Foundations of HPC Assignment - Final Report

Davide Rossi and Sara Candussio

21/03/2023

Contents

Game of Life

Introduction

The aim of this exercise is to implement a parallel version of Conway's Game of Life, which is a zero-player game (i.e. its evolution is determined by its initial state, requiring no further input). In our program, the initial configuration is randomly generated and there is the possibility to observe how the world evolves through some snapshots.

The universe or the world of the Game of Life is a two-dimensional grid of square cells, and each of them interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent.

The grid behaves like a torus: the first line will be adjacent to the last line and the second line, and the same is done by columns.

The rules given by the assignment are the following:

- a cell remains/becomes alive if it has 2 or 3 neighbors alive;
- otherwise, a cell remains dead/dies.

There could be different evolution types, according to what we define as "neighbors" of a cell and how we decide to update the world.

In fact, if we decide that we want to evaluate the state of the world "one cell after the other", then we need to do an **ordered** evolution, i.e. the evaluation of a cell depends on the state of its left neighbor and its above neighbors. In this scenario, we are considering an implicitly serial procedure.

Another option would be to evaluate the state of the world "one cell independently from the others". This means that while in the previous case we needed to consider the update of the neighbors before evaluating a certain cell, now we can just look at the initial world state and evaluate independently each cell. At the end, we just put all these updated cells together and they compose the new state of the world. This will be referred to as **static** evolution.

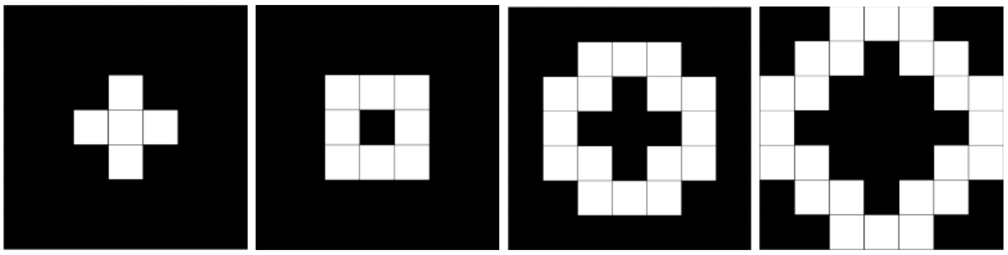


Figure 1: Example of static evolution. Black represents dead cells, white alive ones.

Methodology

In order to implement the Game of Life in these 2 types of evolution, we first needed to establish how we should read and write the snapshots that needed to be taken.

There was given to us a sample file (`read_write_pgm.c`) that performs these two operations and we just used it in order to implement serial reading and serial writing phases. Both of them read or produce a `pgm` file in order to "communicate" with a `void` world structure.

We chose to focus on the parallelization of the initialization of the world and on the parallelization of the evolution of it and to avoid to parallelize reading and writing phases.

Our program works like this:

- if specified by the user in the command line through `-i` command, the world is initialized by `-np x` processes. This consists in subdividing the random generation of the rows of the matrix between all the processes. When all of them have finished, each set of lines is sent to process 0 who will be responsible of writing on a file. This is what does the file `w_init.c`.
- if the user chooses to run an ordered evolution (`-r -e 0` or simply `-r`), this is simply executed in serial. This can be found in the file called `grow_ordered.c`),
- if the user chooses to run a static evolution (`mpirun -np x ./main.x -r -e 1`), this could be executed both in serial (if `x=1`) or in parallel (if `x>1`). If the user tries to run the static evolution with a number of processes greater than the number of lines, a message error is produced, asking for a reasonable number of processes (we refer to `grow_static.c`).

Implementation

Main function and arguments passing: `main.c`

In the `main` function, we wanted to handle the parameters passed to the command line. We chose to define some default values of these parameters: the default world size is 100, the evolution type is ordered, the number of iterations is 50, the snap recurrence is 5, the default file name is `init` and, by default, no action is done.

When we pass different parameters, the default values are overwritten by the new, current values. Due to what was asked by the assignment, if the snap recurrence is `0` then the snap is done only at the end of the game.

If the user asks for action through `-i` command, then the `choose_initialization` function is called.

If the user asks for run through `-r`, it is possible to choose between ordered evolution with `-r e0` by invoking `run_ordered` and between static evolution with `-r e1` by invoking `run_static`. Both the run functions take as inputs the file name, the number of iterations of the game and the recurrence of the snapshot.

World initialization: `w_init.c`

In the `main` function, the function `choose_initialization` is called. This defines a `world` of size `size*size` and of type `unsigned char`. We preferred to use `unsigned chars` instead of integers or double or whatever because they belong to the smallest type that is able to contain both `0` and `255`, that are the outputs of `read_write_pgm.c` functions.

Then, if the number of processes is greater than 1, a parallel function will be called. Otherwise, the program will run a serial version of it.

`initialize_serial.c`: This function takes the file name, the world and the size of a square matrix as inputs. It allocates an array of `unsigned chars` of dimension `size*size` and it fills it in the following way:

```
for(long long i=0; i<size*size; i++){

    int val = rand()%100;

    if(val>70){
            world[i]=MAXVAL; //white = alive
    }else{
            world[i]=0; //black = dead
    }
}
```

We chose the indexes to be `long long` because `long` range is [-2,147,483,647, +2,147,483,647]: too small for our purposes. On the other hand, we did not expect `size` to exceed 2,147,483,647, so we assumed that `long` was enough for it. Then we wrote that world on the `pgm` file through the `write_pgm_image` function and we freed the previously allocated space.

`initialize_parallel.c:` This time, the function takes as parameters also the process rank and the overall number of processes: we wanted for each process to generate its own world part. To do that, we defined a `process_world` array which contains a certain number of world rows previously determined.

To divide the workload, we used the following algorithm:

```
int* rcounts = (int *)malloc(pSize*sizeof(int));
int* displs = (int *)malloc(pSize*sizeof(int));
int smaller_size;
int cumulative=0;
if(pRank==0){
    for(int i=0; i<pSize; i++){

      smaller_size = size%pSize <= i? size/pSize: size/pSize+1;
      rcounts[i] = smaller_size*size;
      displs[i] = cumulative;

      cumulative = cumulative+rcounts[i];
    }
}
```

`rcounts` and `displs` are 2 arrays containing respectively the number of elements that each process should be manage and the index of the starting element for each process.

To fill them, we can establish that if the modulus of the number of lines with respect to the number of processes is greater than the index of a process, then that process will manage one line more. We chose to just compute this arrays with the process 0. For example, if there are 10 lines in a matrix which need to be divided between 4 processes, then the process 0 will have 3 rows, the process 1 will have 3 rows, process 2 only 2 rows, such as process 3. Then these values are multiplied by the number of elements in each row, obtaining (in the case of a square matrix) a `rcounts` array made of `30-30-20-20` elements.

To compute `displs`, we just needed to cumulatively add the number of elements that are assigned to each process, in order to obtain the index of the first element of each process: `0-30-60-80`.

These arrays were broadcasted among the different processes by using:

```
MPI_Bcast(rcounts, pSize, MPI_INT,0, MPI_COMM_WORLD);
MPI_Bcast(displs, pSize, MPI_INT, 0, MPI_COMM_WORLD);
```

Focusing on a single process, we generated in parallel (through OpenMP) a random seed for each MPI process and we divided the workload among different threads. After this, we gathered the generated set of lines to the process 0 by using:

```
MPI_Gatherv(process_world, rcounts[pRank], MPI_UNSIGNED_CHAR, world, rcounts,
displs, MPI_UNSIGNED_CHAR, 0, MPI_COMM_WORLD);
```

This returns to the `choose_initialization` function and, if the rank of the process is 0, these parts are printed in a snapshot.

`grow_ordered.c`

This is the first evolution mode and it consists in waiting the update of the previous cells before evaluating a new one. Due to its serial nature, we chose to just implement it in serial.

When `run_ordered` is called, the world is read from the file called `filename` (an input parameter) and it is saved on a `world` array of `unsigned chars`.

Then a grow function is called on it, and it performs a number of iterations equal to the ones specified by the user (`times` variable, another `run_ordered` input parameter). At each iteration, the `update_world` function is called and the snapshot is taken only if `i%snap==0`.

The `update_world` function has the aim of selecting a cell, look at its neighbors and establish whether the unit must live or die. Since the world grid is a torus, we needed to manage the edge cells by defining as neighbors the cells in the opposite places in the matrix.

For example, in a 4x4 matrix, the cell is the 0 position has as neighbors the cells with indexes: 15, 12, 13, 3, 1, 7, 4 and 5.

This can be done by the use of modules or by a branch conditions.

We used modules in the following way:

```
for(long i=0; i<sizex*sizey; i++){
    long left = sizex*(i%sizex==0);
    long right = sizex*((i+1)%sizex==0);
    long square = sizex*sizey;
    long up = square*((i-sizex)<0);
    long down = square*((i+sizex)>=square);

    unsigned char status=(
          matrix[i-1 + left]                  // west el
        + matrix[i+1 - right]                 // east el
        + matrix[i-sizey + up]                // north el
        + matrix[i+sizey - down]              // south el
        + matrix[i-1-sizey + left + up]       // north-west el
        + matrix[i+1-sizey - right + up]      // north-east el
        + matrix[i-1+sizey + left - down]     // south-west el
        + matrix[i+1+sizey - right - down]    // south-east el
        )/MAXVAL;

}
```

This algorithm applies a correction to each cell that avoids a branch condition. If the element is on the "border", then the algorithm will show a positive value for at least one of the "correction variables" `left`, `right`, `up` or `down`.

In fact, if the first element of a 4x3 matrix is considered, its neighbors will be determined by: 3 (west element: `left=sizex=4`), 1 (east element), 8 (north element), 4 (south element), 11 (north-west element), 9 (north-east element), 7 (south-west element) and 5 (south-east element).

Another algorithm to evaluate the world is the following:

```
double invsizex = 1.0/sizex;
double invMV = 1.0/MAXVAL;
for(unsigned int k=0; k<sizex*sizey; k++){
        long col = k%sizex;
        long r = k*invsizex;

        long col_prev = col-1>=0 ? col-1 : sizex-1;
        long col_next = col+1<sizex ? col+1 : 0;
        long r_prev = r-1;
        long r_next = r+1;
        int sum = world[r*sizex+col_prev]+          // west el
                  world[r*sizex+col_next]+          // east el
                  world[r_prev*sizex+col]+          // north el
                  world[r_next*sizex+col]+          // south el
                  world[r_prev*sizex+col_prev]+     // north-west el
                  world[r_prev*sizex+col_next]+     // north-east el
                  world[r_next*sizex+col_prev]+     // south-west el
                  world[r_next*sizex+col_next];     // south-east el
      sum = sum*invMV;
}
```

In both cases, `status` or `sum` contain the number of dead elements, as defined in `w_init.c`. We decided to test both of them 10 times on matrices of 15.000x15.000 elements in order to establish which of them was the faster method. It turned out that the modules method took a mean of `30.08583` seconds with standard deviation equal to `0.1643053`. The second method instead took only a mean of `15.91139` with a standard deviation of `0.03571708`, turning out to be undoubtely the best algorithm to evaluate the world.

`grow_static.c`

This is the second method we implemented to update the world, and the one that gives the most interesting possibilities with the use of MPI and OpenMP.

This program reads a given file, uses it as initial status and evolves this status for the desired number of iterations, saving a snapshot whenever `iteration%snap==0` and, at the end, returns the time needed for the execution.

The main function of the algorithm is `run_static`, which initializes the MPI environment and all the instruments needed for the execution in the parallel case.

It reads the starting file, ensuring that the number of processes is smaller or equal than the number of world rows: if not, a message error is produced.

A temporary world is created, which exceeds the real world grid because of 2 more rows: one "above" the world, one "below". These two are placed there in order to copy the values of the first true row in the last row and the values of the last true world in the first one.

At this point we needed to divide the workload among processes in the exact same way as we did in `initialize_parallel.c`, but this time we sent for each process also the two additional rows.

As we said before, in a 10x10 matrix divided among 4 processes, 3 rows will be given to process 0, 3 to process 1, 2 to process 2, 2 to process 3. This time there will be 2 additional rows, so the matrix will be 12x10 and the process 0 will take 5 rows, the process 1 also 5, while the last 2 processes will take only 4 rows each. Obviously, these processes' rows will be partially overlapping:

Then a different function to grow the evolution model is called: `grw_serial_static` whether the detected number of MPI processes is 1, `grw_parallel_static` otherwise.

These two functions then call the proper routine to update the world and save the output whenever needed. The update is done by using a second, auxiliary, world: at each iteration, the algorithm uses one world to read the current status, updates the status and saves the result in the other one.

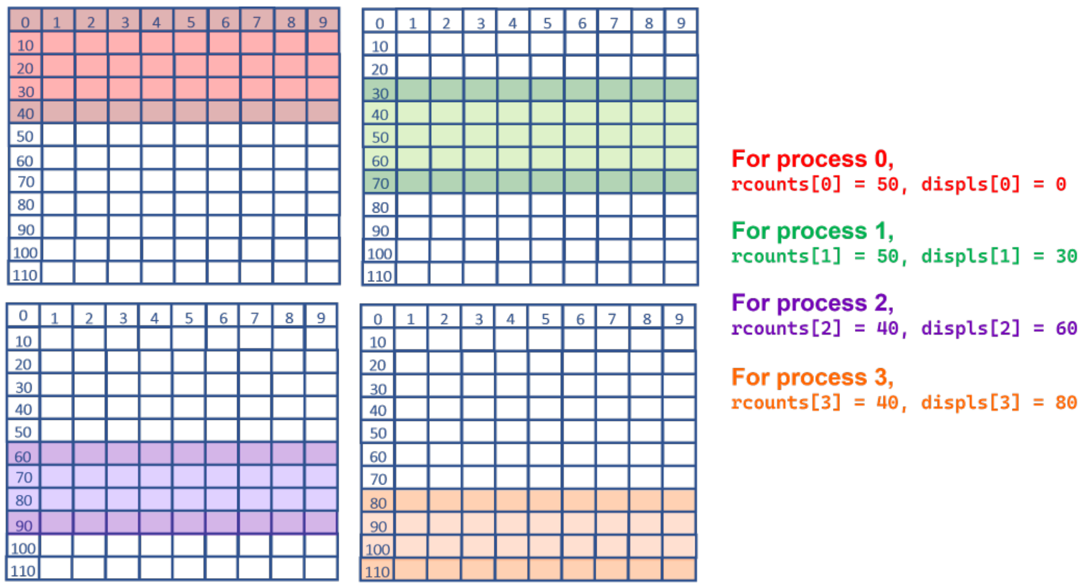

For process 0,
rcounts[0] = 50, displs[0] = 0
For process 1,
rcounts[1] = 50, displs[1] = 30
For process 2,
rcounts[2] = 40, displs[2] = 60
For process 3,
rcounts[3] = 40, displs[3] = 80

Figure 2: Processes overlap

To avoid waste, the updated world becomes the one on which performing an evaluation, and the “old” one becomes the one on which saving the update. A double pointer is used to determine which world is the reading one and which one is the updating one.

```
unsigned char * ptr1=(i%2==0)? new_world: temp_new_world;
unsigned char * ptr2=(i%2==0)? temp_new_world: new_world;
evaluate_world(ptr1, ptr2, size, (long)scounts[pRank]*invsize, times, pRank, pSize, &status, &req);
```

Inside `evaluate_world`, the algorithm used to do the single update is the second one we explained on the `grow_ordered.c` paragraph. After the update on the second world, the first and the last rows of each process need to be exchanged among processes.

At each iteration, each process updates its group of rows (excluding the supporting rows), then `MPI_Isend` and `MPI_Recv` are used to exchange the rows between the processes so that each process will be able to have also the supporting rows updated.

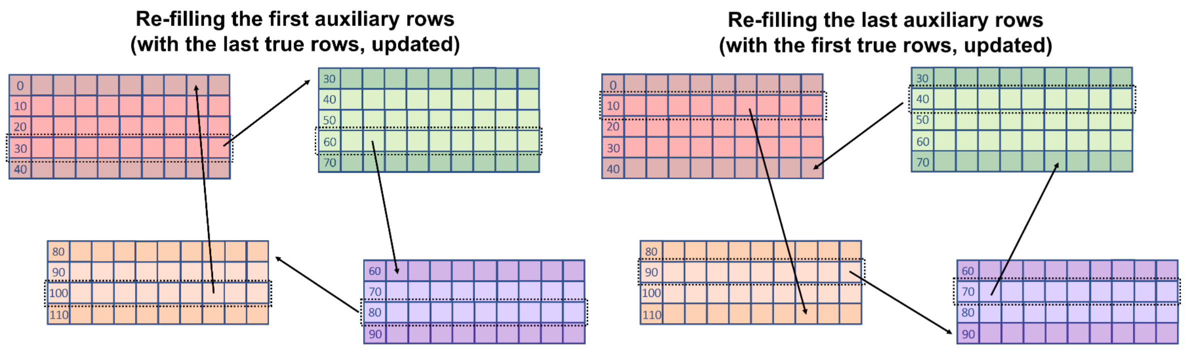

Re-filling the first auxiliary rows
(with the last true rows, updated)
Re-filling the last auxiliary rows
(with the first true rows, updated)

Figure 3: Refilling the matrices

OpenMP is also used to open parallel regions for each MPI process that will further parallelize the work in the loops.

If the number of iteration is divisible for `snap`, then the program gathers all the updated processes worlds and join them in a global world that will be printed in a snapshot.

```
if(i%snap==0){
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Gatherv(&ptr2[size], rcounts_g[pRank], MPI_UNSIGNED_CHAR, world, rcounts_g, displs_g,
        if(pRank==0){
            char * fname = (char*)malloc(60);
            sprintf(fname, "snap/snapshot_STATIC_%03d",i+1);
              write_pgm_image(world, MAXVAL, size, size, fname);
            free(fname);
        }
}
```

An important observation is the fact that we implemented `grow_static.c` in 2 different ways.

The first logic that we followed consisted in the following procedure:

- only the process 0 reads the entire matrix and scatters it through `MPI_Scatterv` among the different processes. To do that, it also needs to evaluate the 2 auxiliary arrays needed in `MPI_Scatterv` as inputs: `rcounts` and `displs`. They need to follow what we showed in Figure 2: each process receives also 2 extra rows. `rcounts` contains the number of elements that need to be scattered to each process, `displs` the index of the first element to scatter to each process; they need to be broadcasted in the same way as we did in the initialization phase. In addition to this `MPI_Scatterv` parameters, we chose to make the process 0 evaluate also `rcounts_g` and `displs_g`, which are the auxiliary arrays required in `MPI_Gatherv` (more details below).
- each process receives its own part of matrix and evolves it for the required number of iterations. To avoid useless operations, we chose to update only the significant rows of each process (i.e. not the first and the last one, which are scattered only to know the state of the neighbors). After this update, each matrix sends and receives the "new" auxiliary rows from the other processes (see Figure 3). If the snap recurrence is met (`iteration%snap==0`), then each process gathers a part its own matrix to process 0: we excluded the auxiliary rows to save the delivery of `2*size*pSize` useless elements to process 0.
- process 0 gathers all the updated set of rows and writes them in a snapshot. Additional rows are no longer required, so `process_worlds` are gathered in a `size*size` matrix.

Due to reasons that we will see in the next section, we tried another implementation:

- each process reads the entire matrix through the read function. Each process will select only its rows (we are referring to Figure 2) and passes its own part to a parallel updating function. `rcounts` and `displs` for `MPI_Scatterv` are no longer required, while the ones needed for `MPI_Gatherv` are still computed only by process 0 and broadcasted.
- each process receives its own matrix and evolves it for the required number of iterations. To avoid useless operations, we chose to update only the significant rows of each process (i.e. not the first and the last one, which are scattered only to know the state of the neighbors). After this update, each matrix sends and receives the "new" auxiliary rows from the other processes (see Figure 3). If the snap recurrence is met (`iteration%snap==0`), then each process gathers a part its own matrix to process 0: we excluded the auxiliary rows to save the delivery of `2*size*pSize` useless elements to process 0.
- process 0 gathers all these updated set of rows and writes them in a snapshot. Additional rows are no longer required, so `process_worlds` are gathered in a `size*size` matrix.

Results & discussion

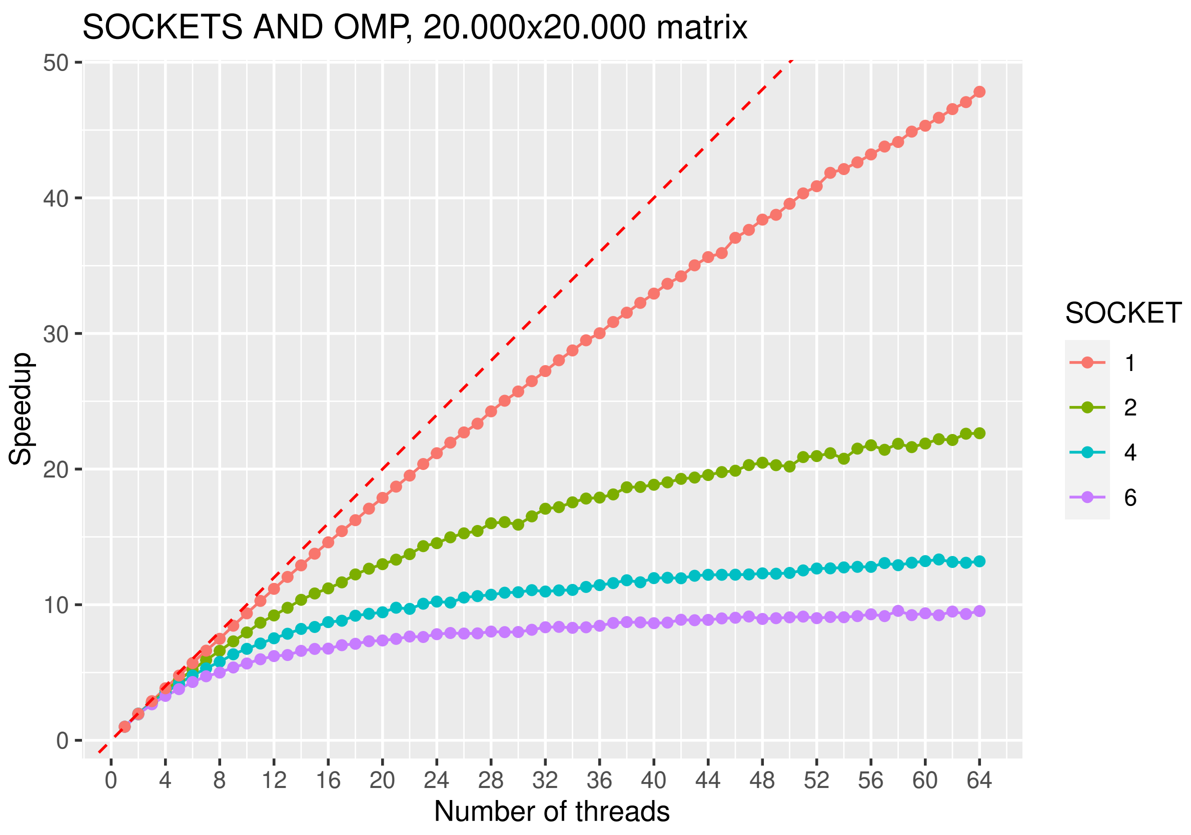
Since the ordered evolution is implicitly a serial procedure, we used only the static evolution to evaluate the scalability of our program. Here we will be discussing:

- OpenMP scalability: we fixed the number of MPI tasks to 1 per socket and we increased the number of threads from 1 to 64 (the number of cores available on the socket) per socket. We used 1, 2, 4 and 6 sockets on 20.000x20.000 and on 25.000x25.000 matrices and we reported the behavior.
- Strong MPI scalability: we used 20.000x20.000 and 25.000x25.000 matrices and we increased the number of MPI tasks by using 2 nodes. We started from 1 MPI task up to $128 \cdot 2 = 256$.
- Weak MPI scalability: we placed only one MPI task per socket by spawning the maximum number of OpenMP threads per socket (64). We increased the matrix size according to the number of sockets involved, in order to keep the workload constant.

OpenMP scalability

To test our program, we had run it on ORFEO on the EPYC nodes with the following settings:

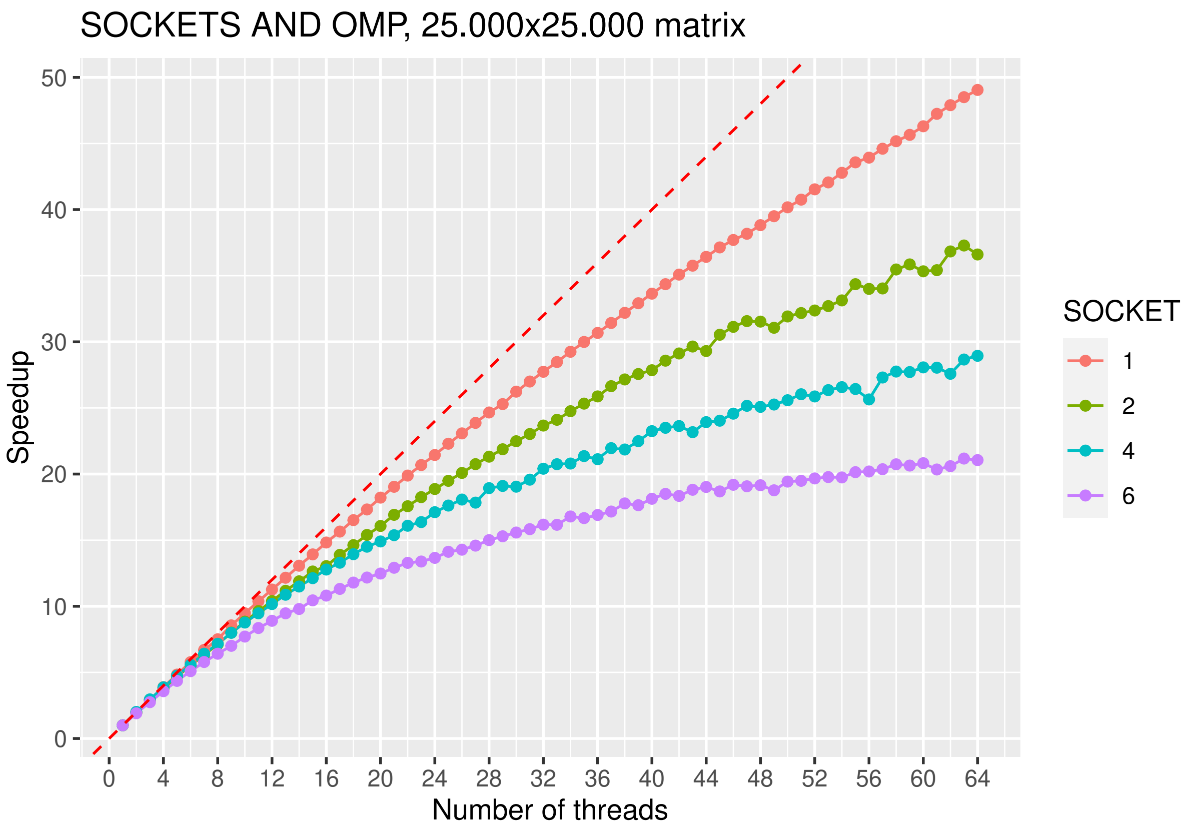
- `OMP_PLACES=cores` and `OMP_PROC_BIND=close`, in order to have the maximum computational power while using cores as close as possible to maximize also the memory usage, and `OMP_NUM_THREADS` going from `1` up to `64`;
- matrices with size 20.000 x 20.000, 50 iterations for each trial, each of them repeated for 5 times in order to take a mean result;
- 3 different runs on 2, 4, 6 sockets, mapping the MPI tasks by socket (while using 1 node) or by core with binding to socket (while using 2 or 3 nodes);



It's clear from this graph that:

- speedup is greater with a smaller number of sockets involved;
- speedup increases when more threads are used.

If another matrix dimension is considered, it is possible to observe that the two previous considerations are still valid, but now the speedup is higher than before.



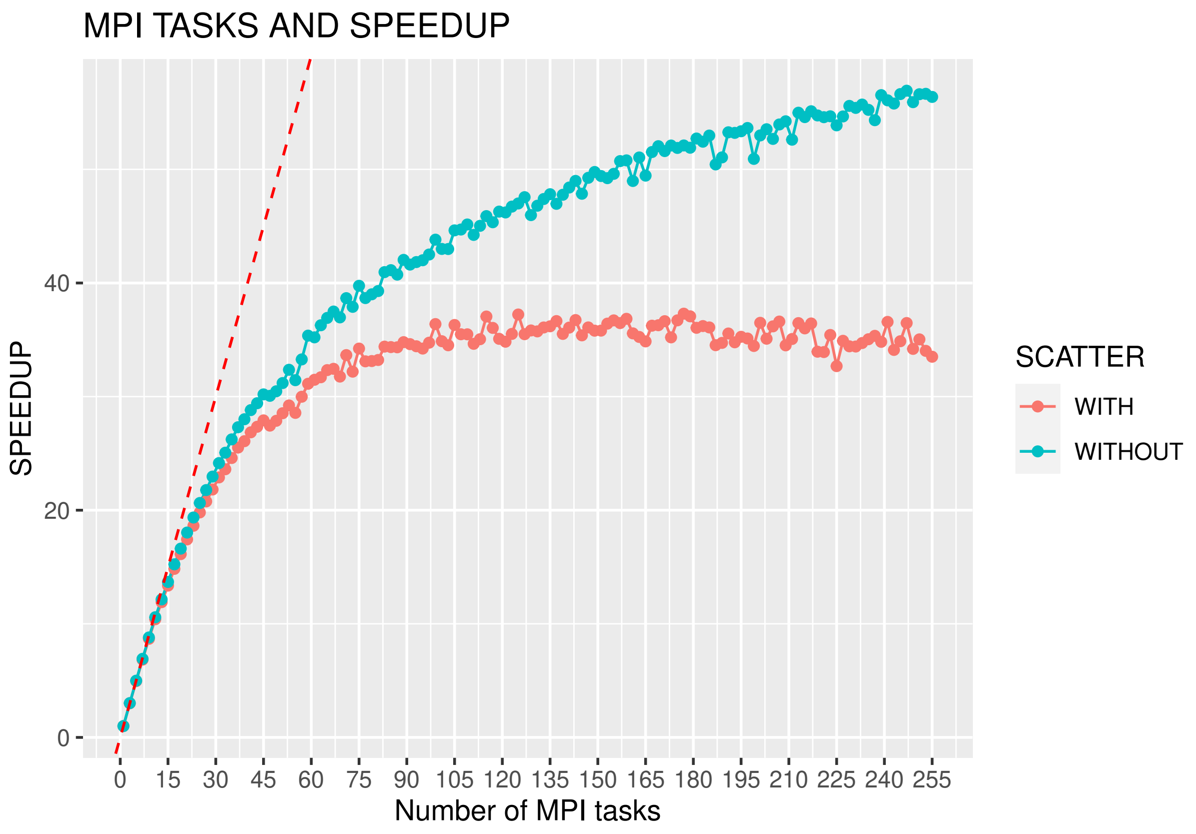
MPI scalability

We tested how `grow_static.c` functions are able to scale when invoked on a growing number of MPI tasks. We had only 2 nodes available, so we decided to test the program with a number of MPI tasks from 1 to 256.

Some implementation notes: we wrote a `run_static` function where it is process 0's business to initialize the temporary world, containing 2 extra rows, and to scatter the correct rows to each process.

We measured with an increasing number of MPI tasks two entities: the overall time necessary to perform the required 50 iterations of the Game of Life, and the time required only by the `MPI_Scatterv` function in each repetition.

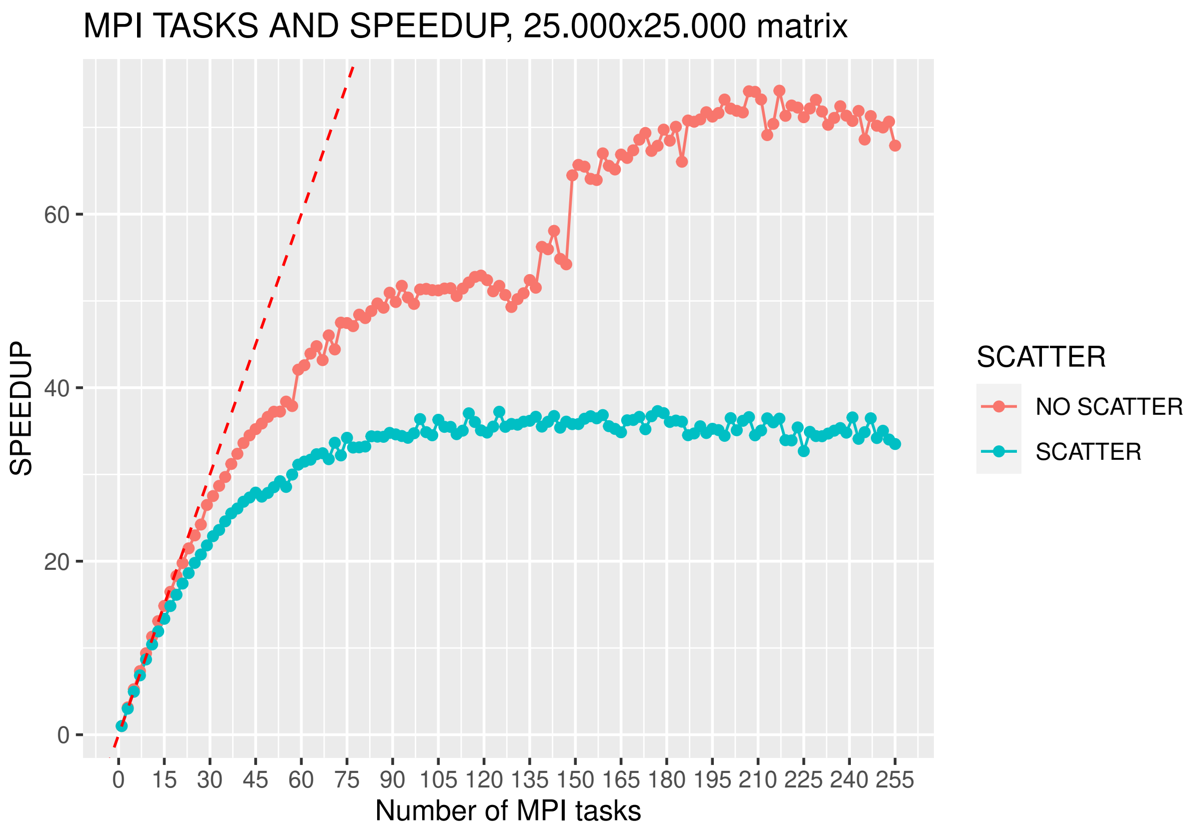
Then we computed the speedup considering the overall time and the time without considering the `MPI_Scatterv` routine:



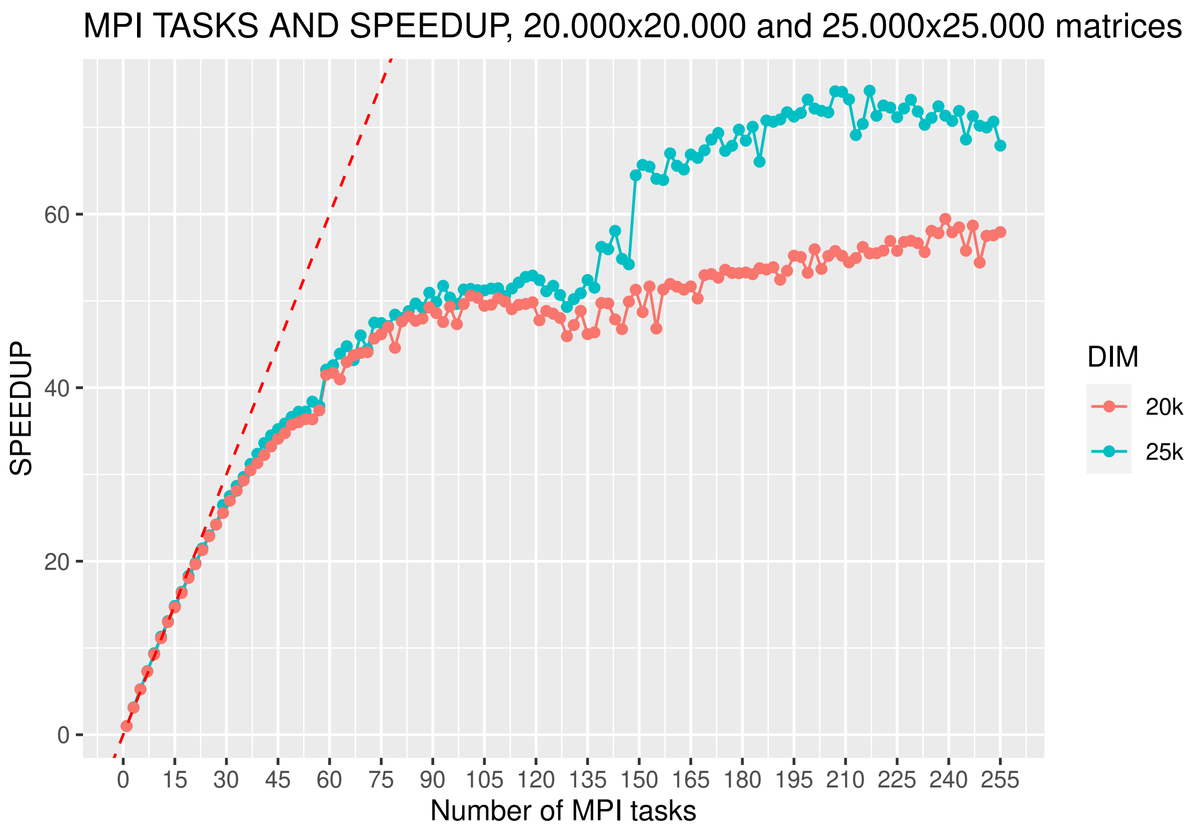
It's very easy to see that the difference between these two lines increases when the number of MPI tasks increases. So, we tried to write a program without using the `MPI_Scatterv` function that clearly turned out to be a performance killer.

The cost of removing `MPI_Scatterv` is that now each process needs to know the entire world before evaluating its own part, just as we described in the implementation part. This implies that the memory usage is greater, but we could expect better performances due to the removal of the previous message passing overhead.

Below, a graph that compares the two implementation choices:



To make sure that the wiggly trend observed in the no-scatter version was just an unfortunate run, we repeated the test with a 20.000x20.000 matrix:



It seems that the first "gap" in correspondence of #MPI tasks = 59 is common above the 2 trials. On the opposite, the descent that can be observed for 25.000x25.000 matrices with almost 140 MPI tasks seems only to be an unfortunate measure.

It is also clear that scalability improves with greater matrices.

Weak MPI scalability

In this section of the assignment, it was required to us to run the code by keeping the workload on each MPI process constant. In the program, the workload is the overall matrix size, which needs to be divided between the MPI processes by giving to each of them a certain number of rows to evolve. In this test, we considered the no-scatter version of `grow_static.c`.

Given that n is the number of elements in each row (i.e. the number of rows in the world matrix), we want that:

$$\frac{n}{number\ of\ MPI\ processes} \cdot n = constant$$

Starting from $n^2 = costant = 10.000^2$ and increasing the number of MPI processes involved, we obtain that with m MPI processes, the `size` of the world should be:

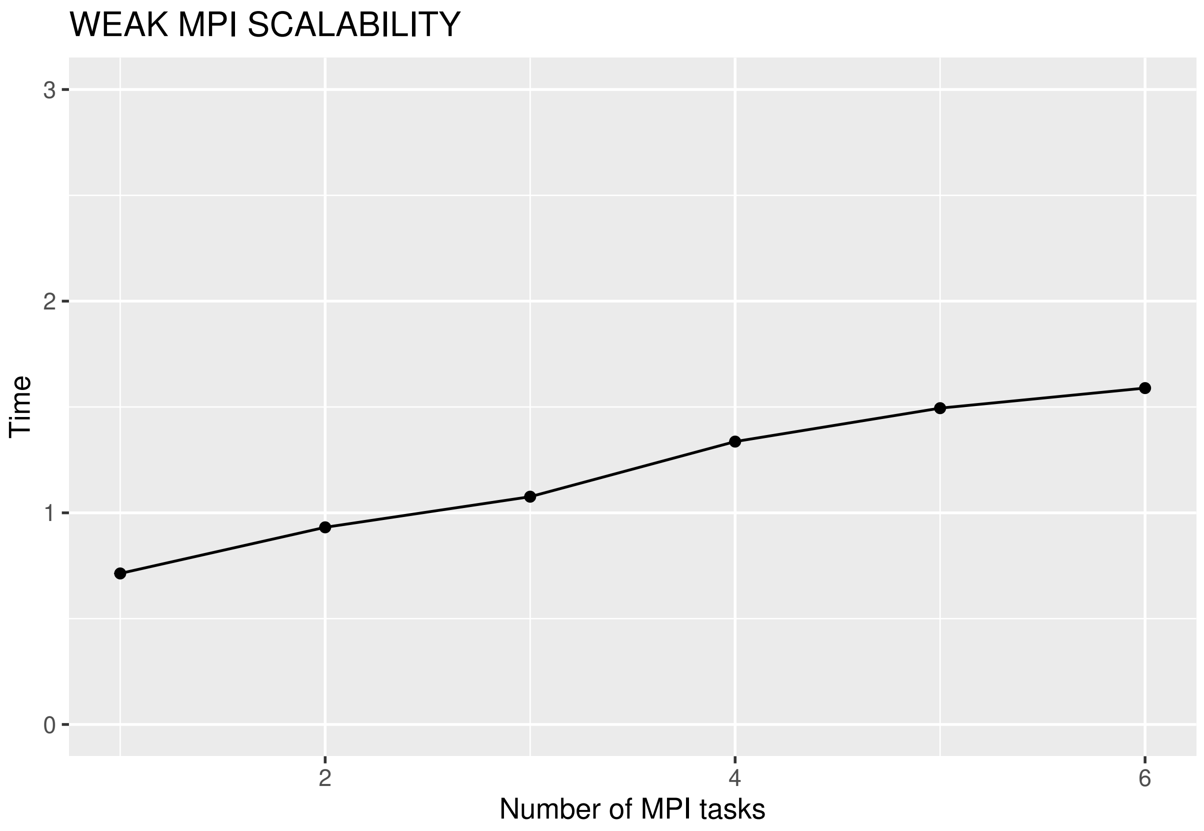
$$n = 10.000 \cdot \sqrt{m}$$

In other words,

# MPI processes	`size`
1	10.000
2	14.142

# MPI processes	`size`
3	17.321
4	20.000
5	22.361
6	24.495

If we plot time needed with different MPI tasks and matrices sizes settings, we obtain:



The time needed to run the parallel Game of Life is almost constant, and the small but constant increase is easily explainable: by increasing the number of MPI processes, an additional communication overhead is introduced.

Conclusions

In conclusion, we are quite satisfied by the overall results obtained by our code: the scalability is pretty good for both OpenMP and MPI.

As it is possible to observe, the speedup could be further improved by considering bigger matrices. An higher number of iterations could also bring to better speedup.

From the strict point of view of the code, a clear improvement could be to parallelize the read and write operations, a possibility that we did not try due to lack of time.

ORFEO tests

INTRODUCTION

The aim of this exercise is to test whether and how different math libraries manage to scale a matrix-matrix multiplication problem. The first request was to test how performances scale when we fix the number of processors used in the computation and we scale the size of the problem. The second request consisted in testing how performances scale when we fix the problem size and we scale the number of processors.

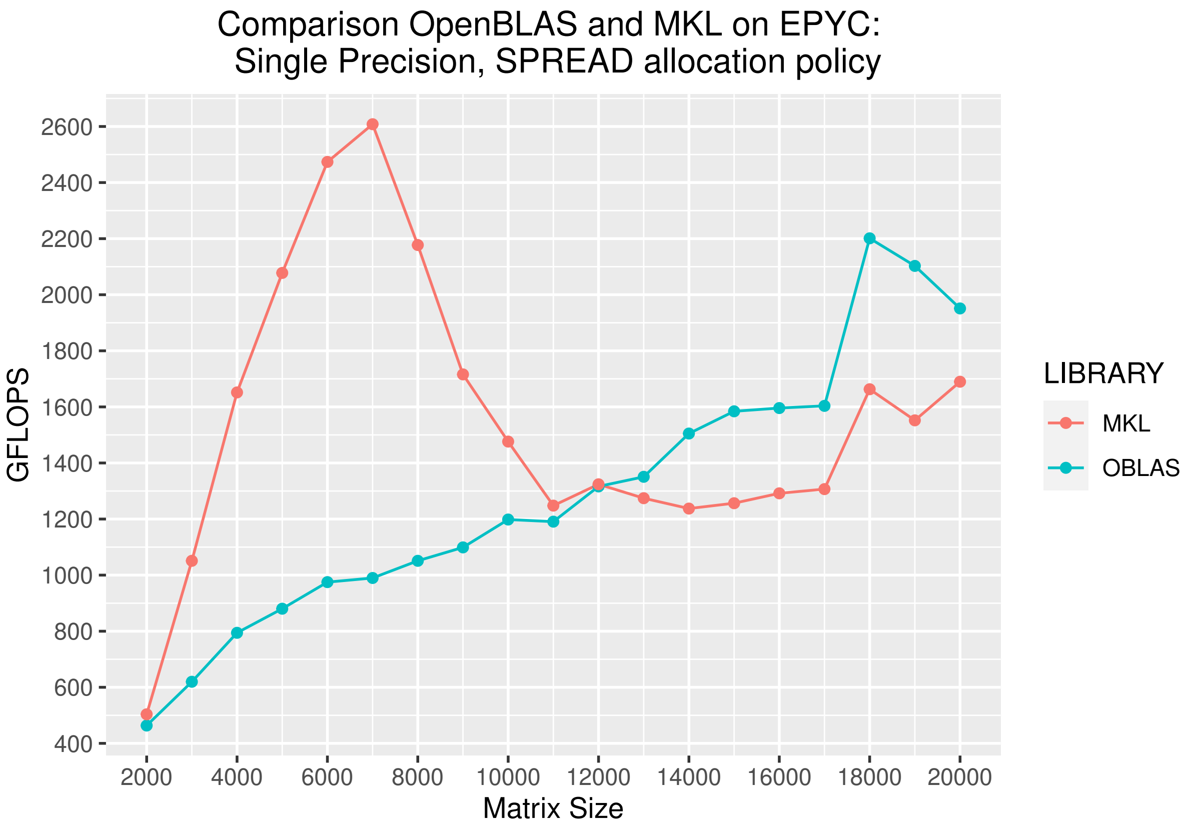
SIZE SCALABILITY

In this first case, we kept the number of cores fixed (it was requested to use only 64 cores if an EPYC node is used) and we tried to scale over the matrix size. It was asked to consider different matrices sizes, going from 2000×2000 to 20000×20000 with a certain step, that we fixed at 1000. We chose to use square matrices of the same sizes: if the first matrix has 2000×2000 elements, then even the second one will have 2000×2000 elements.

We set `OMP_PLACES=cores` and we repeated each trial for 10 times in order to take the mean of the results for each matrix size.

A first binding policy attempt: SPREAD

Single precision

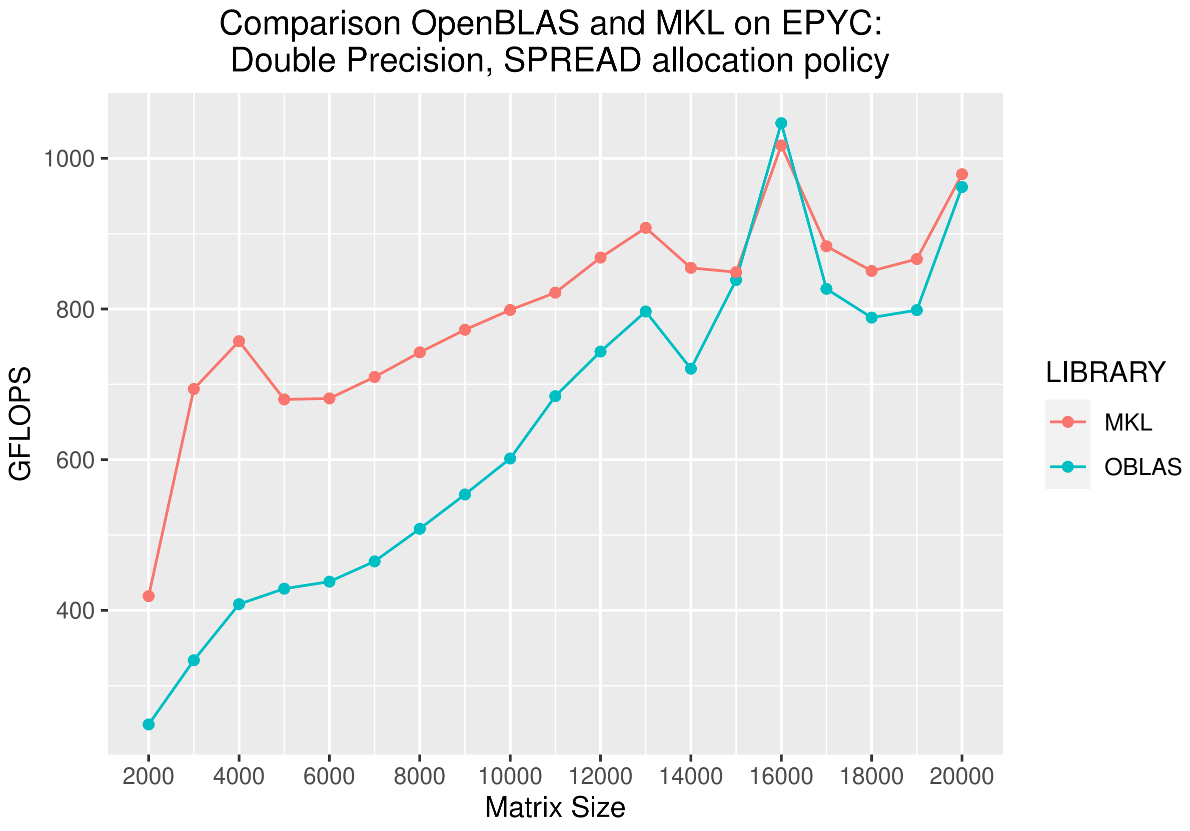


On an EPYC node, `spread` policy in single precision shows different trends with respect to the considered library. MKL seems to perform better with small matrix sizes: it has a peak in correspondence to $size =$

7.000, with 2.600 $GFLOPS$. After that peak, MKL performances decrease and they improve only in correspondence to $size$ = 18.000, though we observe less than 1.700 $GFLOPS$. On the other hand, for OpenBLAS the number of $GFLOPS$ grows almost accordingly with the matrix size: it is possible to observe it decrease only after matrices of $size$ = 18.000.

At a size of 12.000, both OpenBLAS and MKL seem to perform equivalently. Another thing that is in common between these 2 trends is that we observe a sudden growth in terms of $GFLOPS$ both for OpenBLAS and MKL for matrix sizes equal to 18.000, even though both decrease for matrices of $size$ = 19.000.

Double precision



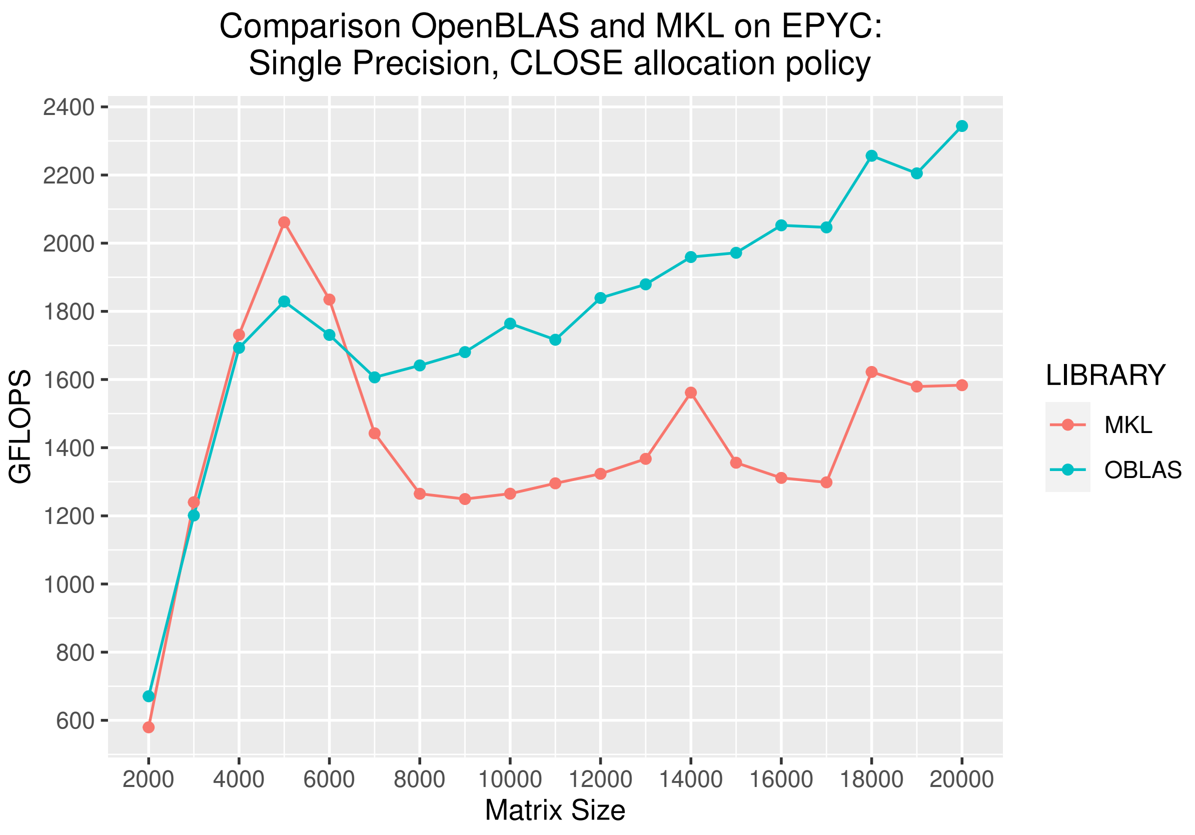
If we now consider a double precision matrix representation, we can spot many differences. First of all, the overall maximum in terms of $GFLOPS$ (1.100) is less than an half the overall maximum that we can observe if we use a `spread` policy allocation (2.600). Then, it's easy to see that MKL shows 2 peaks exactly as seen in the single precision case, but this time they can be found in correspondence to different matrix sizes. In double precision, both OpenBLAS and MKL show almost the same trend, and MKL is, almost always, the more performant. On the other hand, the peak performance in this problem setting can be observed by using OpenBLAS.

To sum up, what emerges from these two analysis is that the expression of matrix elements in double precision almost halves the performance (i.e. the number of $GFLOPS$) that we can reach. If we manage to use single precision numbers, we can state that if matrix size is below 12.000 it's more convenient to use MKL in order to achieve higher performances; otherwise, OpenBLAS would be the best option (at least until a matrix size of 20.000).

A different binding policy: CLOSE

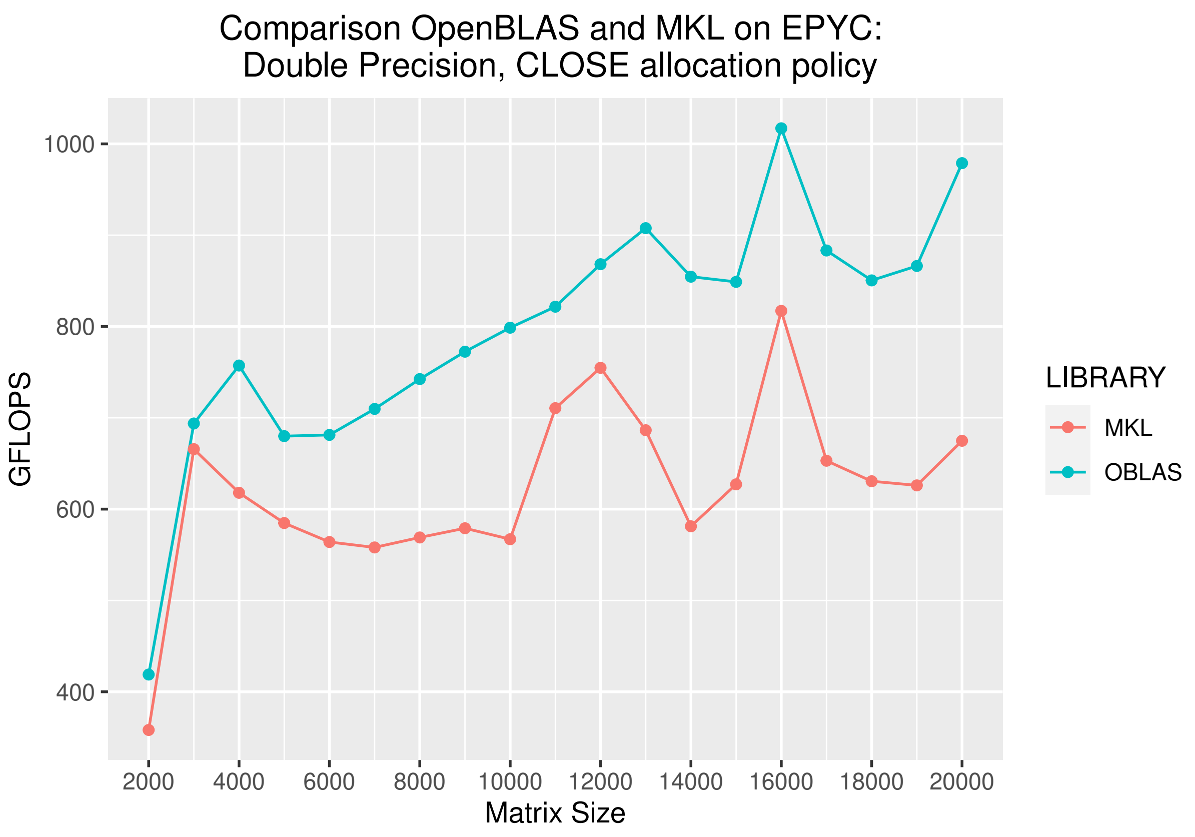
Single precision

In a `spread` allocation policy - the one that we saw before - work is distributed with a round-robin order on cores in different sockets. In this second attempt, we tried with a `close` policy, where work is distributed in cores as close as possible.



This graph highlights that now MKL is better/slightly better than OpenBLAS for matrices of sizes between 3.000 and $\sim$ 6.500; for all other matrix sizes, OpenBLAS apparently outperforms MKL. Despite this, the maximum observed value of $GFLOPS$ is 2.400, while for a `spread` policy allocation it was greater (2.600). What changes in this allocation policy is that now the OpenBLAS performance trend seems to grow accordingly to the matrix size (which is really good), while before we had only a high value for matrices of size under 18.000. Up to what we can observe, `close` appears to be a more appropriate allocation policy accordingly to its ability to better scale if we increase the matrix size.

Double precision



Here again we observe an initial decrease in the performances, but this time it happens for smaller matrix sizes (matrices of 3.000 for MKL and 4.000 for OpenBLAS).

Comparison with the peak performance of the processor

The theoretical peak performance for a single socket on an EPYC node is:

$$P_{peak} = n_{cores} \cdot frequency \cdot \frac{FLOPS}{cycle} = 64 \cdot 2.66Gz \cdot \frac{FLOPS}{cycle}$$

Since we are using AMD Epyc 7H12 (the one that we can find on ORFEO cluster), we can find that $FLOPS/cycle$ are 16 for double precision and 32 for single precision.

This means that:

$$P_{peak}^{SP} = 64 \cdot 2,6\ Gz \cdot 32\ \frac{FLOPS}{cycle} = 5324,8\ GFLOPS$$

$$P_{peak}^{DP} = 64 \cdot 2,6\ Gz \cdot 16\ \frac{FLOPS}{cycle} = 2662,4\ GFLOPS$$

If we compare these peak performances with the maximum ones that we have obtained, we can see that:

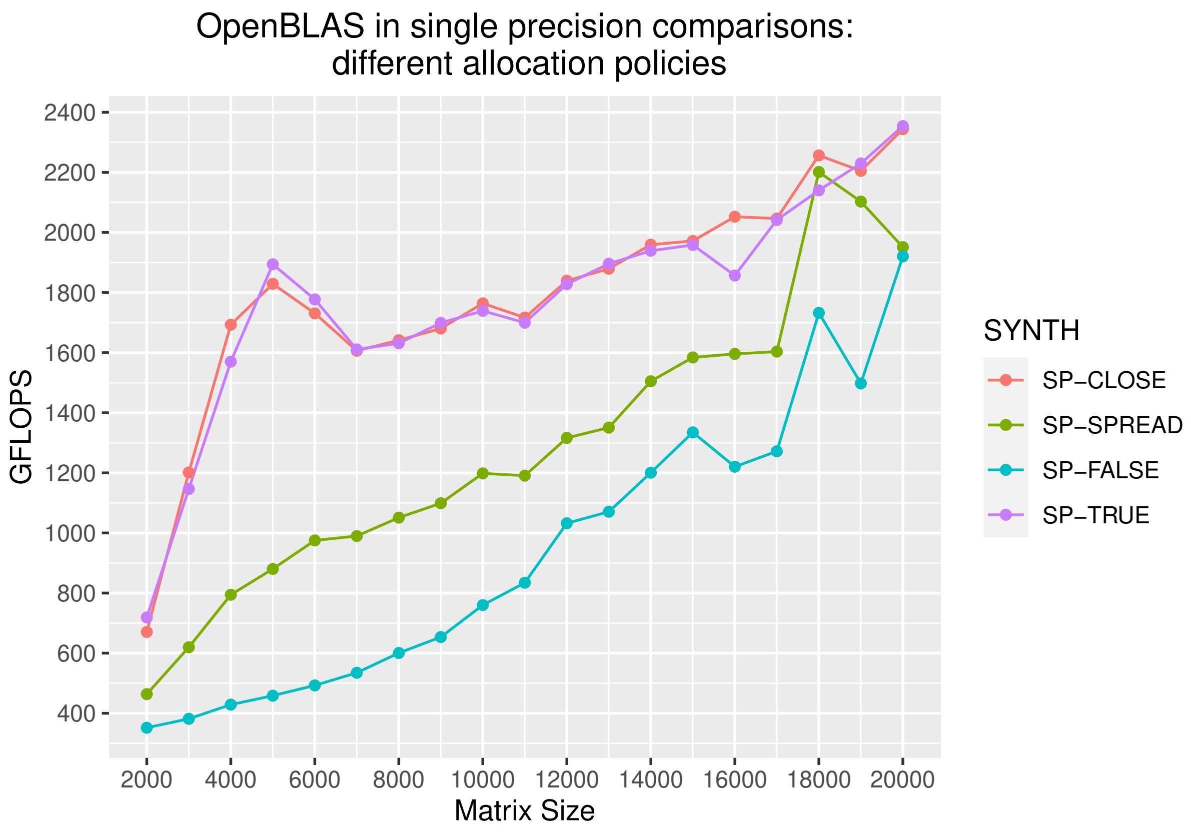
PRECISION	LIBRARY	empirical peak	% of theoretical
SINGLE	OpenBLAS	2.200	41%
SINGLE	MKL	2.600	48%
DOUBLE	OpenBLAS	1.050	39%
DOUBLE	MKL	1.023	38%

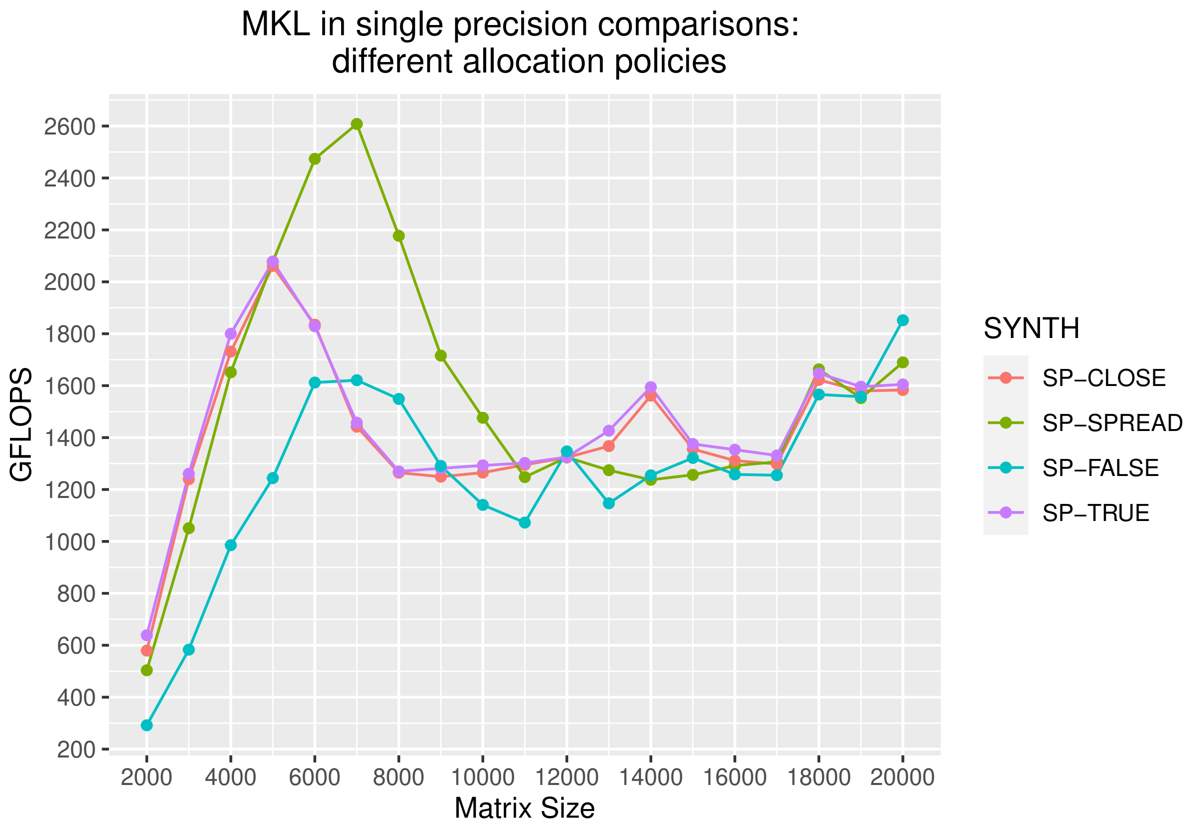
Intepretation of the results and conclusions

In the first case we analyzed (i.e. single precision with `spread` policy) we noticed that MKL had a huge decrease in performance for matrices with size greater than 7.000. This is easily explainable considering the cache size: total cache size in a EPYC node is 584 MiB (512 MiB of L3, 64 MiB of L2, 8 MiB of L1), that is equivalent to 612.368.384 *Bytes*.
The maximum size such that 3 matrices with `float` entries can fit in that value is about 7.100. We can deduct that from that value on, MKL needed to access the RAM in order to store the matrices entries, thus lowering the performances. Since OpenBLAS didn't seem to suffer this problem from the tests that we did, we can suppose that it did a better usage of memory.

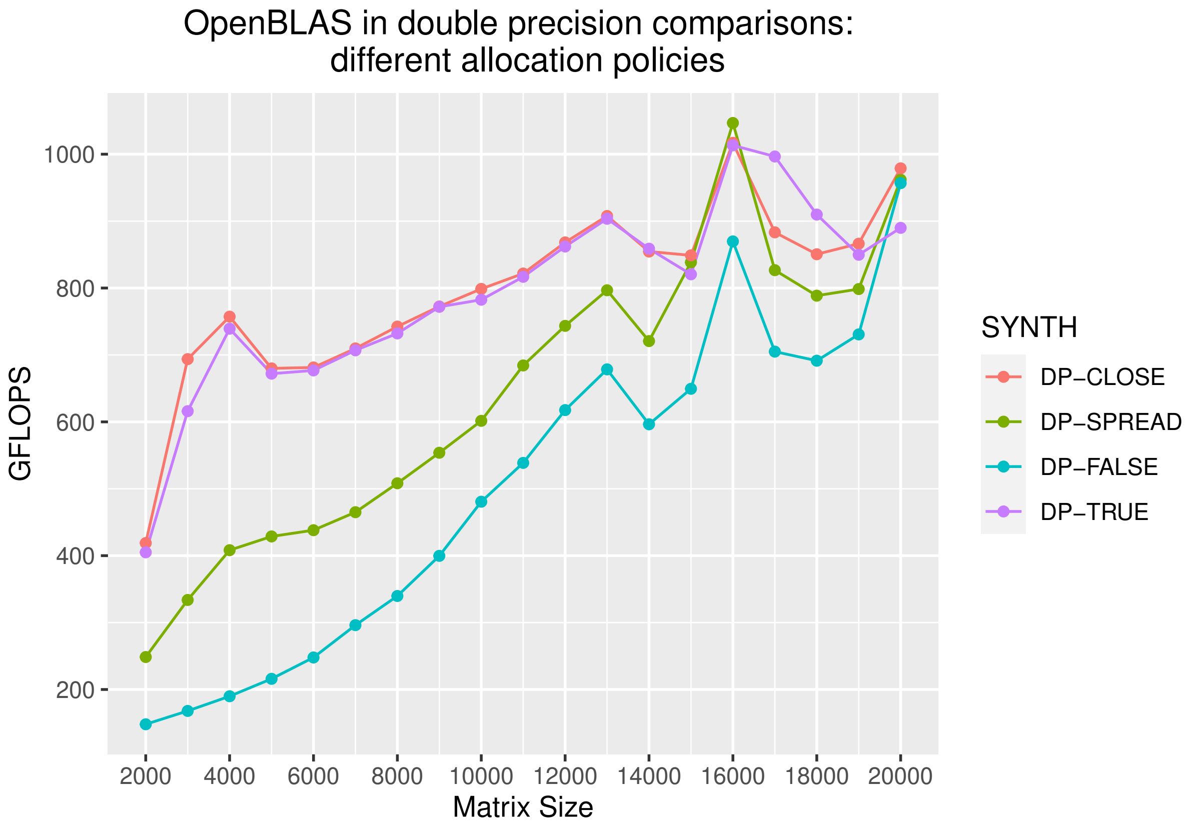
When shifting to double precision (still using `spread` allocation policy), we observed a similar behavior but with different values: with the same calculations as before we can find that the maximum size such that 3 matrices with `double` entries can fit the cache is about 5.000, hence we expect from MKL the same behaviour as before. This is confirmed by data (we refer to the second graph).

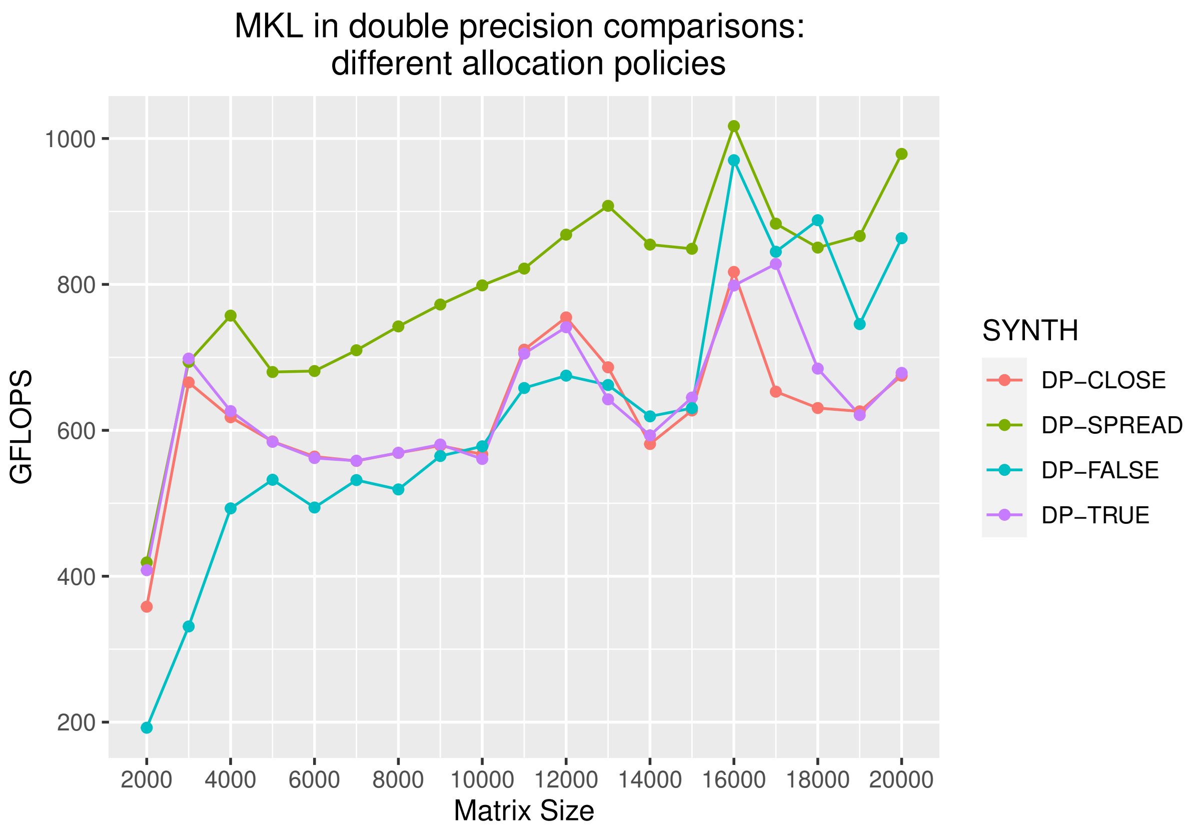
By changing binding policy and shifting to `close`, we noticed the same behaviour as before from MKL. This time that behaviour is also shared by OpenBLAS: both libraries have a decrease in performance for sizes larger than 5.000 for single precision and 3.000 for double precision (third and fourth graph). What distinguishes the two trends is that OpenBLAS seems to recover better, showing a better scaling than the former for all the values on, also having an increasing trend (while MKL seems to stay almost constant).


OpenBLAS in single precision comparisons:
different allocation policies
GFLOPS
Matrix Size
SYNTH
SP−CLOSE
SP−SPREAD
SP−FALSE
SP−TRUE


MKL in single precision comparisons:
different allocation policies
GFLOPS
Matrix Size
SYNTH
SP−CLOSE
SP−SPREAD
SP−FALSE
SP−TRUE

From what we can observe, OpenBLAS manages the memory in a totally different way with respect to MKL in single precision. In fact, while the former has clearly better performances with the `close` policy than with the `spread` one, for the latter this is not so clear: if for medium size matrices (from ≈ 5.000 to ≈ 11.000) the `spread` policy seems to be the best choice, for smaller and bigger matrices it has a trend similar -and also slightly worse in some cases- to the other policies. When we asked for 64 cores in a EPYC node with a `close` policy, we were asking the OS to place them in the same socket before considering other sockets' cores. If we go for `spread` policy instead, cores will be round-robin chosen across different cores in the socket. Since in an EPYC node we can find 2 sockets, twice of L3 cache will be available. This will help libraries with worse memory management.





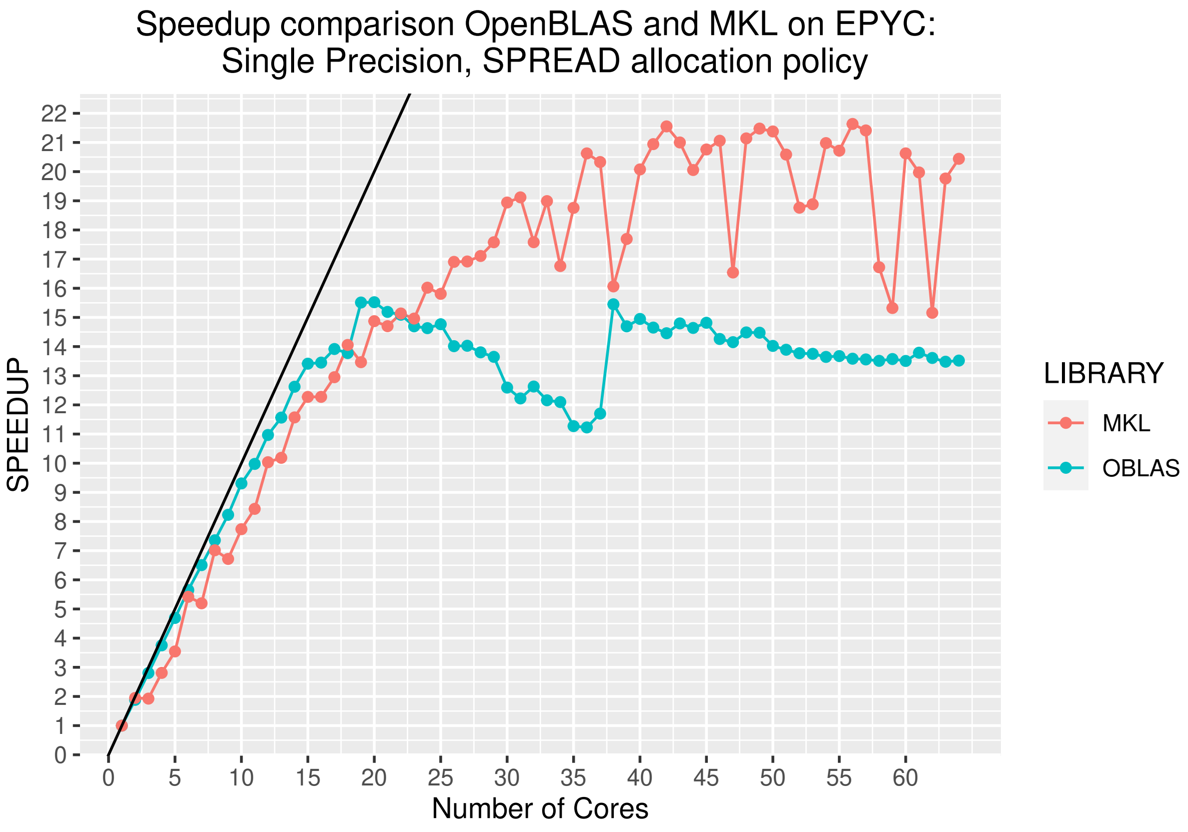
Switching to double precision, OpenBLAS seems to keep a trend similar to the previous case (`close` policy seems to be better than `spread`), even if the difference is less marked in this case, especially for relatively big matrices (from $size = 15.000$ on). On the contrary, MKL seems to perform best with the `spread` policy in a much evident way than in the single precision case.

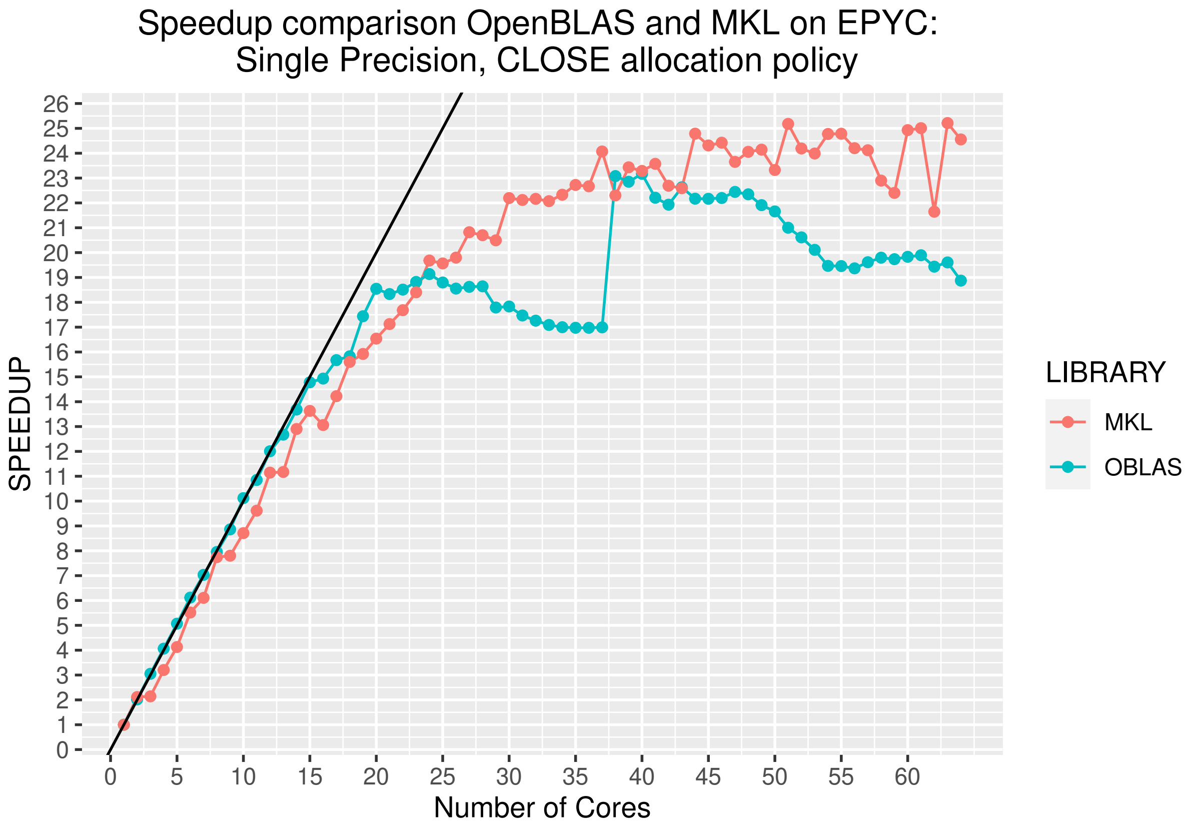
A final note: we also tried the `none` policy, on its two declinations: `false` and `true`. As a result, we obtained that if we let the OS migrate the threads by using the `false` policy, this causes remarkably worse performances (on average, they are worse than any other policy). If, on the contrary, we forbid the OS to migrate the threads, the result is (as we can see in the graphs above) a scaling that is basically equal to the `close` policy, both in single and in double precision for both the libraries.

CORE SCALABILITY

In this second case, we kept the matrix size fixed (12000 x 12000) and we tried to scale over the number of cores (from 1 to 64). Also in this case we used `OMP_PLACES=cores` and we repeated each trial for 10 times and we took a mean of the results.

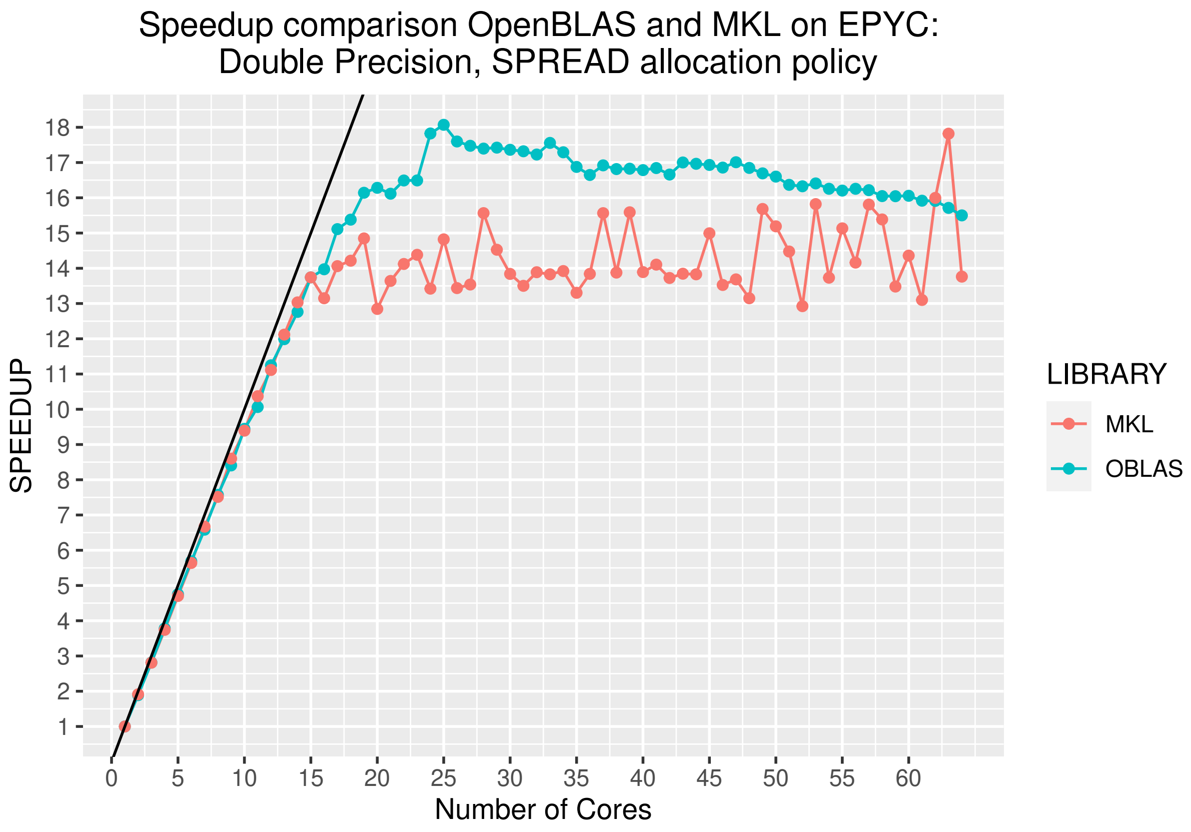
Single precision

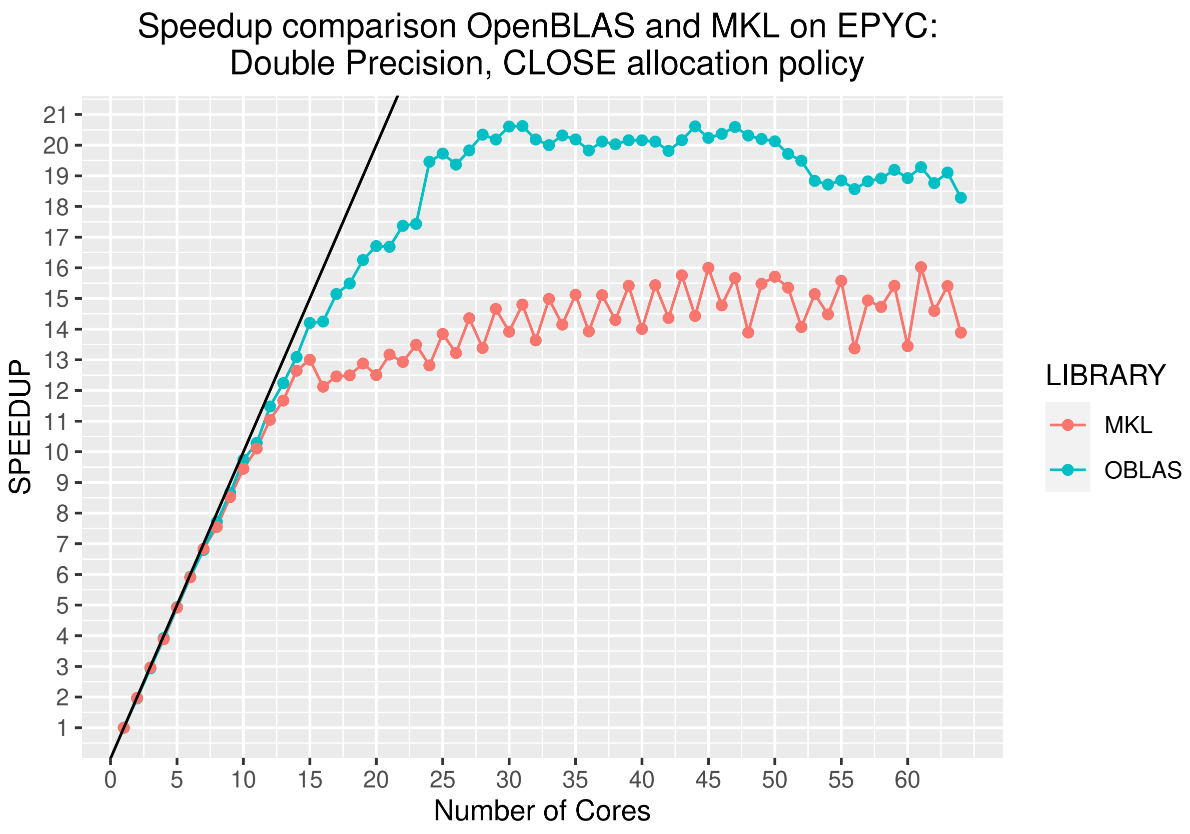




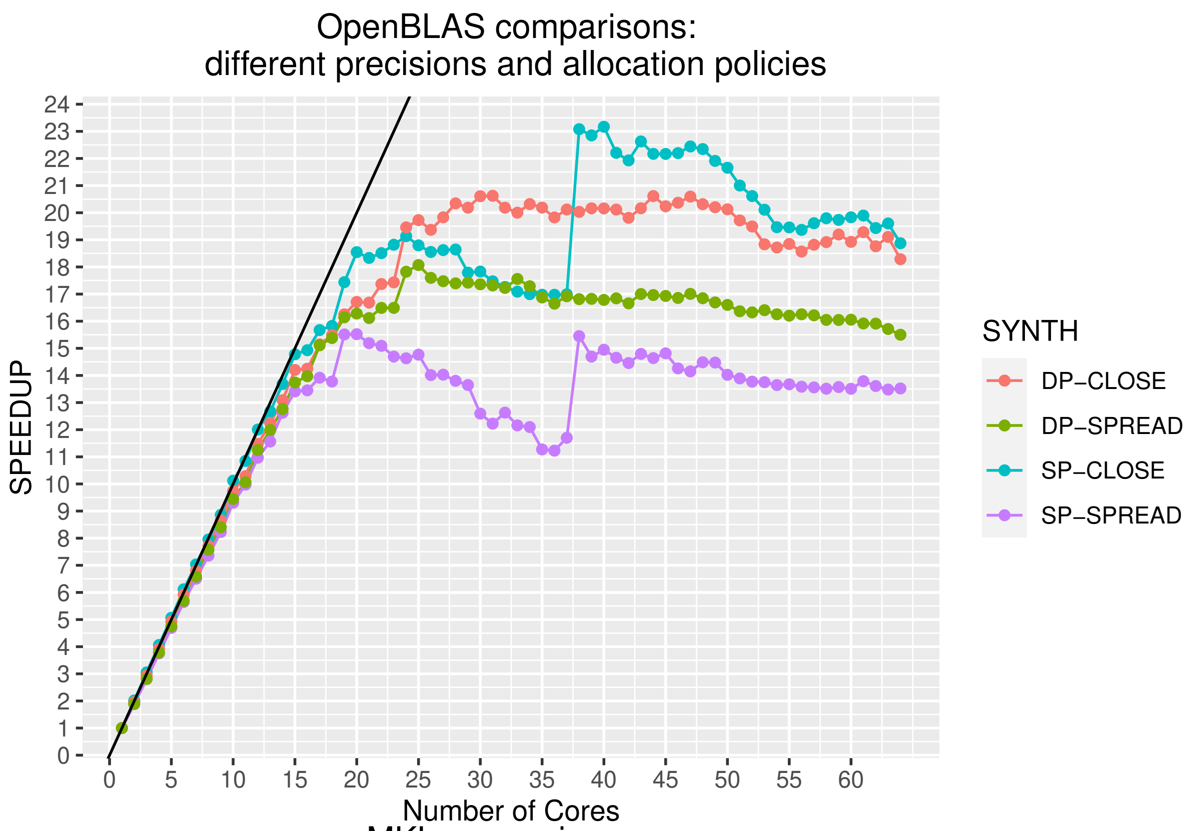
As we can see, also in this case MKL and OpenBLAS seem to behave differently, but we can appreciate a substantial difference for 20 cores on. In fact, while until about 20 cores the speedup (defined as $\frac{T(1)}{T(n)}$) is almost perfect (i.e. coincident to the ideal one) for both the libraries, for greater number of cores OpenBLAS seems to be outperformed by MKL. This is able to maintain a quite good scalability until about 30 cores. This behavior is quite similar both for `spread` and `close` policies, although the latter seems to achieve a slightly better scaling for both libraries, having a highest speedup for almost all the number of cores.

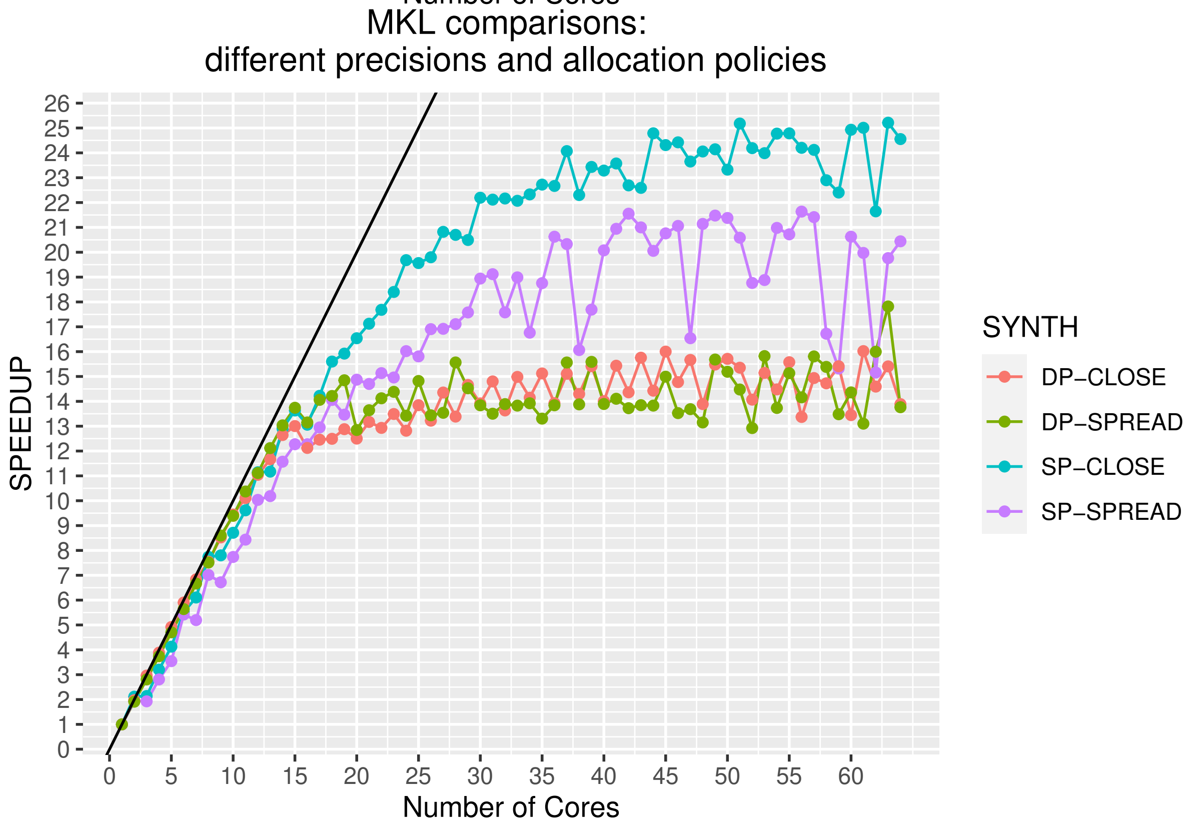
Double precision





If in single precision MKL seemed to outperform OpenBLAS, in double precision we have a opposite situation: scalability is nearly perfect for both libraries until about 15 cores, but while MKL gets worse from that point on (it maintains a more or less constant speedup both with `spread` and with `close` policies), OpenBLAS clearly outperforms the other library, maintaining a still very good speedup until 25 cores, particularly with the `close` policy for which the difference is marked.


OpenBLAS comparisons:
different precisions and allocation policies
SPEEDUP
Number of Cores
SYNTH
DP−CLOSE
DP−SPREAD
SP−CLOSE
SP−SPREAD


MKL comparisons:
different precisions and allocation policies
SPEEDUP
Number of Cores
SYNTH
DP−CLOSE
DP−SPREAD
SP−CLOSE
SP−SPREAD

Summarizing, OpenBLAS and MKL seem to behave very differently: while if we use OpenBLAS the binding policy seems to influence a lot the scalability (`close` policy seems to be more appropriate in general), with MKL this behavior doesn't seem to play a decisive role (although `close` policy still seems to be better in the single precision case).