University of Trieste
586SM - Foundations of High Performance Computing

Assignment Report

A.Y. 2022/2023

Tommaso Fonda - IN2000156

September 22, 2023

Contents

1 Parallel programming

1.1 Introduction

A parallel implementation of a variant of Conway's Game of Life has been developed. The environment the game takes place in is a **square**, bidimensional grid with periodic boundary conditions. The program supports two kinds of evolution strategies, i.e. *static* and *ordered*, which behave as per the specification document. The grid's initial conditions and periodic snapshots are all persisted on disk as PGM files, according to the PGM Format Specification.

1.2 Implementation

The program is hybrid, i.e. parallelisation has been achieved by using both MPI and OpenMP. The main idea is that the program will be run with few MPI processes (e.g. one per socket), which will then fill the socket by spawning OpenMP threads. Of course, the program is nonetheless able to run in "extreme" setups, such as completely serial (one MPI process and one thread) or suppressing the threads' parallelism (i.e. n MPI processes and only one thread per process). The C++ programming language has been chosen to implement the program. This choice supports the usage of the Boost.MPI library to handle most of the MPI calls. C++ and Boost.MPI were chosen to minimize (actually, to avoid at all) the usage of pointers and manual memory allocation (i.e. explicit calls to `malloc()` and similar `libc` functions, and/or `libstdc++` equivalents), thus making the code more safe and less prone to memory errors, leaks or corruption. This goal has been achieved, since in the whole code base, not a single pointer is used. Argparse was used to parse the command line arguments.

To build the program, GNU make has been employed. In order to write correct, safe code, particular attention has been devoted to the choice of compiler warnings to enable via make's `CPPFLAGS` variable. All warnings suggested by Jason Turner in his book *Learning C++ Best Practices* have thus been enabled.

Development was initially carried out locally on a dual-socket Dell T5600 workstation, featuring two Intel Xeon E5-2680 CPUs and 64 GB of RAM, in a Distrobox container running Fedora 36, to emulate the software environment found on the Orfeo cluster. Once the initial implementation was finished, further testing and development has been conducted on Orfeo. Git has been used as the VCS.

1.2.1 Building Boost.MPI on Orfeo

Since the Orfeo cluster does not offer the Boost.MPI and Boost.Serialization libraries, it has been necessary to build them from source. The task has been carried out following the official tutorial, adapting it when needed, since it contains some typos and mistakes.

1.2.2 Data structures and layout

The cells are held in memory as `unsigned chars`. This is not the most efficient approach, since a binary attirbute such as "dead"/"alive" could well be represented by a single bit, but it is the most convenient, as it provides a direct mapping between the bytes representing each pixel in the PGM images and the cells' representation in memory. Cells are collected into a `std::vector`.

The whole grid is split among the MPI processes in blocks of adjacent rows. The blocks are balanced as much as possible. That is, if working with 4 MPI processes on a 15x15 grid, ranks 0-2 will get 4 rows, while the last process will only get 3. This number is called `rank_rows`. Each process allocates an unidimensional `std::vector` capable of holding `(rank_rows + 2)` rows, in order to accomodate for the halo row it will receive from both its "neighboring" processes, that are the processes working on the rows immediately before and after its own.

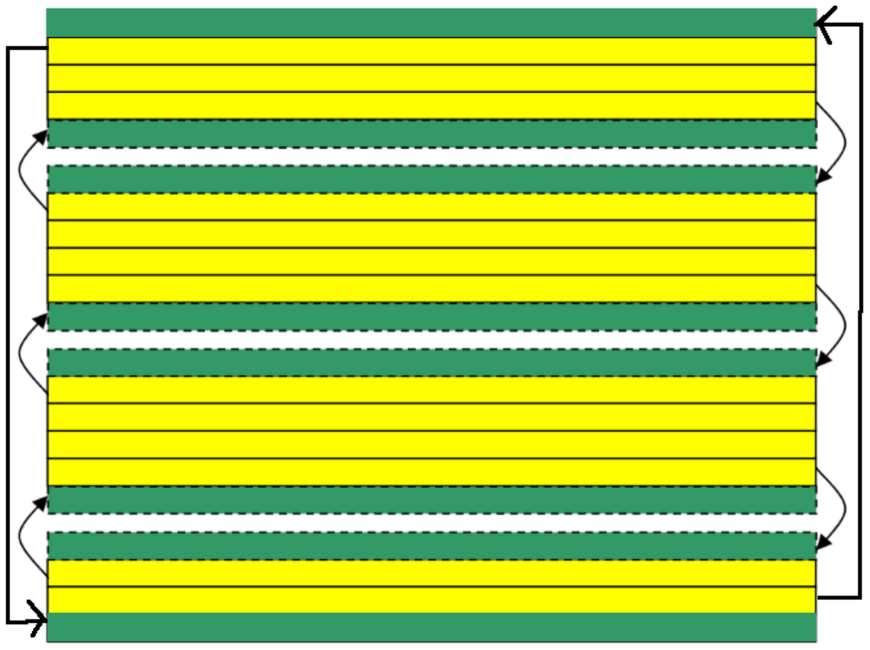


Figure 1: Grid decomposition with periodic boundary conditions.

1.2.3 Choice of the MPI routines to use

Inter-process exchange of data has been implemented by simple point-to-point send and receive calls. To improve the readability of the code, four macros have been defined: `SEND_{FIRST,LAST}_ROW` and `RECEIVE_{TOP,BOTTOM}_HALO` (check the code snipped below for their definition). Each macro's name reflects the operation carried out from the perspective of the rank that executes it. As an example, supposing the program is running with 4 MPI processes, if rank 0 calls `SEND_FIRST_ROW`, it means that it sends the first row of the chunk it is working on to rank 3 (the preceding rank). Rank 3 will have to receive this data by calling `RECEIVE_BOTTOM_HALO`, which indicates that it will place the data received in the last row of its buffer (which is dedicated to the bottom halo), since those data represent, indeed, the first row immediately below the chunk rank 3 is working on.

```
#define SEND_LAST_ROW \
        world.send(next_rank, LAST_ROW_OF_SENDING_RANK,
        ↪   rank_chunk.data() + rank_rows * grid_size,
        ↪   grid_size);
#define SEND_FIRST_ROW \
        world.send(prev_rank, FIRST_ROW_OF_SENDING_RANK,
        ↪   rank_chunk.data() + grid_size, grid_size);
#define RECEIVE_TOP_HALO \
        world.recv(prev_rank, LAST_ROW_OF_SENDING_RANK,
        ↪   rank_chunk.data(), grid_size);
#define RECEIVE_BOTTOM_HALO \
        world.recv(next_rank, FIRST_ROW_OF_SENDING_RANK,
        ↪   rank_chunk.data() + (rank_rows + 1) * grid_size,
        ↪   grid_size);
```

Parallel file I/O has been implemented by collective read/write operations, namely `MPI_File_{read,write}_at_all`. The classic OpenMPI C bindings have been used in this case, because the Boost.MPI library does not provide any wrapper offering the same functionality. This represents a Level 1 access pattern according to the paper *Optimizing Noncontiguous Accesses in MPI-IO* (by Rajeev Thakur, William Gropp, Ewing Lusk). Since all ranks need to read/write one (large) chunk of contiguous data from the file, and not noncontiguous portions scattered here and there (remember that each rank works on a subset of adjacent rows of the grid), usage of these collective operations is enough to achieve optimal performance (according to page 4 of the aforementioned paper, if only contiguous I/O is performed, Level 1 and Level 3 access patterns are equivalent).

Here is an example to illustrate the parallel file access pattern, in particular, the `PgmUtils::write_chunk_from_file()` function is shown. The

`PgmUtils::read_chunk_from_file()` function, responsible for parallel reading of PGM files, is logically structured the same way. Both when reading and when writing, rank 0 is responsible of handling the header (not shown here, header handling is implemented outside these two functions).

```
void PgmUtils::write_chunk_to_file(const std::string& filename,
 ↪  const PGM_HOLDER& chunk, const std::streampos start_offset,
 ↪  const ulong leading_halo_length, MPI_Comm comm)
{
        MPI_File file;
        MPI_File_open(comm, filename.c_str(), MPI_MODE_CREATE |
         ↪  MPI_MODE_WRONLY, MPI_INFO_NULL, &file);
        std::size_t size = chunk.size() - 2 *
         ↪  leading_halo_length;
        MPI_Offset offset =
         ↪  static_cast<MPI_Offset>(start_offset);
        MPI_File_write_at_all(file, offset, chunk.data() +
         ↪  leading_halo_length, size, MPI_CHAR,
         ↪  MPI_STATUS_IGNORE);
        MPI_File_close(&file);
}
```

1.2.4 OpenMP parallelisation strategy (static evolution)

The static evolution strategy allows for parallelisation using OpenMP threads.

The code that performs static evolution is located in the `evolve_static()` function. The section to be parallelised is of course the `for` loop that updates each cell of the current iteration's chunk, based on the status of the cells in the previous iteration's chunk. Initially, a naive parallelisation was implemented by simply adding `#pragma omp parallel for` (plus the appropriate `shared` clause) before the aforementioned `for` loop.

```
#pragma omp parallel for shared(rank_chunk, next_step_chunk)
for (i = start; i < end; i++) {
        char alive_neighbors =
         ↪  count_alive_neighbors(rank_chunk, i);
        next_step_chunk[i] = (alive_neighbors == 3 ||
         ↪  alive_neighbors == 2) ? CELL_ALIVE : CELL_DEAD;
}
```

This trick alone yielded a significant speedup, but it did not look like the optimal solution, as the whole pool of threads is forked and joined in each step of the evolution. Forking and joining a thread is a costly operation, so it would be beneficial to only fork the threads once in each execution of the program, and then reuse the same threads in each evolution step of the simulation. This has been achieved by using OpenMP tasks instead of a `parallel for` loop: it is now possible to open a parallel region that wraps the evolution's `for` loop, and then only have the master thread perform all the code contained in it (including MPI calls). The tasks are then created by the master thread by means of a `taskloop` OpenMP construct: the outer loop in the snippet below is only for generating the tasks, while the inner loop represents each task's actual work.

```
#pragma omp taskloop shared(rank_chunk, next_step_chunk) grainsize(1)
        for (auto j = start; j < nthreads * elements_per_thread; j += elements_per_thread) {
                auto i = j;
                for (; i < j + elements_per_thread; i++) {
                        char alive_neighbors = count_alive_neighbors(rank_chunk, i);
                        next_step_chunk[i] = (alive_neighbors == 3 || alive_neighbors == 2) ? CELL_ALIVE : CELL_DEAD;
                }
        }
        // still need to take care of the remaining elements, this is why variable 'i' was declared before the inner for loop.
```

After all the simulations had been run, it was found out that most OpenMP implementations, in programs such as this one, would not cyclically fork and join the threads, if the "naive" parallel for loop solution was employed. Instead, the threads would be forked only once and still kept alive, even when inactive. In order to confirm this, part of the simulations have been re-run after reverting the commit that introduced the `taskloop` implementation. The observed performance was very similar, which confirms that the two parallelization strategies behave similarly. Still, the taskloop commit was reapplied, as it provides a guarantee that no matter which OpenMP implementation one is using, the threads will not be joined and re-forked continuously.

Finally, loop unrolling by a factor of 2 and of 4 were tested, and it was discovered that the execution time of the program diminished significantly when unrolling by a factor of 2, and even more when unrolling by a factor of 4. The final parallelisation strategy that was chosen is `taskloops` with unrolling by 4.

1.2.5 Ordered evolution

The ordered evolution strategy is serial by its very nature, so there is no way to parallelise it. It can only scale in the sense of memory usage, i.e. by using multiple MPI processes placed on different nodes, simulations on bigger grids can be performed.

Ordered evolution has been implemented starting always from the top-left corner of the grid. This corner always belongs to rank 0. Thus, for a single step of ordered evolution to run properly, three cases need be distinguished in the code. Assume N is the number of MPI ranks:

1. rank 0 must be the first one to start. It must first receive the two halo rows from ranks 1 and $N-1$, then perform its computation, and then send the first and last rows of its chunk (that at this point has been evolved) to ranks $N-1$ and 1 respectively;
2. ranks 1 to $N-2$ have to send their first row to the preceding rank, to allow that rank to perform their portion of work. Then they must receive both halo rows (the top one, coming from the preceding rank, has already been evolved, the bottom one has not, yet, because it comes from the following rank, which works on a lower portion of the grid). At this point they are ready to perform the evolution, after which they send their last row to the following rank, which needs it because it will soon start to perform its portion of work;
3. rank $N-1$ starts by sending its first row to the preceding rank, just as decribed above, but it also has to immediately send its last row to rank 0, to allow rank 0 to kick off the whole evolution iteration.

The implementation of ordered evolution follows.

```
inline __attribute__((always_inline)) void
↪  update_cell_ordered(PGM_HOLDER& rank_chunk, ulong j)
{
        char alive_neighbors =
        ↪  count_alive_neighbors(rank_chunk, j);
        rank_chunk[j] = (alive_neighbors == 3 ||
        ↪  alive_neighbors == 2) ? CELL_ALIVE : CELL_DEAD;
}

void evolve_ordered(PGM_HOLDER& rank_chunk, PGM_HOLDER& unused,
↪  mpi::communicator world)
```

```
{
        const ulong rank_rows = (rank_chunk.size() / grid_size)
        ↪  - 2;
        if (world.size() != 1) {
                if (world.rank() == 0) {
                        RECEIVE_TOP_HALO;
                        RECEIVE_BOTTOM_HALO;
                        for (auto j = grid_size; j < (rank_rows
                        ↪  + 1) * grid_size ; j++) {
                                update_cell_ordered(rank_chunk,
                                ↪  j);
                        }
                        SEND_FIRST_ROW;
                        SEND_LAST_ROW;
                } else if (world.rank() == world.size() - 1) {
                        SEND_LAST_ROW;
                        SEND_FIRST_ROW;
                        RECEIVE_BOTTOM_HALO;
                        RECEIVE_TOP_HALO;
                        for (auto j = grid_size; j < (rank_rows
                        ↪  + 1) * grid_size ; j++) {
                                update_cell_ordered(rank_chunk,
                                ↪  j);
                        }
                } else {
                        SEND_FIRST_ROW;
                        RECEIVE_BOTTOM_HALO;
                        RECEIVE_TOP_HALO;
                        for (auto j = grid_size; j < (rank_rows
                        ↪  + 1) * grid_size ; j++) {
                                update_cell_ordered(rank_chunk,
                                ↪  j);
                        }
                        SEND_LAST_ROW;
                }
        } else {
                for (auto j = grid_size; j < (rank_rows + 1) *
                ↪  grid_size ; j++) {
                        update_cell_ordered(rank_chunk, j);
                }
        }
}
```

1.2.6 Function inlining

In order to avoid code duplication and, at the same time, the performance cost of an excessive number jump instructions, many short helper functions have been defined as `inline __attribute__((always_inline))`, to not only suggest, but actually force the compiler to inline them.

1.2.7 Memory allocators comparison

The performance of three memory allocator configurations has been tested. These configurations are:

- vanilla, default C++ memory allocator;
- aligned allocator provided by the Boost.Align library;
- mimalloc, an allocator developed by Microsoft, known for its great performance.

In order to implement Boost's aligned allocator, the vectors' declarations must be changed from `std::vector<unsigned char>` to `std::vector< unsigned char, boost::alignment::aligned_allocator<unsigned char, 64> >`, where 64 is the cache line's size, to which the memory must be aligned.

In order to implement mimalloc, the vector's declaration must be `std::vector< unsigned char, mi_stl_allocator<unsigned char> >`, and appropriate header files must be included, as described in the official documentation.

Testing has shown that mimalloc and Boost's aligned allocator both perform similarly, and slightly better than the standard allocator. For reference, working on a 10k by 10k grid, evolving it for 100 steps, with two sockets and one MPI process per socket, and spawning 12 threads on each socket, the program's implementation using the standard allocator took roughly 136 seconds to execute, while the implementations which used Boost's aligned allocator and mimalloc both took roughly 130 seconds, on an Epyc node. Mimalloc was finally chosen as the allocator to use in this program.

1.2.8 Runtimes measurement

The Boost.MPI library offers a handy `boost::mpi::timer` class, which is basically a wrapper for `MPI_Wtime()`. To use it, it is sufficient to instantiate it - `mpi::timer timer;` - and then call its `elapsed()` method when desired. Here is how this class is used in this program.

```
mpi::timer timer;
PGM_HOLDER rank_chunk =
↪ PgmUtils::read_chunk_from_file(filename, rank_rows *
↪ grid_size, rank_file_offset_streampos, grid_size,
↪ static_cast<MPI_Comm>(world));
```

```
PGM_HOLDER next_step_chunk(rank_chunk.size());

#pragma omp parallel
{
#pragma omp master
{
nthreads = omp_get_num_threads();
for (uint i = 1; i <= simulation_steps; i++) {
    evolver(rank_chunk, next_step_chunk, world);
    if (evolution_type == 1)
            rank_chunk.swap(next_step_chunk);
    if (snapshotting_period) {
        if (i % snapshotting_period == 0) {
            save_snapshot(rank_chunk, i,
            ↪  rank_file_offset_streampos, world);
        }
    } else {
        if (i == simulation_steps) {
            save_snapshot(rank_chunk, i,
            ↪  rank_file_offset_streampos, world);
        }
    }
}
}
}
double elapsed = timer.elapsed();
double avg = mpi::all_reduce(world, elapsed,
↪  std::plus<double>());
avg = avg / world.size();
if (!world.rank()){
        std::cout << grid_size << "," << world.size() << "," <<
        ↪  nthreads << "," << avg << std::endl;
}
```

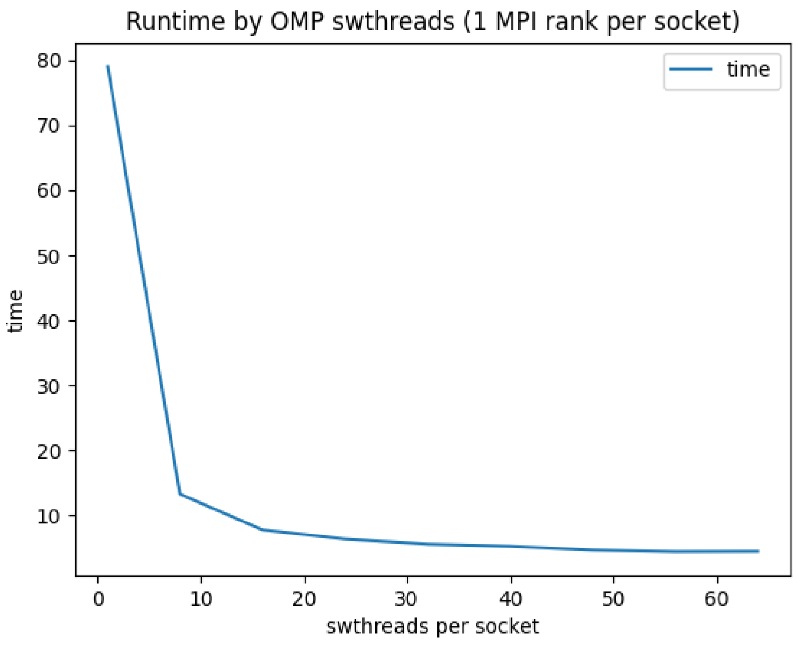
The timer thus wraps all parallel IO operations and, obviously, the program's main loop. At the end of the execution, rank 0 prints the grid's size, the MPI world communicator's size, the number of OpenMP threads and the ranks' average elapsed time to the standard output, as comma-separated values.

1.3 Results

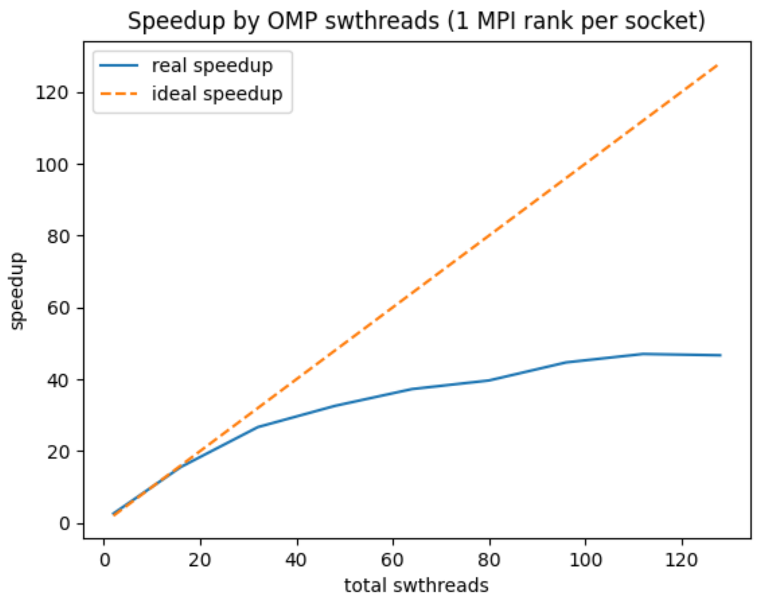
All tests have been performed repeating each simulation four times, and averaging the results.

1.3.1 OpenMP scalability

To perform this test, an Epyc node was requested, 2 MPI processes were spawned and mapped to the sockets, and an increasing number of OpenMP swthreads were spawned, binding them to their master's socket. The grid size used was 10k x 10k.



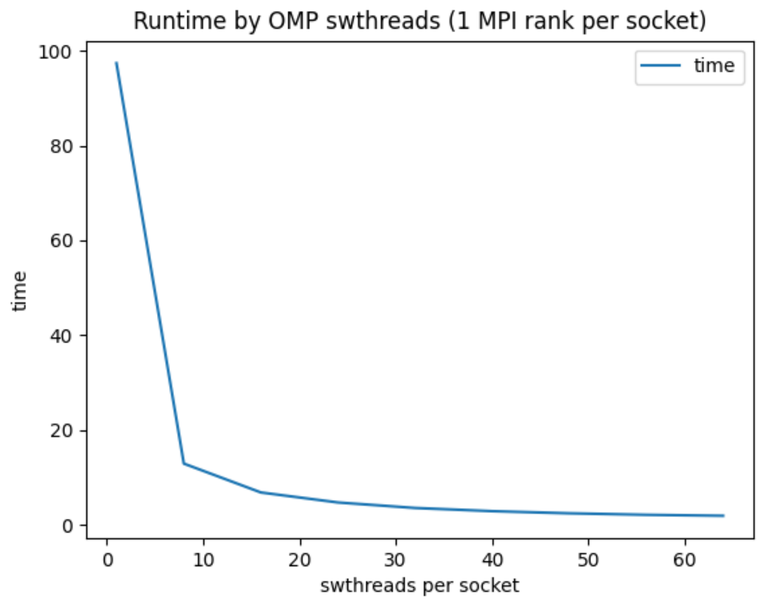
It looks like the runtime reduction was substantial as the number of threads increased from small values up to only 16. The associated speedup plot follows.



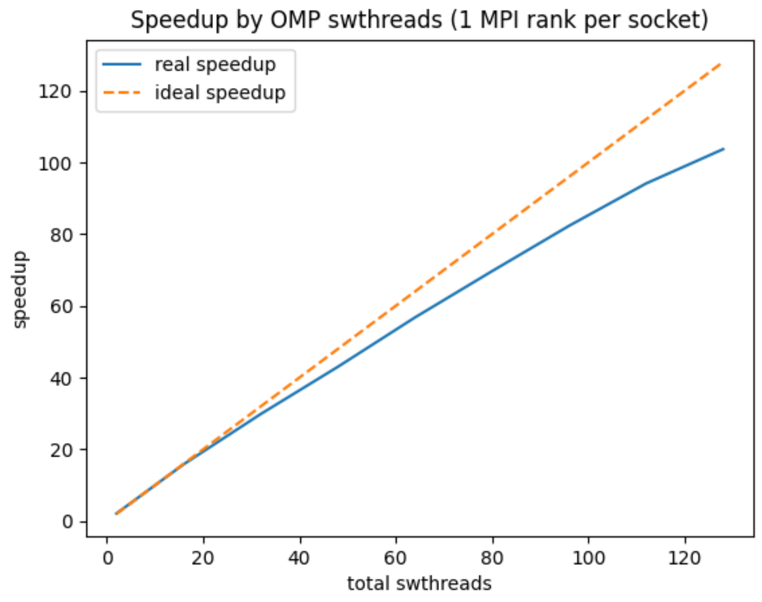
The obtained speedup was not good. What could have been the cause? By analizing the code, it was observed that the `evolve_static()` function managed memory inefficiently, because both buffers were passed in as references, which likely prevented the compiler from doing Return Value Optimization or using move semantics. After restructuring the code, it was possible to introduce a `swap()` of the two buffers, which improved performance measurably.

The test was performed again, using the same grid size, and the obtained speedup was more than doubled. For reference: with 112 threads, the speedup was around 40 in the previous test, while in this new one it was around 92. Way better than before!

To better analyze the scaling behaviour, the test was repeated once more, with a bigger grid size: 16k x 16k. Here's the new plot of the runtimes.



And the speedup plot.



This result is quite good. Of course it's far from the ideal case when the number of threads is big, because negative effects such as false sharing when updating each thread's chunk boundaries have a bigger impact when more threads are used.

1.3.2 Strong MPI scalability

Keeping the grid size fixed to 16k x 16k, a strong MPI scalability was performed on 2 Epyc nodes. The runtime and speedup plots follow.

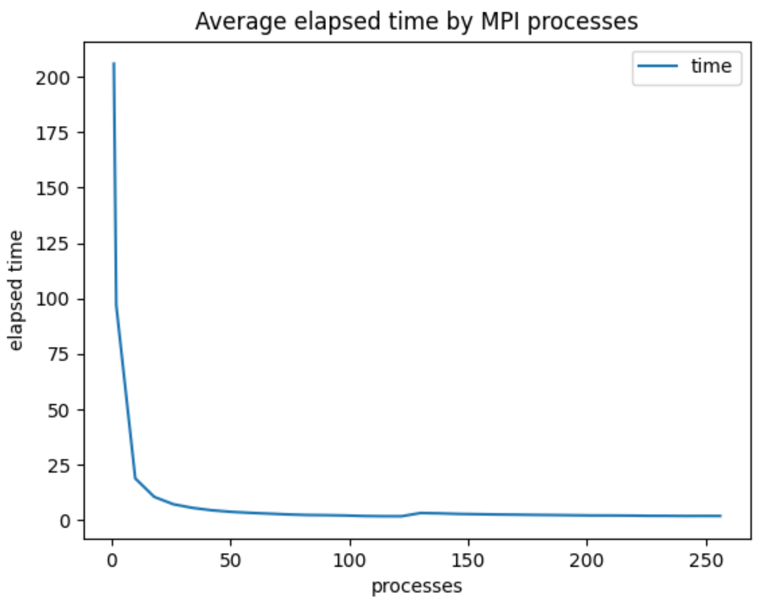


Figure 2: Strong MPI scaling (runtime) on two **Epyc** nodes.

Clearly, as soon as the number of MPI processes exceeds 128, i.e. as soon as the simulation starts to involve both nodes, there is a speedup drop, most likely due to the fact that while staying in one node, shared memory is used as a communication method by the OpenMPI implementation, and then, when the second node participates too, the (Infiniband) network is used.

However, before that point, the speedup is extremely good (it's basically identical to the ideal one). In order to try to improve the scalability across multiple nodes, the code was restructured and refactored to use MPI nonblocking send and receive routines, thus interleaving communication and computation. This, however, did not yield any improvement. `mpirun`'s binding policies were then tweaked to balance the workload across all the available NUMA regions, but this too did not yield any improvement.

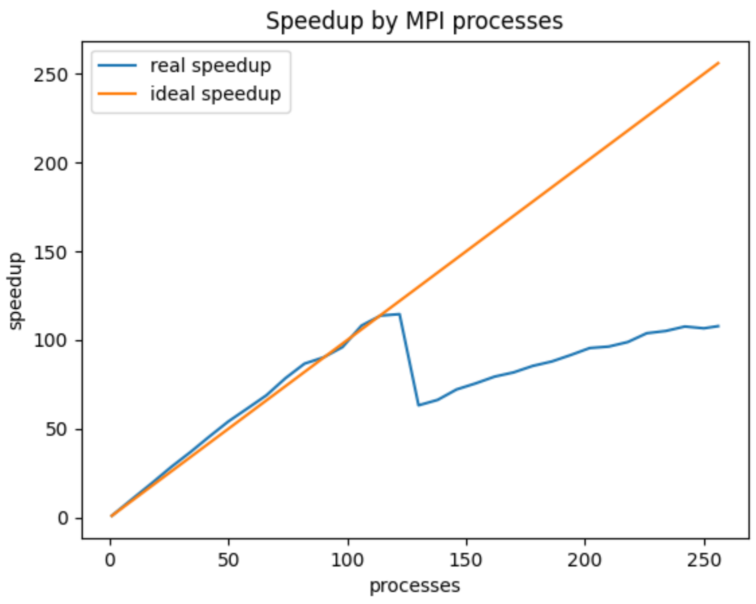


Figure 3: Strong MPI scaling (speedup) on two **Epyc** nodes.

Conversely, on Thin nodes the results are much better.

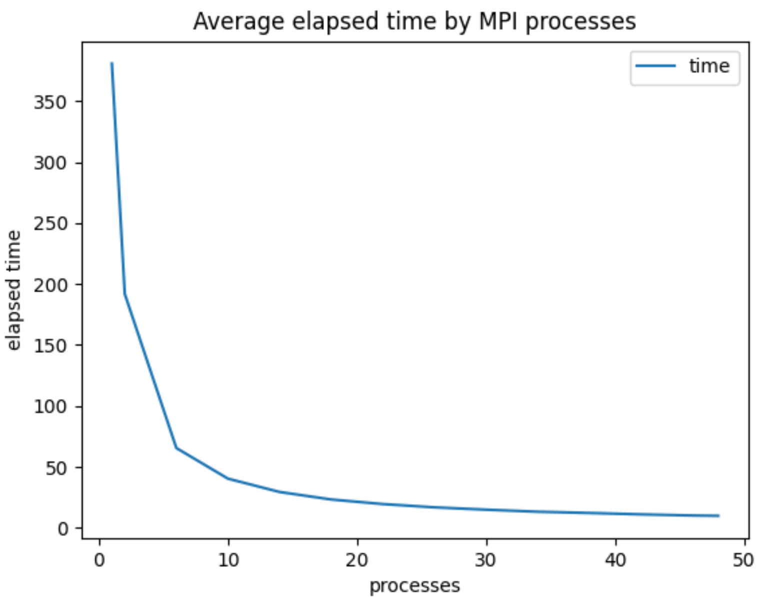


Figure 4: Strong MPI scaling (runtime) on two **Thin** nodes.

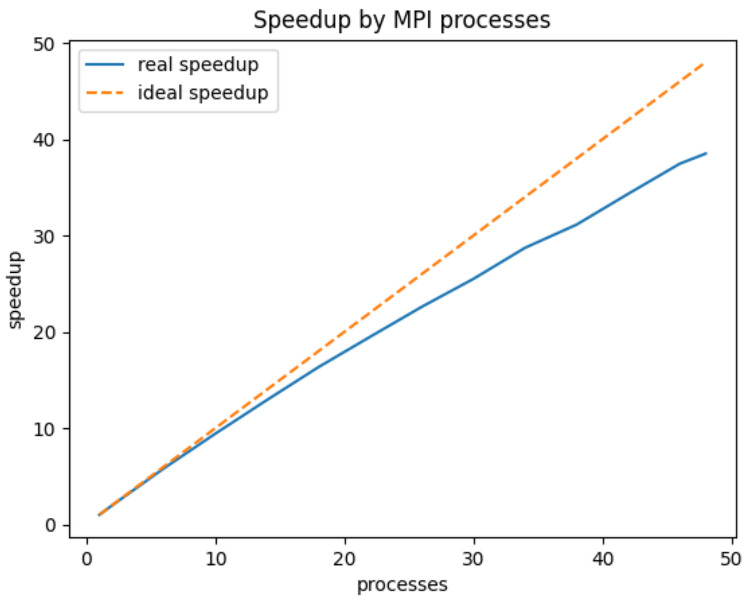


Figure 5: Strong MPI scaling (speedup) on two **Thin** nodes.

These results suggest that the performance of MPI message exchanges between Epyc nodes over the network is suboptimal, possibly due to either a misconfiguration or a temporary problem with Epyc nodes.

1.3.3 Weak MPI scalability

This test has been performed on 4 Thin nodes. The number of MPI processes was increased run after run: 1, 2, 4, 6, 8 (binding them to the sockets). The number of OpenMP swthreads was fixed to 12 (to saturate all occupied sockets, as requested).

To keep the workload per MPI process fixed (net of small rounding errors), the following grid sizes (i.e. heights/widths) were used.

Ranks	Grid size
1	8000
2	11314
4	16000
6	19596
8	22627

Table 1: Mapping between number of MPI processes and grid size.

The efficiency plot follows.

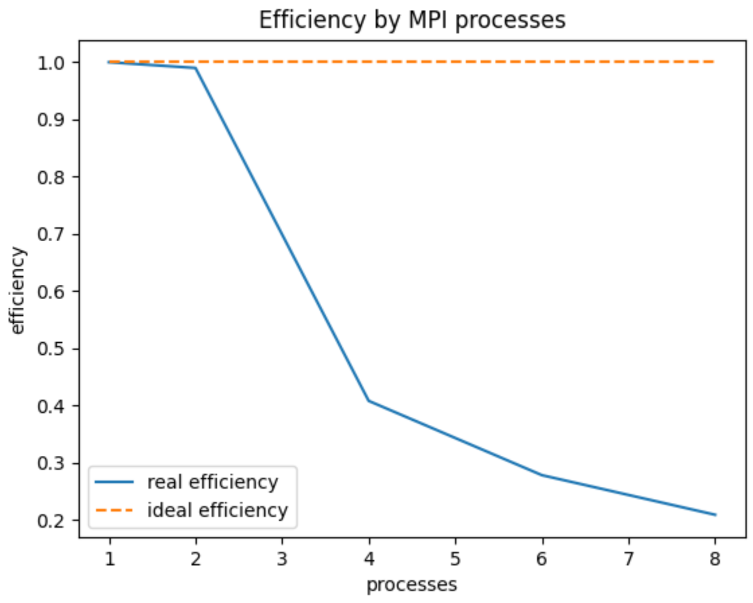


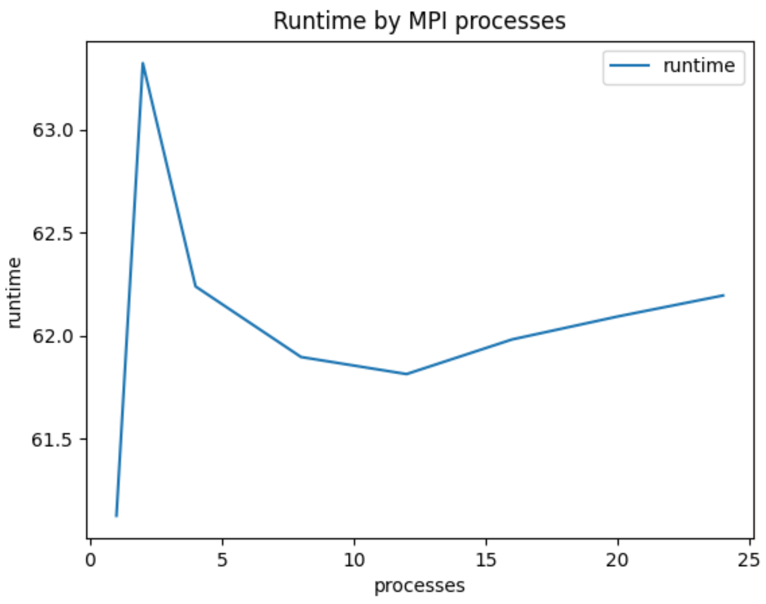
Figure 6: Weak scaling efficiency on 4 Thin nodes.

The weak scaling behaviour is not good at all. There is a huge drop when executing the program on more than one node. Again, this could be related to the network becoming the scalability bottleneck. In order to confirm or contradict this hypothesis, testing on a larger pool of nodes would be a good idea, to check if the efficiency settles between 20 and 30%, but Orfeo's limit for students is of 4 nodes per job.

1.3.4 Ordered evolution results

Ordered evolution cannot be parallelized, but some tests were performed to check the overhead introduced by MPI communication routines. Using an 8k x 8k grid, evolving it for 50 steps, with an increasing number of MPI processes (ranging from 1 to 24 on a Thin node), the runtimes were collected, averaged and plotted.

It can be seen that the runtimes are mostly constant, with some small fluctuations when using only a few processes, which is surprising, because MPI routines are actually called fewer times when the number of ranks is low!



2 Matrix multiplication benchmarking

2.1 Introduction

This section presents the results of some benchmarks that have been run on the cluster. Three BLAS libraries have been compared on the task of matrix multiplication: MKL, OpenBLAS, and BLIS. While the first two are available by default on the cluster, BLIS required compiling from source. Since the intention was to run the benchmarks on both Epyc and Thin nodes, the BLIS library was built twice, once per node type, in order to prevent compatibility issues caused by potential machine-dependant optimizations performed by the compiler.

2.2 Building the executables

One single source file was provided. It has been modified to suppress useless output and print to the standard output only the relevant pieces of information as comma-separated values, in order to allow easy output redirection to CSV files. The provided Makefile has been modified adding `-O3 -march=native -mtune=native` to the `CFLAGS`, in order to optimize the machine code as much as possible. It has then been built in two different flavours (i.e. using single- and double-precision floating point numbers as the datatypes) on both node types.

2.3 Running the benchmarks

The executables have then been run by means of shell scripts submitted to Slurm. Each script repeated each measurement a number of times, to smoothen fluctuations in the results (by averaging them). Finally, results were compared with the theoretical peak performance of each node.

The "fixed number of cores, increasing matrices' size" benchmark has been run with only one OpenMP allocation policy, i.e. 1 swthread per core, filling the whole node. The matrices' size has been increased starting from 2000x2000 up to 30000x30000, as this upper bound seemed more appropriate to allow the CPUs to express all their potential. The step size has been set to 2000, in order not to monopolize the nodes for too long. Only the described allocation policy was used because, when a node is saturated with swthreads, other pinning policies do not make sense (but it is important not to allow swthread migration between cores, as it can hurt performance). The OpenMP documentation on ARM's developer portal suggests the same.

The "fixed matrix size, increasing number of cores" benchmark has been run setting the matrices' size to 10000x10000. Three allocation policies have been tested: `OMP_PLACES=cores OMP_PROC_BIND=close`, `OMP_PLACES=sockets OMP_PROC_BIND=master` and `OMP_PLACES=cores OMP_PROC_BIND=spread`. In the first and third cases, the number of swthreads was scaled up to the maximum possible (node saturation), while in the second case, it only made sense to saturate one socket, because beyond that point, other swthreads would have been scheduled on the already saturated socket, hurting performance.

2.4 Theoretical peak performance

2.4.1 Epyc nodes

According to this resource, the CPUs used in Epyc nodes are capable of 16 double-precision AVX2 FLOPs per cycle. Assuming that the single-precision FLOPs per cycle is thus 32, the total number of GFLOPS for a dual-socket Epyc node can be calculated.

$$\begin{aligned}\text{Tpp} &= 2 \cdot \text{cores\_per\_socket} \cdot \text{base\_clock} \cdot 32 = \\ &= 2 \cdot 64 \cdot 2.6 \cdot 10^9 \cdot 32 = 10648 \text{ GFLOPS.}\end{aligned}$$

2.4.2 Thin nodes

To calculate the theroetical peak performance for Thin nodes, the starting point is the number of (single precision) operations per cycle reported on WikiChips (see "Skylake (server)"): 64.

$$\text{Tpp} = 64 \cdot 2 \cdot 12 \cdot 2.6 \cdot 10^9 = 3994 \text{ GFLOPs}$$

2.5 Results

2.5.1 Fixed cores, expanding matrices

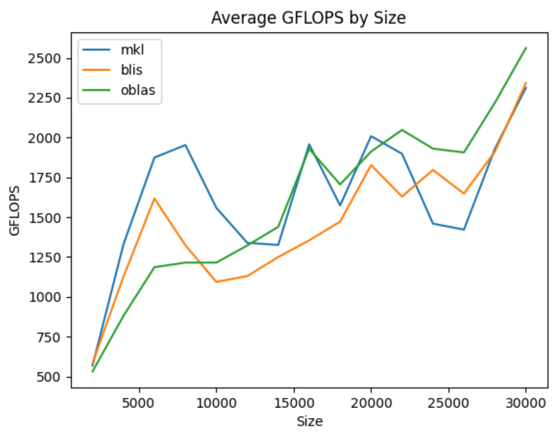


Figure 7: **Single-precision** fixed cores, expanding matrices benchmark on Epyc.

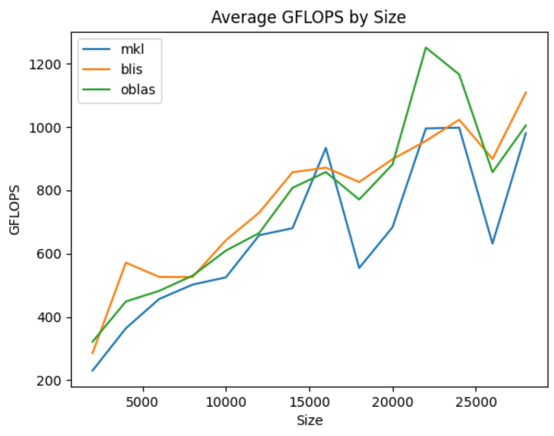


Figure 8: **Double-precision** fixed cores, expanding matrices benchmark on Epyc.

The measured GFLOPs oscillate a lot, and are anyway very far from the theoretical peak, which is not drawn to avoid excessive extension of the y axis. It also appears that on Epyc nodes the source-built blis library performs quite well, especially for the double-precision case, where it exhibits the most consistent behavior. The mkl library, on the other hand, suffers from sudden drops in the measured performance.

The reason why the measured performance is so inconsistent could lie in the fact that Epyc CPUs have quite a complex architecture, with cores grouped together in different NUMA regions inside each socket. This configures three possible memory access cases, i.e. a core might have to retrieve data either from RAM directly attached to the NUMA region it belongs to, or from RAM attached to another NUMA region of the same socket, or from memory attached to the other socket (which is the costliest scenario). Add the usual three-level caching mechanism on top of this, and the final amount of sources of non-uniformities is quite high.

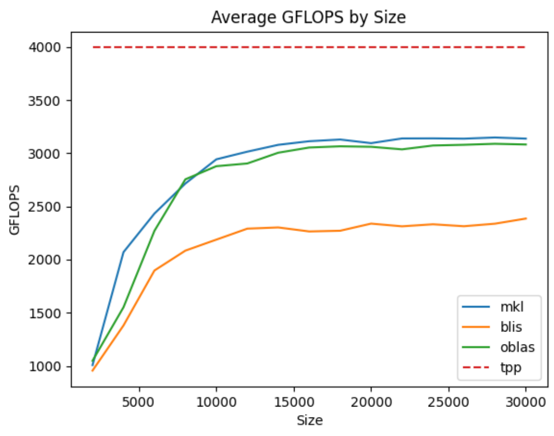


Figure 9: **Single-precision** fixed cores, expanding matrices benchmark on Thin.

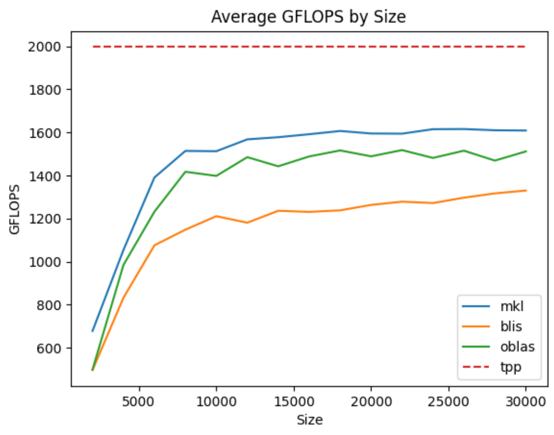


Figure 10: **Double-precision** fixed cores, expanding matrices benchmark on Thin.

On Thin nodes, it can be noted that the blis library performs quite poorly, compared to the other two. Mkl is consistently the best performing one. This is not surprising, since Mkl is developed by Intel. The obtained values are reasonably close to the peak performance and are quite consistent.

2.5.2 Fixed matrices, increasing number of cores

Unsurprisingly, the sockets/master allocation policy produced results that are perfectly aligned to those obtained with cores/close. This was expected, since when using only one socket, sockets/master and cores/close are equivalent. Plots follow for the cores/close policy.

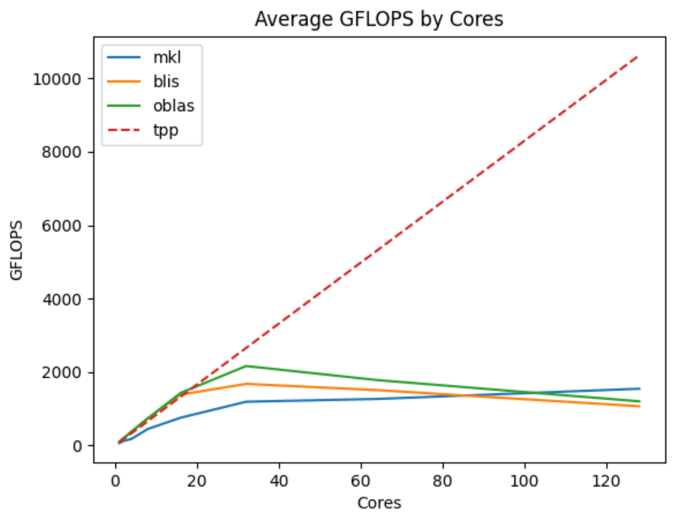


Figure 11: **Single-precision** fixed matrices, increasing cores on Epyc, 10k x 10k matrices, cores/close allocation policy.

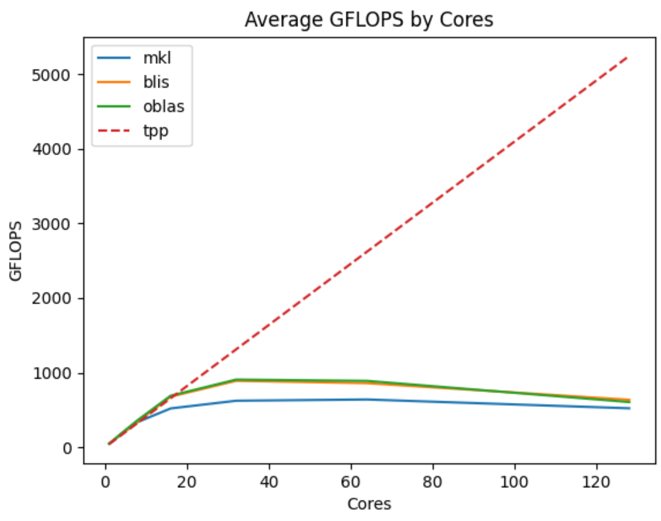


Figure 12: **Double-precision** fixed matrices, increasing cores on Epyc, 10k x 10k matrices, cores/close allocation policy.

The plots show that it would have probably been better to run these benchmarks with bigger matrices, in order to allow the code to fully express its scaling potential. Anyway, mkl again proved itself to be the worst performing library on Epyc nodes.

Increasing the matrices' size to 20k x 20k, the trend is simialr, but indeed, higher GFLOPS values are reached, especially in the single precision case.

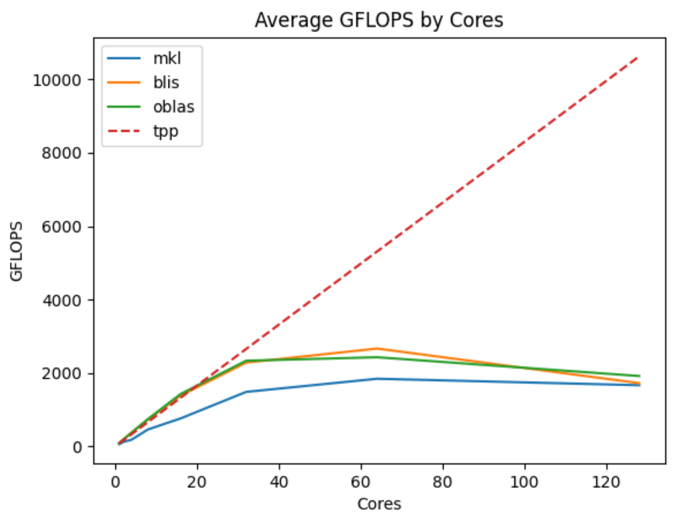


Figure 13: **Single-precision** fixed matrices, increasing cores on Epyc, 20k x 20k matrices.

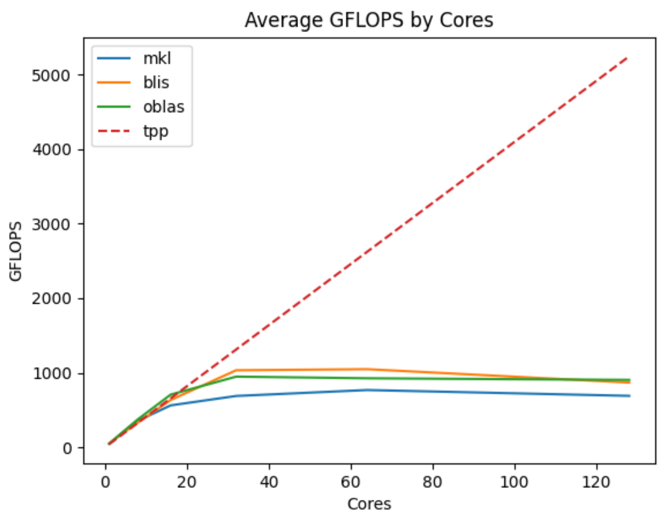


Figure 14: **Double-precision** fixed matrices, increasing cores on Epyc, 20k x 20k matrices.

When using the cores/spread allocation policy, the results are similar, although slightly worse.

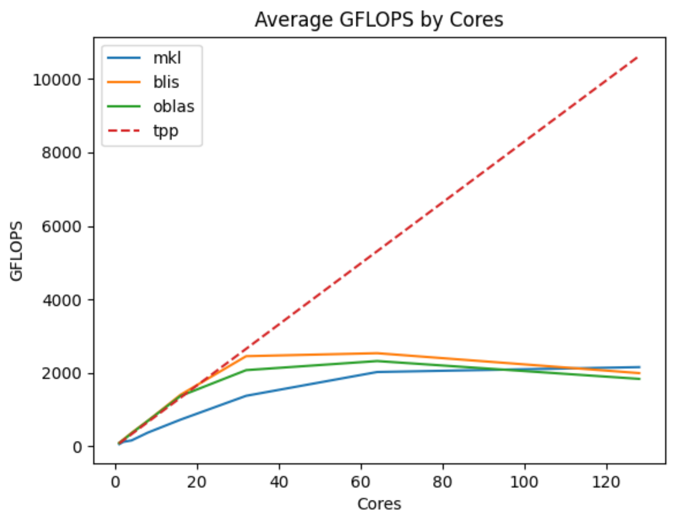


Figure 15: **Single-precision** fixed matrices, increasing cores on Epyc, 20k x 20k matrices, cores/spread allocation policy.

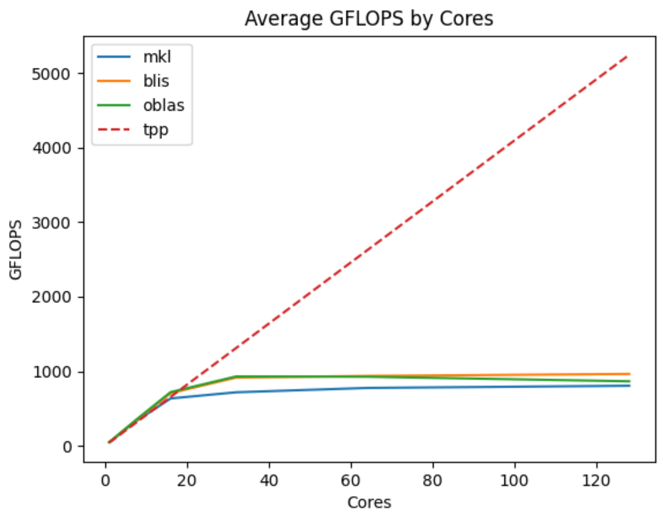


Figure 16: **Double-precision** fixed matrices, increasing cores on Epyc, 20k x 20k matrices, cores/spread allocation policy.

Let us now move to Thin nodes.

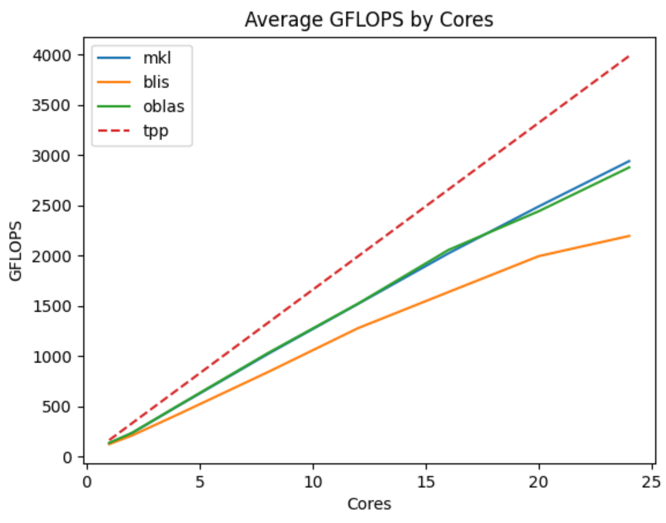


Figure 17: **Single-precision** fixed matrices, increasing cores on Thin, 10k by 10k matrices, cores/close allocation policy.

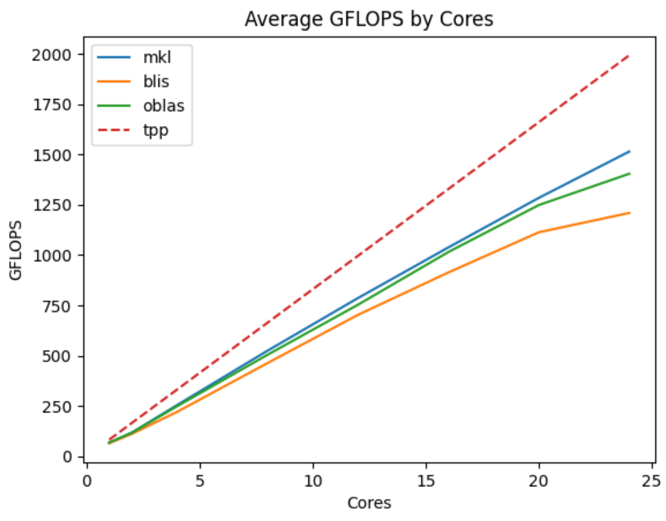


Figure 18: **Double-precision** fixed matrices, increasing cores on Thin, 10k by 10k matrices, cores/close allocation policy.

Due to the smaller core count of Thin nodes, these plots do not show saturation in the number of GFLOPs reached. Blis's poor performance on Thin nodes is proven again, while the other two libraries are pretty much equivalent. It can be noted that in the right portion of the double-precision plot, when all the lines start to bend slightly, mkl's seems to bend less than

OpenBLAS's and Blis's, again suggesting that under high loads, the mkl library is able to squeeze the maximum amount of performance out of the CPUs.

3 Lessons learnt

Carrying out this assignment has been an opportunity to improve my work methodology and understand which design & implementation practices are good and which are not. Recapping:

- safe code prevents headaches: using C++ and Boost.MPI is a very effective way to write cleaner and safer MPI code, compared to using C;
- use the tools available: avoid reinventing the wheel. There's plenty of high quality, open-source libraries that can fulfill almost any need (e.g., in this project, Boost, mimalloc and Argparse). Using them makes writing code faster and contributes to the project's robustness.
- verify assumptions at design time: if I hadn't assumed that using a parallel for loop implied a bigger overhead with respect to tasks, I could have gone for the simpler solution straight from the beginning. Implementing the `taskloop` has however been an interesting exercise, to acquaint myself with a more advanced OpenMP construct.