REPORT OF THE PROJECT FOR THE COURSE OF FOUNDATIONS OF HIGH PERFORMANCE COMPUTING

FOUNDATIONS OF HIGH PERFORMANCE COMPUTING REPORT

Da Vinchie Lisa

Student ID: SM3500574

Data Science and Scientific Computing

University of Trieste

September 2023

Contents

1 Exercise 1: Game of Life **1**

1.1 Introduction 1

1.1.1 Workload division 2

1.1.2 Initialisation 3

1.1.3 Ordered Evolution 3

1.1.4 Static Evolution 3

2 Results **4**

2.1 Theorethical speed-up 4

2.2 OpenMP scalability 4

2.3 Strong Scalability 5

2.4 Weak scalability 6

3 Exercise 2 **7**

3.1 Introduction 7

3.2 Implementation 8

3.3 Results 9

3.4 Theoretical peak performance 9

3.4.1 Size scalability 9

3.4.2 Fixed matrix dimension 11

ABSTRACT

1 Exercise 1: Game of Life

1.1 Introduction

In this exercise we were asked to implement Conway's game of life using a hybrid MPI+OpenMP code. The general rules are that:

- Each playground is a matrix of size $x \times y$;
- Every cell can be either alive (value 1) or dead (value 0);
- Each cell becomes or remains alive if 2 or 3 of its neighbours are alive, it dies or continues to be dead otherwise;
- The playground must be evolved for n steps and an image of the system must be saved every s steps.

We have two methods to evolve the playground

- Ordered: evolve one cell after another in row major order. In this case, the status of each cell depends on the previous cells, so this operation is not parallelizable.
- Static: evolve each cell based on the information contained in the playground of the previous cycle, that does not change while upgrading.

For both methods, we are asked to study three kinds of scalability:

- OpenMP scalability: keeping the number of processes fixed to 1, vary the number of threads from 1 up to the maximum number of cores in the socket (12 for THIN, 64 for EPYC); the size of the matrix remains fixed for the whole evolution, in order to have a constant total workload.
- Strong scalability: keeping the matrix size fixed, increase the number of processes from 1 up to the maximum number of sockets available, saturating each socket with OpenMP threads.
- Weak scalability: increase the number of processes from 1 up to the maximum number of sockets available, like in the Strong scalability, but keeping the workload per process fixed; I achieved that by using rectangular matrices, with one side fixed to 10000 and the other one increasing proportionally with the number of processes, as $10000 \times N_{processes}$.

The program had to be implemented so that it was divided into two parts:

- Initialisation: initialise a playground of size $x \times y$, specifying its name.

- Evolution: read an already existing playground and evolve it, choosing between the static and the ordered method.

Those options can be selected using inline commands when running the executable file:

- `-i` or `-r`: initialise a new playground or run an existing one.
- `-x` and `-y`: specify width and height of the matrix; this command is needed only if we are initialising the playground, because if we are running it the program will automatically read the matrix dimensions from the header of the PGM image.
- `-f <filename>`: if it is used after the `-i` command, it specifies the name of the playground that will be created. if it is used after `-r`, instead, it indicates the name of the playground that will be evolved.
- `-e <number>`: if it is followed by 0, the program will use the ordered method for the evolution, while if it is followed by 1 the static evolution will be used; of course, this command is to be used only if the command `-r` was previously specified.
- `-n <number>`: it specifies the number of steps of the evolution.
- `-s <number>`: it specifies every how many steps a snapshot will be taken.

1.1.1 Workload division

In all the sections, the work is divided between processes and threads in the following way:

- First the workload is divided between the processes, giving to each process a specific number of rows to evolve, that in the program is called `rows_per_proc`; even though we try to give to each process the same number of rows, that is not always possible, so the value of `rows_per_proc` can vary depending on the process.
  So, in each process will be allocated a matrix, called `image`, that has the same number of columns as the playground, but a number of rows equal to `rows_per_proc + 2`: `rows_per_proc` rows are used in the evolution, while the first and the last row are only used to keep the values of the neighbouring cells.
  If we are performing ordered evolution, this matrix will be evolved. If, instead, we are using the static method, this matrix will only be used to calculate the number of live neighbours, and we will need to allocate an auxiliary matrix for each process for the evolution.
- If we are performing static evolution, the work to be done on each `image` will be split between the OpenMP threads of the process, giving to each thread some cells to evolve in parallel.

1.1.2 Initialisation

This part of the code is used to create a random PGM image of the desired dimensions and write it into a pgm file. Since the size of the image can be cery big, i decided to create the matrix in parallel, using a hybrid MPI and OpenMP code.

1.1.3 Ordered Evolution

This kind of evolution is intrinsically serial, since the evolution of every cell depends on the evolution of the previous ones. In order to prove that, I decided to use an MPI code where every process evolves a portion of the martrix and then passes the necessary information to the folllowing process; I did not perform OpenMP scalability for ordered evolution, since dividing this kind of evolution between OpenMP threads is very hard.
I divided the playground as described in Section 1.1.1; inside the cycle of the generations, I started another cycle on the number of processes: at every cycle, the process receives the first and the last rows of `mage` from the previous and following process, evolves the portion of matrix that was assigned to it, then communicates its second and penultimate rows to the previous and following peocesses.
The communication is done using `MPI_Send` and `MPI_Recv`, that are two MPI functions that implement blocking communications; that ensures that, every time we are sending a row, the program will not advance until the message is received; in order to make sure that everithing is processed in the correct order, I used many `MPI_Barrier` commands.
To measure the time needed by the program to evolve the palyground, I saved the value of the current time in two places of the program, that are, before and after the for cycle on the number of generations; the difference between those quantities is the time required to evolve the playground, and it is saved in a csv file, that also saves informations about the kind of evolution used, the size of the matrix and the number of processes and threads used.

1.1.4 Static Evolution

This kind of preocess, instead, can be parallelized, since the status of every cell depends on the previous generation; however, exactly because of that, I had to allocate two matrices for each process instead of one:

- One to keep the status of the cells obtained from the previous cycle, called `image`, that has dimension `rows_per_proc + 2` $\times$ `x` in order to represent the rows that we want to evolve and the two neighboring rows.

- One to write down the result of the evolution, called `aux_image`, that has dimension `rows_per_proc` $\times$ `x`.

The workload division between the MPI processes is very similar to the one done in the ordered evolution; after that, I furtherly divided the workload between the OpenMP threads, by assigning some cells of the `image` to each one of them for the evolution.
At the end of each iteration, for each process, the values of the cells of `aux_image` are assigned to the cells of `image`, from row 1 to row `rows_per_proc`, with a simple for cycle; the rows 0 and `rows_per_proc + 2 -1`, instead, must be received from the previous and the following process, and at the same time the rows 1 and `rows_per_proc` must be communicated to the same two processes. To achieve that, I used again `MPI_Send` and `MPI_Recv`, similarly to what I did in the ordered evolution. This time, however, more than one send and receive operation is performed at the same time, since at the end of the cycle all the `image` matrices are fully evolved.
The time required is measured and saved similarly as in the ordered evolution.

2 Results

2.1 Theorethical speed-up

The scalability of the program is measured with the speed-up, that is calculated as

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

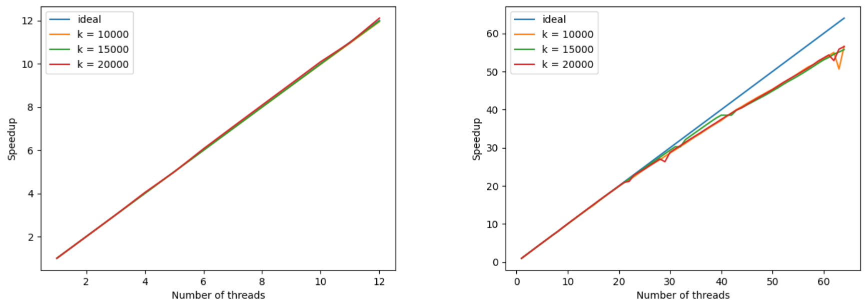
For the ordered evolution, the theorethical speed-up is always one; actually, we expect it to be a bit less than that in practice, since the communications between processes cause overhead. For the static evolution, instead, there are two cases:

- For the OpenMP and Strong scalability, the theoretical speed up is linear
- For the Weak scalability it is constant

In this section, I will show and analyse the results obtained for the two kinds of evolution.

2.2 OpenMP scalability

As we can see in Figure 1, the shape of the measured speed-up is almost linear and very close to the theorethical speed-up, with the THIN nodes giving better results that EPYC; this can be explained with the fact that EPYC nodes have a larger number of cores per socket: that forces the program to perform more communications between nodes, leading to more communication overhead. Moreover, when using the EPYC nodes, we can observe some fluctuations, to be attribuited to some overhead that we cannot control. The results are pretty similar for all the three matrix sizes used.



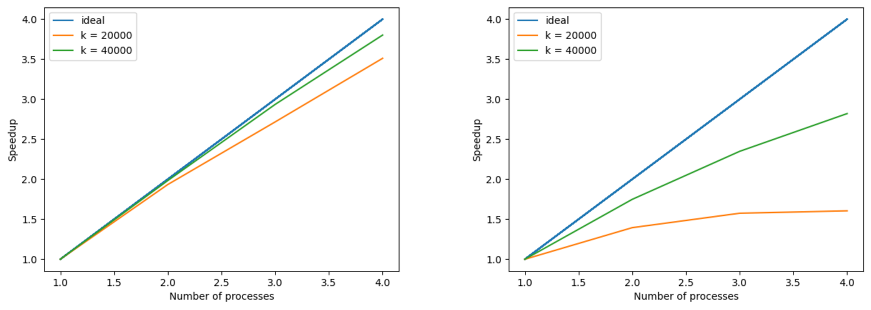
(a) Scalability on THIN node (b) Scalability on EPYC node

Figure 1: OpenMP scalability for the static evolution, obtained with different matrix sizes and 50 steps

2.3 Strong Scalability

As shown in Figure 2, the shape of the mesured speed up for the static method has a linear trend; however, it is evident how the measured speed ups are very different from the ideal ones and, contrary to the OpenMP scalability, the greater the matrix size the better the results. This, probably, is explainable with the fact that the required time for communications remains almost the same for all the matrix sizes; as the dimensions of the matrix grow, the time for communication will represent a smaller fraction of the total time, impacting less on the speed-up.
The communication overhead is probably also responsible for the non perfectly linear trend of the data: as we can observe, the greater the number of processes, the greater the difference between theorethical and measured speed-up. This can be explained with the fact that if we have few processes, we will have few communications between close sockets, since I used the `close` affinity policy; a greater number of processes, instead, will lead to more communications, possibly between sockets on different nodes.

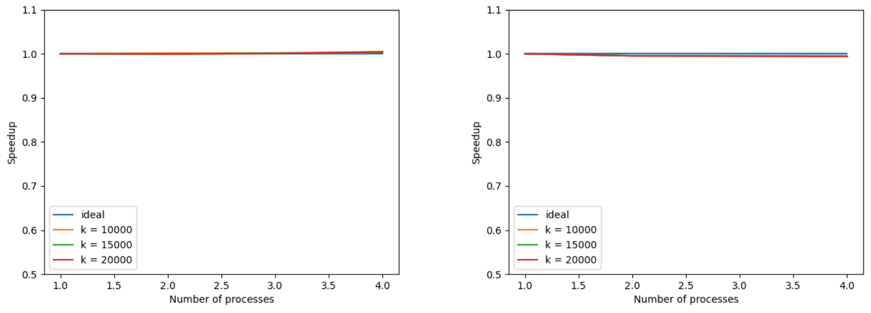
For the ordered evolution, instead, the measured results are very close to the theoretical ones, with differences that become visible only when using 3 or 4 processes, as we can see in Figure 3. Those differences can be attribuited, again, to communication overhead.
In Figure 3a, we can notice that sometimes the measured speed up is slightly greater than one, that is theorethically impossible; this, probably, is due to the fact that,



(a) Scalability on THIN node (b) Scalability on EPYC node

Figure 2: Strong scalability for the static evolution, obtained with different matrix sizes and 50 steps

when only one process is present, the communication of the neighboring rows is performed in a different way, since I did not use `MPI_Send` `MPI_Recv` as in the case of `n_procs > 1`; in this case, I communicated the statius of the two rows with a simple for cycle, and that, maybe, is less efficient, leading to an unexpectedly large time for the 1-process case. This may have affected the computation of the speed-up, leading some of them to be greater than one despite the additional communication overhead.

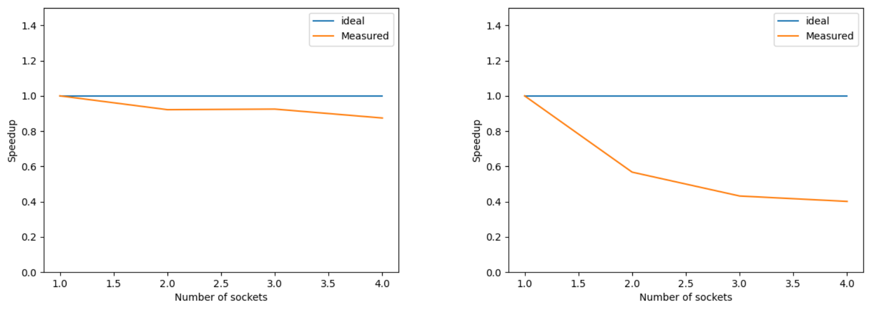


(a) Scalability on THIN node (b) Scalability on EPYC node

Figure 3: Strong scalability for the ordered evolution, obtained with different matrix sizes and 50 steps

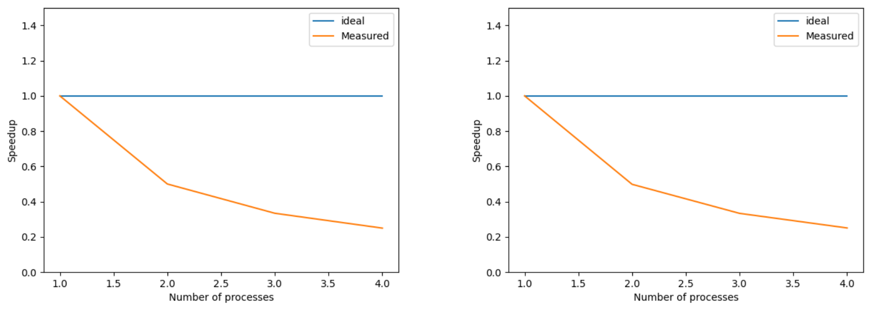
2.4 Weak scalability

As I previously said, in the weak scalability the workload per process is constant, so the speed up should be constant and equal to one. As we can see from Figures 4 and 5, the obtained results are very different from the expected ones, since they present a clear decreasing trend. This can be explained with the fact that, contrary to the strong and openmp scalability, the bigger the matrix size and number of processes, the bigger the fraction of time dedicated to communication.



(a) Scalability on THIN node (b) Scalability on EPYC node

Figure 4: Weak scalability for the static evolution, keeping the matrix width fixed to 10000 and varying the matrix height and 50 steps



(a) Scalability on THIN node (b) Scalability on EPYC node

Figure 5: Strong scalability for the ordered evolution, obtained with different matrix sizes and 50 steps

3 Exercise 2

3.1 Introduction

In this exercise, the goal is to compare the performance, measured in Gflops/s, of different math libraries, that are used to perform matrix-matrix multiplication, either in float or double point precision. The three libraries that we are given to compare are

- MKL

- OpenBlas
- Blis

The exercise is divided into two tasks, that are:

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

Both tasks can be performed in different conditions, that must be chosen between one of the given options whenever we run the program:

- Library: MKL, OpenBlas or Blis
- Partition: THIN or EPYC
- Precision: Double or Float
- Threads affinity policy: spread or close
- Fixed quantity: cores or matrix dimension

3.2 Implementation

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

- gemm.c: it is the C file that I used to perform the multiplications and save the results. It is a slightly modified version of the dgemm.c code, that was given to us along with the assignment, where I added the possibility to save the obtained time and performance results in a csv file.
  The program accepts command line arguments to specify:
  - The library to use, to be chosen between **MKL**, **OPENBLAS** and **BLIS**.
  - The desired precision, to be chosen between **USE_DOUBLE** and **USE_FLOAT**.
  - If we want to save the results in a csv file, using the **SAVE_RESULTS** flag.
  - The dimensions of the two matrices that we want to multiply. Since the number of rows of the first matrix must be equal to the number of columns of the second one, we only have three positional arguments; personally, I decided to use only square matrices.

  Since this file is used for both tasks, there is only one version of it, that is in the exercise1 folder

- Makefile: it is used to compile and run the gemm.c program, for all the three libraries; using the phony targets **float** and **double**, we can decide whether we want to run the program with float or double precision.
- Sbatch files: I used one sbatch file for every possible combination of fixed quantity and partition, that is, four sbatch files in total, that are used in order to:
  - Specify the partition that we intend to use and allocate the required number of cores
  - Load the modules
  - Specify the thread affinity policy
  - Call the Makefile in order to compile the C file
  - Use a for cycle in order to run the executable files for different matrix dimensions or cores

3.3 Results

3.4 Theoretical peak performance

The theoretical peak performances for each core can be calculated either as

$$TPP = \text{cores} \times \text{frequency} \times \text{flops per cycle} \tag{2}$$

or by dividing the peak performance of the whole node for the number of cores of the node, depending on what data we have.
For the EPYC nodes I used the first method, since all the information needed is reported here; using those data I calculated that the theoretical peak performance for each core is

$$\begin{aligned} TPP^{double}_{EPYC} &= 2.6\,\text{GHz} \times 16\,\text{Flops/cycle} = 41.60\,\text{GFlops/s} \\ TPP^{float}_{EPYC} &= 2 \times TPP^{double}_{EPYC} = 83.20\,\text{Gflops/cycle} \end{aligned} \tag{3}$$

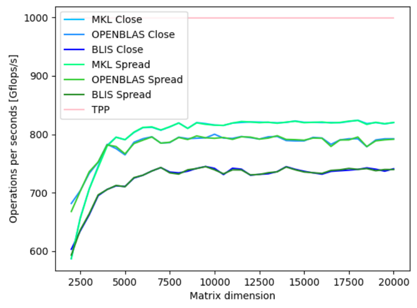
In the case of THIN nodes, instead, I couldn't find any information about the frequency or the flops per cycle, but the course slides reported the theoretical peak performance for the whole node and the number of cores per node, so I used the second strategy:

$$\begin{aligned} TPP^{double}_{THIN} &= \frac{1997\,\text{GFlops/s}}{24\,\text{cores}} = 83.21\,\text{GFlops/s} \\ TPP^{float}_{THIN} &= 2 \times TPP^{double}_{THIN} = 166.42\,\text{GFlops/s} \end{aligned} \tag{4}$$

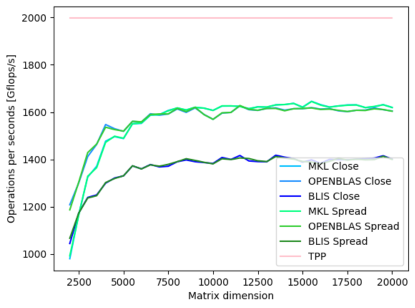
3.4.1 Size scalability

In the following figures I examine the relation between performance, measured in Gflops, and matrix dimension, keeping the number of cores constant.

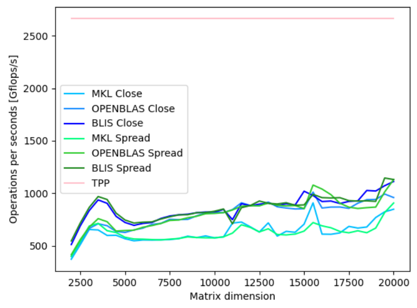
As required in the assignment, in the this case I fixed the number of cores to 12 for the THIN nodes and to 64 for the EPYC nodes; the matrix dimensions, instead, varied from 2000 to 20000, using steps of 500; for each matrix dimension, the programs were run five times.



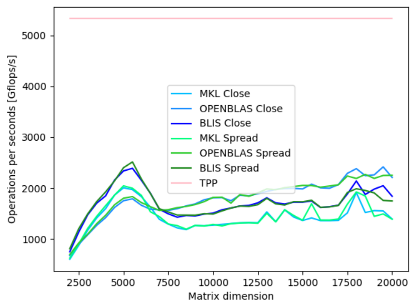
(a) THIN node and double precision



(b) THIN node and float precision



(c) EPYC node and double precision



(d) EPYC node and float precision

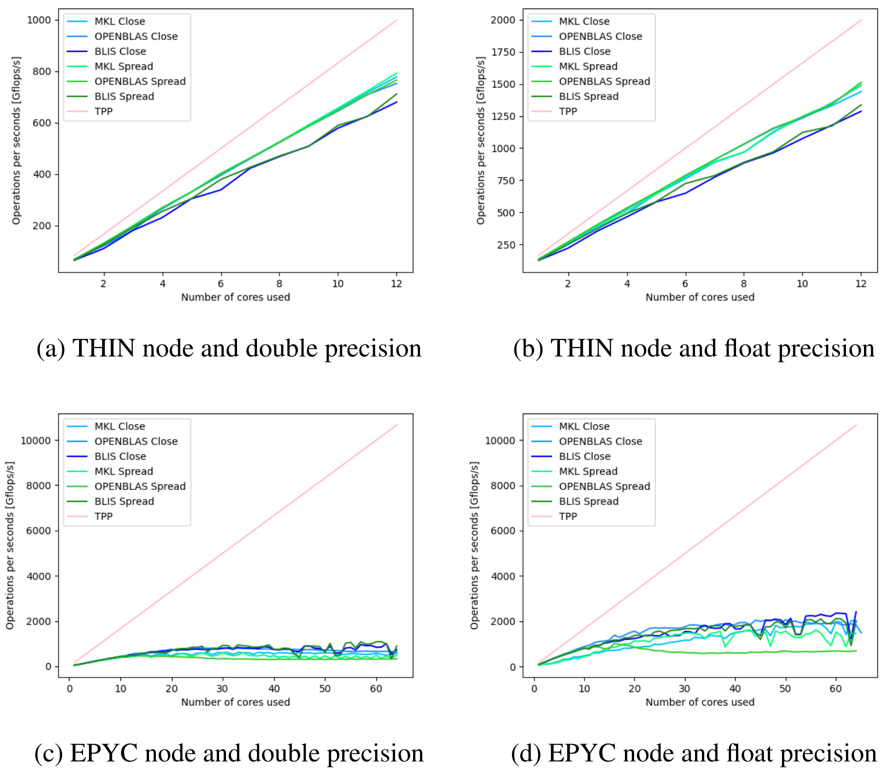
Figure 6: Results obtained fixing the number of cores and changing the size of the matrices

Ideally, the performance should be constant, since the number of cores is fixed; the theoretical peak performances for each kind of node and precision can be calculated by multiplying the corresponding TPP per core, calculated in Equation (4), for the number of cores used. As we can see from Figure 6, the measured performance is much lower than the theoretical one, with the THIN nodes being closer to TPP than the EPYC ones. The thread affinity does not affect the performance since we are using just one socket and all its cores, so the way that we use to choose them does not change the final result.
When using THIN nodes, the library with the best performance is MKL, while the one with the worst performance is BLIS; when using EPYC nodes the roles are reversed and the differences between the three libraries are smaller.
We can also notice that the graphs of the two nodes present two different trends: when using the THIN nodes, the performance increases until the matrix dimension is more or less 5000, then it stabilises. When using the EPYC nodes, instead, the performance increases rapidly

for small matrix dimensions,reaching a peak between 2500 and 5000, then it decreases rapidly and increases slightly for greater matrix dimensions.
This can be explained with the different number of cores of the two kinds of node: THIN nodes only have 12 cores, while EPYC nodes have 64 cores.

3.4.2 Fixed matrix dimension

In the following figures I examine the relation between performance, measured in Gflops, and number of cores, keeping the dimension of the two matrices constant.
As we can see in Figure 7, all the three libraries perform way better on THIN nodes rather than on the EPYC ones; as we can see from figures 7c and 7d, the measured performance is pretty close to the theoretical one when the number of



(a) THIN node and double precision

(b) THIN node and float precision

(c) EPYC node and double precision

(d) EPYC node and float precision

Figure 7: Results obtained fixing the size of the matrices and changing the number of cores