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

Andres Bermeo Marinelli

September 19, 2023

Contents

1 Benchmarking MKL, OpenBLAS, and BLIS **2**
- 1.1 Introduction . . . 2
- 1.2 Methodology . . . 2
  - 1.2.1 Compiling BLIS and obtaining binaries . . . 2
  - 1.2.2 Using a fixed number of cores . . . 3
  - 1.2.3 Using a fixed matrix size . . . 3
- 1.3 Results and Discussion . . . 4
  - 1.3.1 Using a fixed number of cores . . . 4
  - 1.3.2 Using a fixed matrix size . . . 5
- 1.4 Conclusion . . . 8

2 Conway's Game of Life **10**
- 2.1 Introduction . . . 10
- 2.2 Methodology . . . 10
  - 2.2.1 Problem decomposition . . . 11
- 2.3 Implementation . . . 11
- 2.4 Results and Discussion . . . 11
- 2.5 Conclusions . . . 11
  - 2.5.1 How to add Tables . . . 11

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	~ 4 TFLOPS	~ 2 TFLOPS
EPYC	128	32	16	~ 10.6 TFLOPS	~ 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, and we slowly increase the size of the matrices being multiplied from $m = 2000$ to $m = 20000$.

THIN Nodes

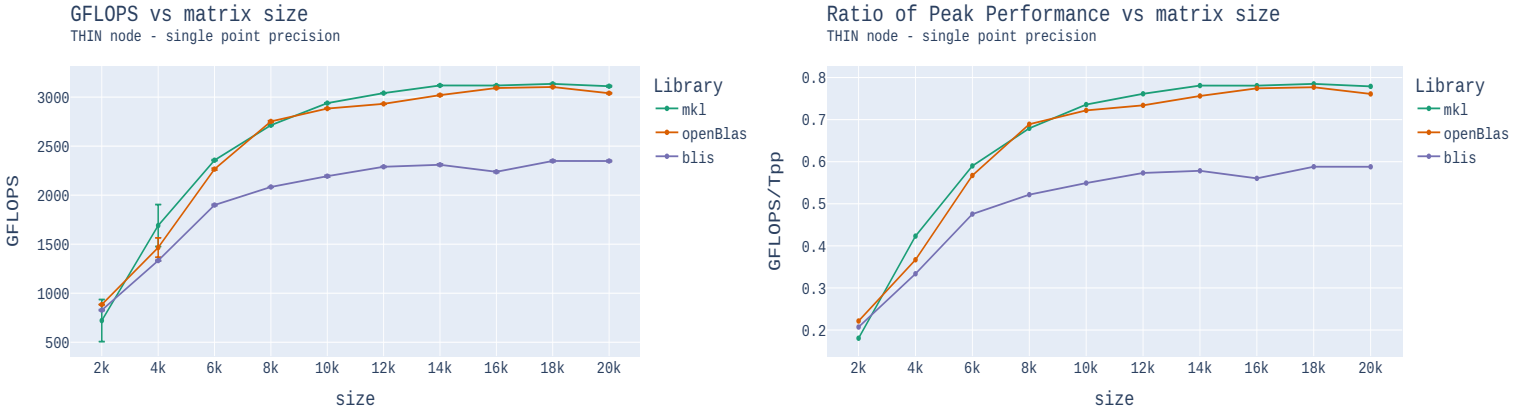

Figure 1.1: Results of SP matrix-matrix multiplication for THIN nodes. `MKL` and `OpenBLAS` perform similarly, outperforming `BLIS` for all matrix sizes.

We see that both `MKL` and `OpenBLAS` are able to reach ~ 3.2 TFLOPS, which is around 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 60% of T_{pp}.

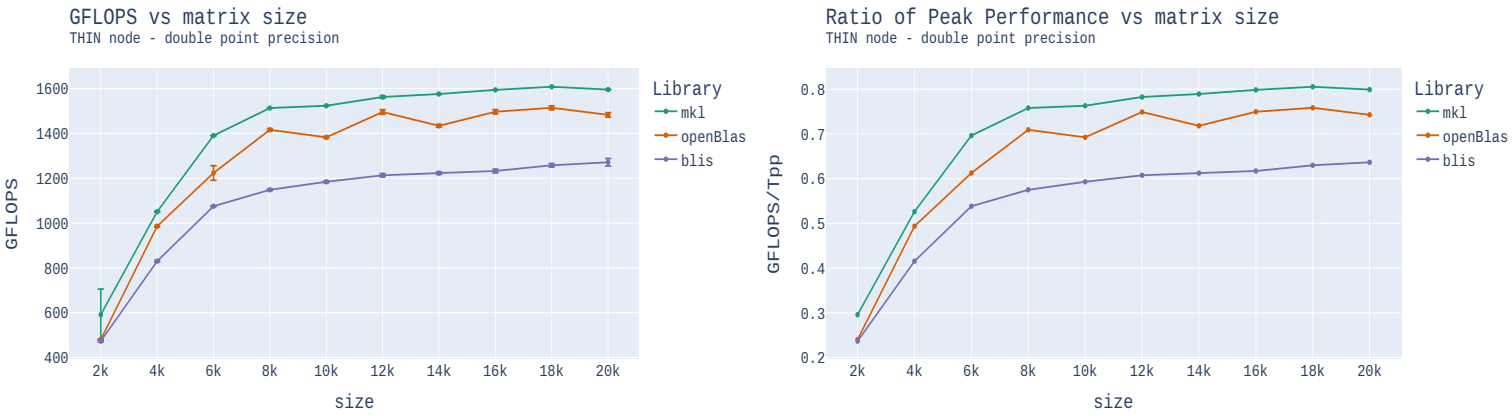

Figure 1.2: Results of DP matrix-matrix multiplication for THIN nodes. `MKL` performs the best, slightly above `OpenBLAS`. Both outperform `BLIS` for all matrix sizes.

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}.

For both graphs, we notice that on `MKL` and `OpenBLAS` are better able to exploit the full potential of a THIN node. On the other hand, `BLIS` seems to only be able to arrive to $\sim 60\%$ of T_{pp}. Therefore, on THIN nodes, if we need to multiply two matrices on THIN, we should always use the first two libraries to get some more performance. To get the absolute best performance, it is preferable to use `MKL`.

Notice that this statement is true for both single and double precision, and for the whole range of matrix sizes we have analyzed.

These results shouldn't be surprising, considering that `MKL` is developed by Intel, and THIN nodes are Intel based. Therefore, we 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 for the same computations on EPYC nodes, which have a T_{pp} of 10.6 TFLOPS for SP and 5.3 TFLOPS for DP.

In this case, we obtain some results. None of the libraries are able to reach more than 20% of T_{pp}. Furthermore, for matrices of size ≤ 9000, `MKL` and `BLIS` outperform `OpenBLAS`. Between sizes 9000 and 12000, `MKL` performs best, and `OpenBLAS` begins to perform better than `BLIS`. For sizes ≥ 12000, `OpenBLAS` performs the best.

Again, we notice that none of the libraries are able to achieve more than 20% 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 dependant on the size of the matrices. However, for large matrices, we should use `OpenBLAS` while for smaller ones, we should use `MKL`.

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 multithrading, through OMP. For THIN nodes, we arrive to 24 cores, while for EPYC nodes, we arrive to 128 cores.

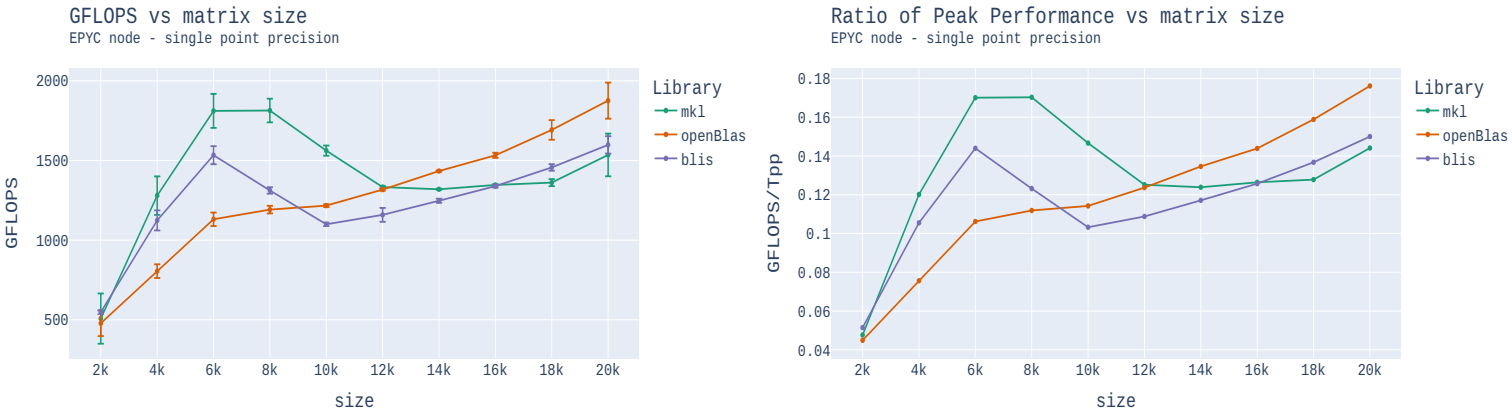

Figure 1.3: Results of SP matrix-matrix multiplication for EPYC nodes. We notice that asymptotically, `OpenBLAS` outperforms `MKL` and `BLIS`, while for small matrices, `OpenBLAS` performs the worst.

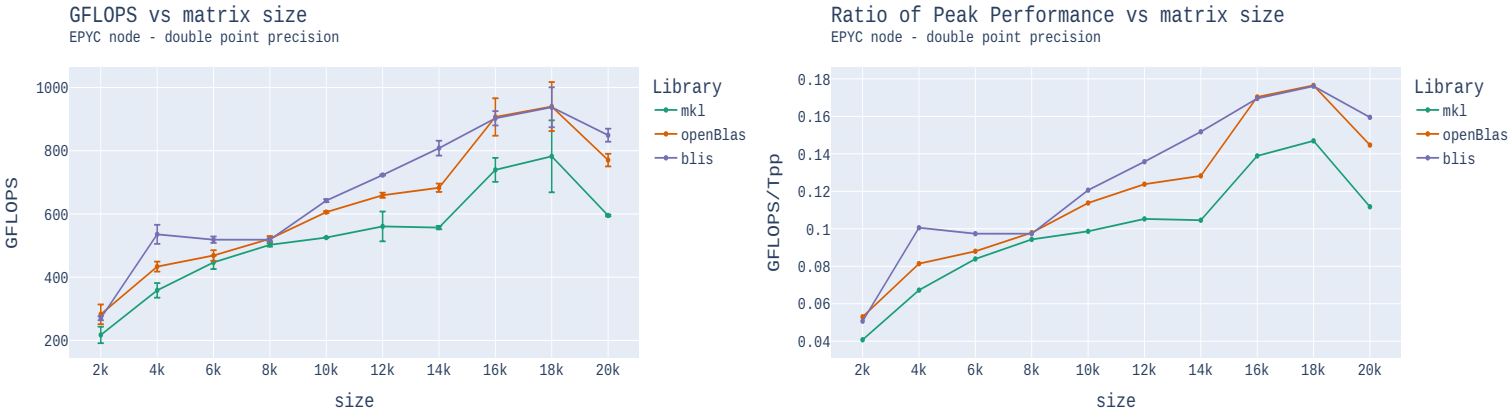

Figure 1.4: Results of DP matrix-matrix multiplication for EPYC nodes. `BLIS` outperforms `MKL` and `OpenBLAS`, for all matrix sizes.

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.

As mentioned above, we chose to always use `OMP_PLACES=cores` for these experiments. However, we used both `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`. In the first case, the threads will occupy first one entire socket, and then, when it is full, the other socket. In the second case, the threads will be placed as spread as possible, on two sockets. In both cases, when we use the full node, we expect the results to be the same.

THIN Nodes

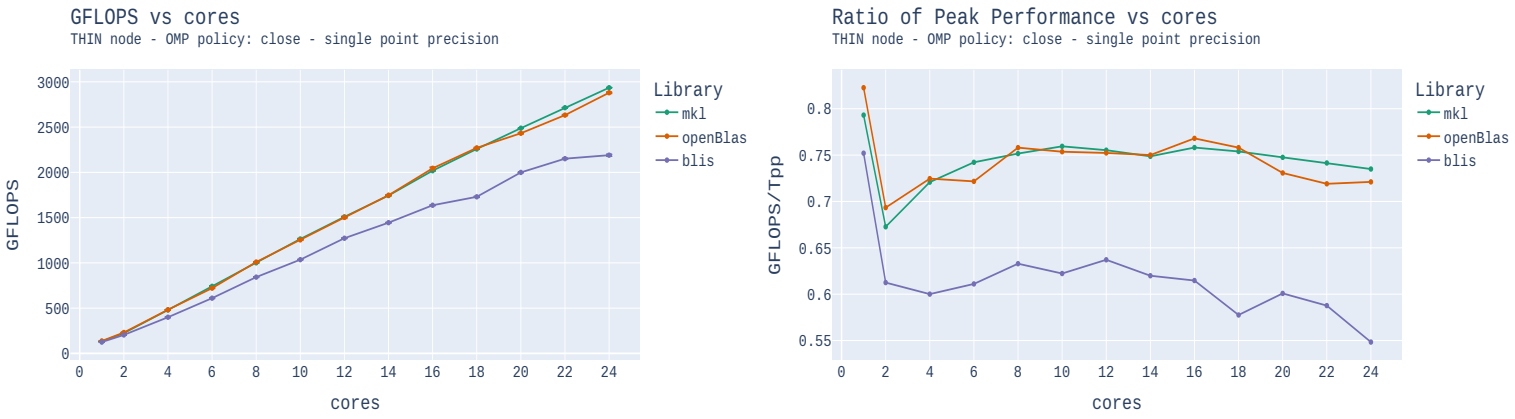

Figure 1.5: Results of SP matrix-matrix multiplication as the number of cores increase, using close policy.

`MKL` and `OpenBLAS` are able to maintain $\sim 75\%$ of T_{pp} from 6 cores and onwards. On the other hand, `BLIS` is able to achieve $\sim 60\%$ of T_{pp}, although with a lot of cores, this figure decreases to 55%. These numbers are consistent with the results that we obtained previously.

Furthermore, 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 and must share resources, such as the higher level caches.

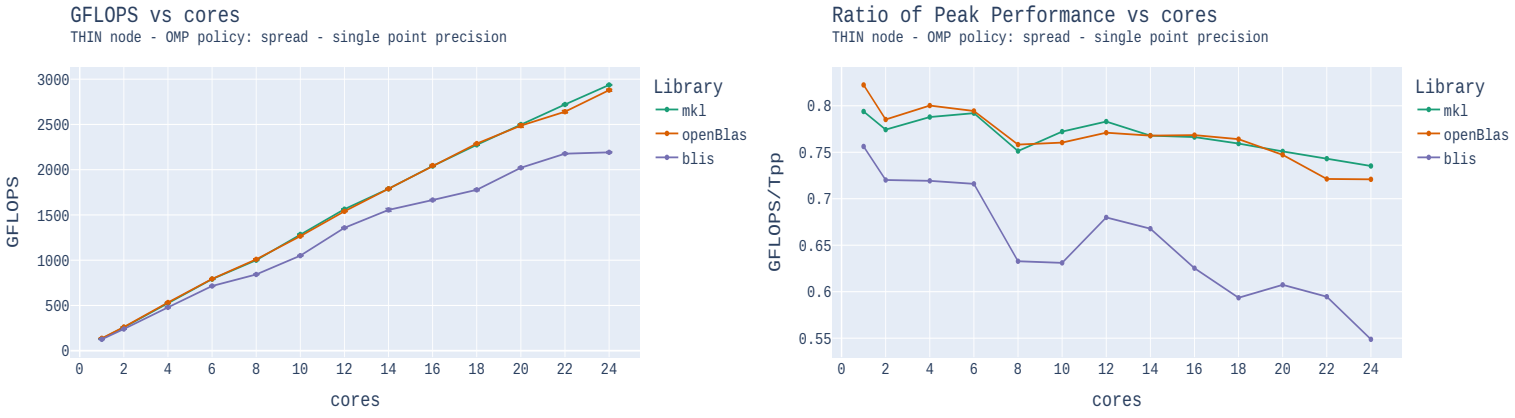

Figure 1.6: Results of SP matrix-matrix multiplication as the number of cores increase, using spread policy.

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 sharing from the beginning.

This highlights the importance of using the correct mapping policy to obtain better performance.

Now we briefly analyze the results obtained for double precision.

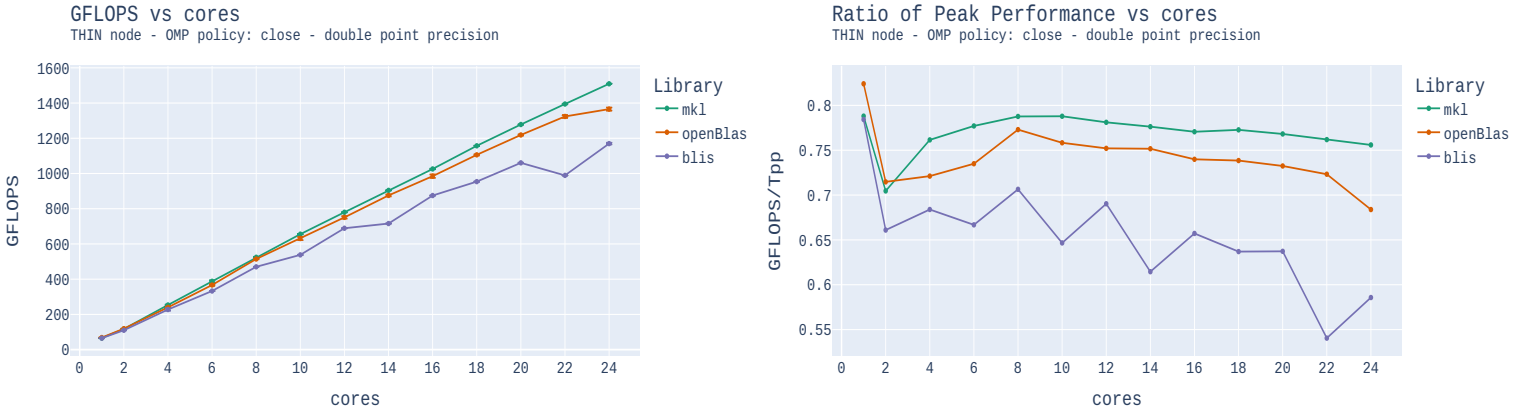

Figure 1.7: Results of SP matrix-matrix multiplication as the number of cores increase, using close policy.

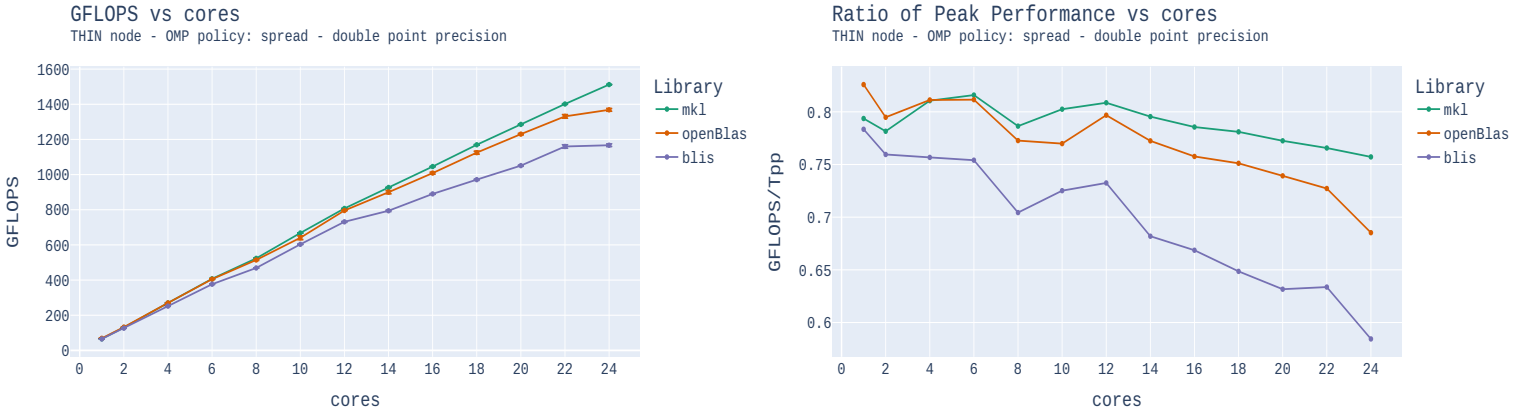

Figure 1.8: Results of SP matrix-matrix multiplication as the number of cores increase, using spread policy.

EPYC Nodes

1.4 Conclusion

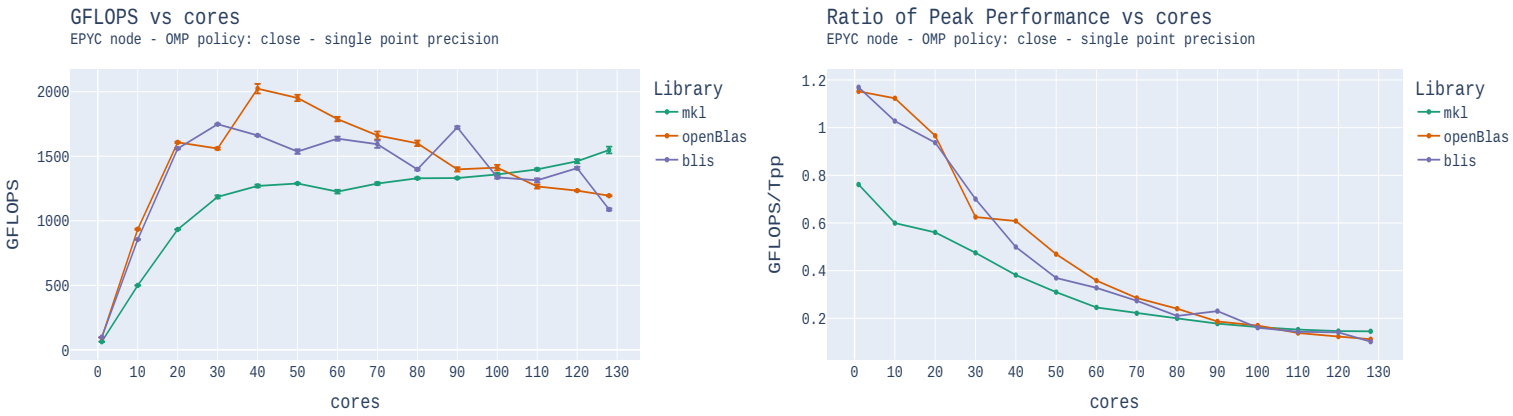

Figure 1.9:

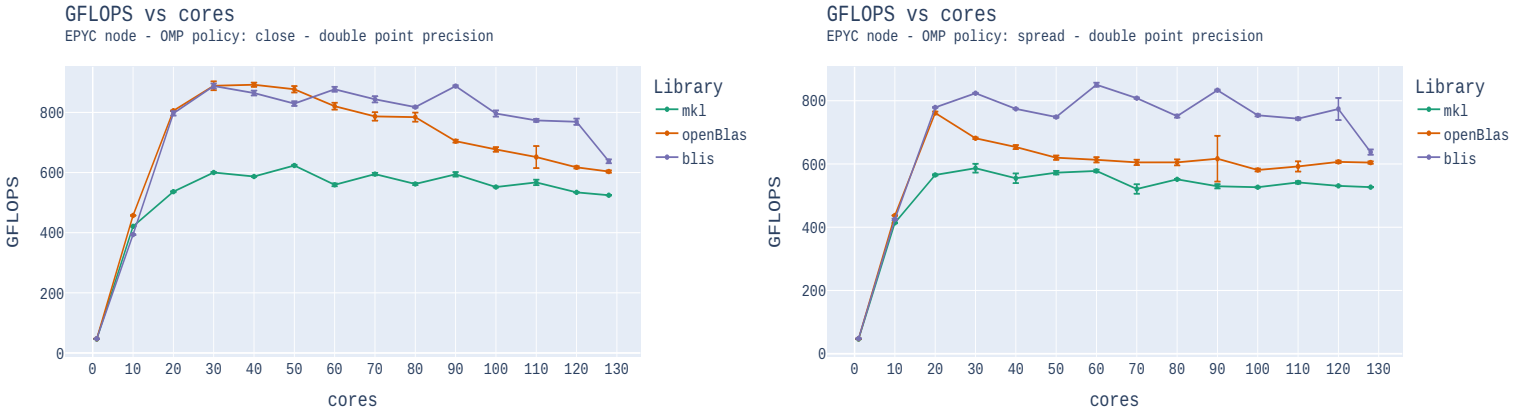

Figure 1.10:

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 can be "alive" or "dead".

The grid evolves over time by looking at a cell's $\mathcal{C}$ eight nearest neighbors and observing the following simple 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 less than two or more than three neighbors dies or stays dead by under or overpolulation, respectively (*death*).

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

Depending on how we update the cells, there exist two methods to evolve the grid: `static` and `ordered`. In `static` evolution, we freeze the state of the grid $\mathcal{G}_t$ at time step t, and compute $\mathcal{G}_{t+1}$ separately, while looking at $\mathcal{G}_t$.

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. In this scenario, the state of each cell depends on the 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 file.
2. Load a binary PGM file and evolve for n steps.
3. Save a snapshot during the course of evolution with frequency s ($s = 0$ means save at the end).
4. Support both `static` and `ordered` evolution.

Lastly, it must use both MPI[3] and OpenMP[4] to paralellize the computations and be able to process grids of considerably high dimensions.

2.2 Methodology

Since programs in MPI need to be rewritten completely from their serial counterparts, we must begin to conceptualize the problem in an encapsulated manner from the start.

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 (this has important consequences on the organizational paradigm we will use).

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

2.2.1 Problem 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 update their "part" of the whole grid.

For this, we need to decide how to split the grid since we can do both a 1D or 2D decomposition of our grid. It is known that a 2D decomposition is more efficient as it can exploit more bandwidth (more workers will send shorter messages at the same time). However, it is more complicated both from an implementation point of view, as well as it is worse for memory access.

Let's talk briefly about this second aspect. Although we will talk about our grid as a 2D array, internally, for efficiency, it will be represented as a 1D array. Therefore, each process will work on a strip of continuous data as shown in the figure below.

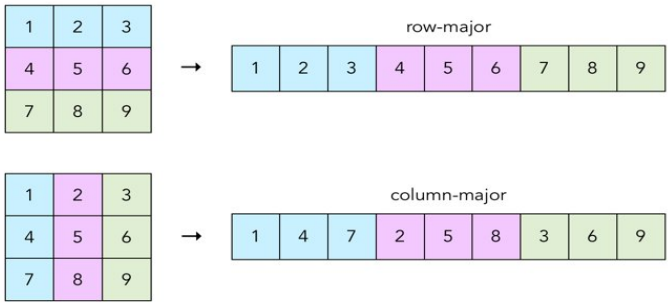

Figure 2.1: .

Fas one long array, in a 1D split, each process will need to work on a continuous "strip" of this memory.

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

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