HPC's project report

Michele Sanna

July 14, 2023

Contents

1 Conway's Game of Life **2**
- 1.1 Introduction . . . 2
- 1.2 Methodology . . . 2
  - 1.2.1 Dynamic update . . . 2
  - 1.2.2 Static update . . . 2
- 1.3 Implementation . . . 3
  - 1.3.1 Initialization . . . 3
  - 1.3.2 Update of the subgrid . . . 4
  - 1.3.3 Input Output . . . 4
  - 1.3.4 Message passing . . . 5
  - 1.3.5 OpenMP usage . . . 6
  - 1.3.6 Testing . . . 6
- 1.4 How does it scale? . . . 6
  - 1.4.1 MPI strong scalability . . . 6
  - 1.4.2 OpenMP strong scalability . . . 7
  - 1.4.3 OpenMP weak scalability . . . 8
  - 1.4.4 Conclusions . . . 9

2 Assignment 2 **10**
- 2.1 Different matrices . . . 10
- 2.2 Different threads allocation policies . . . 11
- 2.3 Strong scaling on MKL and OBlas . . . 12

Chapter 1

Conway's Game of Life

1.1 Introduction

In this work we're going to study the scalability and the parallelization design of a program that simulates the popular Conway's game of life. Briefly, this game requires the update of a grid (in this case a square grid, that works like a torus), where the behaviuor of every cell depends on the state of the adjacent cells. The choices concerning the parallelization design and the programming optimization are discussed in part 2 and 3. The code follows the specifications given and is tested against a variety of cases. In part 4 and 5 is discussed the scalability of this implementation.

1.2 Methodology

1.2.1 Dynamic update

Given the inherently serial nature of the dynamic update mode, as described in the assignment description, i didn't find a way to parallelize it that doesn't give different results with respect to the straightforward serial version. A possible way for using parallelism (that doesn't modify the result of the computation) is to parallelize the neighbour state check for every single cell (for example: every thread checks the state of one different neighbour cell), but this doesn't lead to any improvement due to the overhead of the parallelization and the simplicity of the task.

1.2.2 Static update

Domain decomposition

The grid of the game is divided in rectangular subgrids along the y axis. Those grids are as many as the number of processes. Every process reads from the file its subgrid, or fills a random subgrid if we're initializing a new game. We're considering the grid as a torus, so the first row is considered adjacent to the last and viceversa. For a process the subgrid is the section of the grid to actually update, while the others two rows are necessary to check the number of alive neighbours for the cells of the first and last rows of the subgrid, but they belong to a section of the grid that will be updated by other processes. Every subgrid has a number of rows equal to the number of total rows divided by the number of processes. If the division is not even, the rest of the division is spread along the first processes. For example, if the rest of the division is 3, the first 3 processes will have a row more than the other processes.

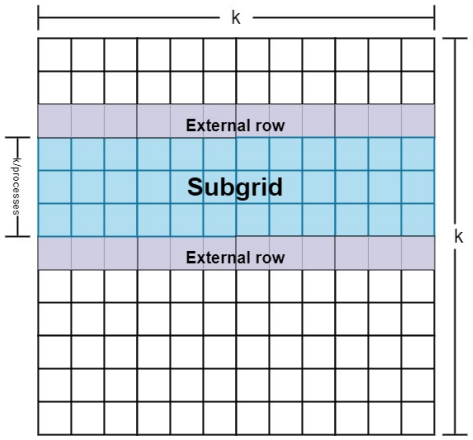

Figure 1.1: A subgrid and its external rows.

Memory exploitation

To better exploit the memory i designed the program in a way that allows the process to never store the entire matrix but only their subgrids. In order to achieve this every process reads a different portion of the same file at the start, and writes a different portion of the grid in the same file at the end. During the computation the processes will send to each other only two rows of the grid and never sees the entire game field.

Message passing

In order to have a consistent update of the grid, the processes need to exchange the two rows not belonging to their subgrid (from now on we will call them "external rows") at every step, indeed they need to have them updated to correctly check the number of alive neighbours for the first and last rows's cells. For the i-th process, the area of the grid represented by the external rows is updated by the processes i-1 and i+1. For the process 0 they are updated by the process n-1 (where n is the number of processes) and process 1. For the process n-1 they are updated by process n-2 and 0.

1.3 Implementation

The program is written in **C** and for the parallelization are used **MPI** processes, for implementig the multithreading it's used the **OpenMP** library. From now on:

- with **k** we will indicate the dimension of the game grid,
- with **world_size** the number of processes,
- with **block** the subgrid representation that a single process sees.

1.3.1 Initialization

After parsing the parameters, every process creates two arrays, **displacements** and **blocksizes**, that are useful to keep trace of the number of rows of every process and the position of its subgrid in the game field without having to calculate them every time. Then, if we're initializing the grid, we allocate the **block** array, that will contain the subgrid and then we fill it with random alive or dead cells. I've choosed a plain array rather than a two dimensional array because i've empyrically observed that it performs better (at the cost of a bit less readable code). If we're reading the grid from a file, the process will read the dimension of the grid from the file, then will read only it's subgrid from it.

1.3.2 Update of the subgrid

For the static update it's used a second array, of the same dimension of **block**, to store the updated values. To update the subgrid every process spawns subthreads. The implementation is pretty straightforward because the domain decomposition of the nested for loops is managed by the omp construct **collapse**. If every thread checks the neighbours and update a different cell with respect to the others, there is no data race going on. Ternary operators are used instead of if constructs in order to optimize (the code, by contrast, is a bit less readable).

The project rules of the assignment are a bit different than the classical rules for the Game of Life, the update function can be compiled both with the classical and with the assignment rules modifying the value of the macro **PROJECT_RULES**.

After the update of the subgrid comes the exchange of the external rows between the processes, then if it's the case we save a snapshot of the grid.

```
unsigned char* staticUpdate(unsigned char* block, int block_rows, long int k, unsigned char* blockB) {
    #pragma omp parallel
    {
        int neighbours = 0;
        #pragma omp for collapse(2)
            for ( int i = 1; i < block_rows - 1; i++ )
                for ( int j = 0; j < k; j++ ) {
                    neighbours = (block[(i - 1) * k + (j > 0? j : k)-1] + block[(i - 1) * k + j] +
                                  block[(i - 1) * k + (j+1)%k] +  block[i * k + (j+1)%k] +
                                  block[(i + 1) * k + (j+1)%k] + block[(i + 1) * k + j] +
                                  block[(i + 1) * k + (j > 0? j : k)-1] + block[i * k + (j > 0? j : k)-1])/MAXVAL;
                    //Here the ternary operator defines the toroid behauviour

                    //The project rules are different from the original rules, so i inserted this directive to enable to switch between official rules
                    #if PROJECT_RULES == 1
                    if (neighbours == 2 || neighbours ==3)
                        blockB[i * k + j] = MAXVAL;
                    else
                        blockB[i * k + j] = 0;
                    #else
                    //We apply the three rules in one single check
                    if (((neighbours == 2 || neighbours == 3) && block[i * k + j] > 0 ) || (neighbours == 3 && block[i * k + j] == 0))
                        blockB[i * k + j] = MAXVAL;
                    else
                        blockB[i * k + j] = 0;
                    #endif
                }
    }
    return blockB;
}
```

Figure 1.2: The file subgrid update function for static mode

1.3.3 Input Output

For the static update case, given the fact that this mode is inherently serial, i used the I/O functions provided in the sample file **read_write_pgm_image.c** (with some small modification, in order to write game fields with large **k**). For the static update case instead i tried to write some faster and scalable function, because I/O is a big limiting factor in scalability.

All I/O functions, even those that I excluded from the final version, are in the **read_write_pgm_image.c** file.

File Writing

To write efficiently into the file and to make file writing scalable I tried to read the file in a parallel way with **MPI I/O**. Unfortunately, the file writing done in this way scales very well when running on a single node, but loses a lot of performance when running on multiple nodes, in such a measure that was overperformed by a simple routine where one process receives the subgrids from all the other processes and writes the entire file. For this reason I instead used a function where all the processes send their subgrid to the master process, and the master process receives and writes the subgrids one at the time, in order to never have the entire grid in the RAM. Even if the routine is de facto serial, I used the **MPI I/O** routines anyway, because they are faster than the **stdio.h** routines. Another reason to not use the **stdio.h** routines is the fact that they create problems when writing a number of bytes greater than the maximum number storable in an integer, I think that this is due to some integer under the hood of the **stdio.h** implementation.

```c
void efficient_write (unsigned char *block, int maxval, long int k, const char* image_name, int world_rank,
int world_size, int block_rows, const int *displacements, const int* blocksizes, const MPI_Datatype row) {
  MPI_Status status;
  MPI_Request req;
  MPI_File fh;
  int offset;
  int written = 1;
  char* str = malloc(70*sizeof(char));
  //Print the header of the file
  sprintf(str, "P5\n# generated by\n# Michele\n%d %d\n%d\n", k, k, maxval);
  offset = strlen(str) + 1;
  if (world_rank == 0) {
    //The master process first writes the offset and its subgrid
    MPI_File_open(MPI_COMM_SELF, image_name, MPI_MODE_WRONLY | MPI_MODE_CREATE ,MPI_INFO_NULL, &fh);
    MPI_File_set_size(fh, (k*k+offset)*sizeof(char));
    MPI_File_write_at(fh, 0, str, offset, MPI_CHAR, MPI_STATUS_IGNORE);
    MPI_File_write_at(fh, offset-1, &block[k], block_rows - 2, row, MPI_STATUS_IGNORE);
    //Then writes all the other process subgrids as it receives them
    while (written < world_size) {
      MPI_Recv(block,  (block_rows-2), row, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
      MPI_File_write_at(fh, offset-1+((displacements[status.MPI_SOURCE]-(2*status.MPI_SOURCE))*k), block, blocksizes[status.MPI_SOURCE] -2, row, MPI_STATUS_IGNORE);
      written++;
    }
    MPI_File_close(&fh);
  }
  else {
    //The other process have only to send their subgrid
    MPI_Isend(&block[k],  (block_rows-2), row, 0, 0, MPI_COMM_WORLD, &req);
    MPI_Request_free(&req);
  }
}
```

Figure 1.3: The file writing procedure

For snapshot saving I used a variant of this function, because with snapshot saving you have to continue the computation after the file writing, so it's not possible to overwrite the **block** array of process 0. In this variant for snapshot saving there is a buffer of the same dimension of **block** to avoid the overwrite.

File reading

For the file reading I used a parallel routine written with **MPI I/O** functions. Luckily, in this case, the use of multiple nodes didn't create any loss in performance. Anyway the header of the file, containing the dimension of the game field and the encoding is read in a serial way: at first every process checks the dimension of the file in the header once at a time, then when the dimension is known they read the file in a parallel way (every process reads only the part which belongs to it). This causes the routine to not scale very well .

1.3.4 Message passing

At the end of every step every process sends and receives from other processes the updated version of its external rows with the rules described in the methodology section. The messages are passed with non-blocking calls, in order to make the message passing routine as fast as possible.

```c
void updateMargins(unsigned char* block,long int k, int block_rows, int world_rank, int world_size, const MPI_Datatype row) {
    MPI_Request req_lower_send, req_upper_send, req_lower_recv, req_upper_recv;
    MPI_Status stat_lower_send, stat_upper_send, stat_lower_recv, stat_upper_recv;
    //Non blocking receiving calls, using module and ternary operator to respect the toroid behaviour
    MPI_Irecv(&block[(block_rows -1) * k] , 1, row, (world_rank + 1) % world_size, 1, MPI_COMM_WORLD, &req_lower_recv);
    MPI_Irecv(block, 1, row, world_rank == 0? world_size -1 : world_rank - 1, 0, MPI_COMM_WORLD, &req_upper_recv);
    //Non blocking sending calls, using module and ternary operator to respect the toroid behaviour
    MPI_Isend(&block[(block_rows - 2) * k], 1, row, (world_rank + 1) % world_size, 0, MPI_COMM_WORLD, &req_lower_send);
    MPI_Isend(&block[k], 1, row, world_rank == 0? world_size -1 : world_rank - 1, 1, MPI_COMM_WORLD, &req_upper_send);
    //Waiting for completion of the request
    MPI_Wait(&req_upper_recv, &stat_upper_recv);
    MPI_Wait(&req_lower_recv, &stat_lower_recv);
    MPI_Wait(&req_upper_send, &stat_upper_send);
    MPI_Wait(&req_lower_send, &stat_lower_send);
}
```

Figure 1.4: The message passing design

1.3.5 OpenMP usage

The functions of the **OpenMP** library are used only in the updating of the subgrids, in the nested for loops that scan the array. I tried to use OpenMP in other chunks of code, but due to the limited dimension of the other loops in the program (they're often of dimension **world_size**), the usage of **OpenMP** constructs didn't result in an improvement in terms of performance, even for large dimensions of the matrix. Also the functional decomposition wasn't a viable option because the program doesn't do much more than initializing an array and updating it.

Despite that, the updating of the subgrids is by far the most computational intensive task, so the use of more **OpenMP** threads is indeed very useful to run the program faster.

1.3.6 Testing

To assess the correct behaviour of the program i used some Game of Life configurations that have a known behaviour. If the macro **IMAGE_TEST** is set to 1, the program will print a .pgm image of variable size containing a column of glider. If the update mode is static, this column should advance through the game field from step to step, if instead the update mode is dynamic the gliders should turn into 4 pixel static blocks.

This is useful not only to verify that the game field it's correctly updated, but also to verify that the file writing doesn't have any errors.

1.4 How does it scale?

In the following section will be shown the results of different tests in order to assess the scalability of the program. For every test present in this section, every point in the plots is an average of 5 different runs with the same configuration, however for this type of experiments i didn't find that the results differed much from run to run, so i didn't plotted the standard deviation. The initial configuration was always a column of gliders (that was useful to check if the program was running without errors, because the result is easily predictable) obtained by reading a file. I made this choice because initializing the game field would be a best case scenario, due to the fact that it's a perfectly scalable process (every MPI process simply initialize its subgrid array and fills it randomly). The file writing was made only once at the end of a run (no snapshots).

1.4.1 MPI strong scalability

The tests for MPI strong scalability were conducted on 3 different sizes of game fields:

- 150000 * 150000,
- 50000 * 50000,
- 10000 * 10000.

All the test were conducted on 4 nodes (as requested in the assignment) and with 50 steps of computation. The choice of big game fields and relatively low number of iteration was made because the updating of the game field it's the part of the program that scales better, so running the program for many iteration would result in better theoretical performance, but the time spent for I/O would be negligible and so the file reading/writing sections of the program would lose importance.

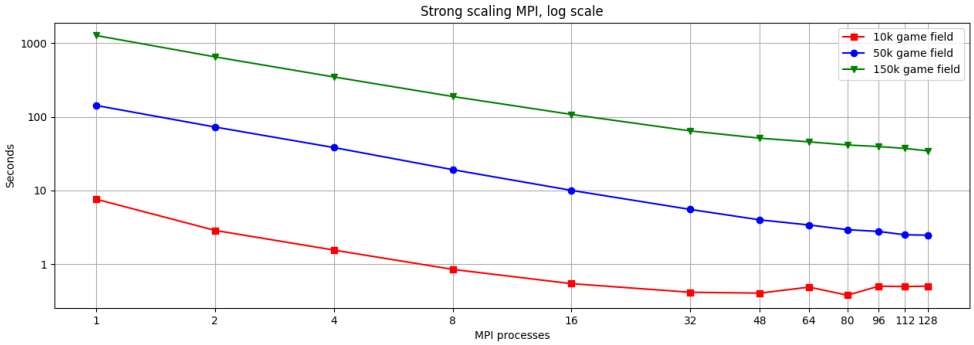

Figure 1.5: Strong scaling of MPI processes, log scale for an easier comparison with the exponential function

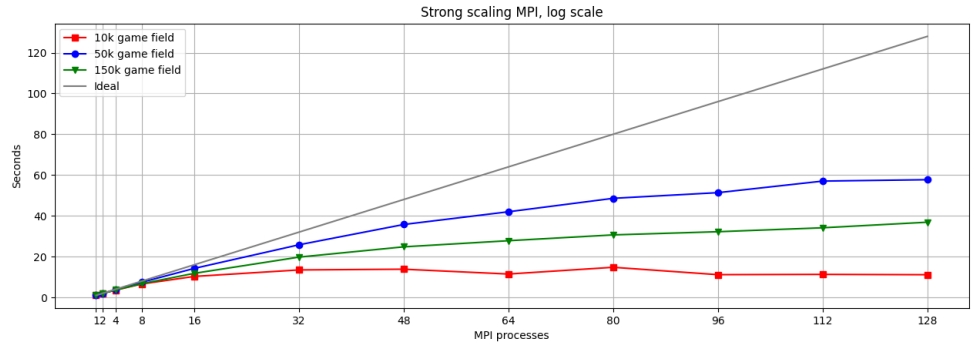

Figure 1.6: Speedup measure for the MPI processes

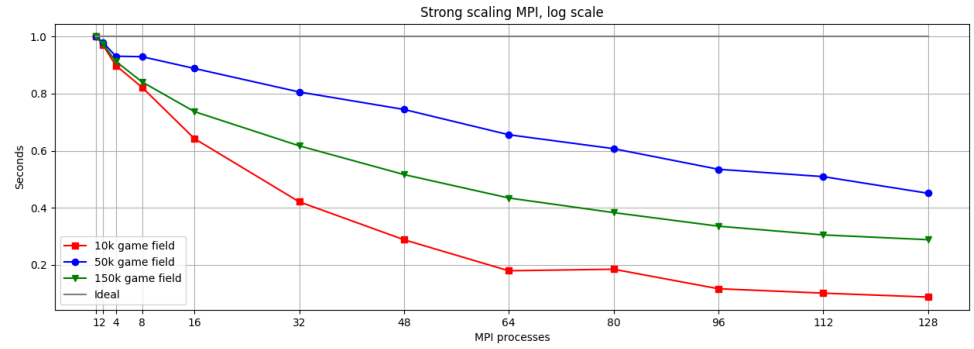

Figure 1.7: Efficiency measure for the MPI processes

1.4.2 OpenMP strong scalability

For this experiment the tests were conducted on a 50000×50000 game field, with 50 steps, and with different number of nodes, but always with an MPI process per socket.

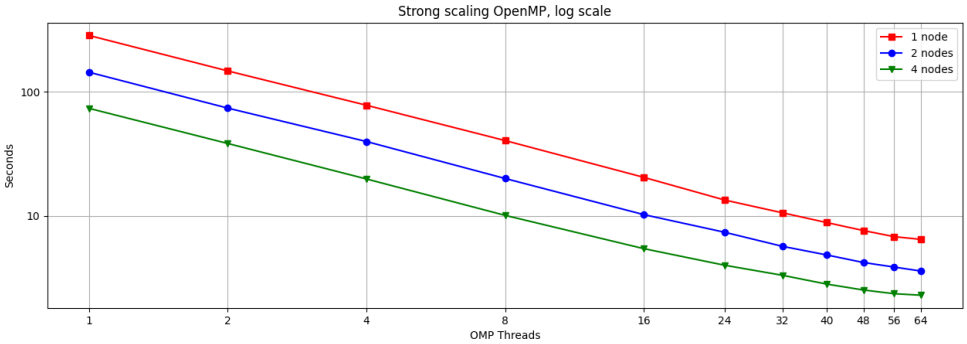

Figure 1.8: Strong scaling of OpenMP threads, log scale for an easier comparison with the exponential function

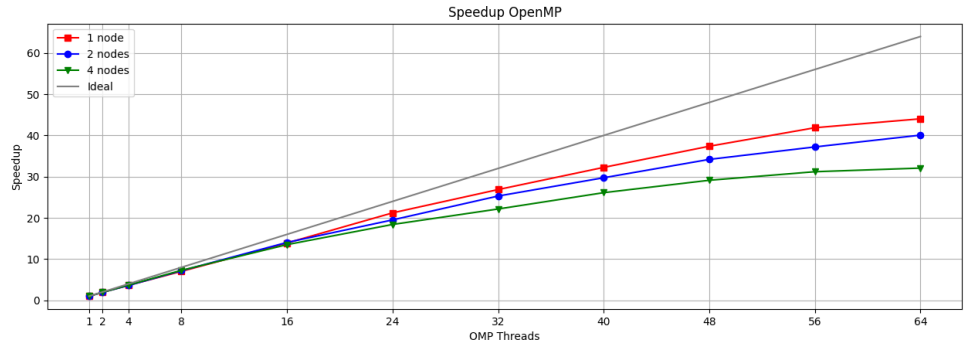

Figure 1.9: Speedup measure for the OpenMP threads

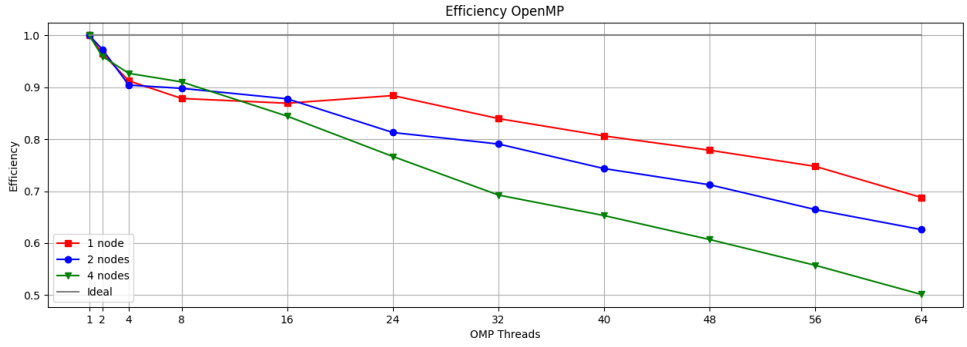

Figure 1.10: Efficiency measure for the OpenMP threads

1.4.3 OpenMP weak scalability

To decide the size of the matrices for this experiments we have to take into account that the workload for the threads doesn't scale linearly with the size of the grid, but rather in a quadratic way, so i chose the following sizes for the k parameter:

- **k**: 25000 for 1 thread
- **k**: 35355 for 2 threads
- **k**: 50000 for 4 threads

- **k**: 70711 for 8 threads
- **k**: 100000 for 16 threads
- **k**: 122474 for 32 threads
- **k**: 173205 for 48 threads
- **k**: 200000 for 64 threads

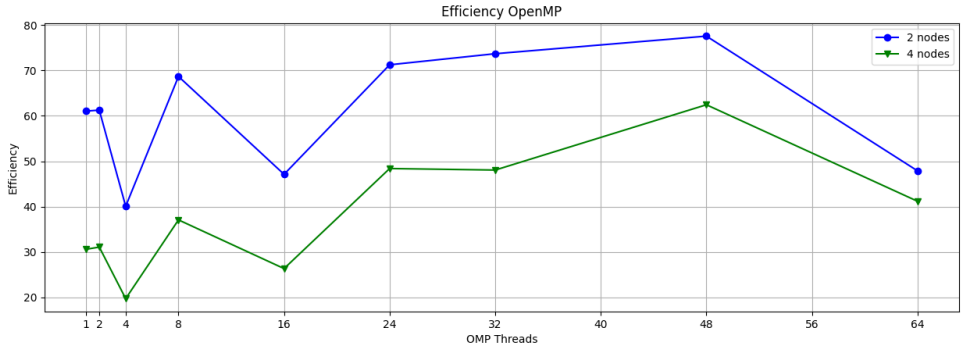

Figure 1.11: Weak scaling performance on OpenMP threads. The behaviour of the program is unusual

The experiments report a strange behaviour, my guess is that this is due to the structure of the files: the 50k, 100k and 200k game fields have much more dead cells, and maybe this leads to faster calculations.

1.4.4 Conclusions

The results from these experiments are much less stable with respect to other tests that I conducted. That's the consequence of having a low number of computation steps and big paramete **k**: with this configuration the I/O part of the program takes a consistent amount of time, moreover the I/O routines that I used don't scale in a linear way: they have a top performance at more less 16 MPI processes and their performance deteriorates with a very high number of MPI processes, and they are a bit unstable too, despite being consistently faster than the serial routines.

Chapter 2

Assignment 2

In this section we will discuss the performance of ORFEO using two libraries for high performance linear algebra: **MKL** (Intel Math Kernel Library), and **OBLAS** (Open Basic Linear Algebra Subprograms). We will compare the performance of these libraries with matrix multiplications of different size (level 3 **BLAS**), with different number of **OpenMP** threads, with different threads allocations options and with different precision (we will use both float operations and double operations).

We will also compare the performance to the theoretical performance of the node. In this case, the thin nodes have two sockets with a Intel Xeon Gold 6126 mounted. The theoretical performance is obtained by:

$$\#cores \times Frequency \times Floating\ operations\ per\ clock\ cycle$$

So in this case we have:

$$12 \times 2.6\ GHz \times 64 = 1996.8\ GFLOPs \sim 2\ TFLOPs\ (float)$$

For the double precision whe should have half the FLOPs.

2.1 Different matrices

For this experiment the gemm.c program was runned with matrices of different size, from 2000×2000 up to 20000×20000. For every size of the matrix the experiment was conducted 15 times, in these results are reported the average times.

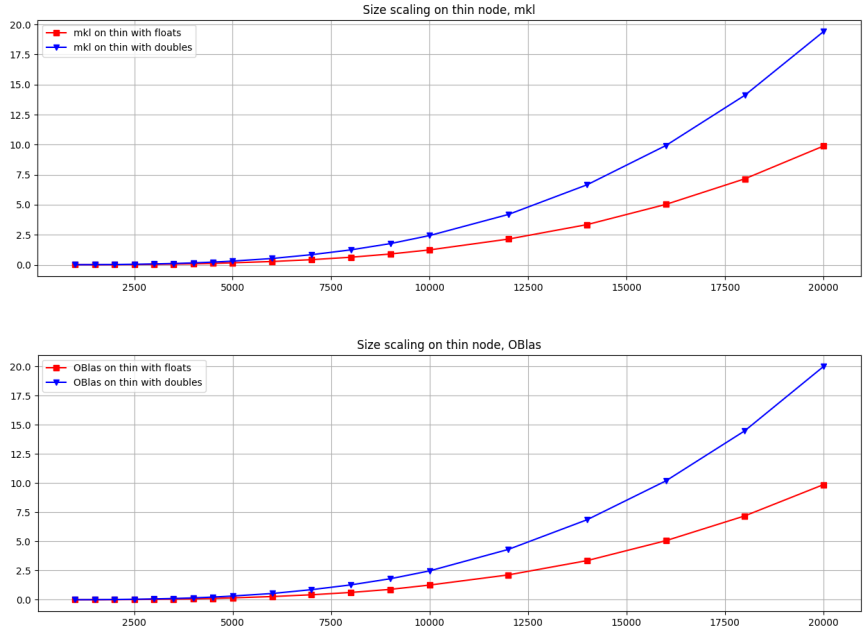

Figure 2.1: Increasing the size of matrices and analysing the time required for the calculation

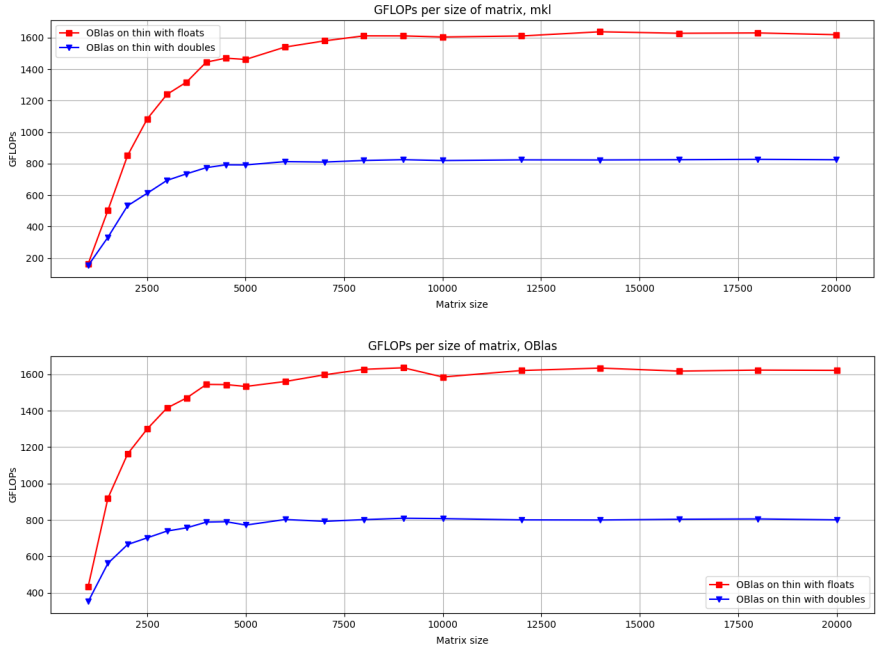

Figure 2.2: Showing the performance obtained in terms of GFLOPs

2.2 Different threads allocation policies

In this sections it's explored how different thread allocations policies affects the performance. The bindings policy used were OMP_PROC_BIND= master — close — spread and OMP_PLACES= sockets — cores . I avoided the option "threads" for te OMP_PLACES variable because the node used for the experiment doesn't have hyperthreading.

MKL

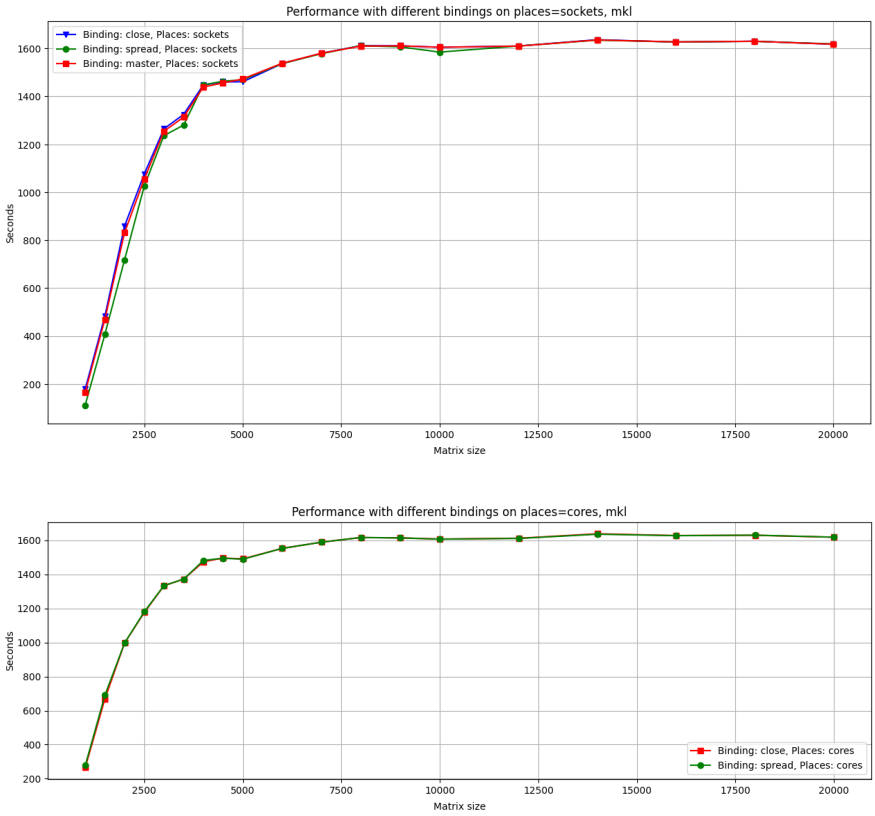

Figure 2.3: Performance under different thread allocation policies in terms of GFLOPs

2.3 Strong scaling on MKL and OBlas

In this case the behaviour has been weird: for limited number of processors the execution time was slower, and this led to weird looking efficiency and speedup results.

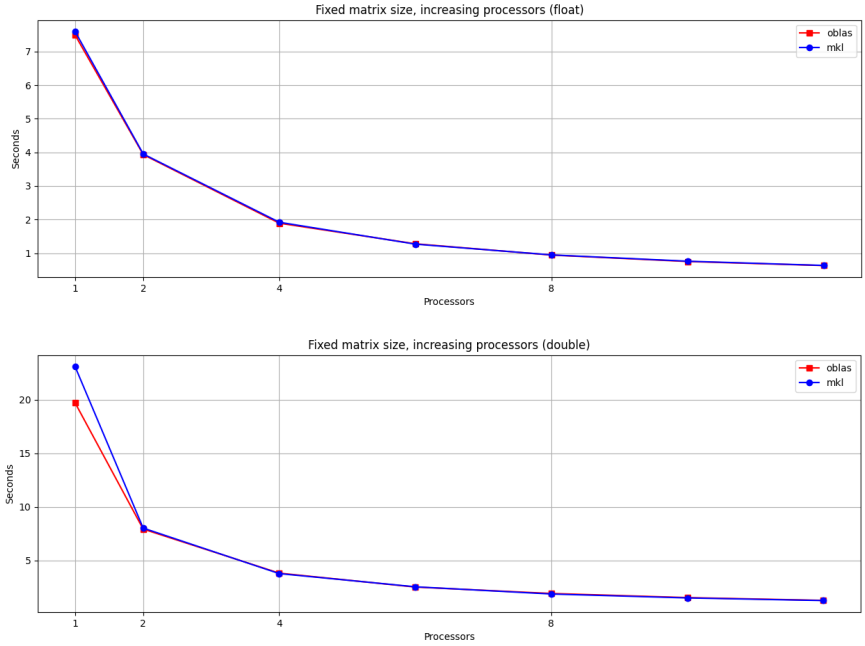

Figure 2.4: Fixed size of the matrix, different number of processors

Speedup

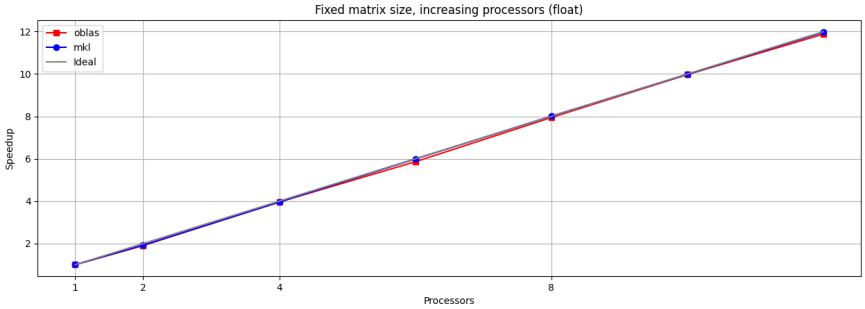

Figure 2.5: Speedup for floats

In this speedup it's already possible to see that the scalability is too good to be true, but with double precision the result become more strange:

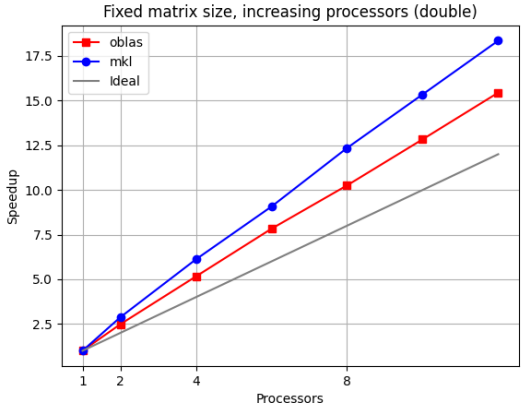

Figure 2.6: Speedup for doubles

Efficiency

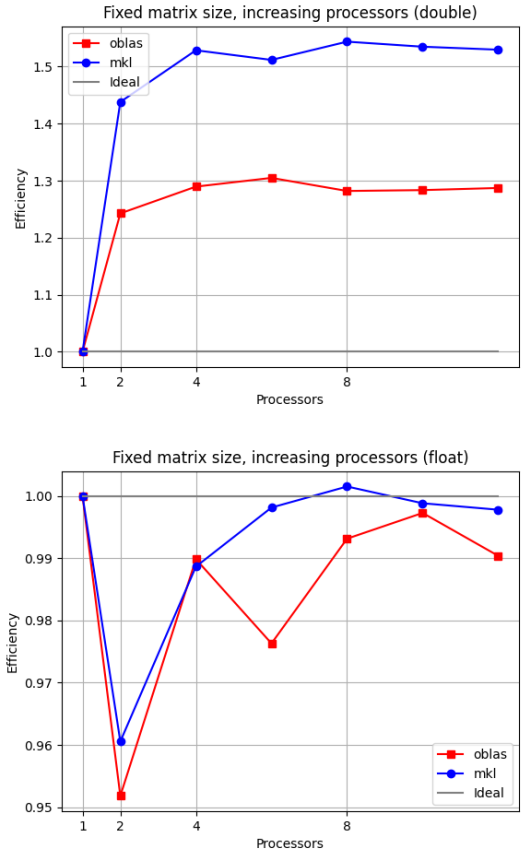

Figure 2.7: Efficiency