High Performance Computing Project - A.A. 2022/2023

Andres Bermeo Marinelli

September 20, 2023

Contents

Chapter 1

Benchmarking MKL, OpenBLAS, and BLIS

1.1 Introduction

In this exercise we compare the performance of three High Performance Libraries (HPC): MKL, OpenBLAS, and BLIS. In particular, we focus on the level 3 BLAS function called `gemm`, which multiples an $m \times k$ matrix A times a $k \times n$ matrix B and stores the result in an $m \times n$ matrix C.

This function comes in two types, one for single precision (float) and the other for double precision (double). Furthermore, it is capable of exploiting paralellism using OpenMP (OMP) to speed up the calculations, provided that we have the required computational resources.

Using squared matrices only, we perform a scalability study in two scenarios. In the first scenario, we fix the number of cores, and increase the size of the matrices from 2000 to 20000. In the second scenario, we fix the matrix size to 10000 and increase the number of cores that `gemm` can use by modifying the `OMP_NUM_THREADS` environment variable.

In both scenarios, we repeat the measurements for both single and double precision, for both THIN and EPYC nodes, using the maximum number of cores.

Furthermore, for the second scenario, we also modify the thread affinity policy of OMP in order to observe any differences.

1.2 Methodology

1.2.1 Compiling BLIS and obtaining binaries

We begin by downloading the BLIS library by using the following commands:

```
$git clone https://github.com/flame/blis.git
$cd blis
$srun -p {NODE} -n1 ./configure --enable-cblas --enable-threading=openmp --prefix=/path/to/myblis
$srun -p {NODE} -n 1 --cpus-per-task={P} make -j {P}
$make install
```

Where `NODE` can be specified as either `THIN` or `EPYC` and `P` are the available cores for each node, 24 and 128 respectively.

With these commands, we have compiled the BLIS library for the desired architecture.

Next, we specify the flag in the Makefile to compile for float or double using `DUSE_FLOAT` or `-DUSE_DOUBLE`. Then, we run:

```
$salloc -n {P} -N1 -p {NODE} --time=1:0:0
$module load mkl/latest
$module load openBLAS/0.3.23-omp
$export LD_LIBRARY_PATH=/path/to/myblis/lib:$LD_LIBRARY_PATH
$srun -n1 make cpu
```

Which will generate the binaries for the desired architecture, with floar or double precision, depending on the flag we used.

To run, we use:

```
$srun -n1 --cpus-per-task=128  ./gemm_mkl.x {size_M} {size_K} {size_N}
$srun -n1 --cpus-per-task=128  ./gemm_oblas.x {size_M} {size_K} {size_N}
$srun -n1 --cpus-per-task=128  ./gemm_blis.x {size_M} {size_K} {size_N}
```

At the end of this procedure, we should have the appropriate binaries for each architecture, and for each type of precision, double or float.

We now detail the steps to obtain the measurements for both scenarios.

1.2.2 Using a fixed number of cores

For this section, we use all the cores available in a THIN or an EPYC node: 24 and 128, respectively.

Since we only use squared matrices, we can describe the dimensions of the matrices with a single number, which we call "size".

For both architectures, we start with a size of 2000 and end with a size of 20000, with jumps of 2000 for a total of 10 sizes. For each size, we repeat the measurement 10 times and report the average and standard deviation.

Finally, we repeat the measurements for both floating point precision and double point precision.

The scripts that were used can be found in the folder `exercise2/scripts`, under the name `es2_1_thin.sh` and `es2_1_epyc.sh`.

It is important to observe that in this section, since we are using the entire node, there is little possibility to play with combinations of thread affinity.

This will be done for the next section.

Furthermore, contrary to the guidelines for the exercise, we decided to use the entire node to benchmark its full capacity, and also to avoid wasting resources.

In fact, to obtain an accurate benchmark, we need to reserve the whole node, regardless of the number of cores we decide to use. This is because if other people began to use the other half of the node, this could introduce additional workloads which interfere with the benchmark.

1.2.3 Using a fixed matrix size

For this section, we fix the size of the matrices to 10000. Then, we slowly increase the number of cores to be used, until we reach the maximum.

To set the number of cores, we change the environment variable `OMP_NUM_THREADS` to the desired value.

For THIN nodes, which have 24 cores, we start using 1 core, then 2 and then we increase by steps of 2, for a total of 13 points.

For EPYC nodes, which have 128 cores, we start from 1, then 10 and then we increase by steps of 10 until 120. We also use 128 cores, to see what happens at full capacity. We obtain a total of 14 points.

We repeat all measurements 10 times and report the average and standard deviation.

As usual, we repeat this process for both floating and double point precision.

In this section, we have the liberty to explore different thread allocation policies since we are not always using the whole node.

We decided to use following combinations:

1. `OMP_PLACES=cores` and `OMP_PROC_BIND=close`
2. `OMP_PLACES=cores` and `OMP_PROC_BIND=spread`

The scripts that were used can be found in the folder `exercise2/scripts`, under the names `es2_2_close_thin.sh`, `es2_2_close_epyc.sh`, `es2_2_spread_thin.sh`, and `es2_2_spread_epyc.sh`.

1.3 Results and Discussion

Before we discuss the results of both exercises individually, we briefly introduce the equation to calculate the theoretical peak performance (T_{pp}) of a machine:

$$T_{pp} = \text{Core Count} \times \text{clock freq.} \times \text{IPC} \tag{1.1}$$

Where `IPC` is the instructions per cycle that the architecture is capable of executing.

This equation is very intuitive. The clock frequency tells us how many cycles per second a single core is able to achieve. The ipc factor tells us how many instructions per cycle the core can execute. This number is different for single precision (SP) and double precision (DP) operations. Finally, we need to multiple this by the number of cores that our machine has.

On orfeo, THIN and EPYC nodes are composed of:

- THIN: 24 Intel(R) Xeon(R) Gold 6126 CPU's at 2.60GHz - Skylake
- EPYC: 128 EPYC AMD 7H12 CPU's at 2.60GHz - Zen 2 (7002 a.k.a "Rome")

Skylake architecture is reported[1] to be able to execute 64 SP FLOP per cycle and 32 DP FLOP per cycle. On the other hand, Zen 2 is reported[1] to execute 32 SP FLOP per cycle and 16 DP FLOP per cycle.

Therefore, we obtain:

Node Type	Total Cores	IPC (SP)	IPC (DP)	T_{pp} (SP)	T_{pp} (DP)
THIN	24	64	32	$\sim$ 4 TFLOPS	$\sim$ 2 TFLOPS
EPYC	128	32	16	$\sim$ 10.6 TFLOPS	$\sim$ 5.3 TFLOPS

Table 1.1: Performance table of THIN and EPYC node architectures.

Now we can proceed to discuss the results of the exercise.

1.3.1 Using a fixed number of cores

As mentioned above, in this section we keep the number of cores fixed to the maximum available in the node, namely 24 for THIN and 128 for EPYC, and we slowly increase the size of the matrices being multiplied from $m = 2000$ to $m = 20000$.

We first show the results for THIN and then for EPYC nodes.

THIN Nodes

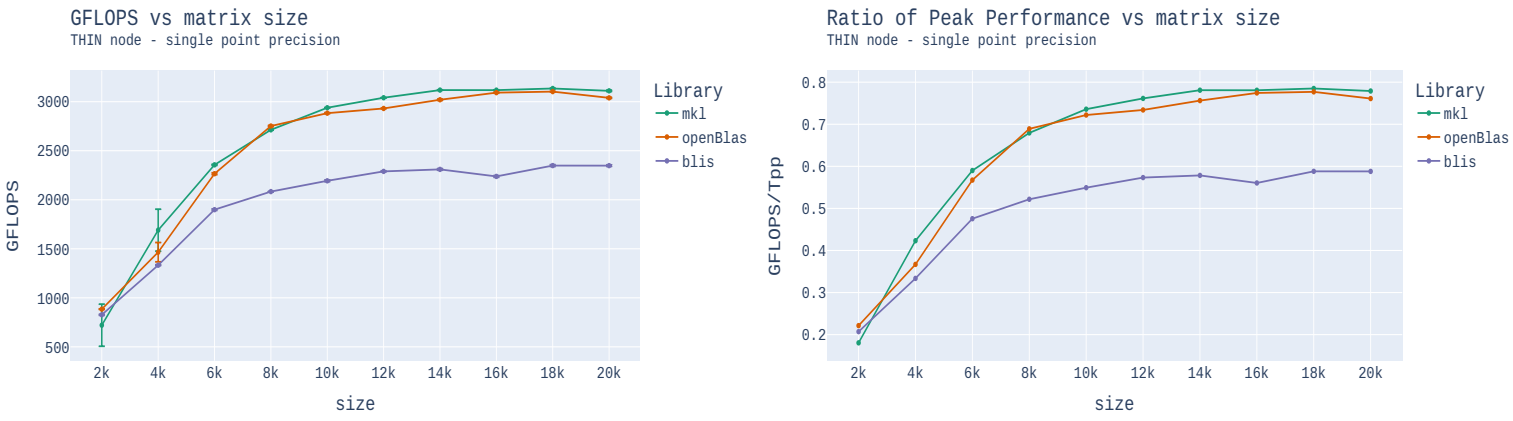

Figure 1.1: Results of SP matrix-matrix multiplication for THIN nodes. `MKL` and `OpenBLAS` perform similarly, outperforming `BLIS` for all matrix sizes.

We see that both `MKL` and `OpenBLAS` are able to reach ~ 3.2 TFLOPS, which is around $\sim 80\%$ of T_{pp}. On the other hand, the `BLIS` library is not able to exploit the full potential of the machine, arriving only to ~ 2.4 TFLOPS, which is $\sim 60\%$ of T_{pp}.

Furthermore, looking at the ratio of peak performance on the right, we observe that for small matrix sizes, none of the libraries are able to fully exploit the theoretical peak performance of the machine. This is most likely because the problem size is so small, that the majority of cores are starving for data rather than crunching numbers. In fact, we are able to reach the best performance when dealing with matrices of size 20000.

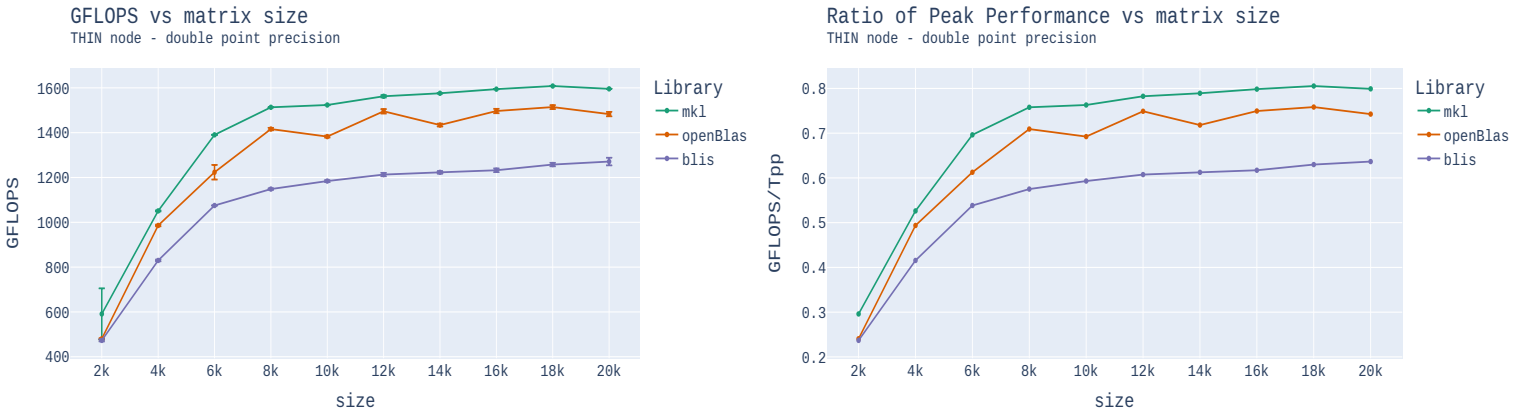

Figure 1.2: Results of DP matrix-matrix multiplication for THIN nodes. `MKL` performs the best, slightly above `OpenBLAS`. Both outperform `BLIS` for all matrix sizes.

We see that `MKL` reaches ~ 1.6 TFLOPS, while `OpenBLAS` is slightly lower, at ~ 1.5 TFLOPS, which is $\sim 80\%$ and $\sim 75\%$ of T_{pp}, respectively. On the other hand, `BLIS` arrives to ~ 1.3 TFLOPS, which is $\sim 65\%$ of T_{pp}.

Furthermore, since double precision is a heavier computation compared to single precision, we see that the libraries perform much better than the their single precision counterparts. For example, looking at the plot for single precision, for matrix size of 4000, we obtain on average around 40% of T_{pp}. On the other hand, for double precision, looking at the same size, we are already at 50% of T_{pp}.

For both SP and DP, and for all matrix size, we notice that `MKL` and `OpenBLAS` are better able to exploit the full potential of a THIN node compared to `BLIS`. Therefore, on THIN nodes, if we need to multiply two matrices, we should always use either `MKL` or `OpenBLAS` to get some more performance. To get the absolute best performance, it is preferable to use `MKL`.

These results shouldn't be surprising, considering that `MKL` is developed by Intel and THIN nodes are Intel-based. Therefore, it is natural to expect that this library is very fine-tuned to their own architecture and is able to exploit the performance of their machines.

Lastly, we observe the impressive results achieved by `OpenBLAS` which is based on the original implementation of Kazushige Goto, and is able to achieve a similar performance to `MKL`, which is maintained by an entire corporation.

EPYC Nodes

Now we show the results on EPYC nodes, which have a T_{pp} of 10.6 TFLOPS for SP and 5.3 TFLOPS for DP.

We first show the results of matrix-matrix multiplication for single point precision.

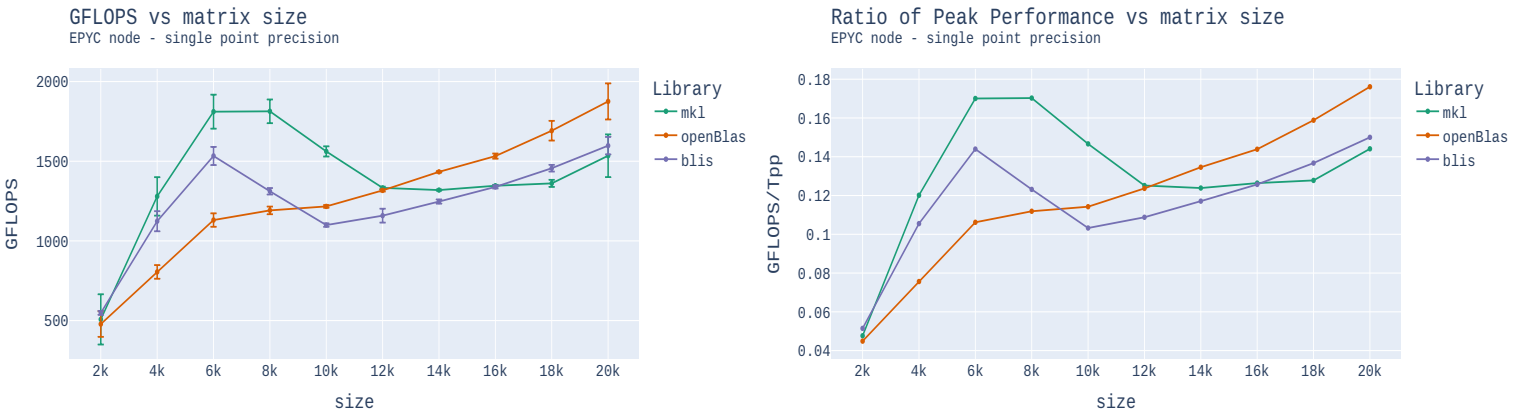

Figure 1.3: Results of SP matrix-matrix multiplication for EPYC nodes. We notice that asymptotically, `OpenBLAS` outperforms `MKL` and `BLIS`, while for small matrices, `OpenBLAS` performs the worst.

In this case, none of the libraries are able to reach more than 18% of T_{pp}. This could be an indication that to properly exploit a full EPYC node, we need to multiply much bigger matrices.

We also notice that for matrices of size $\leq$ 9000, `MKL` and `BLIS` outperform `OpenBLAS`. Between sizes 9000 and 12000, `MKL` performs best and `OpenBLAS` begins outperform `BLIS`. For sizes $\geq$ 12000, `OpenBLAS` performs the best.

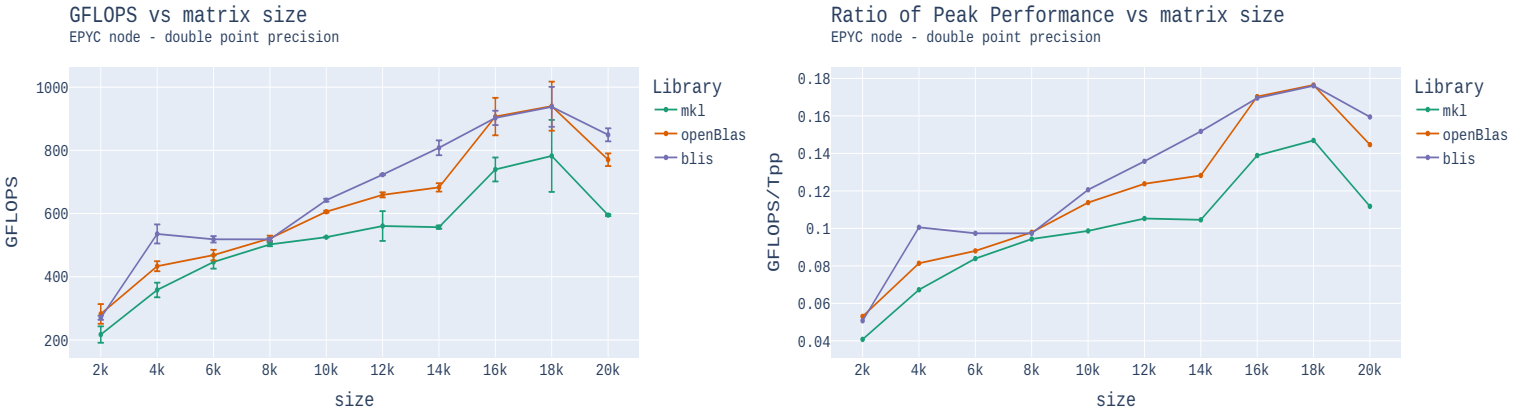

Figure 1.4: Results of DP matrix-matrix multiplication for EPYC nodes. `BLIS` outperforms `MKL` and `OpenBLAS` for all matrix sizes.

Again, we notice that none of the libraries are able to achieve more than 18% of T_{pp}. However, in this case, `BLIS` outperforms the other two libraries for all matrix sizes. The next best performer is `OpenBLAS`, followed by `MKL`, which performs the worst.

In conclusion, on EPYC nodes, it seems that for DP matrix-matrix multiplication, it is better to use `BLIS`, while for SP, it is very dependent on the size of the matrices. For large matrices, we should use `OpenBLAS` while for smaller ones, we should use `MKL`. Furthermore, evidence suggests that to fully exploit and EPYC node, we need to deal with much bigger matrices. So if we are multiplying matrices of size up to 20000, it is much more convenient to just use a THIN node to get more performance.

1.3.2 Using a fixed matrix size

In this section, we fix the matrix size to 10000 and we slowly increase the amount of cores that the libraries can exploit for multithreading through OMP. For THIN nodes, we arrive to 24 cores, while for EPYC nodes, we arrive to 128 cores.

Furthermore, since we are slowly increasing the number of cores that the libraries can use for multithreading, we can study the effects of using different thread allocation policies. In particular, we chose to use `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`, while always using `OMP_PLACES=cores`.

In the first case, the threads will slowly occupy first one entire socket, and then, when it is full, the other one. In the second case, the threads will be placed as spread apart as possible, most likely on different sockets. In both cases, when we use the full node, we expect the results to be the same.

Furthermore, in contrast to the previous part of the exercise, we compare the GFLOPS obtained with the T_{pp} calculated with the cores that are being used. In other words, in equation1.1, instead of using the full 24 or 128 to calculate T_{pp} for the whole node, we use the number of cores we are using for that calculation.

We first analyze THIN and then EPYC nodes.

THIN Nodes

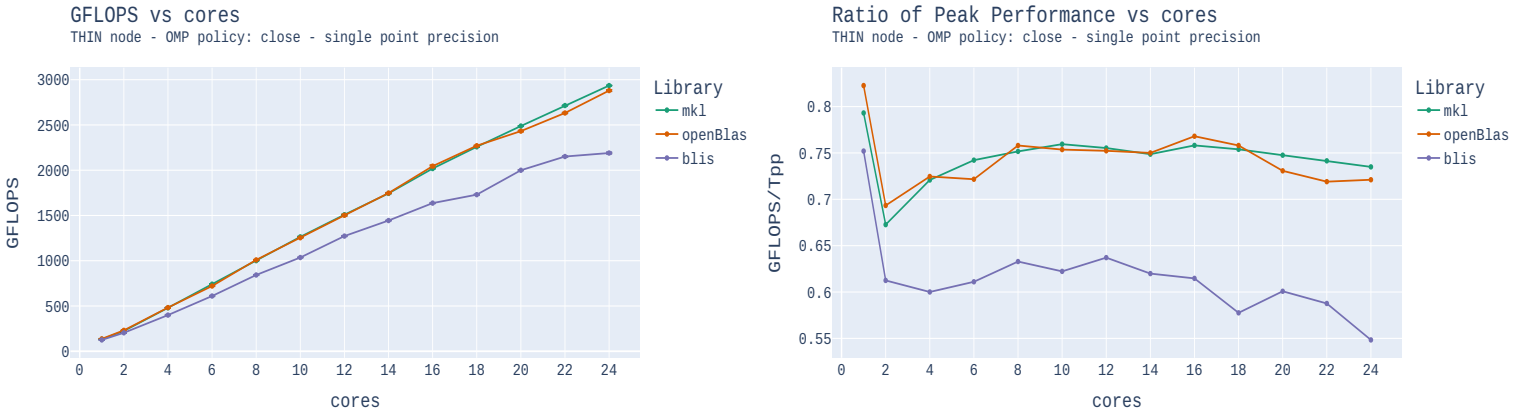

Figure 1.5: Results of SP matrix-matrix multiplication as the number of cores increase, using OMP close policy. `MKL` and `OpenBLAS` obtain the best performance.

As we can see from the graph, both `MKL` and `OpenBLAS` are able to maintain $\sim 75\%$ of T_{pp} for cores ≥ 6. On the other hand, `BLIS` is able to achieve $\sim 60\%$ of T_{pp}. With 24 cores this figure decreases to 55%.

Interestingly, we notice that with 2 cores, there is a significant performance drop. This is most likely explained by the fact that both cores are mapped to the same socket due to the close policy and must share resources such as the higher level caches, causing some contention.

Furtermore, once again, we notice that the best performing library is `MKL` which is Intel-based, closely followed by `OpenBLAS`.

Now we analyze the results using a spread policy.

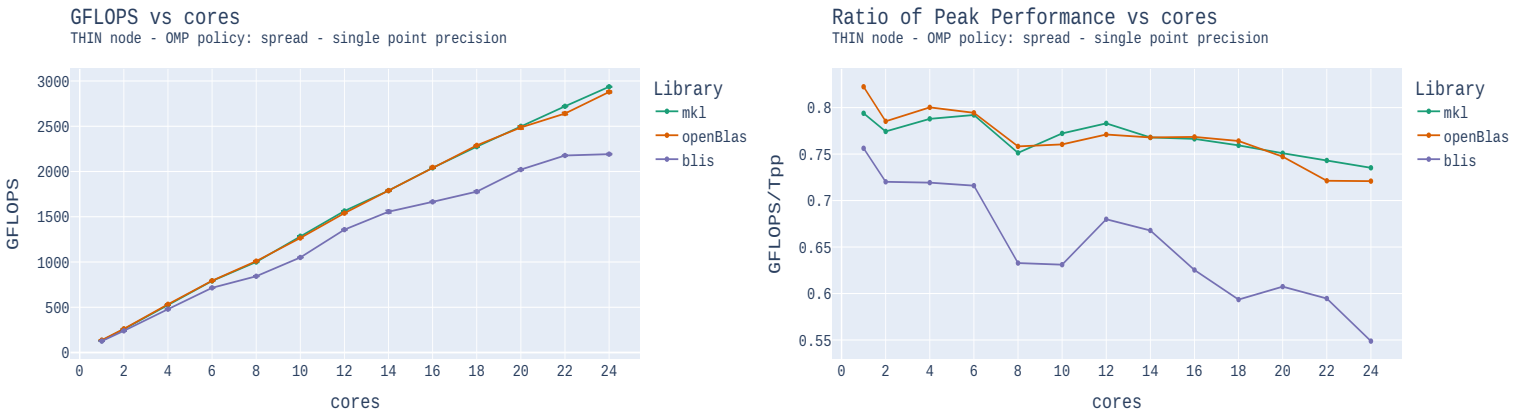

Figure 1.6: Results of SP matrix-matrix multiplication as the number of cores increase, using OMP spread policy. `MKL` and `OpenBLAS` are the best perfomers.

Using a spread policy, we obtain very similar results to the case where we use a close policy. The main difference is that we don't have the same performance drop at 2 cores. This is most likely due to the fact that with a spread policy, each core is mapped to its own socket and there is no contention for resources.

In fact, we notice that on average, the performance is slightly better than the close policy counterpart, and this is probably due to better resource usage from the beginning, since threads don't have to compete for resources immediately. This highlights the importance of using the correct mapping policy to obtain better performance.

Now we briefly analyze the results obtained for double precision.

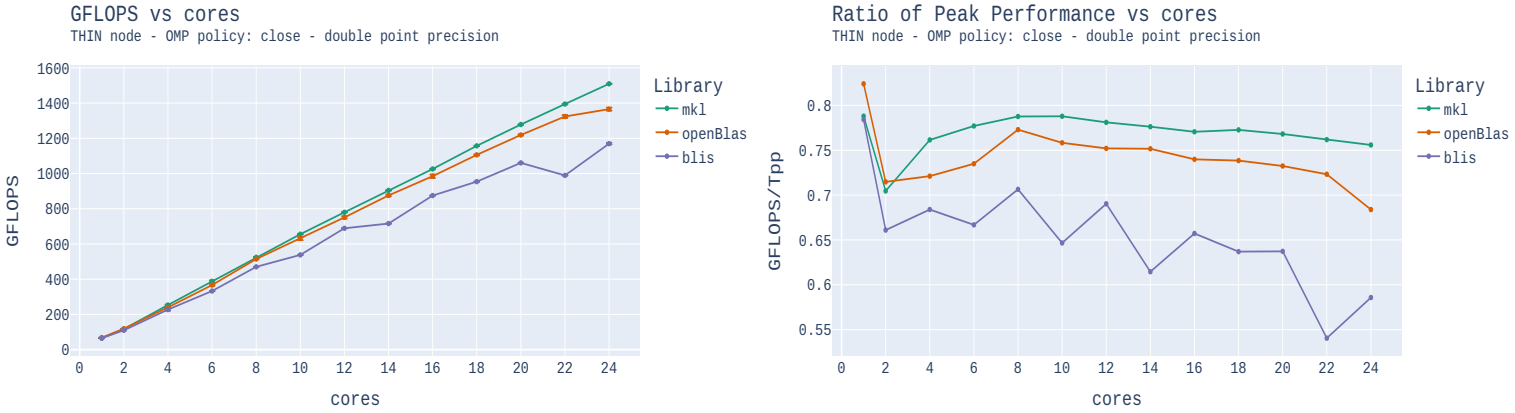

Figure 1.7: Results of DP matrix-matrix multiplication as the number of cores increase, using close policy. `MKL` and `OpenBLAS` perform the best.

Similarly to the case of single precision, `MKL` is able to achieve around $\sim 75\%$ of T_{pp}. However, `OpenBLAS` suffers from a bit of performance degradation compared to the SP case.

Once again, we notice the immediate drop in performance as soon as we use 2 cores, which is probably caused by the close policy.

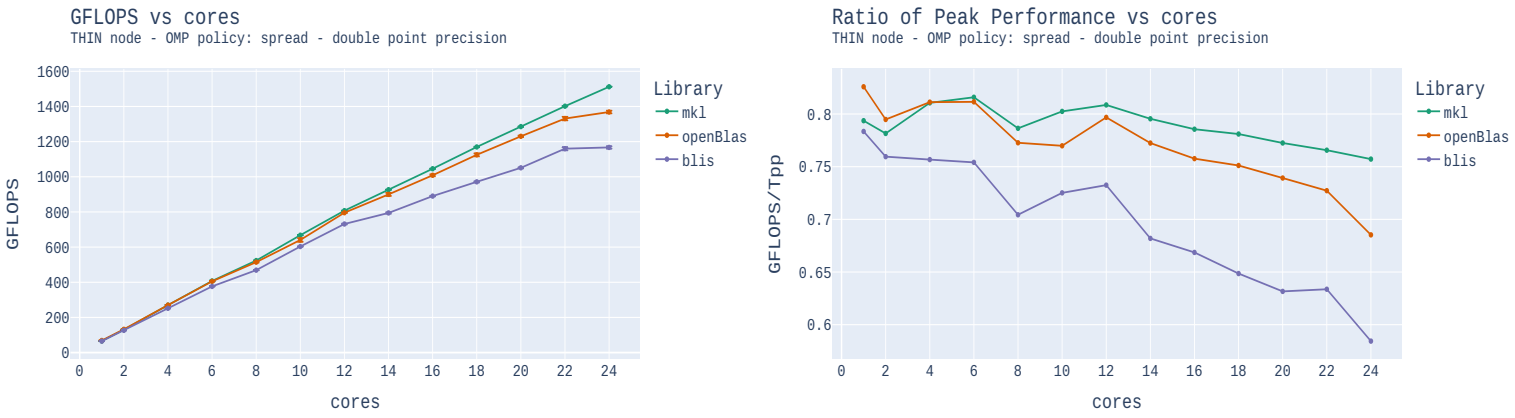

Figure 1.8: Results of DP matrix-matrix multiplication as the number of cores increase, using spread policy.

The analysis and discussion is very similar to the case of single point precision. We no longer see the drop at 2 cores, which is due to better resource usage from the beginning. Compared to the close policy, there is a slightly better performance for this scenario.

As usual, `MKL`, which is Intel-based, performs the best, closely followed by `OpenBLAS`, and finally, `BLIS`, which performs the worst.

Finally, we analyze EPYC nodes.

EPYC Nodes

Considering the results obtained in the first part of the exercise, we expect that `MKL` won't be the dominating library anymore since we are using an AMD architecture. We also expect some more fluctuation in performance among the libraries, similarly to how there was a dependency on the matrix size.

Furthermore, since we obtained low performance with matrices up to 20000, using all 128 cores, we don't expect an asymptotic improvement. What may happen however, is that when we use less cores, the performance will be much better compared to the theoretical peak performance (per number of cores this time).

We begin by analyzing the SP case with close policy.

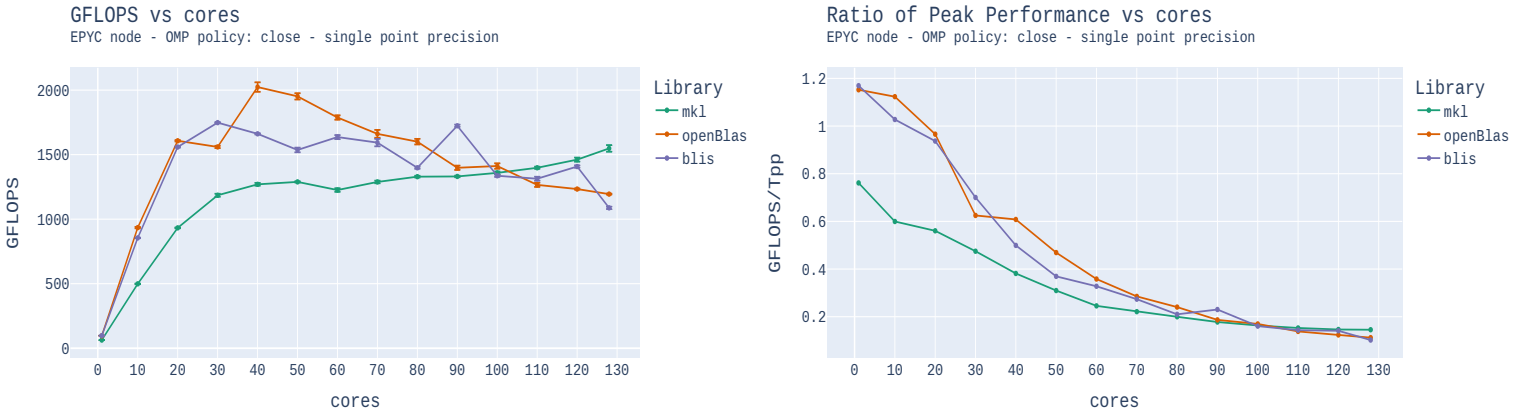

Figure 1.9: Results of SP matrix-matrix multiplication as the number of cores increase, using close policy. The best performance is obtained by `OpenBLAS` with 40 cores.

Looking at the graph, we see that the best absolute performance is obtained by `OpenBLAS` at 40 cores, however, this is only $\sim 60\%$ of T_{pp}. On the other hand, for $10 - 20$ cores, the performance almost on part with T_{pp}. This tells us that to fully exploit the machine as intended, with a matrices of size 10000, we only need a handful of cores.

This information is consistent with our previous hypothesis that to fully exploit an EPYC node, we need to consider much larger matrices.

An immediate consequence of this observation, is that as we use more and more cores, the performance deteriorates considerably. When we use the full node, the performance is $\sim 20\%$ of T_{pp}, which is higher than what we obtained for the first part of the exercise.

Contrary to what happens in THIN, we do not notice any severe drops in performance which could be due to the close policy mapping. However, this will be more noticeable once we analyze what happens when we use the spread policy.

Now we analyze the SP case using spread policy.

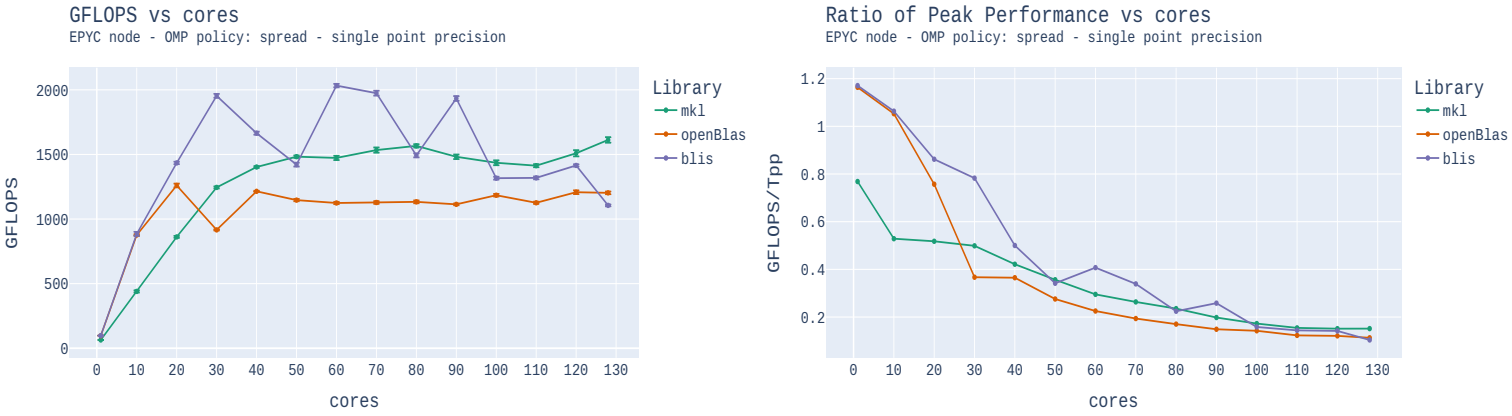

Figure 1.10: Results of SP matrix-matrix multiplication as the number of cores increase, using spread policy.

There is a big change. The first thing is that now `BLIS` obtains the best performance, although it suffers from considerable fluctuations. Furthermore, `OpenBLAS`, which used to perform the best, now performs the worst out of all three libraries.

We also notice that using the spread policy causes both `MKL` and `BLIS` to improve and `OpenBLAS` to worsen. This is interesting because it seems to suggest that the latter library is better able to handle resource contention while the first two are better at fully exploiting resources.

Once again, we notice that as we use more cores, we get worse performance.
Now we look at the DP case with close policy.

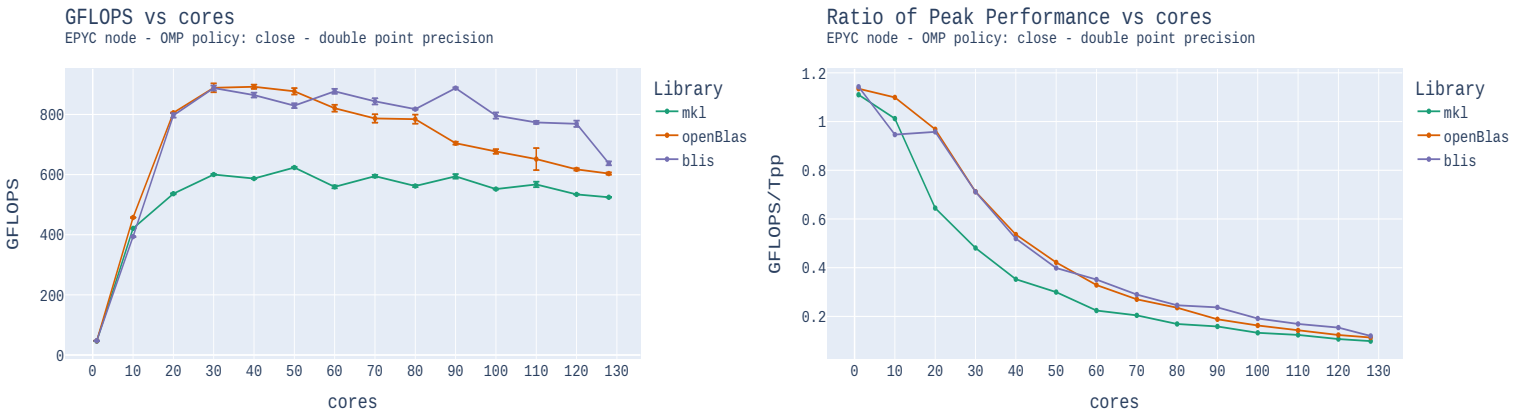

Figure 1.11: Results of DP matrix-matrix multiplication as the number of cores increase, using close policy. The best performance is obtained by `BLIS`, followed by `OpenBLAS`.

For double precision, using a close policy, we observe that `BLIS` performs the best for many cores, while for the first 64 cores, it performs close to `OpenBLAS`, which is the best. `MKL` peforms the worst.

Finally, we analyze the DP case with spread policy.

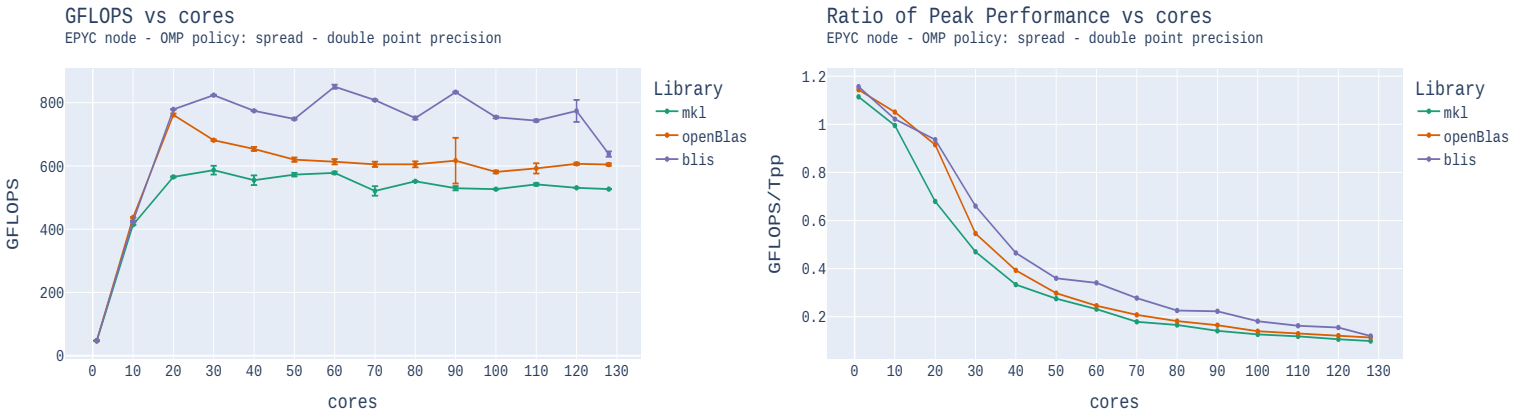

Figure 1.12: Results of DP matrix-matrix multiplication as the number of cores increase, using spread policy.

Using a spread policy makes the gap between `BLIS` and `OpenBLAS` much wider. Also, `BLIS` seems to be more capable at maintaining the throughput as the number of cores are increasing. Of course, this means however, that the peformance compared to the theoretical peak is deteriorating.

1.4 Conclusion

After analyzing the results, we can state many conclusions. The first and most obvious one, is that it is better to use Intel-based implementations for Intel machines. This is evidenced by the fact that `MKL` consistently performed the best in all scenarios for THIN, although it was closely followed by `OpenBLAS`.

Next, if we are dealing with non-Intel architecures, such as EPYC, it is not advisable to use `MKL`, as it consistenly underperforms compared to `OpenBLAS` and `BLIS`. We also notice that `OpenBLAS` and `BLIS` behave completely differently, depending on the mapping policy we use.

We find that for matrices up to 20000, THIN nodes are able to achieve $\sim 75\%$ of T_{pp}, with `MKL` and `OpenBLAS`. However, such matrix dimensions are to small to be able to fully exploit an EPYC node which has 128 cores. In fact, in all of our experiments, we are unable to obtain more than 20% of the T_{pp} of the full node.

Finally, as an improvement to this exercise, it would be very interesting to analyze what happens with an EPYC node, using matrices of size 50000 or bigger perhaps, to understand whether we can obtain better performance in this case.

Chapter 2

Conway's Game of Life

2.1 Introduction

This exercise is devoted to implementing a scalable version of Conway's Game of life[2]. The game consists of a $k \times k$ grid, where each cell $\mathcal{C}$, at position (i, j), can be "alive" or "dead".

The grid evolves over time by looking at each cell's eight nearest neighbors and observing the following rules:

- A dead cell with exactly three live neighbors comes to live (*birth*).
- A live cell with two or three neighbors stays alive (*survival*).
- A dead or live cell with less than two or more than three neighbors dies or stays dead (*death*).

These seemingly simple rules give rise to many interesting behaviors and patterns[3].

There exist two methods to evolve the grid: `static` and `ordered` evolution. In `static` evolution, we freeze the state of the entire grid $\mathcal{G}_t$ at time step t, and compute $\mathcal{G}_{t+1}$, separately, while looking at $\mathcal{G}_t$. This corresponds to maintaining two buffers, one for the current generation and the other for the new generation.

On the other hand, in `ordered` evolution, we start from a specific cell, usually in position $(0, 0)$ (top left), and update the elements inplace. This means that the state of each cell depends on the evolved state of some of its neighbors. We call this "ordered" evolution, because the choice of starting point, and the order in which we evolve the grid, lead to different results. In this scenario, the state of each cell instrinsically depends on the history of evolution of *all* the cells before it.

Our implementation must satisfy the following requirements:

1. Randomly initialize a square grid ("playground") of size $k \times k$ with $k \geq 100$ and save it as a binary PGM[4] file.
2. Load any binary PGM file and evolve for n steps.
3. Save a snapshot during the course of evolution with a frequency s, where $s = 0$ means saving only at the end.
4. Support both `static` and `ordered` evolution.

Lastly, it must use both MPI[5] and OpenMP[6] (OMP) to paralellize the computations and be able to process grids of considerably high dimensions.

2.2 Methodology

In this section, we briefly describe some of the important choices for the implementation, as well as their implications. We will proceed by layers, tackling the more abstract and higher level problems first and then slowly going into detail about more technical choices.

The first most important step is to think about how to paralellize the problem in order to use MPI and OMP properly.

Since programs in MPI need to be written completely with paralellization in mind from the start, we must begin to conceptualize the problem in an encapsulated manner from the beginning.

At a high abstract level, we must make two important choices:

1. How we will decompose the problem.
2. How we will perform the IO.

How we solve the first problem is of fundamental importance to ensure that our program is scalable and efficient. Also, it has important consequences on the type of paralellism we will need to use. On the other hand, the second problem will determine the organizational paradigm that we will use in our code, as we will see later.

We briefly discuss both of these topics more in depth in the following subsections.

2.2.1 Decomposition

To exploit paralellism, we must first identify the concurrency in our application and then apply some form of decomposition. The two most important types of decomposition are `domain` and `functional` decomposition.

In `domain` decomposition, multiple workers are performing the same set of instructions on different portions of data (SIMD). In `functional` decomposition, workers are performing different instructions on possibly different data (MIMD).

In the case of `static` evolution, each cell can be updated independently from each other, as long as we have acccess to its neighbors. Therefore, one immediately obviously form of paralellism is for each MPI process to process a "part" of the whole grid. This is an example of `domain` decomposition, and it is shown below:

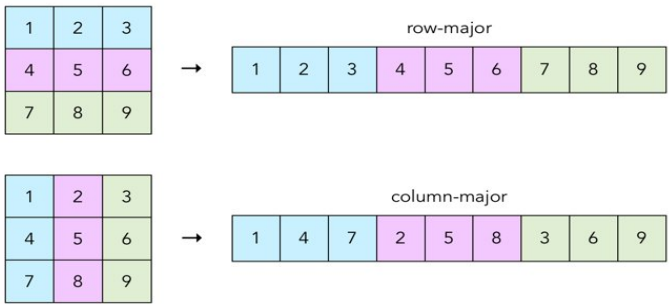

Figure 2.1: Decomposition of an array among three processes. Although the array is conceptualized as a 2D structure, internally, for efficiency, it is represented as a 1D contigous block of memory.

However, we need to decide how to split the grid. We can do a 1D decomposition by "stripes", as shown in image 2.1 or a 2D decomposition by "blocks". It is known that a 2D decomposition is more efficient as it can exploit more bandwidth as more workers will send shorter messages contemporarily. However, it is more complicated both from an implementation point of view and worse for memory access.

Let's talk briefly about this second aspect. Although we talk about the grid as a 2D array, internally, for efficiency, it will be represented as a 1D array.

If we do a 1D split, each process will work on a thick strip of data which continuous in memory. On the other hand, if we do a 2D split, each process will work on a block of memory which is not contiguous and located every certain "jumps" in the 1D array. As a consequence, as each process traverses its block of the grid, it will have more cache misses compared to the 1D case where we perform a simple a linear access.

Therefore, for reasons stated above, and for simplicity and readability, we decided to use the 1D splitting.

Lastly, we must consider how to handle halo regions. Since each process will process a strip (composed of 1 ore more rows) of the grid, the rows at the boundary of the strip must be handled with special attention. These rows - 2 per process - need information that is contained at the boundary of the strips of other processes. Therefore, we need to set up message passing between processes to handle the exchange of the boundary rows, also known as `halo` rows.

2.2.2 MPI IO

The problem specifications require that we must save the grid with a certain frequency s. There are multiple ways to achieve this, and they all have important implications on the scalability of the code.

The simplest way is for each process to write its part of the grid to a separate file. This however, is extremely messy as we loose the flexibility of using a different number of processes between separate runs. More importantly, for a large number of processes, the filesystem will not be able to handle all these requests and will be completely overloaded and possibly crash. This means that for $P >> 1$ we would have very serious problems, which is undesirable, since we wish to use as many processes as we want/have.

The second solution, is to use a master-slave approach. In this paradigm, one process is designated as the master process, while all the other are its slaves. The master will coordinate the IO, and with a frequency s, it will collect information from all the slaves, and take care of writing everything to a file. This approach, although very common, is not scalable, as each process needs to send its part of the grid to the master process. So for $P >> 1$, the bottleneck will be in the communication.

Lastly, the third, and best solution, is to use MPI IO, which was meant to handle exactly this problem. The idea behind MPI IO is very similar to message passing. Each process will call a collective function to read/write its part of the grid and the MPI implementation will take care of exploiting the parallel filesystem and reducing the amount of read and write operations. Furthermore, this is the only solution that takes advantage of the parallel filesystem which allows for a unified logical view of the file, while allowing the file to be stored in physically different locations. This is very advantageous for scalability since many processes can contemporarily write in different physical locations, which increases the throughput of the program. We note that this is a form of `functional` decomposition.

2.3 Implementation

2.4 Results and Discussion

2.5 Conclusions

2.5.1 How to add Tables

Bibliography

[1] wikichip. "Floating-point operations per second (flops)." (), [Online]. Available: https://en.wikichip.org/wiki/flops.

[2] LifeWiki. "Conway's game of life." (), [Online]. Available: https://conwaylife.com/wiki/Conway%27s_Game_of_Life.

[3] LifeWiki. "Category: Patterns." (), [Online]. Available: https://conwaylife.com/wiki/Category:Patterns.

[4] J. Poskanzer. "Pgm." (), [Online]. Available: https://netpbm.sourceforge.net/doc/pgm.html.

[5] OpenMPI. "Openmpi." (), [Online]. Available: https://www.open-mpi.org/.

[6] OpenMP. "Openmp." (), [Online]. Available: https://www.openmp.org/.