Foundations of High Performance Computing

Final Project

MSc Data Science and Scientific Computing

University of Trieste

Samuele D'Avenia, Elena Rivaroli

Academic Year 2022-23

September 27, 2023

Contents

1 Exercise 1

1.1 Introduction

The Game of Life[4] is a cellular automaton and mathematical game developed by British mathematician John Conway. The game is played on a two-dimensional grid, where each cell is either alive or dead and has a total of 8 neighbours (all pictures will be images where black corresponds to alive and white to dead). At each iteration the system evolves and each cell may update its state based on the number of alive neighbours it has.

The implementation presented here uses a periodic boundary, meaning that for example cells in the same row at the first and last column are neighbours. Figure 1 helps in picturing this by showing the 4×4 matrix and what sections are neighbours.

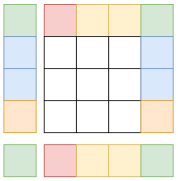

Figure 1: Periodic Boundary

Three different evolution methods are implemented and briefly described below:

- **Static evolution**: at each time step the system is frozen. Then each cell checks the state of the adjacent neighbours based on the frozen system and evolves according to it.
- **Ordered evolution**: at each time step we start from the first entry in the grid all the way to the last. In an ordered way, each cell checks the state of the adjacent neighbours and evolves immediately.
- **White-Black evolution**: consider the playground as it was a chessboard and at each time step first evolve the white positions and then the black ones.

The aim of this exercise is to implement a parallel version of the game using a hybrid approach of OpenMP and MPI, and analyze the scalability of each of the different evolution methods. Three different scalability studies are considered:

- **OpenMP scalability**: see execution time for increasing number of threads (and cores) per task.
- **Strong MPI scalability**: see execution time for increasing number of MPI tasks.
- **Weak MPI scalability**: while keeping the workload per process fixed, see the execution time as the number of processes increases.

The code is implemented in order to meet the requirements provided in the assignment.

1.2 Methodology

1.2.1 Domain decomposition

In order to decompose the workload among the different MPI processes, we decided to use domain-decomposition.
The matrix is traversed by row to compute all the updates: as such we opted for a decomposition by row.
The number of rows each process has to work on is the same, with some adjustment if the number of rows is not a multiple of the number of processes.
In order to update its rows each process will also need to access the row above its first row and the row below its last one, as they are neighbours. As such these also need to be stored in the memory of each process and updated according to the evolution method. From now on we will be referring to the rows which a matrix updates as *working rows*, while the two additional ones will be called *frame rows*. Figure 2 shows how a 5×5 matrix would be subdivided between 3 processes on the left, while on the right we have what section of the matrix is contained in process 0, with the mentioning of the frame and working rows.

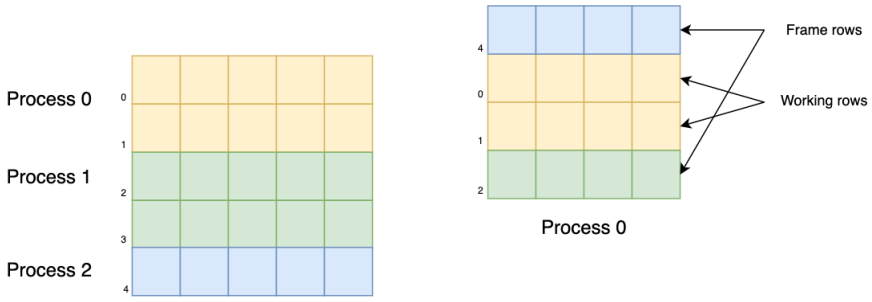

Figure 2: MPI: Domain decomposition

Keep in mind that each process does not update directly its frame rows. For this reason the processes working on the neighbouring sections of the matrix will have to send messages to each other.
Additionally, when required, OpenMP are to further decompose the workload on the working rows of each process.

1.2.2 Matrix initialization, reading and writing

We start by presenting how the matrix used for the grid is generated, written and read from file, as all the steps are implemented with some level of parallelization. To be more specific all three steps use MPI, but only in the random generation of the initial matrix, OpenMP threads are employed.
It is important to mention that all the images are stored using the pgm format, meaning that

the file consists of a header with some information, including file dimension, followed by the matrix entries.
As mentioned above, each MPI process contains a section of the matrix to work with. We decided to generate the initial images where each cell is initialized at random to be alive or dead.
As such, when the matrix is created, we want each process to generate only the rows it will have to operate on. We decided that inside each MPI process we would employ OpenMP threads. Each thread generates an equal number of rows. We opted for a static allocation as it has a smaller overhead and dynamic allocation does not help since each chunk has the same workload. Keeping the seed variable as private to each thread is necessary to ensure scalability but also to avoid every thread generating the same numbers.
Now we describe how the process of writing to file and reading from file is executed. There are many ways to perform this.

- One possible way is to have each process writing on a different file individually and then reading from it. This would however require us to join the different files in a second moment. Moreover, for a large number of processes it would create many files. Another drawback of this approach is that it requires further care if we try to read the files using a number of processes different than the one used for writing.

- Another possibility is to only write to file from a single MPI process and do the same for reading. This would however require us to have one process gathering the rows before writing to file and similarly having to scatter the rows to each process when reading.

- The last option is to use the MPI I/O functions, which allow us to have every process writing to file its section of the matrix, with each one using a different file pointer. Reading is also performed in the same way, with each process reading the specific section.

We decided to use the last option as it seemed the most natural one with the least amount of drawbacks.
As a final note, we decided to have each process read also its frame rows from file, to avoid having to send additional messages. We settled on this approach because reading these two additional rows does not require additional seek operations on file. The only two processes which have to perform an additional seek are the first and last process.

1.2.3 Static evolution

We start by describing the static evolution method. At each time step the state of the system is frozen before evolving. Then, each cell in the system is updated based on the state of the "frozen" neighbours. To perform this evolution an auxiliary matrix is used.
The workload is distributed among the different MPI processes using domain decomposition. Moreover, each MPI process updates the cells in its working rows in parallel using OpenMP

threads over the rows of the matrix.
After this, each process has the correctly updated entries in its working rows. However, the frame rows are not yet updated. For this reason each process needs to send and receive two messages from and to its neighbours with the corresponding rows. We settled on using non-blocking operations for both send and receive operations. This requires some additional checks to ensure these operations are successful. Further description is provided later in Section 1.3.3.
Figure 3 shows an example of static evolution.

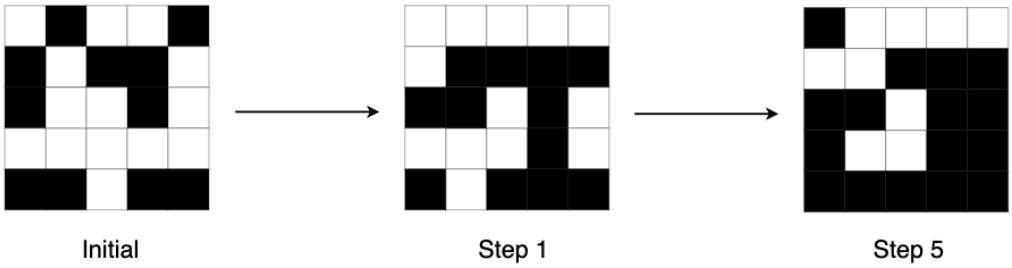

Figure 3: Example of static evolution with initial matrix and steps 1 and 5

1.2.4 Ordered evolution

We now consider the ordered evolution method. This requires us to update all the cells in row-major order in each time step, starting from the first one to the last one. As such, unlike the static case, the update step for a cell requires knowledge of the new state of the previous ones. For this reason, as we will see also in the results, this is a serial process. However, we still implemented a version of this evolution that uses a combination of MPI and OpenMP. This was done to assess whether a better memory usage is achieved.
Just like the static evolution, each MPI process has its working rows and frame rows. Just like before, each process will update its working rows using OpenMP threads. However, to ensure that the matrix is evolved correctly there are some synchronization requirements which were not present in the static case.

- During each time step an MPI process can only update its working rows after receiving its upper frame row from the previous process.
- A thread can only start updating the rows it is assigned only after all previous threads have finished theirs.

To enforce these, we require multiple adjustments. Further discussion of how this is done can be found in subsection 1.3.
Figure 4 shows an example of ordered evolution.

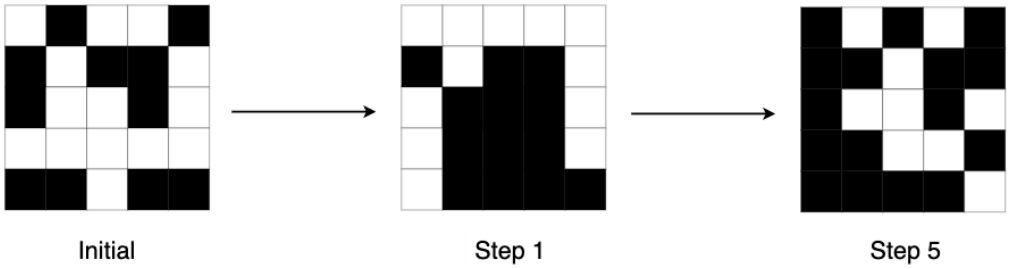

Figure 4: Example of ordered evolution with initial matrix and steps 1 and 5

1.2.5 White-Black evolution

As a final way to evolve we consider the White-Black method. This requires us to imagine the playground as if it was a chessboard with black and white tiles (not to be confused with the dead or alive state of a cell).
Then, at each time step we first update all the white tiles, assuming that the first tile of the chessboard is white. Secondly, we update all the black ones, always in a row major order.
This method is an example of ordered evolution, which as already mentioned is intrinsically serial. As such, we decided to not implement the parallel version with MPI processes, as we expect that it would only lead to a behaviour similar to the ordered evolution.
However, a parallel version using OpenMP threads is described in this report. If we focus on white tiles in one row, we can notice that their update is independent of one another as they are not neighbours. Hence, we use threads to update the different cells of the same colour in a row. The only thing to note is that if we are working with a matrix of odd size, additional care is required to subdivide the thread work. This is because the last tile in the row and the first one have the same colour and are neighbours. This requires us to enforce that the last cell is updated only after the update of the first one in the same row. An example of white-black evolution is shown in Figure 5.

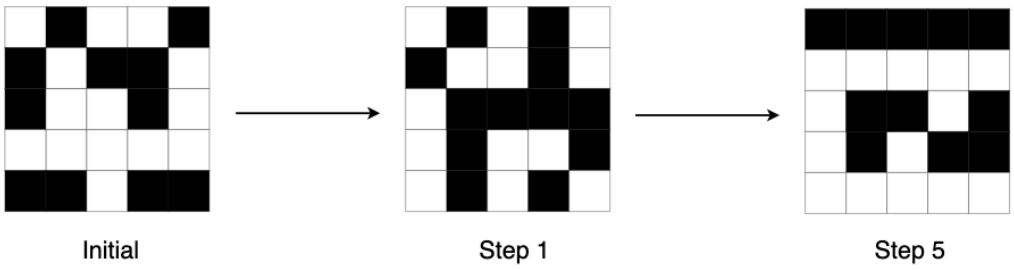

Figure 5: Example of white-black evolution with initial matrix and steps 1 and 5

1.3 Implementation

This section presents more technical details with the same structure as section 1.2.
The implementation discussed assumes that the starting point is a square matrix of dimension $k \times k$.
Also note that we opted to work with matrices represented as a one-dimensional array of `unsigned char` type in the C implementation. This way each entry in the matrix only costs one byte and they are all contiguous in memory. Each process allocates dynamically the amount of memory required to store its working and frame rows.
As a final consideration about the code, for every evolution method which uses MPI and OpenMP, there are two functions, one which uses both MPI and OpenMP and another one which only uses OpenMP. The latter is used whenever the executable is called with only 1 MPI process, to avoid issues with the messages and the additional costs that calling those has.
With all these in mind, we have an executable which, depending on the parameters passed, can do these:

- Generate a random matrix of a specified size and write it to a pgm file.
- Read a matrix from a file and evolve it statically for a certain number of steps while also printing it to file every s steps, where s can be specified by the user.
- Read a matrix from a file and evolve it using the ordered method for a certain number of steps while also printing it to file every s steps, where s can be specified by the user.

1.3.1 Domain Decomposition

Let us start by discussing how the matrix is distributed among the different MPI processes more in detail.
If we have a square matrix of size k and we have n MPI processes, each process gets $n_{process} = \frac{k}{n}$ rows to work on. Then the remainder of n/k rows are assigned one to each process starting to process 0. This means that each process will now have to work on a $n_{process} \times k$ or $(n_{process}+1) \times k$ matrix. To do this, we also have to add the two frame rows which are required for correctly updating the cells.
Inside each MPI process the working rows are divided among the OpenMP threads in the same way. This ensures that each thread works on contiguous rows. Since we are using domain decomposition each thread has to do the same amount of work so the default static schedule for OpenMP for parallelization is used as it is the most suitable.

1.3.2 Matrix generation, file reading and writing

We start by discussing how a random matrix is generated in our code. The code discussed here can be found in the `read_write_parallel.c` file of the Github repository. Each process

allocates the amount of memory required to store the working rows only. This allocation is done using the `malloc()` C function. This is done to avoid the overhead needed when allocating with `calloc()` but also to avoid having the whole matrix loaded in the master thread memory. The code below shows how a single process which has to initialize `rows_init` rows allocates its memory.

```
unsigned char* ptr = (unsigned char*)malloc(rows_init*k*sizeof(unsigned
    char));
```

After this, an OpenMP parallel region is opened by each process, and each thread is assigned a subset of rows to generate. The `seed` is set privately using the C `clock()` function. To avoid having the same random matrix generated by each process and thread, we specialize it using its thread id and rank number as can be seen in the code below:

```
int my_id = omp_get_thread_num();
unsigned int seed = clock();
seed = seed * rank + my_id;

#pragma omp for
for (int i = 0; i < rows_initialize*k; i++){
    unsigned char random_num = (unsigned char) rand_r(&seed) % 2;
    ptr[i] = (random_num==1) ? 255 : 0;
}
```

Now a description of how this matrix is written to file is provided. The aim is to have each process writing only the rows it has generated. First of all, the master process creates the file and writes the pgm header using standard C I/O functions.
After it has finished writing the header, the file is opened again using MPI I/O functions and each process is assigned an individual file pointer to write to a specific section of the file. The location where each process should access the file can easily be determined since we know how many rows each process has to write. This operation is done using the collective `MPI_File_write_all()`, as it works better than the non-collective version when multiple processes are used to write concurrently [2]. This is shown in the code below:

```
MPI_File_open(MPI_COMM_WORLD, fname,
            MPI_MODE_APPEND | MPI_MODE_RDWR,
            MPI_INFO_NULL, &fh );

// Decide where on file each MPI process should write.
disp = (rank >= k % size) ?
        (rank * rows_init +k%size)* k *sizeof(unsigned char) :
        rank * rows_init * k *sizeof(unsigned char);

MPI_File_seek(fh, disp, MPI_SEEK_CUR);
MPI_File_write_all(fh, ptr, rows_init*k, MPI_UNSIGNED_CHAR,
    MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

Finally let us focus on reading the matrix from file. As stated in 1.2 we want each process to read its working rows with the additional two rows of the frame.
First of all process 0 reads the header of the pgm image and stores the position in the file where this header ends. This value is broadcast to all the other processes, so they know where the matrix starts in the file.
After this each process places its file pointer in the position where its upper frame row starts. Then it proceeds to reading all the way until it has read its lower frame row. Note that this means that all processes only have to do one pointer movement, while processes 0 and the last one have to do an additional one since they have to read both the first and last row of the file at some point. For the same reason as the write version, the collective function `MPI_File_read_all()` is used. Figure 6 shows the different file pointer movements for two processes.

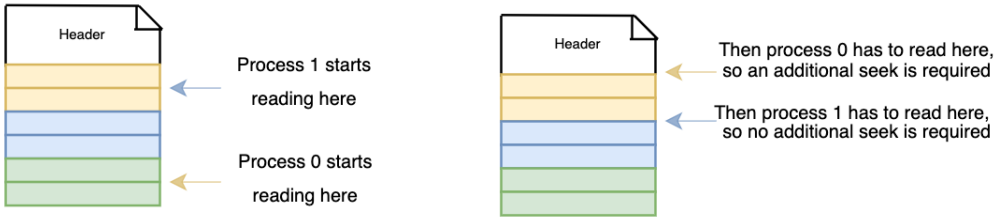

Figure 6: Parallel File Read with MPI I/O

1.3.3 Static evolution

As previously mentioned our code executes a certain number of steps of static evolution on our matrix.
The implementation we present requires the usage of an auxiliary array. This is needed because one array stores the frozen system, while the other one is used to store the cells as they are updated.
After the auxiliary array is filled with the updated cells, one possibility would be to move at every iteration the content of the auxiliary array into the original one. However, this is not required, as one can simply swap the memory location pointed by each pointer at the end of every time step, to ensure that the original pointer always has access to the memory region with the evolved system. To make it a bit clearer a pseudocode is shown below:

```
ptr: points to memory location with the initialized array
aux: points to memory location with the non-initialized array

1. Iterate all ptr and store the new state of every cell in aux.
2. Now aux points to the memory region with the correct array.
3. Swap the memory addresses stored in ptr and aux.
4. Now ptr points to the memory region with the correct array.
```

The idea for parallelization using OpenMP threads inside each process was already explained above. With this evolution method each thread can update its assigned rows in parallel with all

the others, so there are no additional synchronization requirements.
When all the threads have updated their corresponding rows, each process contains its correctly updated working rows. However, it still needs the updated frame rows before it can move to the next iteration.
This means that before each process can move to the next iteration, it needs to send its first and last (working) row to its neighbouring processes and also receive its frame rows from those. To do this we implemented two non-blocking send and receive for each process. For all processes other than the first and last they are implemented like this:

```
// Send message to process before
MPI_Isend(next+k, k, MPI_UNSIGNED_CHAR, rank-1, rank + n_step,
    MPI_COMM_WORLD, &request[0]);
// Send message to process after
MPI_Isend(next + rows_read*k, k, MPI_UNSIGNED_CHAR, rank+1, rank + n_step,
     MPI_COMM_WORLD, &request[1]);

// Upper frame row receive
MPI_Irecv(next, k, MPI_UNSIGNED_CHAR, rank-1, rank-1 + n_step,
    MPI_COMM_WORLD, &request[2]);
// Lower frame row receive
MPI_Irecv(next + k + rows_read*k, k, MPI_UNSIGNED_CHAR, rank+1, rank+1+
    n_step, MPI_COMM_WORLD, &request[3]);
```

Note that in this code `next` is the array containing the updated matrix and `k` is the number of columns of the original matrix.
As mentioned above with this approach we need to ensure that before a process moves to the next iteration it has correctly sent and received all necessary rows. This is done with the aid of a `MPI_Request` array and a call to `MPI_Wait_all()` at the end of each time step.

1.3.4 Ordered Evolution

As previously explained in 1.2.4, we need additional synchronization to ensure that the different processes and threads start working only after previous ones have finished their work. To enforce this inside each MPI process, the threads should iterate over the rows serially and in an ordered way. This is done with the aid of the `#pragma omp ordered` directive. Inside the parallel loop with the ordered clause we define an ordered region where the threads execute in ordered way one after the other. The code below shows this:

```
#pragma omp for ordered
  for(int i=1;i<k;i++){
    for(int j=0; j<k;j++){
      #pragma omp ordered
      {
       int n_neigh = current[(j-1+k)%k+i*k] + current[(j+1+k)%k+i*k] +
            current[(j-1+k)%k+(i-1)*k] + current[(j+1+k)%k+(i-1)*k] +
            current[(j-1+k)%k+(i+1)*k] + current[(j+1+k)%k+(i+1)*k] +
            current[(i-1)*k+j] + current[(i+1)*k+j];
```

```
        current[i*k+j] = (n_neigh > 765 || n_neigh < 510) ? 0 : 255;
      }
    }
  }
```

Now we also have to discuss how the different MPI processes exchange messages. We describe how this works for a process which is neither the first nor the last. These two only have some small adjustments in the order of the message operations in the code, but the idea is the same. At the beginning of every time step a process must wait until it has received its upper frame row from its previous neighbour. All these receive operations are implemented using a blocking receive, as this ensures that a process only starts working after the previous one has finished updating. The code below shows an example of such message:

```
if(rank != 0){ // Upper row receive
    MPI_Recv(current, k, MPI_UNSIGNED_CHAR, rank-1, rank-1 + n_step,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Whenever a process receives the required message, it operates on its working rows using OpenMP threads as described above with an ordered for loop. After this, each process has to do the two following operations:

- It has to send two messages. These are: a message to the previous process where it sends its first working row (which will be used at the next time step) and a message to the following process containing its final working row. These send messages are implemented using non-blocking send operations and there is not even the need for an additional call to `MPI_Wait_all()`, since the blocking receive ensure that the buffer is not modified. The code is as below:

  ```
  // Send message to process after
  MPI_Isend(current + rows_read*k, k, MPI_UNSIGNED_CHAR, rank+1, rank +
      n_step, MPI_COMM_WORLD, &request[1]);
  // Send message to process before
  MPI_Isend(current+k, k, MPI_UNSIGNED_CHAR, rank-1, rank + n_step,
      MPI_COMM_WORLD, &request[0]);

  ```

- It has to use a receive operation, to ensure that before it moves to the next time step, the process after has correctly communicated the row which will be used as lower frame row. For this reason this is implemented with a blocking receive as shown below:

  ```
  MPI_Recv(current + k + rows_read*k, k, MPI_UNSIGNED_CHAR, rank+1, rank
      +1+n_step, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  ```

1.3.5 White-Black evolution

As previously mentioned, with this method we need to consider the playground as if it was a chessboard, where we assume that the first tile is white. This method proceeds by first updating

all the white tiles in the playground and then all the black ones.
We decided to use OpenMP threads to update the different tiles of the same colour in the same row. As such, we need to work on one row at a time, and then the threads should iterate over the columns. To ensure that all tiles of the correct colour are selected, we simply have to update every two tiles.
We have two different situations depending on whether the number of columns of the matrix is odd or even. We decided to implement two different functions to separate these two cases.

- If the number of columns of the matrix is odd, we need to ensure that the last tile is updated only after updating the first tile of the same row, since they are of the same colour and they are neighbours. To ensure this, we iterate and parallelize with threads over all the columns except the last one. Afterwards, only one thread updates the last tile of that row. This is done using the `#pragma omp single` construct.
- If the number of columns is even, we just need to iterate over all the columns of the same colour of a row without any additional care. This is the case portrayed in Figure 7.

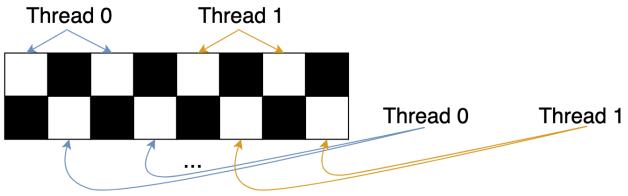

Figure 7: White-Black parallelization

The code below shows how we update the white cells of the matrix in case the number of columns is even:

```
for(int i=1;i<k;i++){
    int t = (i+1)%2;
    #pragma omp for
    for(int j = t;j<k;j+=2){
        int n_neigh = current[(j-1+k)%k + i*k] + current[(j+1+k)%k+i*k]+
            current[(j-1+k)%k+(i-1)*k] + current[(j+1+k)%k+(i-1)*k] +
            current[(j-1+k)%k+(i+1)*k] + current[(j+1+k)%k+(i+1)*k] +
            current[(i-1)*k+j] + current[(i+1)*k+j];
            current[i*k+j] = (n_neigh > 765 || n_neigh < 510) ? 0 : 255;
    }
}
```

Once all the white cells and the black ones are correctly updated, the code proceeds with the next evolution step.

As a consequence of how this evolution method works, we expect that it will be slower than the others. This is because for every time step, it has to access the whole matrix twice (for white and then black tiles), while all the others only have to do this once.

1.3.6 Implementation choices for different scalabilities

As a final note we will discuss more technical choices on how we decided to run our code to obtain the results shown below.
All the codes are run using the Epyc nodes available. For all time results concerning evolution methods, we fixed the number of evolution steps to 100 in all of them.
Concerning the binding policy for the OpenMP threads, we ran everything by using the default `OMP_PLACES=cores` and opting for a `close` binding policy. This choice was done in order to try and have threads working inside the same NUMA region, so that they can access the same memory. Moreover, this encourages the usage of a shared L3 cache (every 4 cores on EPYC). We expect that different options would alter our results when we study the OpenMP scalability with a small number of threads.

OpenMP scalability
For what concerns OpenMP scalability we decided to run all our codes with two nodes and fixing the number of MPI processes to one per socket. As such, we have a total of four processes. This is done by using the commands `--map-by node` and `--bind-to socket`. Separating the four processes on four different sockets ensures that each process can have up to 64 cores and exactly one is used by each thread.
We worked on two different matrices of size 20000×20000 and 10000×10000.

Strong MPI scalability
Regarding strong MPI scalability we opted to run all our codes on two nodes. We decided to map each MPI process to a NUMA region of a node. This is done to have the capacity to run more processes on the same node than by using a socket for example. The code is run on two nodes and taking measurements from 1 up to 16 processes. Since each NUMA region contains 16 cores, the number of OMP threads is fixed to 16.
At first, we placed the MPI processes using `--map-by node` and `--bind-to numa`. By doing so, processes working on contiguous rows of the matrix are placed in the two separated nodes, starting with the first one.
However, we thought that a way to improve the communication among processes would be to place processes working on contiguous rows of the matrix on the same node. This reduces the number of messages which are sent and received "outside the node". To obtain this we ran the code using `--map-by numa`.
A simplified example of these two situations can be seen in Figure 8, where on the left we can see how the processes are placed if we use `--map-by node` and `--bind-to numa` while on the right if we use `--map-by numa`. In section 1.4 we will see how this changes the results.

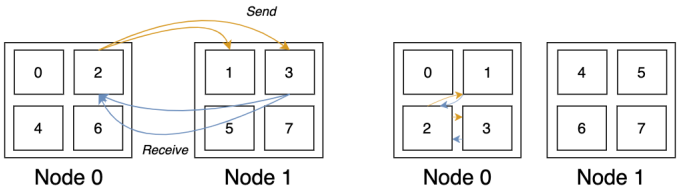

Figure 8: How 8 processes are placed on 2 nodes with 4 NUMA regions: LHS `map-by node bind-to numa`, RHS `map-by numa`

Weak MPI scalability

To study the weak MPI scalability of the different evolution methods, we have to increase the number of processes while keeping the amount of workload per process constant.

We decided to start with a matrix of size 10000×10000 and 1 process, so that is the amount of workload each process should have.

We ran the code using `map-by node bind-to socket` as before using the maximum number of threads, and measured scalability as we increased the number of processes from 1 to 6.

Size	Processes
10000×10000	1
14142×14142	2
17320×17320	3
20000×20000	4
22360×22360	5
24494×24494	6

Table 1: Size for increasing number of processes

1.3.7 Time measurements

The first option to collect timings, which is often used in serial code, is to use CPU times. The issue with this option is that when multi-threaded code is run, the sum of the CPU time of each thread is returned, which does not reflect how costly our program is in terms of total time. For this reason we settled on using the `omp_get_wtime()` function, which uses wall-clock time.

Another thing to consider is how to collect the timing since we have many processes and threads working on the same code. Since we wanted to have the runtime of the code from start to finish, we decided that there would be a barrier at the end of each code section for which times are reported, followed by only one process printing the runtime. This way we are sure that the time we collect actually corresponds to the whole code having finished running, and not to some process having finished before.

1.4 Results & Discussion

Now we present the results obtained for the different evolution methods. At the end of this section, some additional timings are provided for the operations of generating the matrix, reading and writing to file.

The time for each different configuration is taken 5 times and are reported in seconds. Then

the plots are obtained using the mean of those. For the figures where the time is shown, an additional band with ± 1 standard deviation is also provided, to give an idea of the variability of the results.
In general we will show two plots. The first will contain the times while in the second the speedup is plotted.

1.4.1 Static Evolution

OpenMP scalability

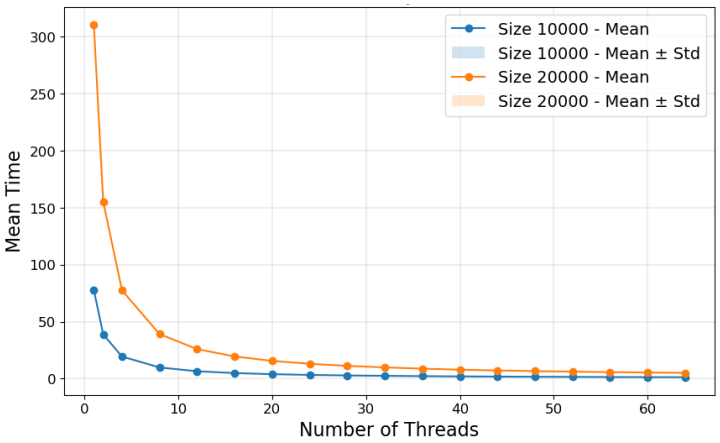

Figure 9: Evolve static OpenMP time

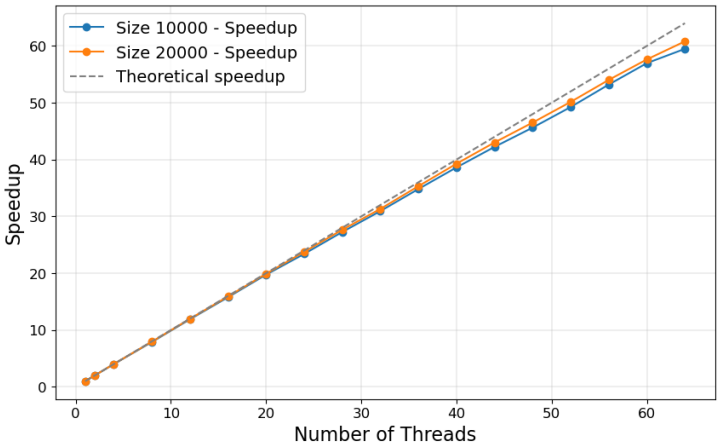

Figure 10: Evolve static OpenMP speedup

By looking at Figure 9, we can notice how the variability bounds are not noticeable. This implies that the results obtained are quite stable.
The speedup plot in Figure 10 shows that scalability is good. One can notice how the graph for both matrices is close to the theoretical speedup plotted in gray. It also appears that the larger matrix scales slightly better for an increasing number of threads. This is expected, as there is a larger amount of workload.

Strong MPI scalability

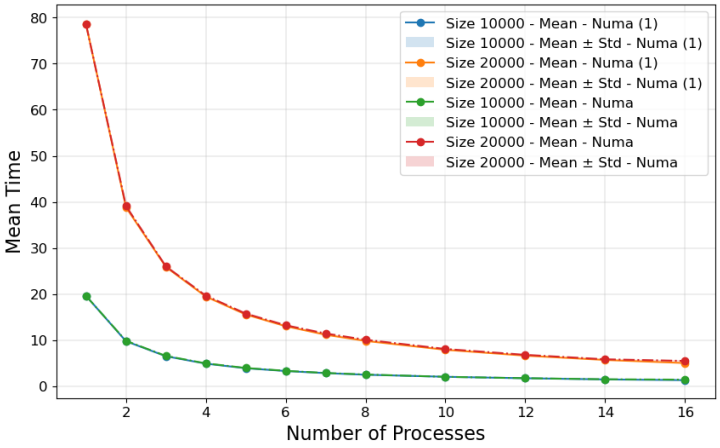

Figure 11: Evolve static Strong MPI time

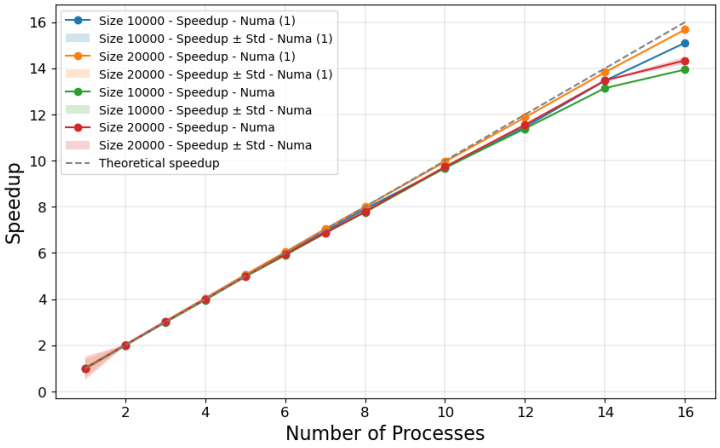

Figure 12: Evolve static Strong MPI speedup

The plots presented here show the results for strong MPI scalability on matrices of size 10000×10000 and 20000×20000 and using the two options for binding MPI processes described in Section 1.3.6. The ones containing (1) in the legend are the ones obtained by using `map-by numa`. Note that the orange curve for (1) and the red one overlap, and the same occurs for the blue and green one.

Similarly to before there does not seem to be much variability. Also when working with strong MPI scalability, the static evolution method appears to be scaling well, with bigger workload scaling slightly better.

It also appears that our intuition about the placement of MPI processes was correct, and reducing the number of extra-node communications seems to help scalabililty, especially for a larger number of processes.

Weak MPI scalability

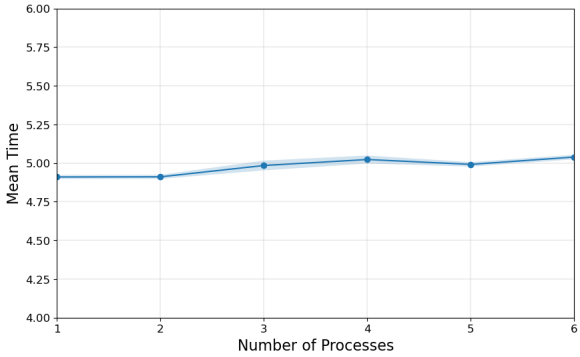

Figure 13: Evolve static Weak MPI time

By increasing the number of processes while keeping the amount of workload per process fixed, the time, as show in Figure 13, remains more or less constant. Also here we notice that variability is very small.

The speedup plot was not included as it does not provide any additional insights.

1.4.2 Ordered Evolution

Due to the intrinsically serial nature of this evolution method, we do not expect it to scale well. For this reason, we reduced the number of options used to run the code. To be more precise:

- We only used one single matrix of size 10000×10000.
- We increased the interval between threads/processes for which measurements are taken.

OpenMP scalability

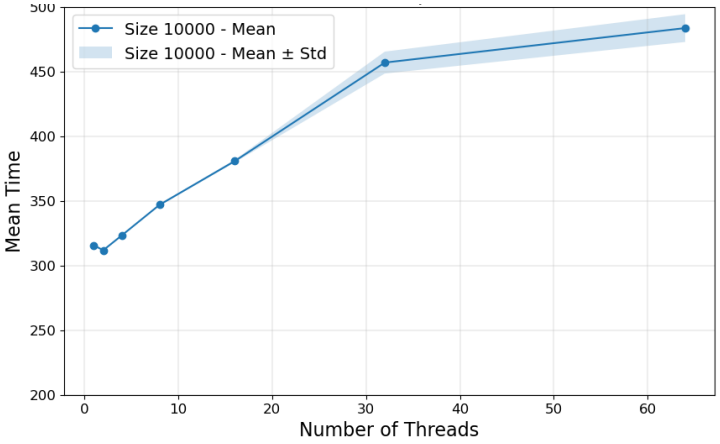

Figure 14: Evolve Ordered OpenMP time

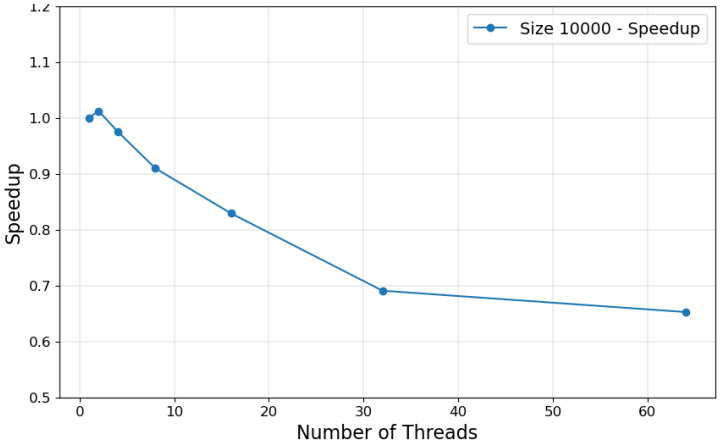

Figure 15: Evolve Ordered OpenMP speedup

As expected both Figure 14 and Figure 15 show that the code does not scale the performance gets worse as the number of threads increases. We presume that this is due to the increased synchronization cost for the ordered execution of the threads.
We can notice a very small improvement when moving from working with one thread to 2. We are unsure if this is due to noise or to better memory usage.
We can also note from Figure 14 that as the number of threads increases so does the variability, which before was very small.

Strong MPI scalability

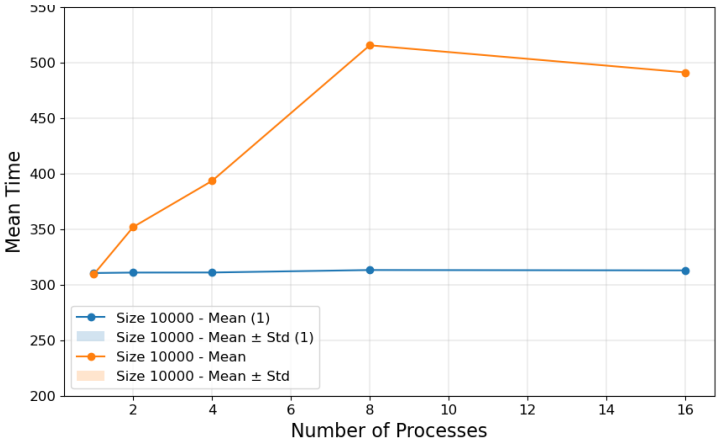

Figure 16: Evolve Ordered Strong MPI time

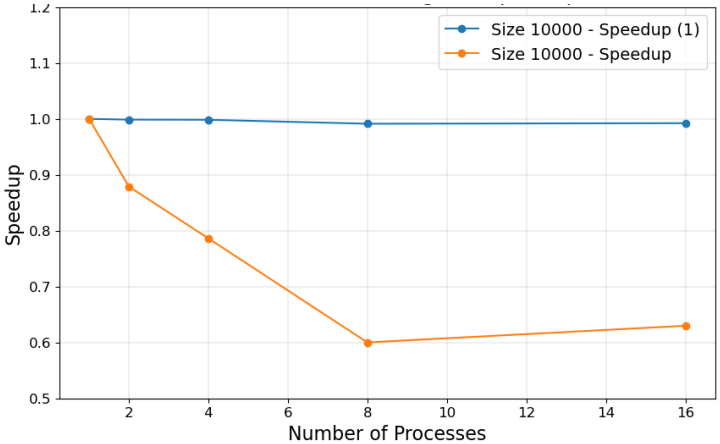

Figure 17: Evolve Ordered Strong MPI speedup

As in the static evolution case, we ran the code using the two different binding options described in Section 1.3.6, with (1) still referring to the usage of `map-by numa`. Both Figure 16 and 17 show that the method does not scale as the number of MPI processes increases. We imagine that this is due to the increasing number of MPI communications among processes which is only an additional cost since the intrinsic seriability of the procedure prevents it from being truly parallelized.
The two different binding policies for MPI processes show much different results. If we focus on the blue plot (corresponding to (1)), we can see that while it does not scale, the time of execution seems to increase in a way that is barely noticeable.
We expect this to be caused by the fact that this option does not lead to an increase in number of extra-node communications as we increase the number of processes, as explained in 1.3.6. Possibly the cost of the higher number of messages is balanced by a better memory usage.

As a final note, we decided not to include the plot for Weak MPI scalability as we have already seen the behaviour of this evolution method, and no additional insights would arise (the results can be found in the Github repository).

1.4.3 White-Black Evolution

As previously mentioned, only OpenMP scalability was performed for this method.
We decided to include two additional odd size matrices to take time measurements. This is done because we expect that our code implementation for those will perform slightly worse.

OpenMP scalability

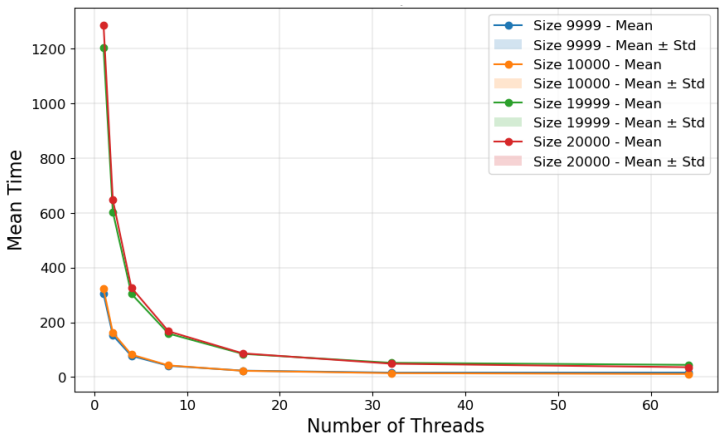

Figure 18: White-Black OpenMP Time

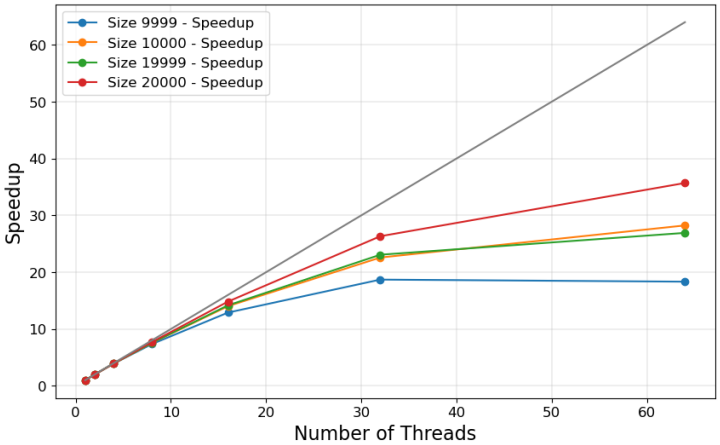

Figure 19: White-Black OpenMP Speedup

As we can see from Figure 18 and Figure 19 the code does not scale well starting at around 16

threads. This could be due to the fact that at every iteration we have to open a parallelized for loop for every row, and this additional cost adds up. Moreover, we can't exclude the possibility of false sharing occurring as the threads work in parallel inside a single row.
As expected, in Figure 19 we can see how matrices with an even number of columns/rows obtain better scale-up than odd ones. One can also notice that the smaller even matrix (in orange) obtains a better speedup than the larger odd one (in green). This goes against what we have seen thus far with matrices with larger workload scaling better.
We expect this to be due to the code for odd matrices having an additional serial portion of the code to update the last column.

1.4.4 Generate, Read and Write

As explained in Section 1.3.2 we also parallelized the operations of random matrix generation, its writing to pgm file and the corresponding reading. However, the time needed for the generation and reading from a file are very small and they do not excessively affect the total execution time of the code by much. Moreover, these operations are executed only once. On the contrary, the write operation could be performed more than once during the evolution. This is because the code implements the possibility to write a snapshot to file every certain number of steps. Therefore, we decided to not further expand the analysis of the first two (some results can be found in the GitHub repository) and we decided to focus on the writing operation. This operation does not use OpenMP threads but only MPI processes. As such below the results obtained for a strong MPI scalability study are reported.

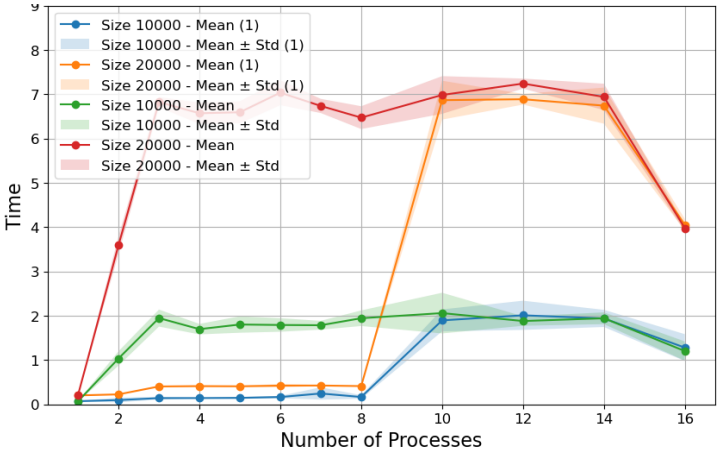

Figure 20: Write Strong MPI time

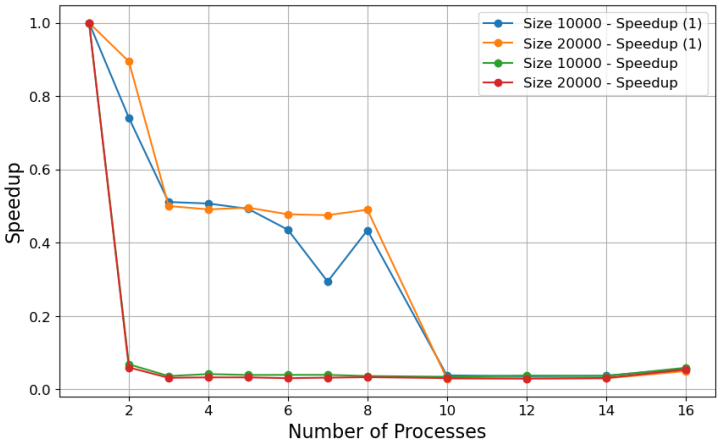

Figure 21: Write Strong MPI speedup

Just like all other previous strong MPI plots, also here we included the two binding options described in 1.3.6, where (1) still denotes the `map-by numa` option.
Regardless of the option or matrix size the process of writing to file using MPI I/O function

does not scale. We also noticed that whenever the operation of writing to file occurs from 2 different nodes the time increases drastically. This occurs both in the (1) option, when we move over 8 processes and also in the other one when we stop working with only one process.
Another thing we notice is the slight improvement when working with 16 processes.
We believe that fixing this would require us to have a further look in the machine network and file system and how it operates with the MPI operations to write to file. As such we do not have enough evidence to support our conclusions on this and further investigation is required.

1.5 Conclusions

We conclude by having a quick comparison of the speedup achieved by the three evolution methods, shown in Figure 22 and Figure 23. Regarding OpenMP scalability we include all the three methods, while for Strong MPI scalability we can only compare the static and the ordered method.

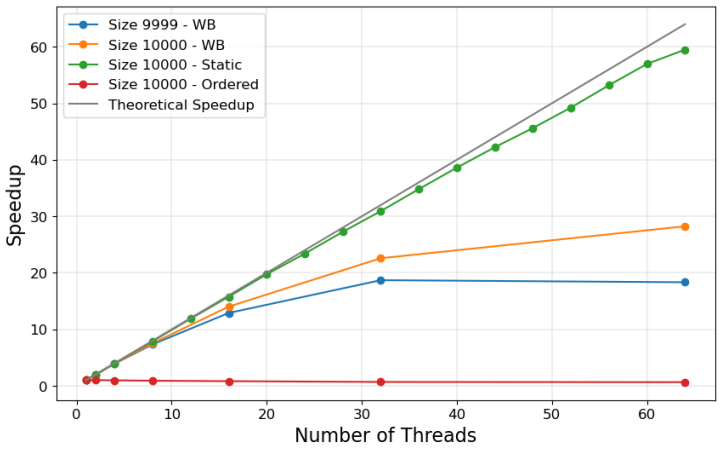

Figure 22: OpenMP comparison

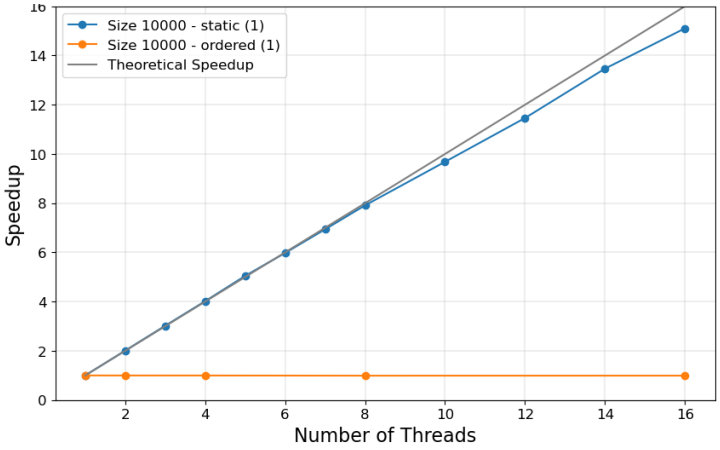

Figure 23: Strong MPI comparison

We start with the plot on the left, showing OpenMP scalability. The static evolution is the only one to obtain a satisfactory level of scalability and is much better than the other methods. For the White-Black method we achieve some scalability, which is better for the even case, but still significantly worse than the static evolution. Finally, the ordered evolution is much worse than both as it does not scale. Note however, that the scalability limitations for the ordered method arise from it being intrinsically serial, while the opposite can be said for the static method.
On the right we have a similar situation, showing strong MPI scalability of static and ordered execution. Similar conclusions can be drawn.
Some possible improvements in our implementation are:

- Introduce the OpenMP I/O operations in reading and writing implementation. This can

lead to a better usage of the memory and so to a better scalability/parallelization of these operations.

- Further investigate how the MPI I/O operations work. This would improve the scalability of our read/write implementation which we have seen previously we were unable to scale.
- Regarding the White-Black evolution method, a possible improvement could be to introduce a further level of parallelization. Consider the even case and only two rows for simplicity. Suppose row 0 has already updated up to a certain point the tiles of some colour. A different thread can start updating tiles of the same colour two steps before in the following row. Figure 24 below helps with this idea.

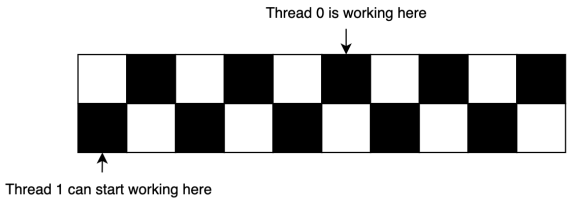

Figure 24: White-Black possible improvement

This idea extends to more than two rows and with some adjustments also to the odd case. To implement this, one could exploit nested parallelism with OpenMP threads. One possible drawback is that for small matrices the amount of synchronization required could outweigh the parallelization advantages.

2 Exercise 2

2.1 Problem Description

The goal of this exercise is to compare the performances of three math libraries available on HPC: MKL, OpenBLAS and BLIS.
The comparison is done by focusing on the level 3 BLAS (Basic Linear Algebra Subprograms) function which is called *gemm*. Gemm stands for *General Matrix Multiply*, and it performs the following operation:

$$C \leftarrow \alpha AB + \beta C$$

Where α, β are scalars, while C, A, B are matrices of size $(M \times N), (M \times K)$ and $(K \times N)$ respectively. Different versions of this function are implemented, one for double precision ($dgemm$) and one for single precision ($sgemm$).
The code which is used is a standard *gemm* code where three matrices A, B, C are allocated. In the code provided the three matrices are all square matrices. Moreover, the scalar parameter are set as $\alpha = 1$ and $\beta = 0$. Matrices A and B are filled and then the BLAS routine computes the matrix-matrix product as $C = A * B$.
The exercise requires us to perform a scalability measure over the size of the matrix with a fixed number of cores and a scalability measure over the number of cores at fixed size of the matrix. The cores are used by OpenMP threads to perform the operations in parallel. We ran all the codes both on the Epyc and the Thin nodes and their performances will be shown. A comparison with the theoretical peak performance is also reported.

2.1.1 Theoretical Peak Performance

In this exercise we will be referring to two different theoretical peak performances.

- In the GFLOPS plot we report the maximum number of operations (in GFLOPS) that can be performed on the hardware available.
- When describing the speedup plots, we refer to theoretical peak performance as the ideal scalability achieved as the number of threads is increased.

The one described here is the one used in the GFLOPS plots. The theoretical peak performance when using a certain number of cores can be computed as:

$$\text{FLOPS} = \text{clock rate} \cdot \text{number of FP operations per cycle} \cdot \text{number of cores}$$

To determine it we need to make some considerations about the two different architectures which are used.
The Epyc nodes consist of two sockets each containing a *AMD Epyc 7H12 64-Core Processor*. By having a look at the specifics for the node one can determine that its clock rate is $2.6GHz$

for each core[1]. We were unable to find exact information on the number of FP operations per cycle performed by this kind of nodes. However, some websites[3] state that most modern Epyc nodes can do up to 16 FPs per cycle in double precision and 32 in single precision. As such to determine the T.P.P. for the Epyc nodes when using n cores we can use these results to obtain $2.6 \cdot 32 \cdot n$ for single precision, and half that amount for double.
Each Thin node consists of two sockets each containing a *Intel(R) Xeon(R) Gold 6126 CPU*. The theoretical peak performance for this can be found in the course slides. We are told that for a total node is 1.997 TeraFLOPS when using double precision, meaning that each core has a peak performance of 83.2 GFLOPS. As such to determine the T.P.P. for the Thin nodes when using n cores we can use these results to obtain $2 \cdot 83.2 \cdot n$ for single precision, and half that amount for double.

2.2 Results & Discussion

In this section the results obtained by running the different implementations of the *gemm* function are reported.
The code provided allows us to obtain and analyze both the time of execution and the GFLOPS achieved. This latter measurement will also be compared with the theoretical peak performance for each node, described in Section 2.1.1. Moreover, the speedup is also computed as the number of cores is increased.
Time and GFLOPS measurement are taken 5 times to obtain an average and an estimate of the standard deviation, which are reported in the plots. The time measurements are reported in seconds.
Section 2.2.1 contains the plots obtained to study the scalability over the number of cores at a fixed size of the matrix. We decided to use a 10000×10000 matrix. Then the number of cores is increased from 1 all the way to use all the cores in the node (meaning 128 for Epyc and 24 for Thin nodes).
Section 2.2.2 contains the plots referring to the scalability study over the size of the matrix at fixed number of cores. Specifically, the number of cores is fixed to 12 for Thin nodes and to 64 for the Epyc ones. Then the matrix size is increased from 2000×2000 to 20000×20000.
We ran all the codes using both single and double precision, using all the three math libraries MKL, OpenBLAS and BLIS. Moreover, we tried different thread allocation policies, which are *close* and *spread*, while keeping the thread places fixed to cores.
As an important note, we always ensured that there was never more than one thread running on the same core.

2.2.1 Cores scalability

This section reports the plots for the cores scalability. We fixed the size of the matrix to 10000×10000 while increasing the number of cores from 1 up to 24 for the Thin nodes and from 1 up to 128 for the Epyc ones.

Epyc - cores - double

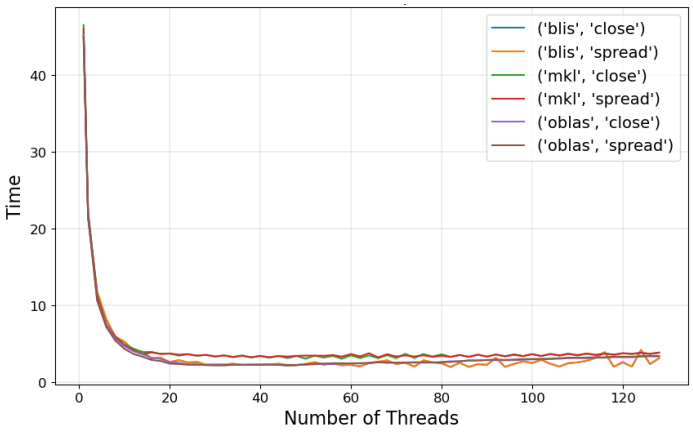

Figure 25: Time core scalability on Epyc with double precision

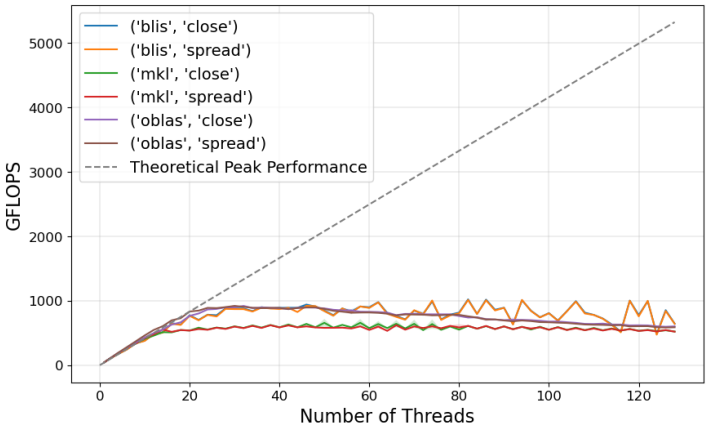

Figure 26: GFLOPS core scalability on Epyc with double precision

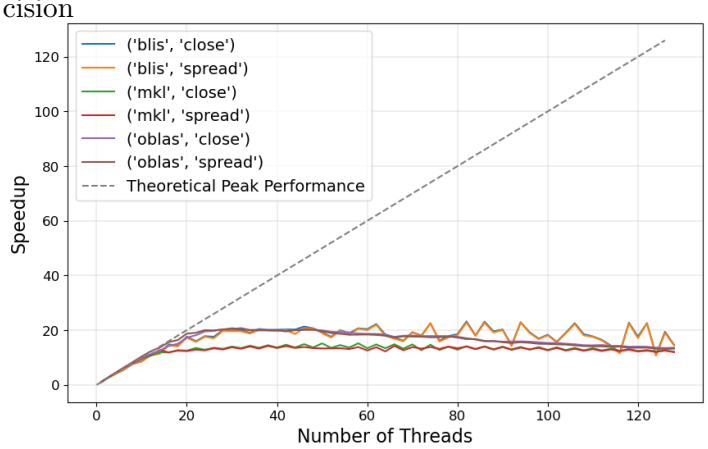

Figure 27: Speedup core scalability on Epyc with double precision

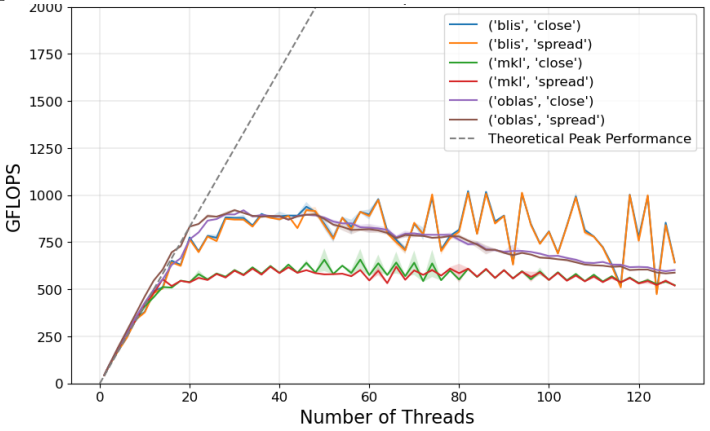

Figure 28: GFLOPS core scalability on Epyc with double precision - zoom

Figures 25, 26 and 27 show respectively the results for time, GFLOPS and speedup achieved, while Figure 28 is just a zoomed version of the GFLOPS one, to better show the different methods. This same structure is also used for results below.

Epyc - cores - single

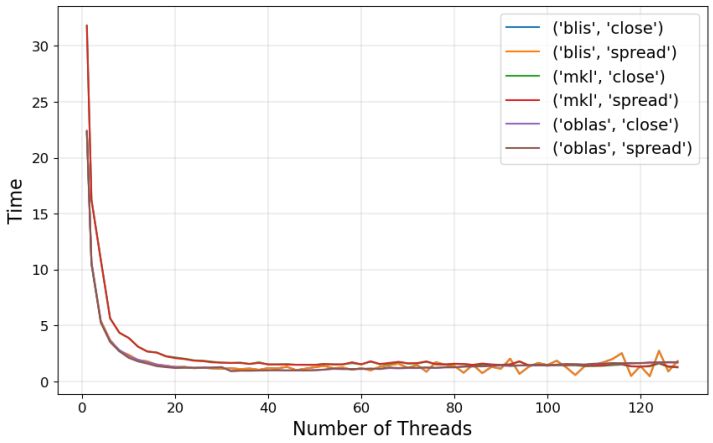

Figure 29: Time core scalability on Epyc with single precision

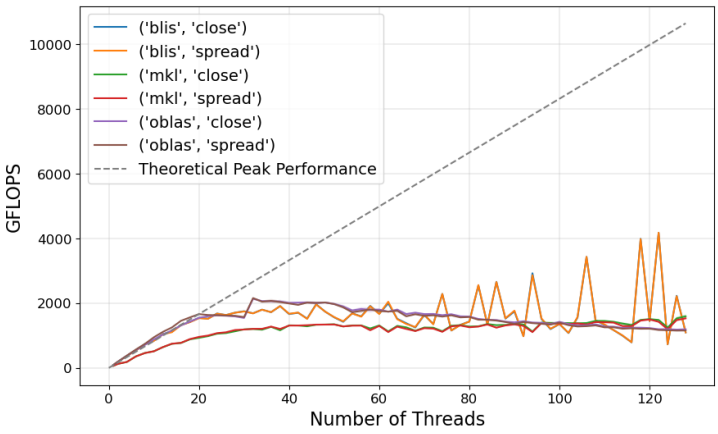

Figure 30: GFLOPS core scalability on Epyc with single precision

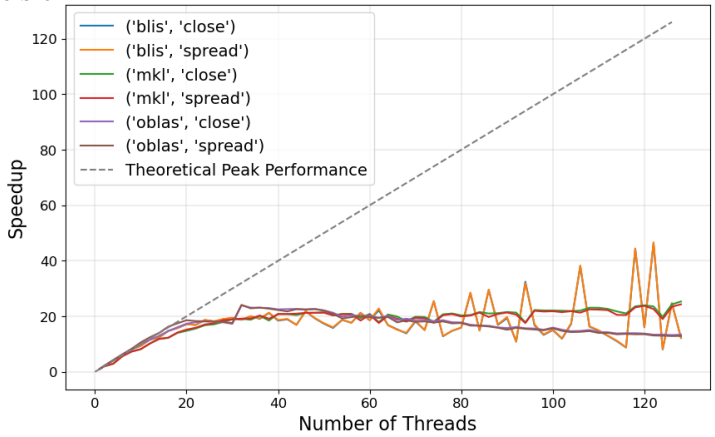

Figure 31: Speedup core scalability on Epyc with single precision

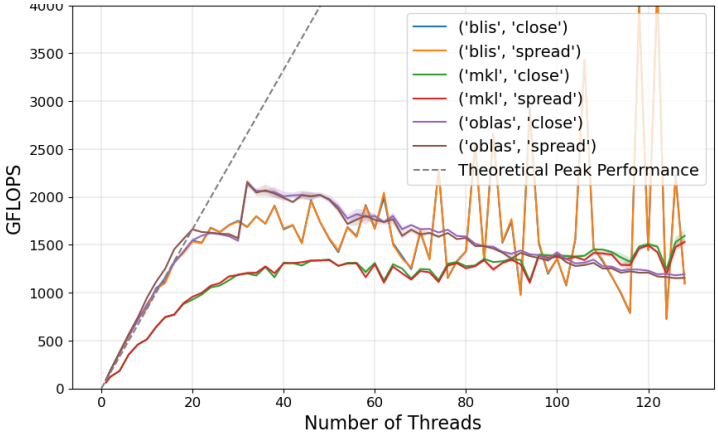

Figure 32: GFLOPS core scalability on Epyc with single precision - zoom

Figures 29, 30, 31 and 32 presented now show the same results but with single precision.
From the graph depicting the time we can notice a decreasing trend as we start to increase the number of threads. Then for both double and single precision, the time to complete this operation stabilizes.
Now focus on the GFLOPS plots. One can notice that as long as the code is run with up to 20 threads, almost all implementations seem to be able to achieve results close to the theoretical peak performance. The only one not exhibiting this behaviour is the MKL implementation when used with single precision. When the number of threads used is larger than 20, there is a different behaviour for each implementation:

- OpenBLAS: it keeps increasing for a bit, after which it shows a decreasing behaviour.
- MKL: it seems to stabilize more or less in both cases.
- BLIS: it seems to oscillate a lot, particularly when single point precision is used.

Regardless of the implementation they all stop improving for a larger amount of threads. We assume that this is due to false sharing occurring more often when a larger number of threads is used on matrices of this size.
Another thing to note is that there is not much difference between the choice of close and spread policy. We were expecting close policy to perform better especially when using a small number of threads.
The highest GFLOPS is achieved by BLIS library in both cases at the peaks of the oscillations. As such, we believe that selecting the appropriate number of threads is essential for an efficient usage of this library. Of course we expect this number to change depending on the matrix size being used.
For what concerns speedup, the plots give more or less the same information as the GFLOPS ones. Up to around 20 threads OpenMP scalability is quite good and close to the theoretical performance.

Thin - cores - double

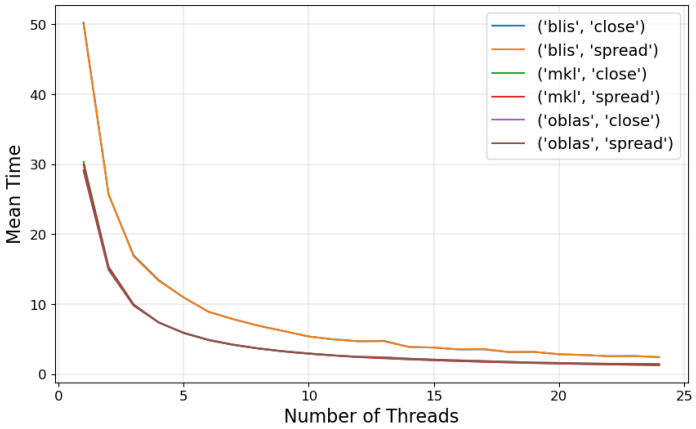

Figure 33: Time core scalability on Thin with double precision

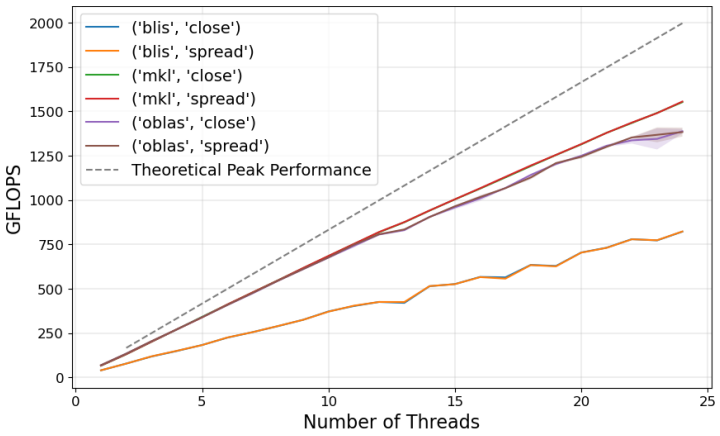

Figure 34: GFLOPS core scalability on Thin with double precision

The analysis reported in this paragraph has a similar structure to the one provided for the Epyc nodes, with figures for both double and single precision results.

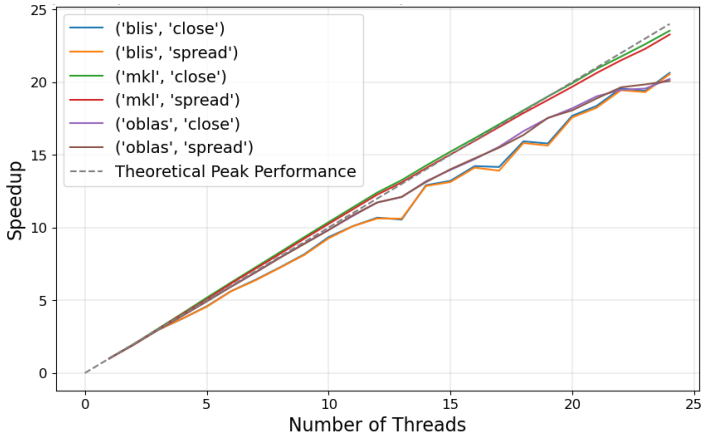

Figure 35: Speedup core scalability on Thin with double precision

Thin - cores single

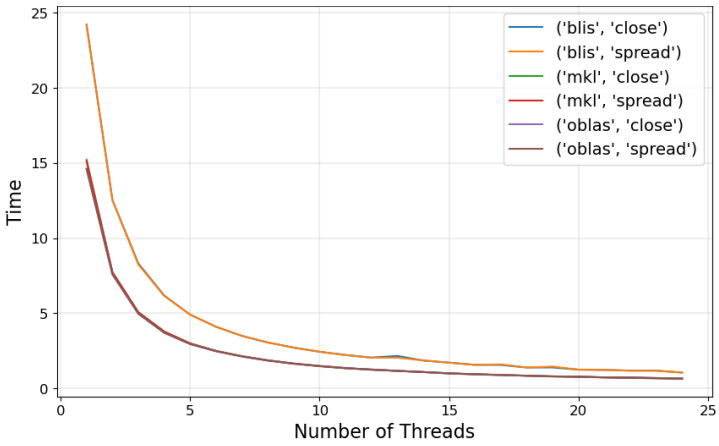

Figure 36: Time core scalability on Thin with single precision

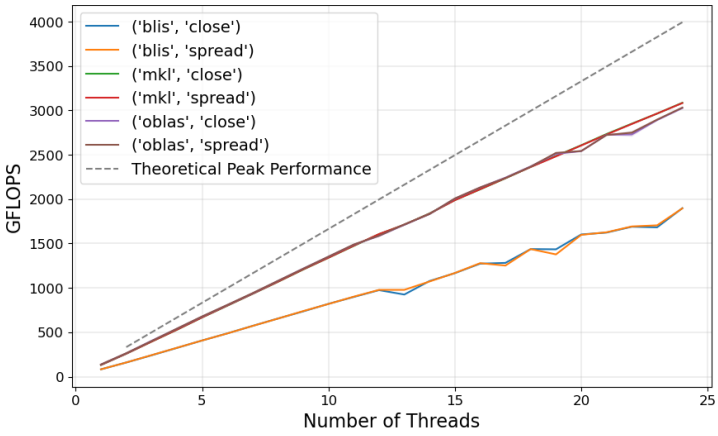

Figure 37: GFLOPS core scalability on Thin with single precision

The results for Thin nodes show that the functions from the MKL and OpenBLAS library outperform BLIS.

By focusing on the GFLOPS plots we can see that in both cases the increase is almost linear. However, they are all well below the theoretical peak performance with this being especially true for the BLIS library.

We can note that MKL and OpenBLAS perform similarly. They only differ when we focus on more than 12 threads with double precision. In this case MKL performs slightly better.

By looking at the speedup plots, all libraries achieve perfect scalability up to 12 threads, with only BLIS lagging behind when using double precision. As we increase the number of threads, MKL remains close to the T.P.P., while the other two deviate from it.

We can additionally notice that once more there is little to no difference between the different

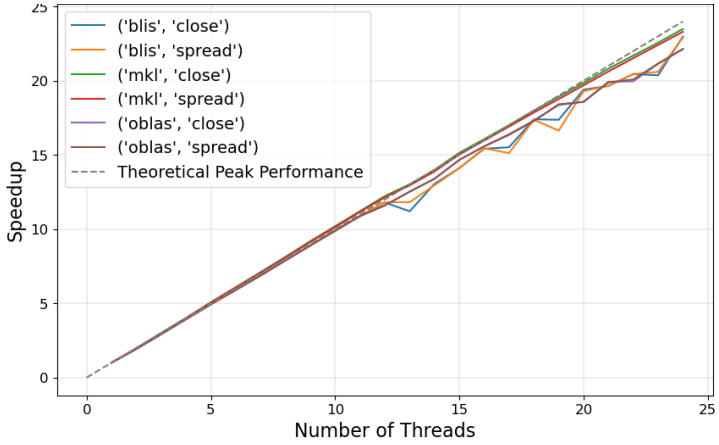

Figure 38: Speedup cores scalability on Thin with single precision

thread binding policies.

Epyc and Thin comparison

By comparing Figures 26 and 34 one thing we note is that the GFLOPS measured on the Thin nodes appear less wiggly and irregular than the ones on the EPYC nodes. Our belief is that this is due to the fact that for Thin nodes the L3 cache is shared for every 12 cores, while this is only true every 4 cores on Epyc. This may reduce the occurrences of false sharing in the L3 cache. The same can be noticed by looking at the equivalent plots with single precision. Making further comparisons would not be useful, as on Thin nodes we only have 24 cores.

We also thought it would be interesting to compare the OpenMP scalability achieved by the two nodes. For comparison purposes Figure 39 shows the speedup achieved using close policy with double precision on the two different types of nodes. Since on Thin nodes we have a maximum of 24 cores, we decided to interrupt the comparison at that value. Overall we can note that the execution on Thin nodes (shown in orange, red and brown) achieves better OpenMP scalability, regardless of the library being used. This is noticeable especially when using more than 5 threads.

Overall it appears that MKL has the best scalability on Thin nodes, but also the worse on Epyc ones.

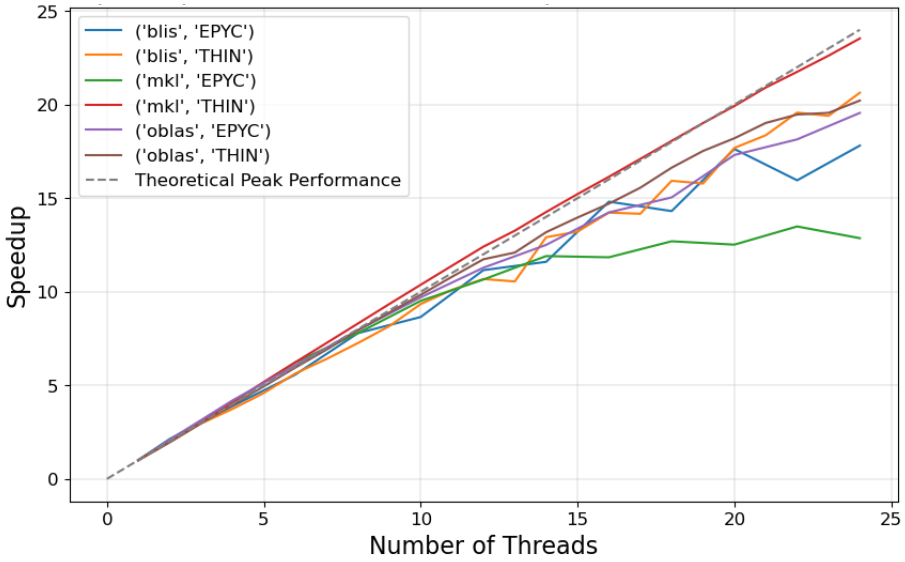

Figure 39: Speedup cores scalability comparison with double precision

2.2.2 Size scalability

This section reports the plots for the size scalability. Specifically the matrix size ranges from 2000×2000 to 20000×20000 increasing the number of rows (and columns) by 1000 each time. The number of cores is fixed to 12 for the Thin nodes and to 64 for the Epyc ones, with exactly one thread running on each.

Epyc - size - double

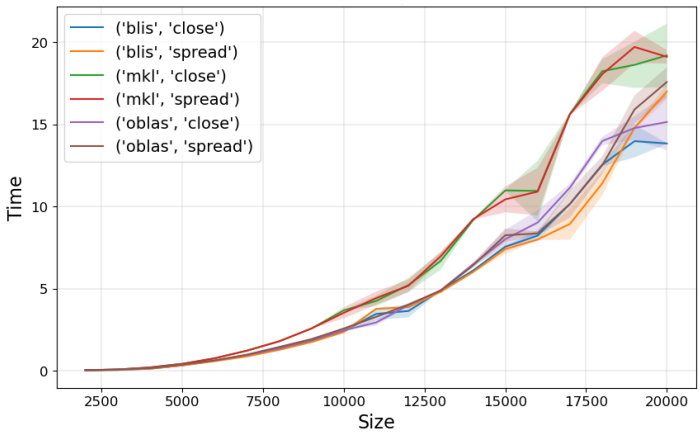

Figure 40: Time size scalability on Epyc with double precision

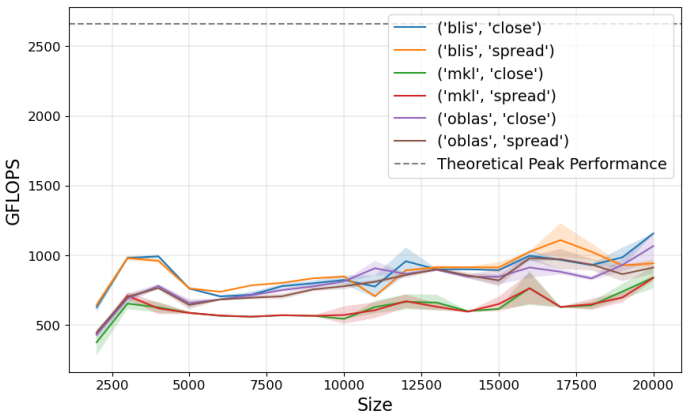

Figure 41: GFLOPS size scalability on Epyc with double precision

Epyc - size - single

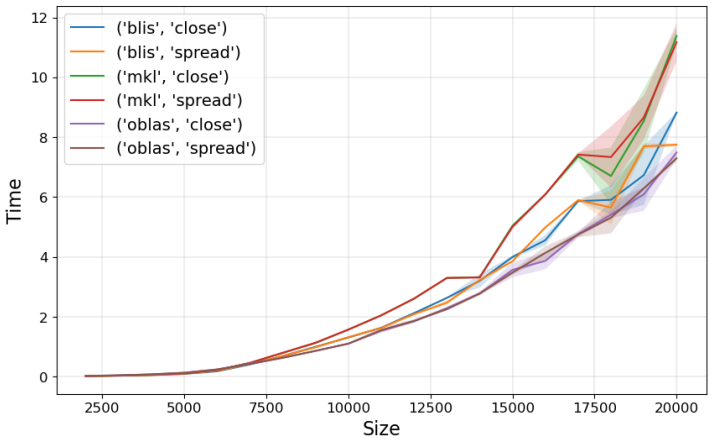

Figure 42: Time size scalability on Epyc with single precision

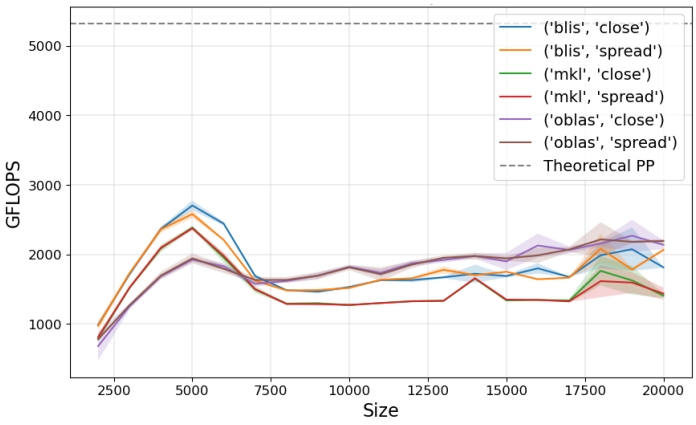

Figure 43: GFLOPS size scalability on Epyc with single precision

Figure 40 and 41 show the time and the GFLOPS achieved while using double precision. The ones below show the same thing but using single precision.

From time graph we can notice an increasing trend, as expected since the size of the matrix increases while maintaining constant the number of threads.

Note that the T.P.P., shown in dashed gray line, is constant since the number of cores used is constant.

By looking at the GFLOPS plots, it appears that MKL performs worse than the other two. Initially there is an increasing trend followed by a decrease. Then, as we approach 20000, a very small increase can be noticed, excluding MKL results.

By having a look at the size of the L3 cache on EPYC nodes, we realized that 3 matrices of size 5000×5000 with single precision, as needed for the exercise and shown in Figure 43, are about the maximum that can fit in it. As such we believe that performance degrades after this as not all matrices can fit in. The same is true for double precision, as shown in Figure 41 with a matrix of size around 3000×3000.

Another important thing to observe, is that regardless of the library or policy, they all remain quite far from the theoretical peak performance.

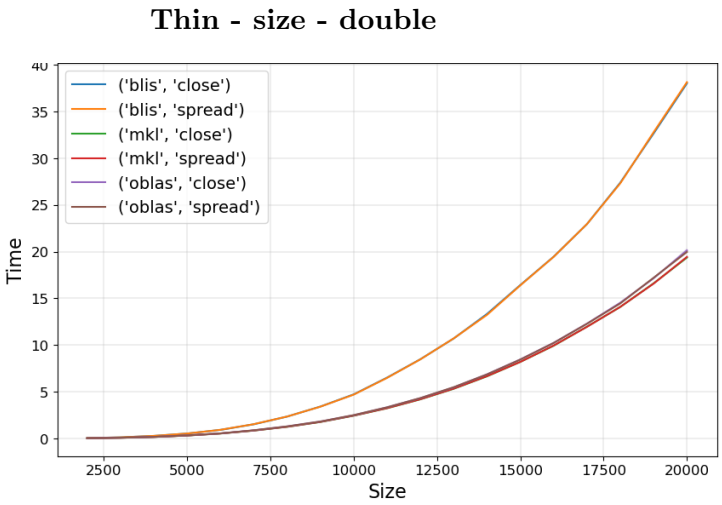

Figure 44: Time size scalability on Thin with single precision

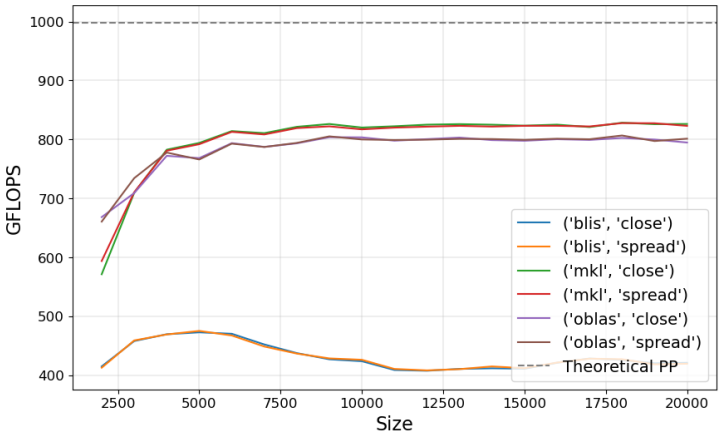

Figure 45: GFLOPS size scalability on Thin with double precision

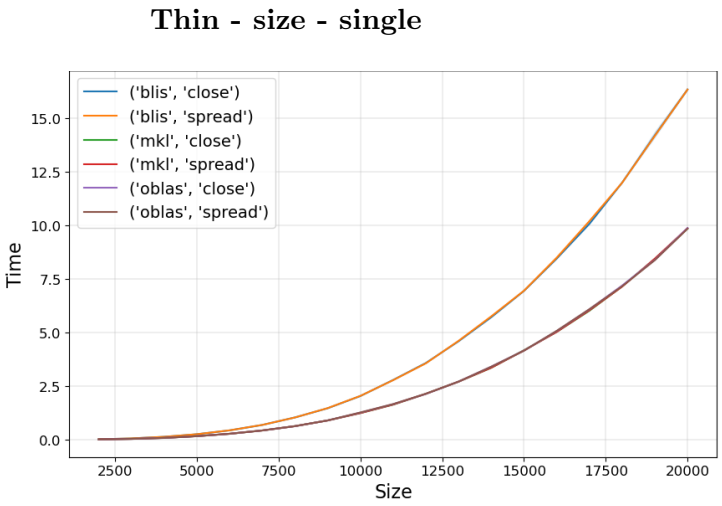

Figure 46: Time size scalability on Thin with single precision

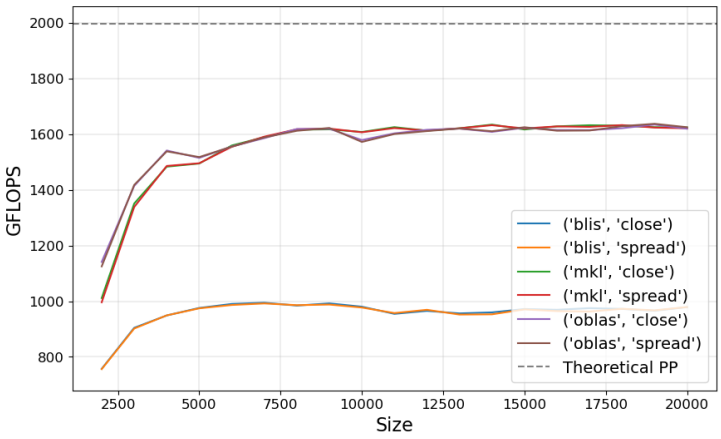

Figure 47: GFLOPS size scalability on Thin with single precision

By having a look at the GFLOPS plots, we can note that BLIS performs much worse than the other two libraries, regardless of the implementation being used. Another general trend to observe is that all methods show an increase in GFLOPS up to matrices of size 5000×5000, followed by a more or less stable behaviour. The only one to differ from this is BLIS, which when working in double precision loses performance.

2.3 Parallel initialization

We noticed that the code provided does not initialize the matrices in parallel. A different approach would have each thread initialize a portion of the matrix in parallel. This comes at the additional advantage of having each thread loading a different part of the memory in cache, which hopefully coincides with how it will be used in the code. The modified code presents the changed section shown below, where the loops for matrix initialization are parallelized with OpenMP:

```
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < (m*k); i++) {
    A[i] = (MYFLOAT)(i+1);
  }
  ...
}
```

We expect this to lead to a better cache usage, as now not only the master thread will have its cache filled with matrix entries.

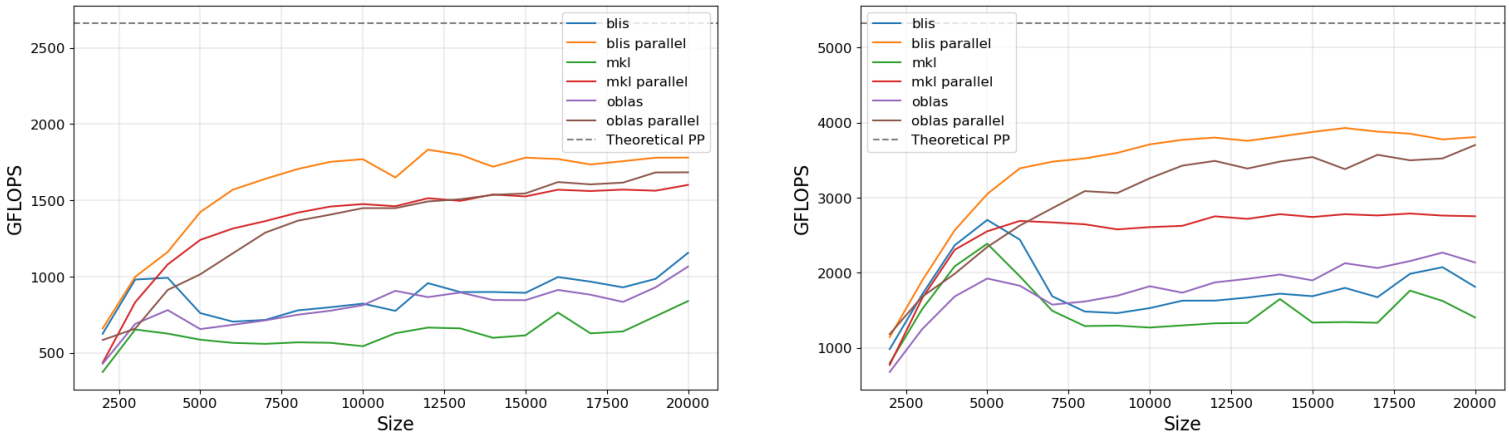

Figure 48: Comparison of GFLOPS with and without parallel initialization on Epyc with double precision on the left and single on the right

In Figure 48 both plots show the results achieved when the initialization of the matrix is done in parallel compared to those when it is not. Note that we only included the results for Epyc nodes with close binding policy for simplicity. We can see that regardless of the precision, we are able to achieve an amount of GFLOPS which is closer to the theoretical peak performance. With this adaptation, BLIS outperforms all other libraries, but also the other two gain in performance. This agrees with our expectations.

2.4 Conclusion

Overall by looking at these plots it becomes clear how the best library to perform the gemm operation depends on the hardware available. In fact our plots show drastically different results depending on the implementation. On the Thin nodes, the MKL library was pretty much outperforming the other two, but on the Epyc ones its performance was poor. Similarly, the BLIS library was capable of achieving higher GFLOPS than the others for some choices of threads and size on the Epyc nodes. Finally, the OpenBLAS library was performing quite satisfactorily on both types of nodes.
Another thing to observe is that not always increasing the number of threads leads to an increase in performance, and for some implementation the appropriate selection can drastically alter the results.

References

[1] AMD. Amd epyc™ 7h12. https://www.amd.com/en/products/cpu/amd-epyc-7h12. [Online; accessed 18-September-2023].

[2] Andrea Mignone. Introduction to parallel programming with mpi — Lecture 5: Parallel I/O. http://personalpages.to.infn.it/~mignone/MPI/lecture5.pdf, 2023. [Online; accessed 13-September-2023].

[3] Microway Blog. Detailed specifications of the amd epyc "rome" cpus. https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-amd-epyc-rome-cpus/. [Online; accessed 17-September-2023].

[4] Wikipedia contributors. Conway's game of life — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=1174265009, 2023. [Online; accessed 13-September-2023].