Foundations of High Performance Computing

Final assignment

Carlo De Nardin (Orfeo username: cdenardin)

January 31, 2023

Section
1

Game of Life - Hybrid Implementation

1.1. Introduction

This section presents a discussion on a potential hybrid implementation of the Game of Life using both MPI and OpenMP. Different types of evolution are discussed:

- **Ordered evolution**: the evolution of a cell is contingent upon the evolution of its neighboring cells (serial)
- **Static evolution**: cells are evolved simultaneously based on their present neighboring cells
- **Black-White (static) evolution**: variation of the static evolution, in which the evolution of black (alive) cells takes place before white (dead) cells evolution

1.2. Parallelism Methodology

Before implementing a parallel solution, it is necessary to identify where parallelism can be applied in this problem. In this case, a hybrid solution is implemented, so the data must be divided among processes and threads.

1.2.1. Processes parallelism

In this particular implementation, the grid has been partitioned among various processes. The partitioning method consists of dividing the grid into chunks of rows so each process will be responsible to process a particular group of rows. If the number of rows within the grid is not uniformly divisible between processes, to the final process will be assigned the remaining rows.

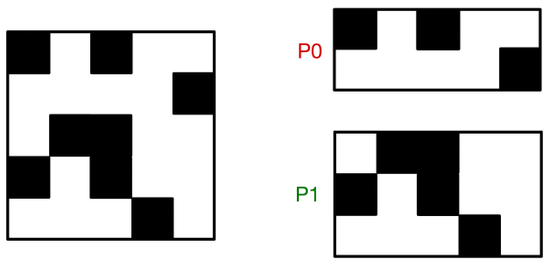


Figure 1: domain decomposition between processes

1.2.2. Threads parallelism

Then, each process will spawn threads which will independently compute the operations on individual cells to determine the evolution of the grid. Threads partitioning is executed by using OpenMP and using the parallel omp `#pragma for`. The for loop works is divided among the threads using the default parameters for chunk dimension and schedule allocation.

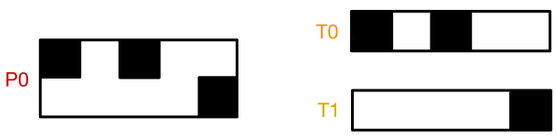


Figure 2: domain decomposition between threads

Once the domain decomposition is complete, each thread will proceed to calculate the number of living neighbors, for each cell, in its designated portion of the grid and evolve it.

1.3. MPI methodology

To calculate the number of living neighbors, each thread needs information from neighbouring processors because the cells on the border do not have 8 neighbors. Since this information is only available from processors on other nodes/sockets, it is necessary to exchange messages in order to obtain the upper and lower rows. In addition, for the calculation of ghost columns, only a simple computation is necessary.

1.3.1. Ghost rows

To transmit the upper and lower rows to the upper and lower processors, respectively, the **`MPI Send`** and **`MPI Receive`** routines are used. In this implementation, these communication exchanges are necessary:

- the first row is sent to the upper process and the lower ghost row is received from the lower process
- the last row is sent to the lower process and the upper ghost row is received from the upper process

1.3.2. Ghost columns

Ghost columns can be calculated using a simple computation once each process has received the ghost rows. That's because without the ghost rows, the computation of the corners would not be possible:

- the last column represents the left ghost column
- the first column represents the right ghost column

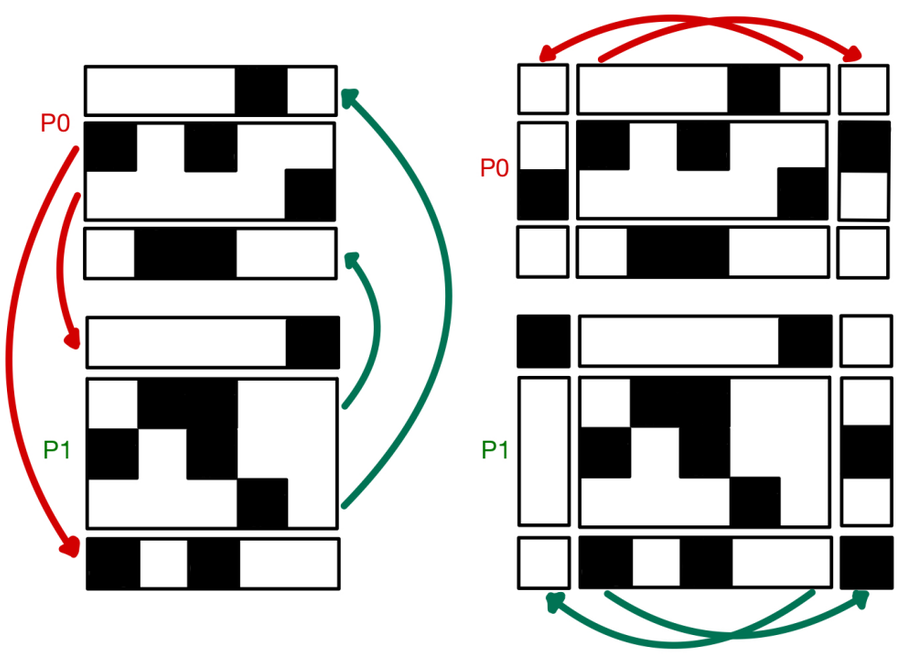


Figure 3: exchange of ghost rows and computation of ghost columns

1.4. Implementation

A possible implementation using static evolution is analyzed, covering the following steps: domain decomposition among processes, message passing (ghost rows and columns) and static evolution.

1.4.1. Domain Decomposition among processes

For the domain decomposition among processes phase, the size of each process's grid is initially computed (the last one may have more data). Process 0 extracts the data from the binary image (PGM) and sends the corresponding portion of data to each process, keeping a portion for itself.

Code snippet

```
int local_rows = rows / size;
int local_cols = cols;
int offset = rows % size;
if (rank == size - 1) {
  local_rows += offset;
}
...
if (rank == 0) {
  ...
  for (int i = 1; i < size; i++) {
    if (i < size - 1) {
      MPI_Send(&full_grid_temp[i * local_size], local_size, ...);
    } else {
      MPI_Send(&full_grid_temp[i * local_size], (local_rows + offset) * local_cols, ...);
    }
  }
}

if (rank != 0) {
  MPI_Recv(&local_grid_temp[0], local_size, MPI_INT, 0, 0, ...);
}
```

1.4.2. Message passing and ghost columns computation

After the data have been splitted among processes, the required rows are exchanged with other processes, and all the data are saved in a matrix with dimensions of rows + 2 and columns + 2.

Code snippet

```
int upper_rank = (rank == 0) ? size - 1 : rank - 1;
int lower_rank = (rank == size - 1) ? 0 : rank + 1;

MPI_Send(&local_grid_wg[...], ..., MPI_INT, upper_rank, 0, ...);
MPI_Recv(&local_grid_wg[...], ..., MPI_INT, lower_rank, 0, ...);

MPI_Send(&local_grid_wg[...], ..., MPI_INT, lower_rank, 0, ...);
MPI_Recv(&local_grid_wg[...], ..., MPI_INT, upper_rank, 0, ...);
```

The calculation of the ghost columns is simply performed by copying the values of the columns as shown in the above Figure 3.

Code snippet

```
for(int i = 0; i < local_rows_wg; i++) {
  local_grid_wg[i * local_cols_wg] = local_grid_wg[(i + 1) * local_cols_wg - 2];
  local_grid_wg[(i + 1) * local_cols_wg - 1] = local_grid_wg[i * local_cols_wg + 1];
}
```

1.4.3. Static Evolution

Static evolution is performed by counting the neighbours of each individual cell, and the result is saved in a new grid to keep the static state. Moreover, this for loop is managed by OpenMP, which generates several threads to split different sections of loop in different cores.

Code snippet

```
#pragma omp parallel for schedule(static)
for(int i = 1; i < rows - 1; i++) {
  for(int j = 1; j < cols - 1; j++) {
    int count = count_alive_neighbors(grid, i, j, cols);
    if (count < 2 || count > 3) {
      grid_ns[i * cols + j] = DEAD;
    } else if (count == 3) {
      grid_ns[i * cols + j] = ALIVE;
    } else {
      grid_ns[i * cols + j] = grid[i * cols + j];
    }
  }
}
```

Code snippet

```
int count_alive_neighbors(int *grid, int i, int j, int cols) {
  int alive_neighbors = 0;
  for(int k = -1; k <= 1; k++) {
    for(int l = -1; l <= 1; l++) {
      if (k == 0 && l == 0) continue;
        if (grid[(i + k) * cols + (j + l)] == ALIVE) alive_neighbors++;
    }

  }
  return alive_neighbors;
}
```

After each evolution or once all evolutions are completed, the result is saved in a binary image in PGM format.

1.5. Results and Discussion

In this section, a review of the scalability of the implementation of the game of life is carried out, with a focus on: strong OMP scalability and strong / weak MPI scalability. The following analysis used a **static evolution** without saving intermediate images (serial task).

1.5.1. OpenMP Strong Scalability

For the OMP Strong Scalability analysis, a task was allocated within a node **epyc[001]**, and subsequently multiple tests were performed by increasing the number of threads that are binded closely. A square grid of constant size **(5000)** and a number of repetitions equal to **100** were used. From the obtained data, the speed up and efficiency were calculated and also represented graphically.

k	Threads	Time in seconds	Speedup	Efficiency
5000	1	84.460072	1	1
5000	2	42.891812	1.969142	0.9845710
5000	4	22.220480	3.801001	0.9502503
5000	8	12.386212	6.818878	0.8523598
5000	12	9.221662	9.158878	0.7632398
5000	16	7.248387	11.652258	0.7282661
5000	24	5.291544	15.961329	0.6650554
5000	32	4.382527	19.272001	0.6022500
5000	48	3.393173	24.891176	0.5185662
5000	64	2.994492	28.205140	0.4407053
5000	96	3.036013	27.819404	0.2897855
5000	128	2.767575	30.517716	0.2384197

Table 1: OMP strong scalability

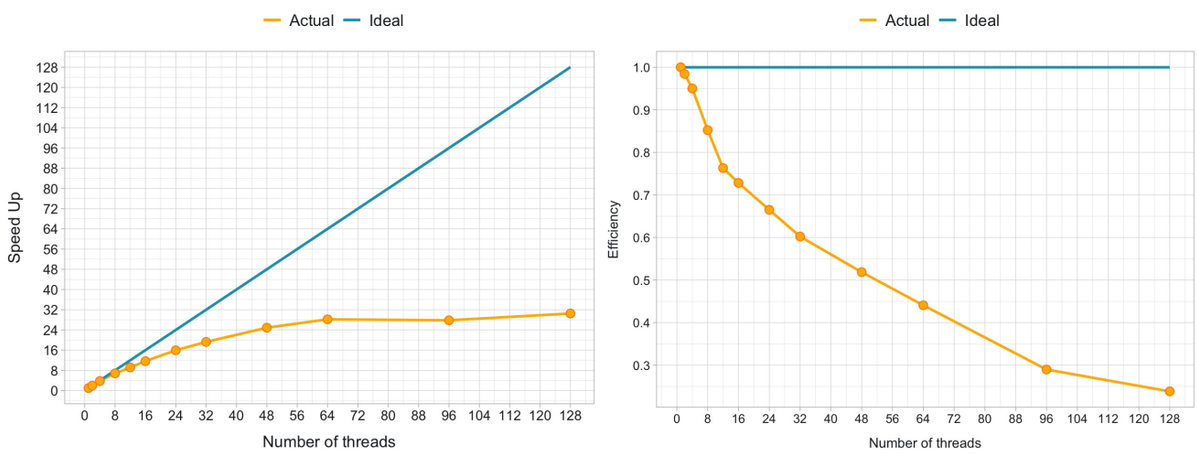


Figure 4: OMP strong scalability graphs, speedup (left) and efficiency (right)

The purpose of this analysis was to understand whether by increasing the number of OMP threads on a fixed problem size the problam scale. From the results obtained, it can be analyzed how the implementation using a number of threads in the range of [8 - 16] scale well.

1.5.2. MPI Strong Scalability

For the MPI Strong Scalability analysis, these considerations have been taken into account:

- the analysis was made on a constant grid dimension and the tests were repeated on different image dimensions
- For each grid dimension a MPI task has been allocated for each socket from 1 to 6 (on Orfeo each node has 2 sockets and no more than 3 nodes can be requested)
- In each socket the workload was spread over 64 OMP threads (saturating the socket)

The aim of this analysis was to evaluate the performance of the implementation when the data are distributed across multiple MPI tasks. The speedup and efficiency were calculated and presented in graphical form for all the tests conducted.

k	Processes	Time in seconds	Speedup	Efficiency
5000	1	3.484226	1	1
5000	2	5.839856	0.5966288	0.2983144
5000	3	3.980710	0.8752775	0.2917592
5000	4	3.867306	0.9009440	0.2252360
5000	5	3.572204	0.9753715	0.1950743
5000	6	3.653418	0.9536894	0.1589482
10000	1	12.673369	1	1
10000	2	16.913718	0.7492953	0.3746476
10000	3	11.847933	1.0696692	0.3565564
10000	4	10.586120	1.1971685	0.2992921
10000	5	9.327808	1.3586653	0.2717331
10000	6	8.492848	1.4922402	0.2487067
15000	1	27.34993	1	1
15000	2	33.03086	0.8280114	0.4140057
15000	3	25.08893	1.0901193	0.3633731
15000	4	21.32455	1.2825564	0.3206391
15000	5	19.48020	1.4039864	0.2807973
15000	6	17.43915	1.5683062	0.2613844
20000	1	47.98243	1	1
20000	2	56.50550	0.8491639	0.4245819
20000	3	41.56062	1.1545166	0.3848389
20000	4	35.36498	1.3567779	0.3391945
20000	5	30.00228	1.5992929	0.3198586
20000	6	28.24807	1.6986092	0.2831015

Table 2: MPI strong scalability

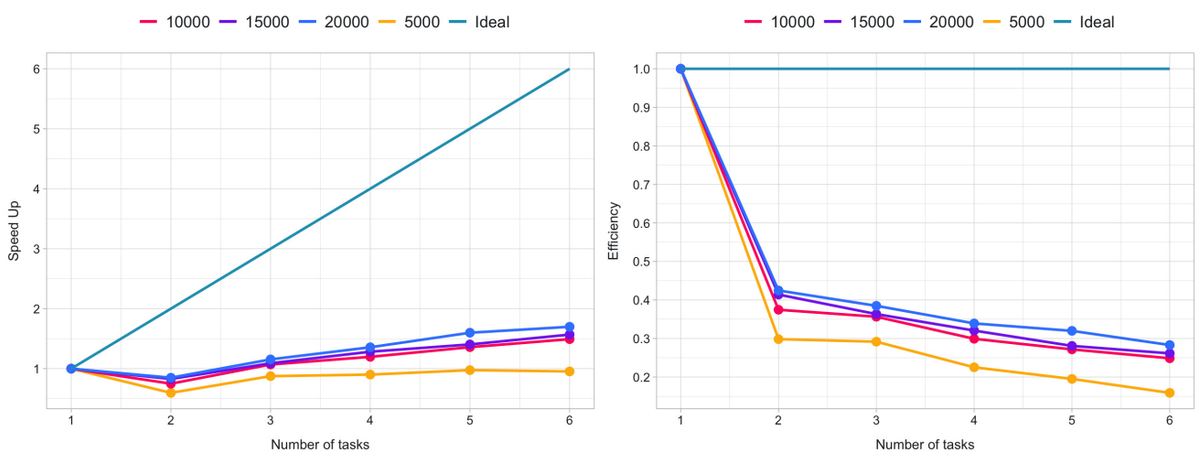


Figure 5: MPI strong scalability graphs, speedup (left) and efficiency (right)

The results show that the implementation doesn't scale well with increased MPI tasks. To understand why, let's examine some of the overhead costs:

k	Processes	Message overhead	Computation time (s)	Total time (s)
5000	1	0.255265	1.738280	3.484226
5000	4	2.401923	0.684036	3.867306

Table 3: MPI strong overhead for k = 5000 and process = 1 and process = 4

As can be observed from the two recorded executions shown in the table above:

- the first execution used 1 MPI process and used 64 OMP threads to distribute the workload
- the second execution used 4 MPI processes, with each process utilizing 64 OMP threads to distribute the workload

The first execution has lower message overhead compared to the second execution. However, each MPI task in the second execution has a lower workload, resulting in a lower computation time. As a result, the total time spent is higer in the second execution, making the use of multiple MPI tasks useless.

1.5.3. MPI Weak Scalability

For the MPI Weak Scalability analysis, these considerations have been taken into account:

- the analysis was conducted while maintaining a fixed workload for each MPI task
- the analysis began with a grid of 5000 x 5000, and the number of processes was increased by doubling the number of cells in the matrix, ensuring each process worked on the same problem size.
- the workload was distributed among 64 OMP threads (saturating the socket) in each socket.

The aim of this analysis was to investigate if the use of multiple MPI tasks with a fixed workload results in a linear execution time or an increase in execution time due to the added message overhead.

k	Processes	Time in seconds	Speedup	Efficiency
5000	1	3.412774	1	1
7071	2	8.950845	0.3812795	0.19063976
8660	3	9.273503	0.3680134	0.12267114
10000	4	10.452709	0.3264966	0.08162415
11180	5	10.996402	0.3103537	0.06207074
12247	6	11.877159	0.2873392	0.04788987

Table 4: MPI weak scalability

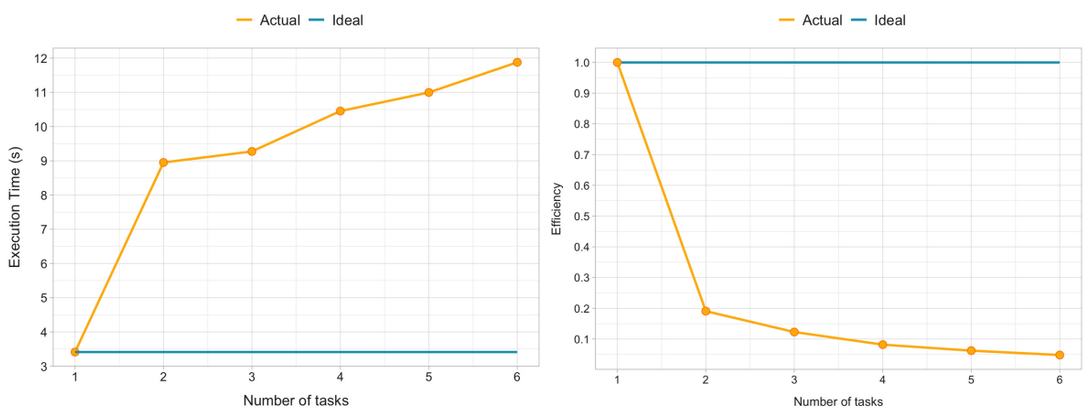


Figure 6: MPI weak scalability graphs, execution time (left) and efficiency (right)

The results indicate that the implementation is not weakly scaling well, as increasing the number of MPI processes leads to an increase in overhead. The table above presents some overhead results.

k	Processes	Message overhead	Computation time (s)	Total time (s)
5000	1	0.259281	1.751660	3.412774
7071	2	2.763381	3.451711	8.950845
8660	3	3.395790	3.062104	9.273503
10000	4	3.529295	3.461008	10.452709
11180	5	4.107266	3.136995	10.996402
12247	6	4.246457	3.419886	11.877159

Table 5: MPI weak overhead

The computation time is evenly distributed with varying numbers of processes. The problems are in the message overhead (increase by increasing the number of processes) and other non-computational overheads.

1.6. Conclusions

Based on the analysis, it can be concluded that the program scales quiet well with OpenMP (up to 16/32 threads) due to its ability to efficiently distribute the workload among multiple threads within a single process. However, when the number of MPI tasks increases, the program does not scale well due to the introduction of message overhead and other non-computational overheads. These overheads can arise from communication and synchronization between multiple processes, leading to a decrease in performance and efficiency. To achieve better scalability in such cases, it may be necessary to optimize the communication and synchronization between processes.

One possible solution, based on the results presented, would be to optimize the data exchange between matrices which, in this implementation, create non-computational overhead.

Section
2

OBLAS and MKL - Matrix multiplication analysis

2.1. Introduction

In this analysis, the performance of two mathematical libraries, OpenBLAS (OBLAS) and Intel Math Kernel Library (MKL), will be evaluated specifically in the context of matrix-matrix multiplication (gemm function). Matrix-matrix multiplication is a fundamental operation in many scientific and engineering applications, and choosing the right mathematical library can have a significant impact on performance.

The gemm function was tested using single and double precision. Double precision is usually slower than single precision as it requires more bits to represent a value. Single precision uses 32 bits to represent a number while double precision uses 64 bits. Therefore, the processor has to process twice as many bits in the case of double precision.

An overview of the tests conducted is now reported:

- The first test was performed on an Orfeo AMD node **(epyc[005])** using a single instance of the gemm function in single and double precision, and dividing the work among 64 cores. The test was carried out using different arguments for the **places (sockets, cores, threads)** and **bind (close, master, spread)** parameters.
- Subsequently, the strong scalability of the OBLAS and MKL algorithms was analyzed as the number of usable cores increased for a fixed matrix size (14000). This test was again performed on the AMD node **(epyc[005])** and with **places = <sockets, cores>** and **bind = close**.
- Finally, a last test was carried out on an Orfeo INTEL node **(thin[007])** using a single instance of the gemm function in single precision and dividing the work among 12 cores. In this case, the previously mentioned policies for the places and bind parameters were also analyzed.

2.2. Epyc AMD node

Initially, the theoretical peak performance of the node was calculated using 64 cores. Each core performs 16 floating point operations per cycle and it has a maximum frequency of 2.6 GHz so the theoretical peak performance calculated was 2.662 TFlops.

2.2.1. Float precision

The above graphs are derived from the results obtained from the tests using single precision with different parameters for **places** and **bind**.

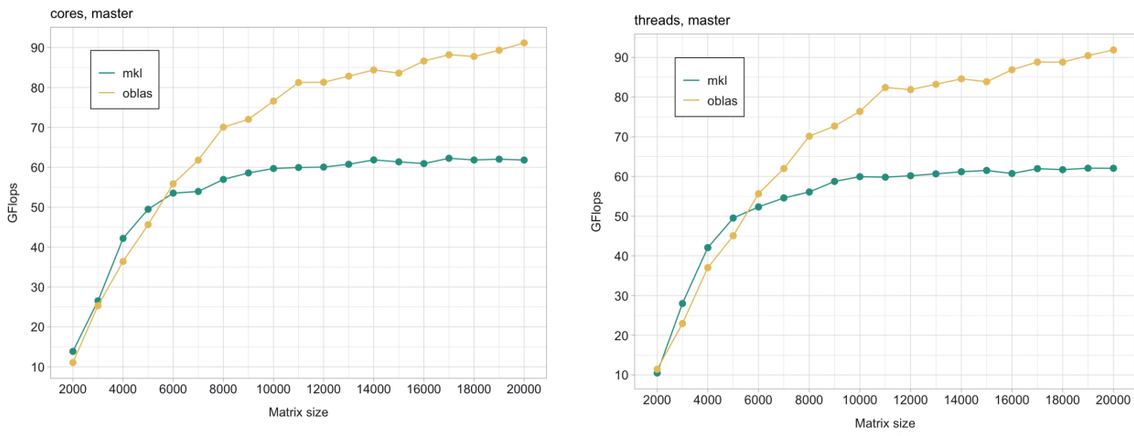
2.2.1.1. Places sockets

From the graphs obtained using the **places = sockets** parameter, we can analyze how the results obtained are quite similar between OBLAS and MKL. In the results, particularly those from MKL, irregularities in the increase in computational power as the matrix size increases have been identified. Moreover, there is no specific difference between the different types of binding.

2.2.1.2. Places cores and threads

These tests were run on the **epyc[005]** node, which, upon initial analysis, does not have SMT (Simultaneous Multithreading) enabled. This assumption was verified using the command **`'srun lscpu | grep "Thread(s) per core"'`**. Thus, it is assumed to obtain similar results between **places = <cores, threads>**.

From the results obtained, it can be seen, that setting the parameter **places = <cores, threads>** gives similar values. Specifically, with a **bind = cores**, OBLAS tends to increase computational power more when the matrices are larger with respect to MKL. Also OBLAS can reach a range of [2100-2400] GFlops, while MKL does not exceed 1800 GFlops for matrices of dimension [15000-20000]. In the case of **bind = spread**, OBLAS tends to perform better than MKL. A strange result was obtained with matrices of size [4000-8000] for MKL, as it reaches the theoretical peak.

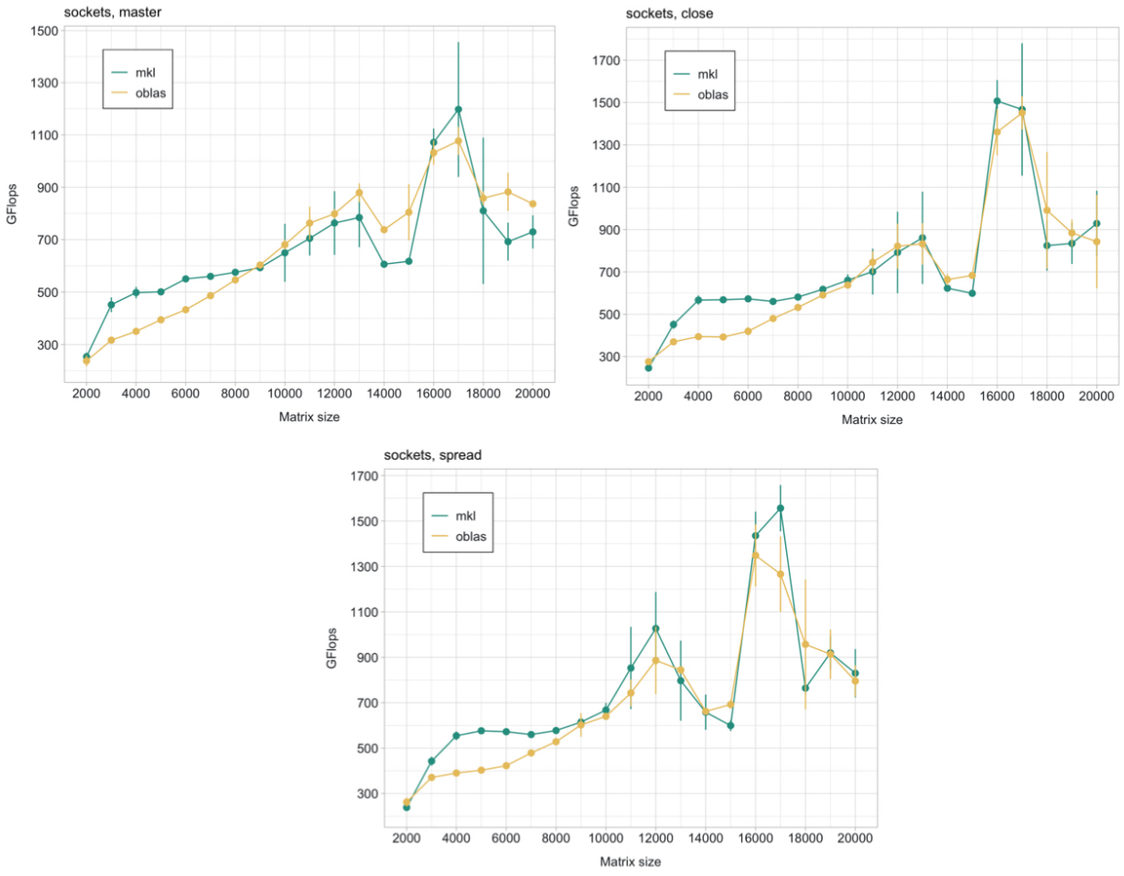


Regarding the **bind = master**, we can see that lower results are obtained. That's because the 64 swthreads are running on a single core (serial). In the case of OBLAS, a peak around 90 GFlops is obtained, while MKL does not exceed 60 GFlops.

2.2.2. Double precision

The same analyses conducted with a single precision are reported with double precision for the same places and bind parameters.

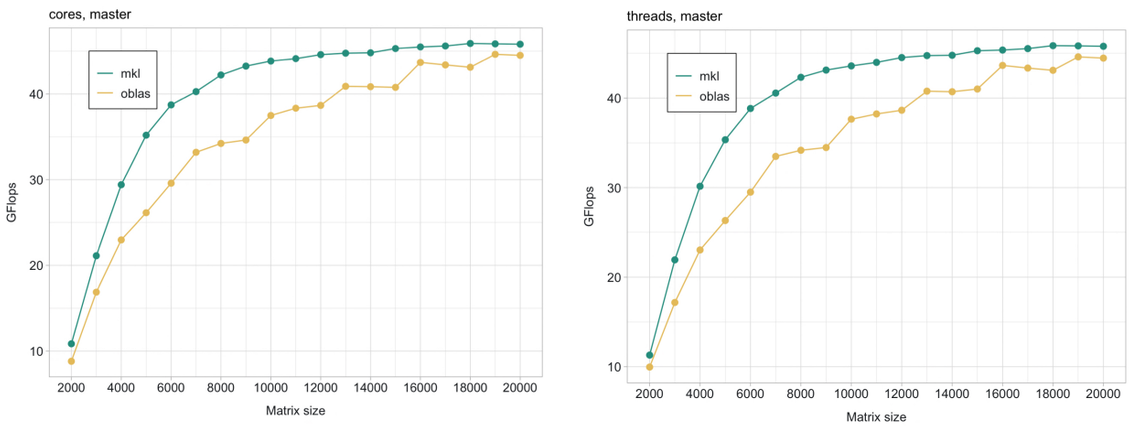
2.2.2.1. Places sockets



The results obtained have been graphically illustrated and we can see that lower calculation power values are obtained compared to the previous single precision tests. Similarly, in this case, the various types of binding policies do not appear to affect the results obtained. Again, inconsistent results may be observed with the error bars.

2.2.3. Palces cores and threads

By using the parameters **places = <cores, threads>** and **bind = <close, spread>**, we can observe a greater increase in computational power for OBLAS compared to MKL.



Analyzing the results obtained using **places = <cores, threads>** and **bind = master**, we can see that in this case, MKL tends to have a slight increase performance compared to OBLAS. Instead before, single precision on OBLAS had obtained significantly higher results compared to MKL.

2.2.4. Single precision vs Double precision

In general single precision is faster than double precision for the following possible reasons (the results obtained depend heavily on the underlying architecture):

- **storage space:** single precision uses only 32 bits to represent a number, while double precision uses 64 bits, which means that the processor needs to handle twice as many bits for double precision.
- **hardware optimization**: Many processors and GPUs are optimized to work with single precision, which makes double precision even slower.

2.2.5. Strong scalability

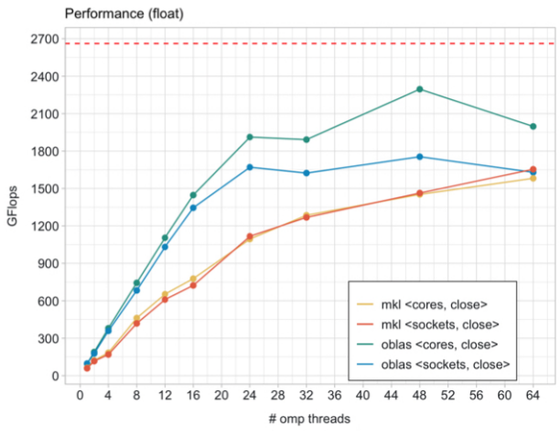
The scalability of the gemm function on an increasing number of cores with a fixed problem size was then analyzed. Given the previous results, a matrix of a fairly high size (14000) was chosen.

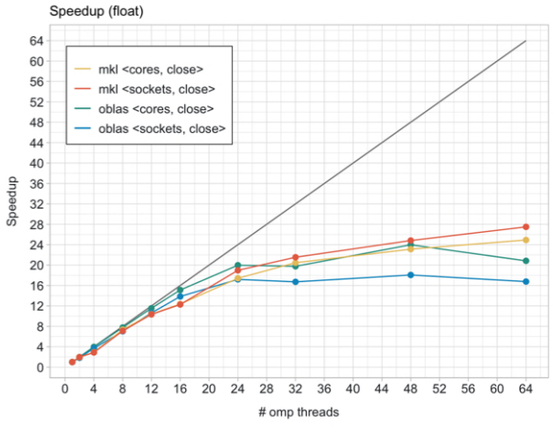
oblas f	sockets, close				cores, close			
threads	gflops	time	speedup	efficiency	gflops	time	speedup	efficiency
1	97.09843	56.525268	1	1	95.7096	57.341612	1	1
2	177.64984	30.900522	1.829266	0.9146329	189.6781	28.933870	1.981816	0.9909081
4	358.78343	15.298667	3.694784	0.9236959	379.9605	14.443869	3.969962	0.9924905
8	683.03839	8.036174	7.033853	0.8792316	743.7497	7.378911	7.771013	0.9713766
12	1031.47132	5.321692	10.621672	0.8851393	1104.9809	4.966678	11.545266	0.9621055
16	1345.23018	4.079668	13.855360	0.8659600	1447.1938	3.792286	15.120592	0.9450370
24	1670.58037	3.285328	17.205364	0.7168902	1912.1089	2.870239	19.977991	0.8324163
32	1623.59653	3.380854	16.719228	0.5224759	1891.6958	2.901218	19.764670	0.6176459
48	1754.48985	3.129278	18.063359	0.3763200	2295.8213	2.391119	23.981076	0.4996058
64	1629.69871	3.370083	16.772663	0.2620729	1997.4658	2.751948	20.836735	0.3255740

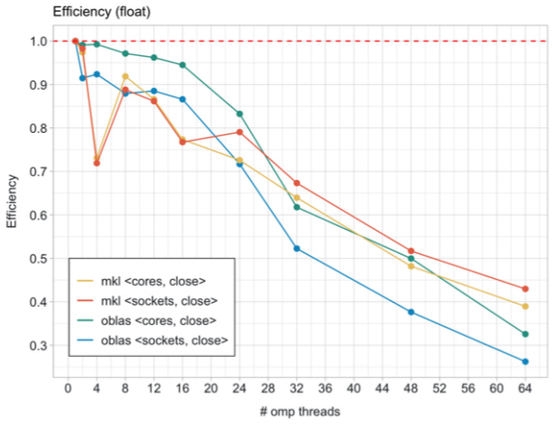
Table 1: OBLAS float strong scalability for matrix of size 14000 x 14000

mkl f	sockets, close				cores, close			
threads	gflops	time	speedup	efficiency	gflops	time	speedup	efficiency
1	58.86909	93.229554	1	1	62.82077	87.361546	1	1
2	116.21242	47.438428	1.965275	0.9826375	122.35698	44.852718	1.947743	0.9738713
4	169.30794	32.416868	2.875958	0.7189895	183.69919	29.875636	2.924174	0.7310434
8	418.44970	13.121123	7.105303	0.8881628	461.78956	11.884464	7.350903	0.9188629
12	609.62931	9.016678	10.339678	0.8616399	652.87508	8.406235	10.392470	0.8660392
16	723.08228	7.593006	12.278345	0.7673966	777.38261	7.059800	12.374508	0.7734067
24	1116.97417	4.914029	18.972120	0.7905050	1094.55975	5.014139	17.423040	0.7259600
32	1268.13834	4.327927	21.541389	0.6731684	1285.66205	4.268742	20.465410	0.6395441
48	1463.74008	3.757062	24.814485	0.5169684	1452.60146	3.778157	23.122795	0.4817249
64	1653.89397	3.391005	27.493189	0.4295811	1581.50861	3.505500	24.921280	0.3893950

Table 2: MKL float strong scalability for matrix of size 14000 x 14000





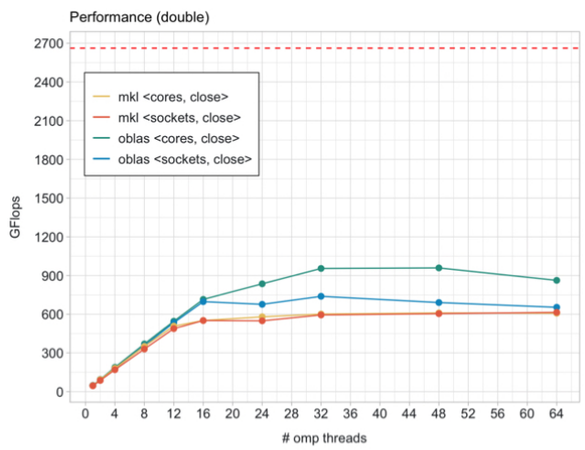


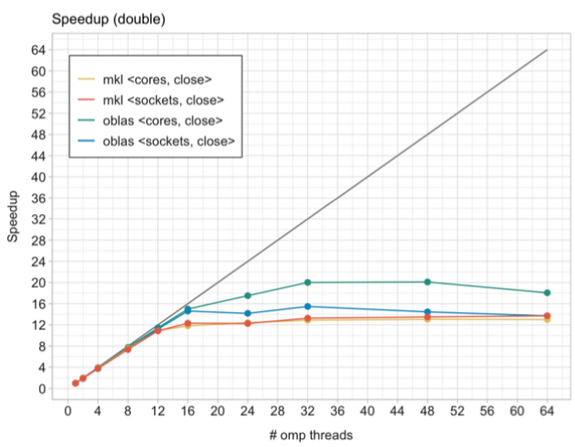
oblas d	sockets, close				cores, close			
threads	**gflops**	**time**	**speedup**	**efficiency**	**gflops**	**time**	**speedup**	**efficiency**
1	47.67515	115.112697	1	1	47.66760	115.131101	1	1
2	90.18997	60.850212	1.891739	0.9458693	94.00879	58.378445	1.972151	0.9860754
4	179.13361	30.638403	3.757138	0.9392844	188.03276	29.186875	3.944619	0.9861547
8	355.27628	15.447191	7.452015	0.9315019	368.75274	14.882612	7.735947	0.9669934
12	534.02661	10.276649	11.201385	0.9334487	545.64510	10.057843	11.446897	0.9539081
16	697.36576	7.869628	14.627464	0.9142165	715.06907	7.674798	15.001190	0.9375744
24	676.58900	8.112675	14.189241	0.5912184	835.47362	6.568736	17.527131	0.7302971
32	738.79432	7.430445	15.492032	0.4841260	954.50594	5.749655	20.024001	0.6257500
48	690.36598	7.951149	14.477492	0.3016144	958.77715	5.724130	20.113293	0.4190269
64	653.73612	8.395197	13.711733	0.2142458	862.24352	6.368381	18.078552	0.2824774

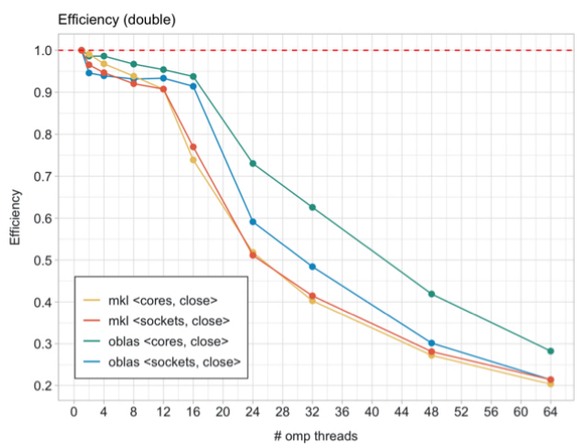
Table 3: OBLAS double strong scalability for matrix of size 14000 x 14000

mkl d	sockets, close				cores, close			
threads	**gflops**	**time**	**speedup**	**efficiency**	**gflops**	**time**	**speedup**	**efficiency**
1	44.77036	122.664157	1	1	46.65119	117.639165	1	1
2	86.37209	63.540395	1.930491	0.9652455	92.45623	59.357951	1.981860	0.9909301
4	169.45343	32.395729	3.786430	0.9466075	180.59913	30.387839	3.871258	0.9678145
8	329.71405	16.658821	7.363316	0.9204145	350.22732	15.669882	7.507342	0.9384177
12	487.87587	11.262632	10.891252	0.9076043	507.83602	10.806654	10.885809	0.9071507
16	551.10968	9.959378	12.316448	0.7697780	551.44080	9.952134	11.820496	0.7387810
24	548.98998	9.998158	12.268676	0.5111948	580.15441	9.460030	12.435390	0.5181412
32	594.30762	9.243221	13.270716	0.4147099	601.28388	9.127149	12.888928	0.4027790
48	604.29933	9.085411	13.501223	0.2812755	609.89583	8.998275	13.073524	0.2723651
64	614.63440	8.934656	13.729029	0.2145161	607.92887	9.029114	13.028871	0.2035761

Table 4: MKL double strong scalability for matrix of size 14000 x 14000







From the results obtained through the strong scalability analyses, we can deduce that:

- OBLAS with **places = cores** and **bind = close** achieves better performance in both the single precision case (elbow point with 24 threads [1900-2000] GFlops) and the double precision

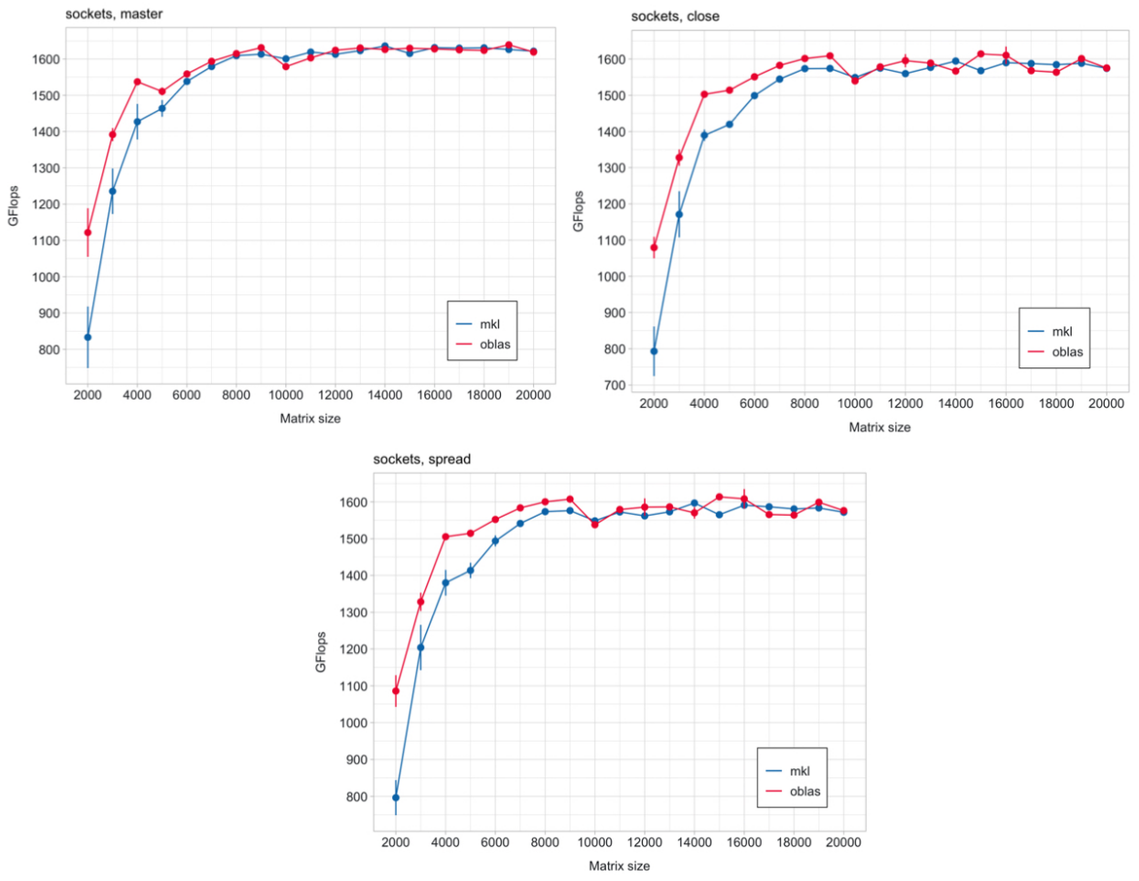
case (elbow point with 16 threads [700-800] GFlops).

- Regarding the speedup, in the case of single precision, MKL performs slightly better, while in the case of double precision, OBLAS achieves better results.

2.3. Thin INTEL node

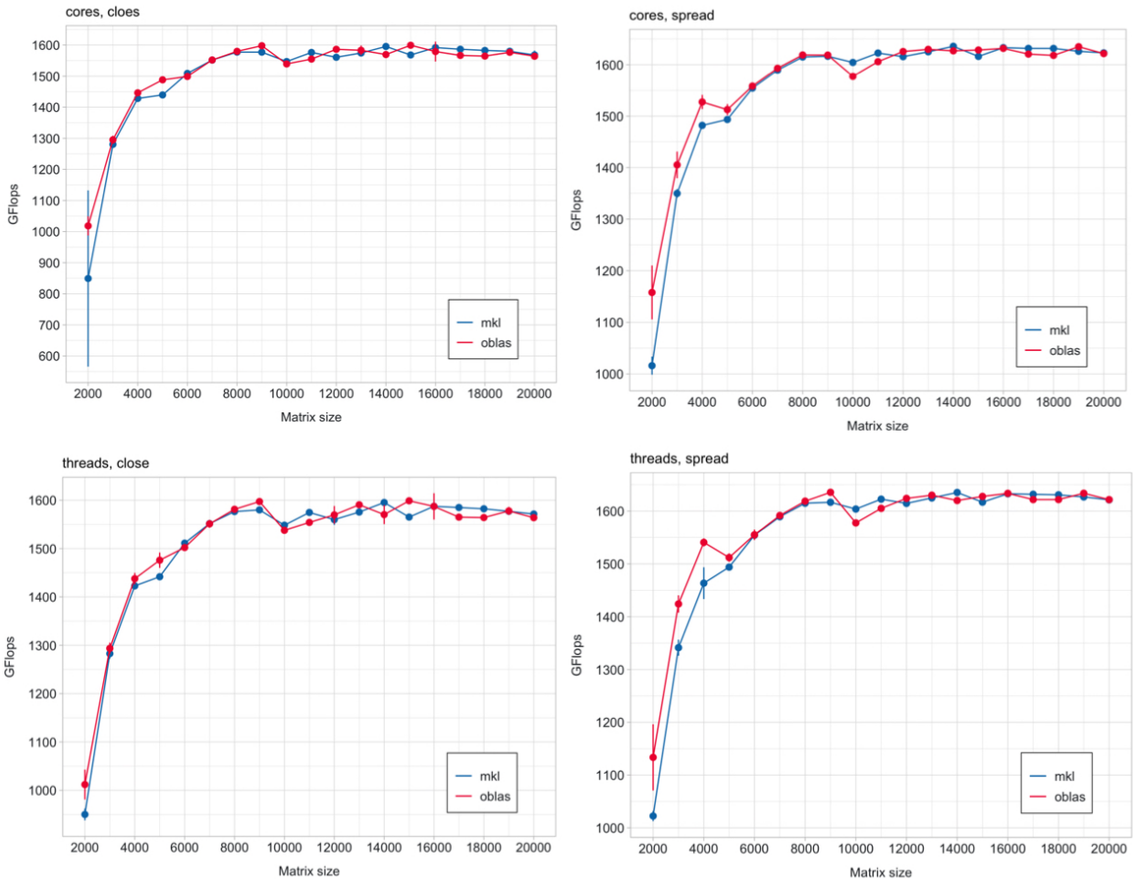
From the analysis performed on the **epyc[005]** node, highly variable results were obtained that depend not only on the algorithm used but also on the underlying architecture. The same tests previously performed are reported here on the INTEL **thin[007]** node. Through the **`'srun lscpu | grep "Thread(s) per core"'`** command, it was found that hyperthreading is disabled on the **thin[007]** node.

2.3.1. Places sockets

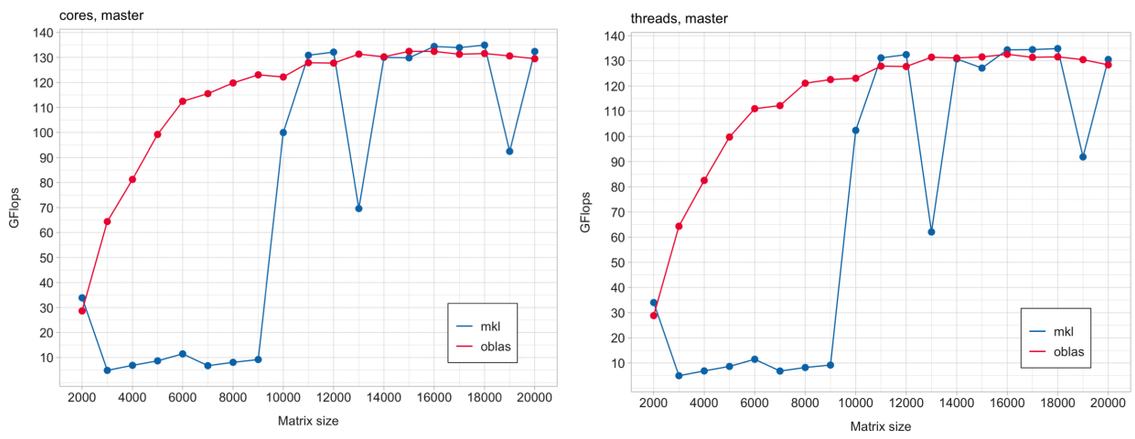


By analysing the algorithms using the parameters **places = sockets** and **bind = <close, master, spread>** similar results are obtained between OBLAS and MKL. OBLAS tends to be more effective when using a small matrix. The Elbow point occurs in the range of [1500-1600] Gflops. Moreover, contrary to EPYC nodes, the results are more stable.

2.3.2. Places cores and threads



The results obtained with the parameters **places = <cores, threads>** and **bind = <close, spread>** are similar to the ones obtained with **places = sockets**. Again, the elbow point is reached at [1500-1600] GFlops with matrices dimension of 8000.



In the results obtained with **places = <cores, threads>** and **bind = master** OBLAS reached the Elbow point in the range of [110-130] GFlops with matrices dimension of 6000. Notice that MKL gave unreliable results.

2.4. Conclusions

In conclusion, the analysis shows that the OBLAS library is slightly more performant than the MKL library when it comes to matrix multiplication. This means that if performance is a priority, the OBLAS library may be the better choice for matrix multiplication operations. However, it's important to consider other factors, such as the architecture used since in the case of an Intel node similar results were obtained.

The analysis shows that single precision computations are more performant than double precision computations. That's because single precision requires less memory and computation time, making

it faster and more efficient for certain types of applications. However, it's important to keep in mind that double precision provides higher precision and accuracy, so it may still be necessary for certain types of computations that require more accurate results. The choice between single and double precision will ultimately depend on the specific requirements of the application and the trade-off between speed and precision.

The test results on the AMD epyc[005] node were slightly inconsistent with errors. On the other hand, the INTEL thin[007] node obtained stable results except for the last case.