Foundations of HPC - Assignment

Filippo Olivo

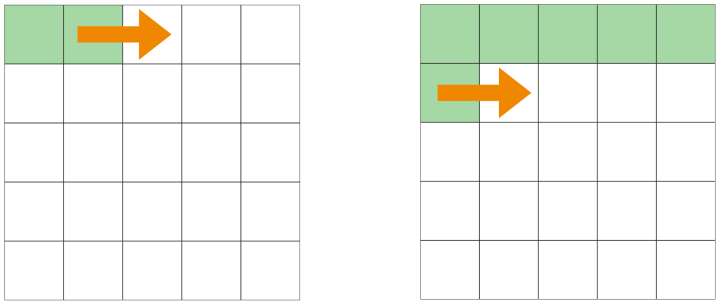
Table of contents

Excercise 1

Introduction

The aim of the excercise is to implement a parallel version of the Conway's Game of Life through MPI and OpenMP. Game of life is a zero player game which evolution depends only on the initial conditions. In this project I have implemented three possible iteration tecniques:

- Static iteration: we freeze the system at each state s_i and then we evaluate the new cell status s_{i+1} based on the system at state s_i
- Ordered iteration: we start from the cell (0,0) and we evolve by lines



- Wave iteration: wave that spread in diagonal from cell (0,0) (futher details below)

Metodology

The program is written in C language and it can perform the three types of operations explained in the previous section. The world must be read from a `pgm` file and which is converted in arrays of `unsigned char`. The choice of using a matrix of `unsigned char` has been made to reduce the usage of RAM. Inside the program the alive cells are represented with the number 0, while the dead cells are represented with the number 255, respectively the black and the white colour in the output grid. The program consist in mainly four modules:

1. Read and write: a support module that read/write from/to a `pgm` file. In order to distribute the data across the nodes the reading operation is done in parallel (if the code is runned on more than one MPI Task)
2. Run static: implement the static iteration already mentioned above.
3. Run ordered: implement the ordered iteration already mentioned above. Ordered iterations are by definition serial and, for this reason, I have only implemented a domain decomposition across the nodes (see below for futher details)
4. Run wave: implement the wave iteration already mentioned above. This implmenetation is done only with OpenMP (see below for further details)

To evaluate the performances I have run multiple test on the Epyc and Thin nodes of the orfeo cluster changing properly the number of MPI processes and OpenMP threads. In this section I am mainly interested in the speed up described by the formula below:

$$\text{Speed up} = \frac{T_s}{T_{np}}$$

Where T_s is the time taken by a single process and T_{np} is the time taken by n processes (note that the ideal speed up is given by Speed up $= n$ (n is the number of processes)

Implementation

In this section I discuss some technical aspects of the code, in particular I focus on the parallelization methods.

Read and write operations

Reading from `pgm` files operation is performed in parallel: from a single `pgm` file the master thread reads the header, captures the parameter, such as the size of the matrix, the max value, and the type of the file, and broadcasts them to other MPI Tasks. Each process reads and stores inside an array of `unsigned char` a matrix of size equal to $size_{world}$xR_i where R_i is the number of rows, calculated by the formula:

$$R_i = \begin{cases} \text{floor}\left(\frac{size_{world}}{n_{MPI\ Process}}\right) + 1 & \text{if module}(size_{world}, n_{MPI\ Process}) > 0 \\ \text{floor}\left(\frac{size_{world}}{n_{MPI\ Process}}\right) & \text{otherwise} \end{cases}$$

The reading is performed through MPI I/O and more precisely by calling the function `MPI_File_read`

Writing is performed in parallel: each MPI Task writes its output in a separate file and a bash script perform the joining of the files.

Initialization

The initialization part is performed in parallel. The work is subdivided among the MPI Tasks and more precisely each of those has to initialize a submatrix of size $size_{world} \cdot R_i$, where R_i is a positive number given by the formula:

$$R_i = \begin{cases} \text{floor}\left(\frac{size_{world}}{n_{MPI\ Process}}\right) + 1 & \text{if module}(size_{world}, n_{MPI\ Process}) > 0 \\ \text{floor}\left(\frac{size_{world}}{n_{MPI\ Process}}\right) & \text{otherwise} \end{cases}$$

A for loop, parallelized through the usage of OpenMP, perform the initialization of the submatrix. The scheduling used for the loop paralleliization is static due to the balanced workload that each thread has to perform at each iteration. Increasing the chunk size does not improve the overall performances and, for this reason, I have decided to keep it to the default value (chunk size = 1). After the initialization, the OpenMP parallel region ends. Each MPI Tasks, in the correct order, writes on the target file its submtrix (the process 0 write also the initial lines that specify the type of file and the dimension of the image).

Static iteration method

The apporch used to parallelize this type of operation is the domain decomposition: each MPI Task has to work on a submatrix of size $size_{world} \times R_i$ as defined in the previous subsection. In order to allow each MPI Task to calculate the new status of its cells, the MPI Task needs to have also the last line of the submatrix of the process before (rank-1 or size-1 for the first process) and the first line of the submatrix of the process after (rank+1 or 0 for the process size-1). Below is shown an example with a matrix 6x6 and 3 MPI Processes (in this case this process has a submatrix of size 2x3 and 3 additional rows).

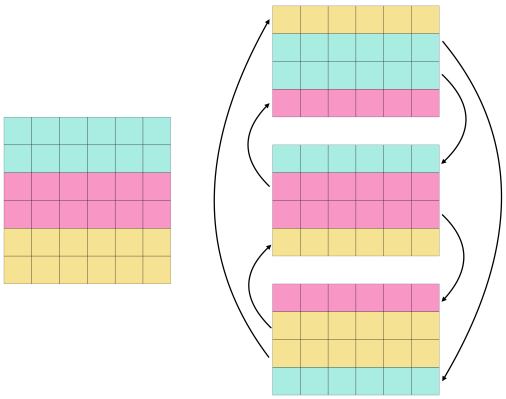


Figure 1: Domain decomposition

Each process create another matrix of the same size of the previous one. The idea is to calculate the status s_{i+1} from the matrix of the status s_i. At the beggining of each iteration each MPI Process:

- Send (through a non-blocking send) its first and last rows
- Received from/to the other processes the necessary rows
- Store them in the local matrix that contains the values of the iteration s_i

In order to avoid errors each at iteration the tag changes. It could happen that a MPI Task is one complete iteration in advance than the following or previous MPI Tasks. Find above the lines that perform the send and receive of the rows:

```
int tag_odd = 2*it;
int tag_even = 2*it+1;
//each process send his fist and last row to respectively the process with rank-1 and rank + 1.
//Process 0 send his fist line to process size -1
//Process size-1 send his last row to process 0
if(rank == size-1){
  MPI_Isend(&world1[world_size],world_size,MPI_UNSIGNED_CHAR,rank-1,tag_odd,MPI_COMM_WORLD,r);
  MPI_Isend(&world1[(local_rows)*world_size],world_size,MPI_UNSIGNED_CHAR,0,tag_even,MPI_COMM_WORLD,r);
  MPI_Recv(world1,world_size,MPI_UNSIGNED_CHAR,rank-1,tag_even,MPI_COMM_WORLD,s);
  MPI_Recv(&world1[(local_rows+1)*world_size],world_size,MPI_UNSIGNED_CHAR,0,tag_odd,MPI_COMM_WORLD,s);
}
if(rank == 0){
  MPI_Isend(&world1[(local_rows)*world_size],world_size,MPI_UNSIGNED_CHAR,1,tag_even,MPI_COMM_WORLD,r);
  MPI_Isend(&world1[world_size],world_size,MPI_UNSIGNED_CHAR,size-1,tag_odd,MPI_COMM_WORLD,r);
  MPI_Recv(world1,world_size,MPI_UNSIGNED_CHAR,size-1,tag_even,MPI_COMM_WORLD,s);
  MPI_Recv(&world1[(local_rows+1)*world_size],world_size,MPI_UNSIGNED_CHAR,1,tag_odd,MPI_COMM_WORLD,s);
}
if(rank != 0 & rank != size-1){
  MPI_Isend(&world1[(local_rows)*world_size],world_size,MPI_UNSIGNED_CHAR,rank+1,tag_even,MPI_COMM_WORLD,r);
  MPI_Isend(&world1[world_size],world_size,MPI_UNSIGNED_CHAR,rank-1,tag_odd,MPI_COMM_WORLD,r);
  MPI_Recv(&world1[(local_rows+1)*world_size],world_size,MPI_UNSIGNED_CHAR,rank+1,tag_odd,MPI_COMM_WORLD,s);
  MPI_Recv(world1,world_size,MPI_UNSIGNED_CHAR,rank-1,tag_even,MPI_COMM_WORLD,s);
}
```

Figure 2: Send and receive process

Then the update method is called it consists in a for loop which, for each cell of status s_{i+1}:

- read the value of all the neighbour from matrix of status s_i
- calculate the value of the new status and write it on the new matrix

This for loop is parallelized with OpenMP: each MPI Task has to update a bunch of cells independently from other OpenMP threads. The best scheduling policy is the static one and this is reasonable due to the almost costant workload of the iterations. As it concern the chunck size, an increase of it does not bring a significant improvement of the overall performances of the program. For this reason, I have decided to keep it at the default value (chunck size = 1). Note that the program keeps in memory only the matrix of the actual and the previous status.

Ordered iteration method

Similar to the static iteration method, the ordered iteration method uses a domain decomposition approch. As in the previous case each process reads a portion of the world. Before starting the iteration process, the last MPI Task sends respectively its last row to process 0 and each process sends his first row to the previous process. Each process, at the end of the elaboration, sends its last row to the previous process. Note that in this case an MPI task start the computation when it receives the last row of the following process.

In this way I expect that the parallelization does not improve the computational time but we have to consider that with this type domain decomposition the memory scales. Because of the fact that each MPI Task has its submatrix directly read from the file the overall size of the matrix can be larger then the RAM of a single node. In the serial problem the entire world must fit inside the RAM of a single node.

Wave iteration method

The data structure that are needed for this method are two matrices of size $size_{world} \times size_{world}$ and an array of struct Cell (below the definition).

```
struct Cell{
  long row;
  long col;
};
```

A single iteration starts with the update of the cell (0,0) of the world. After the update, a struct Cell variable is created for all the neightbour of the cell (0,0) and stored in an array. The only purpose of the struct is to represent in a more simple way the cells that must be updated in the next step of the iteration.

Before starting the next step of the single iteration, also the matrix that represent the previous status of the world is updated with the new values. On the following step each OpenMP thread updates a bunch of Cells presents in the array using the previous state world (`for` loops parallelized through OpenMP). This procedure continues until the last row and column are updated.

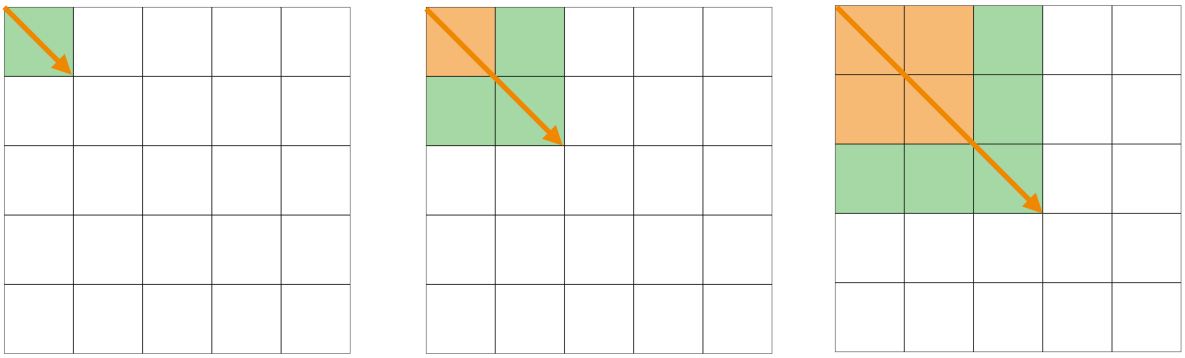


Figure 3: Wave iteration

The best schedule of the for loop is static, as expected, due to the balanced workload of each iteration of the for loop. I have chosen to use chunck size equal to 1 in order to parallelize the the update procudure starting from the second step of each iteration.

Code validation

To asses that the code works properly I have started considering a very small problem with size 10×10 and the serial version of the program. First I have updated the initial world with only one time and assess that the result was correct. After having assessed that the serial

program works properly for a single iteration I have tried to do two iterations and also in this case I manually verified the output. Below the used example:

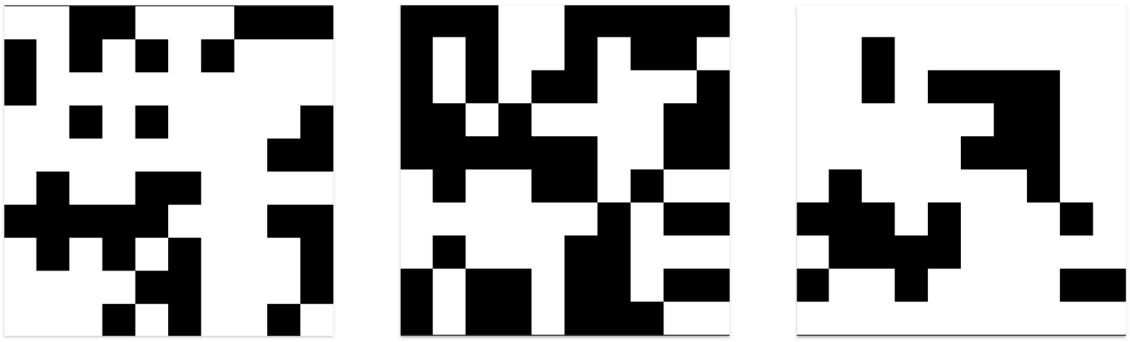


Figure 4: Test run static

After this procedure I have assesed that the serial cose works properly. To test the parallel computation I have calculated the sha256 for the serial and the parallel output (for different number of MPI tasks, OpenMP threads and size of the world) and verified that these two values were equal.

Result and performances

I have performed several test to evaluate the computation time and the speedup of the different type of evolution, with different size and with different OpenMP and MPI options. The time has been taken from the start to the end of the MPI parallel region, which almost coincide with the overall time of the program.

Iterate static

The measure made for this type of iteration method are:

- **OpenMP Scalability**
- **MPI Strong Scalability**
- **Weak MPI Scalability**

All the measurements in this section have been made on the Epyc and Thin node of the High performance computing cluster Orfeo whithout the SMT.

OpenMP Scalability

We want to measure how much the program scales when we increase the number of OpenMP threads. The test has been made with the OpenMP option `OMP_PROC_BIND=close` and I have placed one single MPI Task for each socket with the option `--map-by socket` when I have used 1 or 2 sockets on the same machine and with the options `--map-by node --bind-to socket` when I have used more than 3 socket (up to 6 socket for Epyc nodes and up to 4 for Thin node). For the Epyc node I have tested two different sizes of the matrix with different number of iterations:

- Size: 25000×25000 and 50 iterations
- Size: 25000×25000 and 100 iterations
- Size 12500×125000 and 50 iterations

The graphs below show the OpenMP scalability for the epyc nodes:

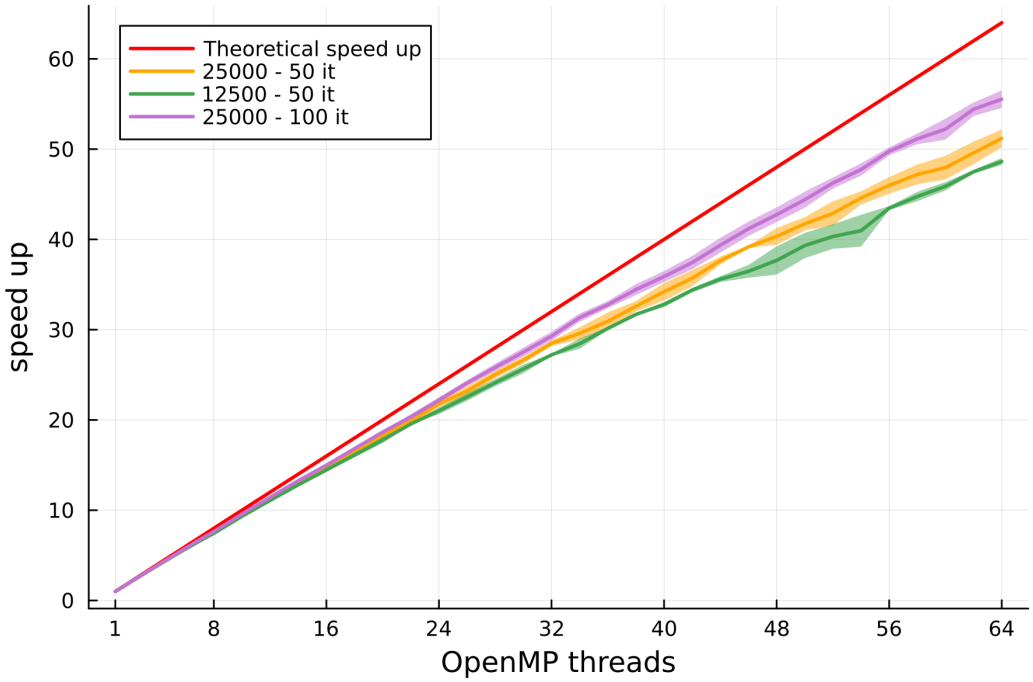


Figure 5: OpenMP scalability - 1 socket

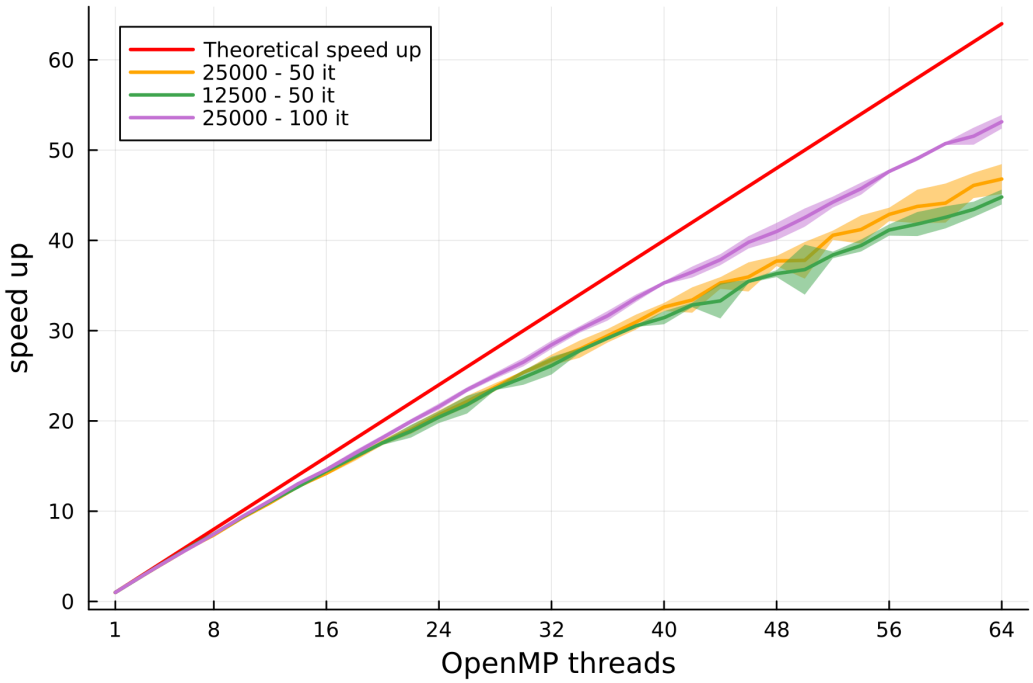


Figure 6: OpenMP scalability - 2 sockets

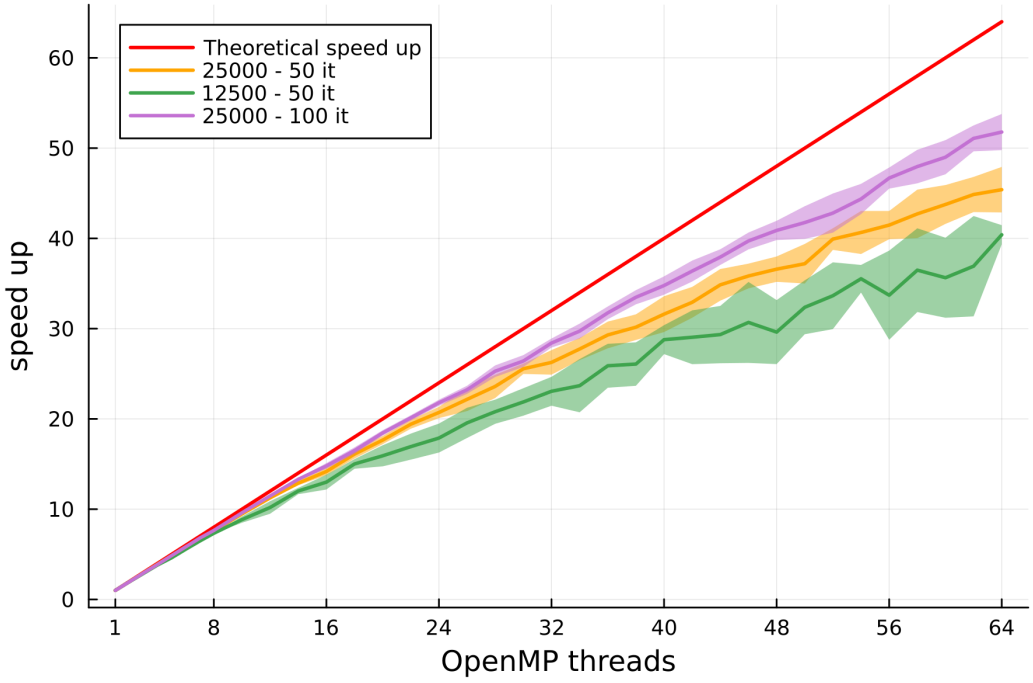


Figure 7: OpenMP scalability - 3 sockets

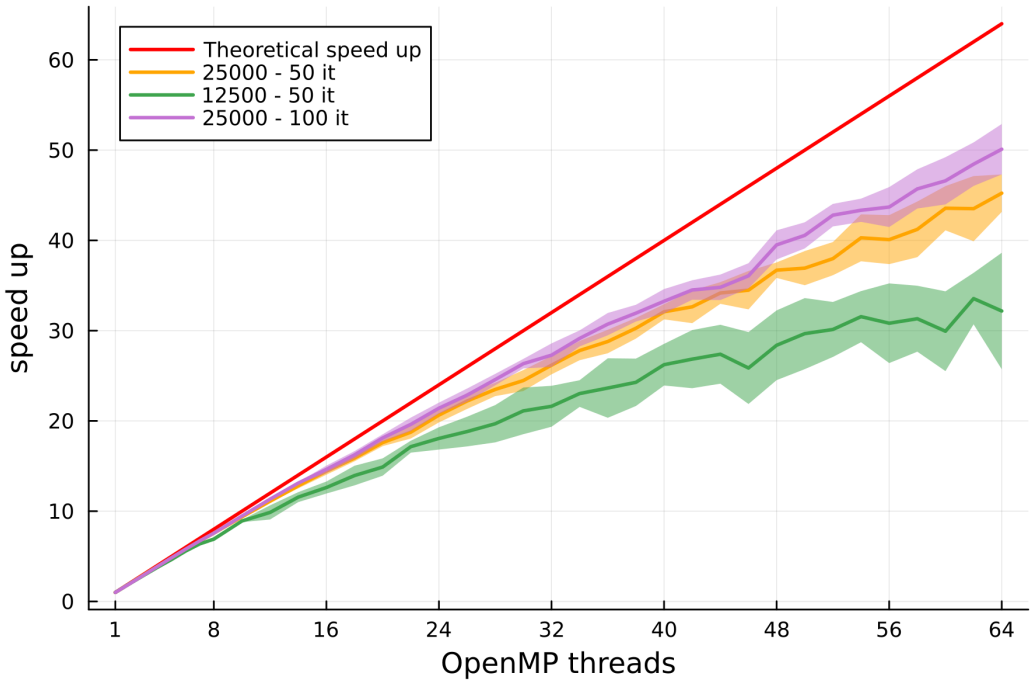


Figure 8: OpenMP scalability - 4 sockets

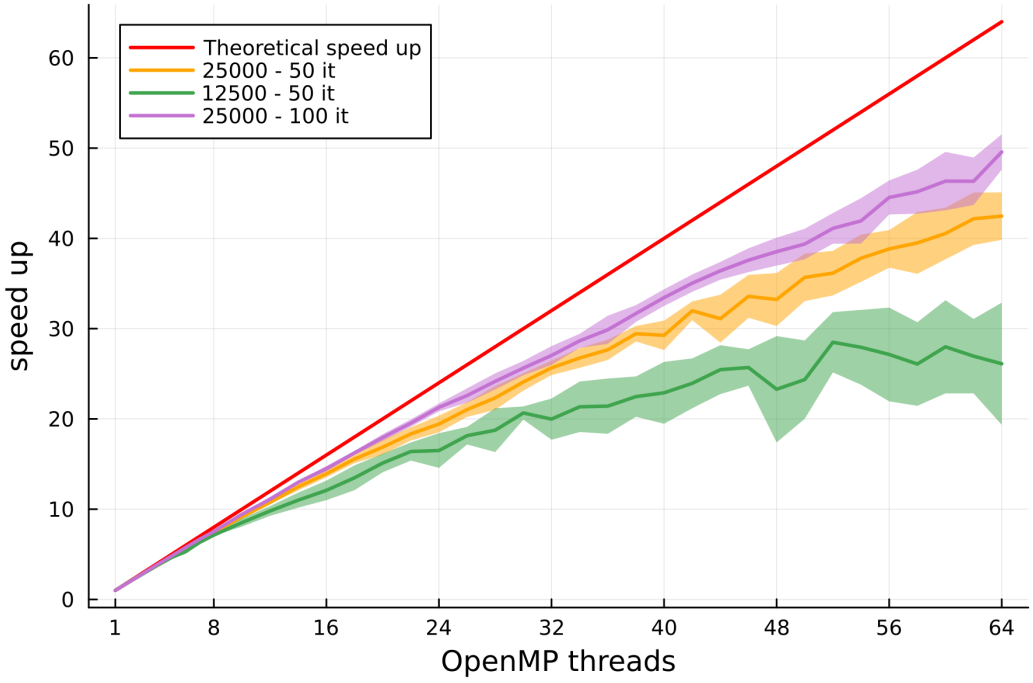


Figure 9: OpenMP scalability - 5 sockets

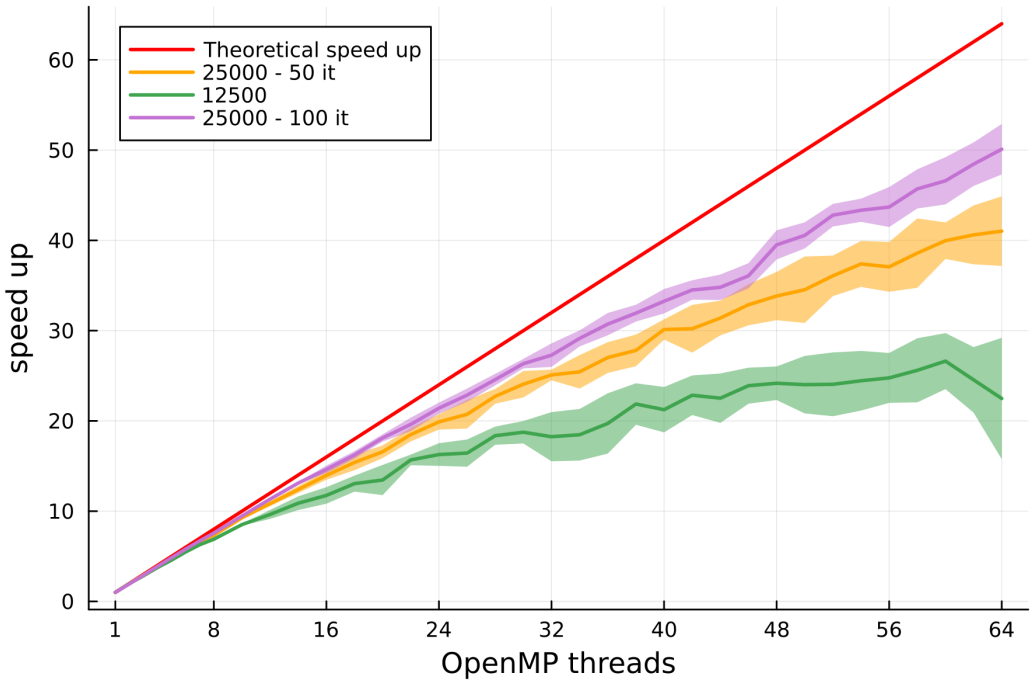


Figure 10: OpenMP scalability - 6 sockets

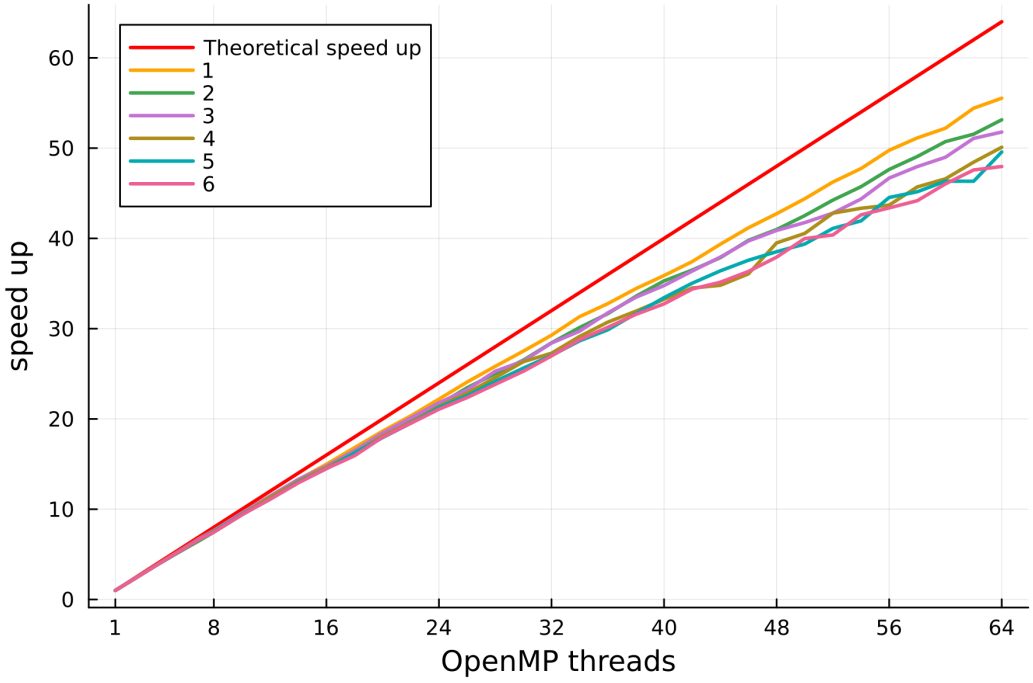


Figure 11: OpenMP scalability - comparison matrix 25000x25000, 100 it

As we can see from all the plots we have that as the workload increases, the speed up increases too. We can also see that as the number of MPI cores increases the speedup decreses. For example if we consider on epyc node the line of the matrix of size 25000×25000 the speedup at 64 OpenMP threads with 1 MPI Task is about 52 indeed for 6 MPI Task the speed up is around 48 (Figure[10]). This difference is even larger if we consider the other two cases.

For the Thin node I have tested different two sizes of the matrix with different number of iterations:

- Size: 12500×12500 and 50 iterations
- Size: 12500×12500 and 100 iterations
- Size 62500×62500 and 50 iterations

Below the graph of the scalability for the thin nodes:

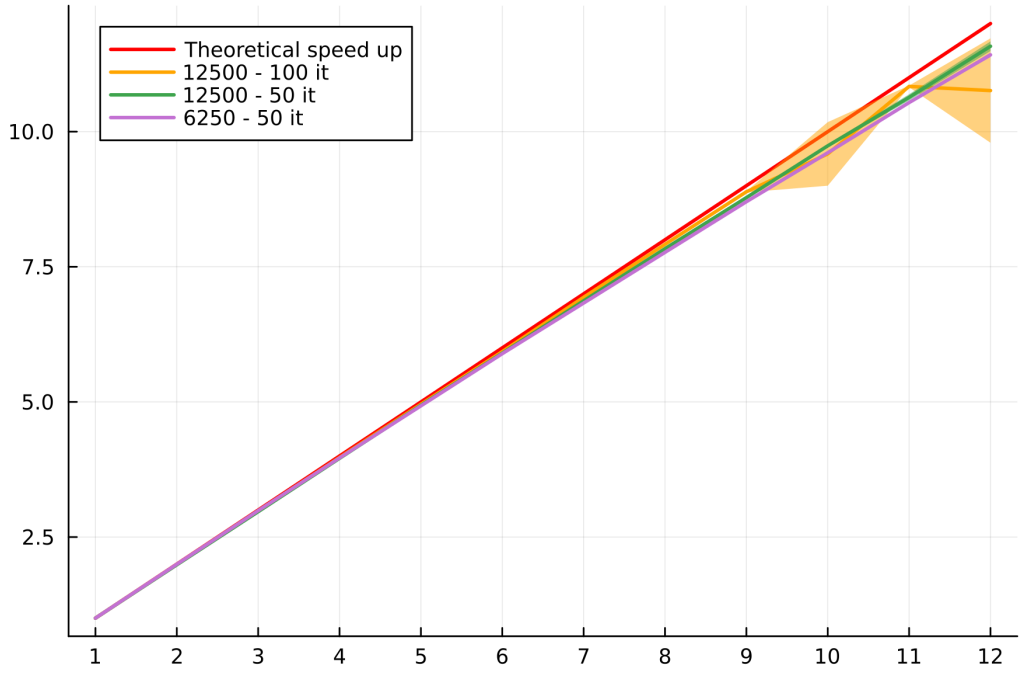


Figure 12: OpenMP scalability Thin - 1 socket

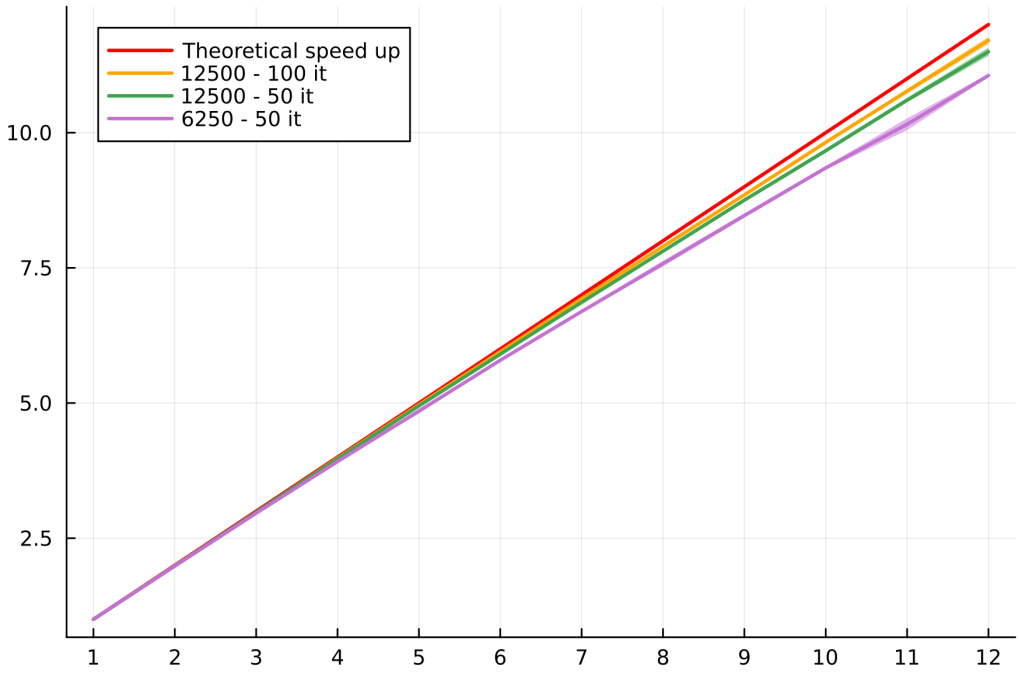


Figure 13: OpenMP scalability Thin - 2 sockets

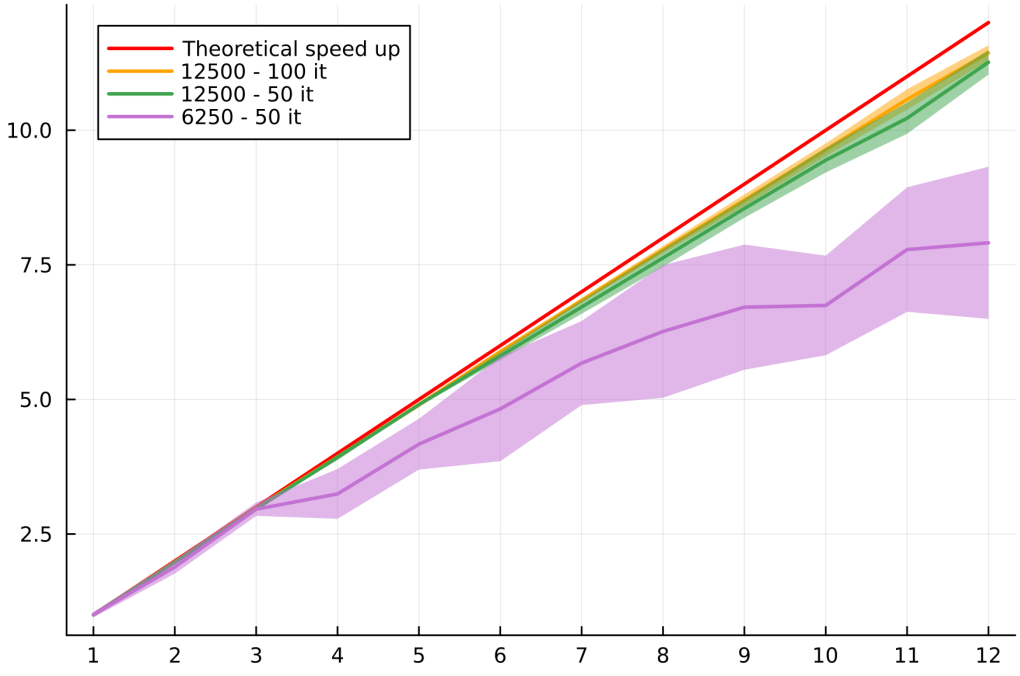
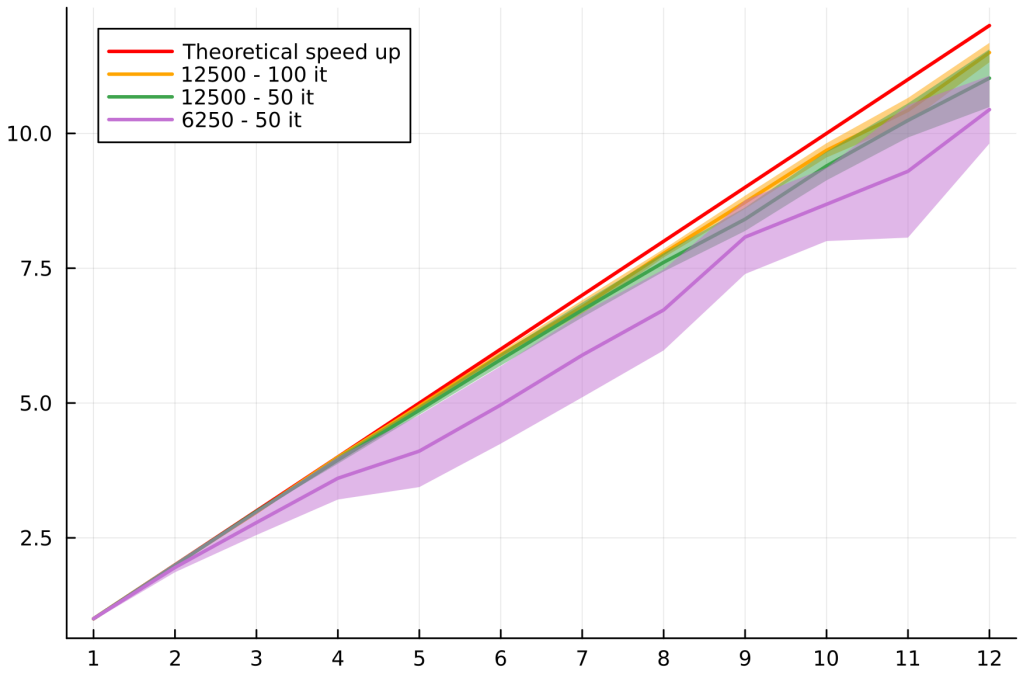


Figure 14: OpenMP scalability Thin - 3 sockets


Theoretical speed up
12500 - 100 it
12500 - 50 it
6250 - 50 it
10.0
7.5
5.0
2.5
1
2
3
4
5
6
7
8
9
10
11
12

All the observations made for Epyc nodes are valid, even, on the Thin nodes

MPI weak scalability

In this section the main idea is to mantain the workload for each MPI Task constant. For this purpose I have decided to increase the size of the matrix: the workload is basically determine by the size of the matrix that each MPI Task has to work on. The formula that I have used to determine the size of the matrix is:

$$x \cdot \frac{x}{n} = s_{serial}^2$$

Where s_{serial} is the size of a column/row of the original world. Below the size of the world that I have used for the analysis:

MPI Tasks	Size
1	10000
2	14143
3	17321
4	20000
5	22361
6	24495

Below two graphs (using respectively 1 OpenMP thread and 32 OpenMP threads) that shows the time of elaboration for the different number of MPI Tasks). More graphs of this kind are available on the github repository. As expected we can see that the computation time is almost constant.

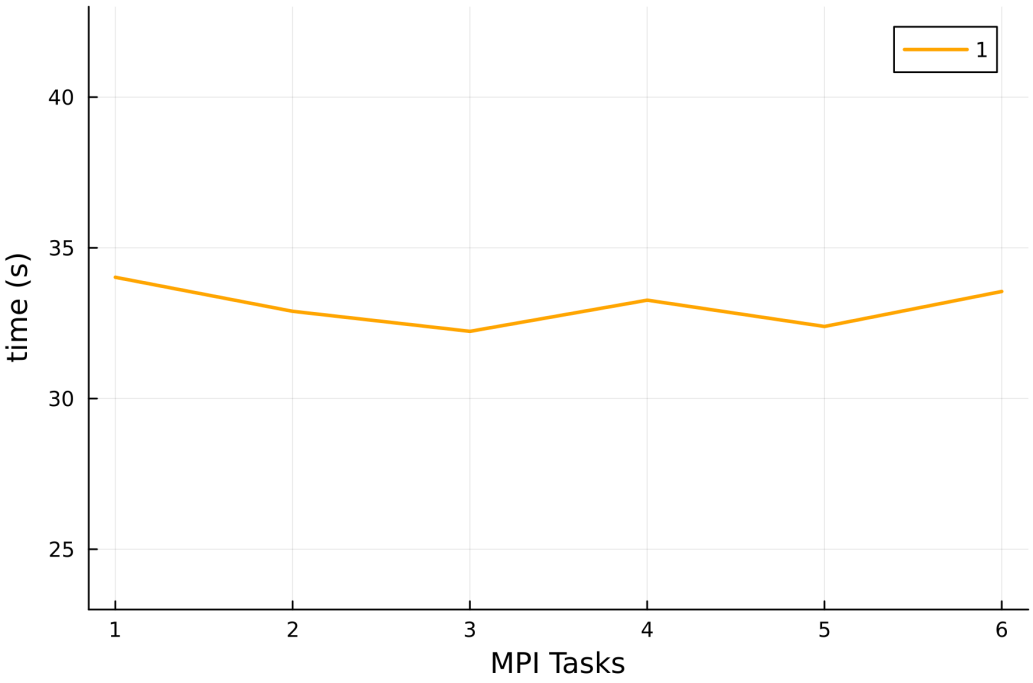


Figure 15: MPI Weak scalability - 1 OpenMP thread

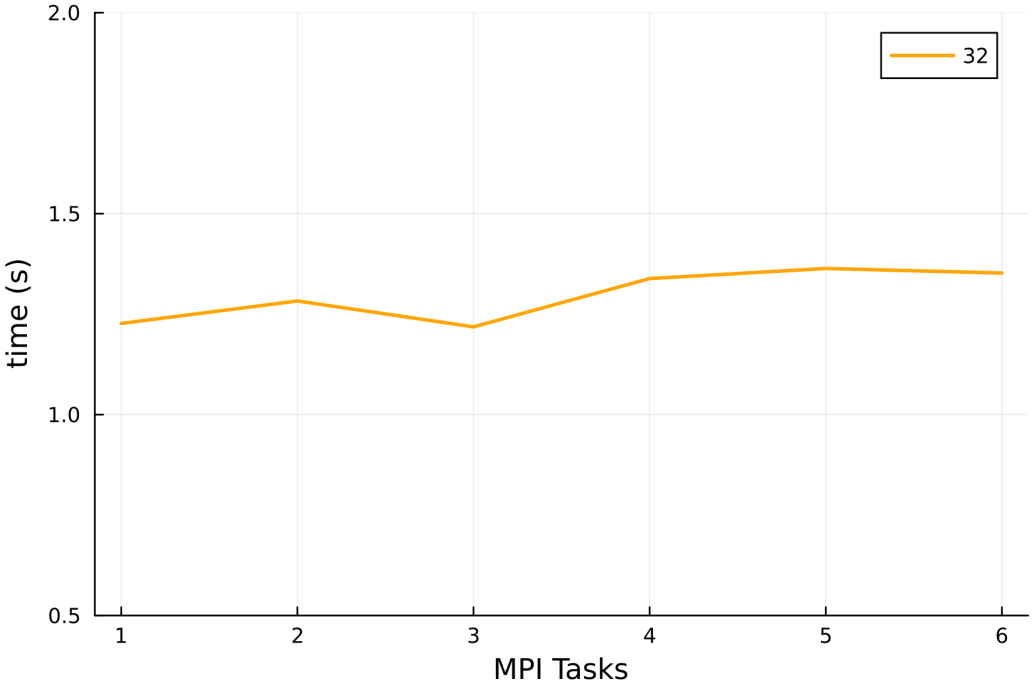


Figure 16: MPI Weak scalability - 32 OpenMP threads

Strong MPI scalability

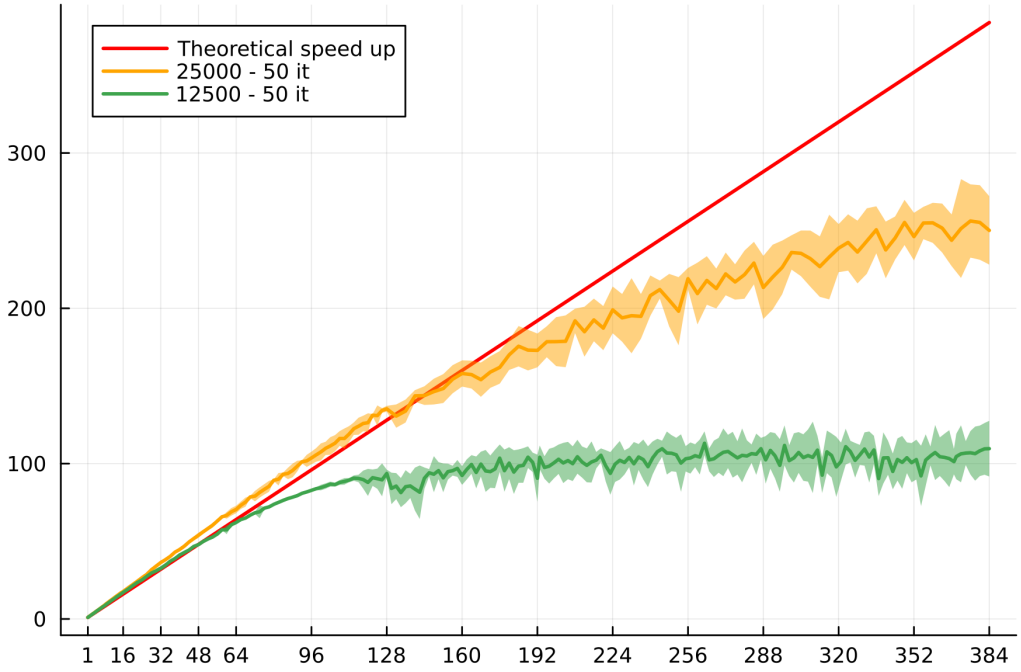
In this section the main idea is to test how the code scales when the number of MPI Tasks increses on a fixed size world. To do so I have run the program using the option `--map-by core` to place a different MPI Task on each core. The test has been perform on epyc nodes with the following parameters:

- Size=25000 $\times$ 250000, it = 50
- Size=25000 $\times$ 250000, it = 100
- Size=12500 $\times$ 125000, it = 100

And on thin node with :

- Size 12500 $\times$ 12500 and 50 iterations
- Size 12500 $\times$ 12500 and 100 iterations

Below the graphs:



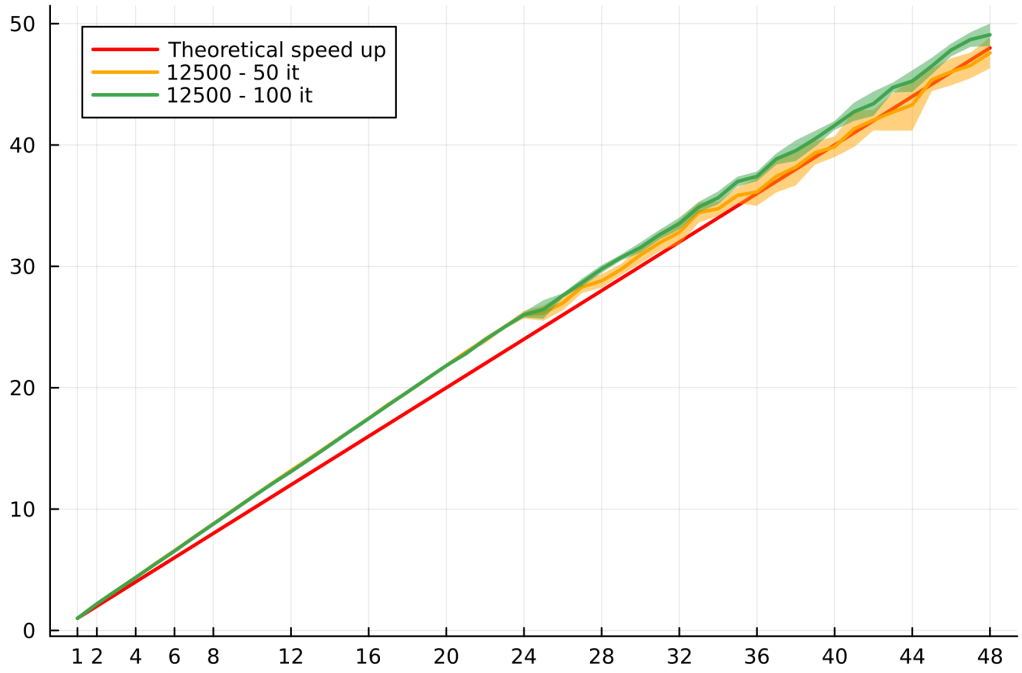


Figure 17: MPI Strong scalability - Thin

We can see, both on epyc and thin nodes, that before "*running out*" of the first node (48 MPI Tasks for thin and 128 MPI Tasks for epyc) the speed up is above the ideal speed up. Another important thing is that, similairly to the OpenMP scalability, the speedup is higher when the workload is higher.

Iterate ordered

The program is strictly serial and, for this reason, I expect that the elapsed time is constant, or increase due to parallelization overhead, when we increases the numeber of MPI Tasks. Also a small increment on the performances are not excluded due to better usage of the cache

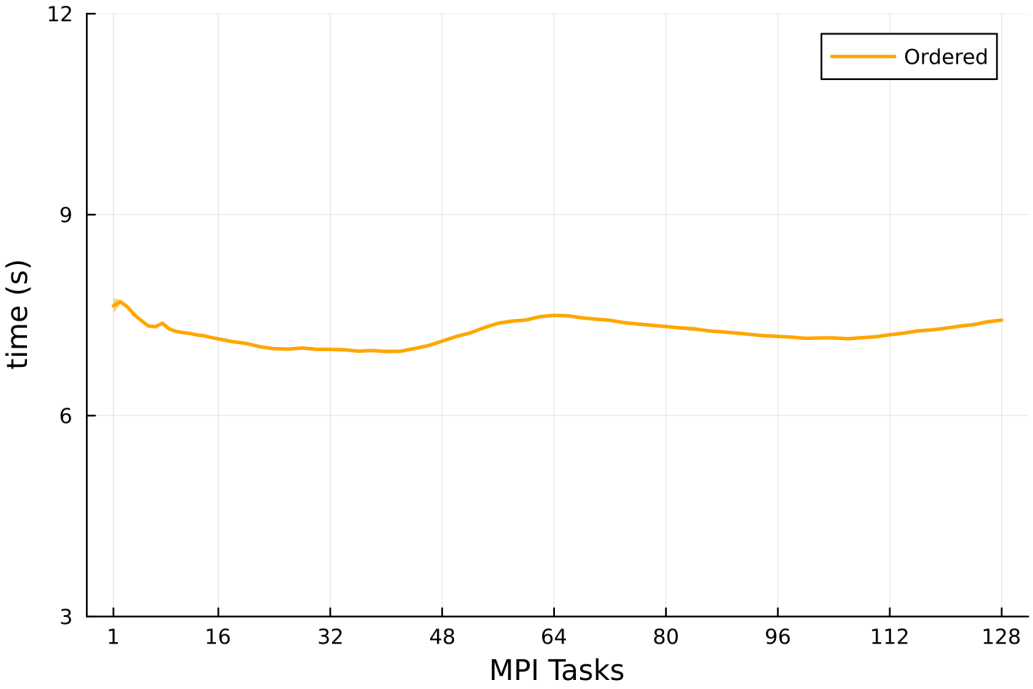


Figure 18: Ordered Iteration - Epyc

Iterate wave

The test has been performed on a world of size 10000 × 10000 and 50 as number of iteration from 1 to 64 OpenMP threads. The OpenMP options were `OMP_PLACES=cores` end `OMP_PROC_BIND=close`. The green in the plot below represent the scalability when no numactl option have been set, while for the red line I have used the following policy

- OpenMP threads from 1 to 16: `--interleave=0`
- OpenMP threads from 17 to 32: `--interleave=0,1`
- OpenMP threads from 33 to 48: `--interleave=0,1,2`
- OpenMP threads from 49 to 64: `--interleave=0,1,2,3`

If we consider the green line we can se that the speed up is higher than the theoretical one for the number of core in the range (14,36). Something similar happen for the red line but fot the number of cores in the range (16,24). We can also se that the speed up without the usage of numactl is higher until 32 OpenMP threads. When we pass from 32 to 34 threads the speed up represented by the red line (wave iteration with the usage of numactl) increase drastically.

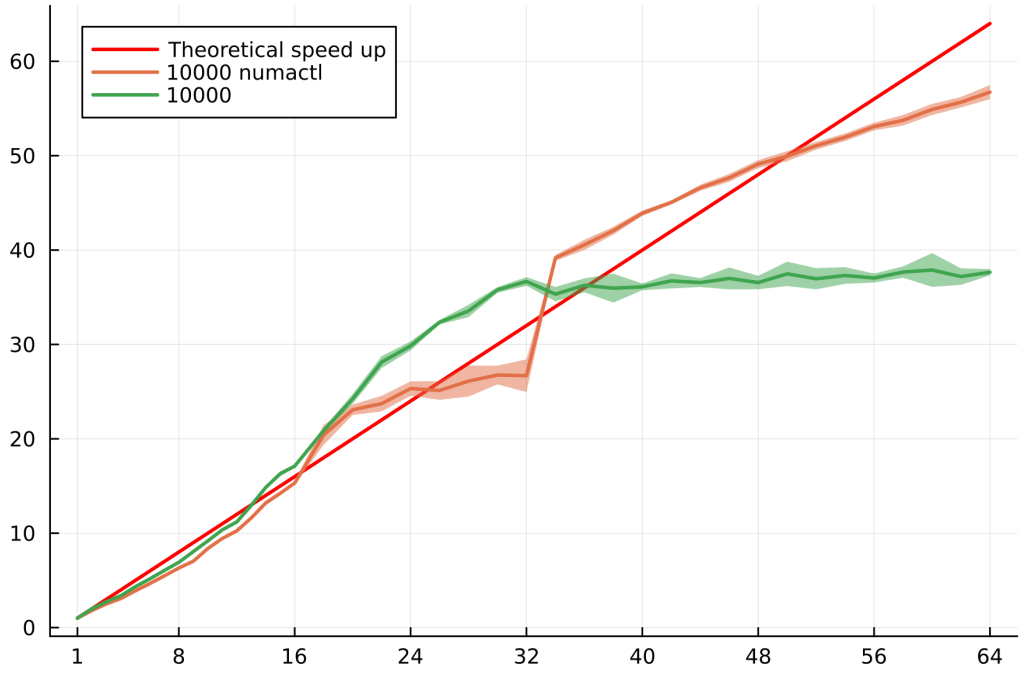


Figure 19: Wave Iteration - Epyc

Final considerations

At the end of this analysis I can say that program achived a good level of scalability both in wave and static iteration methods. As it concerns the ordered iteration my conclusion is that it is stricly serial and the only possible parallelization is the one presented in the section above which give as memory scalability.

Some possible improvements could be:

- Optimization of the writing procedure through MPI I/O.
- Parallelize the reading from the file with openMP in order to improve also the memory allocation

Excercise 2

The goal of the excercise is to compare the performances of the math libraries OpenBlas, Blis and MKL on the epyc node. The metric that has been used to compare the performances is the number of floating point operations per second, measured in GFlops. We are mainly intrestated in analyzing the size scaling and the core scaling of the three algotithms. To achive

the goal I have used a slightly modified version of the code `dgemm.c` present in the Assignment folder.

Size scaling

For the size scaling I have increased the size of the matrix from 2000×20000 to 20000×20000 by steps of 500 (at each step I have increased both the number of rows and the numbrer of columns in order to have always square matrices). For each size and math library I have taken 10 different measurment.

Below the tested policies (default options: `OMP_NUM_THREADS=64` and `OMP_PLACES=cores`):

- `OMP_PROC_BIND=close`, no numaclt options
- `OMP_PROC_BIND=close`, `--interleave=0,1,2,3`
- `OMP_PROC_BIND=spred`, no numaclt options
- `OMP_PROC_BIND=close`, `--interleave=0,1,2,3,4,5,6,7`

Single precison

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

$$P = n.cores \cdot frequency \cdot \frac{FLOPC}{cycle} = 64 \cdot 2.6GHz \cdot 32Flops = 5324,8GFlops$$

A AMD Epyc 7H12 (the one that we can find on orfeo cluster) can deliver 16 double precision $\frac{FLOPS}{cycle}$ and 32 single precision $\frac{FLOPS}{cycle}$

The table below shows the peak performances:

Settings	OpenBlas	Size OpenBlas	Blis	Size Blis	MKL	Size MKL
close, no numactl	2493.85	19500	2579.73	5500	2097.68	5500
close, –interleave=0,1,2,3	3838.42	18500	3930.98	19500	2770.25	20000
spread, no numactl	2553.95	18000	2959.80	7500	2199.13	7000
spread, –interleave=0,1,2,3,4,5,6,7	4444.82	19500	4314.61	19500	3042.04	7000

The graphs below represent the scaling of math library when the policy changes (one graph for each policy). In all the three math libraries the usage of numactl options improve the overall performance. This is reasonable because of the lower average time that an OpenMP thread needs to access the RAM (better memory allocation). We can also see, on blis graph, that the curve of the policy spread with the usage of numactl, has a local maximum for $n = m = k = 9000$. A possible explanation for this behaviour cuold be linked to a optimal cache usage at those sizes.

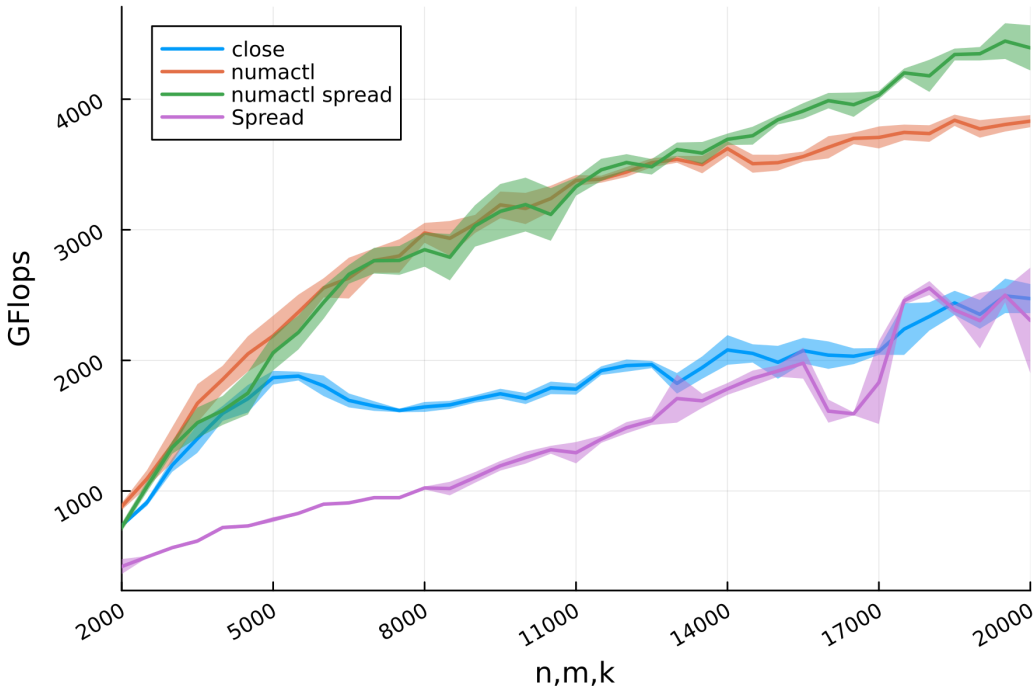


Figure 20: Comparison OpenBlas - Float

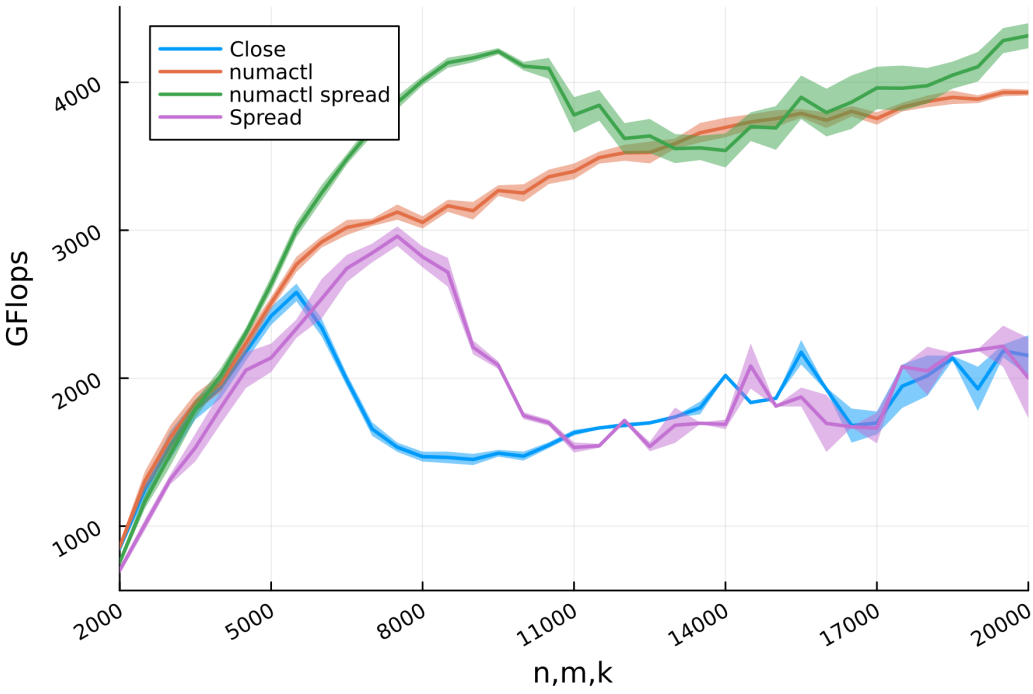


Figure 21: Comparison Blis - Float

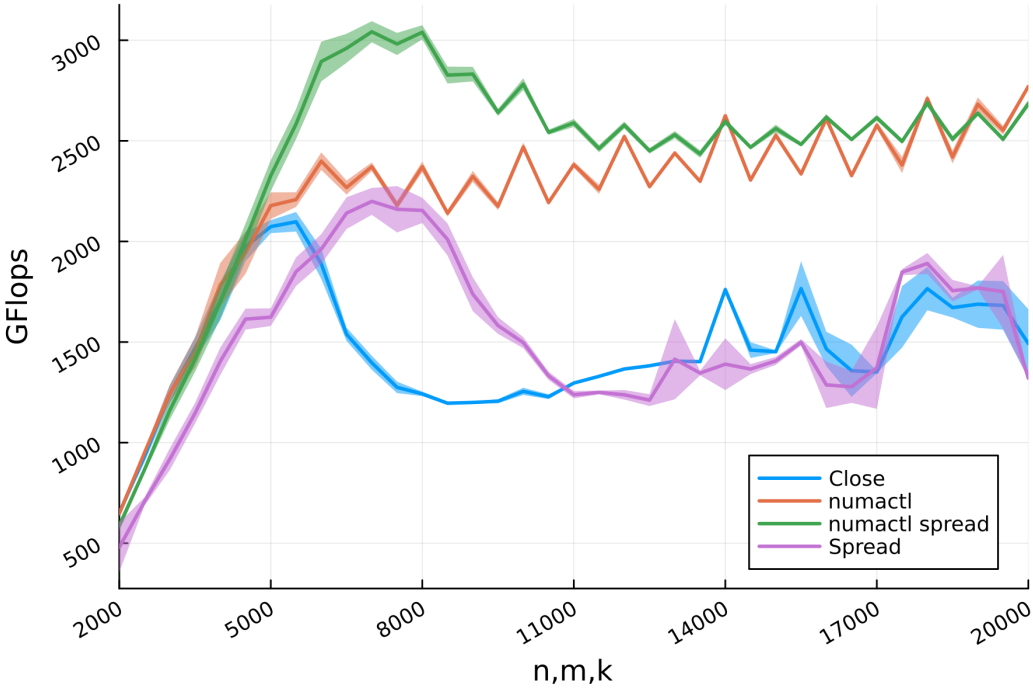


Figure 22: Comparison MKL - Float

In the graphs below I have gathered togheter the different math libraries (one plot for each policy).

For the close policy with no numactl options we can see that at the beggining Blis performs better then the other until $n = m = k = 5500$. Starting from $n = m = k = 6000$ the GFlops of all the libraries fall down until $n = m = k = 10000$. Note that from $n = m = k = 7000$ openBlas performs better than Blis.

With the policy with `OMP_PROC_BIND=close` and `numactl --interleave=0,1,2,3` we have that Blis performs slightly better than OpenBlas.

Finally with the policy `OMP_PROC_BIND=spread` and `numactl --interleave=0,1,2,3,4,5,6,7` Blis performs better than the others from $n = m = k = 2000$ until $n = m = k = 1200$. After that value Blis and OpenBlas are almost equivalent

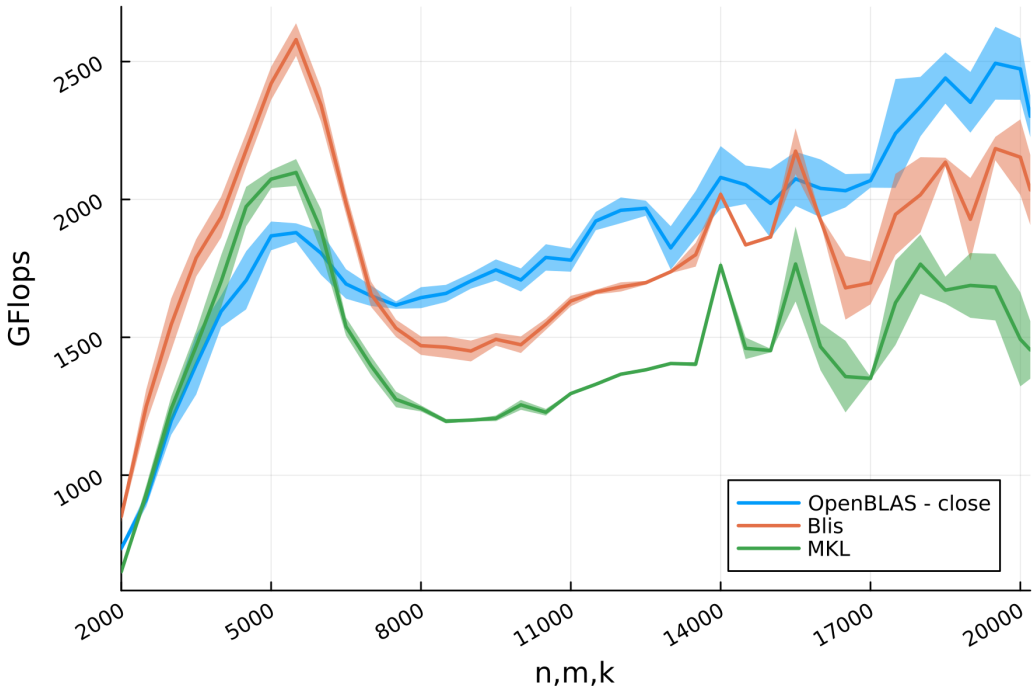


Figure 23: Comparison close, no numactl options - Float

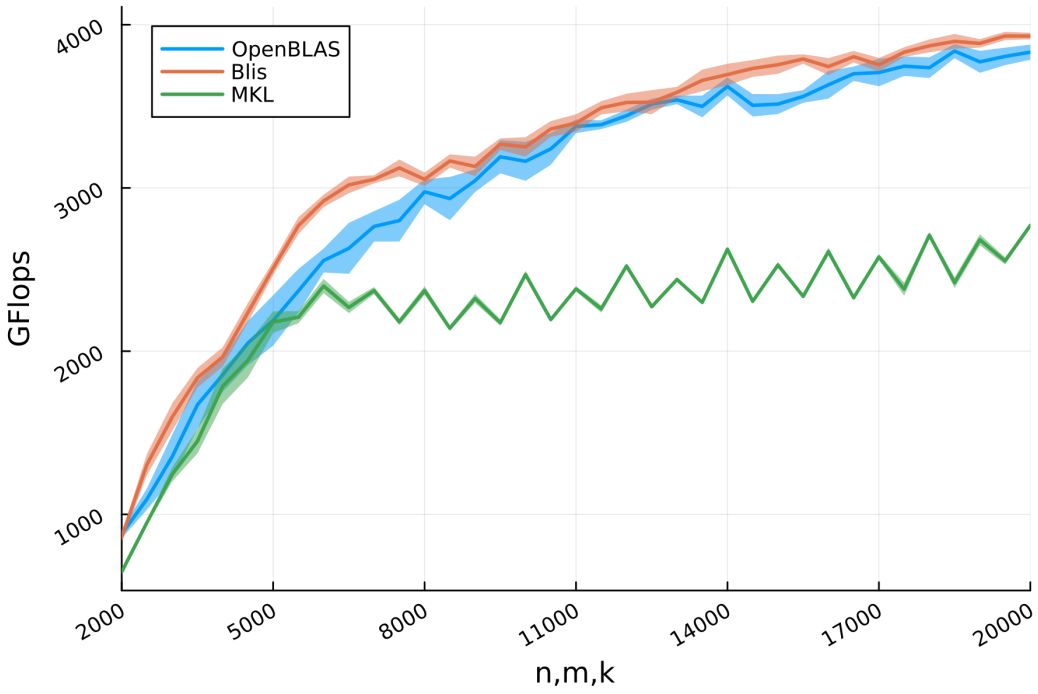


Figure 24: Comparison close, with numactl - Float

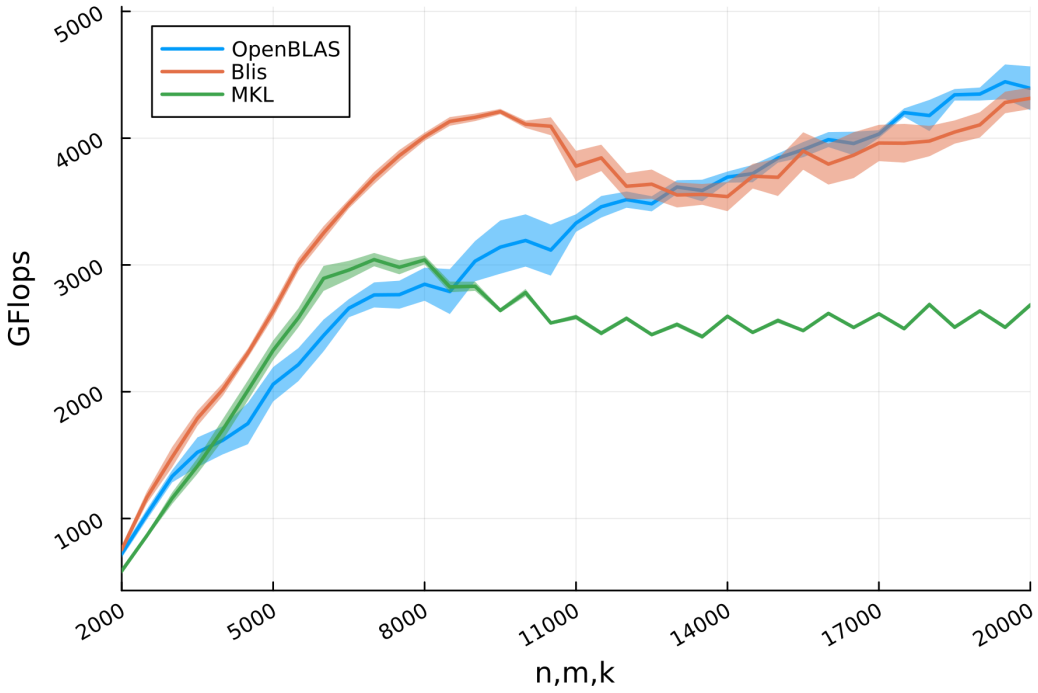


Figure 25: Comparison spread, with numactl - Float

Double precision

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

$$P = n.cores \cdot frequency \cdot \frac{FLOPC}{cycle} = 64 \cdot 2.6GHz \cdot 16Flops = 2662.4GFlops$$

The table below shows the peak performances:

Settings	OpenBlas	Size OpenBlas	Blis	Size Blis	MKL	Size MKL
close, no numactl	1134.09	16000	1168.78	16500	881.48	16000
close, –interleave=0,1,2,3	1842.97	18500	1909.73	16500	1683.39	20000
spread, no numactl	1225.07	17000	1222.11	17000	974.12	16500
spread, –interleave=0,1,2,3,4,5,6,7	2244.77	20000	2583.36	20000	1618.16	20000

The graphs below represent the scaling of math library when the policy changes (one graph for each policy).

From the OpenBLAS plot we can see that the usage of numactl improves the performance. If we look at `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread` both with the usage of numactl we can understand that until $n = m = k = 15000$ they are almost equivalent. After that `OMP_PROC_BIND=spread` overcame `OMP_PROC_BIND=close` if we consider the curves with numactl.

The Blis graph shows that `OMP_PROC_BIND=spread` with the usage of numactl is the best policy almost everywhere. It is important to highlight that there is a large drop in performance at $n = m = k = 11000$ for all the policies.

As it concern MKL we can se that the policy `OMP_PROC_BIND=spread` with the usage of numactl performs better in the initial part and, after a peak at $n = m = k = 10000$, it has a drop in performances. On the tail the policy `OMP_PROC_BIND=close` with numactl works better then the others.

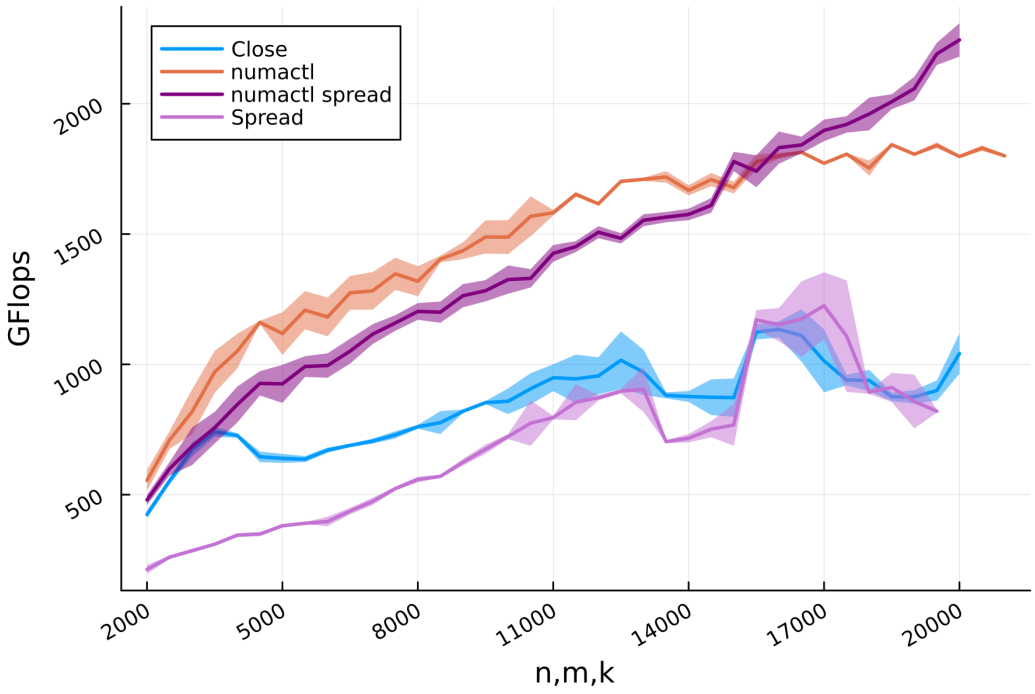


Figure 26: Comparison OpenBlas - Double

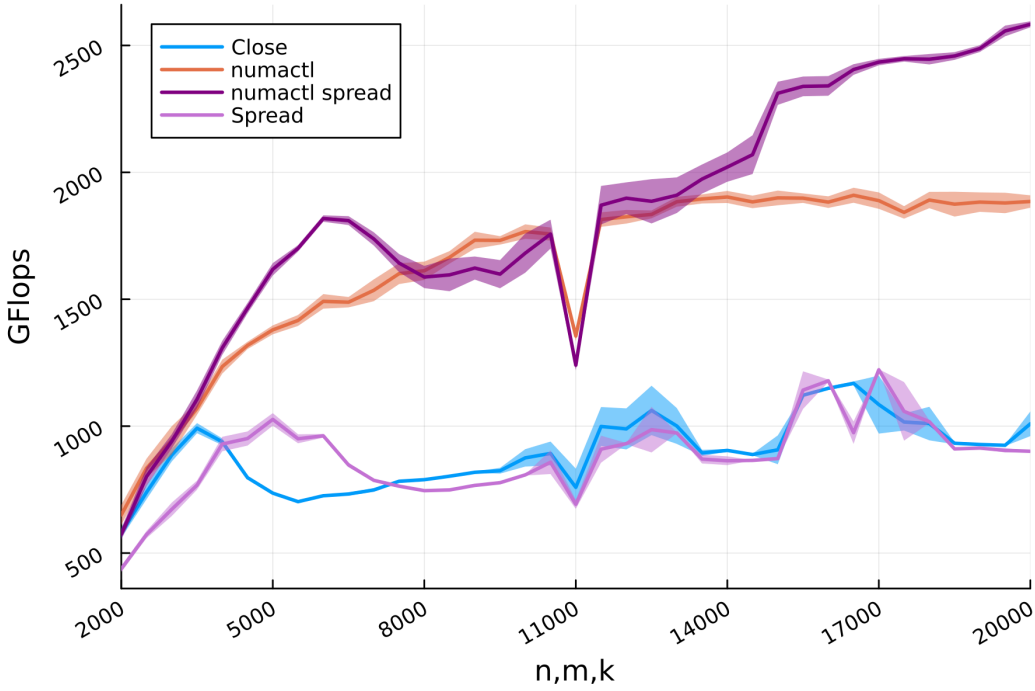


Figure 27: Comparison Blis - Double

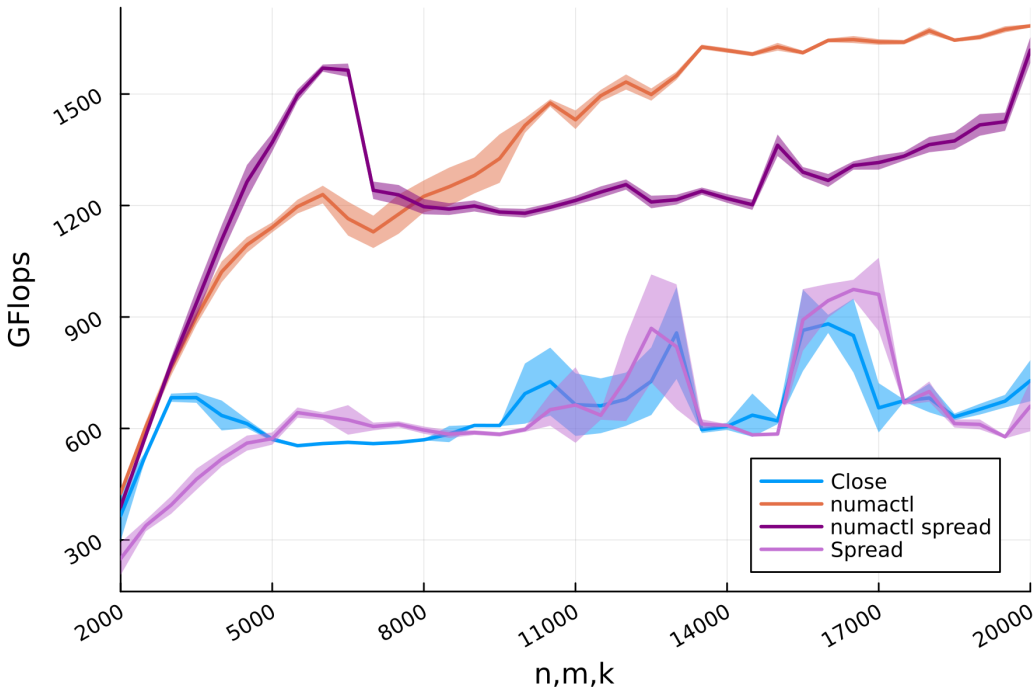


Figure 28: Comparison MKL - Double

In the graphs below I have gathered togheter the different math libraries (one plot for each policy).

If we look the graph of the policy `OMP_PROC_BIND=close` with the usage of numactl we can see that the best math library is Blis followed by OpenBlas. The worst one is MKL. All the three lines follows almost the same trend

In the end, the graph of the policy `OMP_PROC_BIND=spread` with the usage of numactl show us that the best library is Blis followed by openBlas and MKL. In this case the peak performance of the best library (Blis) is 2244.77, much larger than the peak performance in the other cases.

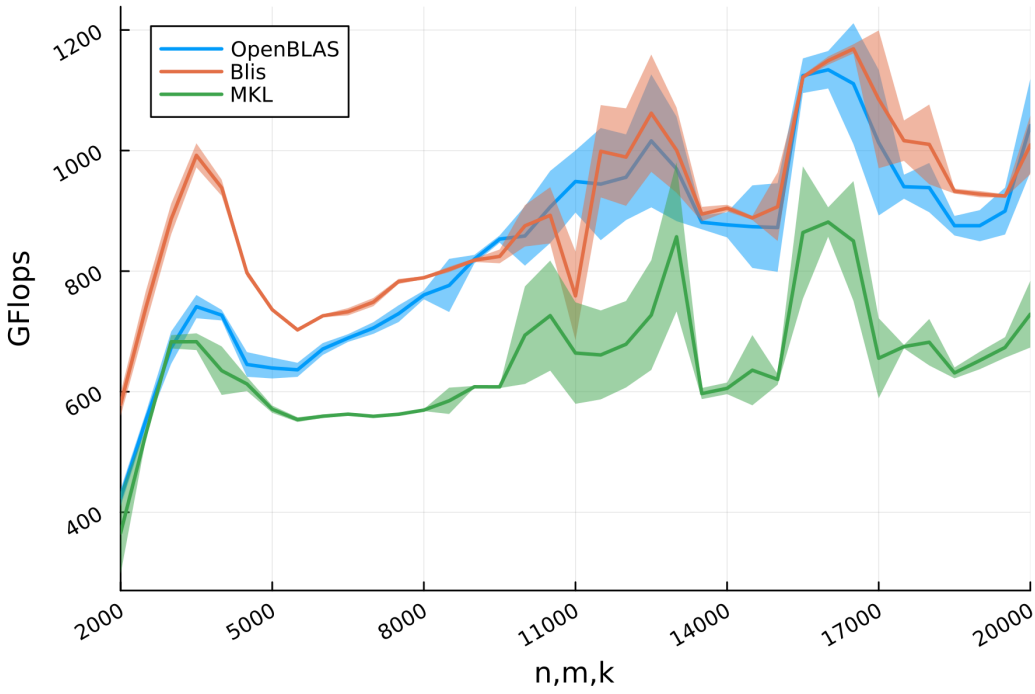


Figure 29: Comparison close, no numactl options - Double

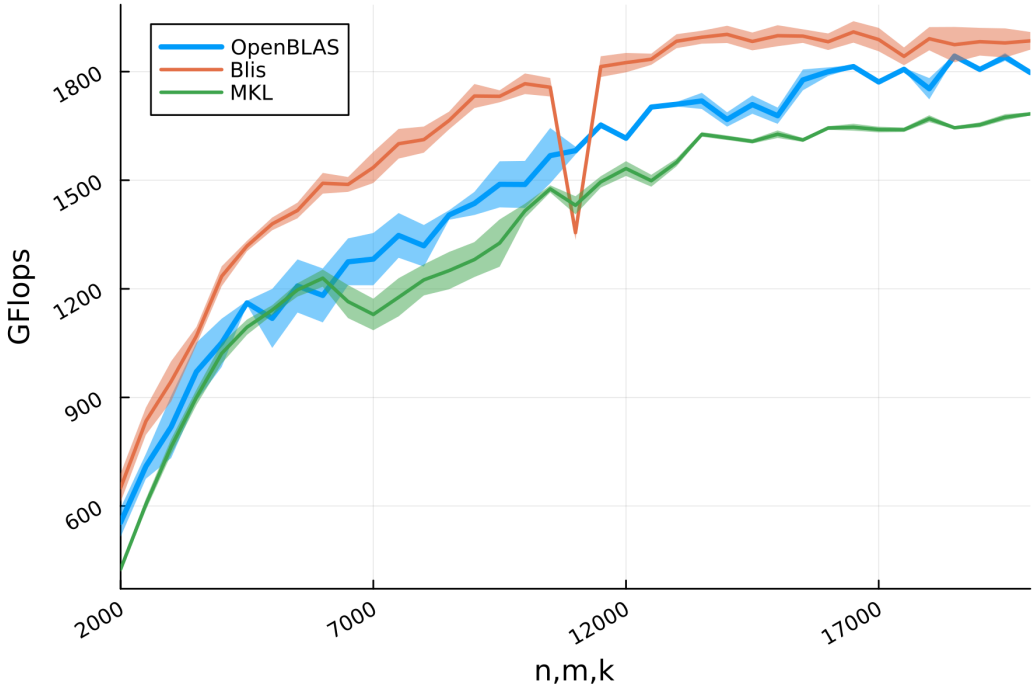


Figure 30: Comparison close, with numactl - Double

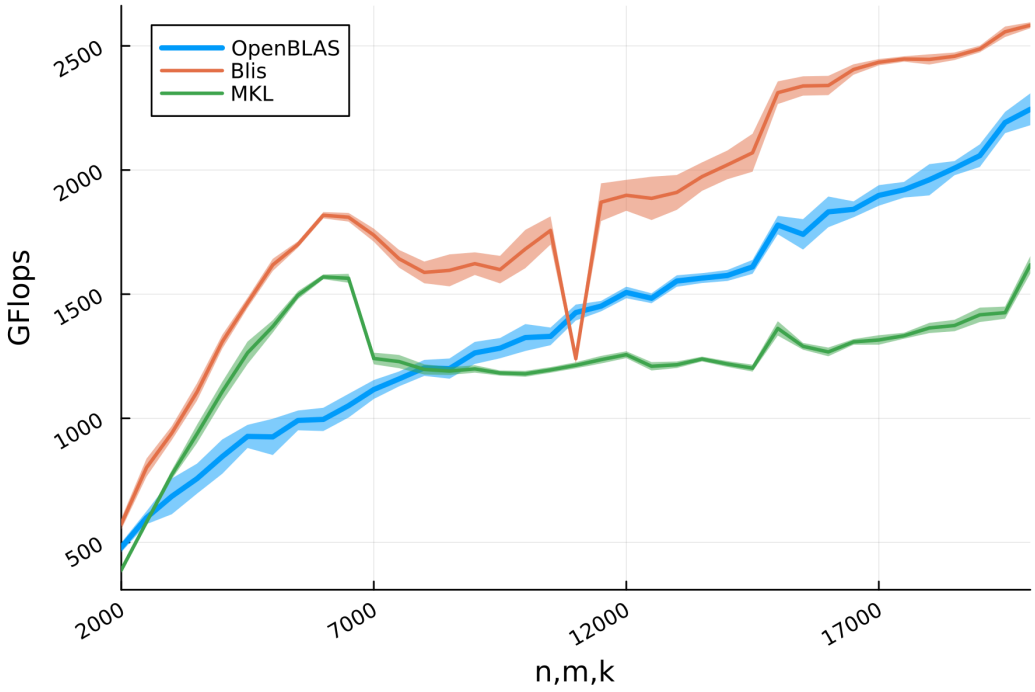


Figure 31: Comparison spread, with numactl - Double

Core scaling

In this section I analyze the behaviour of the different math libraries on a fixed value of n, m, k when the number of OpenMP threads increases. I have run the program for two different values of the parameter, 10000 and 20000. The policy that I have tested are:

- `OMP_PROC_BIND=close` whitout numactl options
- `OMP_PROC_BIND=close` with the following numaclt policy:
  - From 1 to 16 OpenMP threads: `--interleave=0`
  - From 17 to 32 OpenMP threads: `--interleave=0,1`
  - From 13 to 48 OpenMP threads: `--interleave=0,1,2`
  - From 49 to 64 OpenMP threads: `--interleave=0,1,2,3`
- `OMP_PROC_BIND=spread` without numaclt policy:

Single precision

The plots below show the OpenMP scalability. In all the three cases we have the maximum scalability when we use the policy `OMP_PROC_BIND=close` and numactl. The graphs of Blis

and MKL when $n = m = k = 10000$ shows that the speed up with `OMP_PROC_BIND=spread` has a strange behavious on the right tail.

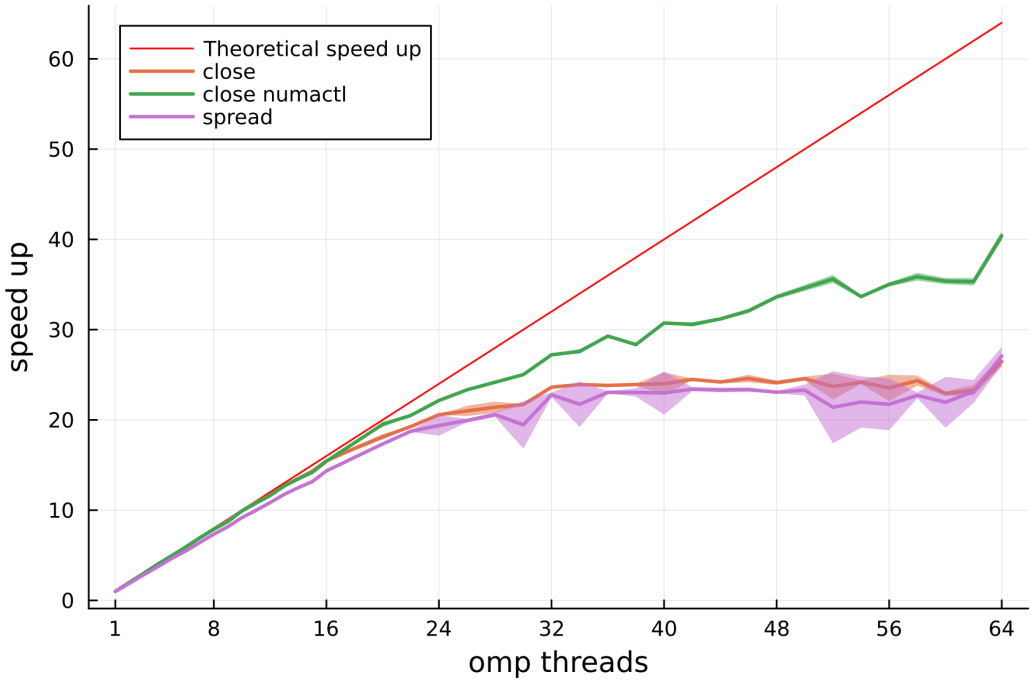


Figure 32: OpenBlas scalability, n=m=k=20000 - Float

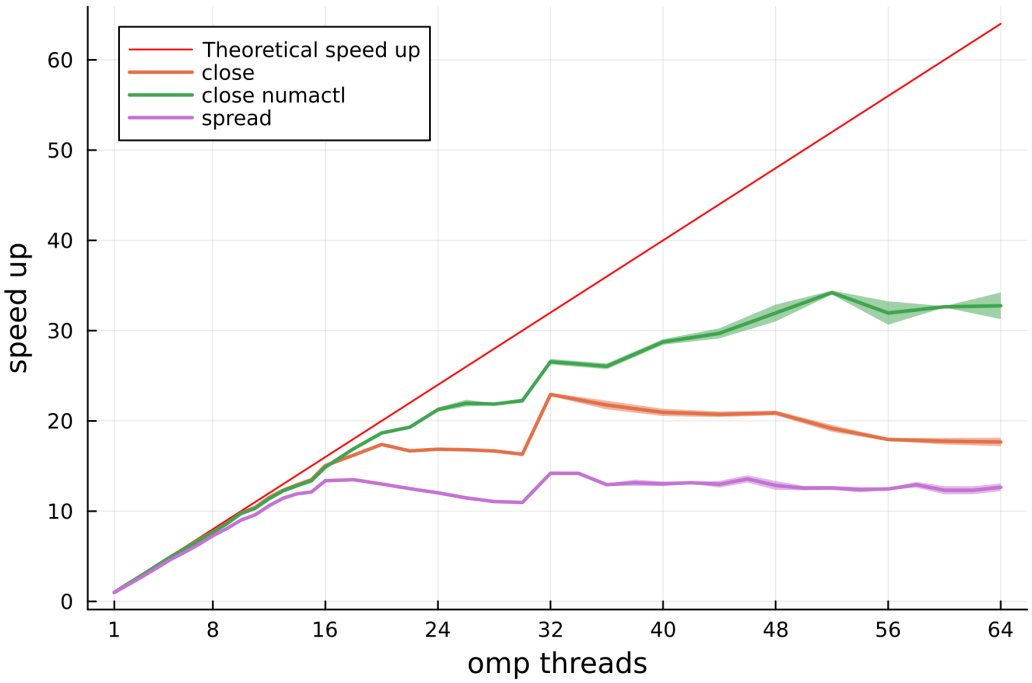


Figure 33: OpenBlas scalability, n=m=k=10000 - Float

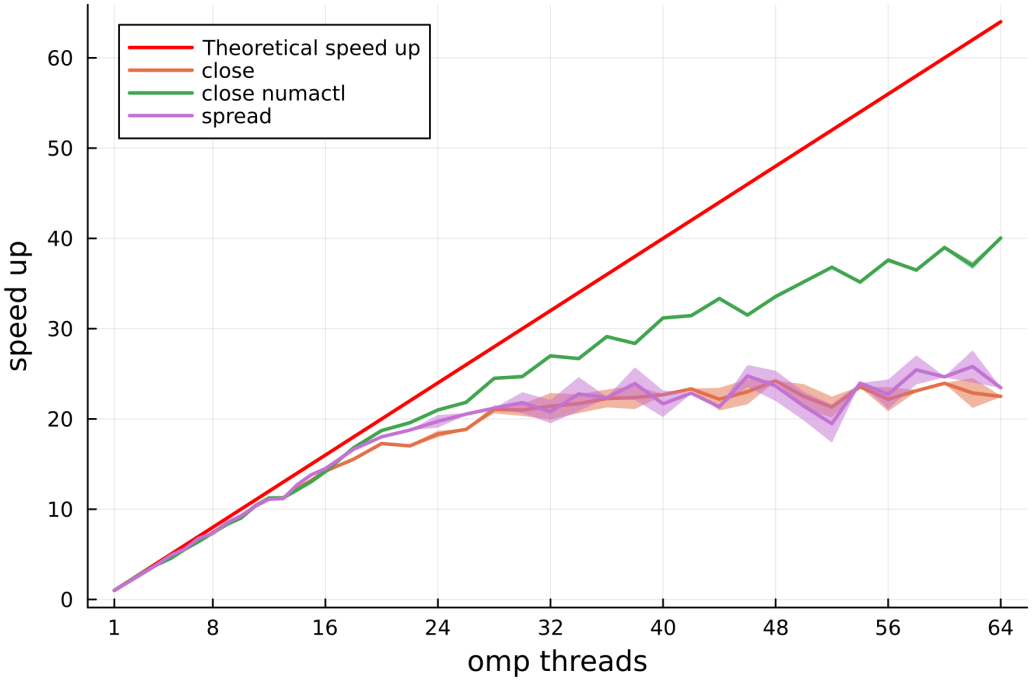


Figure 34: Blis scalability, n=m=k=20000 - Float

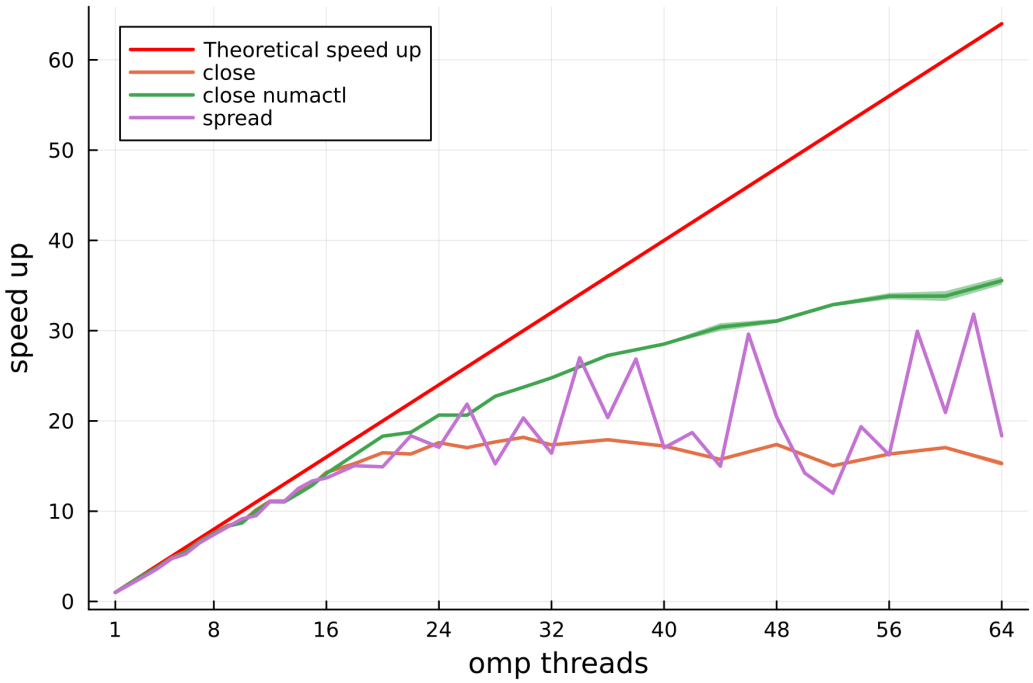


Figure 35: Blis scalability, n=m=k=10000 - Float

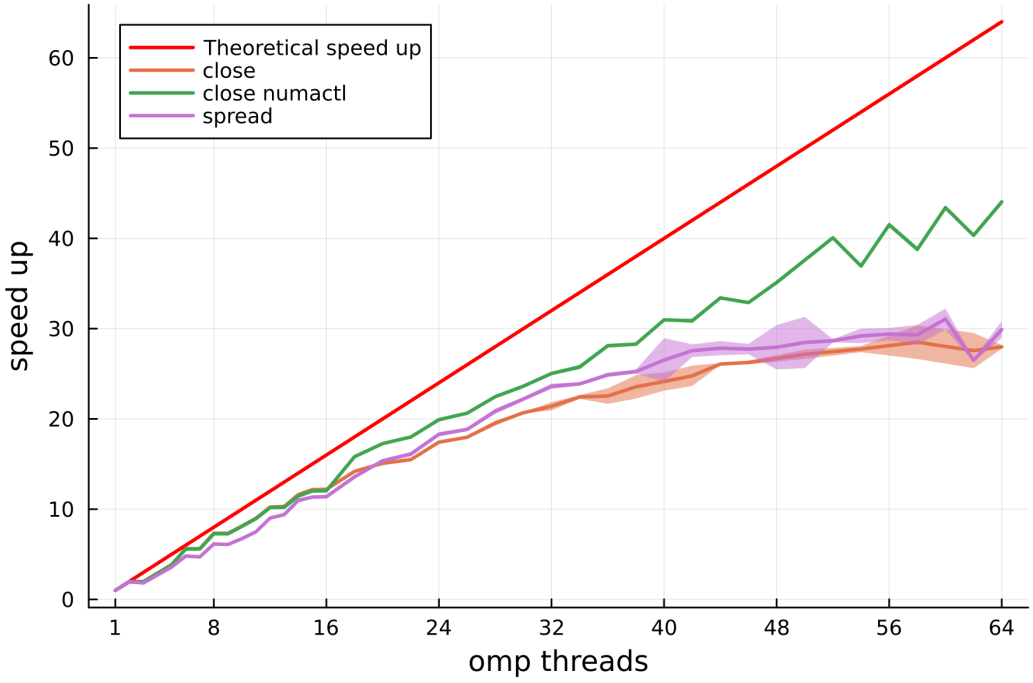


Figure 36: MKL scalability, n=m=k=20000 - Float

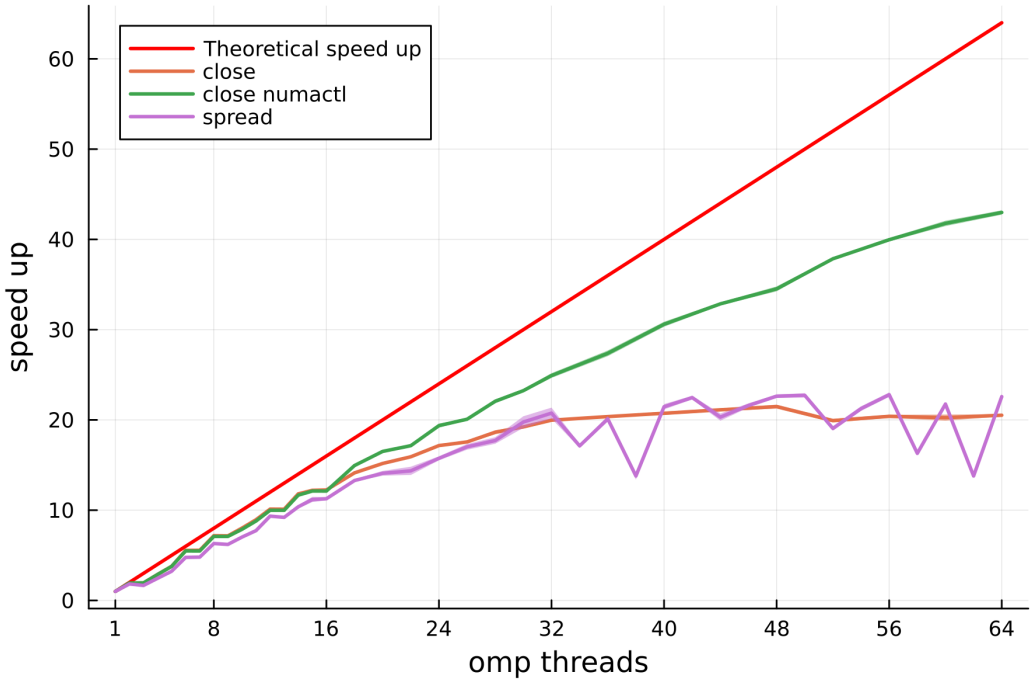


Figure 37: MKL scalability, n=m=k=10000 - Float

Double precision

The plots below show the OpenMP scalability. In all the three cases we have the maximum scalability when we use the policy `OMP_PROC_BIND=close` and numactl. We can notice, also in this case, a strage behaviour of the speed up for Blis with `OMP_PROC_BIND=spread` and $n = m = k = 10000$.

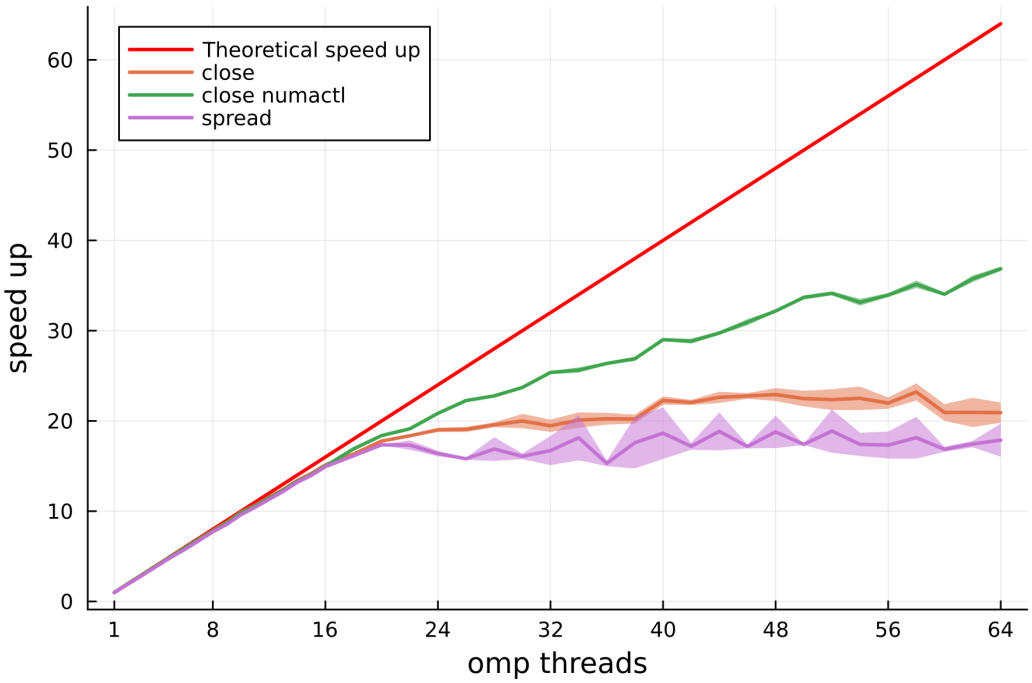


Figure 38: OpenBlas scalability, n=m=k=20000 - Double

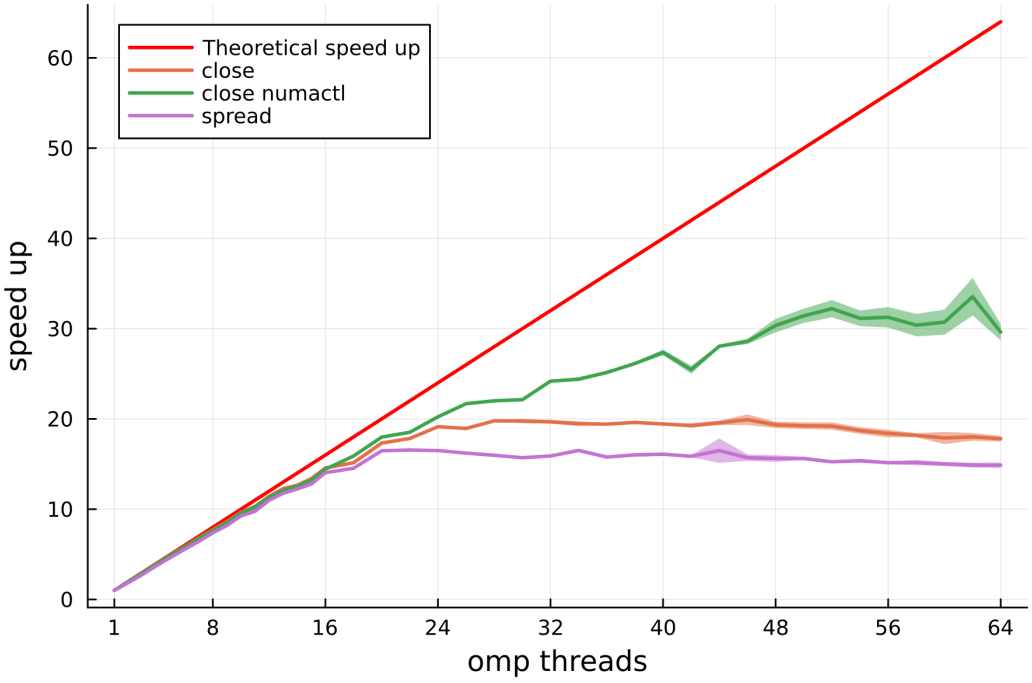


Figure 39: OpenBlas scalability, n=m=k=10000 - Double

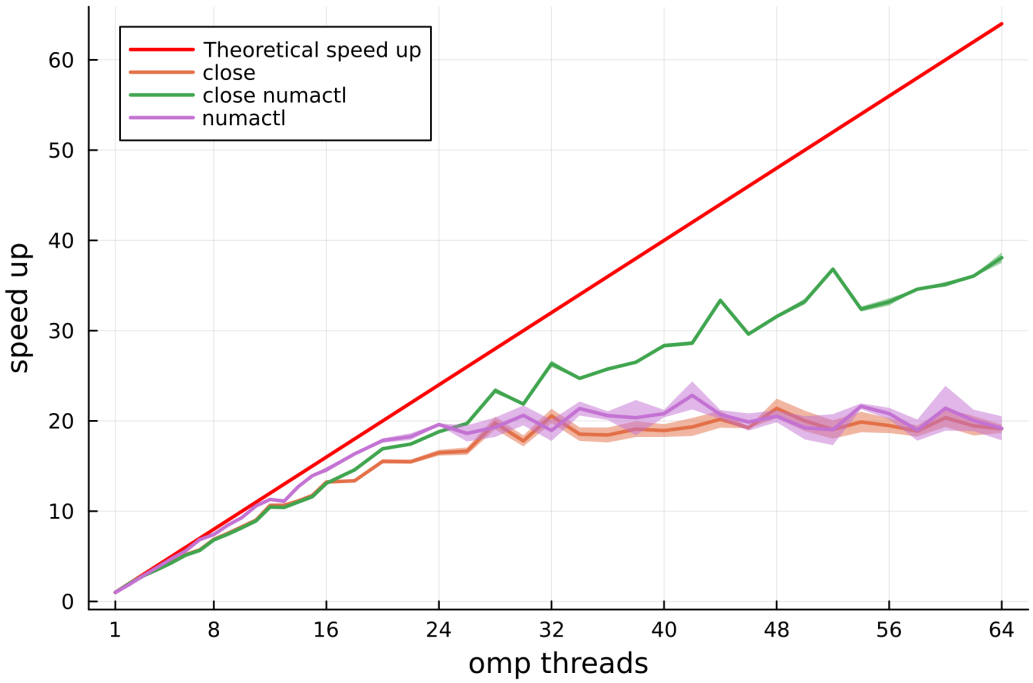


Figure 40: Blis scalability, n=m=k=20000 - Double

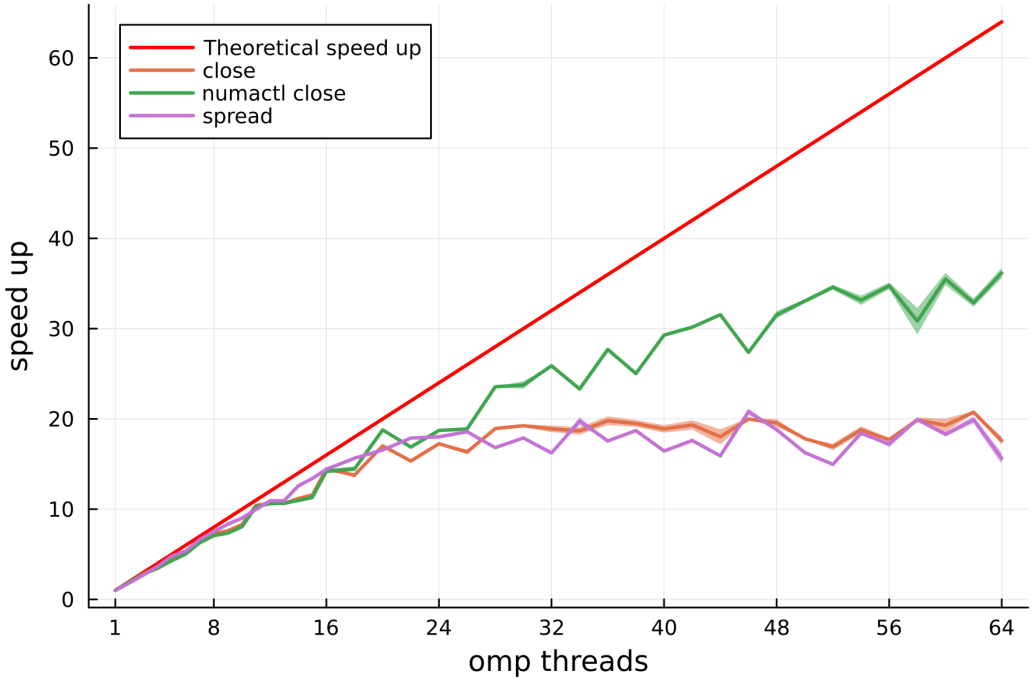


Figure 41: Blis scalability, n=m=k=10000 - Double

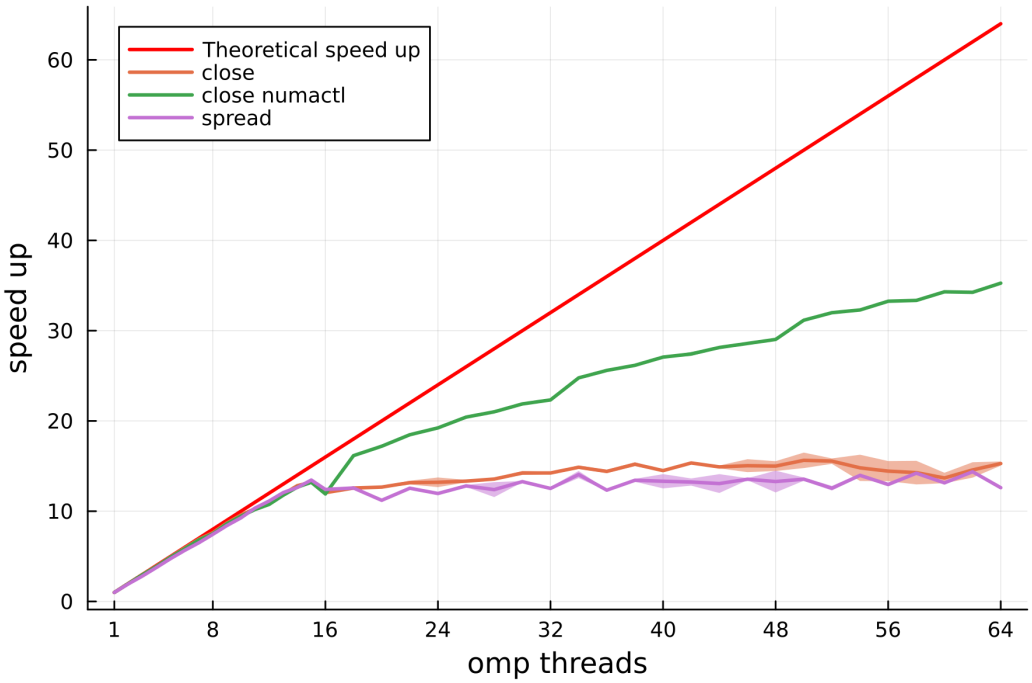


Figure 42: MKL scalability, n=m=k=20000 - Double

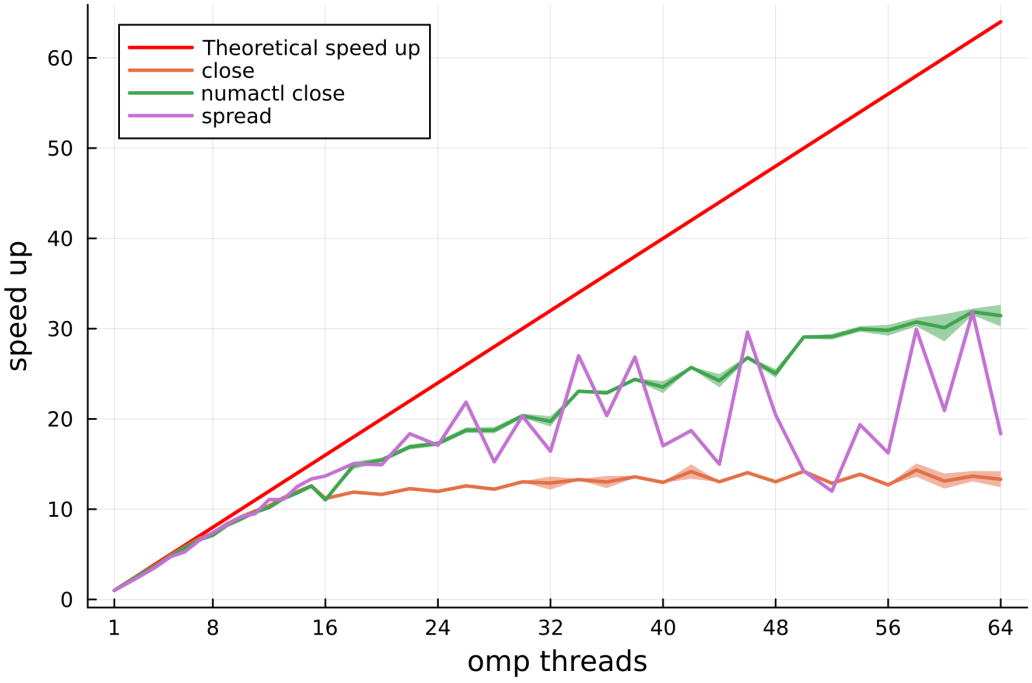


Figure 43: MKL scalability, n=m=k=10000 - Double