High Performance Computing Assignment 2022/2023

Filippo Olivetti & Cristian Curaba

March 2023

Indice

1 Exercise 1

1.1 Introduction

1.1.1 Rules of the Game of Life (GOL)

Each cell can be either “alive” or “dead” depending on the conditions of the neighboring cells:

- a cell becomes, or remains, alive if 2 to 3 cells in its neighborhood are alive;
- a cell dies, or do not generate new life, if either less than 2 cells or more than 3 cells in its neighborh are alive (under-population or over-population conditions, respectively). In this way, the evolution of the system is completely determined by the initial conditions and by the rules adopted to decide the update of the cells’ status.

1.1.2 Notations

This report aims to describe the algorithms chosen for running the Game of Life and their performances, according to the "size" of the problem. Firstly, we define precisely what are the inputs of the problem and what are the outputs.

The main inputs are:

- matrix size, that is a rectangular grid isize $\times$ jsize (in the experiments we will focus just on square matrices);
- number n of steps to be calculated;
- the type of evolution: static or ordered;
- the frequency s of screenshot requested during the run.

Considering the hardware point of view, there are other important inputs such as the number of MPI processes (nMPI) and OMP threads (nOMP).

The outputs are the final snapshot after n steps and the time measured during the run of the algorithm which is crucial for the scalability analysis (we will focus on it later).

1.1.3 Methodology

The general setting of this work is organised into the main.c and two source files: write_pgm_images.c and game_functions.c. The main actually record all the inputs and chooses to run the static or the ordered algorithm. In write_pgm_images.c there are the read and write functions for *pgm* images (this is taken from github repository assignment1). Finally, game_functions.c contains all we need to run the game, in particular there are:

- void initialize_game(int isize, int jsize, const char *image_name, double prob) just creates a isize $\times$ jsize matrix with every cell alive with probability prob and it saves it in a file with name image_name.
- void run_game(int isize, int jsize, const char *image_name, int total_steps, int step_for_snap, int evolution_type) gets the initial matrix from image_name, then run the static (or ordered if evolution_type == 0) for total_steps number of steps.
- unsigned char upgrade_cell(unsigned char **pixels, int i, int j, int isize, int jsize) is the nucleus of the algorithm: it returns a char value (’0’ if alive,’255’ if dead) that represents the new status of the cell (i, j), given the 2d array pixels. This function follows the rule of the game, and note that it just needs a "local" information of the neighbourhood of (i, j) to give an output, not the all matrix.

The key part of run_game algorithm is the nested ’for’ loop with the upgrade_cell function. This for loop can be parallelized using OMP pragmas, and this is the main and unique point in which we used openMP tools. The entire run_game function is built in order to exploit a possible increasing number of MPI processes in the following way:

- We split the starting 2d array into nMPI slices with horizontal cuts (so the slices are of $k \times$ jsize, where k is to be determined). If isize is a multiple of nMPI, all the slices will be equal of dimension: $\frac{\text{isize}}{\text{nMPI}} \cdot \text{jsize}$. Otherwise if the euclidean division gives us isize $= q \cdot \text{nMPI} + r$, then the slices will have vertical length of:

$$q+1, q+1, \ldots, q+1, q, \ldots, q$$

with exactly r times the number $q+1$. Hence every MPI process performs its task only on its own slice. It is now obvious that at the beginning of every iteration we need an MPI_Scatterv call to distribute the slices, and at the end of the iteration an MPI_Gatherv to rebuild the entire image (see **Conclusions** section from improvements on this part).

1.2 Implementation

1.2.1 upgrade_cell **function**

We start from the simple function upgrade_cell. It doesn't exploit any OMP threadization nor MPI processes, because the operations done are very simple and fast. The only possible problem is the presence of the boundaries, but it is easily overcome by using some basic modular arithmetic. Firstly, we store the coordinates of the neighbours, in the following way:

```
int i_mod[3] = {(isize+i-1)%isize, i%isize, (isize+i+1)%isize};
int j_mod[3] = {(jsize+j-1)%jsize, j%jsize, (jsize+j+1)%jsize};
```

Then we evaluate the sum of the number stored in the neighbours, and check if it is equal to $255 \cdot 5$ or 2 (in this case the cell (i, j) becomes alive, that is '0').

1.2.2 run_game **function**

The codes of this function are logically divided into two parts, the first concerning the ordered evolution, the other the static evolution.

Ordered evolution This code exploit only OMP threadization, because it is intrinsically serial. So, the core of this function is the following:

```
#pragma omp parallel for shared(isize,jsize,pixels) collapse(2)
for( int i=0; i<isize; i++){
    for( int j=0; j<jsize; j++){
        pixels[i][j] = upgrade_cell(isize,jsize,pixels,i,j);
    }
}
```

The previous nested 'for' cycle is inside a big 'for' cycle that iterates on the number of steps (time_steps is the variable that counts the steps). Inside there is also:

```
if( time_steps%step_for_snap == 0){
    sprintf(snapshot_string, "snapshot_O_%05d", time_steps);
    write_pgm_image(pixels,MAXVAL, isize,jsize,snapshot_string);
}
```

Static evolution In the static evolution we exploit MPI and OMP features, as generally explained in the section before. After the call MPI_Comm_rank(MPI_COMM_WORLD, &rank), we split the 2d array pixels into nMPI "slices":

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// Here we split the image along the y axis
int sub_isize = isize / size;
int rest = isize%size;

// In the following way we can share part of the image in pieces of sub_isize and sub_isize
    +1
int rank_minor_rest = (rank < rest);
int start = rank * sub_isize + (rank) * (rank_minor_rest);
int end = (rank + 1) * sub_isize + (rank +1) * (rank_minor_rest);

//we need this because the previous assignment from a certain point adds only zeros
if (!rank_minor_rest){
        start += rest;
        end   += rest;
}
```

Then, we need to create two array of integers for the calls of MPI_Scatterv and MPI_Gatherv, which are respectively senddispls and recvdispls. The entry i of such arrays specifies the displacement (relative to sendbuf or recvbuffer) from which to take the outgoing data to process i.

Other two necessary arrays, one for MPI_Scatterv one for MPI_Gatherv, are sendcounts and recvcounts. Their scope is to specify the number of elements to send (or receive) to (from) each processor. Here there is the code and then an explanation:

```
int sum = 0;
int double_jsize = 2 * jsize;
for(int i = 0; i < size; i++){
    recvcounts[i] = (sub_isize + (i < rest)) * jsize;
    sendcounts[i] = recvcounts[i] + double_jsize;
    recvdispls[i] = sum;
    senddispls[i] = sum - jsize;
    sum += recvcounts[i];
}
```

Concerning the MPI_Gatherv call (so, recvcounts and recvdispls) we just need to create the slices as explained in the **Methodology** section, that is $q \times$ jsize or $(q+1) \times$ jsize depending on the rest.

For the MPI_Scatterv call, we want to update a slice of dimensions $k \times$ jsize that begins at row i. So, we need at least a slice of $(k+2) \times$ jsize that begins at row $i-1$ (then the function upgrade_cell works). This is what exactly we are doing in the previous code for senddispls and sendcounts. However, the reasoning do not work with the first and the last slices because we would use rows indexes of -1 and isize, that are not defined. So, we fix this problem by sending each time all the 2d array, that is the aim of the following code.

```
senddispls[0] = 0;
senddispls[size-1] = 0;
sendcounts[0] = isize * jsize;
sendcounts[size-1] = isize * jsize;
```

Clearly, this is unoptimized version because we send each time all "central" datas of the array, which is pointless for the first and the last slice. In any case, all other slices receive just the information they need, so there are no obvious optimizations in this sense. Now, the core of the static evolution function is almost straightforward. After the allocations of the arrays data_image and sub_image, the following code is inside a 'for' loop that iterates over the number of steps.

```
MPI_Barrier(MPI_COMM_WORLD);
MPI_Scatterv(pixels, sendcounts, senddispls, MPI_UNSIGNED_CHAR, data_image,
    end_minus_start * jsize, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);

#pragma omp parallel for shared(temp_init, xsize, jsize, pixels, sub_image) collapse(2)
for(int temp = temp_init; temp < end-start+temp_init; temp++){
    for(int k = 0; k < jsize; k++){
        sub_image[temp-temp_init][k] = upgrade_cell(isize, jsize, data_image, temp, k);
    }
}

MPI_Barrier(MPI_COMM_WORLD);
MPI_Gatherv(sub_image, (end-start) * jsize, MPI_UNSIGNED_CHAR, pixels, recvcounts,
    recvdispls, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
```

The variable end_minus_start is an integer (different for each MPI process) that represents the number of rows contained in data_image. Also temp_init is a variable defined outside the global 'for' loop, and represents the row index of data_image that corresponds to the cell (i, j) that has to be updated. The methods for snapshots creation is the same of the ordered case.

As we can read from the previous analysis, what we have done is a domain decomposition among MPI tasks, and threads are used only with the nested for loop. What follows is a scalability analysis for both OMP and MPI features.

1.3 Results & discussion

Recall that Amdahl's law states that "the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used". This is the key point we wish to underline of experiments: the improvable part is not always the slowest part of the code.

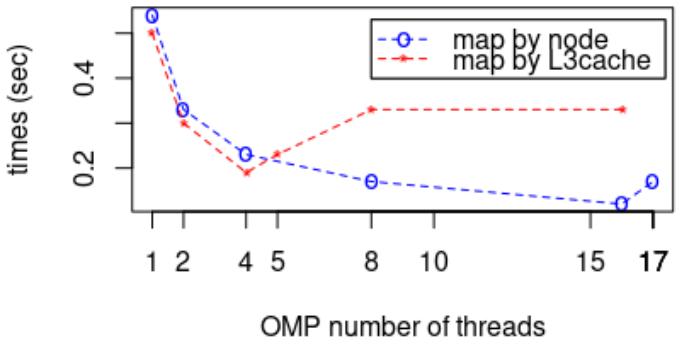
The section is divided into a OMP and MPI scalability analysis, for each case we provide some meaningful examples and relative comments. When we talk about "the time of the nested 'for' loops" we refers to the time measured to execute the lines 5 to 10 of the previous code. All datasets are present in the GitHub repository, in the 'statistics' folder.

1.3.1 OpenMP scalability

Example 1: Here are reported the inputs data of our first experiment

> Size: 1000×1000, $n = 100$, $s = 50$, nodes: 3 EPYC, MPI processes = 8, OMP_PLACES=cores, OMP_PROC_BIND=close

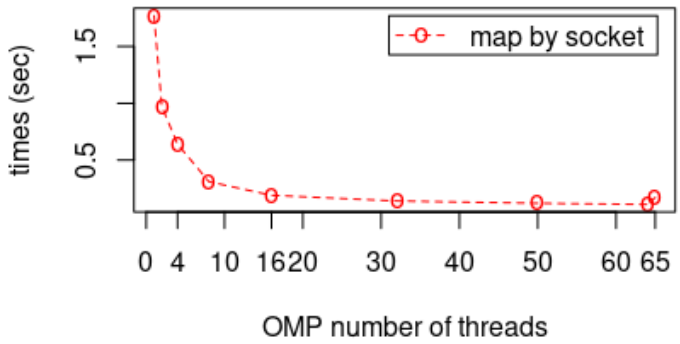
The results are shown in the following plot:



Note that a map by L3cache increase performance until the number of threads is 4, then it increases t time needed. This is simply explained by the fact that a L3cache is divided among 4 cores and then at mos 4 threads (since SMT = 1 in this case) can be exploited for each MPI process. The same reasoning holds for map by node option: indeed, when we consider 8 MPI processes and 16 threads for each process, then 128 cores are sufficient. However, by the option -map-by node all MPI processes are spread among the 3 node we have requested.

Something strange happens if we try to use the –map-by socket option with $nMPI = 8$ (not shown in figure): the time taken is approximately indipendent with respect to OMP threads and it is 12.6 ± 0.5 seconds. Note that we have requested 3 EPYC nodes, so we have theorically at most 6 sockets (with 64 cores each), hence when we use more than 6 socket the slurm has some troubles in finding the appropria location for each MPI process. Usually, I know how many sockets I can use, so I will ask for less: let us decrease $nMPI$ for testing properly the -map-by socket option.

> Same as before but with MPI processes = 2.



Note that in the datasets (in GitHub) the time measured for a MPI_Scatterv (idem for MPI_Gatherv) starts from the call of MPI_Barrier to the end of MPI_Scatterv. Every MPI call takes a time of the order of 10^{-4} seconds, which is really fast compared to the updating for loop which takes, for every single iteration with just one thread, approximately 10^{-2} seconds. By Amdahl's law, we can make the process faster, increasing the number of OMP threads (also increasing the number of MPI processes, but we will s later).

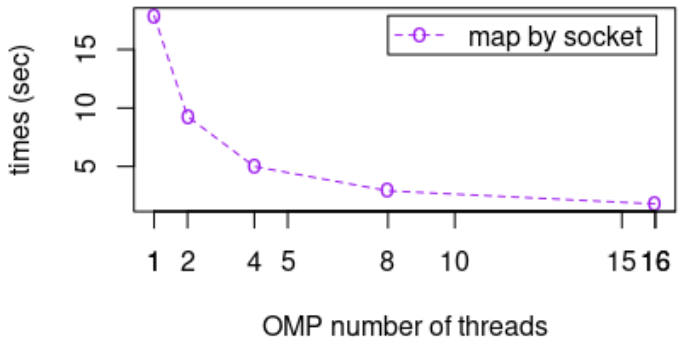
Note that when we consider 65 threads for both MPI processes, then we have to consider 2 distinct no because one node contains 128 cores (less then $65 \cdot 2 = 130$). This fact makes the MPI calls a little slower and it is the reason of the strange red right tail of the plot.

We also repeated Example 1 with a fixed $nMPI = 8$ with a -map-by numa option: we got (consul the GitHub repository) is a very smooth scalability when we increase $nOMP$.

Finally, we have tried to change binding policies (between 'spread' and 'close') and also places (choos
'threads'). The times measured are identical, however when we have chosen 'threads' as places and 'close'
as binding, the time is 2% less, we think because the shared memory among threads is the nearest possi
and this reduces time when threads search for data_image.

Example 2: We try now to increase the size of the matrix. We use 2 MPI processes, with a map by
socket option (from experiments I've seen that the discussion above about the number and location of M
processes is still valid).

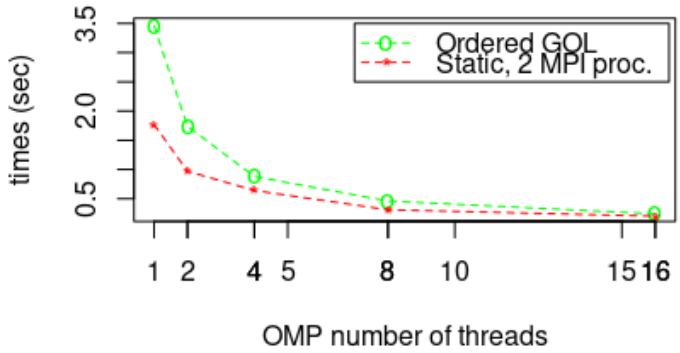
> Size: 10000 $\times$ 10000, $n = 10$, $s = 5$, nodes: 3 EPYC, MPI processes = 2, OMP_PLACES=cores,
> OMP_PROC_BIND=close



We can see from the plot above that if we double the number of threads, then the time almost halves.
is possible because MPI_Scatterv and MPI_Gatherv calls take about 0.02 seconds, while the two nested 'f
loops take 0.8 seconds. In this case, by Amdahl's law, increasing the number of OMP threads is the right
choice because the improvable time is a large fraction of the total.

If we change threads policy, for instance we set OMP_PROC_BIND as 'spread', then the times we
measure are very similar to the 'close' case, MPI_Gatherv call is slightly slower, 'for' loops and MPI_Scatte
remain constant. We have also tried to set OMP_PLACES as 'threads' with both 'close' and 'spread' policies,
and results are similar. Only with 'spread' the times are slightly increased because the 'for' loops are
slower in accessing shared memory as they are far from each other within the same socket (as with 'cores'
places this effect was not measurable).

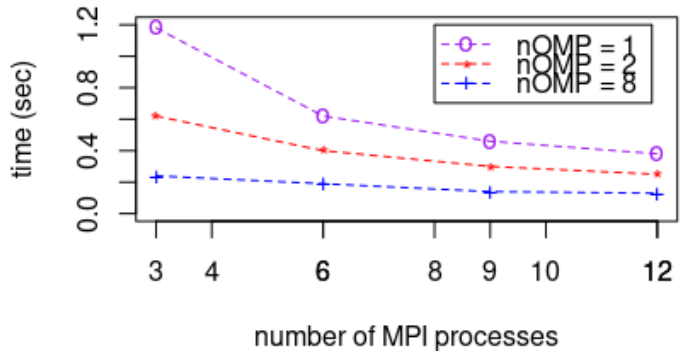
Ordered Game of Life (GOL): The final OMP scalability test it is for the ordered Game of Life, that
is intrisincally serial, so only a OMP scalability analysis is meaningful. The input datas are the same of
Example 1. From the plot below we can see that time almost halves when we double the number of thread
however, the static game with 2 MPI processes almost halves the times (as expected).



1.3.2 Strong MPI scalability

Example 1:

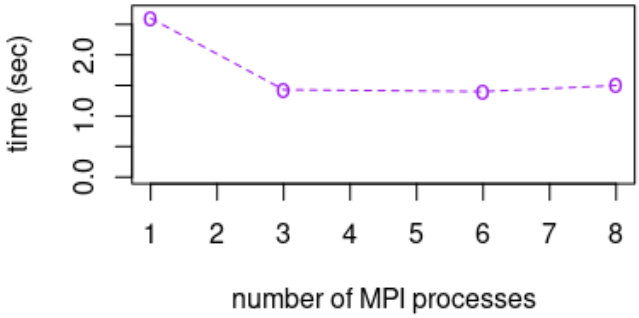
> Size: 1000 $\times$ 1000, $n = 100$, $s = 50$, nodes: 3 EPYC, map by numa, OMP_PLACES=cores,
> OMP_PROC_BIND=close



The description of the plot above starts from the times recorded with one single MPI process with one sin thread: an overall time of 3.55 seconds (with nMPI = 1, nOMP = 1, shown in figure), with $3 \cdot 10^{5}$ for every MPI calls and $3.5 \cdot 10^{-2}$ seconds the nested 'for' loop. The improvable part is a fraction 99.8%, so we can increase both the number of MPI processes and threads for each process. The results are pretty clear, and we also know when to stop increasing nMPI or nOMP. Indeed, when we increase nMPI also MPI calls are a bit slower, however, what we reach when we choose nMPI = 12 and nOMP = 8 is that 'for' loops take $3 \cdot 10^{-4}$ seconds each, and also MPI calls $4 \cdot 10^{-4}$ to $2 \cdot 10^{-4}$ seconds. Amdahl's law tells us that at this point it's difficult to obtain better performance because the improvable part now is not the largest one.

Example 2:

Size: 10000×10000, $n = 10$, $s = 5$, nodes:3 EPYC, map by numa, OMP_PLACES=cores, OMP_PROC_BIND=close, OMP_NUM_THREADS=16



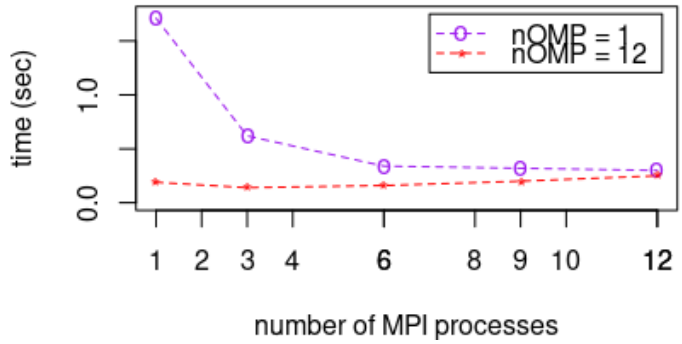
This experiment shows that a high number of MPI processes could be of low utility. Indeed, when nMPI = 1 a single nested 'for' loop takes approximately 0.2 seconds, while an MPI calls 0.015 seconds. However, even with nMPI = 8 the time of the 'for' decrease to 0.04 seconds and MPI calls increase to 0.04 seconds each this situation the improvable time is half of the total, however, the gain in time due to the increase of nM is balanced by the loss caused by slower MPI calls.

Note that in the dataset is also present the experiment on the same input with a -map by socket and *nOMP* = 64, *nMPI* = 2. It takes a total of 1.1 seconds, which is better than 1.5 seconds with the same amount of cores (i.e. map-by numa, *nOMP* = 16 and *nMPI* = 8). This is a natural consequence of a decrease in the number of MPI calls, and it's the best way of doing the experiment.

Example 3:

Size: 100×100, $n = 5000$, $s = 2500$, nodes:3 EPYC, map by numa, OMP_PLACES=cores, OMP_PROC_BIND=close

This time the 2d array is very tiny but it is updated for a long time. With a single MPI process, the nested 'for' loops take 10^{-4} seconds for every iteration. So, increasing nMPI gives us better performance until 'for' loops take the same time as MPI calls. The stationary time is 0.34 seconds (with nMPI = 9), hence, we can scale with respect to MPI processes.



With nOMP = 12 we see nice performance, however we come across a danger when we try to increase nMPI. The reason is simple: the nested 'for' loops are so fast that if we increase nMPI the delay due to slower MPI calls is not recovered by a faster 'for' loop.

1.3.3 Weak MPI scalability

The goal now is: given an initial size, show to run-time behaviour when you scale up from 1 socket (saturat with OpenMP threads) up to as many sockets you can keeping fixed the workload per MPI task.

In order to maintain fixed the MPI workload, we use several increasing input images dimensions $\{(500, 500), (500, 1000), (1000, 1000), (1000, 1500)\}$ with respectively nMPI = 1, 2, 4, 6. The places for MPI processes are socket (of three different EPYC nodes), so we are going to use nOMP = 64.

All the statistics for the experiment are precisely state in Weak_MPI_scalability.csv in the statistics folder. However, the significant conclusion is that even MPI processes has the same amount of work, the overall time increases principally because of increased number of messages that we need to send. Indeed, every MPI_Gatherv and MPI_Scatterv become a bit slower when we increase the size of the input images (and also nMPI).

Given the previous image dimensions and nMPI size, we set $n = 500$ and $T = 250$. The overall times vary from 0.14 sec of the (500, 500) case up to 1 sec of the (1000, 1500) case. If we take a look to the specific time, we note that the 'for' loop time remains more or less constant for every test, while MPI calls becom up to 10 times slower as we increase the size of the input array.

1.4 Conclusions

This report concludes that increasing the number of MPI processes is not always the right choice: it can happen, the bottleneck is due to communication processes, you have to consider that increasing nMPI usually slows the calls of MPI_Scatterv and MPI_Gatherv, and sometimes, especially when the 2d array is very big (time depends mainly on the bandwidth) or very tiny (message latency becomes a bottleneck) could nullify the gain in time for the 'for' loops. In addition, increasing nMPI is useful when the improvable part is a consistent fraction of the total work (as Amdahl's law suggests). We have not seen a significant disadvantage in increasing nOMP.

1.4.1 Further improvement

The code we implemented could be optimized, in particular, the MPI calls. Indeed, the first and the last slices receive every time the whole 2d array, and this could slow down a little bit the communications. Some possible improvements are the following:

- We can use better MPI calls just for adjacent slices, instead of building the image after every iteratio
- In principle, it should also be possible to decrease the memory occupied by the image, encoding it w bool values. Indeed we have only two states (alive or dead), operations should use the bool values and the translation from bools to pgm file should be effective only if snapshots are required.
- we can compress-decompress the sparse matrix for the communication messages if bandwidth is "t low" respect to matrix sizes.

2 Exercise 2

2.1 Introduction

This assignment aims to understand, test and show the performance of dense matrix product (provided b
MKL and OBLAS libraries) running over some nodes of *ORFEO*. In particular,the benchmark includes an
analysis of the scalability in two ways:

- Measuring time and Gflops of executions increasing the size of the matrix at a fixed number of cores
(64 for EPYC and 12 for THIN) for single and double precision. Different threads allocation policies
were also tested.
- Measuring time and Gflops of executions increasing the number of cores at a fixed matrix size for bo
single and double precision.

The algorithm used is called gemm (general matrix multiplication) and is tested only with squared ma
with both float (32 bits) and double (64 bits) precision.

2.2 EPYC and THIN node details

The scalability studies are performed for *THIN* and *EPYC* nodes. We share below the technical information
obtained through the command lscpu:

- EPYC-01: it has two sockets, each equipped with an AMD Epyc 7h12 (a 64 core CPU). Each socket
is divided in 4 numa regions. CPUs max MHz is 2600 and min MHz is 1500. The sum of all cache
memory (both sockets) is the following: L1d is 4MiB; L1i is 4MiB, L2 is 64MiB and L3 is 512 MiB
(only 32 instances instead of 128). Simultaneous multi thread is disabled.
- THIN-007: it has 2 sockets, each equipped with an Intel Xeon Gold 6126 (a 12 core CPU). Each socket
contains just one numa region. CPUs have max 3700 MHz and min 1000 MHz. The sum of all cache
memory (both sockets) is the following: L1d is 768 KiB, L1i is 768 KiB, L2 is 24 MiB, L3 is 38.5 MiB
(2 instances instead of 24). Simultaneous multi thread is disabled.

We want to point out that on /sys/devices/system/cpu/smt of the THIN node there was the file
control with 'on' written in it. It means that SMT is supported by the CPU and enabled. All logical CPUs can
be onlined and offlined without restrictions.

The theoretical peak performance is given by:

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

There exist different values to evaluate the cpu frequency, for the purpose of theoretical peak performan
we decided to use the maximum value since we want to estimate an upper bound. We also found some
discrepancy between web data[1] and those reported with lscpu. In particular, for the THIN node case, the
direct inspection provides 3.7GHz as max and 1GHz as min frequency while on web we founded 2.6GHz a
base with a boost up to 3.3GHz.

The number of floating operations per clock cycle is hard to evaluate since depends on different factor
We estimated it based on a web research[2]. The peak performance amounts to:

- EPYC: $64 \times 2.6\text{GHz} \times 16(\text{float}) = 2662$ Gflops.
- THIN: $12 \times 3.3\text{GHz} \times 64(\text{float}) = 2534$ Gflops.

We'll see later that the peak performance of the EPYC node is almost reached while the peak performanc
of the THIN nodes is way above, probably caused by an overestimation of the upper bound.

2.3 Slurm job

In the following we report an example of slurm job file that we used to submit on Orfeo.

[1] https://www.intel.it/content/www/it/it/products/sku/120483/intel-xeon-gold-6126-processor-19-25m-cache-2-60-ghz/specifications.html

[2] https://www.amd.com/en/products/cpu/amd-epyc-7h12

```
#!/bin/bash
#SBATCH --no-requeue
#SBATCH --job-name="Curaba_test"
#SBATCH --partition=EPYC
#SBATCH --nodes=1
#SBATCH --exclusive
#SBATCH --time=2:00:00

module load architecture/AMD
module load mkl
module load openBLAS/0.3.21 -omp

export code=/u/dssc/ccuraba00/scratch/Foundations_of_HPC_2022/Assignment/exercise
export OMP_NUM_THREADS=64

cd $code
make clean
make cpu

gcc -fopenmp 00_where_I_am.c -o 00_where_I_am.x
rm where_I_am.csv
./00_where_I_am.x >> where_I_am.csv

for i in {0..20}
do      let size=$((2000+2000*$i))
        for j in {1..3}
        do
                echo $size
                ./gemm_mkl.x $size $size $size >> 1_double_mkl_EPYC.csv
                ./gemm_oblas.x $size $size $size >> 1_double_oblas_EPYC.csv
        done
done
```

The executable 00_where_I_am.x is a simple code that writes on the where_I_am.csv file where (which core) each thread is executed. It's just for checking purpose.

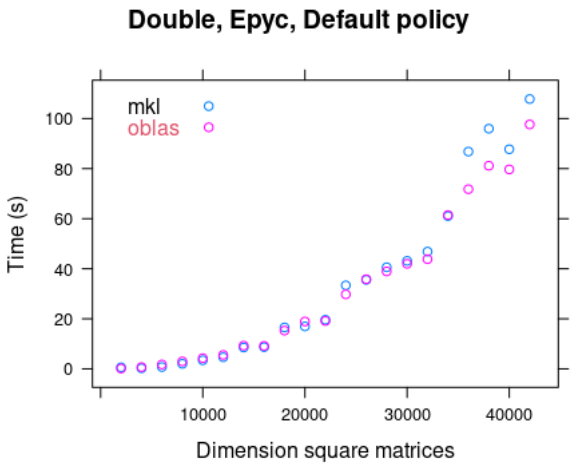
We also highlight that, for each matrix dimension, we executed 3 test and calculate the mean so we c get a more robust estimate of time and gflop scalability.

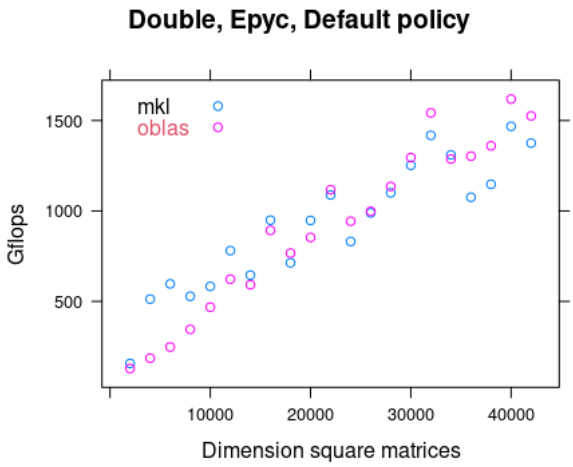
2.4 Results

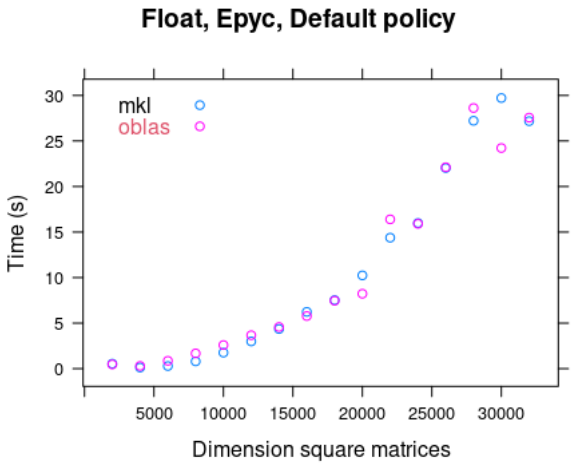
It follows some consideration we provided looking at the graphs:

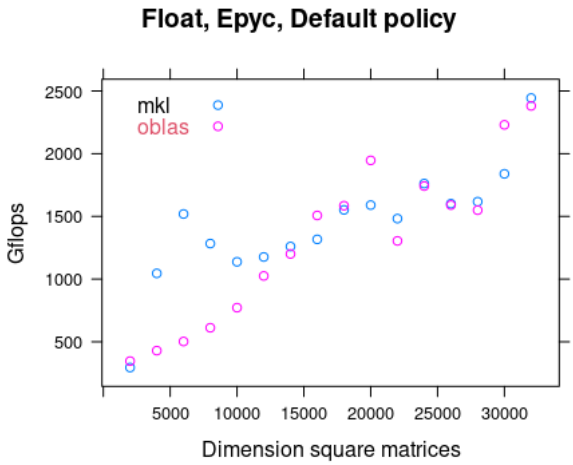
- There are some 'spikes' on the number of gflops graph for *EPYC* node with mkl library. We don't understand it with certainty we would need to go deep into the code and hardware details though we can make some guesses. For instance probably the dimension is used in the code to exploit the full capacity of the cache memory, trying to avoiding cache misses. For instance, growing this dimension can produce a lot more cache misses.
- There's a huge different in the performance of the two libraries for *THIN* node. We discarded various hypothesis to explain such a difference. Most of them where based on the fact that the node could contain non-homogeneous CPU's (some faster than other) but we have observed the same difference also when all the *THIN* node is used. Even if, as stated before, the mkl library can take advantage of SMT, this doesn't seem to be enough to explain such difference, especially on intensive and optimized computational task such as matrix multiplication. The only hypothesis we formulated (that we haven't discarded) to explain such behaviour is the following: how the mkl library code produces a favorable vectorization of data that exploits the computational capacity of the CPUs. It's like performing twice the operations per clock.
- The number of float operations per second is approximately twice the double operations per second as expected. This rule doesn't seem to fit rigorously on *EPYC* node: in fact float operations per seconds tend to be $\frac{3}{2}$ or even less then double operations per seconds. This is due to technical hardware details which can be complicated and we don't handle in this report.

- Since for *EPYC* node the number of gflops tends to increase linearly respect to the square dimension (we remember that the number of operations require by the task increases approximately as a cubi function respect to the matrix dimension) we observe that time increases approximately as a quadr function. On the other hand the *THIN* node reaches the plateau[2] quickly and his time seems to grow up as a cubic function as expected.

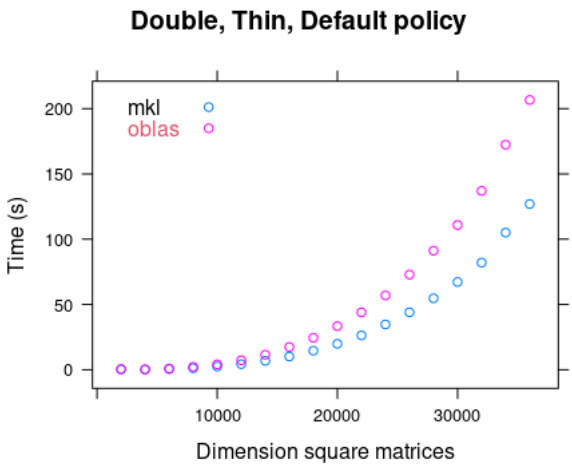


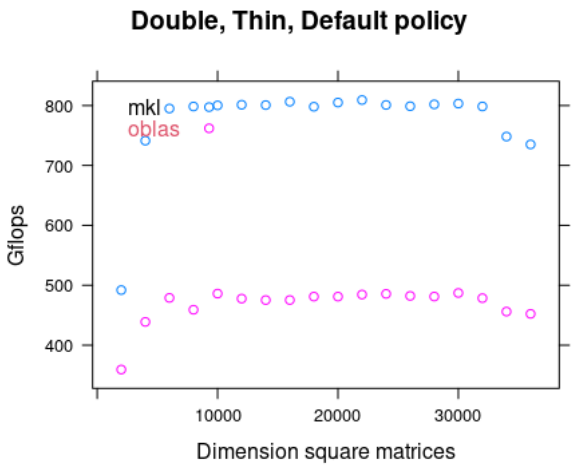


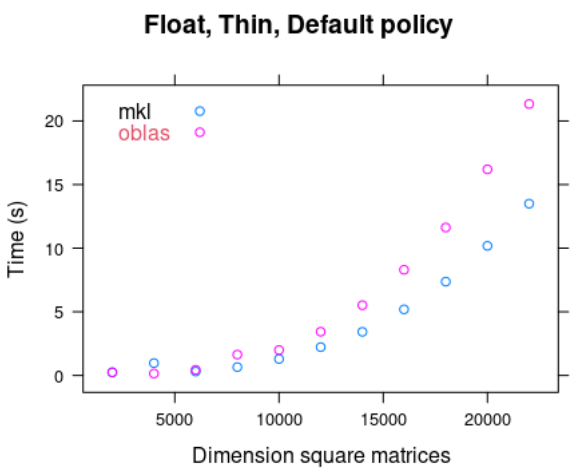


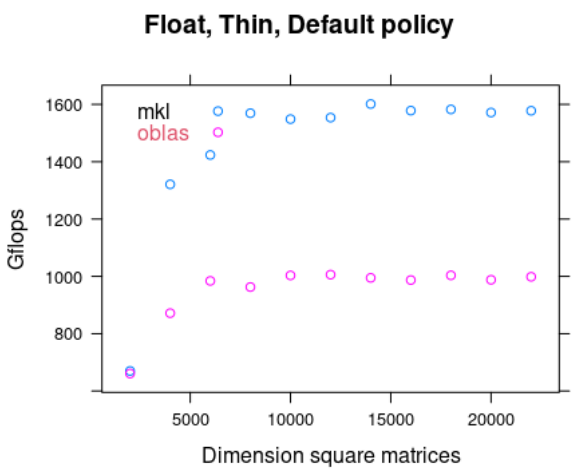


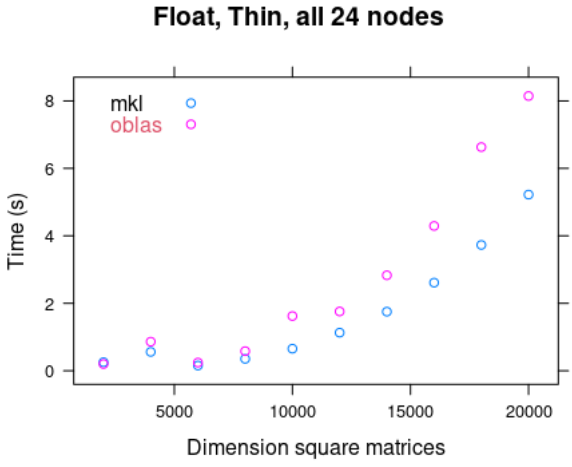
[2] maximum number of floating operation per second

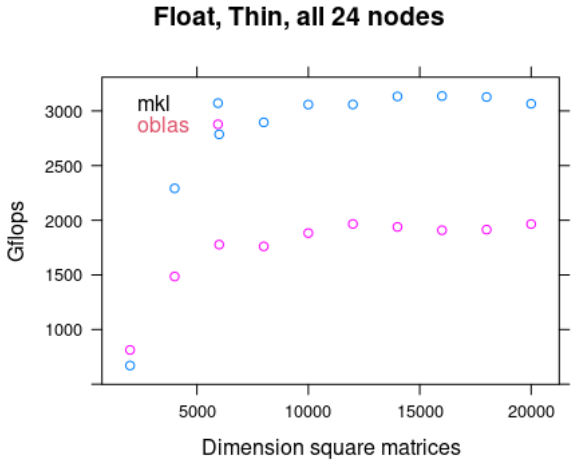

Double, Thin, Default policy
mkl
oblas
Time (s)
0
50
100
150
200
10000
20000
30000
Dimension square matrices

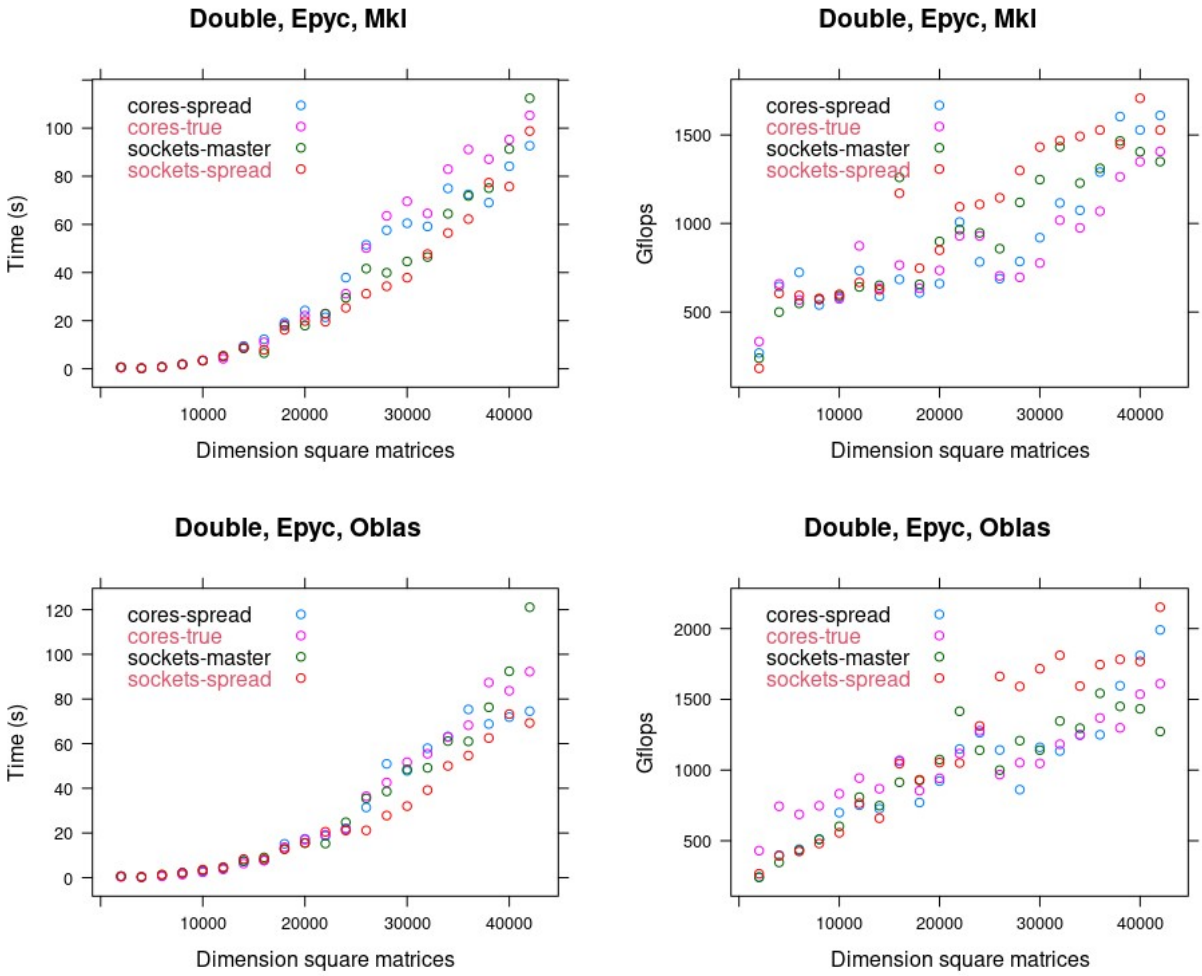

Double, Thin, Default policy
mkl
oblas
Gflops
400
500
600
700
800
10000
20000
30000
Dimension square matrices


Float, Thin, Default policy
mkl
oblas
Time (s)
0
5
10
15
20
5000
10000
15000
20000
Dimension square matrices


Float, Thin, Default policy
mkl
oblas
Gflops
800
1000
1200
1400
1600
5000
10000
15000
20000
Dimension square matrices

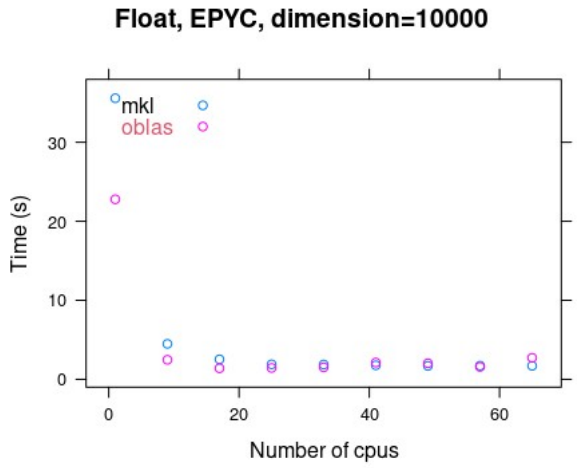

Float, Thin, all 24 nodes
mkl
oblas
Time (s)
0
2
4
6
8
5000
10000
15000
20000
Dimension square matrices

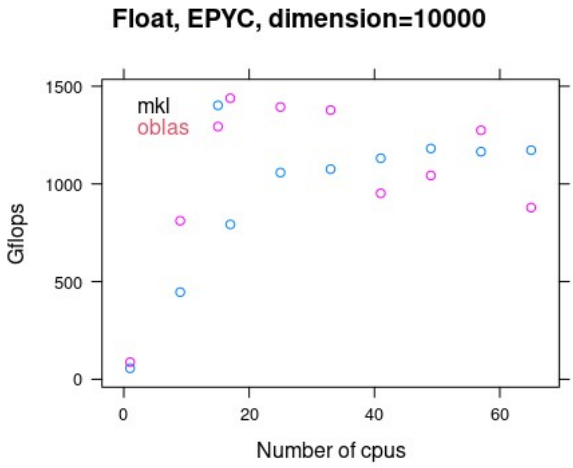

Float, Thin, all 24 nodes
mkl
oblas
Gflops
1000
1500
2000
2500
3000
5000
10000
15000
20000
Dimension square matrices



Giving a look at the different thread affinity policy chosen we can deduce that:

- All policies assign each of the 64 threads at a different core (there aren't constrained policies like co master) so, since the task is computational expensive and doesn't move much the memory, they perform nearly the same.
- The policy sockets-master constrain the threads to use just one node, so it's using less L3 cache me and less dram (only 4/8 *NUMA* regions used). This produce a bit worse performance on average.
- The policy cores-true forbids the migration of threads: this constrain doesn't allow the operative system to optimize the computation and results in a bit worse performance.
- The spread policies (cores-spread and sockets-spread) are nearly the same: both uses all *NUMA* regions available and indeed they perform basically the same.





Looking at the last figures (in particular the gflops graph) we can see that the strong scalability propri
is greatly satisfied until it reaches about 16 cpus (clearly this plateau is case higher the matrices
dimension - higher is the maximum number of CPUs that we can use to minimize the computational time