Università degli Studi di Trieste

Game of Life and GEMM

Foundations of High Performance Computing

Gabriele Codega
Sep 2023

Contents

Introduction **2**

1 Game of Life **3**
1.1 Introduction . . . 3
1.2 Methodology . . . 4
1.3 Implementation . . . 5
1.3.1 Initialisation . . . 5
1.3.2 Evolution . . . 6
1.4 Results . . . 8
1.4.1 Ordered evolution . . . 8
1.4.2 Static evolution . . . 9
1.5 Conclusion . . . 13

2 GEMM benchmark **15**
2.1 Introduction . . . 15
2.2 Methodology . . . 15
2.3 Results . . . 15
2.3.1 Scalability over size . . . 15
2.3.2 Further considerations on scalability over size . . . 19
2.3.3 Scalability over number of threads . . . 19

Preamble

This report covers the solutions to the exercises for the final exam in Foundations of High Performance Computing held at University of Trieste in a.y. 2022/2023.

The report is meant to be a description of the work and a discussion of the results. As such, it is divided into two parts, one for each exercise (see more about the assignment on this GitHub page).

The first part covers the discussion about implementation and scalability tests for a hybrid MPI/OpenMP code implementing Conway's Game of Life. There I discuss some implementation choices and show some measurements of the performance for my code.

The second part covers the benchmark of level 3 BLAS functions from different linear algebra libraries. The contents are mostly related to the discussion of results and possible ways to improve performance.

Chapter 1

Game of Life

1.1 Introduction

Conway's Game of Life simulates the evolution of a population over time. The population is modelled as a matrix, whose entries indicate whether each individual is alive or dead. The evolution of this population happens according to some rules, which determine the state of each cell in the next generation depending on the state of neighbouring cells in the current generation. For more details about the rules, visit for instance [1] or [2].

The goal of this assignment is to build a scalable hybrid MPI/OpenMP code which implements a modified version of Conway's Game of Life. The code is required to have some features which allow the user to control parameters such as the grid size, the input file and the type of evolution. The code also implements two kinds of evolution, namely *ordered* and *static*, which evolve the grid dynamically, one cell after the other, and statically, freezing the whole playground for each generation.

In this code, we disregard the state of a cell to determine wether it will be alive or dead in the next generation. This is a modification to the original rules and has some implications, such as that known stable or repeating patterns will no longer be stable or repeating.

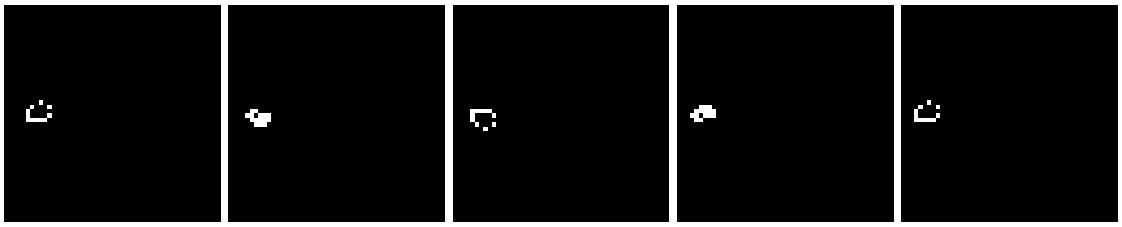

Figure 1.1: *Example of a moving pattern, evolved with static evolution and standard rules.*

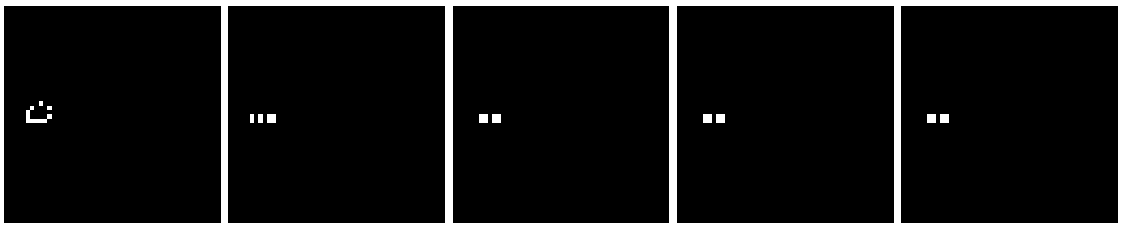

Figure 1.2: *Example of a moving pattern, evolved with ordered evolution.*

More details about the usage of the code can be found in this GitHub repository.

1.2 Methodology

There are two main tasks to be taken care of: evolving the playground and data I/O.

Evolution This is the core of the application and even in its serial version it requires some care. The first issue to tackle is that updating the value of a cell requires to check whether some conditions on the number of neighbours hold, and checking for these conditions with `if` statements is not optimal for performance.

After taking care of this, another issue is related to finding an efficient pattern to access data in the grid, and consequently a feasible domain decomposition among tasks and threads. There are many possibilities when it comes to defining a suitable access pattern to 2D data, such as row-major, column-major or strided. Since the code is implemented in C, the most natural way to access data is in a row-major pattern and, although for some applications it might not be as good as a strided access, in this case it would be a viable choice. Since the data are accessed by row, the natural domain decomposition would be to divide the grid in such a way that each MPI task gets a certain number of rows, which then will be redistributed among OMP threads.

The last issue to tackle to have a working evolution is communication among tasks. As a matter of fact, since MPI works in a distributed memory environment, it is necessary that processes communicate to eachother the boundaries of their regions, so that every process can correctly compute the number of neighbours for each cell. The pattern for communications depends on the type of evolution and we shall see the details later on, however, in general every process needs to communicate with the preceding and the following, in a circular fashion.

Data I/O The issue of data I/O arises as we require that the code has the ability to initialise a playground from a file and to periodically write a checkpoint to a file.

Since the target machine has a parallel filesystem, it is sensible to use shared files, to which each MPI task can access at the same time.

The input part is related to the problem of proper memory initialisation. Indeed, as data should be as close as possible to the process that access them, it is important that each thread initialise its own chunk of the playground. To achieve this there are at least two possibilities, which determine two different ways to tackle the input.

The first possibility is to have each thread initialise its own memory to some value, say zero, then have one single thread make the call to an MPI read function and overwrite the pre-initialised memory with the actual data.

The second possibility is to have each thread make a call to an MPI read function and only read its portion of the matrix, which can be used to directly initialise the memory to the initial conditions.

The first possibility has the advantage that one single thread is involved in communications, with the drawback of having to traverse the matrix two times, one of which using a single thread.

The second possibility is more convoluted. It requires more care in handling communications, since it may be the case that only one thread is allowed to make an MPI call at any given time, and overall requires possibly many more calls to MPI functions, all of which introduce some overhead. The pro of this solution is that only one traversal of the memory is required, which might become beneficial for large playgrounds.

However, since the input happens only once per run, this piece of code is not expected to become a bottleneck for performance.

On the other hand, for the output one needs a bit more care, as the user may decide to write a checkpoint at every evolution step, in which case having an efficient output is crucial.

As for the input, we have two possibilities, one that involves using one single thread and one that involves using all the threads. In this case, since there is no need to initialise memory, using one single thread might be the best choice, since it avoids the necessity of having many MPI calls and hence reduces the parallel overhead.

1.3 Implementation

Let us now discuss in some detail the implementation of the ideas presented above. For clarity we shall devote a subsection to each part of the code.

1.3.1 Initialisation

Random grid

The code has the ability of generating a grid of given size, filled with pseudorandom values in $\{0, 1\}$.

Each task needs to initialise the memory to store its chunk of the grid and to do so it has to compute its workload. Each task computes its workload independently of the other tasks.

```
procwork = ysize/numproc + (procrank < (ysize%numproc));
```

This expression assigns at least $\lfloor ysize/numproc \rfloor$ rows to each tasks and the remaining rows are evenly distributed starting from the first process, one row per process until all rows have been assigned. This small work imbalance is not significant as the workload per task increases.

At this point, each task can allocate the required memory, which is initialised in a parallel for by OMP threads. To ensure that all sections of the grid are different from eachother, each thread initialises a random number generator with a different seed

```
long int seed = time(NULL) + thid + procrank;
srand48(seed);
```

This is likely not the best way to generate random numbers in parallel, but for the purpose of this code this is good enough. The values needed for the grid are either 0 or 1, which are generated as `(int)(drand48()*2)`.

Once the playground is initialised it can be written on a file for later use, operation that requires a bit more machinery.

For starters, the chioce format for data is pgm. This means that the first thing to do when writing a new file is producing a suitable header. The header contains some essential information but in general is very short, hence it can be written by a single process. To be more specific, the root process uses one single thread to create a file, write the header and take note of the length of said header. Once the root is done writing, it broadcasts the length of the header to all other processes, that need this information in order to write in the correct location of the file.

Adopting one of the possibilities listed above, the actual parallel write is done by employing only one thread, wich writes the whole chunk.

```
#pragma omp master
    MPI_File_write_at_all(fhout,writeoffset+procoffset,(void*)(grid),
        procwork,MPI_BYTE,&status);
```

As can be seen from this snippet, this function requires an offset and the buffer size, which are specific to each process and need to be computed beforehand.

The buffer size (`procwork`) is the workload assigned to each process, which is computed as in the snippet above.

The offset is given by the sum of `writeoffset`, which is the length of the header, and another offset which is process specific. The process offset refers to the offset of each process relative to the beginning of the data and is given by the following expression

```
procoffset = ysize/numproc * procrank + \\
        (procrank >= (ysize%numproc)) * (ysize%numproc) + \\
        (procrank < (ysize%numproc)) * procrank;
```

This expression assigns to each process a base offset, given by the base work of the processes with lower rank, plus some other offset given by the number of extra lines assigned to processes with lower rank in the case where the workload can not be perfectly split.

From file

Initialisation from file is required every time the code is used to evolve a playground (i.e. it is not used to generate a playground) and happens in two distinct steps.

The first step is to read the header, which, as for writing, is handled by a single thread in a single task. This thread saves all the important information from the header into an array, which is then broadcast to all the tasks. At this point, each process can compute its workload and offset, just as explained above, and can allocate the required memory. Each process then opens a parallel region where threads compute their own workload and offsets. The second step to initialisation is then the actual reading of the data, which happens like this

```
#pragma omp critical
    MPI_File_read_at_all(fh,globaloffset+procoffset+thoffset, (void*)(
        mygrid+xsize+thoffset),thwork,MPI_BYTE,&status);
```

In this snippet, `globaloffset` and `procoffset` are the same as above, while `thoffset` is the offset of each thread's data with respect to the beginning of the process' data and can be computed similarly to `procoffset`. Finally, `thwork` is the workload assigned to each thread and is analogous to `procwork`.

1.3.2 Evolution

As anticipated, there are two kinds of evolution, which are handled by different routines and require different communication patterns, which we shall examine separately.

Despite the differences in communication and in the order of updating the grid, both routines apply the same rules to determine whether a cell shall live on to the next generation. Namely, for this version of the Game of Life, a cell lives on to next generation, or becomes alive, if the number of live neighbours is either 2 or 3. This condition on the number of neighbours has to be tested once for every cell for every evolution step, which in practice is quite a lot of times. Using an if statement to check the condition would then be of harm to the performance of the code, so the check for this condition can be implemented as

```
grid[irow + j] = (nneigh == 2) + (nneigh == 3);
```

This expression yields the desired result since `==` evaluates to 1 when the equality holds or to 0 otherwise and at most one of the two conditions can be true at the same time.

Note that this expression can be adapted to implement the standard evolution rules with a simple modification

```
grid[irow + j] = grid[irow+j] * (nneigh == 2) + (nneigh == 3);
```

Moreover, both evolution types share the fact that the grid has periodic boundary conditions. To implement such conditions, the playground is expanded by an additional two rows, which accomodate copies of the top and bottom rows. This way we can enhance spatial data locality, as when updating the first row we can count the neighbours inside the copy of the last row instead of accessing the actual last row, which most certainly could not fit in cache with the first row. Periodic conditions along the x axis are achieved by means of the modulo operator.

Note that all processes allocate two extra rows of memory, which are used to accomodate a copy of the top and bottom rows from neighbouring processes, which are needed for computing the number of live neighbours of the process' own top and bottom rows (Fig. 1.3).

We can now move on to analysing the two routines separately.

Ordered

In ordered evolution, each cell is updated immediately as the number of its live neighbours is determined and its status immediately influences the status of the neighbours. This evolution then is intrinsically serial, as every cell has to wait for the update of the previous.

In practice the evolution starts from the top left corner of the grid and moves along the rows, left to right, until it reaches the bottom right corner.

The serial nature of this evolution imposes that communications and evolution can not happen at separate times. For instance, if all processes were to communicate their boundaries to their

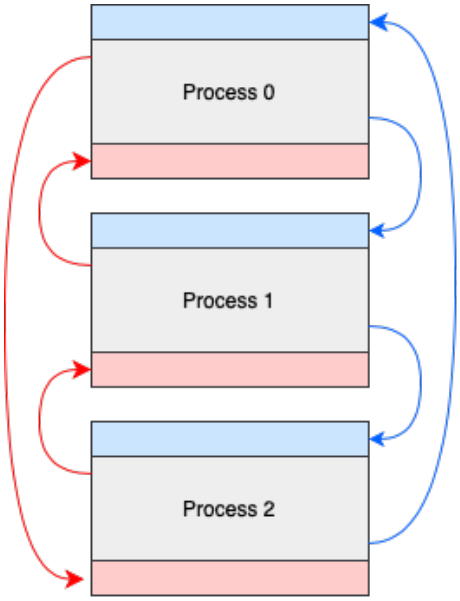

Figure 1.3: *Representation of the memory allocated per process. Gray parts represent actual grid data, while coloured parts represent the extra rows allocated to accomodate boundaries. Arrows indicate which rows from the actual grid are stored in the boundaries.*

neighbours before the start of evolution, after process 0 were finished updating its chunk, process 1 would start updating the first row based on the previously received, not updated, values in the last row of process 0. This is not what we want. Moreover, we want a way for processes to execute their work in order, from the first to the last.

Both of these problems can be solved by appropriately placing communications during the evolution process.

The general idea is that each process waits to receive the top boundary before beginning the evolution of its chunk and sends its bottom row to the next process as it is done updating it. At the same time, each process makes sure to communicate its top row to the preceding process before the latter starts evolving its bottom row.

More precisely, the communications happen as follows.

The first communication happens before the evolution loop begins and it is a nonblocking send from the last process to the first process, which contains the boundary required for the evolution to start. After this communication, each process but the root sends the top row to the preceding process, which uses it as the bottom boundary. The root does not have to send its top row to the last process just yet, as, if did so, the last process would get an outdated value for its bottom boundary. All these communications are nonblocking.

The evolution loop begins and each process posts a receive request for top and bottom boundary. After that, it waits for the receive top and the send top requests to complete. At the first iteration, the only process that receives the top boundary is the first one, so the evolution can start.

Each process may employ many threads to evolve its chunk of grid, so, to make sure that execution is indeed ordered, the parallel loop over the grid contains an `omp ordered` clause. Inside the loop are some instructions reserved to the first and last threads, which involve communications. Namely, the first thread sends the first row to the preceding process immediately after updating it. This send already has a matching receive posted earlier. The last thread, on the other hand, before starting its work needs to wait for the receive bottom request to complete. After the process is done updating its chunk of grid, the last thread sends the bottom row to the next process, which is waiting for its receive top request to complete. As the request completes, the evolution can continue as just described.

This communication pattern requires some extra care, since not all communications will complete. In fact, as processes finish the last evolution step, they exit the loop and they do not post receive requests. This means that there are going to be unmatched send requests, which we can

safely cancel at the end of the evolution, as they are not required to complete. This step is required to be able to correctly finalise the MPI region.

Static

In static evolution, the state of the grid is frozen at each iteration, the number of live neighbours is computed for all cells in the grid and the whole playground is updated before going to the next iteration.

Since the whole grid gets updated based on its values at the previous step, communications and evolution can happen at distinct times. The drawback of this method is that it requires the allocation of yet another grid, which is meant to store the number of live neighbours for each cell.

Communications between processes happen at the beginning of each evolution step and, to avoid deadlocks, the pattern is slightly different for even and odd rank processes.

For even rank the pattern is

1. Nonblocking receive top boundary
2. Nonblocking receive bottom boundary
3. Send top row
4. Send bottom row

whereas for odd rank processes the pattern is

1. Send top row
2. Send bottom row
3. Nonblocking receive top boundary
4. Nonblocking receive bottom boundary

A call to `MPI_Wait` ensures that each process has the correct boundaries before computing the number of neighbours.

The update of the grid requires two phases: computing the number of live neighbours and updating the values. Both these tasks are carried out in OMP parallel for regions, with static scheduling. Note that this kind of evolution requires twice as many accesses to the grid with respect to the ordered evolution.

1.4 Results

In this section we are going to discuss some results concerning the scalability of the code. Note that to obtain these results the code was compiled with the GNU C compiler, using the highest optimisation flag, the implementation of MPI was OpenMPI 4.1.5 and the machine was a cluster of nodes equipped with AMD Epyc 7H12 processors.

1.4.1 Ordered evolution

This kind of evolution is intrinsically serial, which means that there should be no benefit in using many threads and tasks. The need to have hybrid MPI code for this evolution arises when the memory required to store the playground is too large for a single node. In addition, having many threads may be beneficial for data locality when the NUMA struncture is complex.

For this evolution I did tests with increasing number of threads in one MPI task and with increasing number of MPI tasks saturated with OMP threads. Results are in Figure 1.4.

From these plots we can observe that increasing OMP threads with one single MPI tasks does only slightly increase the execution time, since it introduces some parallel overhead. On the other hand we clearly see that the same is not true when increasing the number of MPI tasks. In particular, for the results in the right panel I used a maximum of 32 MPI tasks, mapped on NUMA

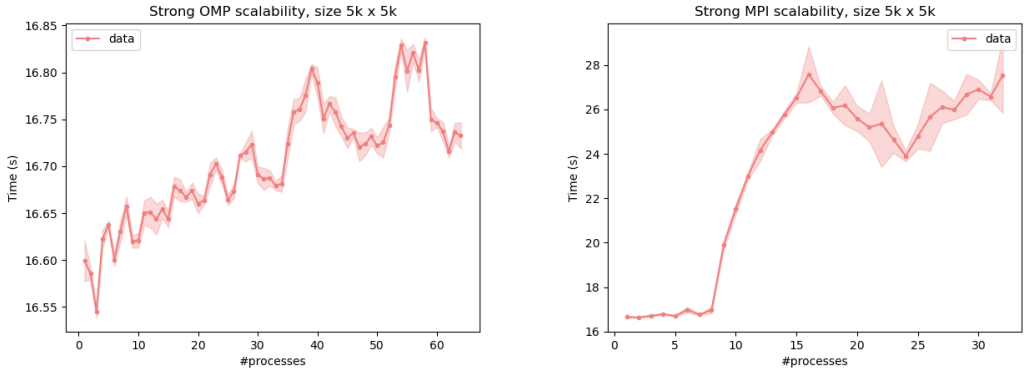

Figure 1.4: *Execution time for odered evolution with different number of threads and MPI tasks. Run on Epyc processors.*

regions across 4 nodes. We can see that for the first 8 tasks, the runtime was almost constant and similar to that in the left panel; as the number of tasks grows beyond 8, though, the runtime starts to grow as well util it somewhat stabilises once again around a higher value. This phenomenon is possibly due to the larger overhead in communications between tasks in different nodes. In particular, the time required to write on file becomes significantly larger and communications in general are slower since they need to go through the network.

1.4.2 Static evolution

For static evolution, increasing the number of processors is expected to yield significant improvements in performance, since the workload can be evenly split among processors and communications are not expected to be a big part of the code.

Strong OMP scalability

This test is aimed at investigating the scalability properties of this code as the number of OMP threads grows with fixed grid size and number of MPI tasks.

Figure 1.5 shows some results for different numbers of MPI tasks.

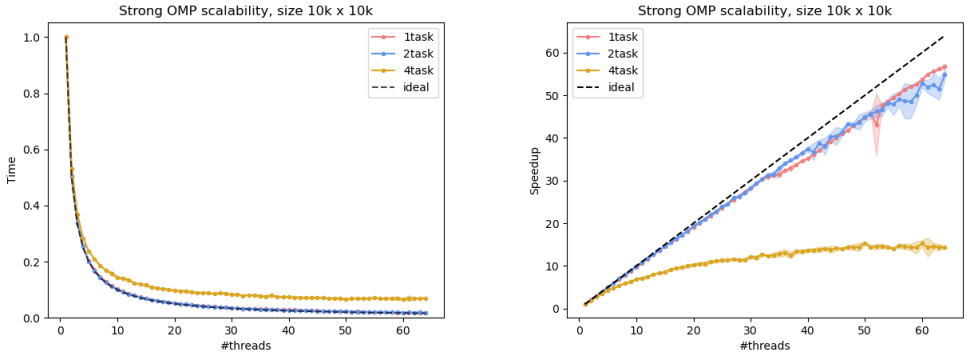

Figure 1.5: *Time, speedup and efficiency for OMP strong scalability with* $1, 2$ *and* 4 *MPI tasks, mapped by socket. Grid size was fixed to* 10000×10000.

As we can see, the scalability with 1 and 2 tasks is quite similar, whereas with 4 tasks it is drastically worse. To be precise, for the former two cases we have a speedup which is very close to the ideal and efficiencies well above 80%, while in the latter, the speedup stays below 15.

More detailed measurements show that the bottleneck is in writing the final state of the grid on a file, and that this operation gets more expensive as the number of MPI tasks increases. Indeed,

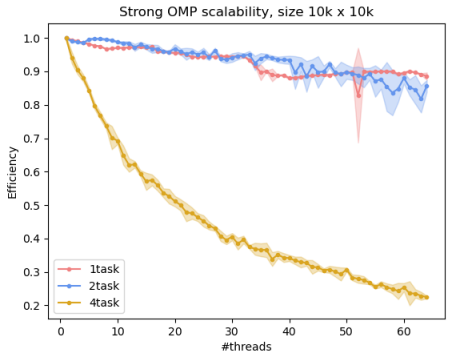

Figure 1.5: *Time, speedup and efficiency for OMP strong scalability with* $1, 2$ *and* 4 *MPI tasks, mapped by socket. Grid size was fixed to* 10000×10000.

since the master thread is the only one responsible for writing, the time required for this operation is almost constant as the number of threads increases, as shown in Fig. 1.6. What is also clear

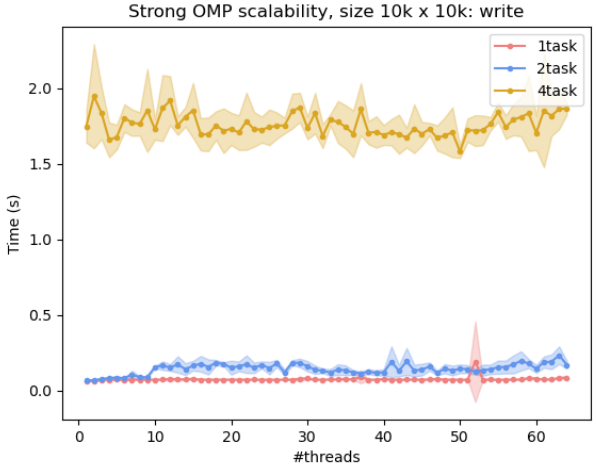

Figure 1.6: *Comparison of time required to write a checkpoint for different number of OMP threads and MPI tasks.*

from this image is that the time for 4 MPI tasks is much larger and variable than for 1 and 2 tasks. The reason behind this phenomenon is that increasing the number of tasks introduces the need of communicating to write on the file, and the communication introduces an overhead. This overhead is almost not noticeable for 2 tasks, as they are mapped on two sockets of the same node. On the other hand, with 4 tasks we are using 4 sockets spread across 2 nodes, hence the communications are not only between a larger number of tasks, but also between tasks which are far away and need to communicate via the network.

The fact that internode communications are slower than intranode communications adds to the fact that when using 4 tasks we are also effectively using 4 times as many cores, which means that the update of the grid will be faster. The overall effect is that for 4 tasks the parallel fraction of the code becomes significantly larger and therefore limits the speedup in a noticeable way.

To get a better understanding of the benefits of running multiple threads we can see the results obtained by only considering the time required to access the grid and update the cells.

Figure 1.7 shows that the time to work on the grid does in fact scale very nicely, to the point where the scaling is almost perfect when using 2 tasks. From the plots we can see that the biggest improvement is with 4 tasks, which is a confirmation of the fact the bottleneck is writing on file.

Moreover, since the time required to write does not depend on the number of threads nor on

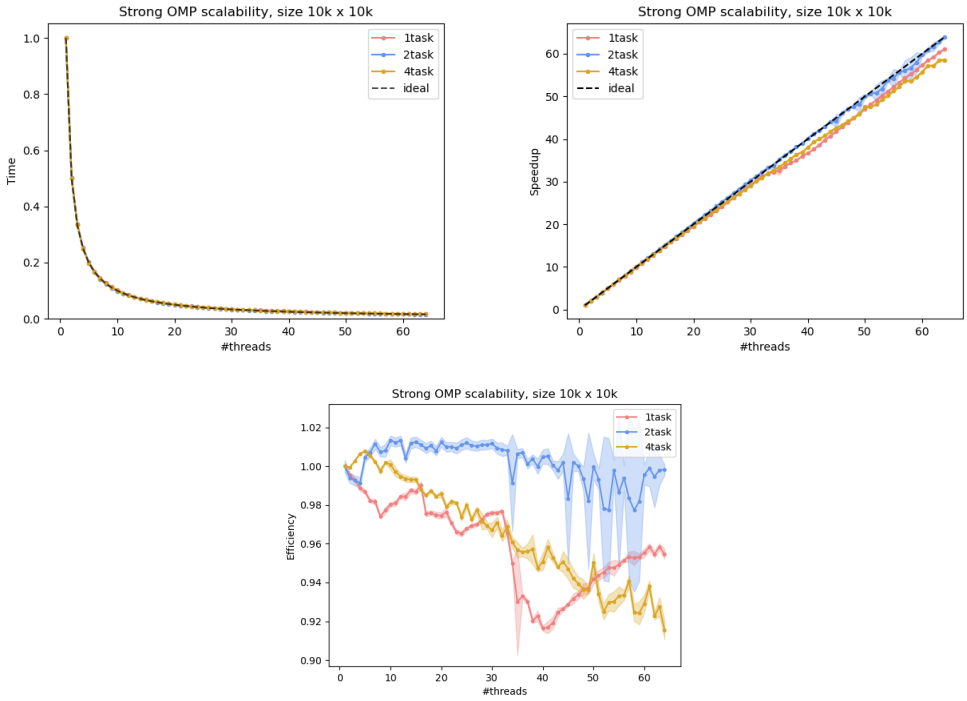

Figure 1.7: *Time, speedup and efficiency for OMP strong scalability with* $1, 2$ *and* 4 *MPI tasks, mapped by socket. The time refers to that needed to update the grid, hence excluding the time required to write on file.*

the number of evolution steps, but only on the size of the matrix, its effect on the performance will be mitigated by having the grid to evolve for a long time.

Strong MPI scalability

This test is meant to investigate the scalability properties for an increasing number of MPI tasks. The choice of bind for MPI tasks can be critical to the performance of an application, which is what we can see from this test.

We can consider two cases, with two different mapping policies: by numa and by socket. In the first case, each task uses 16 threads, while in the second case each task uses 64 threads. In both cases we fix a grid size and vary the number of tasks. Results are in Fig 1.8

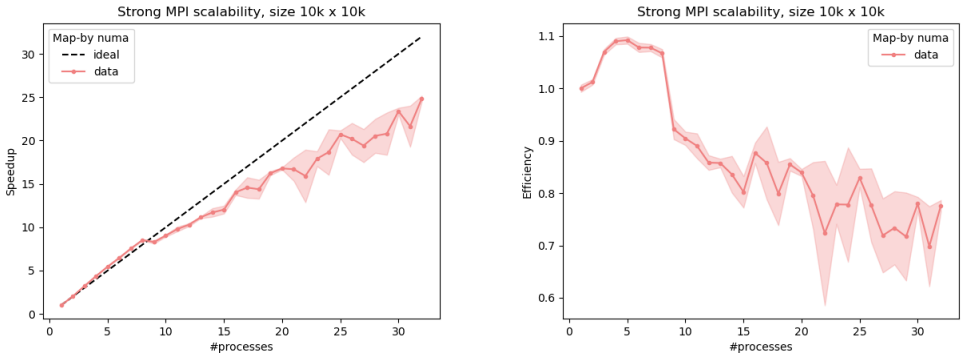

Figure 1.8: *Speedup and efficiency for MPI strong scalability with different task mapping. Top: numa, bottom: socket.*

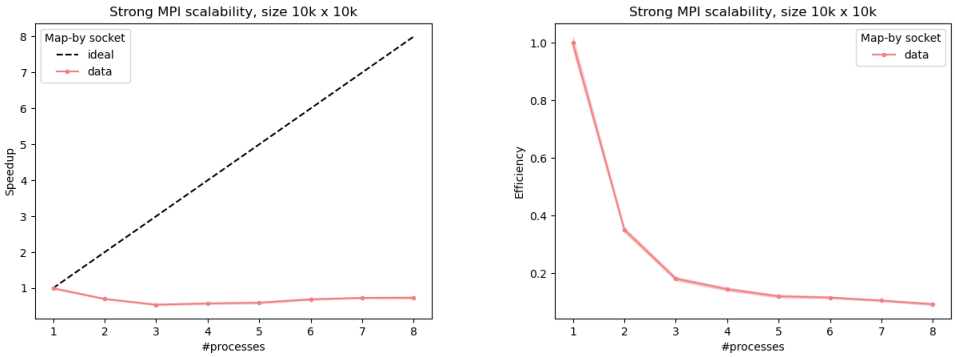

Figure 1.8: *Speedup and efficiency for MPI strong scalability with different task mapping. Top: numa, bottom: socket.*

From these plots it is clear that mapping MPI tasks by socket does not produce the expected effect. Indeed, the speedup immediately goes below 1, indicating that using more tasks leads to an increase in time, and the efficiency drops accordingly. Since these results were quite unexpected, the measurements were repeated with the same parameters and the results were the same. The reason behind this behaviour is not clear, since running the code with `mpirun --report-bindings` shows that the tasks are in fact mapped to the different sockets and use all the requested cores. I was not able to find an explaination for this phenomenon.

Mapping tasks by numa regions, on the other hand, seems to be quite beneficial, as we can see from the top row in Figure 1.8. In particular, the first plot shows that the speedup increases with the number of tasks, without approaching some upper limit, and the efficiency stays above roughly 70%.

A closer inspection shows that when using up to 8 tasks, the speedup is superlinear, as can be also noticed by efficiencies larger than 100%. This phenomenon is probably due to the fact the first 8 tasks are mapped onto the same node, and since the mapping is by numa region, data locality is possibly quite high, which speeds up the access to memory.

As the number of tasks grows beyond 8, we can clearly observe a drop in efficiency and a slower growth in speedup. The reason for this slowing is probably due to the overhead introduced by internode communications. In particular, Figure 1.9 shows that the time to communicate boundaries between processes remains almost constant, whereas the time required to write a checkpoint inceases significantly after growing tasks outside of a single node. Moreover, from this plot we

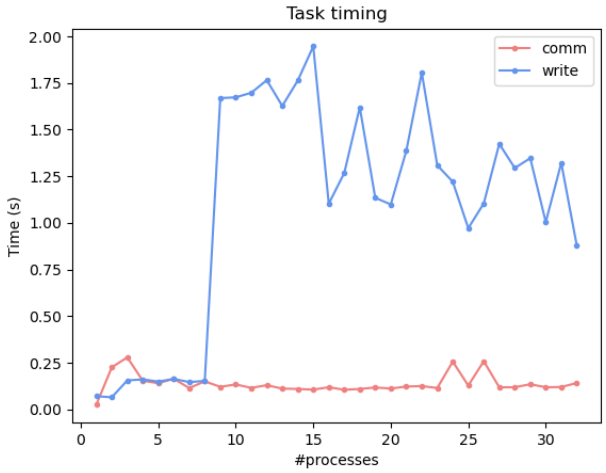

Figure 1.9: *Time required by boundary communications and checkpoint writing as a function of the number of MPI tasks (mapped by numa).*

can see that the time to write is subject to large oscillations, which are likely due to the network performance at a given time.

Weak MPI scalability

As we increase the problem size and number of tasks, so as to keep the workload per task fixed, we would expect the runtime to remain constant.

The weak MPI scalability test was performed by varying the number of MPI tsks from 1 to 32, mapping them by numa regions and saturating them with OMP threads. The workload per task was fixed to 10000×1000. Figure 1.10 shows the time measurements.

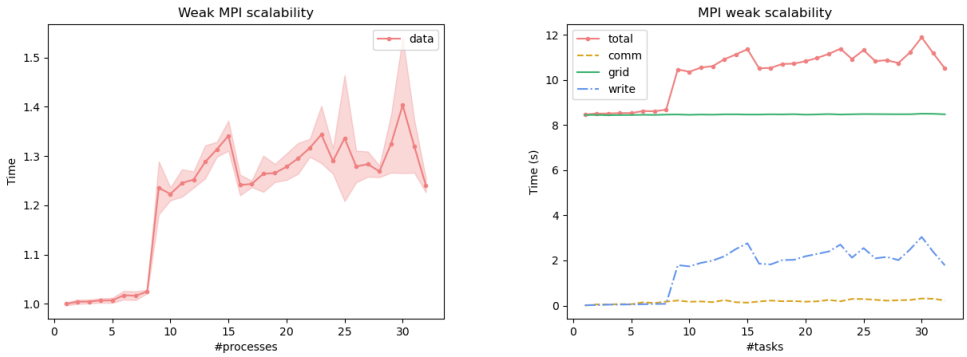

Figure 1.10: *Left panel: total time required. Right panel: time required for different tasks.*

The left panel shows that the total time does not quite remain constant as the number of tasks increases. Moreover, from the plot we can identify two regions; for $\#processes \leq 8$, the increase in time is slow and steady, while for $\#processes > 8$ the runtime is immediately larger and more variable.

The first region is the one where all the tasks belong to the same node and hence communications are quick. In the second region, the tasks start to be placed on two or more different nodes and consequently communications need to happen across the network, hence are slower. In both regions introducing more tasks adds some overhead in communications, which is especially visible in collective MPI calls, such that used to write on a shared file.

In particular, from the right panel we can see that the time required to update the grid and communicate boundaries stays almost constant, while the time required to write to file increases significantly when the number of tasks varies from 8 to 9. The good news is that despite some oscillations, the time required to write seems not to be increasing with the number of tasks. Moreover its effect on the performance could be mitigated by evolving the grid for a long enough time, keeping the number of write operations constant.

1.5 Conclusion

As we can see from the results reported above, the implemented code has good scalability properties, both with respect to the number of OMP threads and MPI tasks. Nonetheless there might be some modifications to be made in order to enhance performance.

The first possible modification could be in the write routine, since it seems that writing to file is the main bottleneck. Note that all the above results refer to runs where only the final state of the grid was saved as a checkpoint. In the case where a checkpoint has to be produced more often, the speed at which the writing happens may become even more important. To speed up this process, a possible solution would be to write to many different files, say one per process, so that to avoid using collective calls that introduce some overhead. The drawback of such an approach is that the user would end up with many files that need postprocessing. An alternative would instead be to initialise the MPI region in multiple mode, instead of serialised mode, so that many threads could write to the file at the same time. The drawback of this approach is that multiple mode is not

supported by all MPI implementations, hence the code would not be as portable. Moreover, since making more calls to a collective write routine would introduce more overhead, to see the benefits of this approach one would probably have to work with much larger files than those considered here.

Another possible modification would be to the access pattern of the matrices. As noted above, although a row-major access performs quite nicely, it is not necessarily the best access pattern in terms of cache utilisation. A different, strided, access pattern could be beneficial for the execution time, at the cost of possibly requiring a more complicated workload subdivision among tasks and threads.

Chapter 2

GEMM benchmark

2.1 Introduction

For this exercise the goal is to benchmark different linear algebra libraries, on different machines. In particular, the benchmark is aimed to measure the performance of level 3 BLAS functions from Intel's MKL, OpenBLAS and AMD's BLIS libraries. The performance is expressed in terms of execution time and GFlops, both for increasing size of the matrices and increasing number of threads, and the target machines were nodes equipped with AMD Epyc 7H12 and Intel Xeon Gold 6126 processors.

2.2 Methodology

The code for this benchmark was provided to us. The code simply initialises three matrices of given size and calls a GEMM function, labeled either sgemm or dgemm, depending on whether the arithmetics are in single or double precision.

The relevant portion of code is the call to the BLAS function, hence a timer measures the time required for the call to return. The time is then used to compute the Flops and the data are written on some output. The only modification I made was to remove some print statements in order to get a cleaner output.

The measurements were repeated several times for different OMP thread binding policies, to investigate how those affect the performance and to try to get the best berformance out of the machines.

2.3 Results

As anticipated, measurements consisted in a scalability test over the size of the matrices and a scalability test over the number of threads. I shall present the results separately.

2.3.1 Scalability over size

Increasing the size of the matrices means that the number of operations and the runtime grows, which in turn means that we can get a better idea of the performance of the funtions we want to benchmark.

In particular, the tests were run with square matrices, of sizes ranging from 2000×2000 to $20'000 \times 20'0000$ in steps of 1000. The number of threads was fixed to 12 for Intel machines and to 64 for AMD machines and the measurements were repeated with different thread allocation policies.

As we shall see, changing allocation policies affects the results to some extent and can be beneficial in trying to achieve the best performance for a specific machine. In our case, a single AMD processor should theoretically be capable of 2.664 $TFlops$, while a single Intel processor should be capable of 1.997 $TFlops$.

AMD Epyc

From plots in Figure 2.1 we can see that the three libraries perform differently from one another and that the performance is indeed affected by the thread allocation policy.

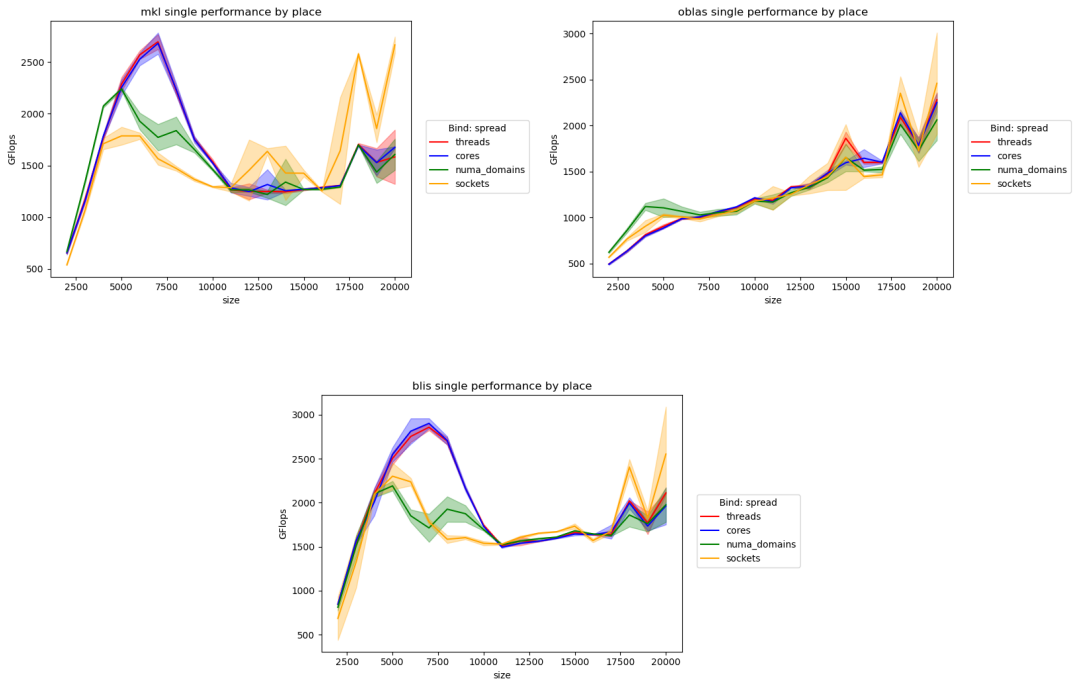

Figure 2.1: *GFlops as a function of matrix size for different OMP places on Epyc processors. Expected peak performance is around* 2.664 *TFlops.*

In particular, the performance of MKL and BLIS is qualitatively similar, although better for BLIS. In both cases we observe a peak in performance for matrix size of $\approx 7000 \times 7000$, obtained by mapping the OMP threads to cores or hardware threads (note that these are the same as multithreading is not active on the processors). As the size increases further, performance suddenly drops until the size reaches $\approx 20000 \times 20000$, where mapping by socket seems to be the better choice.

The profile for OpenBLAS, on the other hand, looks much different, with an almost monotonic increase in performance as the size grows. Moreover, different choices of OMP places do not seem to affect the results when `OMP_PROC_BIND=spread`. This is no longer true when we bind threads close together, in which case, while the performance for MKL and BLIS remains qualitatively the same, for OpenBLAS we can observe a clear improvement in performance when mapping processes to cores/hwthreads (Fig. 2.2).

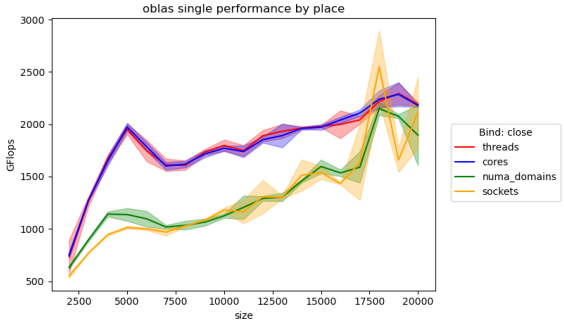

Figure 2.2: *Performance of OpenBLAS library as a function of matrix size, for* `OMP_PROC_BIND=close`*. A comparison with the previous picture shows an improvement in performance.*

As far as choosing different OMP places goes, note that the standard allows for another choice, `ll_cache`, which was deliberately omitted. This option binds threads to physical cores that share the last level cache, L3 in this case. With this choice of place, all the libraries performed very poorly regardless of the bind policy, reaching peak performance of ≈ 300 *GFlops*. Such poor performance makes it not worth to even compare this case to the other presented above.

Having seen how different places affect the performance of libraries, we can now directly compare the libraries when using the same parameters. Figure 2.3 shows the comparison.

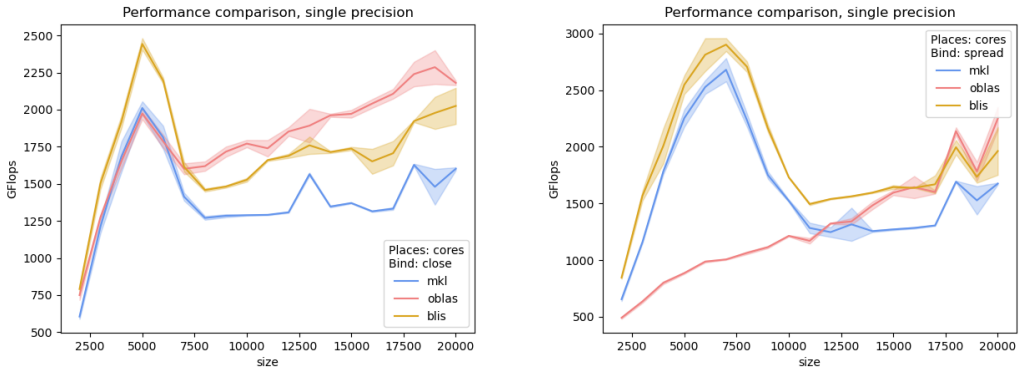

Figure 2.3: *Performance comparison between different libraries for single precision arithmetics, with different thread binding policies.*

From these plots we can see that the best overall performance for single percision is achieved by BLIS on relatively small matrices, with size $\approx 7000 \times 7000$. For larger sizes, though, both BLIS and MKL exhibit a rapid drop in performance, for both binding policies. However, in the left panel we can see that OpenBLAS, with close binding, is able to sustain a better performance with respect to the other two, which also seems to be the best performance for this range of matrix size.

For double precision arithmetics we can draw similar conclusions, in that mapping threads by core generally yields the best performance, binding threads spread apart is better for MKL and BLIS and the performance of OpenBLAS increases almost monotonically (Fig. 2.4).

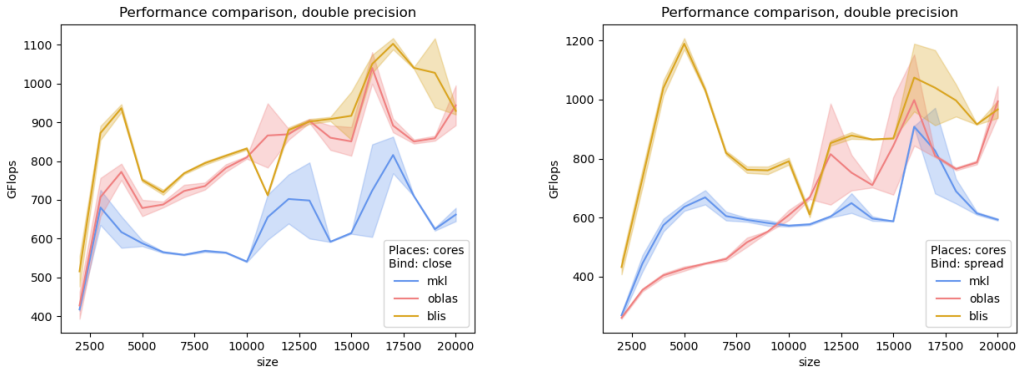

Figure 2.4: *Performance comparison between different libraries for double precision arithmetics, with different thread binding policies.*

From this comparison we can observe that the best overall performance is achieved by BLIS, followed by OpenBLAS with close binding.

Intel Xeon Gold

On Intel machines the results are quite different.

Let's start once again by looking at how the choice of place affects the performance (Fig. 2.5).

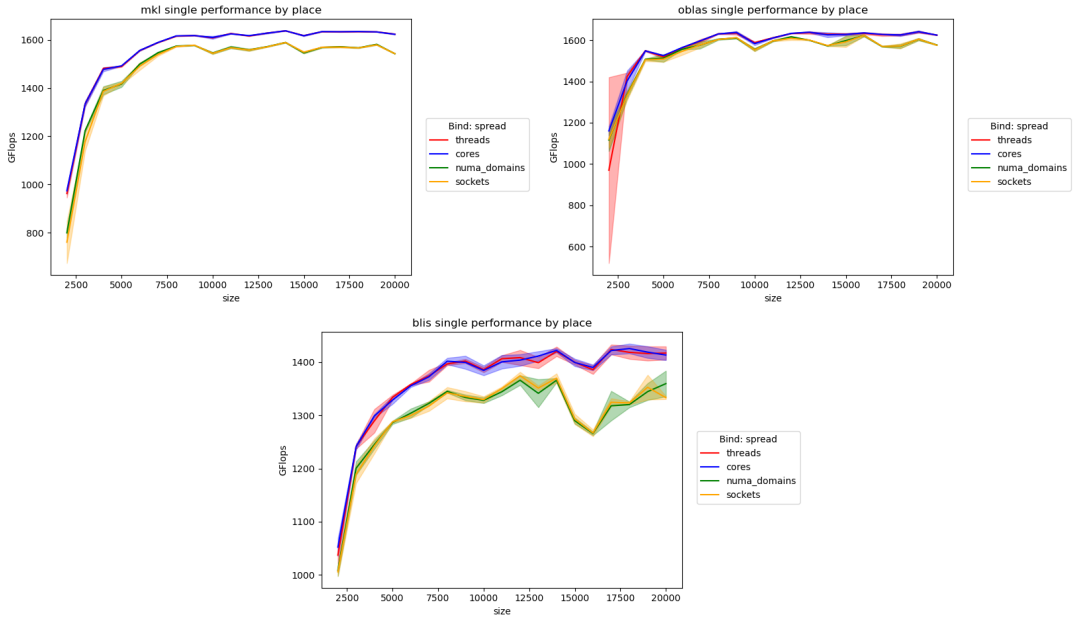

Figure 2.5: *GFlops as a function of matrix size for different OMP places on Intel processors. Expected peak performance is around* 1.997 *TFlops.*

From the plots we can immediately see that the performance on Intel processors is qualitatively different from the performance on AMD processors. Most notably, the performance in this case grows almost monotonically util it saturates to some value which is slighlty different from case to case.

Another observation is that the simpler NUMA structure of the processors is reflected by the fact that mapping to numa is equivalent to mapping to socket, which is not the case for Epyc processors. Moreover, in this case we can see a clear gap between the mapping to cores (or hwthreads) and numa regions (or sockets), which again, was not so clear on Epyc processors.

Note in addition that in this case the choice of `OMP_PROC_BIND` does not affect the results qualitatively, although close binding causes a slightly worse performance.

From these plots it is also quite easy to see that the best performance is achieved by MKL and OpenBLAS, although BLIS was specifically compiled for this architecture. A more clear view is given in Figure 2.6.

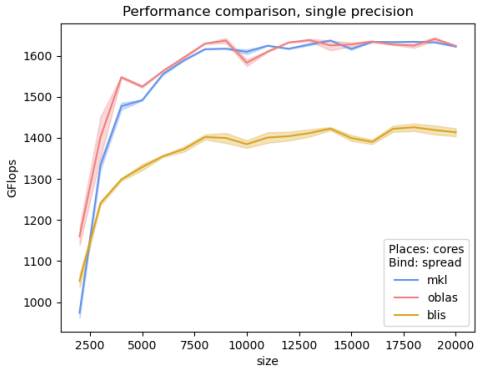

Figure 2.6: *Comparison of performance in terms of GFlops vs matrix size for the three libraries on Intel Xeon Gold.*

The same considerations can be made for double precision arithmetics, with the exception that the best performance is almost halved and the time is almost doubled, as is to be expected.

2.3.2 Further considerations on scalability over size

As is clear from the plots above, although the maximum performance achieved is close to the maximum expected, there is still room for improvement. The biggest concern is the sudden drop in performance on Epyc processors for sizes that exceed ≈ 7000; indeed, although on Intel processors the performance is not quite optimal, it is at least sustained over increasing matrix sizes.

I suspect that this phenomenon is due to the NUMA structure of Epyc processors. As a matter of fact, in the code provided to perform the benchmarks, all the memory allocated to store the matrices is initialised by a single thread. The result of this operation is that when using many threads to perform calculations, the vast majority of those will not be close to the data they access, which can be a serious bottleneck for performance. Indeed, by modifying the code and using a parallel for to initialise the data, we can observe a significant improvement in performance (Fig. 2.7).

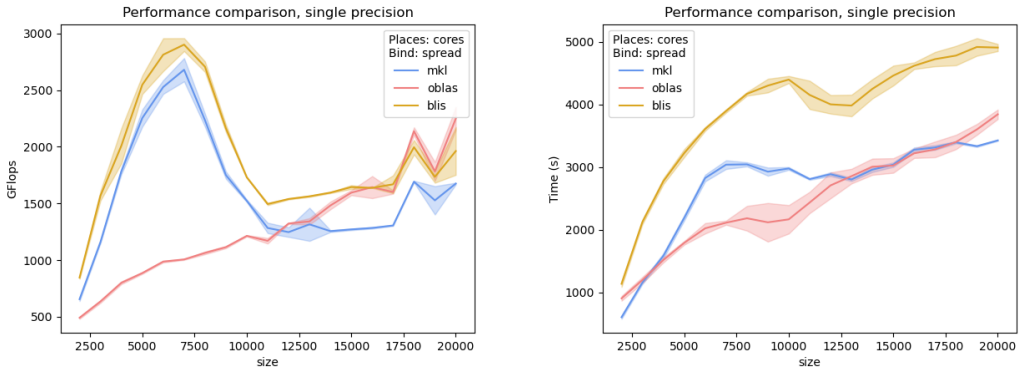

Figure 2.7: *Performance in GFlops as function of matrix size on Epyc processors. Non-NUMA-aware allocation on the left; NUMA-aware allocation on the right.*

By comparing these plots we can see the big improvement gained by NUMA-aware memory allocation, not only in terms of peak performance but also in terms of consistency.

This result is not surprising in terms of expecting an improvement after the proper allocation, but it is quite surprising in terms of the performance reached. As mentioned before, for Epyc processors the expected peak performance was about 2.7 $TFlops$, which is significantly less than what measured here.

2.3.3 Scalability over number of threads

This scalability test was performed by fixing a size of 10000×10000 and progressively increasing the number of threads. The scalability properties of the libraries can be evaluated in terms of execution time, speedup and efficiency.

AMD Epyc

Figure 2.8 shows the measurements of time, speedup and efficiency on Epyc processors.

From the first plot it is immediately clear that none of the libraries scales perfectly, since there is a visible gap between data and the ideal time. The poor scalability becomes even more clear when looking at the speedup in the second plot. From this we can see that the three libraries have similar speedup, with MKL and OpenBLAS being a bit more consistent than BLIS. All the experimental curves seem to approach a value of about 15 or 20 as the number of threads grows larger. This may suggest that there is a significant parallel overhead. The poor speedup immediately reflects on the poor efficiency, which goes as slow as $20 - 30\%$. In fact, although for a number of threads smaller

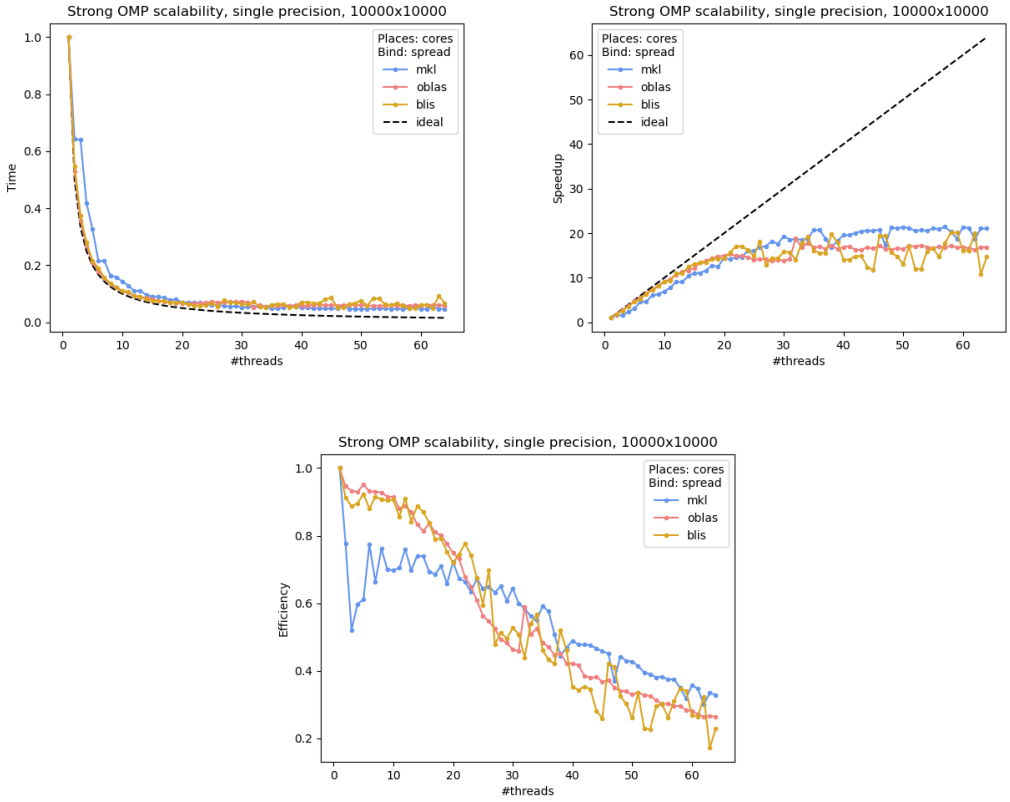

Figure 2.8: *Time (normalised), speedup and efficiency of the different libraries on Epyc processors. Single precision arithmetics.*

than about 20, OpenBLAS and BLIS have a good efficiency, larger than MKL, as the number of threads grows beyond 20, all the efficiencies rapidly decrease.

For double precision arithmetics, analogous considerations hold, even though the data seems much more regular (Fig. 2.9).

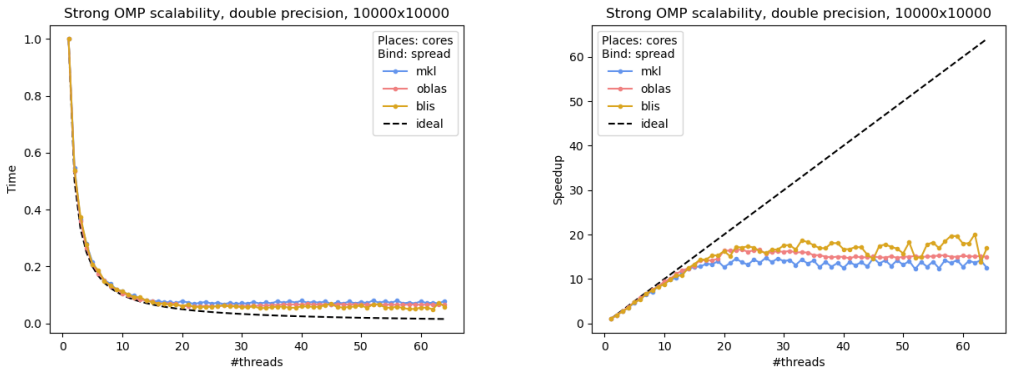

Figure 2.9: *Time (normalised), speedup and efficiency of the different libraries on Epyc processors. Double precision arithmetics.*

Again, the first plot clearly shows that the scaling is not ideal, which is even more clear from the second plot. In this case, BLIS has the best speedup and MKL the worst. Efficiency also rapidly drops, after a very small region where it remains at around 90%.

These very unsatisfactory results can be improved by accounting for the NUMA structure when allocating memory (Fig. 2.10).

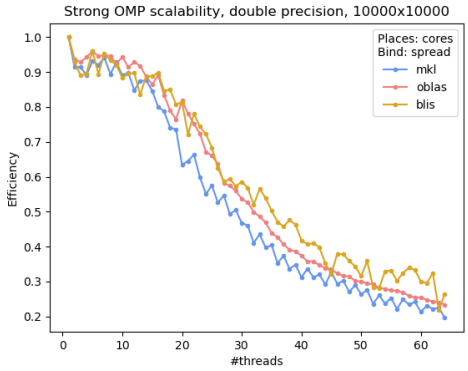

Figure 2.9: *Time (normalised), speedup and efficiency of the different libraries on Epyc processors. Double precision arithmetics.*

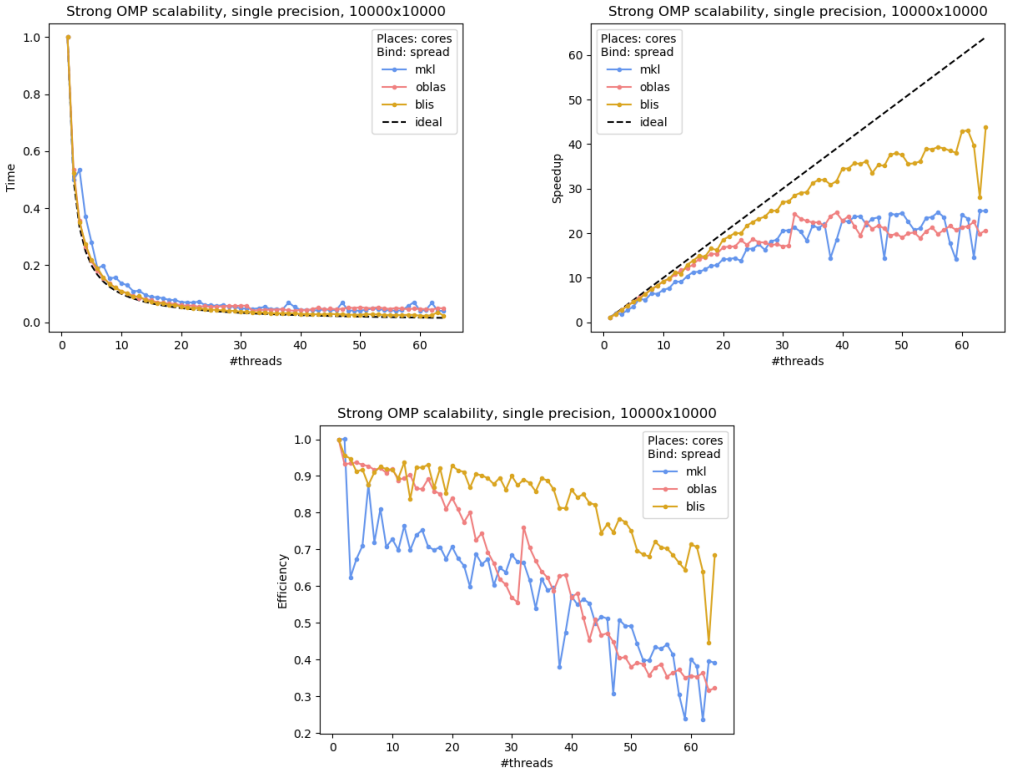

Figure 2.10: *Time (normalised), speedup and efficiency of the different libraries on Epyc processors, with NUMA-aware memory initialisation. Single precision arithmetics.*

The plots show that the scalability properties of all the libraries improve when properly initialising the memory, the biggest effects being on BLIS. In particular, the speedup seems to approach ≈ 20 for MKL and OpenBLAS and ≈ 40 for BLIS. This implies that efficiency stays above $\approx 30\%$ for MKL and OpenBLAS and $\approx 60\%$ for BLIS.

Although these are significant improvements with respect to the previous results, the scalability is still far from perfect.

Intel Xeon Gold

Scalability on Xeon processors is much better than on Epyc processors, as we can see from Figure 2.11.

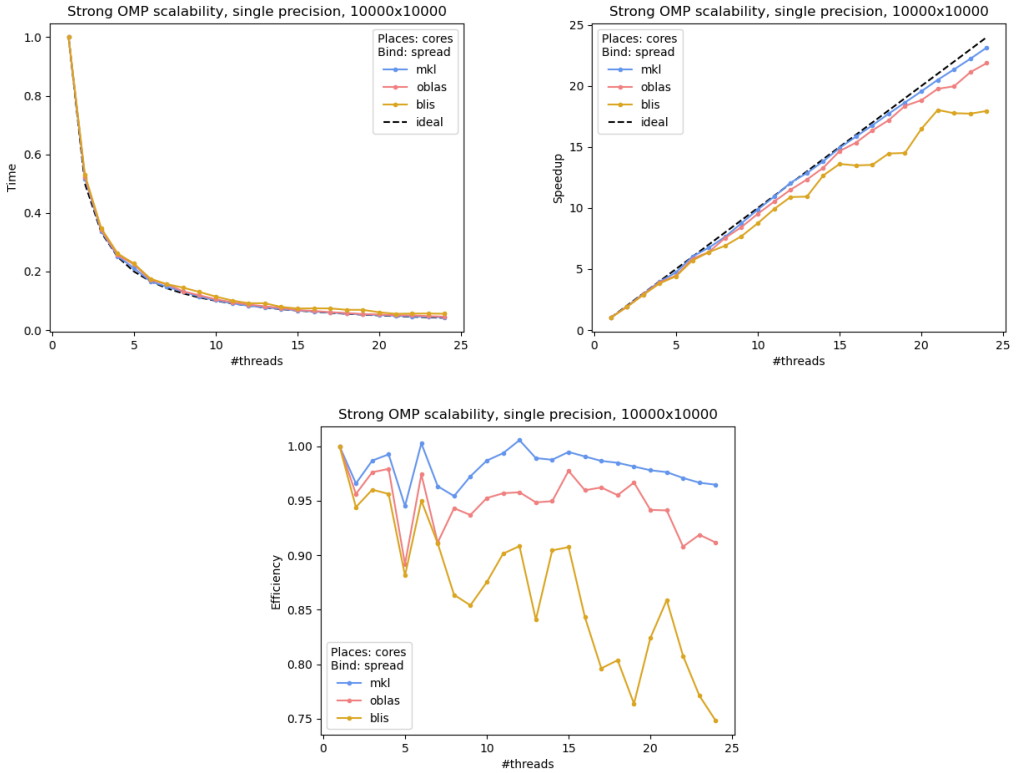

Figure 2.11: *Time (normalised), speedup and efficiency of the different libraries on Xeon Gold processors. Single precision arithmetics.*

The better scalability properties with respect to Epyc processors are immediately visible from the first plot, since in this case there is no clear gap between data and the ideal time. By looking at the speedup plot we can clearly see that the scalability is indeed almost perfect for MKL and OpenBLAS and, even though not quite as good, still much better with respect to Epyc also for BLIS.

By analising the speedup we can make two important observations. The first one is that writing and compiling code for a specific architecture can make a big difference. This can be seen by the fact that MKL, developed by Intel, goes from being the worst performing library on Epyc to the best performing, with almost perfect scaling properties. The second is that the limiting factor for speedup on Epyc processors is very likely due to some parallelisation overhead and the complex NUMA structure, and not to serial, non-parallelisable fractions of code.

Efficiency is this case is very good, with MKL staying above 95% and BLIS, the worst performing, staying above 75%.

For double precision arithmetics the results are the same.

Bibliography

[1] *Conway's Game of Life*. URL: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

[2] *ConwayLife*. URL: https://conwaylife.com.