any live cell with fewer than two live neighbours dies (underpopulation).
- Any live cell with more than three live neighbours dies (overpopulation).
- Any live cell with two or three live neighbours lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbours comes to life.

The initial configuration of cells can be created by a human, but all generations thereafter are completely determined by the above rules. The goal of the game is to find patterns that evolve in interesting ways – something that people have now been doing for over 50 years.
We note that, since the grid size is finite and periodic boundary conditions have been imposed, the cells can be represented on the surface of a torus. The behaviour of some patterns therefore deviates in the long term from the behaviour in an infinite grid in two dimensions, usually the standard in the literature.

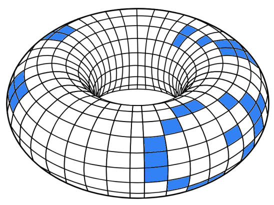


Figure 1: The initial configuration of a Gosper glider gun in a finite grid with PBC. With this topology, the pattern does not exhibit an unbounded growth.

In this first section of the report, a parallel **C++** implementation of Game Of Life using both **OpenMP** and **MPI** is presented. The classic evolution is what has been referred to as *static* in the assignment. In this evolution each cell gets updated according to the rules only when all the others have checked their own status.
As per requested, the *ordered* variation has also been implemented, in which each cell is updated right after checking its neighbours, without waiting for the other cells to check their own status.
In the rest of this section, after illustrating the approach used to tackle the problem from an algorithmic point of view, some coding details and the choices made are explained. At last, results are presented and discussed and some possible improvements to the program are proposed.
Scalability analysis was conducted using nodes from the ORFEO cluster, more specifically the EPYC (AMD) nodes were used.

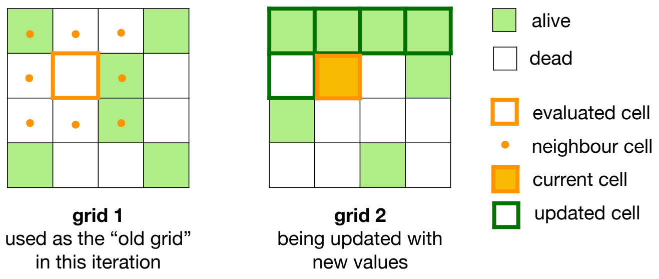
1.2 Methodology

In our work we focused on the *static* evolution mode. It seems to us that the *ordered* evolution mode is intrinsically serial and no parallelisation can be implemented about computation. As a first step, we implemented a working serial version of Game of Life. Next, we parallelised certain regions of the code via openMP. Finally, we made the program executable on systems with distributed memory using OpenMPI libraries for message passing between processes.
For hybridization we opted for simple openMP parallel regions inside each MPI task: message passing is used for passing data between different nodes, while the computation of each part of

[1] https://conwaylife.com/

the grid is carried out under shared memory.
It is possible to distinguish three phases in the structure of a computer program that wants to implement Game Of Life: the **initialisation** of the grid, the actual **computations** on the grid and the **printing** of the grid. The first process is carried out once per run of the program, while the number of runs for the second and third phases depends on the user's demand (based on the command line arguments `-n` and `-s`).
We point out that the focus and the time to be spent on optimising the performance of each of these parts depends on the intended use of the program. There are many parameters in this respect, including the size of the matrix, the number of iterations and the frequency of printing. For example in this sense we have assumed that the purpose is not to study the eventual behaviour of very complex patterns requiring hundreds of thousands of iterations, a problem that seems to be generally addressed by the use of pattern databases [2].

- As far as the initialisation of the grid is concerned, our program has two initialisation modes: one that internally generates a random grid of a fixed size, and one that generates the grid from a human-readable format file inserted inside an empty grid of a fixed size. The grid is then printed as an image and, when it will be read, stored in a one-dimensional array-like data structure.
- The actual calculations consist in looping through the grid and, for each cell, calculating the number of living neighbours, taking into account boundary conditions. This number determines the evolution of the cell. For the static evolution mode, the two steps - the calculation of the neighbours of a given cell and the actual update of the cell - are each performed on a different grid. It is possible to avoid copying the grid at the end of each complete grid update by alternating the roles for these two tasks: the grid that at the first iteration was used to calculate the number of living neighbours at the next iteration will be the one containing the new cell values.

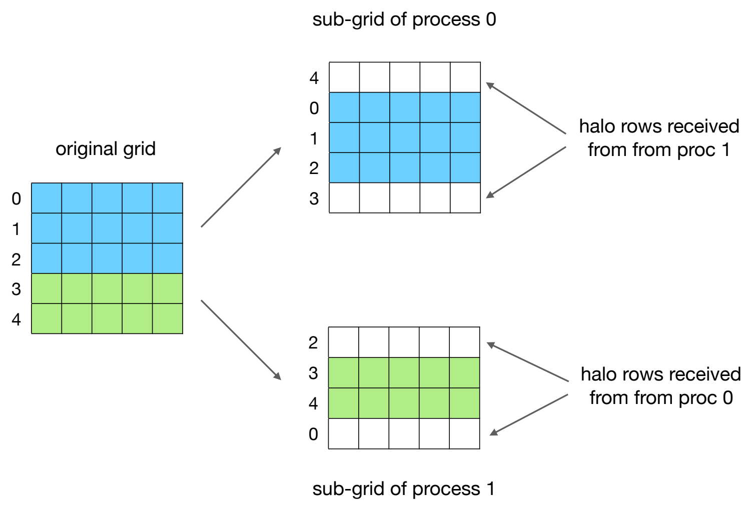


In the next *tick* grid 2 will be the "old" grid and grid 1 will take in the new values.

- Printing the grid to file uses the pbm file format in the binary (raw) version. To do this, groups of 8 matrix elements are converted into characters and printed on file, taking into account the complications associated with the end of rows. This image format makes it possible to map 1 bit to 1 pixel, thus achieving rather small image sizes.

Our idea for MPI parallelisation is to divide the grid in smaller subgrids of roughly the same size (through horizontal cuts if we store matrices in row-major order), so as to get a number of subgrids that is equal to the number of processes used. In order for the calculations to proceed in parallel, with a subgrid assigned to each process, it is necessary for each of these subgrids to possess *halos*. The *halos* are in this case additional rows, one above and one below the subgrids, which allow for the actual subgrids to be updated. To maintain the correctness of the algorithm, they must be replaced at the end of each iteration with data from other subgrids. For this reason, an exchange of data between each subgrid and its neighbouring subgrids (above and below) is necessary at the end of each iteration.
Pictures probably make these processes clearer:

[2] See for example https://golly.sourceforge.net


sub-grid of process 0
original grid
halo rows received from from proc 1
halo rows received from from proc 0
sub-grid of process 1

In order to print the entire grid, each process sends its subgrid, without the haloes, to a selected process that is deputed to write to file.
With openMP it is possible to parallelise the computations on the grid only in the case of the static evolution mode: a domain decomposition can be performed on the loop that updates the cells of the grid. In the initialisation phase it's also possible to parallelise a part of the reading of the grid. More information on the implementation of openMP can be found in the next subsection. Some ideas about possible functional decompositions are briefly discussed in the conclusions but were not implemented.

1.3 Implementation

The program was written in the C++ language because we are more familiar with it and it should not perform worse than C. Furthermore C++ offers convenient containers such as `std::vector` for storing data and avoiding pointer arithmetic operations.
This part of the report contains some detail about the actual code and includes some snippets. For the complete commented version one can consult the github repository.

1.3.1 Initialisation

We implemented two ways of initialising the grid: `random` and `file`. In random initialisation mode the `init_grid` function fills an already allocated `std::vector` of size equal to the square of the parameter entered with `-k` with equiprobable random binary values. In file initialisation mode, it is necessary to provide via the `-f` argument an image in pbm format with magic number P1 (therefore human readable). The `loadMatrix` function opens the file and writes the pattern it contains to the approximate centre of a square grid of side k. In both cases, the resulting vector of size k*k is subsequently written to file via the `createSnapshot` function. An image called `snapshot_00000` is thus generated in raw pbm format with magic number P4. `createSnapshot` calls the `createCharacterFromBits` function for groups of 8 binary values which are converted into the corresponding char. A number of module operations take care of the values at the end of each row in the grid that must be artificially expanded to correctly produce the image when k is not a multiple of 8. Raw pbm files are a marked improvement over P1 ASCII pbm as the images occupy much less space in the disk, at the price of no longer being human readable.

```
// From a grid (stored in arr) print a pbm P4 image
void createSnapshot(const std::vector<int>& arr, const int size, const int step) {
  int resto = size%8; // number of significant bits for the last character in a row
  int quot = size/8; // number of characters for a row

  // Open a binary file called "snapshot_s" in write mode as snapshot
  //...

  snapshot << "P4\n" << size << ' ' << size <<"\n"; // header of a pbm file with magic number 4
  int i = 0;
  while (i < size*size) {
    int i_mod = i%size; // index of the column
    if (i_mod < quot*8) { // numbers not at the end of the row
      std::array<int, 8> beet = {arr[i],arr[i+1],arr[i+2],arr[i+3],arr[i+4],
      arr[i+5],arr[i+6],arr[i+7]};
      char c = createCharacterFromBits(beet); // transform 8 bits into a character
      snapshot << c; // write the char to file
      i = i + 8; // go to the next group of 8 numbers
    }
    // if we are at the end of a row we should expand artificially the group to 8
    else { std::array<int, 8> boot = {0,0,0,0,0,0,0,0}; // initialise to 0 the array
      for (int j = 0; j < resto; j++) { // write only the signficant part
        if (j < resto) boot[j] = arr[i+j];
      }
      char c = createCharacterFromBits(boot);
      snapshot << c;
      i = i + size - (i%size); //goes to the first element of the next row
    }
  }
}

// Convert an array of 8 zeros and ones into the corresponding character
char createCharacterFromBits(const std::array<int, 8>& bits) {
    char result = 0;
    for (int i = 0; i < 8; i++) {
    // shift the value of result to the left by 1 bit (using the << operator) and then
    // perform a logical OR operation (using the | operator) with the value of the current bit
        result = (result << 1) | bits[i];
    }
    return result;
}
```

1.3.2 Reading the snapshot

When the program is called with the `-r` argument, first an `std::vector<int> grid(k*k)` is allocated. The program then reads the initial grid from a file, which by default is named `snapshot_00000.pbm` which is an image in P4 format (like those generated by the program runned with the `-i` argument). The reading of the image consists of two steps: in the first step, the `readSnapshot` function opens the file and sequentially copies the values from the file into a previously initialised vector of characters of the right size. The second step is performed by the `convertChars` function, which converts the contents of the vector of characters to integers via the `createBitsFromCharacter` function and writes them into the grid vector. This process is parallelised via openMP with a parallel for. As in the case of the `createSnapshot` function, a series of module operations deals with the values at the end of a row in order to reconstruct the grid correctly.

```
// Convert a vector of characters into a vector of integers (in fact a grid)
void convertchars(std::vector<char>& char_file, const int size, std::vector<int>& grid){
  int len = char_file.size(); // number of characters in the vector
  int k = size % 8; // number of significant bits of the last character of a row
  int n = (size/8) + 1; // number of characters that codify a single row of the grid
  #pragma omp parallel for
  for(int i = 0 ; i < len; i++) {
    char c = char_file[i];
    int rw = i/n; // number of rows already written
    int cl = i%n; // number of characters already converted in the current row
    if (k == 0) { // special case in which k is a multiple of 8 and so all bits are significant
      std::array<int, 8> bits = createBitsFromCharacter(c); // obtain the array in 0s and 1s
      std::copy(bits.begin(), bits.end(), grid.begin() + i*8); // write them in the grid
    } else { // k is not a multiple of 8
      if ((i+1) % n == 0 ) { // at the end of a row, write only the first k bits
      std::array<int, 8> bits = createBitsFromCharacter(c);
      std::copy(bits.begin(), bits.begin() + k, grid.begin() + rw*size + 8*cl);
      } else { // not at the end, write all the array in the grid
        std::array<int, 8> bits = createBitsFromCharacter(c);
        std::copy(bits.begin(), bits.end(), grid.begin() + rw*size + 8*cl);
      }
    }
  }
}

// Convert a character into is binary representation and write it in an array of size 8
std::array<int, 8> createBitsFromCharacter(char c) {
  std::array<int, 8> bits{};
  for (int i = 0; i < 8; i++) {
    bits[i] = (c & (1 << (7 - i))) != 0;
  }
  return bits;
}
```

1.3.3 MPI division of the grid

All the previous steps are carried out by just one MPI process (the one with rank 0). Once the grid on process 0 is obtained, a number of calculations are performed to determine how to horizontally section the grid. We have ensured that the sizes of the sections to be sent to the MPI processes are as balanced as possible and that they only contain complete rows. The grid sections are sent via a `Scatterv` collective MPI communication and are received by the processes in already allocated `subgrid1` vectors. At a later stage, halo rows are also sent via a non-blocking MPI communication of type `Isend`. At the end of these operations, each process has its grid section plus the two halo rows in the subgrid1 vector. A second vector, `subgrid2`, of the same size as subgrid1, is initialised for each MPI process.

```
// Initialise arrays for storing indexes to divide the grid
  int main(..){
      //...
      int* sendcounts = new int[world_size];
      int* sendcounts_elem = new int[world_size];
      int* displs = new int[world_size];

      how_many_rows = k / world_size;
      rest = k - (world_size * how_many_rows);

      // determine how many rows should each process receive
      for (int i = 0; i < world_size; i++) {
        if (i < rest) {
            sendcounts[i] = how_many_rows + 1;
        } else {
            sendcounts[i] = how_many_rows;
            }
      }

      // determine the row indeces
      displs[0] = 0;
      for (int i = 0; i < world_size-1; i++) {
          displs[i+1] = displs[i] + sendcounts[i];
      }

      // determine how many elements should each process receive and their grid starting point
      // multiply by the number of elements in a row (cols)
      for (int i = 0; i < world_size; i++) {
          sendcounts_elem[i] = sendcounts[i]*k;
          displs[i] = displs[i]*k;
      }
     }
    //...
    MPI_Scatterv(grid.data(), sendcounts_elem, displs, MPI_INT, &subgrid1[k],
    sendcounts_elem[world_rank], MPI_INT, 0, MPI_COMM_WORLD);
    //...
}
```

1.3.4 Grid Update

Each MPI process contains two `std::vectors<int>`: subgrid1 and subgrid2. They take on the roles of 'old grid' and 'new grid' in turn. The alternation is carried out by an `if` that differentiates between even and odd iterations. The difference lies in which of the two subgrids plays the role of the old grid and which plays the role of the new grid. Since at iteration 1 the subgrid containing the original grid section is subgrid1, it follows that it will play the role of old grid for all odd iterations. In both cases, the `static_update_grid` function is called. This function contains a for loop on the cells of the new grid. In the loop, two operations are performed for each cell: the `check_neighbours` function calculates how many alive neighbours the same cell in the old grid has, while the rest of the loop contains a switch that updates the current cell in the new grid according to the value returned by `check_neighbours`. This for loop is parallelised using an `openMP for collapse`.

```
// Function that, given a grid, its size and the indexes of a cell
calculates the number of its alive neighbour
int check_neighbours(const std::vector<int>& grid, const int x,
const int y, const int rows, const int cols) {
  int count = 0;
  #pragma omp simd collapse(2)
  for (int dy = -1; dy <= 1; dy++) {
    for (int dx = -1; dx <= 1; dx++) {
      if (dx == 0 && dy == 0) continue;  // skip the current cell

      // handle wrapping
      int nx = x + dx;
      // boundary conditions on the rows
      if (nx < 0) { nx += rows; } else if (nx >= rows) { nx -= rows; }

      int ny = y + dy;
      // boundary conditions on the columns
      if (ny < 0) { ny += cols; } else if (ny >= cols) { ny -= cols; }

      if (grid[nx*cols + ny]) count++; // nx*cols + ny index of the cell in a one dimensional array
    }
  }
  return count; // number of alive neighbours
}

// Function that upgrades a grid in the static evolution mode
// old_grid is used for evaluation, grid will contain the updated values
void static_update_grid(const int rows, const int cols, std::vector<int>& grid,
                        std::vector<int>& old_grid) {
    #pragma omp parallel shared(grid, old_grid)
    {
      #pragma omp for schedule(static) collapse(2)
        for (int i = 1; i < rows - 1; i++) { // to avoid updating halo rows
          for (int j = 0; j < cols; j++) {
            int alive = check_neighbours(old_grid, i, j, rows, cols); // evaluated on the old_grid
            int index = i*cols + j; // index of the cell
            switch(alive) {
              case 2:
                grid[index] = old_grid[index]; break; // the value is the old value
              case 3:
                grid[index] = 1; break;
              default:
                grid[index] = 0; break;
            }
          }
        }
    }
}
```

1.3.5 Exchanging halo rows

Every subgrid needs to receive the updated halo rows after every full update of the grid. The MPI processes form a ring in which each of them sends the second row of its subgrid to the previous process (identified by `world_rank - 1`) and the second to last row of its subgrid to the next (`world_rank + 1`). Modulo operations take into account the `world_size`. We used the non-blocking version of the `MPI_Send` and `MPI_Recv` functions to avoid deadlocks.

```
// Function used to send and receive between MPI processes the halo rows
void mpiCalls(std::vector<int>& subgrid, const int k, const int world_rank, const int world_size) {
  MPI_Request request1, request2;

  MPI_Isend(&subgrid[k], k, MPI_INT, (world_rank - 1 + world_size) % world_size,
            0, MPI_COMM_WORLD, &request1);
  MPI_Irecv(&subgrid[subgrid.size() - k], k, MPI_INT, (world_rank + 1) % world_size,
            0, MPI_COMM_WORLD, &request1);

  MPI_Isend(&subgrid[subgrid.size() - 2*k], k, MPI_INT, (world_rank + 1) % world_size,
            1, MPI_COMM_WORLD, &request2);
  MPI_Irecv(&subgrid[0], k, MPI_INT, (world_rank - 1 + world_size) % world_size,
            1, MPI_COMM_WORLD, &request2);

  MPI_Wait(&request1, MPI_STATUS_IGNORE);
  MPI_Wait(&request2, MPI_STATUS_IGNORE);
}
```

1.3.6 Printing the snapshot

The snapshot frequency is determined by the `-s` argument in the program call. When the iteration number is a multiple of s a collective MPI communication of type `Gatherv` is performed to send the grid sections, excluding halo rows, to process 0. Here the complete grid within the grid vector is recreated. The function `createSnapshot` already described in the initialisation modes is called and creates an image named `snapshot_0...0n` where n is the iteration number and the number of zeros is equal to 5 minus the number of digits of n. Again, there is a distinction to be made with respect to the parity of the iteration number: when this is odd in each process it is subgrid1 that contains the most recently updated values and for this reason `Gatherv` must collect the values from them. Of course, the same applies to even iterations and subgrid2.

```
    if (i%s == 0) {
        if (i%2 == 0) { // odd iterations
          //reconstruct the matrix from subgrid1, excluding halo rows
          MPI_Gatherv(&subgrid1[k], sendcounts_elem[world_rank], MPI_INT, grid.data(),
          sendcounts_elem, displs, MPI_INT, 0, MPI_COMM_WORLD);
          if (world_rank == 0) createSnapshot(grid, k, i); //dump the grid inside snapshot_0000i
        }

        if (i%2 == 1) { // even iterations
          //reconstruct the matrix from subgrid2, excluding halo rows
          MPI_Gatherv(&subgrid2[k], sendcounts_elem[world_rank], MPI_INT, grid.data(),
          sendcounts_elem, displs, MPI_INT, 0, MPI_COMM_WORLD);
          if (world_rank == 0) createSnapshot(grid, k, i); //dump the grid inside snapshot_0000i
        }
      }
```

1.4 Results and Discussion

We performed some benchmarking to evaluate the performance of the program on the EPYC nodes of the ORFEO cluster. More precisely we did study openMP strong scalability and MPI weak and strong scalability.
As we previously noted a number of decisions on the parameters have to be made, therefore what we evaluated captures the performance only of a specific usage of the program.
The times did not include the generation of the grid (`-i` phase) but they include the initial reading of the pbm image (i.e. all the `-r` phase). The program was run with `-s 0` so the snapshot is created only after the last iteration, and the time of this printing is included.
Every result regarding the time reported in the tables below was obtained as the mean of five tries.

1.4.1 Strong openMP scalability

This benchmark was performed on the `epyc002` node. It consists in evaluating the runtime of the program for a fixed grid size with only one MPI process, while increasing the number of cores available for openMP.

- Grid size: 10000
- Number of iterations: 200

Number of threads	Elapsed time	Speedup	Efficiency
1	313.684	1.00	1.00
2	158.938	1.97	0.99
4	81.155	3.86	0.97
6	56.647	5.53	0.92
8	42.696	7.35	0.92
12	29.195	10.74	0.90
16	22.430	13.98	0.87
24	15.564	20.15	0.84
32	11.966	26.21	0.82
48	8.543	36.71	0.76
64	6.938	45,21	0.71
96	5.364	58.47	0.61
128	4.765	65.82	0.51

Table 1: Average values for openMP strong scalability on an EPYC node

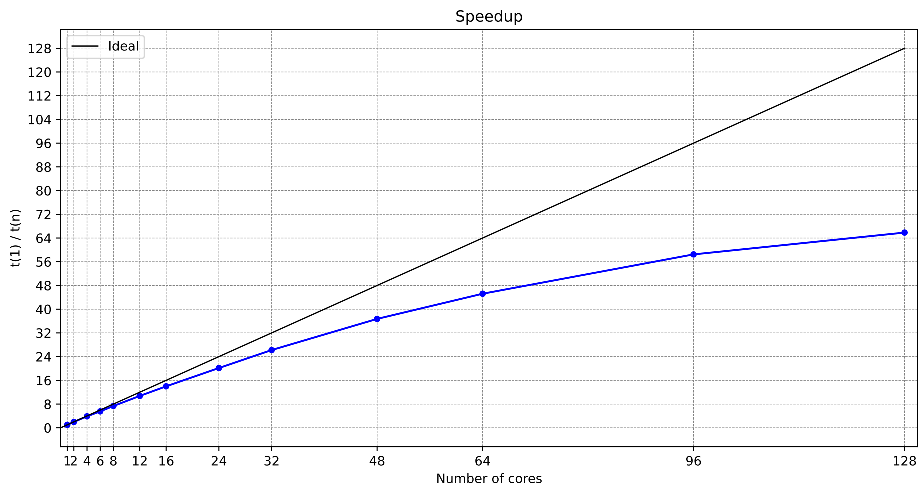


Figure 2: Actual speedup (in blue) versus ideal speedup for openMP strong scalability

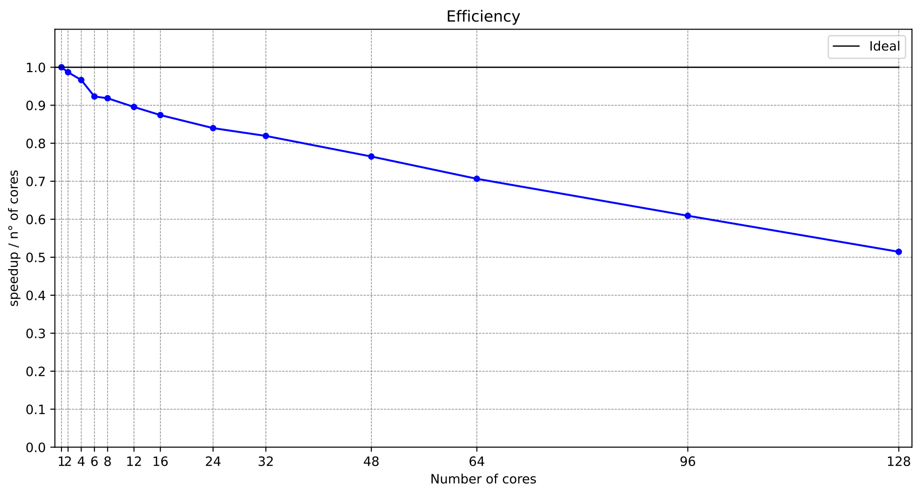


Figure 3: Actual efficiency (in blue) versus ideal efficiency for openMP strong scalability

1.4.2 Strong MPI scalability

The strong MPI scalability is evaluated by fixing a grid size and a number of cores per MPI process while increasing the number of MPI processes. This scalability was evaluated using the `epyc001`, `epyc002` and `epyc007` nodes with a maximum of two MPI processes per node, one per socket, for a total maximum of 6 processes. We repeated these calculations with three different grid sizes. The environment variables `OMP_PLACES` and `OMP_PROC_BIND` were set on `cores` and `close` respectively.

- Grid size: 5000, 10000, 20000
- Number of threads per process: 64
- Number of iterations: 500

Grid size	MPI processes	Elapsed time	Speedup	Efficiency
5000	1	4.367	1.00	1.00
	1	2.808	1.55	0.78
	3	2.305	1.89	0.63
	4	2.141	2.03	0.51
	5	1.932	2.26	0.45
	6	1.902	2.29	0.38
10000	1	14.549	1.00	1.00
	2	8.195	1.78	0.89
	3	6.077	2.39	0.80
	4	5.070	2.87	0.71
	5	4.182	3.48	0.69
	6	4.073	3.57	0.60
20000	1	58.463	1.00	1.00
	2	33.166	1.76	0.88
	3	23.889	2.45	0.81
	4	19.447	3.01	0.75
	5	16.651	3.51	0.70
	6	25.763	2.27	0.38

Table 2: MPI Strong scalability results for three different grid size

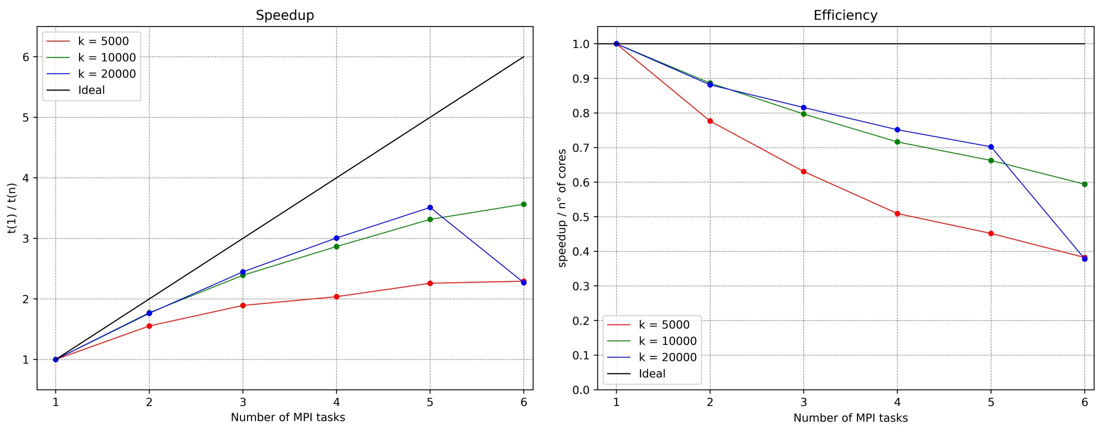


Figure 4: Actual speedup and efficiency against ideal ones for MPI strong scalability

1.4.3 Weak MPI scalability

To evaluate weak MPI scalability we have to increase the number of MPI processes and the grid size at the same time to have a constant problem size per process. This scalability was evaluated using the `epyc001`, `epyc002` and `epyc007` nodes with a maximum of two MPI processes per node, one per socket, for a total maximum of 6 processes. The environment variables `OMP_PLACES` and `OMP_PROC_BIND` were set on `cores` and `close` respectively. In this case the efficiency is defined in the same way speedup was defined in the strong scalability.

- Starting grid size 10000
- Number of threads per process 64
- Number of iterations 500

Grid size	MPI Processes	Elapsed time	Efficiency
10000	1	14.474	1.00
14142	2	16.048	0.90
17320	3	16.772	0.86
20000	4	19.243	0.75
22360	5	19.627	0.74
24495	6	35.972	0.40

Table 3: MPI weak scalability results

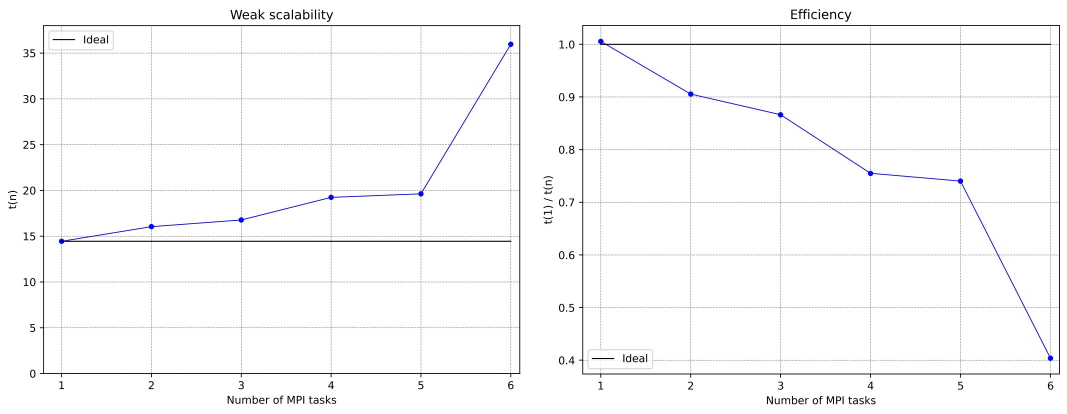


Figure 5: Results and efficiency results against ideal values

We have to note that it seems that something prevents the program to correctly scale for 6 MPI processes with larger grids. After some unsuccessful attempts made by modifying some environmental variables we tried the same benchmark on nodes `epyc004`, `epyc005` and `epyc006` except to find that this behavior of our program was present on these nodes even with 5 MPI processes.

1.5 Conclusions

The scalability results obtained suggest that, although not ideal, the openMP speedup curve is at least monotonously increasing while maintaining an efficiency higher than 0.5 for 128 cores. Aside from the aforementioned behaviour the MPI scalabilities are not too bad either, the program effectively benefiting from an increase in the number of processes.
Many steps can be taken in order to improve the code and its performances. The following is a non-exhaustive list of the improvements we thought of but which were not implemented due to lack of time and/or technical expertise.

- we think that MPI communication management is the main vulnus of our code. Better management of communication policies would certainly lead to better performance.
- it is possible to compute and print in parallel with openMP or MPI, with some threads/a process that print the grid while the others compute the next iteration. This could lead in increasing performances especially when the frequency of the snapshots is high.
- we used only a very simple openMP/MPI hybridization. It is very likely that more refined interactions between the two can lead to better results, especially when increasing the number of MPI processes.
- the initial reading of the grid can be parallelized using MPI I/O, with each process that reads from a file its section of the grid. This also avoid the `Scatterv` communication.
- it is possible that, especially for large sparse grids, reducing the dimension of the data passed via MPI can improve the performance. We experimented with a variant of the run-length-encoding (RLE) but for the size we used the shorter communication time does not compensate for the encoding and the decoding of the grid. Alternatively this encoding could be use to store more efficiently the grids for a later conversion to pbm images.
- as already mentioned, a pattern database can be included to obtain the grid at a certain iteration without having to calculate all the previous iteration.

2 Exercise 2

2.1 Introduction

In this analysis, we evaluated the computing performance of the ORFEO nodes using two of the major bench-marking libraries available: the Intel Math Kernel Library, **MKL**, and the Open Basic Linear Algebra Subprograms, **OBLAS**.
In both cases the focus has been on matrix-matrix multiplication (level 3 BLAS routines). The algorithm used is called *gemm* (general matrix multiplication) and it was tested both for single (32 bit representation) and double precision (64 bits representation). We expect the double precision performance to be half of that in single precision.
We did scalability studies both for the THIN (Intel) and the EPYC (AMD) nodes:

- A THIN node has two sockets, each equipped with an Intel Xeon Gold 6126 (a 12 core CPU)
- A EPYC node has two sockets, each equipped with an AMD Epyc 7h12 (a 64 core CPU)

The theoretical performance is computed as:

$$\text{Number of cores} \times \text{Frequency} \times \text{Floating operations per clock cycle}$$

which amounts to:

- EPYC: 64×2.6 GHz $\times$ 16 (float) $= 2662.4$ GFLOPs ~ 2.7 TFLOPs
- THIN: 12×2.6 GHz $\times$ 64 (float) $= 1996.8$ GFLOPs ~ 2 TFLOPs

The number of operations per cycle for the THIN nodes CPUs is higher because they support two AVX-512 FMA units[3].
The theoretical peak performance evaluated in this way is compatible with both the value found on the ORFEO website[4] and the actual measured performance.
The number of operations per cycle for the EPYC nodes CPUs is lower because they only support one AVX2 unit. We note that when we first ran the benchmarks on epyc nodes, we realized that the Gflops obtained were much higher than the theoretical limit (around 6 Tflops). Upon inspection (using lscpu) we realized that the nodes we were using (`epyc001`, `epyc002`) had hyperthreading activated, although it shouldn't provide better performance on HPC tasks like matrix multiplication. We tried to explicitly set `OMP_NUM_THREADS` to 64 and this lowered the performance to around 3.5 Tflops. This means that, at least on those nodes, fixing -n to 64 when allocating the job does not guarantee that the benchmark uses just 64 cores, but it is likely that it was running on all the 128 cores of the node. Since a performance of 3.5 Tflops, although more reasonable, is still not compatible with the theoretical one, we decided to carry out our analysis on the node `epyc004`, which instead exhibited a normal behaviour.

2.2 Results

We now illustrate the results obtained by our analysis. More specifically, the performance of both THIN and EPYC nodes was studied

- leaving the openMP environment variables unset
- specifying values for the environment variables `OMP_PLACES` and `OMP_PROC_BIND`

`OMP_PLACES` is used to specify the places on the machine to which the threads are bound, while `OMP_PROC_BIND` is used to specify the binding policy (thread affinity policy) which prescribes how the threads are assigned to places.
In our case there was no point in setting the places to *threads*, since the nodes we used did not have hyper-threading activated. The possible combinations are therefore (binding | places):

close | cores   spread | cores   master | cores   close | sockets   spread | sockets   master | sockets

We now encourage the reader to take a look at the graphs in the following two pages.
The THIN nodes exhibit close to ideal scalability, as one can observe especially from the speed up and efficiency graphs (Figure 7 and 8).
On the other hand, EPYC nodes show trends that are very distant from the theoretical ones and not very smooth at all.
Upon thinking on the reasons why, two hypotheses came to mind: one is that of a more complex memory structure in the EPYC nodes with respect to the THIN ones. The second one is that the EPYC nodes execute 16 operations per clock cycle, compared to the 64 of the THIN; it is true that the EPYC have more than five times the number of cores of the THIN, but more communication is needed.
So it looks like increasing the number of cores is nowhere near enough to compensate for fewer operations per cycle.

[3]https://ark.intel.com/content/www/us/en/ark/products/120483/intel-xeon-gold-6126-processor-19-25m-cache-2-60-ghz.html
[4]https://www.areasciencepark.it/piattaforme-tecnologiche/data-center-orfeo/

2.2.1 THIN nodes

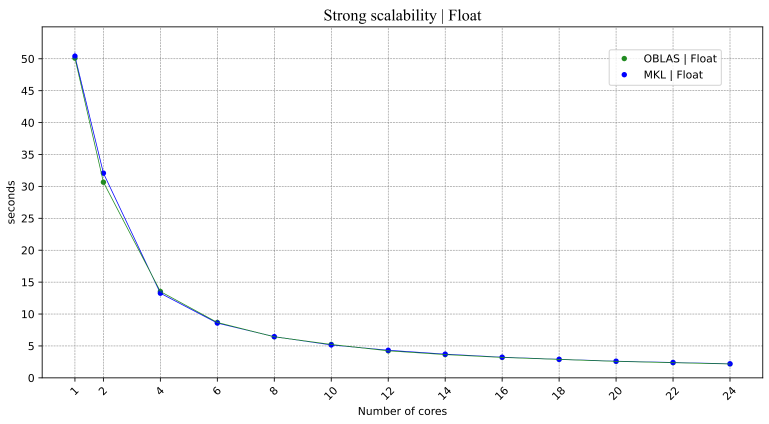


Figure 6: Strong Scalability

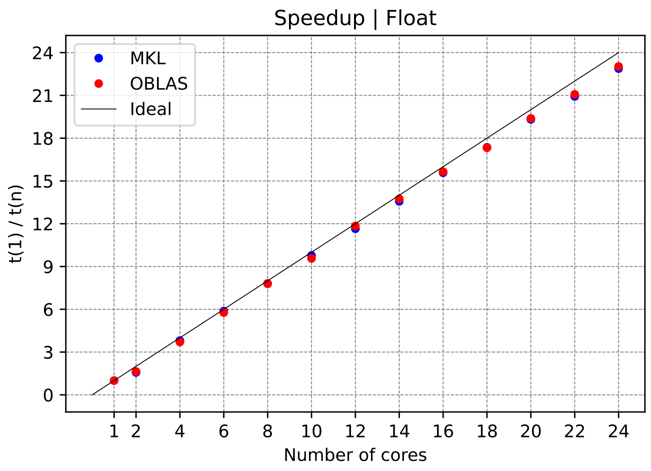


Figure 7: Speed-Up

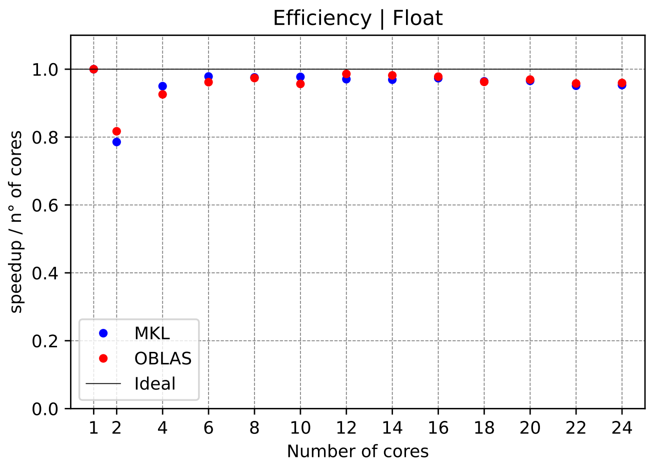


Figure 8: Efficiency

2.2.2 EPYC nodes

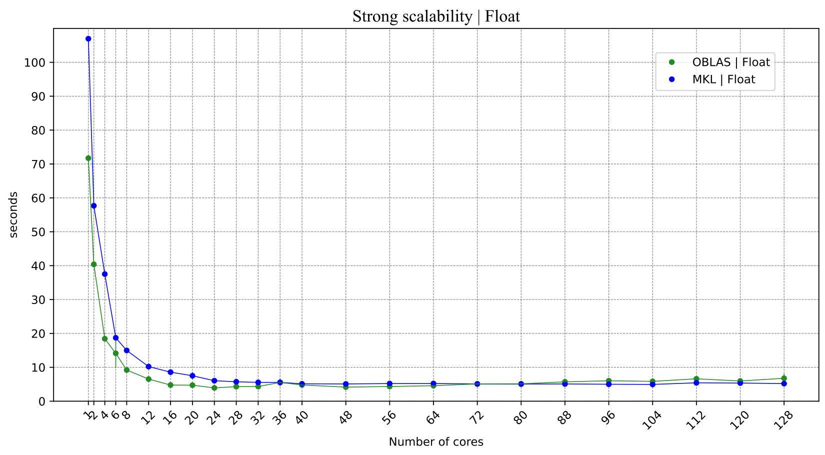


Figure 9: Strong Scalability

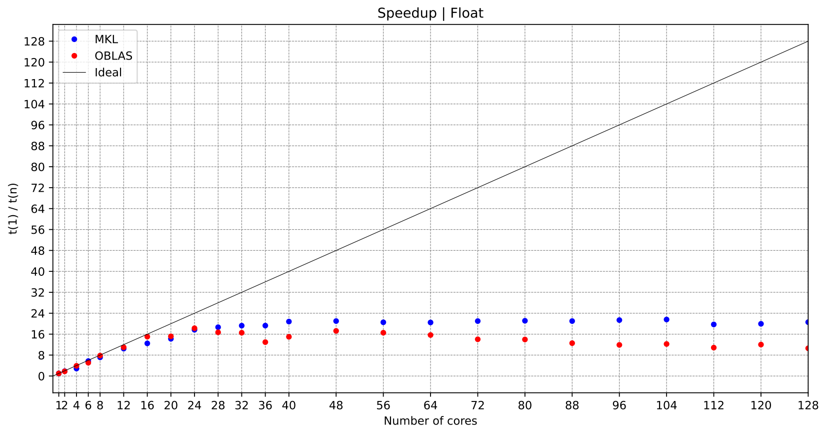


Figure 10: Speed-Up

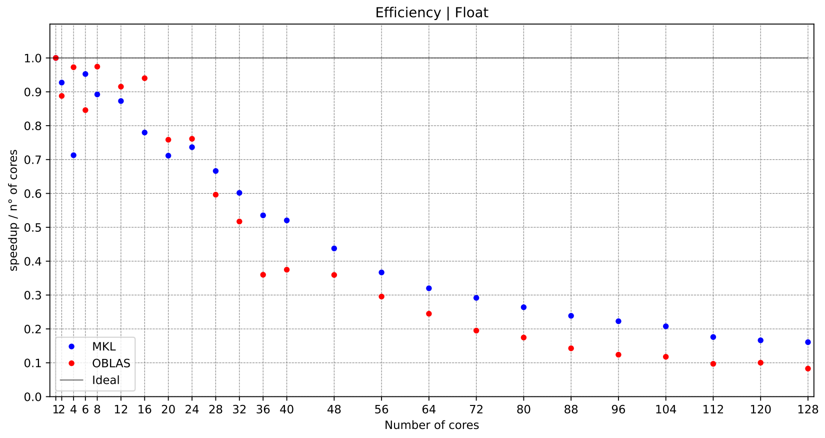


Figure 11: Efficiency

2.3 GFLOPs saturation curve

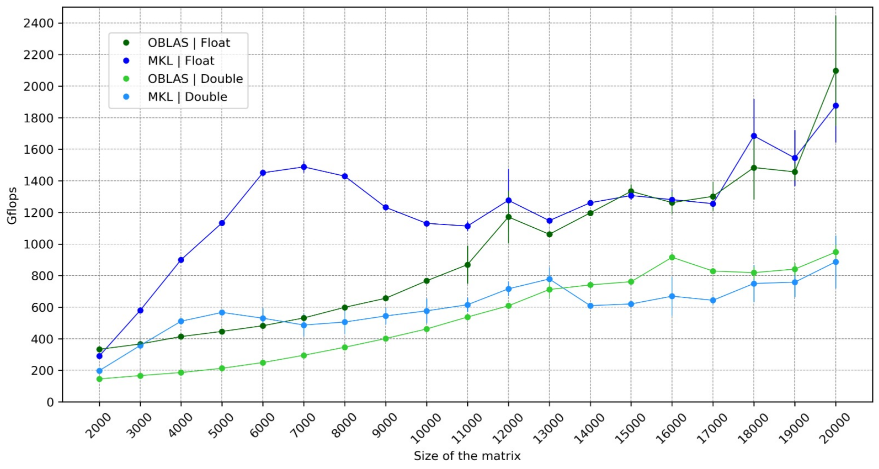


Figure 12: GFLOPs curve for an EPYC node

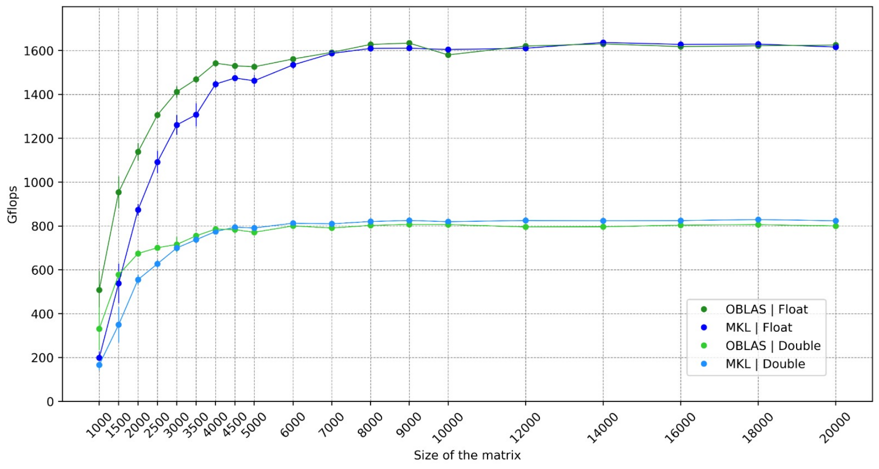


Figure 13: GFLOPs curve for a THIN node

Both graphs show that the double precision performance saturates at a GFLOPs value that is approximately 50% that of single precision, as expected.
Again, while the THIN nodes exhibit close to ideal behaviour, the EPYC nodes curve shows a more unclear pattern. This is especially true for the MKL curve, while the OBLAS one, despite not having the expected shape, has less variability. One explanation could be that MKL is developed by Intel and therefore is more optimized for their CPUs.
As we mentioned above, we also experimented with threads affinity. The graphs are included in the Appendix section.

3 Appendix

3.1 Other graphs from exercise 2

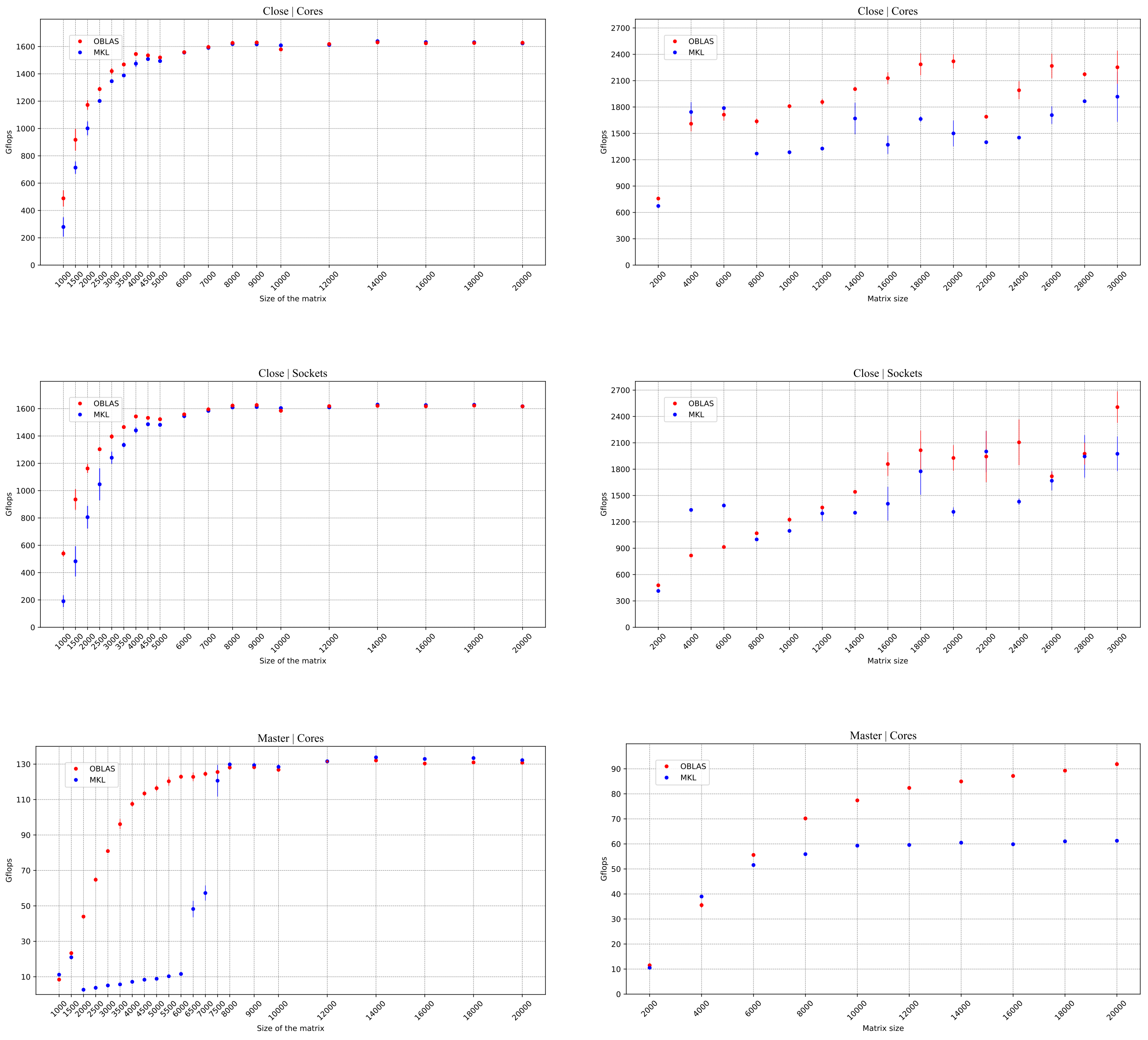


Figure 14: THIN on the left, EPYC on the right

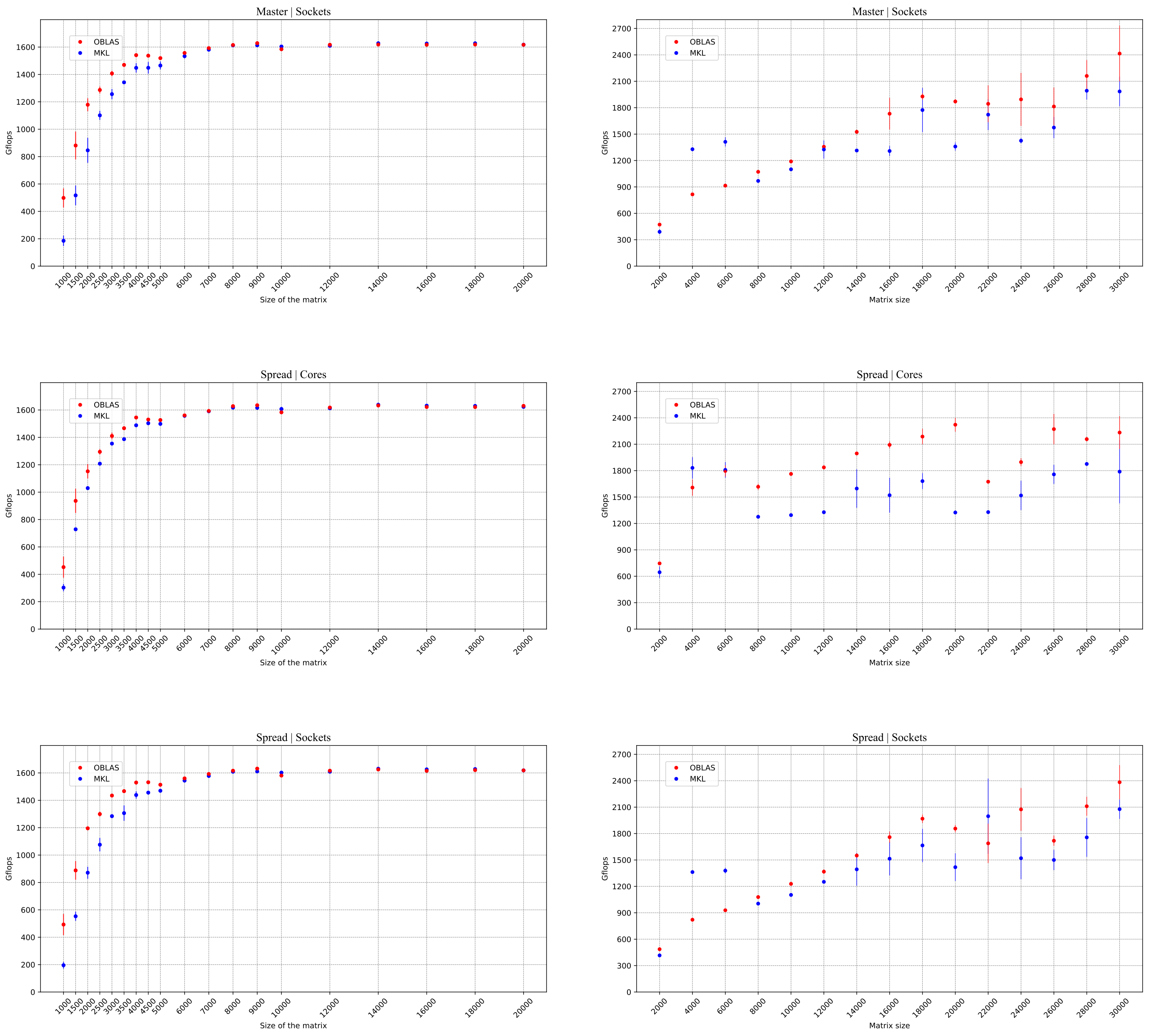


Figure 15: THIN on the left, EPYC on the right