Project Report

Simone Brusatin

January 31, 2023

1 Exercise 1

The goal of this exercise is to implement Conway's game of life, parallelizing it through an hybrid implementation of *OpenMP* and *MPI*. A detailed description of the game rules and the exercise requirements can be found on the exercise pdf.
Note: in the original game the update rule depends also on the value of the cell itself and not just the neighbours.

1.1 Methodology

The code is written in *C*. The playground for the game is a `.pbm` image. The code deals with the images by converting them to *char arrays*. The code is capable of both reading existing images and writing new images to file.
When running through terminal the main function decides what action to take through passed arguments. There are two main modes:

- *Initialize* generates a random image of the indicated size. It is parallelized only with *OpenMP*.
- *Run* executes the game's algorithm on the provided image, for the desired number of steps, saving at the end or every s steps. It can run in two modes *ordered* or *static*. The ordered evolution is intrinsically serial, so it is not parallelized. The static evolution divides the domain and spreads it among the MPI tasks which ulteriorly parallelizes with *OpenMP*.

To measure the timing of the program I ran it on the *orfeo* cluster, with varying options. I gathered the data and used it to study the results. I looked primarly at the *speedup* and the *total time*.

1.2 Implementation

Here we will look at the more technical aspects of the code. Parallelization explanation refers to the *static* evolution as the *ordered* evolution does not parallelize.

1.2.1 Initialization

The initialization process is done by only one MPI process, generating random numbers and filling the playground. This process is parallelized through an *omp for*, taking care that each thread has its own seed. The schedule is set to static as the workload is balanced, with a minimum size of k (= side of the matrix) to try to minimize read/write conflicts among threads.
The process insn't parallelized through distributed memory because the matrix has to be gathered in a single array to be able to write it to file. The communication overhead would overcome the benefits of multiple MPI processes, as the process of generating random numbers is not computationally intensive.

1.2.2 Data Storing

The playground is stored in **char** arrays. Technically other datatypes could have been used but it would have been sub-optimal as chars only take up a byte of memory.
The arrays are dinamically allocated with `malloc`. A more efficient option would have been `calloc` as the data would have been all contigous in the memory, grating faster access, but there is the risk that there wouldn't be enough space to allocate the arrays, as the playground size is, theoretically, unlimited.
The same matrix is stored with a single `malloc` call. Another option is to store rows separately, allocating each one. I dind't do this because storing the matrix as a "line" simplified the process of MPI communication.

1.2.3 Domain Decomposition

Let n be the number of MPI processes, k be the side of the matrix. Each process is initially assigned $k \ // \ n$ rows (integer division), then the remainder of the division is spread, one row at the time, among the processes, in order, until the remainder is zero. In this way all the matrix is divided and the workload for each process varies, at maximum, by one row. Snippet of the code:

```
// calculate workload = matrix size / network size
my_wl = k / net_size;
if (my_wl * net_size < k) // because we did integer division
    if (my_rank < k-(my_wl*net_size))
        my_wl++; // distribute remainder among processes
```

The workload of each process is then shared to the master process in order to distribute the domain correctly. Technically the master process could calculate it on its own, but it isn't implemented.
A more naive approach might have been to give all the remainder to a single process, but for bigger playground sizes, the workload imbalance would have been too significant.

Each process the procedes to allocate for itself two extra rows, to store the borders of its neighbours' domain, as they are required by the algorithm.

1.2.4 Initial Matrix Transmission

Only the root process reads the `.pbm` file and stores it in an array, which will also be used to store the full matrix when needed. In theory, all process could read the image but they would be accessing the same data all at once, which is sub-optimal, furthermore, it would require to store a matrix of size k^2 for each MPI process.
The root than proceeds to send to the other processes the parts that they require, through `MPI_Send`. This could have been done better: a `MPI_Broadcast` would have minimized the sends done by the root, spreading them among the other processes, but would have required to allocate enough memory to store the full matrix. This could have been tweaked to minimize memory occupation.

1.2.5 Matrix Updating

Each process runs the game on its part of the matrix. A **for** loop is performed on the *non border* cells, parallelized with an *omp for* statically scheduled as the operations is always the same, with minimum size k to avoid conflict among threads.
The neighbours of the cell get calculated and a value is written in a new matrix.
The borders of this new matrix are then transmitted to the "neighbouring" processes, as they are

required to evaluate the next evolution. This is done with a non blocking send. We can use the `MPI_Isend` because we are certain that the data will not get modified until it has been recieved: right after the send, there is a blocking recieve so processes are actually waiting for their neighbours. For extra security, MPI messages are uniquely tagged.
I could have used `MPI_Bsend` but it would have been slower due to the time that it takes to copy the buffer.
The received values are stored in the *new* matrix. When the receving process is finihed, the two pointers are swapped: the *new* matrix is set as the *current* one and viceversa: so on the next iteration the process will write on the old matrix.
Another approach could have been to copy the data from one array to the other, but it would have been way less efficient.

1.2.6 Saving

The program saves at the end, or every s steps if requested at runtime. To do this the root process uses `MPI_Gatherv` to get the full matrix. A simple `MPI_Gather` cannot be used as the amount of bytes transmitted is not the same for every process.

1.2.7 Neighbour Counting

Neighbour counting is implemented in two ways. If a single process is present then the full matrix is viewable to the process. If more than one process is present, only parts of the matrix are present. The difference relies on the first and last row storage.
In single process mode they are treated "normally" but in the multiprocess mode, since the domain is decomposed, they are stored as borders, in other words, there is no jump.

1.3 Results

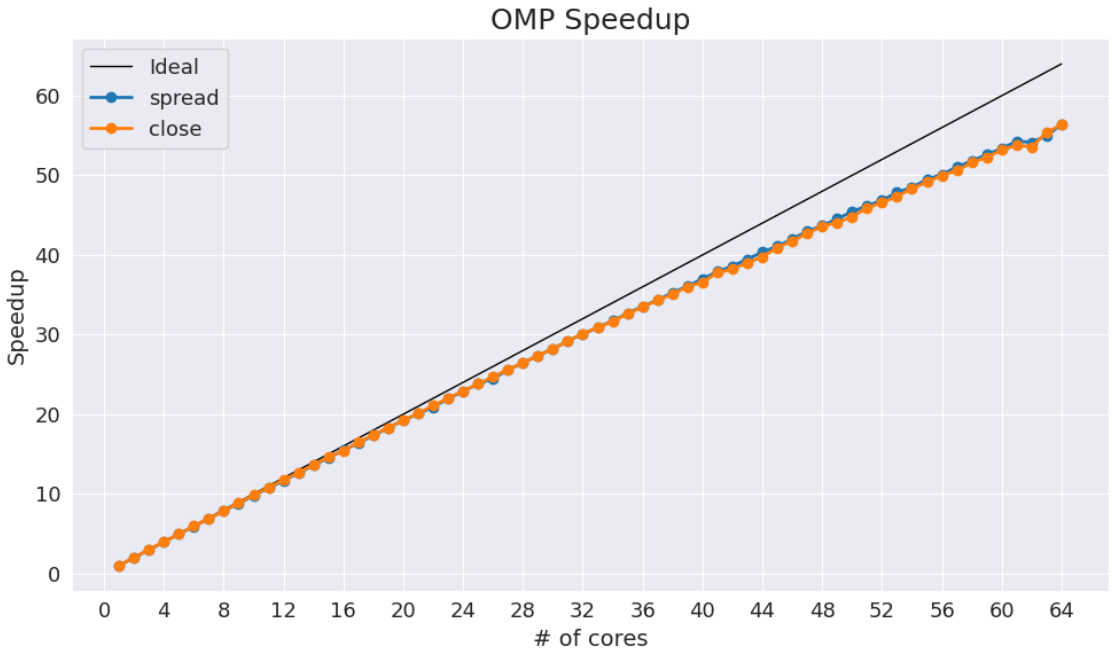
In this section we will look at the performance of the program on *orfeo* epyc nodes. Measurements are performed multiple times and the mean is considered.
What are we measuring? The measurement starts at the calling of the *run* function and ends right before the return. So we are measuring:
- time to read and transmit the initial matrix.
- time to update the matrix for n steps.
- time to gather and write to image the final form of the matrix.

1.3.1 OMP Scaling

Here we will look how the code scales when we fix the MPI processes to 1 and increase the number of omp tasks (`OMP_PLACES=cores`) from 1 to 64 (number of cores present on a socket). The size k of the playground is set to 10,000.
In the plot we can see the speedup, for both `OMP_PROC_BIND` set to close and spread, given by

$$S(n) = \frac{T(1)}{T(n)}$$

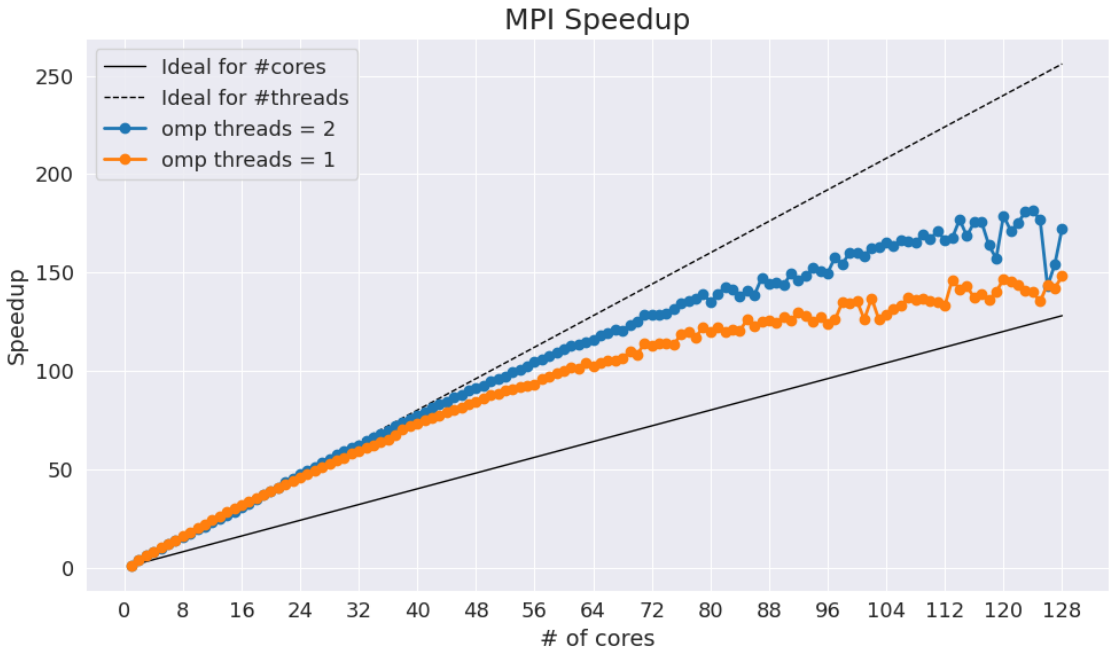
where n is the number of omp tasks and $T(n)$ is the time taken by n processes.
The *ideal* speedup is 1:1, altough this is ideal as not all the code is parallelized (nor parallelizable).



1.3.2 Strong MPI Scaling

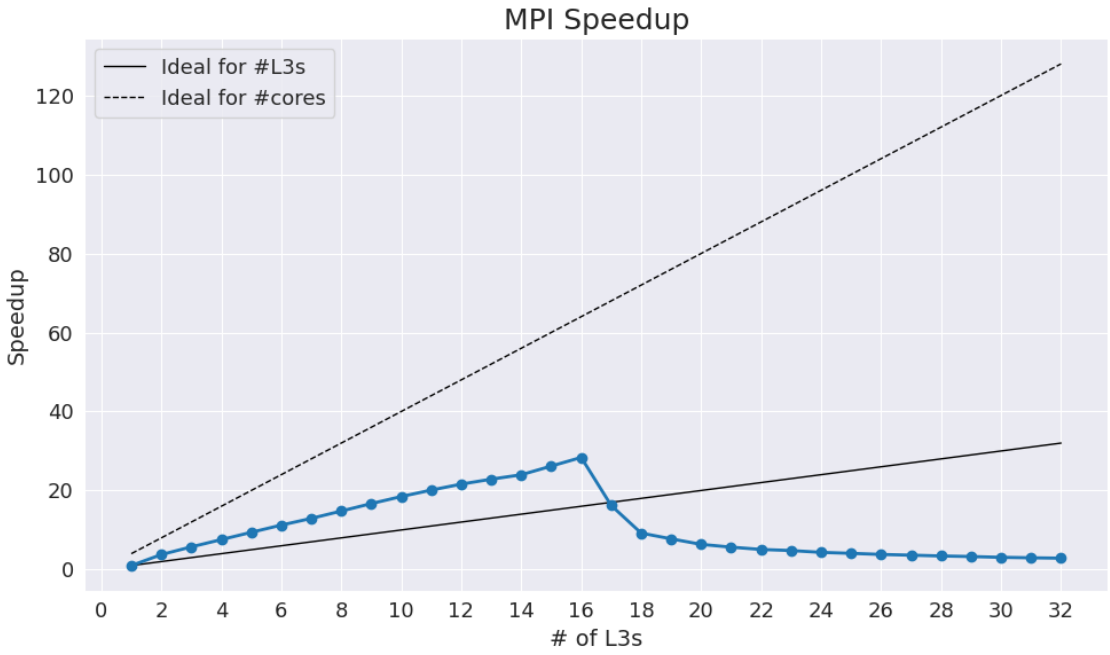
In this section we see how the program scales when increasing the number of MPI tasks, keeping OMP tasks fixed.
I tried to map the tasks to various "places", saturated with omp tasks.

First, lets look at how it scales when we run `mpirun -np n --map-by core ./main.x ...`. For this I set `OMP_PLACES=threads`, and `OMP_THREADS` both to 2 and to 1. **Note**: at the time of the experiment epyc001 and epyc002 had multithreading activated, this is no longer the case.



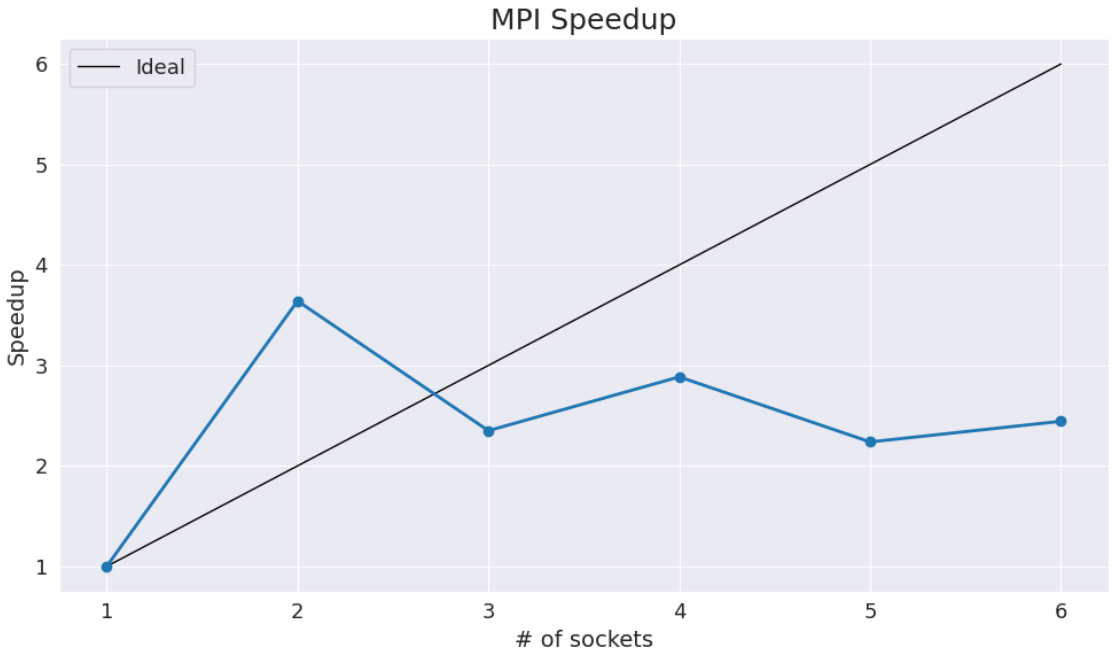
The speedup is considered on the MPI tasks. We can see that the program scales better that ideal 1:1.
For higher #cores the speedup is worse, due to the workload per task being too small.

Next I tried to scale on the L3 cache with `OMP_PLACES=cores` and `OMP_NUM_THREADS=4`, that is the number of cores that share the same L3 cache.



We can see a huge drop in performance when the code starts to use two different sockets.

Finally we will look how it scales on the sockets, saturated with omp tasks (so 64).

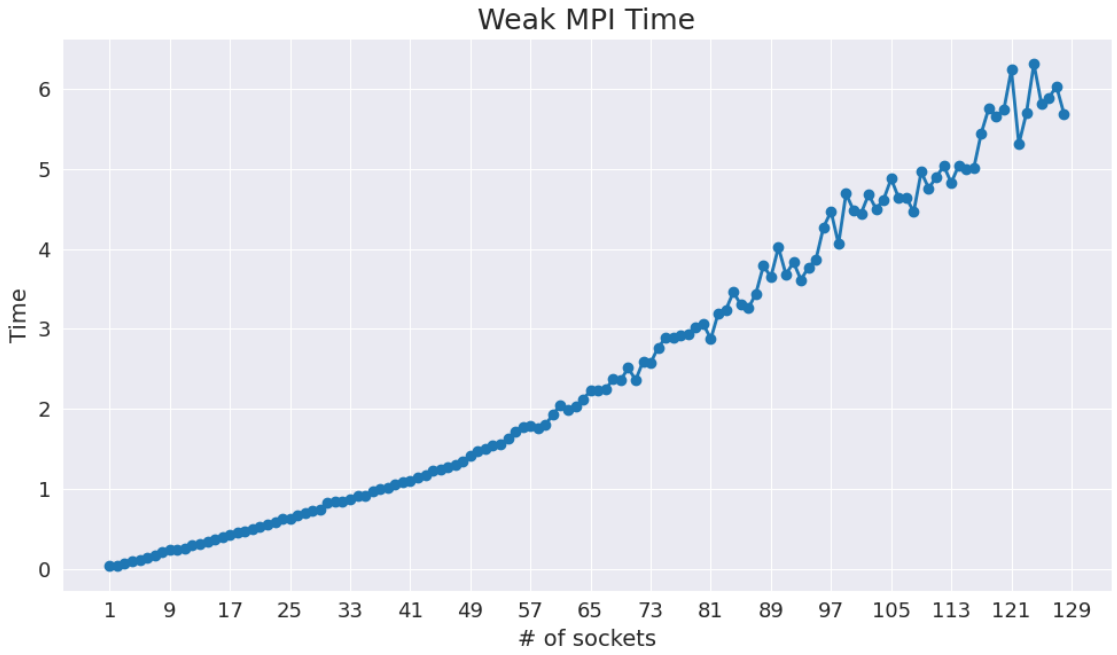


Here the performance drops when we change nodes.

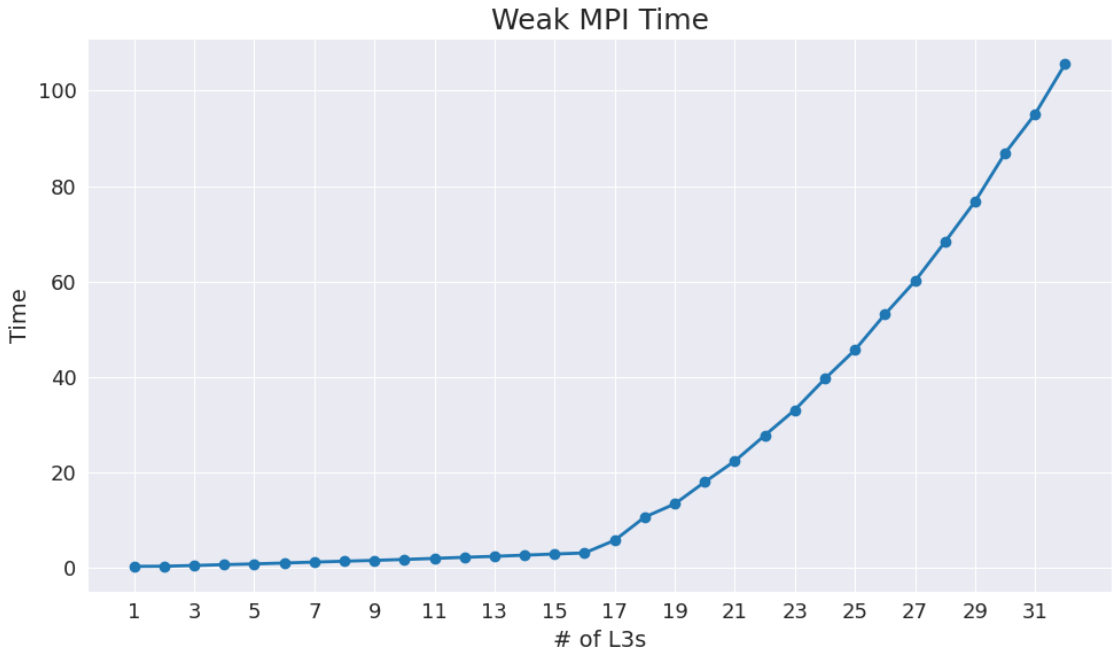
1.3.3 Weak MPI Scaling

Now we will measure how the program scales when the matrix size will vary in total, but the workload per MPI task remains constant.
Idealy total time should remain constant but this is hardly achievable due to communication time.

Let's start by keeping the workload per core fixed.

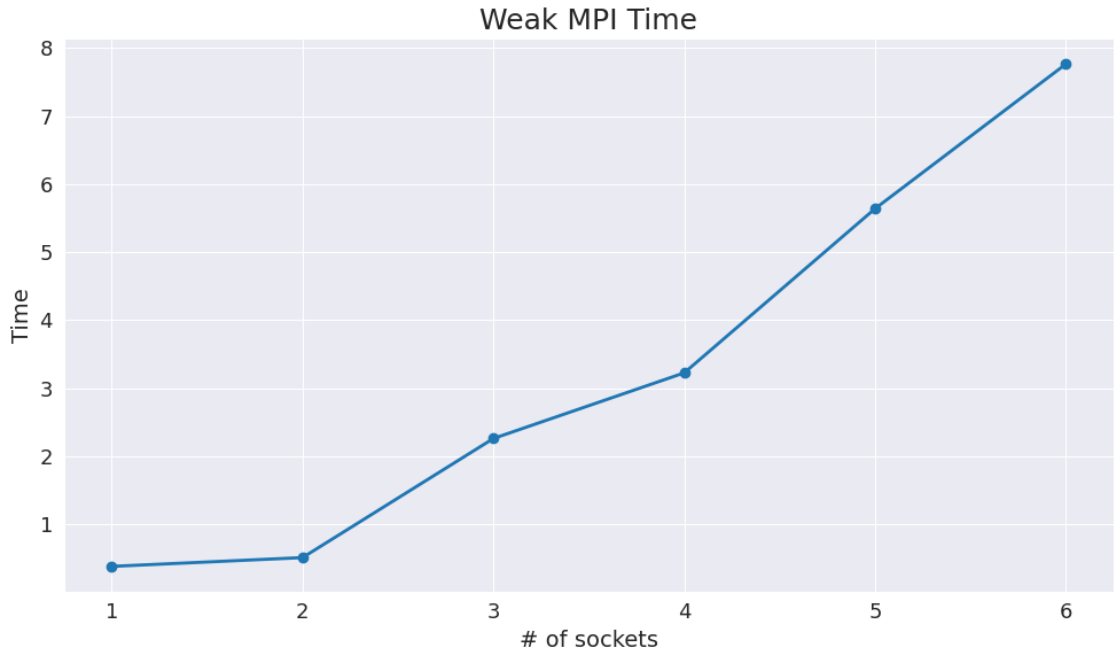


Now let's see the scaling by L3 cache.



Up to 16 cores, it works well, after that, the processes are placed on two sockets, drastically slowing communication.

Now let's try the same thing for sockets.



1.4 Final Considerations

Here are some improvements that I could have made and didn't already cover.

- When generating random numbers I consider the *modulo 2* operation, as only 0s or 1s are accepted in the matrix. This is suboptimal, and only 0s and 1s could be generated directly.
- I initiate the MPI region in the run function. It should be moved at the start of the main.
- I should test the idea of all the process reading the initial image, instead of communicating it.

2 Exercise 2

The goal of this exercise is to compare the performance, in particular the GFLOPs, of the math libraries *OpenBLAS*, *MKL* and *BLIS* on the *EPYC* and *THIN* nodes present on the orfeo cluster, for both single and double precision.

We have two types of scaling to measure: size scaling and core scaling.
Note that by *size* I refer to the number of rows of a matrix and not the total problem size. Since the matrices are squared and are all of the same size, as in $A, B, C \in \mathbb{R}^{\text{size} \times \text{size}}$. The total problem size can be univocally obtained by it.

- For the size scaling I increased the size of the matrices from 2,000 to 20,000 in steps of 1,000, repeating the measurement 10 times.

- For the core scaling I chose the intermediate matrix size of 10000 and scaled from one core to the number of cores present on the socket (64 for *EPYC* nodes, 12 for *THIN* nodes). I repeated the measurement 5 times on *EPYC* nodes and 10 times on *THIN* nodes.

In particular to gather the data on the *AMD* nodes I used the nodes *epyc001* and *epyc002* which, at the time of the benchmarking, had multithreading activated.

The relevant function call to measure is `GEMMCPU` in the `gemm.c` file, which is a slightly modified version of the `dgemm.c` file provided by the professor. The change only consists of printing size, time and GFLOPS in a format suitable for a `.csv` file.
The programs measure elapsed time and obtains the GFLOPS with the formula $\frac{2 \cdot size^3}{elapsed\ time \cdot 10^9}$.

The data is gathered by running bash scripts (found in `./exercise2/scripts/`), opportunely modified to call the desired library.

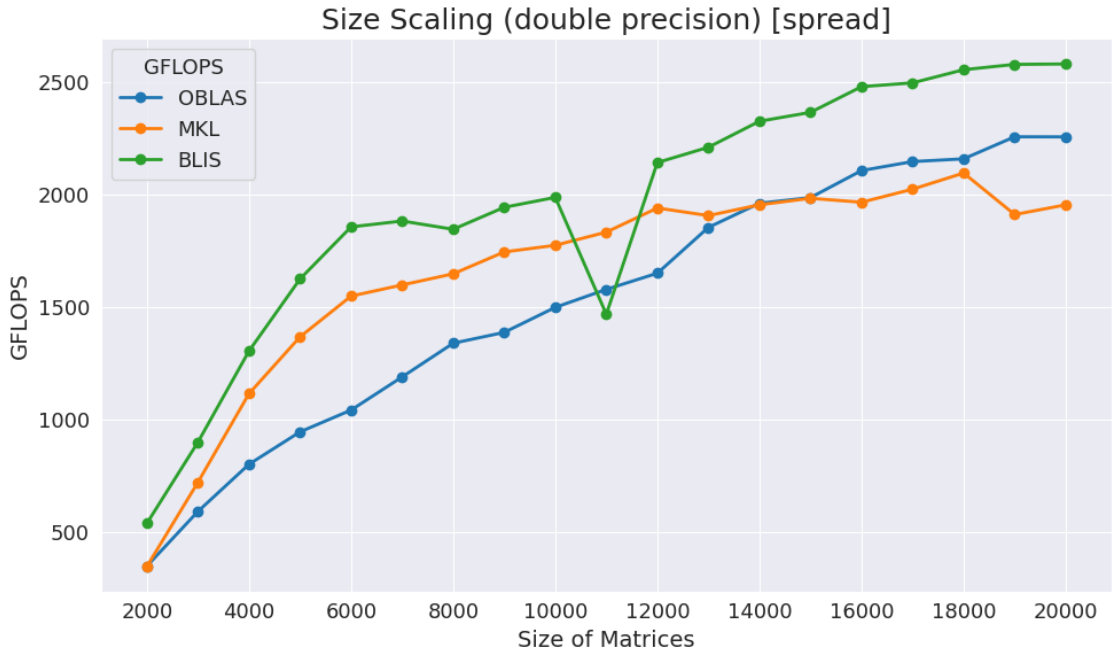
Now lets discuss the results obtained.

2.1 EPYC Nodes

2.1.1 Double Precision

Compiled with the **USE_DOUBLE** option in the Makefile.

Size Scaling The data is gathered by increasing the size of the three **square** matrices A,B,C from 2,000 to 20,000 with steps of size 1,000. The timing is performed 10 times and we consider the mean. To gather the data we used 64 cores with `OMP_PROC_BIND=spread`.



From the plot we can see that the BLIS library achieves the better performance overall. We can also see that for smaller sizes (< 14000), MKL performes better than OpenBLAS.

We can also notice a drop in the BLIS GFLOPs for size 11,000. This is not a random error (as we can see in the table below) but it's due to cache resonance.

	m	k	n	time	GFLOPS
90	11000	11000	11000	1.823896	1459.512941
91	11000	11000	11000	1.771730	1502.486621
92	11000	11000	11000	1.781291	1494.421612
93	11000	11000	11000	1.915901	1389.424464
94	11000	11000	11000	1.848808	1439.846441
95	11000	11000	11000	1.767532	1506.054915
96	11000	11000	11000	1.928238	1380.534927
97	11000	11000	11000	1.771703	1502.508765
98	11000	11000	11000	1.779068	1496.288740
99	11000	11000	11000	1.762646	1510.229291

The theoretical peak performance is

$$FLOPs = cores \times frequency \times FLOP/cycle$$

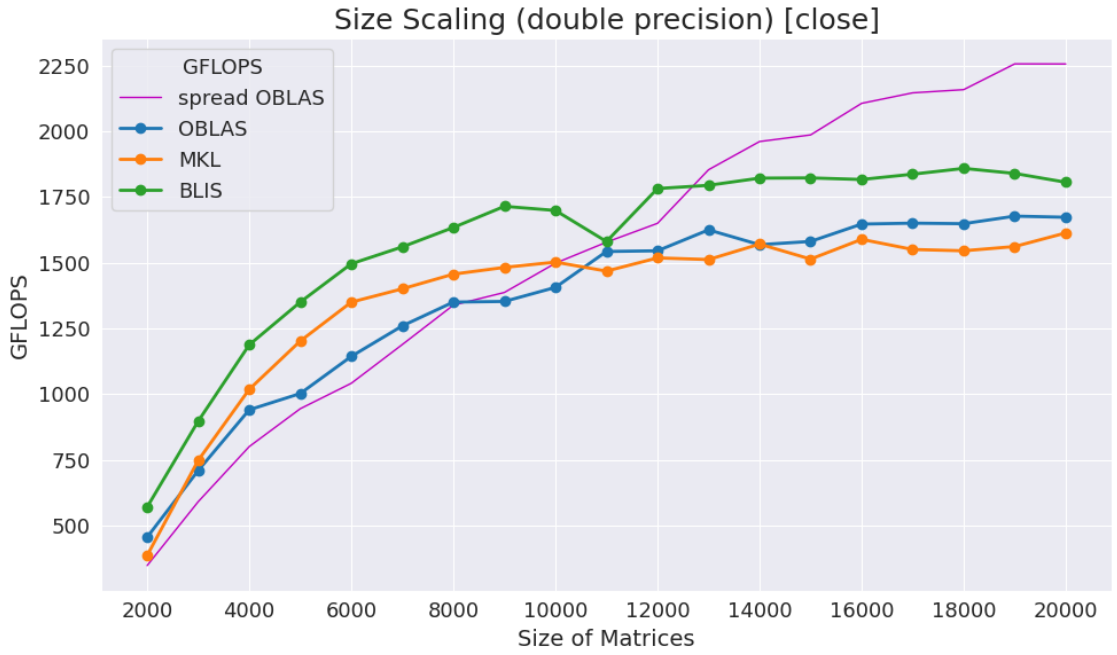
The maximum frequency for the *AMD* nodes is 2.6 GHz, and they can deliver up to 16 floating operations per cycle so we get

$$FLOPs = 64 \times 2.6 \times 10^9 \times 16 = 2,662,400,000,000$$

Equivalent to 2,662.4 GFLOPs. Which are nearly achieved by the BLIS library at size 20,000.

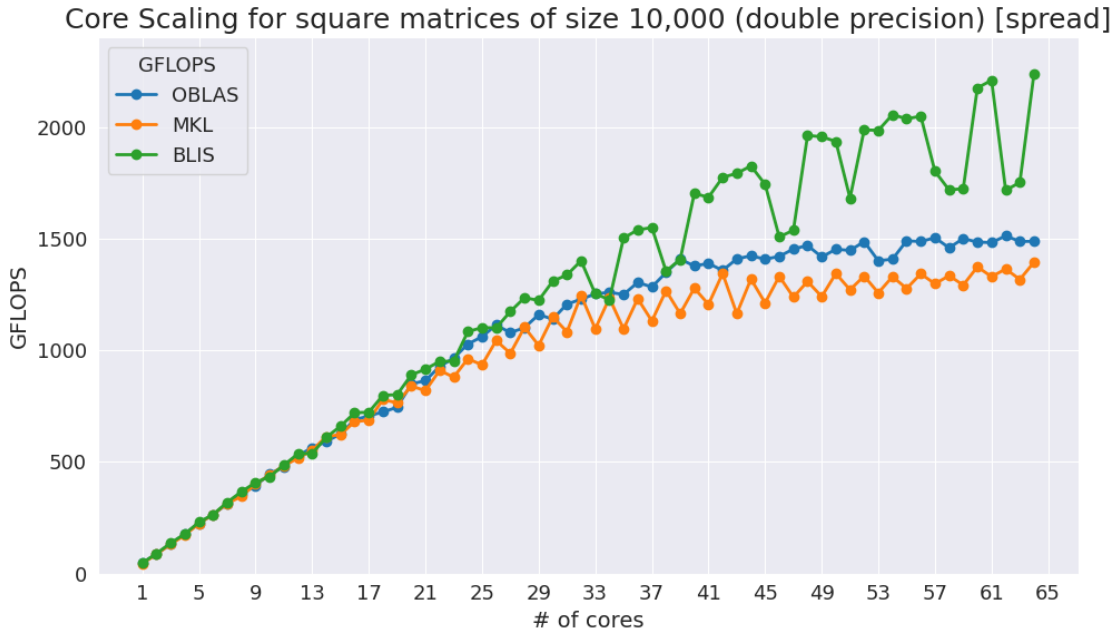
	m	k	n	time	GFLOPS
180	20000	20000	20000	6.141156	2605.372593
181	20000	20000	20000	6.378644	2508.370009
182	20000	20000	20000	6.151718	2600.899639
183	20000	20000	20000	6.157115	2598.619492
184	20000	20000	20000	6.236439	2565.566761
185	20000	20000	20000	6.152600	2600.526603
186	20000	20000	20000	6.150416	2601.449960
187	20000	20000	20000	6.193239	2583.462239
188	20000	20000	20000	6.208953	2576.924115
189	20000	20000	20000	6.219330	2572.624406

Now lets see the results with `OMP_PROC_BIND=close`. In magenta we can see *OpenBLAS* from the **spread** data, for comparison.



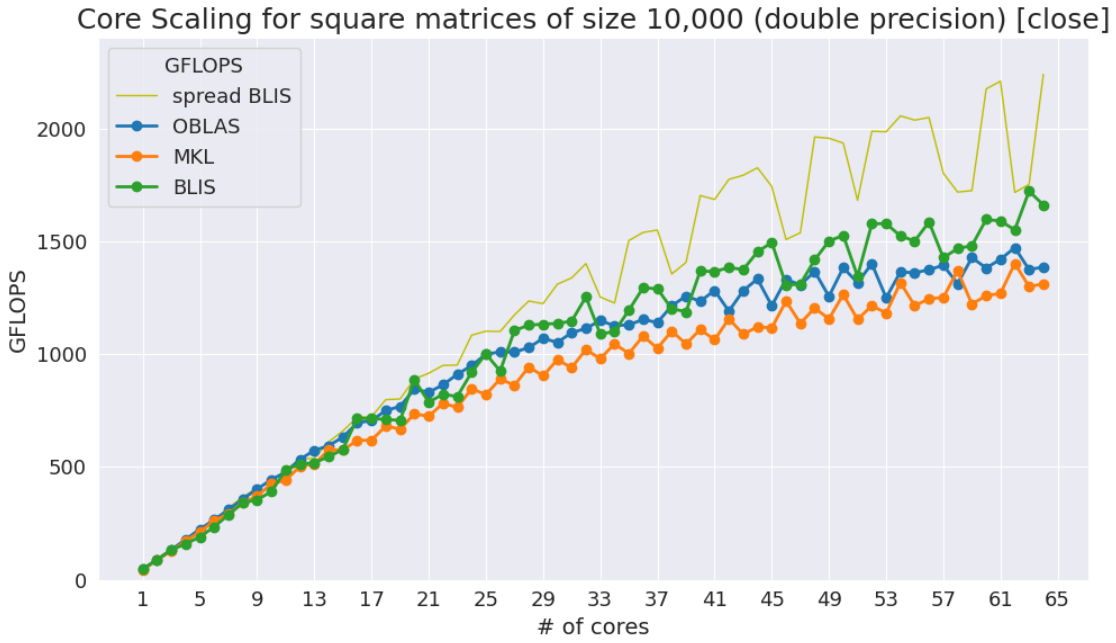
Core Scaling Now we will see how the GFLOPS increase as we increase the number of cores utilized in the computation.
Data is gathered from 5 observation for each "number of cores" utilized and we are considering the mean. I fixed matrix sizes to 10000 to perform the computation.

In the following graph you can see how the libraries scale with `OMP_PROC_BIND=spread`



Up to ~20 cores the scaling is linear for all libraries. The best performing library is *BLIS*, and its scaling is linear up to 32 cores.
With the *MKL* and the *BLIS* libraries we can see that there are drops in the output. This is probably due to how the libraries share the data between the threads.

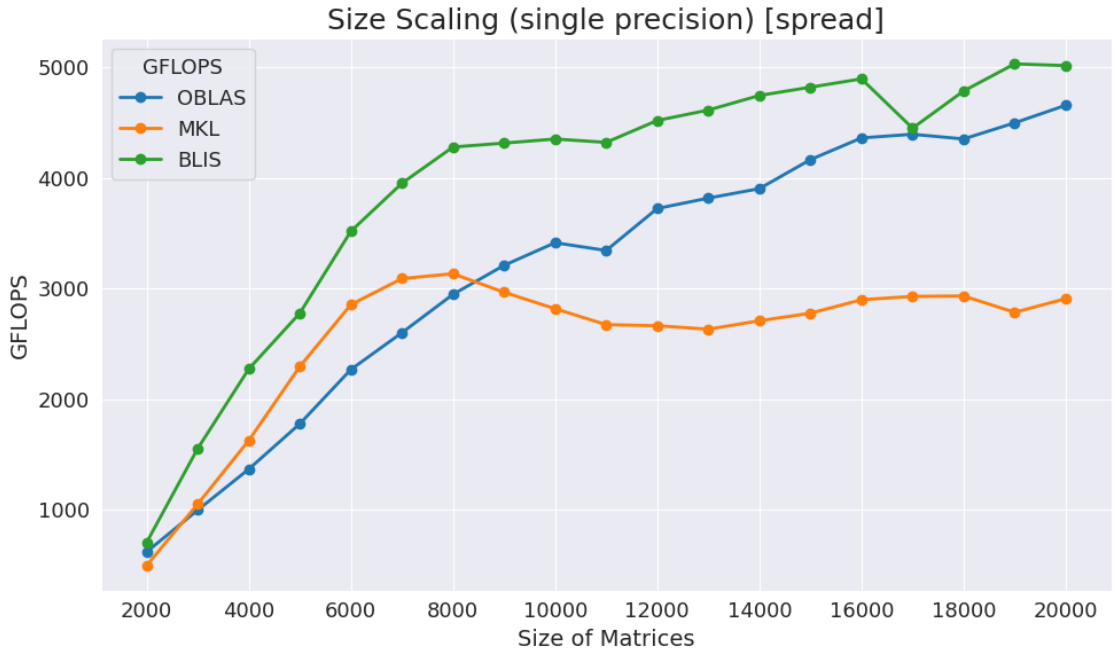
I tried to repeat the experiment with `OMP_PROC_BIND=close`, obtaining worse results. In yellow I plotted the spread *BLIS* data for reference, the plot scale remains the same.



2.1.2 Single Precision

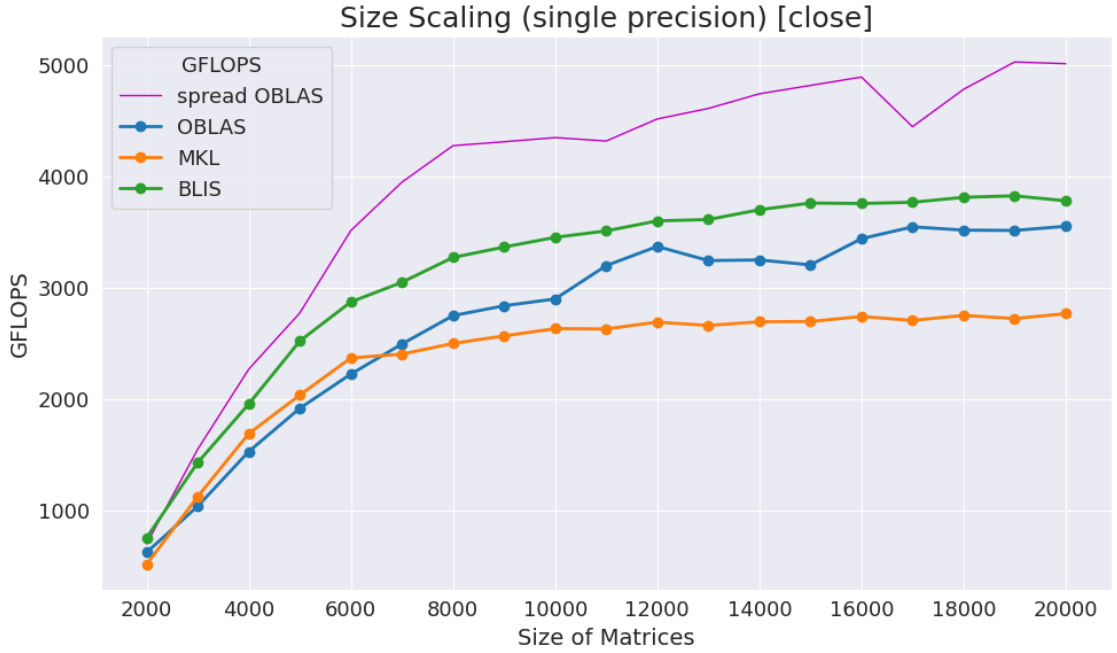
Now we repeat the same process as above, but we compiled the code with the option **USE_FLOAT** in the `Makefile`.

Size Scaling The data is gathered by increasing the size of the three **square** matrices A,B,C from 2,000 to 20,000 with steps of size 1,000. The timing is performed 10 times and we consider the mean. To gather the data we used 64 cores with `OMP_PROC_BIND=spread`.



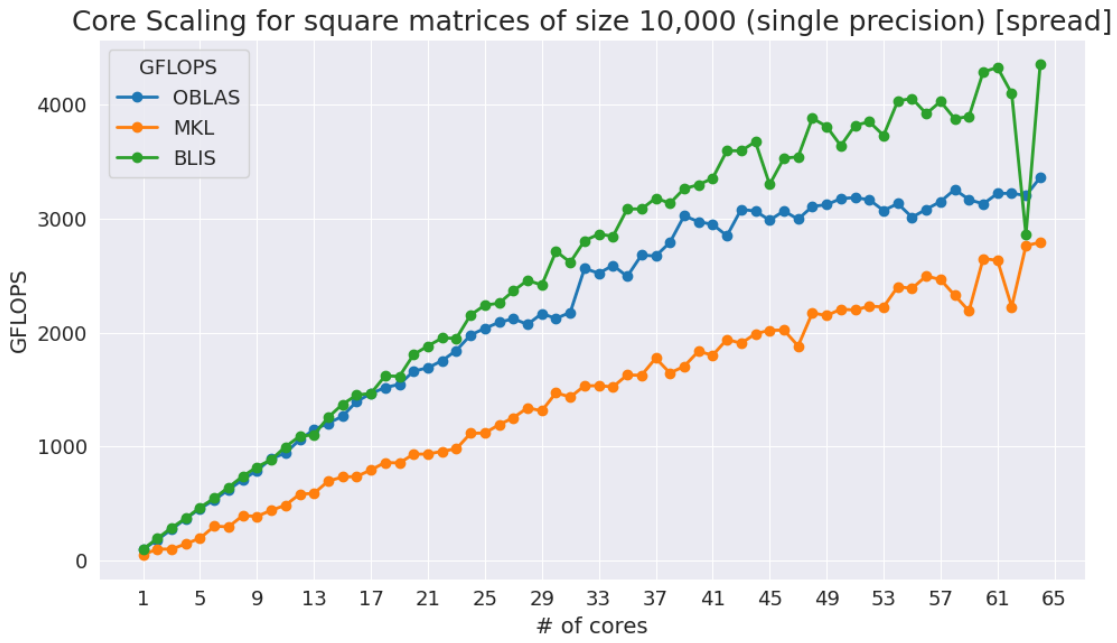
Note that the theoretical peak performance is two times the prevoiusly calculated one, so 5,324.8 GFLOPs, due to the precision used. I didn't reach it with my measurements, probably larger matrix sizes are required.

Lets see the result with `OMP_PROC_BIND=close`. In magenta we can see the spread *OpenBLAS* plot for comparison.



Core Scaling Now we will see how the GFLOPs increase as we increase the number of cores utilized in the computation.
Data is gathered from 5 observation for each "number of cores" utilized and we are considering the mean. I fixed matrix sizes to 10000 to perform the computation.

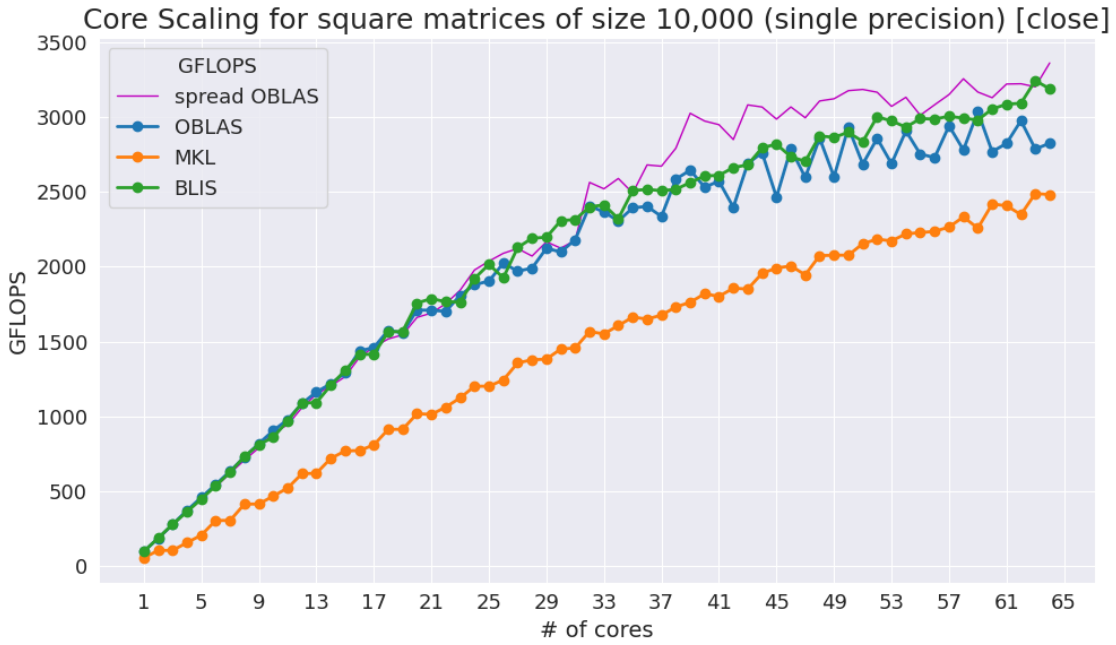
In the following graph you can see how the libraries scale with `OMP_PROC_BIND=spread`



Here we can notice that the libraries scale more linearly with respects to the double precision data. As before the best performing library is *BLIS*.
We can notice a huge drop at 63 cores, which is not a statistical error:

	cores	m	k	n	time	GFLOPS
310	63	10000	10000	10000	1.111818	1798.856298
311	63	10000	10000	10000	1.116953	1790.585159
312	63	10000	10000	10000	1.163291	1719.259995
313	63	10000	10000	10000	1.181964	1692.098561
314	63	10000	10000	10000	1.138019	1757.440016

Now let's repeat the experiment with `OMP_PROC_BIND=close`. In magenta I plotted the spread *OpenBLAS* for comparison.



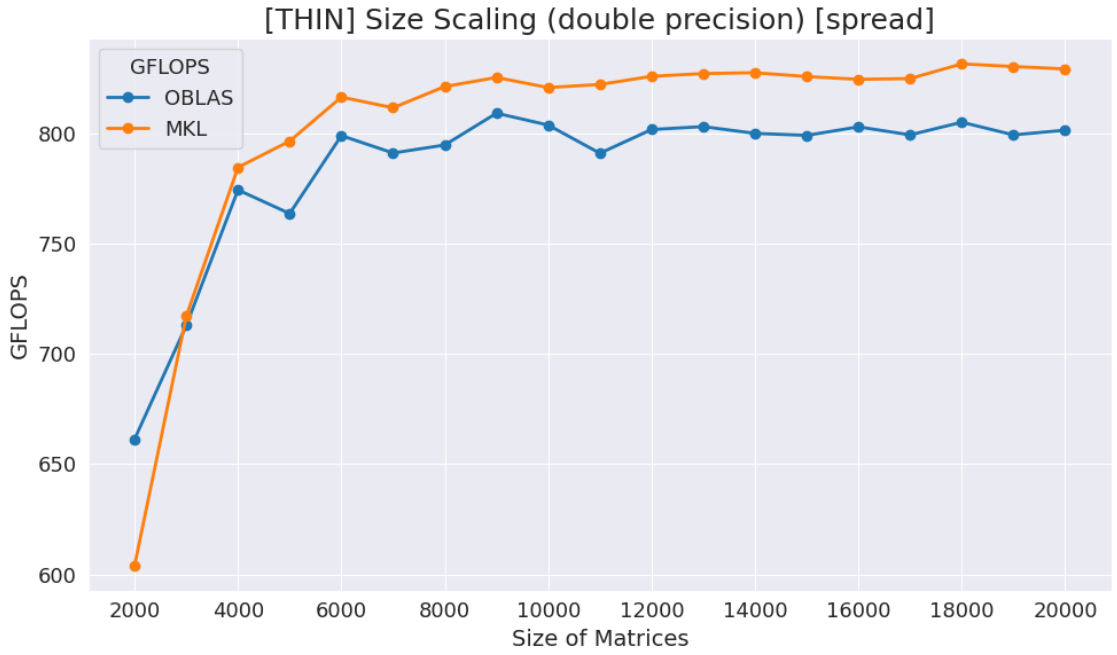
2.2 THIN Nodes

Now we repeat the same process for the *OpenBLAS* and *MKL* libraries on the THIN nodes. We won't benchmark the *BLIS* library as it should be recompiled appropriately for the INTEL nodes. Note that we still used the *gcc* compiler and not intel's *icc* compiler in the makefile.

2.2.1 Double Precision

Compile the `Makefile` with the flag **-USE_DOUBLE**.

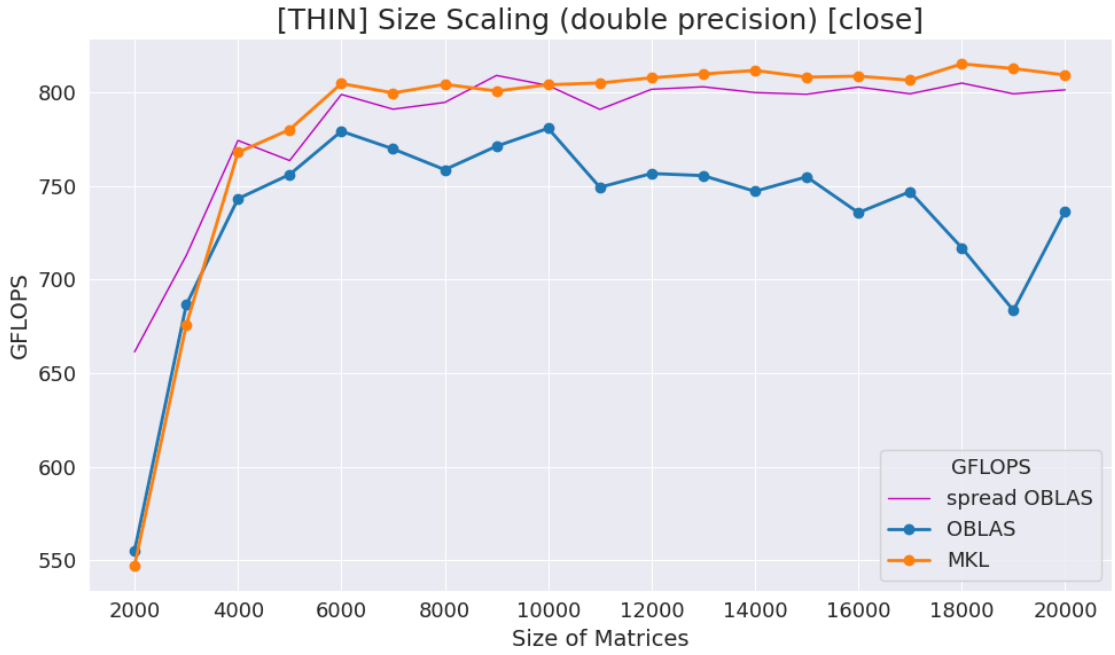
Size Scaling The data is gathered by increasing the size of the three **square** matrices A,B,C from 2,000 to 20,000 with steps of size 1,000. The timing is performed 10 times and we consider the mean. To gather the data we used 12 cores with `OMP_PROC_BIND=spread`.



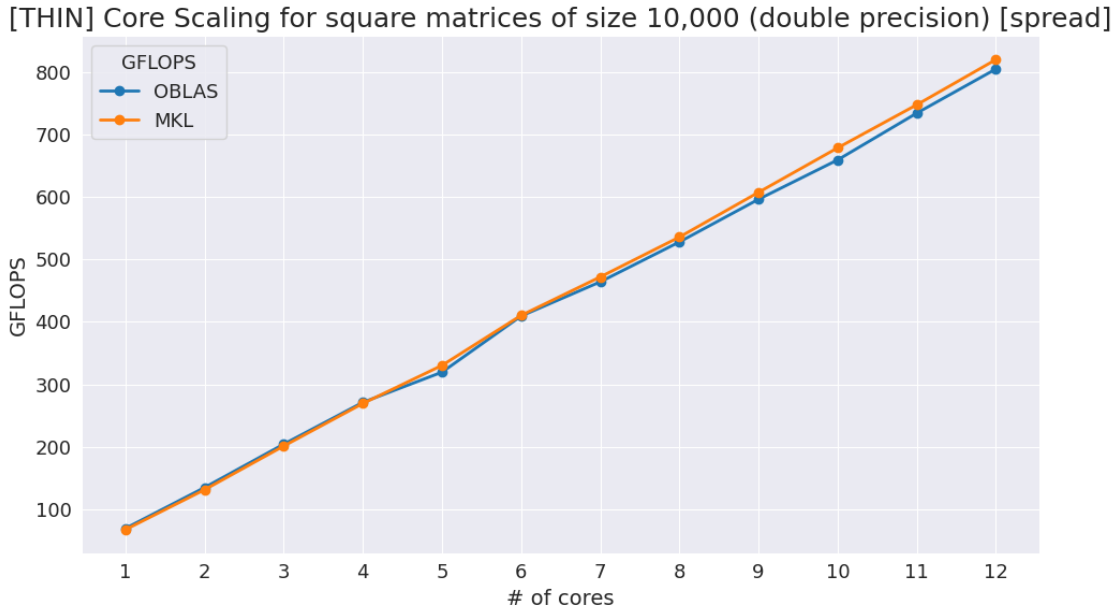
For the INTEL nodes, peak performance is 652.8 GFLOPs according to the producer. Somehow I surpassed it.

We can see that we reach the peak at smaller sizes respect to the AMD nodes. We can also notice that the *MKL* library is performing better on these nodes, this makes sense as the author of the *MKL* library is intel.

Now we repeat the experiment with `OMP_PROC_BIND=close` and compare the results with the spread bind policy. I plotted spread *OpenBLAS* in magenta for comparison.

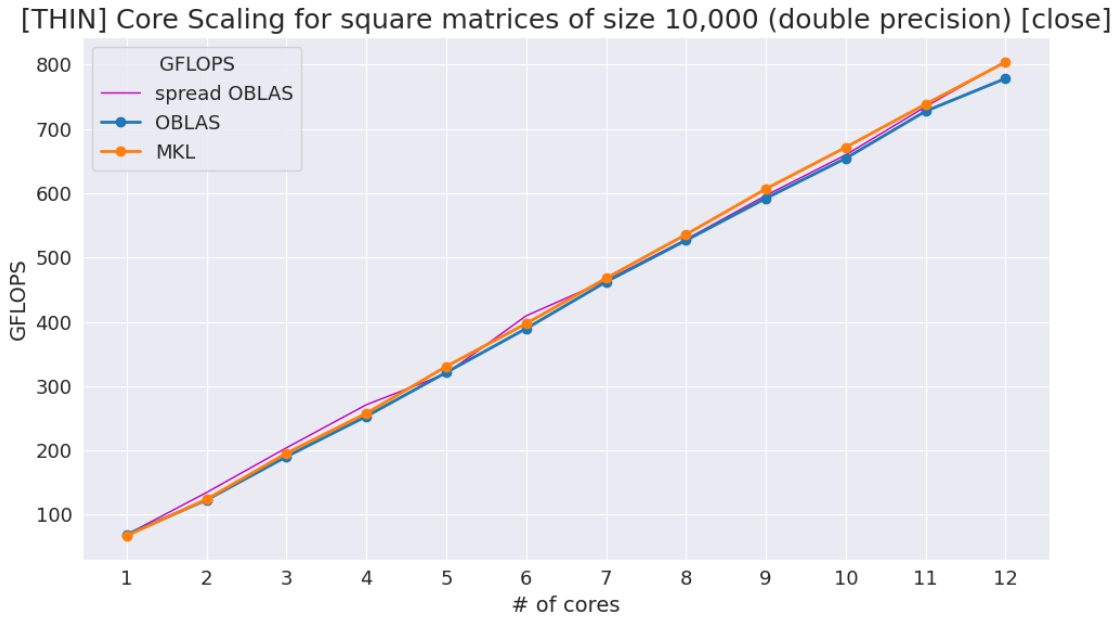


Core Scaling Now we will see how the GFLOPS increase as we increase the number of cores utilized in the computation.
Data is gathered from 12 observation for each "number of cores" utilized and we are considering the mean. For the data gathering we utilized OMP_PROC_BIND=spread. We fixed matrix sizes to 10000 (square matrix) to perform the computation.



Both libraries scale linearly.

Again, we repeat the process for `OMP_PROC_BIND=close` and compare the results, plotting spread *OpenBLAS* in magenta.

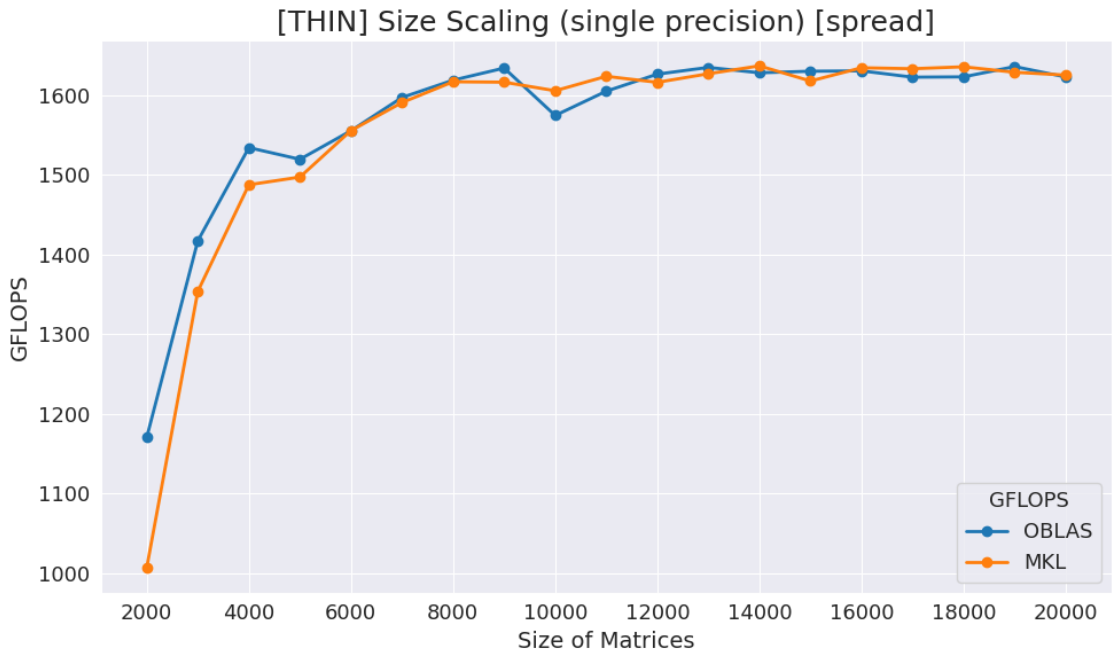


The proximity core placing policy doesn't seem to affect performance on these nodes.

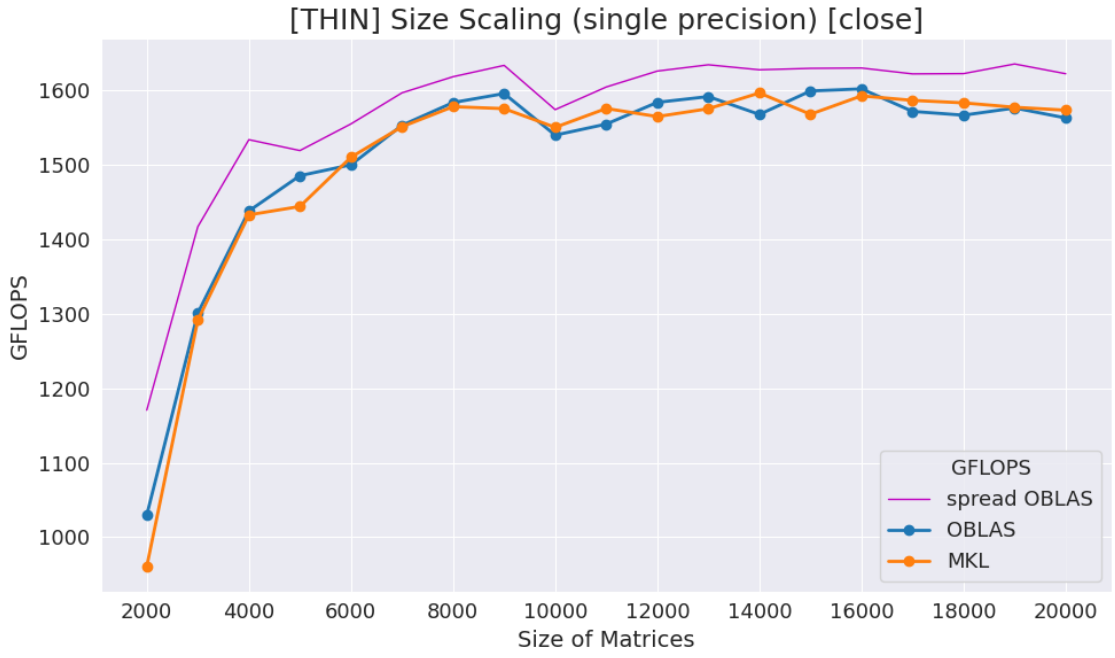
2.2.2 Single Precision

Now we repeat the same process as above, but we compiled the file with the option **USE_FLOAT**.

Size Scaling The data is gathered by increasing the size of the three **square** matrices A,B,C from 2,000 to 20,000 with steps of size 1,000. The timing is performed 10 times and we consider the mean. To gather the data we used 64 cores with `OMP_PROC_BIND=spread`.

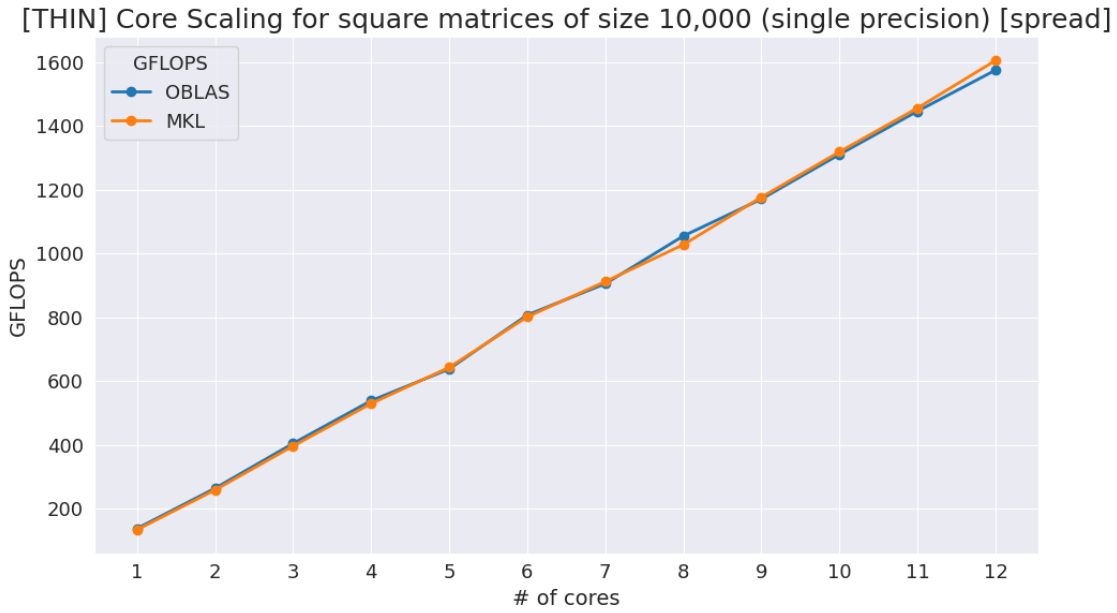


Again, lets change binding policy to close. In magenta spread *OpenBLAS* for reference.



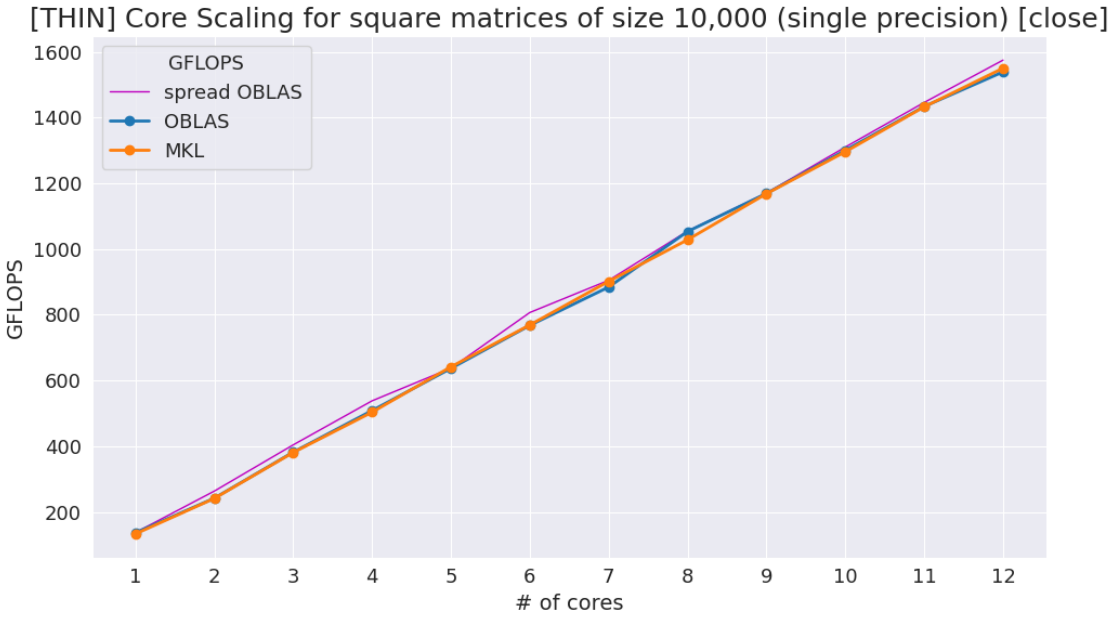
2.2.3 Core Scaling

In this section we will see how the GFLOPS increase as we increase the number of cores utilized in the computation.
Data is gathered from 12 observation for each "number of cores" utilized and we are considering the mean. For the data gathering we utilized *OMP_PROC_BIND*=spread. We fixed matrix sizes to 10000 (square matrix) to perform the computation.



As we can see, both libraries scale linearly.

Lets compare now wth the results obtained by using `OMP_PROC_BIND=close`. In magenta we can see spread *OpenBLAS* for reference.


[THIN] Core Scaling for square matrices of size 10,000 (single precision) [close]
GFLOPS
spread OBLAS
OBLAS
MKL
1600
1400
1200
1000
800
600
400
200
GFLOPS
1
2
3
4
5
6
7
8
9
10
11
12
# of cores