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

Andres Bermeo Marinelli

September 22, 2023

Contents

Chapter 1

Benchmarking MKL, OpenBLAS, and BLIS

1.1 Introduction

In this exercise we compare the performance of three High Performance Computing 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 with 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 $2k$ to $20k$. In the second scenario, we fix the matrix size to $10k$ and increase the number of cores that `gemm` can use by modifying the `OMP_NUM_THREADS` environment variable. We start with one core and end with the maximum number available in the node.

In both scenarios, we repeat the measurements for both single and double precision, for both THIN and EPYC nodes.

Furthermore, for the second scenario, we also modify the thread allocation 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/lib auto
$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, which are 24 and 128, respectively.

With these commands, we compile 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 float or double precision, depending on the flag we use.

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.

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 $2k$ and end with a size of $20k$, with jumps of $2k$ 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 and double point precision.

The scripts that were used can be found in the folder `exercise2/scripts`.

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 allocation policies.

This will be done for the next section.

Furthermore, contrary to the guidelines of 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 begin 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 $10k$. 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. 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`.

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 using $\mathcal{C}$ cores:

$$T_{pp} = \mathcal{C} \times \text{clock freq.} \times \text{IPC} \tag{1.1}$$

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 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, namely 24 for THIN and 128 for EPYC, and we slowly increase the size of the matrices from $m = 2k$ to $m = 20k$.

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

THIN Nodes

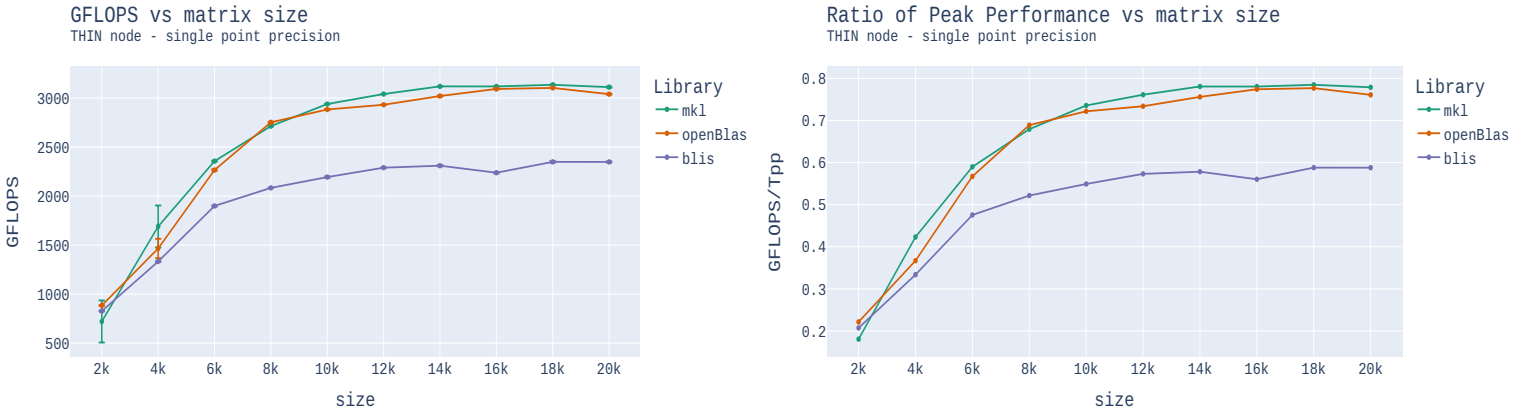

Figure 1.1: Results of SP matrix-matrix multiplication for THIN nodes. `MKL` and `OpenBLAS` perform similarly, outperforming `BLIS` for all matrix sizes.

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 $20k$.

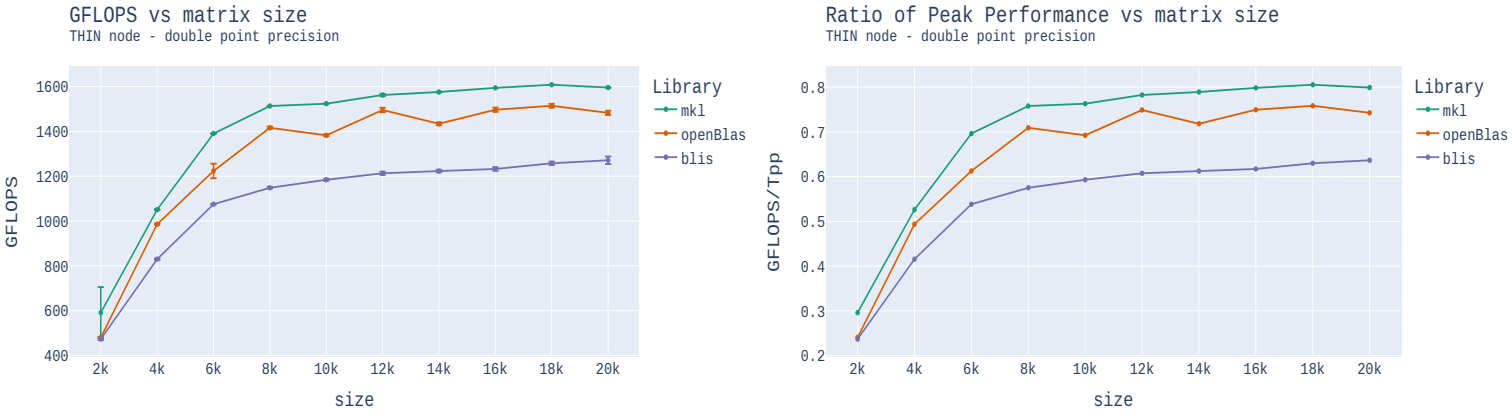

Figure 1.2: Results of DP matrix-matrix multiplication for THIN nodes. `MKL` performs the best, slightly above `OpenBLAS`. Both outperform `BLIS` for all matrix sizes.

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 better than the their single precision counterparts. For example, looking at the plot for single precision, for a matrix size of $4k$, 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 SP precision.

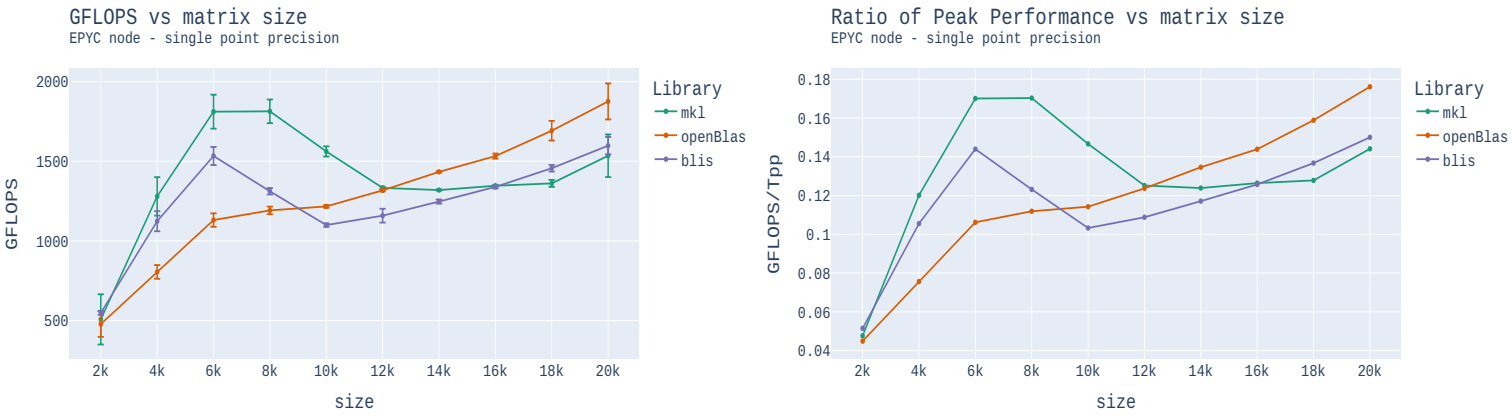

Figure 1.3: Results of SP matrix-matrix multiplication for EPYC nodes. We notice that asymptotically, `OpenBLAS` outperforms `MKL` and `BLIS`, while for small matrices, `OpenBLAS` performs the worst.

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 use bigger matrices.

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

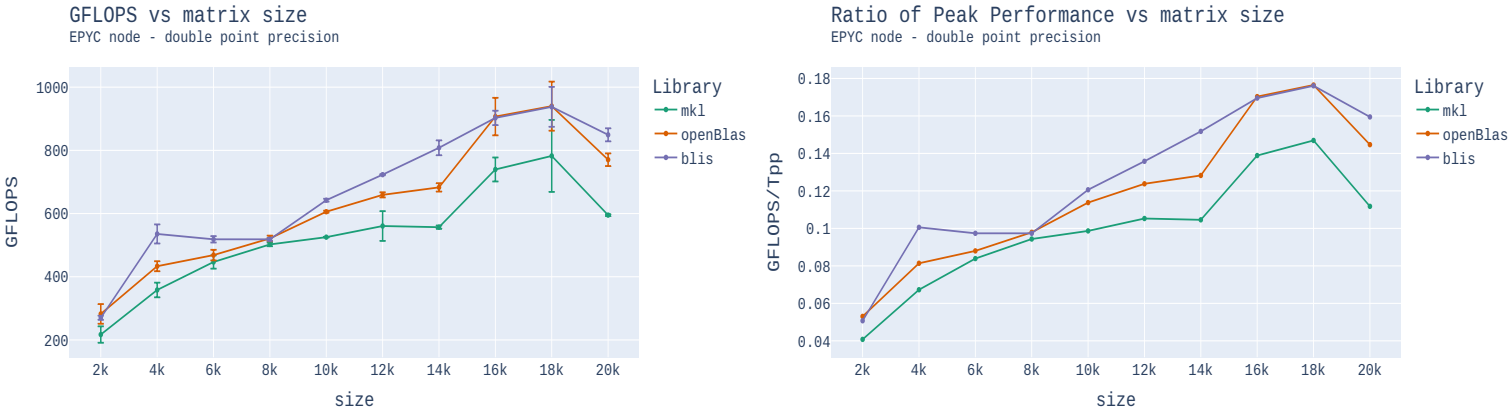

Figure 1.4: Results of DP matrix-matrix multiplication for EPYC nodes. `BLIS` outperforms `MKL` and `OpenBLAS` for all matrix sizes.

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 bigger matrices. So if we are multiplying matrices of size up to $20k$, it is 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 $10k$ 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} for the cores being used. In other words, in equation 1.1, we substitute the number of cores we are using for that calculation.

We first analyze THIN and then EPYC nodes.

THIN Nodes

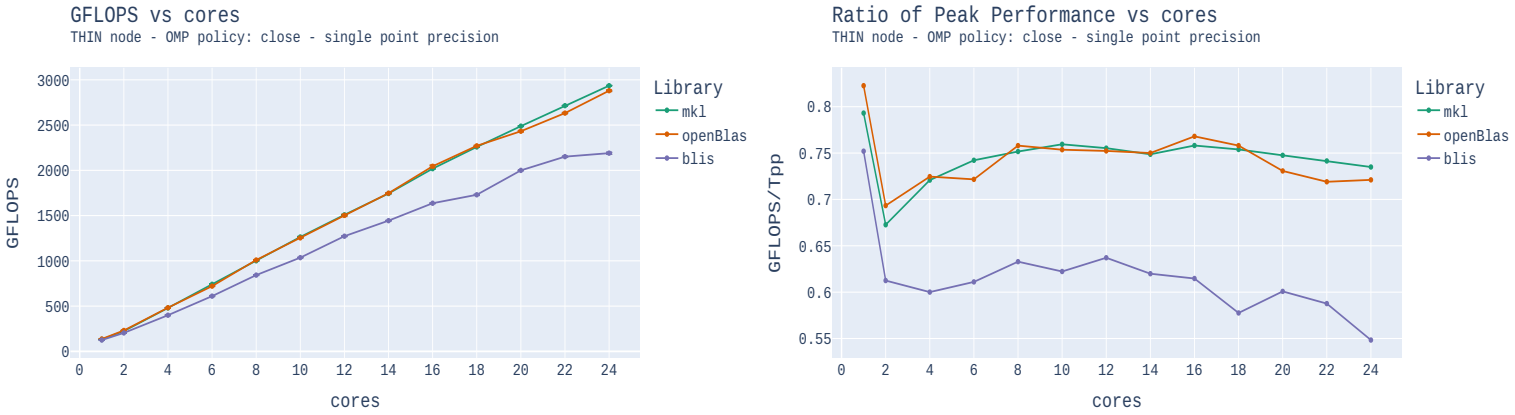

Figure 1.5: Results of SP matrix-matrix multiplication as the number of cores increase, using OMP close policy. `MKL` and `OpenBLAS` obtain the best performance.

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, with a close policy, both cores are mapped to the same socket, and must share resources, causing some contention.

Furthermore, 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.

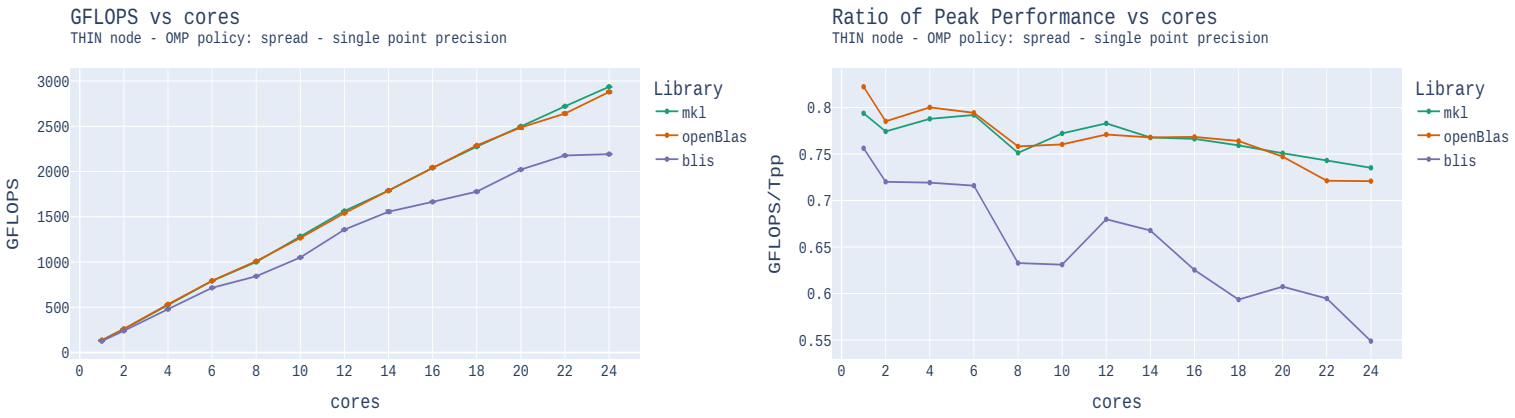

Figure 1.6: Results of SP matrix-matrix multiplication as the number of cores increase, using OMP spread policy. `MKL` and `OpenBLAS` are the best perfomers.

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 allocation policy to obtain better performance.

Now we briefly analyze the results obtained for double precision.

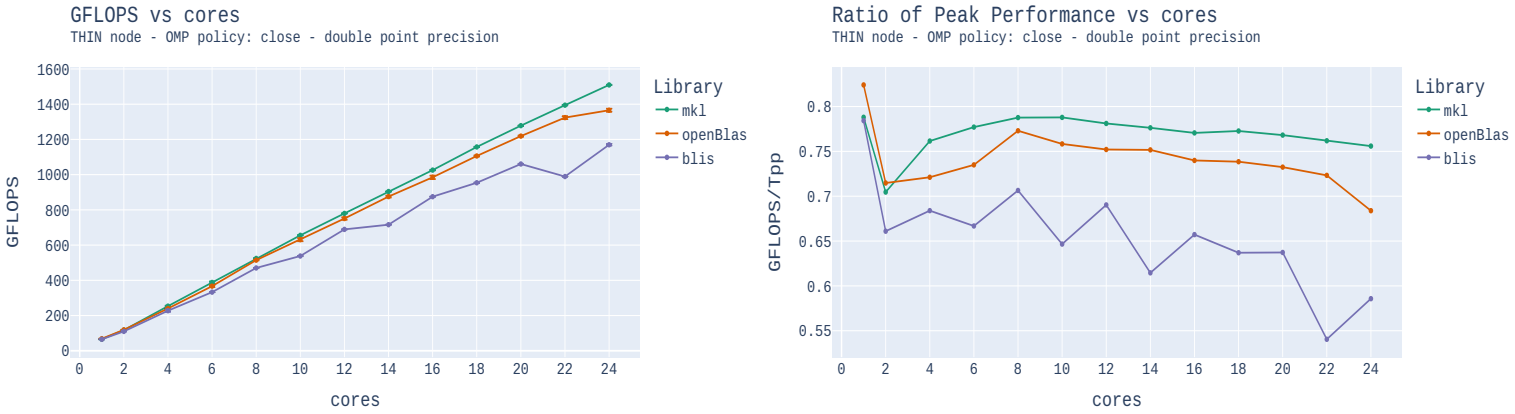

Figure 1.7: Results of DP matrix-matrix multiplication as the number of cores increase, using close policy. `MKL` and `OpenBLAS` perform the best.

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.

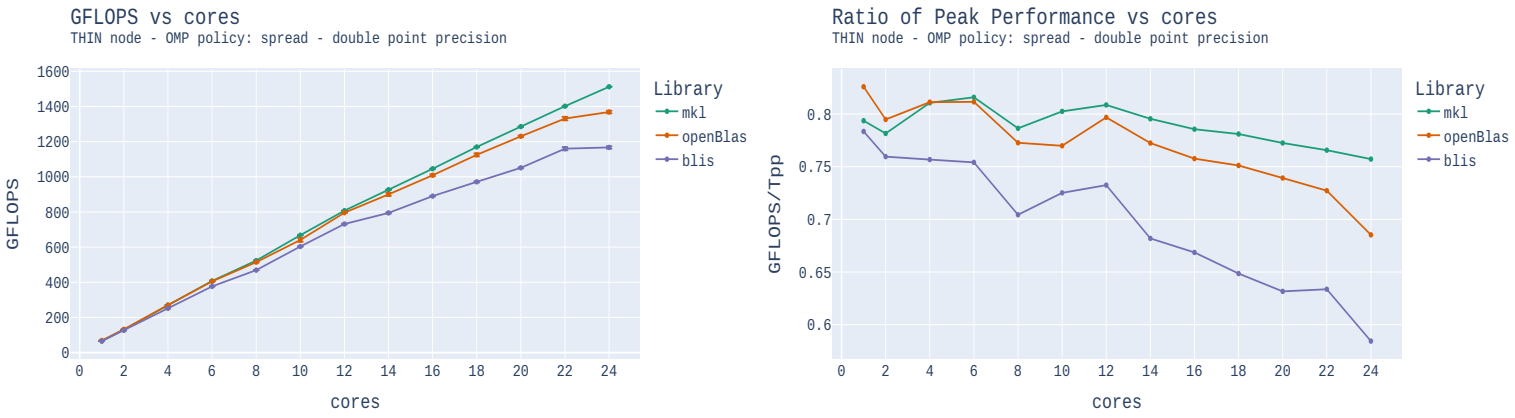

Figure 1.8: Results of DP matrix-matrix multiplication as the number of cores increase, using spread policy.

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.

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 in the earlier exercise.

Furthermore, since we obtained low performance with matrices up to $20k$, using all 128 cores, we don't expect an improvement. What may happen however, is that when we use less cores, the performance will be better compared to the theoretical peak performance (per number of cores this time).

We begin by analyzing the SP case with close policy.

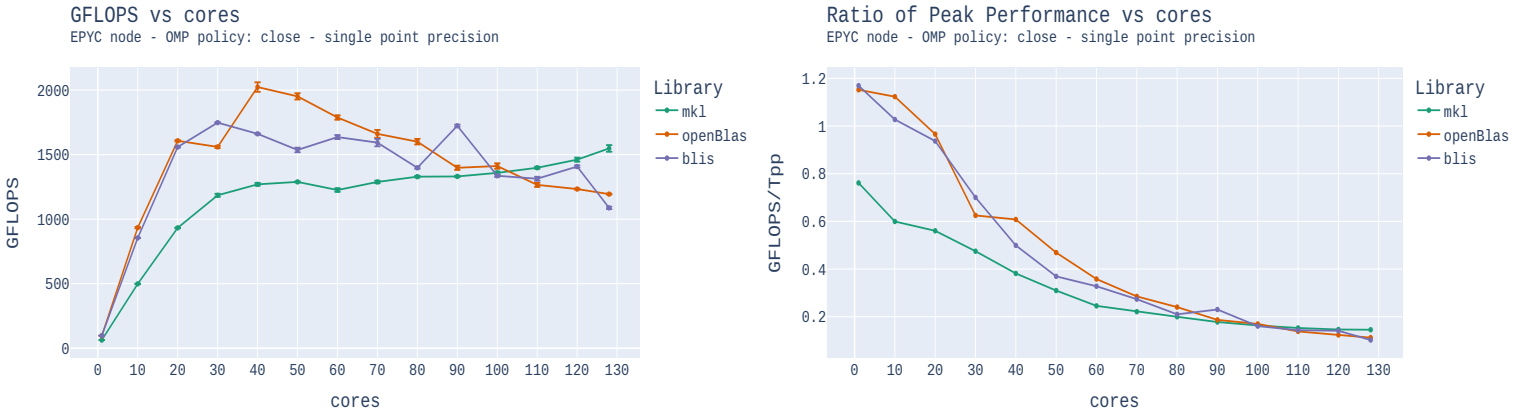

Figure 1.9: Results of SP matrix-matrix multiplication as the number of cores increase, using close policy. The best performance is obtained by `OpenBLAS` with 40 cores.

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 is almost on part with T_{pp}. This tells us that to fully exploit the resources we use, with a matrices of size $10k$, 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 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 slightly 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.

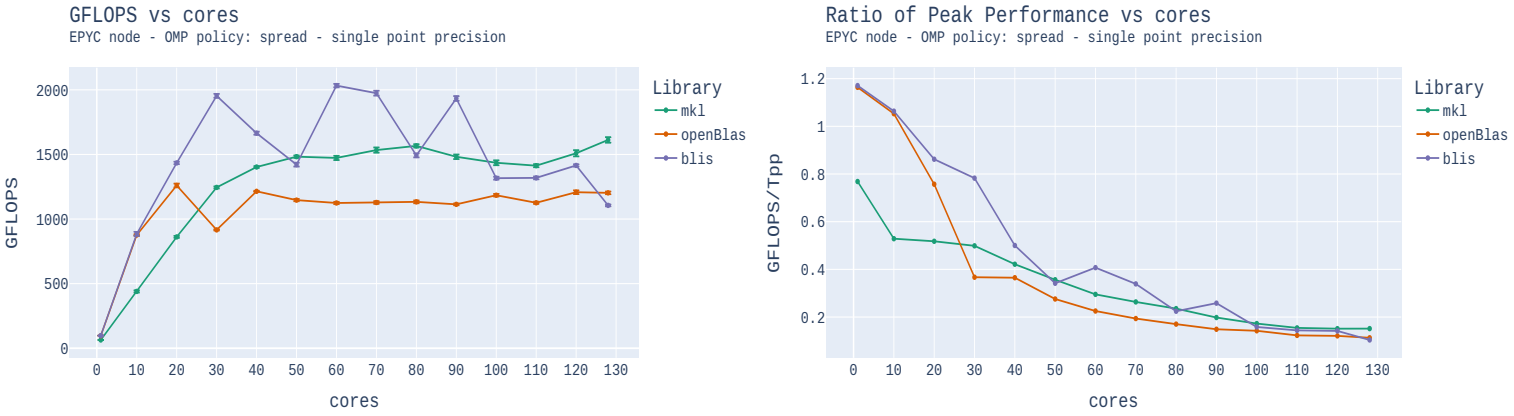

Figure 1.10: Results of SP matrix-matrix multiplication as the number of cores increase, using spread policy.

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. Once again, we notice that as we use more cores, we get worse performance.

Now we look at the DP case with close policy.

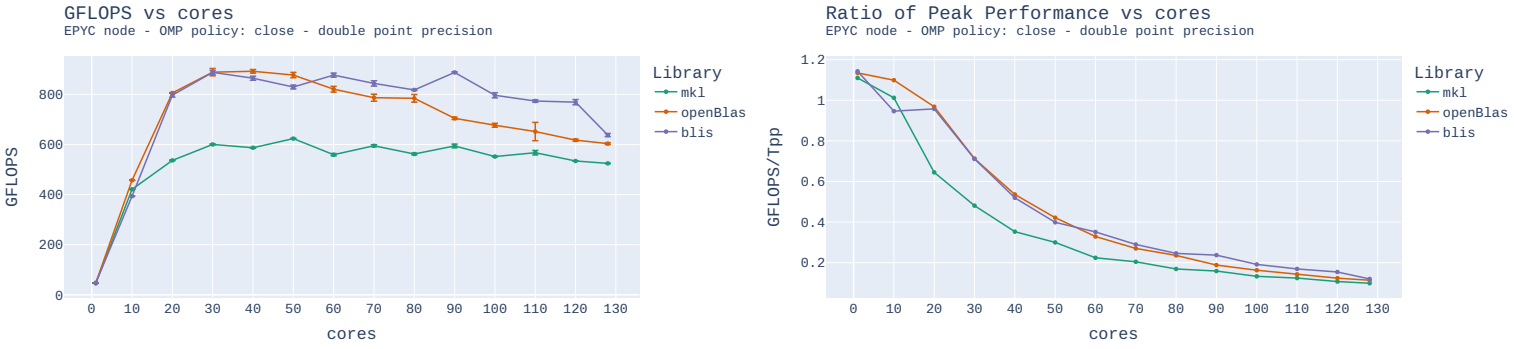

Figure 1.11: Results of DP matrix-matrix multiplication as the number of cores increase, using close policy. The best performance is obtained by `BLIS`, followed by `OpenBLAS`.

For double precision, using a close policy, we observe that `BLIS` performs the best for cores ≥ 60, while cores ≤ 60, it performs close to `OpenBLAS`, which is the best. `MKL` peforms the worst.

Finally, we analyze the DP case with spread policy.

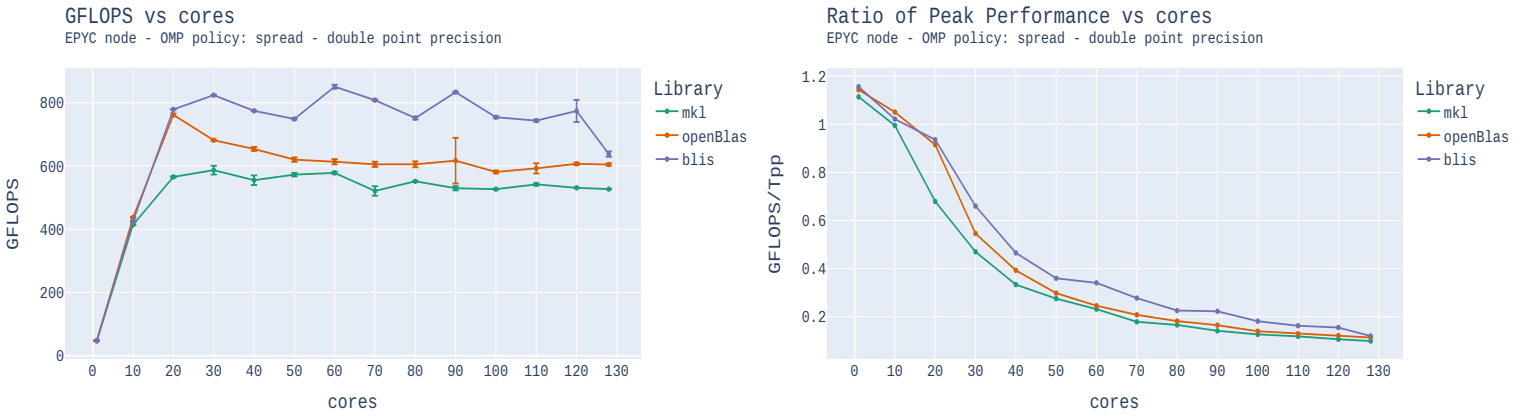

Figure 1.12: Results of DP matrix-matrix multiplication as the number of cores increase, using spread policy.

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 Conclusions

After analyzing the results, we can state multiple 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 thread allocation policy we use.

We find that for matrices up to $20k$, THIN nodes are able to achieve $\sim 75\%$ of T_{pp} with `MKL` and `OpenBLAS`. However, such matrix dimensions are too small to be able to fully exploit an EPYC node. In fact, in all of our experiments, we were unable to obtain more than 20% of the T_{pp} of the full EPYC 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 $50k$ or bigger, 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 live or dead 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.

Using our implementation, we will perform a study of the scalability of our code in three cases:

1. Strong OMP scalability: Using a grid of fixed dimensions and fixing the number of MPI processes to one per socket, show the run-time behaviour as the number of OMP threads per process is slowly increased.
2. Strong MPI scalability: Using a grid of fixed dimensions and fixing one OMP thread per process, show the run-time behaviour as the number of MPI tasks is increased across as many nodes as possible.

3. Weak MPI scalability: Starting with an initial grid size, show the run-time behaviour when scaling up from one socket, saturated with OpenMP threads, up to as many sockets as possible, keeping fixed the workload per MPI task.

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 more abstract 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 with paralellization in mind from the start, we must begin to conceptualize the problem in an encapsulated manner from the beginning.

We must make two important choices:

1. How we will decompose the problem.
2. How we will perform the IO.
3. How to mix MPI and OMP.

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 that we will 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. Finally, the third solution will determine the effectiveness of our hybrid code.

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 obvious form of paralellism is for each MPI process to work on a "part" of the whole grid. This is an example of `domain` decomposition, and it is shown below:

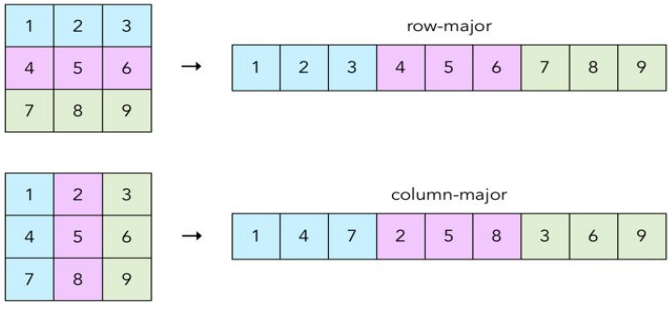

Figure 2.1: Decomposition of an array among three processes. Although the array is conceptualized as a 2D structure, internally, for efficiency, it is represented as a 1D contigous block of memory.

However, we need to decide how to split the grid. We can do a 1D decomposition by "stripes", as shown in figure 2.1 or a 2D decomposition by "blocks", as shown in figure 2.2.

It is known that 2D decomposition is more efficient. 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 strip of data which is 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. This can be visualized in the following image:

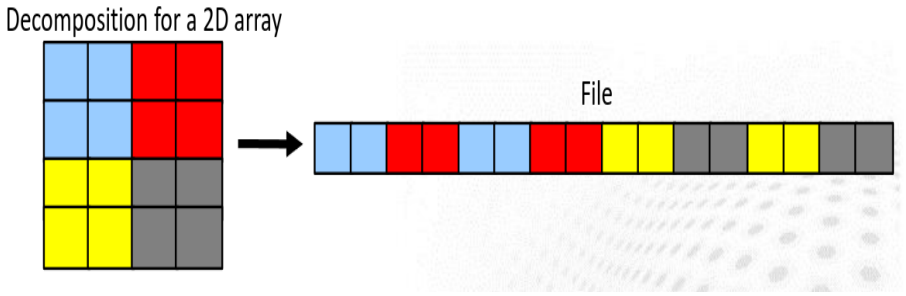

Figure 2.2: 2D decomposition of a grid. As we can see, each process needs to deal with non-contiguous memory. Image courtesy of [7].

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

Lastly, we must consider how to handle "halo" regions. Since each process will work on a strip 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 belongs to other processes. Therefore, we need to allocate some extra space for these additional rows, and set up message passing between processes to handle the exchange of these boundary rows, also known as `halo` regions.

Below we show the idea:

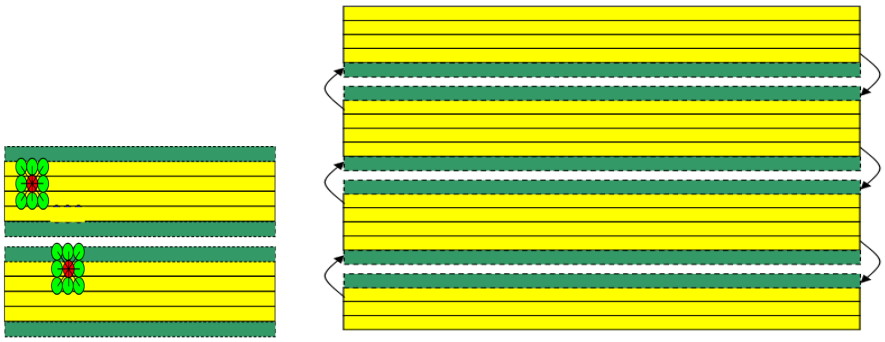

Figure 2.3: On the figure in the left, we can see that we can process cells that are internal with no problem. However, at the boundary we need information that belongs to another process. Therefore, we allocate some extra space and do a halo exchange as shown in the figure on the right.

Finally, in the case of `ordered` evolution, we cannot update each cell independently, since we modify the elements inplace in an ordered fashion. Therefore, we cannot parallelize this process. The only thing that we can do is use MPI to be able to process larger grids than we could in a conventional computer. In this case, each process will compute its part of the grid in order, rather than in parallel.

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 most naive 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.

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. The MPI implementation will take care of exploiting the parallel filesystem and reducing the amount of read and write operations. This is very advantageous for scalability since many processes can contemporarily write in different physical locations, while still having a unified view of the file. We note that this is a form of `functional` decomposition.

2.2.3 Mixing MPI and OMP

Now that we have discussed at an abstract level how to distribute the workload among the MPI processes, we need to think about how to integrate OMP to obtain a hybrid implementation.

In `static` evolution, since each process can update its part of the grid independently, the most natural approach is to use OMP to parallelize this computation. For this, there are a few options.

The simplest option is that each MPI process will spawn a parallel region at *each* iteration t to accelerate its computations.

Another, more advanced approach, is to use `MPI_THREAD_MULTIPLE`. In this mode, each OMP thread can make an MPI call contemporarily. This mode must be handled with care but it will enable us to create a single parallel region.

We implemented both cases to do some comparisons.

Finally, for `ordered` evolution, we cannot use OMP, because the update depends on the order which is intrinsically serial.

2.3 Implementation

In this section, we discuss the most technical details of our implementation, from the datatypes used, to compiler keywords, to optimization of branches, to memory allocation, to optimization of for loops. We will also describe how to use the code, what software stack is needed, and what tools were used.

Our implementation was done in C.

2.3.1 State representation

We begin by discussing how to represent data. Since each cell can be alive - 1 - or dead - 0 -, theoretically, we need a single bit to represent the state. This means that every byte of data can represent the state of 8 cells. However, this approach is complex since we need to do bit manipulation to access the information of each cell. As a result, the code will less readable and longer.

The next best idea, which we used, is to use the smallest datatype possible, i.e a `char`, which is one byte long, to represent the state of a single cell. In this way, we are less memory efficient, but we gain in readability and ease of implementation. In particular, we use an `unsigned char` which is of the same size, however, in this way, the compiler can possibly optimize the operations since it doesn't need to keep track of the sign.

All of our implementations contain the following definitions:

```
1 #define DEAD 0
2 #define ALIVE 1
```

2.3.2 Distributing the grid among MPI processes

Given a $k \times k$ grid, we need to tell each MPI process how many rows it must deal with. To do this, we can divide the total number of rows by the number of processes - i.e the size of the MPI Communicator - and distribute the remainder among a subset of them.

This is done through a common trick:

```
    my_rows = total_rows/size + 1*(rank<(total_rows%size));
```

In this way, all processes will have an equal workload, except for some ranks which will have to process one more row.

2.3.3 Allocating data

Now that we have determined how many rows each process has to deal with, each rank must allocate the appropriate memory in heap to store its portion of the grid.

In `static` evolution, we need to keep two buffers, one for the current generation, and one for the next one. In `ordered` evolution, we only have one buffer, and we update each cell inplace as we traverse the array.

As mentioned previously, it is not efficient to allocate a 2D array, although it is more naturally aligned with the problem statement. This is because 2D arrays are not necessarily contiguous in memory and we also have an overhead of following pointers.

Therefore, we allocate 1D arrays, which are contiguous in memory and we use a macro to be able to access the array as if it were a 2D structure.

Lastly, for each process, we need to allocate two additional rows to accomodate the halo regions. Also, to optimize for cache access, we explicitly align the data to the cache line size of THIN and EPYC nodes, which is 64 Bytes.

This can be seen in the following snippet:

```
// defining cache line size
#define CACHE_LINE_SIZE 64
// macro to access data in 2D fashion
#define DATA(i,j) (data[(i)*cols + (j)])

...

// aligned allocation
data = (unsigned char *) aligned_alloc(CACHE_LINE_SIZE, (my_rows + 2) * cols);
```

Of course, for `static` evolution, we perform two such allocations.

2.3.4 PGM Files - Header

Now that we have discussed how we perform the allocation, we briefly discuss PGM binary files.

PGM are portable binary files composed of a header and of binary data. In our case, the header is the following line:

```
P5 {rows} {cols} {maxval}\n
```

The `P5` is a magic number that establishes we are using a PGM file with binary data, where each byte corresponds to a pixel in the image. The color is interpreted according to the `maxval` value. Without going into too much detail, by setting `maxval` to one, we are establishing that 1's will be drawn as white (alive) and 0's will be drawn as black (dead).

Whenever we save a snapshot of the evolution, we need to write this header. This will be done by one single process - rank 0.

Also, as we will see later, to use MPI IO, we need to find out the length of this header in bytes. All these operations are performed in the following snippet:

```
#define HEADER_FORMAT_STRING "P5 %lf %lf %d\n"
#define MAX_VAL 1

...

// We get the header size
```

```
unsigned long int header_size = snprintf(NULL, 0, HEADER_FORMAT_STRING, rows, cols,
    MAX_VAL);

// allocate the string
char * header = malloc(header_size + 1)

// write the header into the string
sprintf(header, HEADER_FORMAT_STRING, rows, cols, MAX_VAL);
```

We will use the header size in MPI IO in the next section.

2.3.5 MPI IO

As discussed above, MPI IO is the best choice to handle the IO when we are dealing with many processes. It takes advantage of the parallel file system to write in multiple locations at once, while keeping a logically unified view of the file. In this way, we avoid the bottleneck of having to send the grid back to the master or overloading the filesystem with too many posix calls.

Below we list the code snippet that we used and then discuss the relevant details.

```

void save_grid(char * restrict fname, MPI_Comm comm, int rank, char * restrict header,
     unsigned long int header_size, MPI_Offset offset, unsigned char * restrict data,
    unsigned long int my_rows, unsigned long int cols)
{

    MPI_File fh;

    // Opening the file in MPI_MODE_WRITEONLY or MPI_MODE_CREATE_ONLY
    const int err = MPI_File_open(comm, fname, MPI_MODE_WRONLY | MPI_MODE_CREATE,
    MPI_INFO_NULL, &fh);

    // Check that the file was opened correctly
    if(err != MPI_SUCCESS)
    {
        fprintf(stderr, "Error opening %s\n", fname);
        MPI_Abort(MPI_COMM_WORLD, err);
    }

    if(rank == 0)
    {
        // we specify the file handle, the byte where we write, what we write,
        // the length of what we write, the data type of what we write, and the
        // status of the writing.
        MPI_File_write_at(fh, 0, header, header_size, MPI_CHAR, MPI_STATUS_IGNORE);
    }

    // collective write operation called by all processes
    MPI_File_write_at_all(fh, offset, data + cols, my_rows*cols, MPI_CHAR,
    MPI_STATUS_IGNORE);

    MPI_File_close(&fh);
}
```

First, all processes call the `MPI_File_open()` routine to open the file. The file is opened in `MPI_MODE_WRONLY` for write only and `MPI_MODE_CREATE` to create the file if it does not exist already.

Then, we choose rank 0 to write the header of the PGM file. We use the MPI function `MPI_File_write_at()` which allows us to specify at what position in the file, in bytes, the data must be written.

For the header, we will write at the beginning of the file, so at byte 0. The rest of the arguments are explained in the snippet.

Finally, for writing the grid, we will use the collective version of `MPI_File_write_at()`, namely `MPI_File_write_at_all()`. This function is called by all the processes. Internally, the MPI implementation will perform this operation in the most efficient way possible.

The arguments are the same as the non-collective version.

The only thing we need to explain, is how we find the byte offset for each process. We first find the local row offset - i.e. which row to start writing from - through the following function:

```
unsigned long int get_my_row_offset(unsigned long int total_rows, int rank, int size)
{
```

```
    // returns the row offset from which the rank must start getting its rows

    // for example: for rank 0, the offset will be zero as it will read
    // my_rows from the start. However, rank 1 will have to read
    // after the rows that rank 0 handles (accounting for the remaineder),
    // and so on.

    // nrows is the interger division between total_rows and size
    unsigned long int nrows = total_rows/size;
    unsigned long int remainder = total_rows%size;

    // The idea is the following, if the division was pefect, we would do
    // nrows * rank. However, we still need to account for the remainder.
    // For n. remainder processes (starting to count from rank 1 because rank 0
    // just gets 0 as offset), we simply need to add the rank to the offset.
    // If we are dealing with a process id which is greater than the remainder,
    // then we simply need to add the whole remainder.

    unsigned long int my_offset = nrows*rank
        + rank*(((unsigned long int) rank) <= remainder)
        + remainder*(((unsigned long int) rank) > remainder);

    return my_offset;
}
```

We illustrate with a brief example. Suppose we have a 14×14 grid and four ranks. The interger division is 3 and the remainder is 2. Since the remainder of the division is 2, two ranks, 0 and 1, will need to handle an extra row.

So rank 0 and rank 1 will deal with $3+1=4$ rows, while ranks 2 and 3 will deal with 3 rows each.

Now, for the offset, rank 0 will just have offset 0. Rank 1 will need to shift by the interger division plus 1 to account for the extra row that rank 0 has, so 4. Similarly, rank 2 has to shift by twice the interger division, plus the extra row that rank 0 has and the extra row of rank 1, so 8.

Rank 3 will have to shift by three times the interget division, plus the extra rows of the previous ranks. However, this time, rank 2 did not have an extra row. So rank 3 just needs to account for *total* extra rows, which are 2.

This sounds complicated, but it is simple once one works through a few examples.

Since the offset for the MPI IO function calls are in bytes, to get the final offset, we multiply the row offset by the number of columns, and by the size of the datatype we use.

Then, we need to add the size of the header in bytes. This is shown in the snippet below.

```
...

const MPI_Offset header_offset = header_size * sizeof(char);

const MPI_Offset my_file_offset = my_row_offset * cols * sizeof(char);

const MPI_Offset my_total_file_offset = my_file_offset + header_offset;

...
```

Finally, reading the file is done in the same exact way, replacing `MPI_File_write_at_all` with `MPI_File_read_at_all`.

However, we need to have an additional step to read the header, and this is done through a normal `fscanf()` call.

2.3.6 Updating Cells - Static Evolution

Now we discuss how we perform the cell update for `static` evolution.

The entire evolution process is contained inside a for loop over time steps. At each iteration, we have to use message passing to exchange the halo rows. This is done using non-blocking operations to hide the latency of the communication.

The following snippet shows how this is done:

```
// we post send and recv requests for two halo regions.
// matrix:
// ---------------------  row 0 (HALO)
// 01010011001100100100  row 1
// 01001000010010011111  row 2              --
// ...                                        | these rows dont need
// 00101011111001100001  row my_rows - 1    __| halo regions
// 01001001000010010100  row my_rows
// ---------------------  row my_rows + 1 (HALO)

// send row: 1 to the previous rank
MPI_Isend(data_prev + cols, cols, MPI_CHAR, prev, prev_tag, MPI_COMM_WORLD, &
    prev_send_request);
// send row: my_rows to next rank
MPI_Isend(data_prev + my_rows_x_cols, cols, MPI_CHAR, next, next_tag, MPI_COMM_WORLD,
    &next_send_request);

// receive from previous rank and put in row 0 (halo region)
MPI_Irecv(data_prev, cols, MPI_CHAR, prev, next_tag, MPI_COMM_WORLD, &
    prev_recv_request);
// receive from next rank and put in row my_rows + 1 (halo region)
MPI_Irecv(data_prev + my_rows_x_cols_p_cols, cols, MPI_CHAR, next, prev_tag,
    MPI_COMM_WORLD, &next_recv_request);
```

Where `data_prev` and `data` are the buffers for the previous and next generation, respectively.

Next, while messages are being exchanged, each process can start to work on the internal portion of its grid, since it doesn't need the halo rows.

Two versions were used to do this: one using a double for loop over rows and columns, and the other using a single for loop over cells.

```
...

// using double for loop
// note we start from row 2 and finish at row my_rows - 1
for(unsigned long int row = 2; row < my_rows; ++row)
{
    for(unsigned long int col = 0; col < cols; ++col)
    {
        upgrade_cell_static(data_prev, data, row, col);
    }
}

...
```

```
...

// using single for loop
for(unsigned long int cell = 2*cols; cell< my_rows*cols; ++cell)
{
        unsigned long int i = cell/cols;
        unsigned long int j = cell - i*cols;
        upgrade_cell_static(data_prev, data, i, j);
}

...
```

Finally, at this point, there is nothing left to do but wait until we receive the halo rows.

```
MPI_Wait(&prev_recv_request, MPI_STATUS_IGNORE);
MPI_Wait(&next_recv_request, MPI_STATUS_IGNORE);
MPI_Request_free(&prev_send_request);
MPI_Request_free(&next_send_request);
```

Then, we process the first and last row of the grid.

```
for(unsigned long int col=0; col<cols; ++col)
{
    upgrade_cell_static(data_prev, data, 1, col);
}
// we only do this if we have more than one row
```

```
for(unsigned long int col=0; col<cols*(my_rows>1); ++col)
{
    upgrade_cell_static(data_prev, data, my_rows, col);
}
```

Finally, if we need to save the grid, we save using MPI IO, and then we swap the pointers and repeat.

Apart from a few differences that we introduce with OMP later, these are the fundamental steps of our evolution.

The last thing is to discuss is the update function itself:

```
void upgrade_cell_static(unsigned char * restrict data_prev, unsigned char * restrict
    data, unsigned long int i, unsigned long int j)
{
    DATA(i,j) = DEAD;

    // we use ternary operators to avoid branches
    // as well as the register keyword to signal to the compiler to
    // keep this data close

    // compute coordinates of neighbors with wrap around of dimensions
    register unsigned long int jm1 = j==0 ? cols-1 : j-1;
    register unsigned long int jp1 = j==(cols-1) ? 0 : j+1;
    // for rows, wrapping is ensured by the presence of halos
    register unsigned long int im1 = i-1;
    register unsigned long int ip1 = i+1;

    // we fetch the neighbors with 8 different instructions to saturate pipelines.
    // We access first the neighbors that are already most likely already
    // in cache due to previous updates.
    unsigned char tmp0=DATA_PREV(im1, jm1);
    unsigned char tmp3=DATA_PREV(i,jm1);
    unsigned char tmp5=DATA_PREV(ip1,jm1);

    unsigned char tmp1=DATA_PREV(im1,j);
    unsigned char tmp2=DATA_PREV(im1,jp1);

    unsigned char tmp4=DATA_PREV(i,jp1);

    unsigned char tmp6=DATA_PREV(ip1,j);
    unsigned char tmp7=DATA_PREV(ip1,jp1);

    unsigned char n_alive_cells = tmp0+tmp1+tmp2+tmp3+tmp4+tmp5+tmp6+tmp7;

    DATA(i,j) = ALIVE*(n_alive_cells==3) + DATA_PREV(i,j)*(n_alive_cells==2);
}
```

The function is well commented and explains some of the optimization tricks we used.

First, we set the cell to be dead regardless, since if there are less than two or more than 3 live neighbors, it dies.

Then, we calculate the wrapped coordinates of the neighbors using ternary operators to avoid conditional branches. We use separate instructions to fetch each neighbor to encourage pipeline saturation.

We aggregate the state of all the neighbors, and apply the rules. This version of the rules may seem strange at first, but it is completely equivalent to the one we described, although in this way, it's a bit more efficient.

Finally, notice the `restrict` keyword for the two pointers in the declaration of the function. This is done so the compiler can optimize better since it knows it will never encounter memory aliasing.

2.3.7 Updating Cells - Ordered Evolution

For `ordered` evolution, the idea is more complex.

We need to make sure that ranks process their parts of the grid in an ordered fashion. We first show the code snippet and then comment:

```
// for ordered evolution, we cannot parallelize, and each process
// must work on their part of the grid in serial order.

// first, each process posts a non blocking receive operation for
// the top halo row and the bottom halo row.
// Since we have ordered evolution each process will then wait
// until the top halo row has been received to start working on
// its part of the grid.
MPI_Irecv(data, cols, MPI_CHAR, prev, next_tag, MPI_COMM_WORLD, &prev_recv_request);
MPI_Irecv(data + my_rows_x_cols_p_cols, cols, MPI_CHAR, next, prev_tag, MPI_COMM_WORLD
    , &next_recv_request);

// however, each process will also have to send its row 1
// (the bottom halo for the previous process) which is necessary
// for the previous process to process the end of its grid.
// This excludes rank 0, because it needs to send its first
// row to rank (size-1) only AFTER it has been updated.
if(rank != 0){ MPI_Isend(data + cols, cols, MPI_CHAR, prev, prev_tag, MPI_COMM_WORLD,
    &prev_send_request);}

// To get things started for rank 0, rank (size - 1) sends the first forward message
// of its last row as a top halo.
if(rank == size-1) { MPI_Isend(data + my_rows_x_cols, cols, MPI_CHAR, next, next_tag,
    MPI_COMM_WORLD, &next_send_request); }

// All processes will wait until receive of top halo row is complete
// (since they need it to start ordered evolution)
//
// the first time, only process 0 should pass.
MPI_Wait(&prev_recv_request, MPI_STATUS_IGNORE);

// Since we have received the top halo row, we can process all
// data EXCEPT the last row - since we are not sure that we have
// received the bottom halo yet.
// So this for loop goes from 1 to my_rows - 1
for(unsigned long int row = 1; row < my_rows; ++row)
{
    for(unsigned long int col = 0; col < cols; ++col)
    {
        upgrade_cell_ordered(data, row, col);
    }
}

// Based on how many rows we have, we behave differently
if(my_rows>1)
{
    // in this case, the for loop above has been executed and
    // now that the first row has been processed, rank 0 can send it
    // to rank size-1
    if(rank == 0){ MPI_Isend(data + cols, cols, MPI_CHAR, prev, prev_tag,
    MPI_COMM_WORLD, &prev_send_request);}

    // We have finished processing all our grid except for last row.
    // For this, we need to make sure that bottom halo has been received.

    // We wait for receive of bottom halo
    MPI_Wait(&next_recv_request, MPI_STATUS_IGNORE);

    // We process last row
    for(unsigned long int col=0; col<cols; ++col)
    {
        upgrade_cell_ordered(data, my_rows, col);
    }

    // now that we have computed the last row, we need to send it
    // to the next process. This will be the top halo for the next
    // process, which in turn will finally pass the MPI_Wait() statement
    // above.
    // All except process size-1 because this will be taken care of in the
    // next iteration of the for loop
    if(rank<(size-1)){ MPI_Isend(data + my_rows_x_cols, cols, MPI_CHAR, next, next_tag
```

```
    , MPI_COMM_WORLD, &next_send_request);}

}
else
{
    // in this case we have skipped the for loop
    // we need to wait to receive the bottom halo from the next rank
    // since we need it to process our only row.
    MPI_Wait(&next_recv_request, MPI_STATUS_IGNORE);

    // Once we receive, we can process row 1
    for(unsigned long int col=0; col<cols; ++col)
    {
        upgrade_cell_ordered(data, my_rows, col);
    }

    // rank 0 sends this row to rank size-1
    if(rank == 0){ MPI_Isend(data + cols, cols, MPI_CHAR, prev, prev_tag,
    MPI_COMM_WORLD, &prev_send_request);}

    // all ranks except last, send their row to the next rank
    if(rank<(size-1)){ MPI_Isend(data + my_rows_x_cols, cols, MPI_CHAR, next, next_tag
    , MPI_COMM_WORLD, &next_send_request);}
}

// we put a barrier since all processes need to wait before until the whole grid
// is processed before checking if they need to save it
MPI_Barrier(MPI_COMM_WORLD);

// Free the send operations
MPI_Request_free(&prev_send_request);
MPI_Request_free(&next_send_request);

// proceed to check if we need to save and swap pointers and repeat
```

The snippet is commented very thoroughly, however, we briefly explain the idea. To process its part of the grid, each rank needs to wait for the previous rank to process its last row, and send it over to be placed in top halo row.

Hence, by placing a wait request for the top halo row, we ensure that the ranks update the grid in the correct order, one at a time.

Furthermore, each rank also needs the bottom halo row to process its internal row. We can send this in the beginning with a non-blocking operation.

Finally, to get things started, rank size - 1 sends its last row to rank 0, which will be the first one to receive a top halo row and pass through the receive operation.

From there, we simply update rows 1 to (my_rows - 1), wait until we receive the bottom halo, process the last row, and then send it to the next rank so it can pass the wait request.

There is a small consideration to make in the case we have only one row, but conceptually, the procedure is the the same.

Lastly, we show the update function:

```
void upgrade_cell_ordered(unsigned char * data, unsigned long int i, unsigned long int
     j)
{
    register unsigned long int jm1 = j==0 ? cols-1 : j-1;
    register unsigned long int jp1 = j==(cols-1) ? 0 : j+1;
    register unsigned long int im1 = i-1;
    register unsigned long int ip1 = i+1;

    unsigned char tmp0=DATA(im1, jm1);
    unsigned char tmp3=DATA(i,jm1);
    unsigned char tmp5=DATA(ip1,jm1);

    unsigned char tmp1=DATA(im1,j);
    unsigned char tmp2=DATA(im1,jp1);

    unsigned char tmp4=DATA(i,jp1);

    unsigned char tmp6=DATA(ip1,j);
```

```
    unsigned char tmp7=DATA(ip1,jp1);

    register unsigned char n_alive_cells = tmp0+tmp1+tmp2+tmp3+tmp4+tmp5+tmp6+tmp7;

    DATA(i,j) = ALIVE*(n_alive_cells==3) + DATA(i,j)*(n_alive_cells==2);
}
```

The function is virtually the same as the static case, with the only difference that we operate on a single buffer.

2.3.8 OMP Integration - Hybrid V1

In this section, we tie everything together to show the final hybrid implementation. As mentioned before, we have two implementations for `static` evolution.

For the first implementation, we create only one parallel region `outside` of the for loop over the evolution time steps. Furthermore, we use MPI Multiple mode, where each thread can make an MPI call contemporarily, as illustrated in the following image:

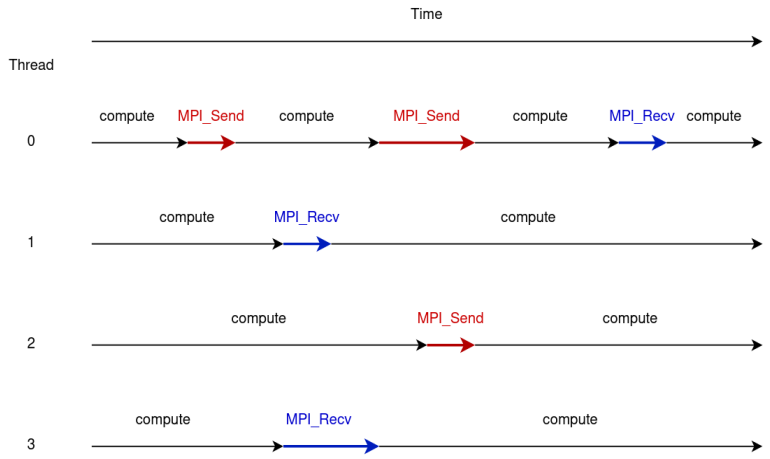

Figure 2.4: Visualization of MPI Multiple mode. Each OMP thread can make MPI calls.

The idea behind this is to avoid the creation of a parallel region every time we iterate over the time steps and have multiple threads make MPI calls contemporarily. To do this, we need to initialize MPI slightly differently, as shown below:

```
int provided_thread_level;
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided_thread_level);
if(provided_thread_level<MPI_THREAD_MULTIPLE)
{
    printf("Can't do thread multiple mode... Aborting");
    MPI_Finalize();
}
```

Then, the for loop over the evolution time steps looks like this:

```
// one parallel region
#pragma omp parallel
for(unsigned long int t = 1; t < n+1; ++t)
{

    // multiple threads make MPI calls

    // nowait eliminates barrier at the end
    #pragma omp single nowait
    MPI_Irecv(data_prev, cols, MPI_CHAR, prev, next_tag, MPI_COMM_WORLD, &
    prev_recv_request);

    #pragma omp single nowait
    MPI_Irecv(data_prev + my_rows_x_cols_p_cols, cols, MPI_CHAR, next, prev_tag,
    MPI_COMM_WORLD, &next_recv_request);
```

```
14
15      #pragma omp single nowait
16      MPI_Isend(data_prev + cols, cols, MPI_CHAR, prev, prev_tag, MPI_COMM_WORLD, &
        prev_send_request);
17
18      #pragma omp single nowait
19      MPI_Isend(data_prev + my_rows_x_cols, cols, MPI_CHAR, next, next_tag,
        MPI_COMM_WORLD, &next_send_request);
20
21      // remaining threads in the meantime start to work on internal part
22      // of grid. We need to keep the implied barrier at the end of the for loop
23      // because the next operation of waiting can only happen if send and recv
24      // have been posted
25      #pragma omp for schedule(dynamic, chunk)
26      for(unsigned long int cell = 2*cols; cell< my_rows_x_cols; ++cell)
27      {
28              unsigned long int i = cell/cols;
29              unsigned long int j = cell - i*cols;
30              upgrade_cell_static(data_prev, data, i, j);
31      }//barrier. Needed, otherwise, if threads havent finished posting
32       //send and receive, the next operation will be an error.
33
34       // we wait until we receive halo rows
35      #pragma omp single
36      {
37          MPI_Wait(&prev_recv_request, MPI_STATUS_IGNORE);
38          MPI_Wait(&next_recv_request, MPI_STATUS_IGNORE);
39          MPI_Request_free(&prev_send_request);
40          MPI_Request_free(&next_send_request);
41      } //barrier needed because we need to get halo cells to process
42        //next rows. Now we can process the first and last row.
43
44      // threads can split row 1 further. Note the nowait clause.
45      #pragma omp for schedule(dynamic, small_chunk) nowait
46      for(unsigned long int col=0; col<cols; ++col)
47      {
48          upgrade_cell_static(data_prev, data, 1, col);
49      }
50
51      // once done with row above, threads will start immediately
52      // on the last row.
53      #pragma omp for schedule(dynamic, small_chunk)
54      for(unsigned long int col=0; col<cols*(my_rows>1); ++col)
55      {
56          upgrade_cell_static(data_prev, data, my_rows, col);
57      }// barrier since I need to wait for all threads to finish before
58       // saving and swapping
59
60      // implied barrier at the end to finish saving and swapping
61      #pragma omp single
62      {
63          ++save_counter;
64          if(s>0 && save_counter == s && t<100000)
65          {
66              sprintf(snapshot_name, "snapshot_%05ld", t);
67              save_grid(snapshot_name, MPI_COMM_WORLD, rank, header, header_size,
        my_total_file_offset, data, my_rows, cols);
68              save_counter = 0;
69          }
70
71          tmp_data = data;
72          data = data_prev;
73          data_prev = tmp_data;
74          tmp_data = NULL;
75      }// barrier
76  } // barrier
```

The snippet is well commented and explained. However, we briefly touch on some important points. Since we create only one parallel region, we will need to be careful with the synchronization. For example, after the for loop on the internal cells, we have an implied barrier that we cannot eliminate, otherwise the wait operation may give an error (if executed before the recv has been posted).

There is another implied barrier after the MPI_Wait() calls, since we need to wait to receive the halo rows before processing the first and last rows of the grid.

There is another implied barrier after we process the last row, since we need to process the whole grid before possibly saving. Finally, there is another implied barrier before going on to the next step of the evolution, otherwise the data regions may not have been swapped.

In total, we have four implied barriers, which, as we will see later, kills the performance.

2.3.9 OMP Integration - Hybrid V2

The second versions tries to be as simple as possible.

During our development path for this project, we wrote a serial version, several pure OMP versions, and a pure MPI version before writing the hybrid version. This is one of the first implementations we had attempted but discarded for being too simple. Finally, we decided to come back to it because of its simplicity.

The idea is simply to put a pragma parallel for construct on top of the first for loop over the rows. In this way, each thread updates a subset of the rows - similar to how each MPI process gets its part of the grid.

The code is shown below:

```
for(unsigned long int t = 1; t < n+1; ++t)
{

    MPI_Isend(data_prev + cols, cols, MPI_CHAR, prev, prev_tag, MPI_COMM_WORLD, &
    prev_send_request);

    MPI_Isend(data_prev + my_rows_x_cols, cols, MPI_CHAR, next, next_tag,
    MPI_COMM_WORLD, &next_send_request);

    MPI_Irecv(data_prev, cols, MPI_CHAR, prev, next_tag, MPI_COMM_WORLD, &
    prev_recv_request);

    MPI_Irecv(data_prev + my_rows_x_cols_p_cols, cols, MPI_CHAR, next, prev_tag,
    MPI_COMM_WORLD, &next_recv_request);

    #pragma omp parallel for schedule(dynamic, chunk)
    for(unsigned long int row = 2; row < my_rows; ++row)
    {
        for(unsigned long int col = 0; col < cols; ++col)
        {
            upgrade_cell_static(data_prev, data, row, col);
        }
    }

    MPI_Wait(&prev_recv_request, MPI_STATUS_IGNORE);
    MPI_Wait(&next_recv_request, MPI_STATUS_IGNORE);
    MPI_Request_free(&prev_send_request);
    MPI_Request_free(&next_send_request);

    for(unsigned long int col=0; col<cols; ++col)
    {
        upgrade_cell_static(data_prev, data, 1, col);
    }

    for(unsigned long int col=0; col<cols*(my_rows>1); ++col)
    {
        upgrade_cell_static(data_prev, data, my_rows, col);
    }

    ++save_counter;
    if(s>0 && save_counter == s && t<100000)
    {
        sprintf(snapshot_name, "snapshot_%05ld", t);
        save_grid(snapshot_name, MPI_COMM_WORLD, rank, header, header_size,
    my_total_file_offset, data, my_rows, cols);
        save_counter = 0;
    }

```

```
44      tmp_data = data;
45      data = data_prev;
46      data_prev = tmp_data;
47      tmp_data = NULL;
48  }
```

It is virtually the same as the version 1, so we comment the most significant details.

First, we create a parallel region at *each* time step.

Second, we only paralellize the outer for loop over the rows, which means that in the extreme case where we have one row per process, the multhreading won't do anything. However, this is a very unlikely scenario, since we would need as many processes as rows. In this case, the issue will be with communication rather than lack of multithreading.

Finally, we explain *why* we initially got rid of this version, and then came back to it, calling it `V2`.

As mentioned above, we first developed some serial and pure OMP versions before doing the hybrid version. This, version, was our first attempt. Originally, one of our main preoccupations was that this version introduced a lot of overhead due to the creation of the parallel regions each time. This eventually led to putting the pragma construct outside of the for loop over the time steps and developing the MPI multiple mode.

Furthermore, another preoccupation, was that in the case of one row per process, this version would not paralellize using OMP. It seemed more practical and elegant that the OMP would *always* parallelize whatever work each process had to do.

However, we realized that by putting the parallel region outside, we needed to insert more barriers. Furthermore, some quick tests on pure OMP versions on our personal machine did not seem to show any real loss in creating the parallel region each time or just once.

In the end, we decided include it, just to test it out.

2.3.10 Software Stack - Running the code

On Orfeo, we always used the Open MPI module compiled against gnu version 12.2.1.

To compile the code, we provided a Makefile, which automatically compiles the source code. We used as many error flags as possible, such as -Wall, -Werror, and -Wextra. Finally, to optimize as much as possible, we compiled with -O3 and -march=native.

The binaries for THIN nodes and for EPYC nodes can be found in the aptly named folders.

To run the code we have the following options:

```
$mpirun [mpi_opt] conway_hybrid_pgm.out -i -f fname -k K
$mpirun [mpi_opt] conway_hybrid_pgm.out -r -f fname -n N -s S -e E
```

The first option is used to initialize and save a random playground of size $k \times k$ with name `fname`. The second option loads a grid named `fname` and evolves it for N timesteps, saving every S steps, with modality E. For the modality, 0 stands for `ordered` evolution, while 1 stands for `static`.

2.3.11 Miscellaneous

We briefly discuss some additional miscellaneous details.

To time the code, we used `MPI_Wtime()` which was inserted right before and right after the for loop over the time steps to avoid including the serial overhead of setting up everything for the evolution. We used a collective reduction operation to compute the average time among all processes.

Furthermore, we decided to conduct all our experiments with zero frequency saving. In other words, all of the execution times we report do not consider the IO, since with zero frequency saving we check whether we need to save *after* the last call to `MPI_Wtime()`. This is not a limitation, although it means that we are not analyzing the impact of the IO in our experiments.

To check for correctness of our code, we created a bash script called `check.sh`. This bash script runs all the implementations we have created, with two famous patterns: the Gosper Glider Gun[8] and the p43 Snark Loop[9]. The script compares the snapshots created at the end for a different number of evolution steps among the different implementations.

Since we have created a very simple serial version which we are sure is correct, this gives us some security in the correctness of our other implementations.

Lastly, we briefly talk about the contents of our folder. We have many implementations. Some of these were developed along the way to test some personal curiosities and to better guide us in the development of our final hybrid code.

In particular, we have a serial version we used mainly to be sure of the correctness of our results. We have three pure OMP implementations, which we used to test different ways to parallelize using OMP and to see some differences with respect to where we place the parallel regions and other aspects. Finally, we also have a pure MPI version which we initially implemented to begin thinking in parallel. This was based on some of the ideas presented here[10], here[11], and here[12]. Then, it became the base for the hybrid code.

In the `patterns_pgm` directory we find the PGM files for the gosper glider gun and for the snark loop. Furthermore, in the `pattern_gifs` we find small videos that we produced on our local machine to visualize the results of our evolution using the `ffmpeg` utility. The snippet below shows how to do this:

```
$mpirun -np 8 conway_hybrid_pgm.v2.out -r -f snark_loop.pgm -n 400 -s 1 -e 1
$ffmpeg -framerate 100 -f image2 -vcodec pgm -i snapshot_%05d my_gif.mkv
$mpv my_gif.mkv
```

2.4 Optional - Algorithm 2

This part is completely optional and not extensively tested for the main assignment. It was developed for my own personal curiosity and I wished to document it since I particularly enjoyed it.

In this section, we implemented a clever reformulation the problem from an algorithmic perspective based on the ideas presented in chapter 17 of [13].

The idea is that for the current implementation, we have to check eight neighbors for `every` cell. This means that we have to perform at least 8 instructions per cell, everytime. However, in Conway's game of life, for the majority of patterns of interest, most of the cells are dead and have all dead neighbors, and they stay this way throughout the evolution. So it is useless and expensive to check all eight neighbors for these cells each time.

In [13] they propose to fix this problem using the following clever trick. Since we only need one bit to encode the state of a cell and we are using an `unsigned char`, we are wasting 7 bits of space. Therefore, we can use four of these 7 bits to directly encode the sum of the 8 neighbors. This is shown in the figure below:

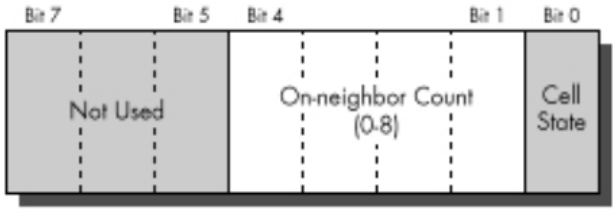

Figure 2.5: The cell representation we use for this algorithm.

In this way, since dead cells with all dead neighbors are represented by a null byte, we can quickly parse the grid for non-null bytes and avoid processing these cells, saving instructions.

However, there are a few things we must be careful with. First of all, whenever we modify the state of a cell by death or birth, we need to modify the neighbor count of all eight neighbors of the cell. Thus, we are introducing a data dependency between adjacent rows, since modifying one row can change the state of the previous and next row.

This doesn't seem like a real loss in the serial case, however, as we will see, paralellizing this algorithm with MPI requires double the amount of communications, more synchronization, and it is overall more complicated. And with OMP the complexity is even higher since we have to be expliticly careful with race conditions since modying one row may inadvertently modify another one on which another thread is working on. This is why we did not attempt to implement a hybrid implementation and just stopped with the pure MPI and pure OMP one.

All this overhead is worth it however, in the case of sparse grids, as we will see. However, in the case of a random grid or non-sparse grids, the added overhead is not worth it.

We show some results using this implementation at the end, as an optional section.

Now we briefly discuss the salient aspects of this implementation.

We proceed by steps, first discussing the serial implementation, in order to explain the fundamental ideas. Then we discuss the MPI and OMP versions, which are more focused on the correct synchronization of the updates.

2.4.1 Algorithm 2 - Serial Implementation

The initialization of the playground is identical to the other implementations.

However, when we need to run the playground, we need to do a bit of preprocessing that wasn't present in the other algorithm. This is to prepare the grid in the format that we described. Essentially, for each cell, we count the neighbors and write it in the 4 bits that we described earlier.

The relevant code is found here:

```
for(unsigned long int i = 1; i < rows+1; ++i)
{
    for(unsigned long int j = 0; j < cols; ++j)
    {
        preprocess_cell(data_prev, i, j, cols);
    }
}

...

void preprocess_cell(unsigned char * data_array, unsigned long int i, unsigned long
    int j, unsigned long int cols)
{
    // calculate j+1, j-1 with wrapping
    register unsigned long int jm1 = j==0 ? cols-1 : j-1;
    register unsigned long int jp1 = j==(cols-1) ? 0 : j+1;
    register unsigned long int im1 = i-1;
    register unsigned long int ip1 = i+1;

    // get neighbors state. We do a bitwise AND with 0x01 to
    // get the last bit which is the state of the cell.
    unsigned char tmp0=data_array[im1*cols+jm1] & 0x01;
    unsigned char tmp3=data_array[i*cols+jm1] & 0x01;
    unsigned char tmp5=data_array[ip1*cols+jm1] & 0x01;

    unsigned char tmp1=data_array[im1*cols+j] & 0x01;
    unsigned char tmp2=data_array[im1*cols+jp1] & 0x01;

    unsigned char tmp4=data_array[i*cols+jp1] & 0x01;

    unsigned char tmp6=data_array[ip1*cols+j] & 0x01;
    unsigned char tmp7=data_array[ip1*cols+jp1] & 0x01;

    // this will be a number between 8 and 0
    unsigned char n_alive_cells = tmp0+tmp1+tmp2+tmp3+tmp4+tmp5+tmp6+tmp7;

    // Now I need to shift everything left by one
    n_alive_cells = n_alive_cells<<1;

    // this will result in something like 000xxxx0 where the x's
    // represent any neighbor count.

    // lastly, if the cell is dead, we leave the last bit as is
    // and if the cell is alive, we need to set the last bit to 1
    // keeping everything else the same.
    // so if cell is dead we have:                      00000000
    // and if we do bitwise OR with nalivecells:        000xxxx0
    // we obtain again:                                 000xxxx0
    //
    // if cell is alive:     00000001
    // and we OR with        000xxxx0
    // we obtain:            000xxxx1
```

```
    // in conclusion,
    // we need to do bitwise OR between nalivecells and DATA(i,j)
    data_array[i*cols+j] |= n_alive_cells;
}
```

Now, the grid is in the format we described.

To perform the update, we need to be more careful this time because of the data dependency we introduced. First, to make everything easier, we copy the whole grid into the buffer of the next generation.

Next, we copy the last row into the top halo region.

Then, we proceed to update the first row. However, this may have changed the top halo, so we need to copy it back into the last row to ensure that it can be updated properly. Then, we process the second row, which in turn may change the first row.

Now that the first row has changed, we copy it into the bottom halo row. Then, we proceed to update all the remaining rows.

However, when we process the last row, this can update the bottom halo. So as the last step, we need to copy the bottom halo back into row 1.

As we can see, this process is more complex, and requires swapping of halo rows twice, which means that in the MPI version, we will have 4 communications per process rather than 2.

Furthermore, we have a bit of serialization, since we have to proceed in exactly that order for the first two rows. This reduces the paralellization a bit.

The relevant code snippet is shown below:

```
//copy buffer into buffer of new generation
memcpy(data+cols, data_prev+cols, grid_size_bytes);

// copy last row into top halo
memcpy(data, data + rows_x_cols, row_len_bytes);

// update row 1
// remember, we *look* at cell(i,j) in data_prev buffer
// but we update cell(i,j) and its neighbors in data buffer

for(unsigned long int j = 0; j < cols; ++j)
{
    if(DATA_PREV(1,j)==0)
    {
        continue;
    }

    upgrade_cell_static(data_prev, data, 1, j);

}

// we update row 2
for(unsigned long int j = 0; j < cols; ++j)
{
    if(DATA_PREV(2,j)==0)
    {
        continue;
    }

    upgrade_cell_static(data_prev, data, 2, j);

}

// copy top halo back into last row
memcpy(data + rows_x_cols, data, row_len_bytes);

// copy row 1 into bottom halo
memcpy(data+rows_x_cols_p_cols, data+cols, row_len_bytes);

// we go through the rest of the array (from row 3 to last row)
for(unsigned long int i = 3; i < rows+1; ++i)
{
    for(unsigned long int j = 0; j < cols; ++j)
    {
        if(DATA_PREV(i,j)==0)
```

```
        {
            continue;
        }

        upgrade_cell_static(data_prev, data, i, j);

    }
}

// we copy bottom halo into row 1
memcpy(data+cols, data+rows_x_cols_p_cols, row_len_bytes);

...


void upgrade_cell_static(unsigned char * restrict data_prev, unsigned char * restrict
    data, unsigned long int i, unsigned long int j)
{
    unsigned char count_of_neighbors;

    register unsigned long int jm1 = j==0 ? cols-1 : j-1;
    register unsigned long int jp1 = j==(cols-1) ? 0 : j+1;
    register unsigned long int im1 = i-1;
    register unsigned long int ip1 = i+1;

    // if we get here we have non zero element
    // so it is either
    // 1. a cell with some neighbors
    // 2. a live cell
    // 3. both
    //
    // However, since we already copied the contents
    // of data_prev, we only need to keep track of
    // events that *change* the status of the cell
    // i.e births and deaths

    count_of_neighbors = DATA_PREV(i,j) >> 1;

    // we first deal with deaths, however, we need
    // to make sure that we only update in the case
    // that the cell was alive and died

    // cell was alive, so last bit is 1
    if(DATA_PREV(i,j) & 0x01)
    {
        // if greater than 3 or less than 2, we kill it.
        // otherwise we dont do anything.
        if( count_of_neighbors>3 || count_of_neighbors<2 )
        {

            // to set cell to dead we need to
            // 1. flip its last bit. We can do this by doing
            // a bit wise AND with ~0x01. i.e 11111110
            // since 1 & 0 = 0 and 1 & 1 = 1, the first 7 bits
            // remain unchanged. while the last one is turned to
            // zero.

            DATA(i,j) &= ~0x01;

            // 2. get neighbors and decrease the neighbor count
            // of each one.
            // This can be achieved by subtracting 00000010 (or 0x02)
            // from each cell.
            // lets check possible limiting cases:
            // 1. will it ever be negative? We need a value smaller
            // than 0x02 for this to happen, namely 0x01.
            // However, by construction, this neighbors had at least
            // one live "on" cell (the one we are killing right now)
            // So the smallest possible value for any neighboring cell
            // is 0x02 (i.e 00000010).
            // So we will never obtain a negative value causing
```

```
116             // strange behavior. Thats good.
117             //
118             // 2. will the subtraction ever change the state of the
119             // cell? No, because we are guaranteed in this way that
120             // the last bit will remain the same
121 
122             DATA(im1, jm1)-=0x02;
123             DATA(im1,j)-=0x02;
124             DATA(im1,jp1)-=0x02;
125             DATA(i,jm1)-=0x02;
126             DATA(i,jp1)-=0x02;
127             DATA(ip1,jm1)-=0x02;
128             DATA(ip1,j)-=0x02;
129             DATA(ip1,jp1)-=0x02;
130         }
131     }
132     else
133     {
134         // the cell is dead.
135         if(count_of_neighbors == 3)
136         {
137 
138             // set to living and increase counter of neighbors
139 
140             // we set the last bit to 1 with a bit wise OR
141             DATA(i,j) |= 0x01;
142 
143             // 2. get neighbors and increase the neighbor count
144             // of each one. // To do this, we need to sum 0x02
145 
146             DATA(im1, jm1)+=0x02;
147             DATA(im1,j)+=0x02;
148             DATA(im1,jp1)+=0x02;
149             DATA(i,jm1)+=0x02;
150             DATA(i,jp1)+=0x02;
151             DATA(ip1,jm1)+=0x02;
152             DATA(ip1,j)+=0x02;
153             DATA(ip1,jp1)+=0x02;
154         }
155     }
156 }
```

The last thing left to discuss is what to do when saving the snapshot. Since the grid is in this special format, we need to extract the information contained in the last bit of each cell. We do this by copying the whole contents of one buffer into the other one, while also doing a bitwise AND with 0x01.

We also attempted to speed up this operation by using explicit vectorized instructions. Basically we did a bitwise AND with 0x01, 32 cells at a time. However, some quick testing on my local machine did not yield significant improvements (perhaps the -O3 optimization already did it for us). Therefore, to avoid making mistakes we left the simple implementation.

This concludes the discussion for the serial implementation. However, we can already appreciate the added complexity.

2.4.2 Algorithm 2 - MPI Implementation

The MPI implementation is conceptually similar to the serialized version we just described. The steps can be summaried as follows:

1. Copy old buffer into new buffer.
2. Non-blocking send last row (my_rows) to next rank and non-blocking receive it in its top halo.
3. Process rows 3 to my_rows $-$ 2 (to hide latency).
4. Wait until receive of top halo row is complete.
5. Process rows 1 and 2.

6. Non-blocking send top halo to the previous rank and non-blocking receive it in its last row.
7. Non-blocking send row 1 to previous rank and non-blocking receive it in its bottom halo.
8. Wait for receive of last row.
9. Process row the second to last row (my_rows − 1).
10. Wait for receive of bottom halo.
11. Process the last row.
12. Non-blocking send bottom halo to next rank and non-blocking receive it in its first row.

The code snippet is shown below:

```
// copy as usual the grid into new buffer
memcpy(data+cols, data_prev+cols, my_grid_size_bytes);

// send last row to next rank
MPI_Isend(data + my_rows*cols, cols, MPI_CHAR, next, next_tag, MPI_COMM_WORLD, &
    next_send_request);

// receive from prev and put in top halo
MPI_Irecv(data, cols, MPI_CHAR, prev, next_tag, MPI_COMM_WORLD, &prev_recv_request);

// process rows 3 until my_rows - 2
for(unsigned long int i = 3; i < my_rows-1; ++i)
{
    for(unsigned long int j = 0; j < cols; ++j)
    {
        if(DATA_PREV(i,j)==0)
        {
            continue;
        }

        upgrade_cell_static(data_prev, data, i, j);
    }

}

// wait to receive top halo
MPI_Wait(&prev_recv_request, MPI_STATUS_IGNORE);
MPI_Request_free(&next_send_request);

// process rows 1 and 2
for(unsigned long int j = 0; j < cols; ++j)
{
    if(DATA_PREV(1,j)==0 && DATA_PREV(2,j)==0)
    {
        continue;
    }
    else if(DATA_PREV(1,j) !=0 && DATA_PREV(2,j)!=0)
    {
        upgrade_cell_static(data_prev, data, 1, j);
        upgrade_cell_static(data_prev, data, 2, j);
    }
    else if(DATA_PREV(1,j)!=0)
    {
        upgrade_cell_static(data_prev, data, 1, j);
    }
    else
    {
        upgrade_cell_static(data_prev, data, 2, j);
    }
}

// send top halo to previous rank so it can be placed back into last row
MPI_Isend(data, cols, MPI_CHAR, prev, prev_tag, MPI_COMM_WORLD, &prev_send_request);
MPI_Irecv(data + my_rows*cols, cols, MPI_CHAR, next, prev_tag, MPI_COMM_WORLD, &
    next_recv_request);
```

```
54
55 // send row 1 to previous rank so it can be placed in bottom halo
56 MPI_Isend(data + cols, cols, MPI_CHAR, prev, prev_tag+1, MPI_COMM_WORLD, &
       prev_send_request1);
57 MPI_Irecv(data + my_rows*cols+cols, cols, MPI_CHAR, next, prev_tag+1, MPI_COMM_WORLD,
       &next_recv_request1);
58
59 // wait for receive of last row
60 MPI_Wait(&next_recv_request, MPI_STATUS_IGNORE);
61 MPI_Request_free(&prev_send_request);
62
63 // process row: my_rows - 1
64 for(unsigned long int j = 0; j < cols; ++j)
65 {
66     if(DATA_PREV(my_rows-1,j)==0)
67     {
68         continue;
69     }
70
71     upgrade_cell_static(data_prev, data, my_rows-1, j);
72 }
73
74 // wait until receive of bottom halo
75 MPI_Wait(&next_recv_request1, MPI_STATUS_IGNORE);
76 MPI_Request_free(&prev_send_request1);
77
78 // process last row
79 for(unsigned long int j = 0; j < cols; ++j)
80 {
81     if(DATA_PREV(my_rows,j)==0)
82     {
83         continue;
84     }
85
86     upgrade_cell_static(data_prev, data, my_rows, j);
87 }
88
89 // send bottom halo to next rank to be placed in row 1
90 MPI_Isend(data + my_rows*cols + cols, cols, MPI_CHAR, next, next_tag, MPI_COMM_WORLD,
       &next_send_request);
91 // blocking receive since we just need to wait for completion.
92 MPI_Recv(data + cols, cols, MPI_CHAR, prev, next_tag, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
93 // free send requests
94 MPI_Request_free(&next_send_request);
```

2.4.3 Algorithm 2 - OMP version

In this section we discuss the pure OMP version of algorithm 2 for completeness. This is the most complex version since we have data interdependency which causes race conditions.

The problem is that if we split the grid among threads, we may have race conditions for the rows at the boundary of the region that each thread works on.

To illustrate this, consider a simple example of a 10x10 grid and 2 threads. Thread 0 gets the first 5 rows, and thread 1 gets the last 5. However, the execution order of threads is not guaranteed by OMP, and it is decided by the OS. This means that if thread 0 starts immediately, while thread 1 is waiting for whatever reason, we may have a situation where thread 0 is working on rows 4 and 5, while thread 1 starts to work on row 6. However, processing rows 4 and 6 at the same time could lead to simultaneous updates on row 5. Also, processing row 5 could change row 6 which thread 1 is working on.

To eliminate this problem, we process the grid in steps of three at a time. In this way, we are sure that each thread cannot interfere with the other. However, this is worse for memory access as we have to traverse the grid 3 times in total.

This makes this approach very inefficient and undesirable, which is why we did not implement a hybrid implementation.

The code snippet is not shown as this implementation was discussed for completness but is not of

practical interest.

2.5 Results and Discussion

As mentioned in the introduction, we analyze Strong OMP, Strong MPI, and weak MPI scalability.

Each of these was repeated on both EPYC and THIN nodes when possible.

2.5.1 Ordered Evolution

As we mentioned previously, `ordered` evolution is intrinsically serial, so the only thing we can do is use MPI to handle much bigger grids. However, this is of very little interest in terms of scalability, since we won't gain anything by using more processes in terms of execution speed as the evolution is not paralellizable. The only paralellization available in this case is the MPI IO, which we do not include in our experiments, since we save with frequency zero. Furthermore, this feature is common to all the evolution modes.

In fact, we expect that as we increase the number of MPI processes, the execution time should stay the same, or get worse due to more communication overhead.

We test this on a THIN node with a random grid of size $10k$ and 200 evolution steps. The results are shown below.

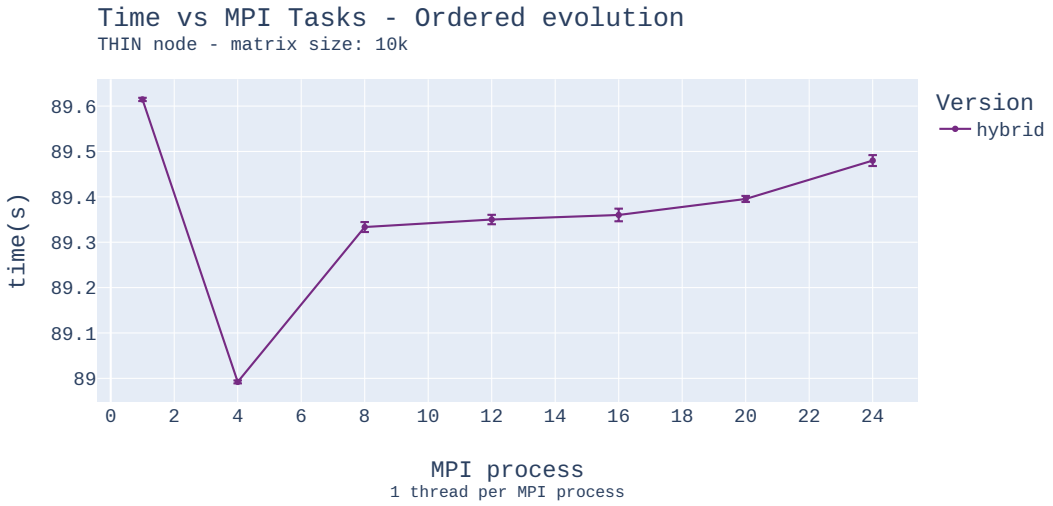

Figure 2.6: Time of `ordered` evolution of a random grid of size $10k$ as we increase the number of MPI processes on a THIN node.

As expected, the execution time stays approximately the same as we increase the number of MPI processes. We also highlight that `ordered` evolution is the same across the different hybrid implementations.

The rest of this section is devoted to `static` evolution which is more interesting in terms of scalability as it can be parallelized.

2.5.2 Static Evolution

Strong OMP Scalability

In this section, we use a single THIN or EPYC node, and fix the number of MPI tasks to 2, pinning each to one socket. Then, we slowly increase the number of OMP threads by changing the `OMP_NUM_THREADS` environment variable. This is done until we saturated each socket - i.e. 12 threads for THIN and 64 threads for EPYC. We use a randomly initialized grid with size fixed to $10k \times 10k$. We use 200 evolution steps and save only at the end. We use `OMP_PLACES=cores` and `OMP_PROC_BIND=close`.

We repeated the measurements 10 times to obtain a statistical average and standard deviation. Below are the results for THIN nodes:

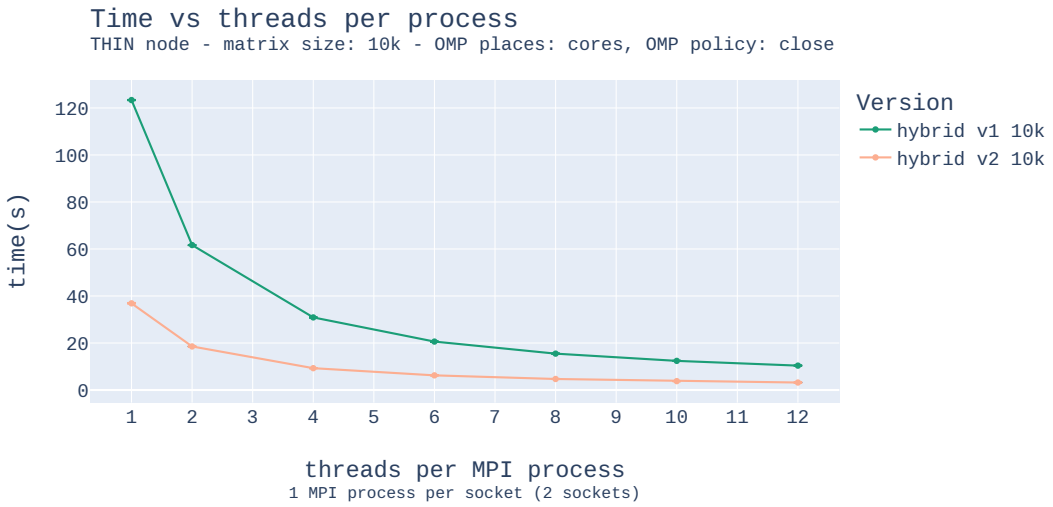

Figure 2.7: Time for THIN nodes with a random grid of size $10k \times 10k$. V1 takes almost three times as long as V2.

As we can see, hybrid version 1 (V1) takes almost three times the amount of time of hybrid version 2 (V2). This confirms our initial supposition that all the barriers that were introduced in V1 killed the performance.

We also decided to test our implementation on bigger matrices. However, due to limitations on job times, we limited our study to V2 for bigger matrices. We repeated the approach for matrices of size $20k \times 20k$ and $40k \times 40k$. This time, the sample size for $20k$ and $40k$ were 5 and 3, respectively.

Below we show the results:

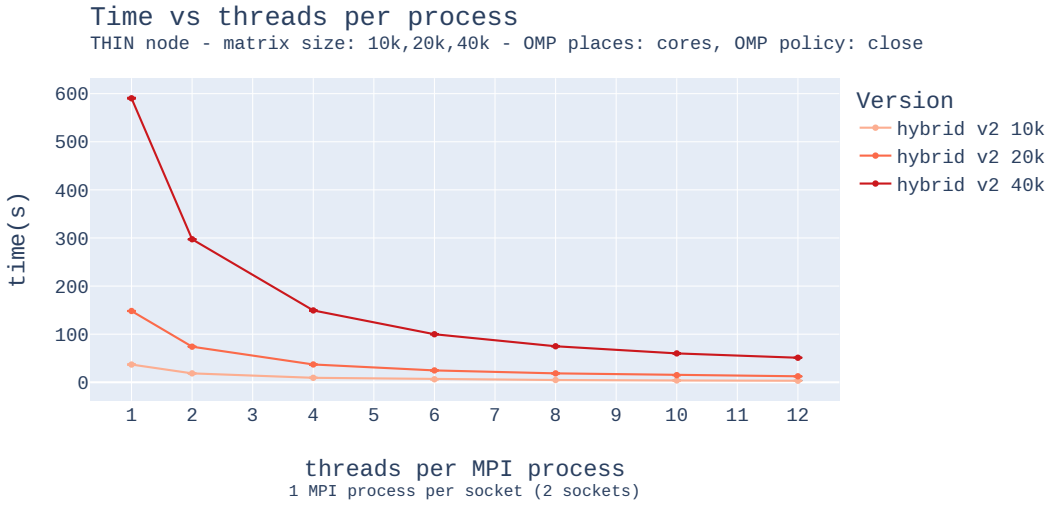

Figure 2.8: Time for THIN nodes for V2 with increasingly bigger grids.

Below we show the speedup for all of these runs.

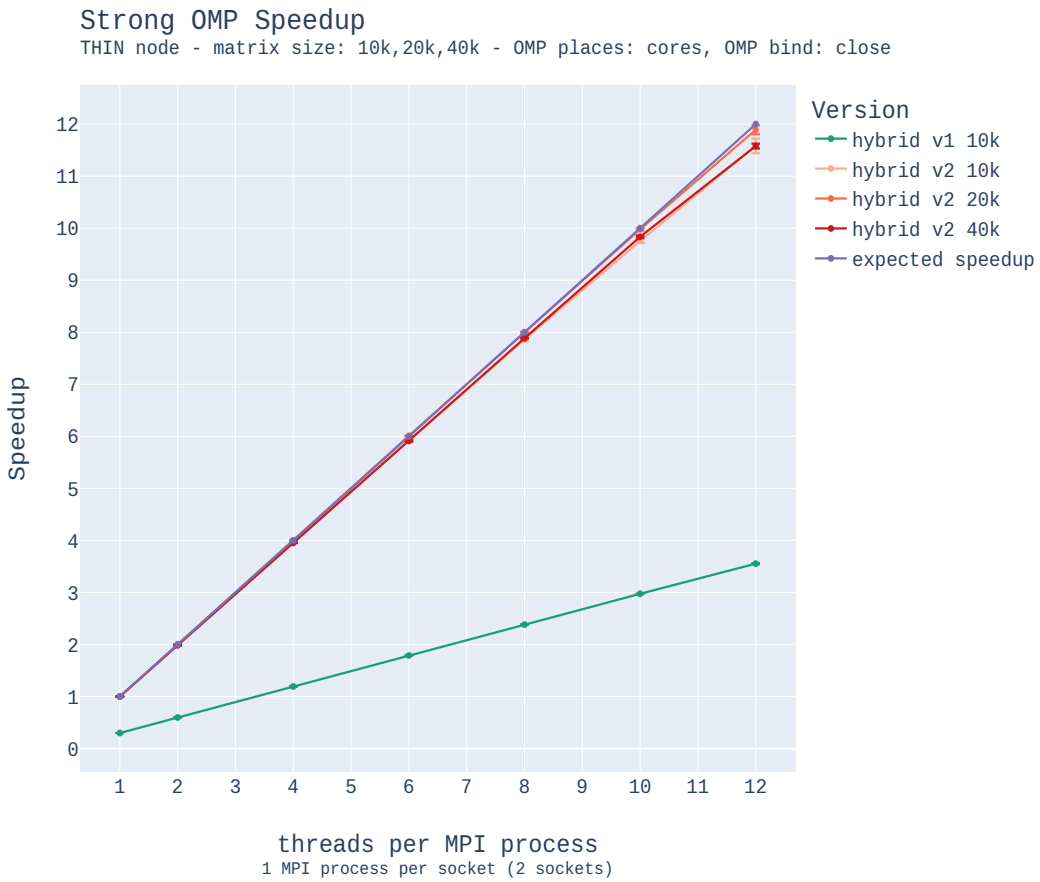

Figure 2.9: Speedup for strong OMP scalability using increasingly bigger grid sizes for V2. As we can see, V1 scales poorly due to the barriers that were introduced. V1 scales very well, indicating that our code is well implemented.

As we can see, the speedup of version V2 is almost perfect. On the other hand, V1 is not scaling well. This is a final indicator that this implementation is not the best, due to the various barriers that were introduced.

Furthermore, we see that V2 is able to scale well with increasingly bigger problem sizes that are 4 and 16 times as big as the original one. This is a good indicator that our code is well written, scalable, and efficient.

Now we repeat the same experiments with an EPYC node.

This time we only ran experiments with a $10k$ grid. However, given the more complex architecture of EPYC nodes, we decided to experiment with the OMP policy.

Initially, we used `OMP_PROC_BIND=close` as we did for THIN. Then, we remembered that EPYC nodes have a particular architecture. They have two sockets, like THIN, however, inside each socket, they have four NUMA regions.

Therefore, we hypothesized that using `OMP_PROC_BIND=spread` would give better results since each thread would intially be mapped into different NUMA regions within the socket. Therefore, it would not have to share the L3 cache.

Below we show the results for both hybrid versions, with both policies. We show both the time and the speedup.

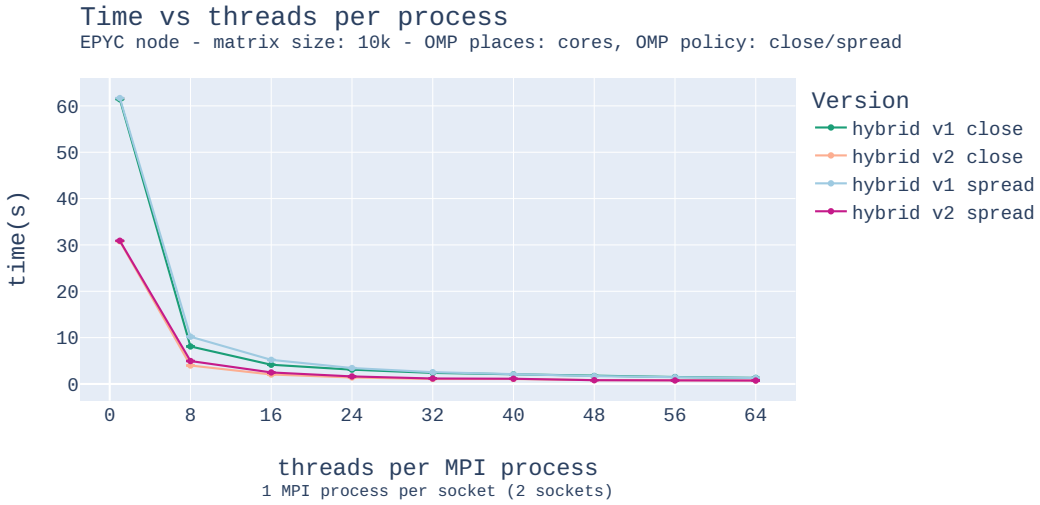

Figure 2.10: Time for EPYC nodes using a random grid of size $10k$ and both `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`. Results for both V1 and V2 are shown.

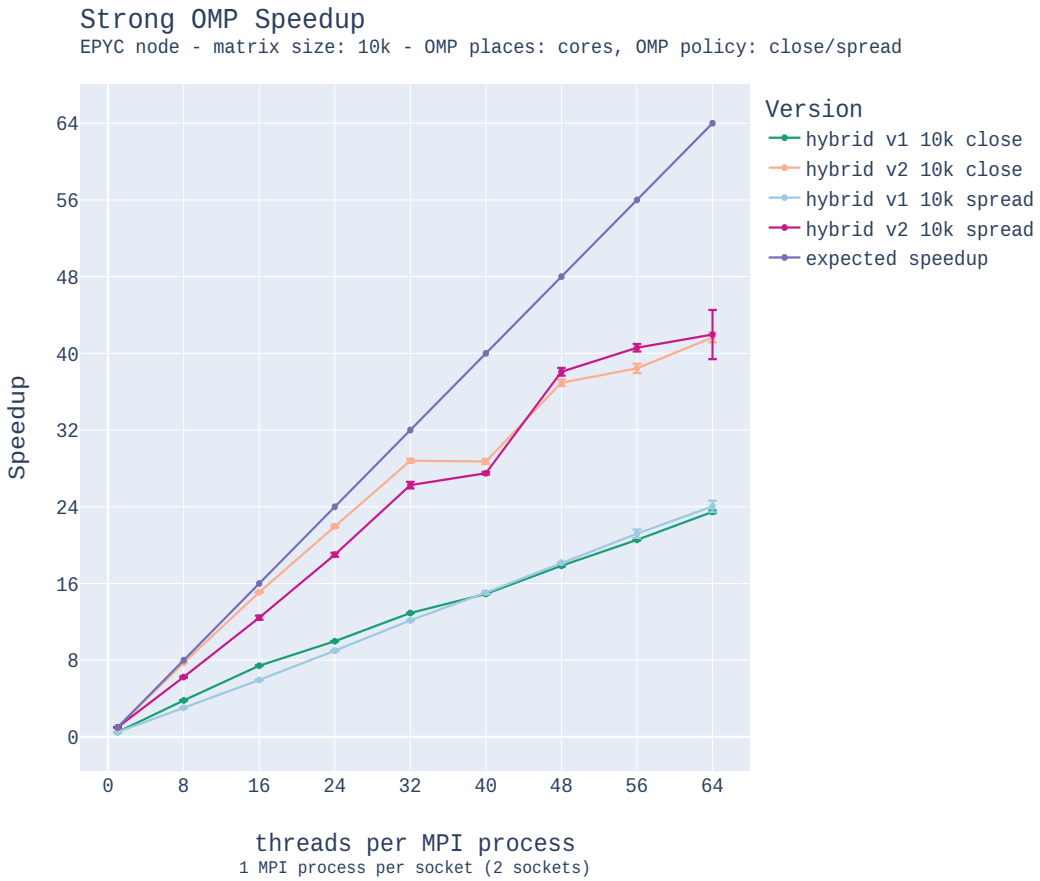

Figure 2.11: Speedup for strong OMP scalability using a random grid of $10k$ with `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`. Results for both V1 and V2 are shown.

First, we notice that the OMP policy barely changes the results. This leads us to think that using different NUMA regions is not as beneficial as we thought. However, considering the results of the previous chapter, it may be the case that with much bigger matrix sizes, we may obtain some benefits.

Furthermore, once again, we see that V1 does not scale well, regardless of the OMP policy that we use. On the other hand, V2 seems to scale linearly until we start to use half of each socket (32 threads per process). A possible explanation for this is that when we start to use more threads we begin to incur in more overhead for creation of the regions, synchronization of threads, resource sharing, and so on. However, when we use the full node, we still achieve $\sim 64\%$ of the theoretical speedup.

As a curiosity, we also ran the experiments only for V2 using a random grid of size $50k$. The results are shown below:

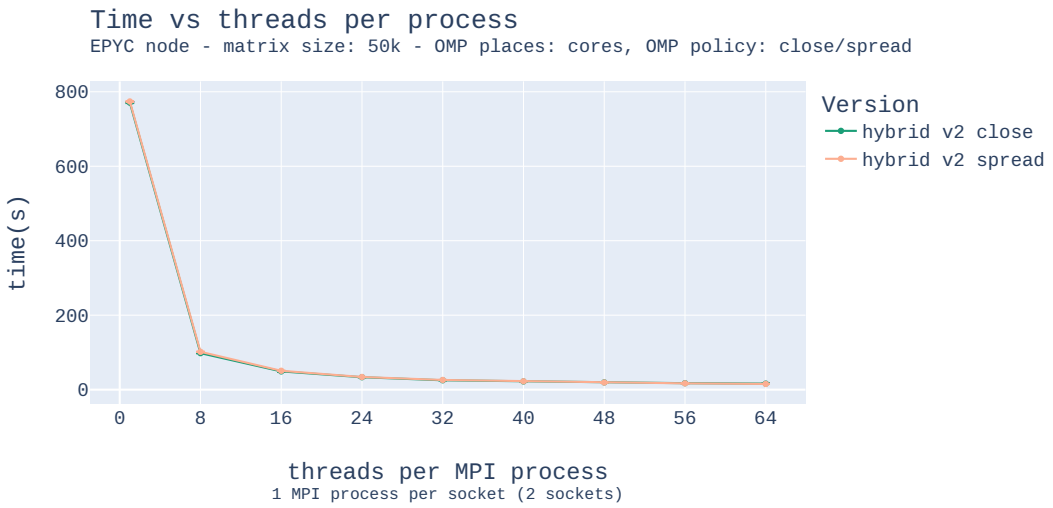

Figure 2.12: Time for EPYC nodes using a random grid of size $50k$ and both `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`. Only measurements for V2 were taken.

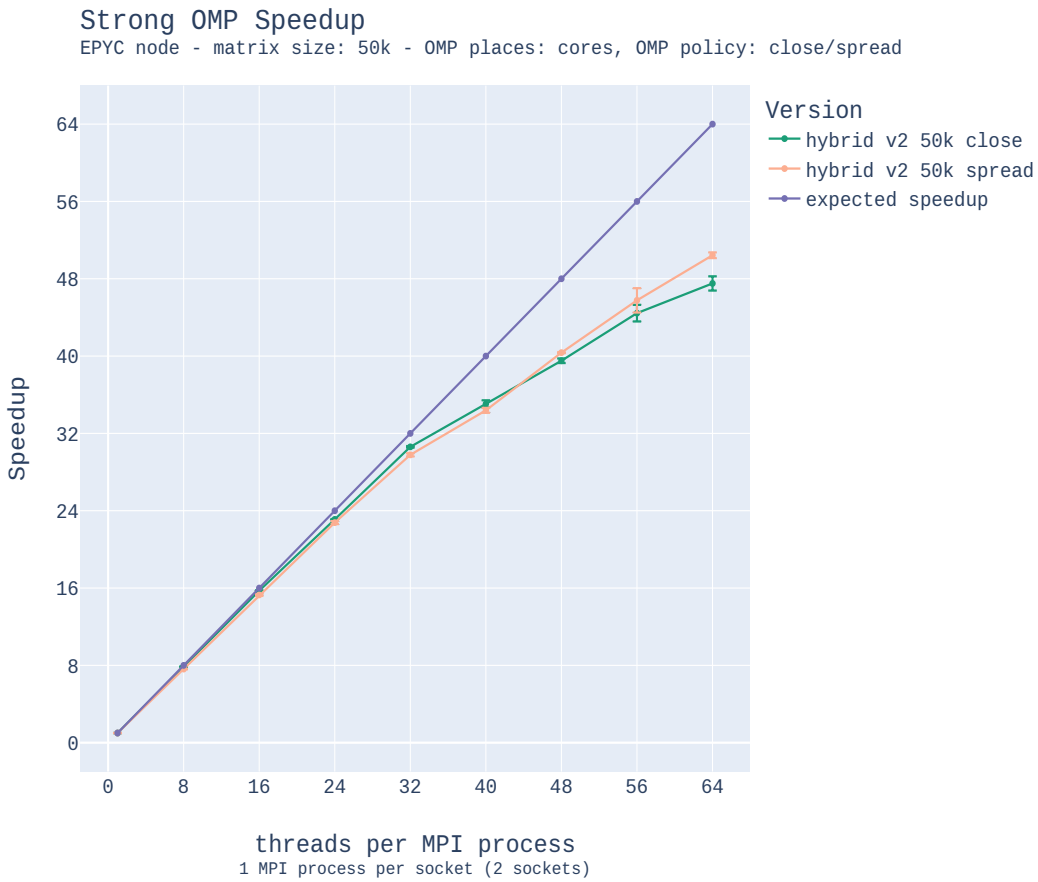

Figure 2.13: Speedup for strong OMP scalability using a random grid of $50k$ with `OMP_PROC_BIND=close` and `OMP_PROC_BIND=spread`.

As we can see, changing the policy seems to have no effect whatsoever on the performance.

However, we notice that when using a bigger grid, the scaling is a bit more stable across number of threads. However, the analysis is practically the same.

Strong MPI Scalability

In this section, we fix the number of OMP threads per MPI process to 1 and we continuously increase the number of MPI tasks, using the most nodes we can get for each architecture.

For these experiments, we wanted to use much larger grids to test the limits of our implementation. However, given the poor results for V1, we decided to test only V2. This choice was also made for practical time constraints as the V1 version took too long to execute on these larger grids.

On THIN, we used 3 nodes and grids of size $50k$, $70k$, and $100k$. The last one is approximately 10 GBytes of data. On EPYC, we used 4 nodes and used the same grid sizes, to have a comparison. Results were taken 3-5 times to get an average (due to time constraints with job sizes).

Below we show the results for THIN nodes:

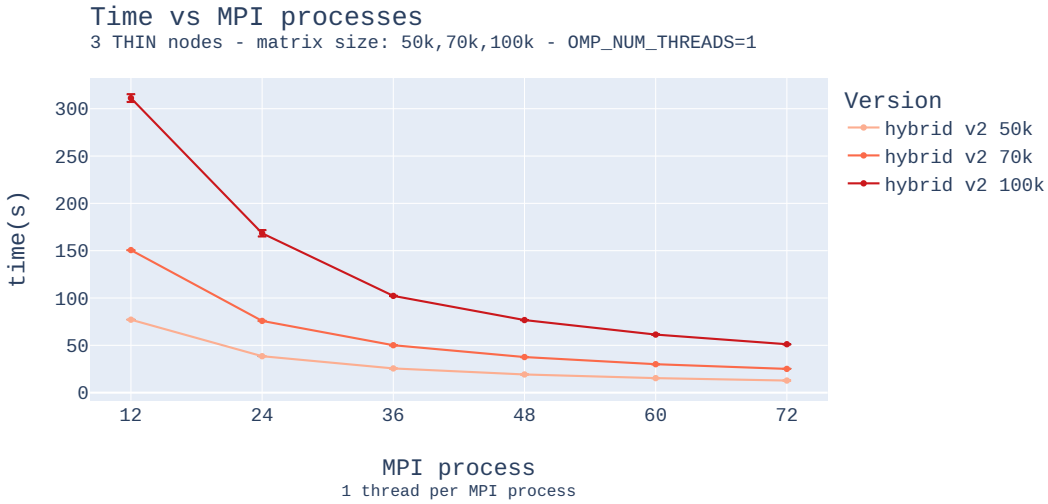

Figure 2.14: Time for THIN nodes using random grids of size $50k$, $70k$, and $100k$ using 3 nodes and one thread per MPI task.

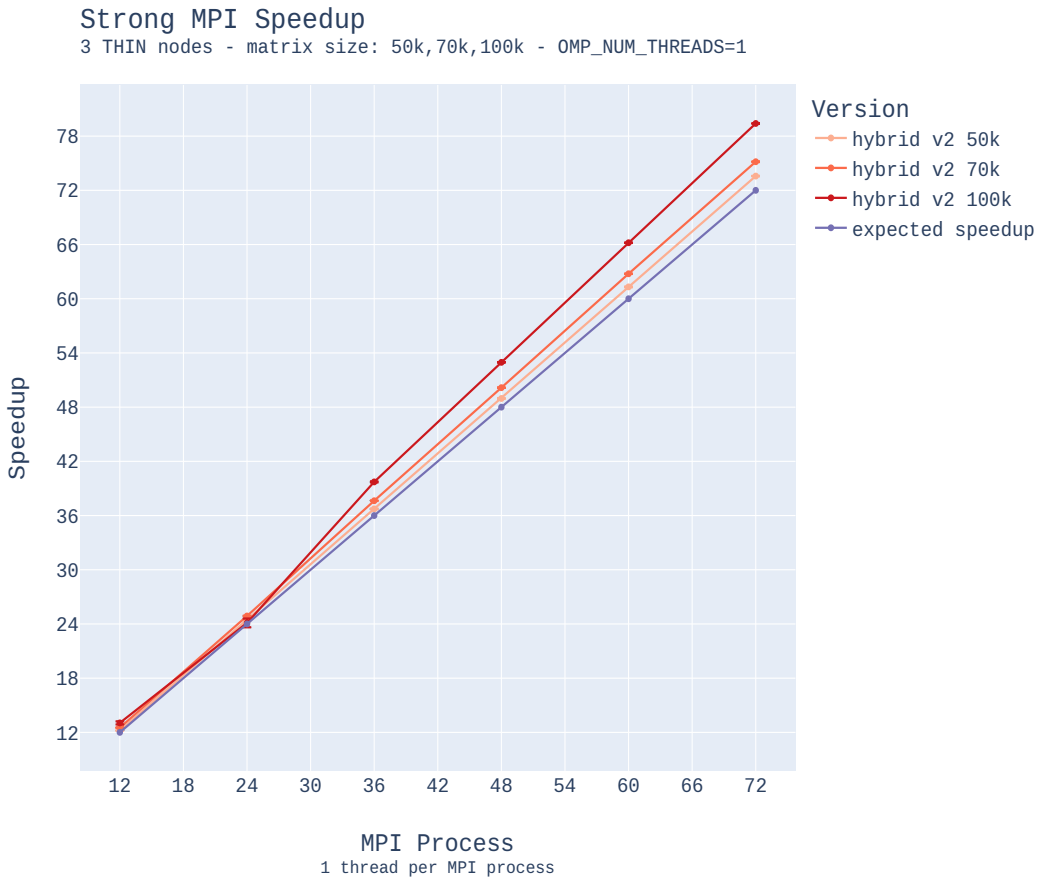

Figure 2.15: Speedup for strong MPI scalability for random grids of size $50k$, $70k$, and $100k$ across 3 nodes. The scalability is superlinear.

As we can see, we obtain a superlinear scalability. This is a very strange occurrence and it can happen when our base value for the serial time is worse than it should be, or if the problem is benefitting from better cache usage.

In our case, we also tested a purely serial code which was slower than the hybrid V2 with one MPI task. This is because in the hybrid implementation we took advantage of non-blocking communications to exchange the halo rows while each process started to work on the internal rows. On the other hand, in the serial implementation we have to wait for the exchange to happen. Therefore, we conclude that the most probable reason for the superlinearity is that as we use more and more processes, the problem size becomes smaller and more of it can fit in the cache, speeding up the computation.

Now, we show the results for EPYC nodes.

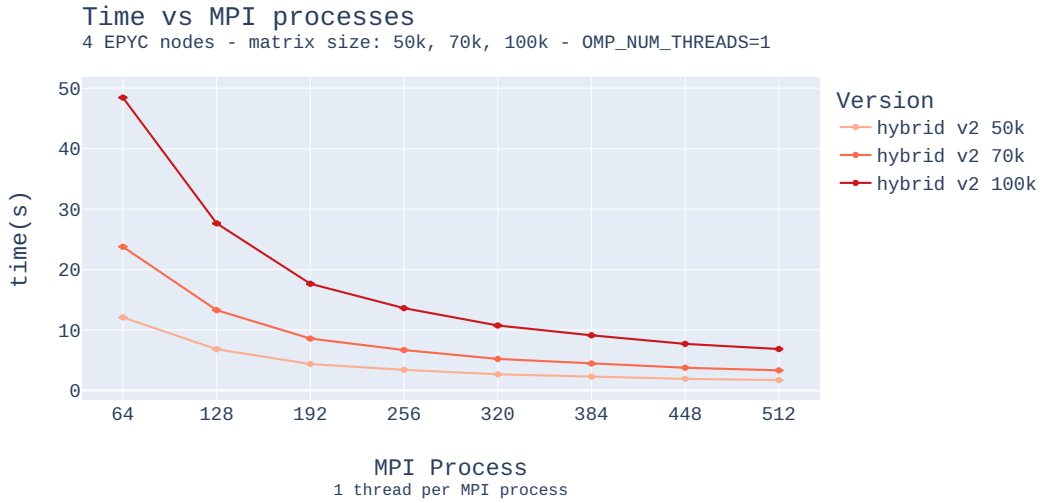

Figure 2.16: Time for EPYC nodes using random grids of size $50k$, $70k$, and $100k$ using 4 nodes and one thread per MPI task.

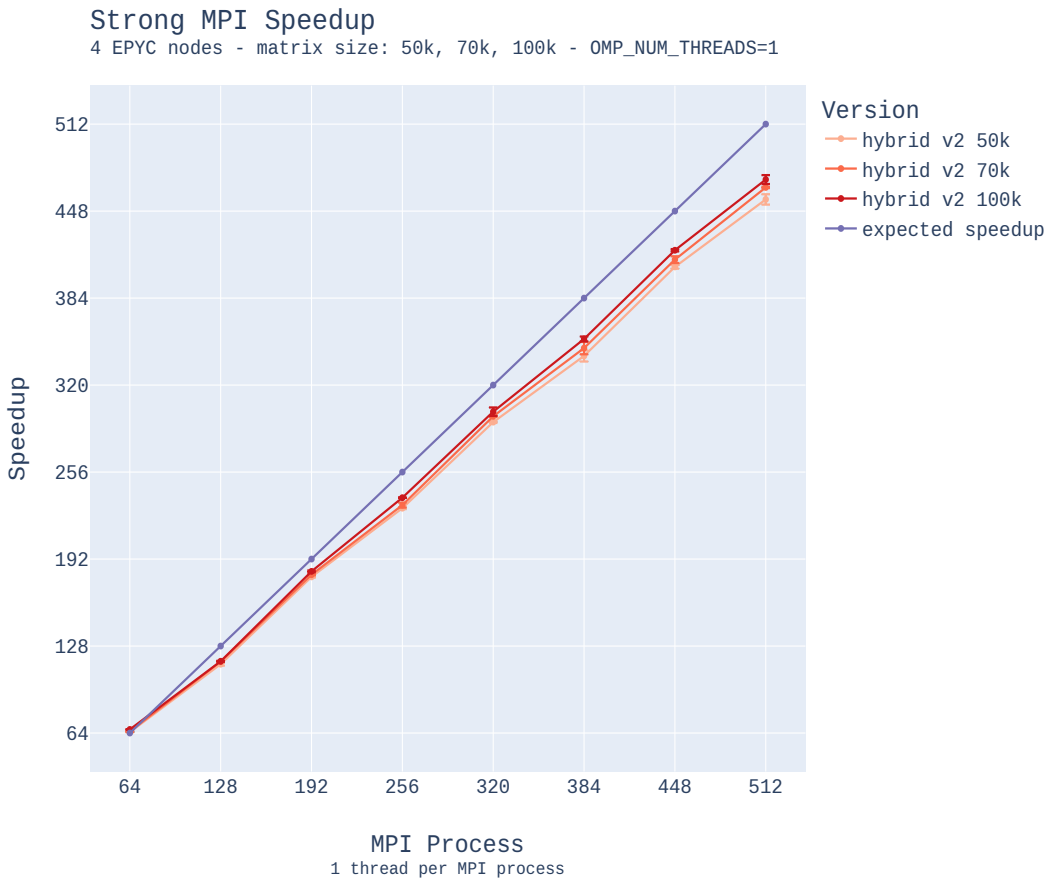

Figure 2.17: Speedup for strong MPI scalability for random grids of size $50k$, $70k$, and $100k$ across 4 nodes. The practically linear across all grid sizes.

In this case, we notice that the implementation is scaling linearly. This is interesting considering that with strong OMP scalability, the scalability stopped being linear within a single node. One possibility for this is that using multiple threads per task adds some thread management overhead which reduces the scalability.

Weak MPI Scalability

In this section, we fix the number of OMP threads to completely saturate a socket - i.e 12 for THIN and 64 for EPYC. Then, we slowly increase the number of MPI tasks, pinning them to the sockets, across as many nodes as possible. We start with an initial grid size of $10k$ and as we increase the number of MPI tasks, we also increase the grid size to keep the workload per MPI task the same. In other words, if we double the number of tasks, we double the workload.

Below we show the dimensions we used and the workload per MPI task.

Size	MPI Tasks	Work per MPI task
10000	1	$\frac{10000^2}{1} = 1.00000 \times 10^8$
14142	2	$\frac{14142^2}{2} = 0.99998 \times 10^8$
17320	3	$\frac{17320^2}{3} = 0.99994 \times 10^8$
20000	4	$\frac{20000^2}{4} = 1.00000 \times 10^8$
22360	5	$\frac{22360^2}{5} = 0.99993 \times 10^8$
24494	6	$\frac{24494^2}{6} = 0.99992 \times 10^8$
26457	7	$\frac{26457^2}{7} = 0.99996 \times 10^8$
28284	8	$\frac{28284^2}{8} = 0.99998 \times 10^8$

Table 2.1: The grid dimensions we used as we increased the number of MPI tasks. Since we are working with squared grids only, it is difficult to obtain exactly the same workload. However, as we can see, the workload remains the same up to a negligible difference.

For THIN nodes, we used three nodes. Below we show the results for the weak efficiency:

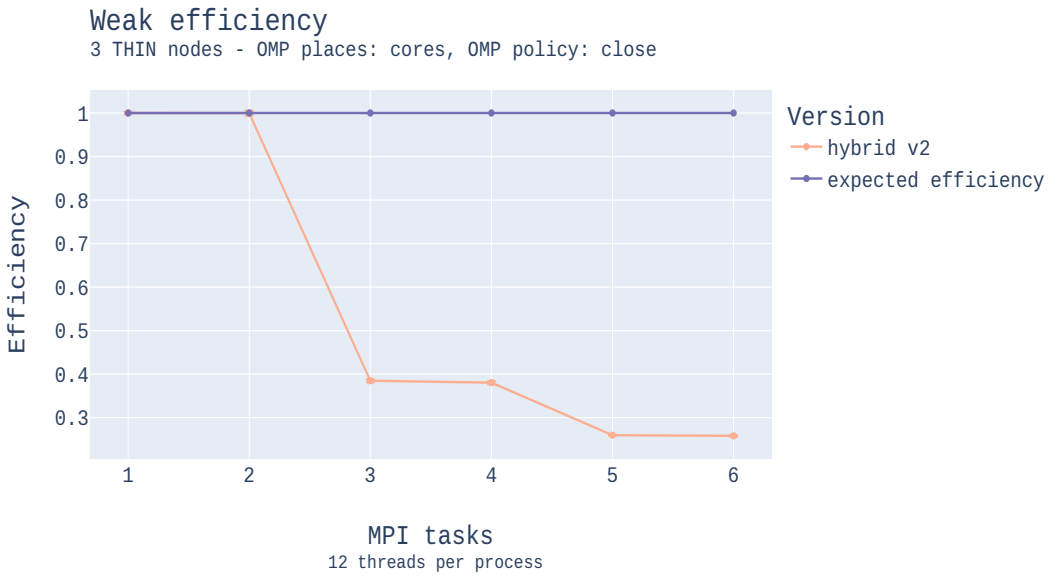

Figure 2.18: Weak efficiency for THIN nodes as we simultaneously increase the number of MPI tasks and the grid size, keeping the workload per process constant.

As we can see, the efficiency becomes worse as we increase the number of MPI tasks and start using more nodes. This is to be expected since this implies a higher communication overhead, more resource contention, and the implementation becomes overall less efficient.

Now we show the results for EPYC nodes. In this case, we used four nodes in total.

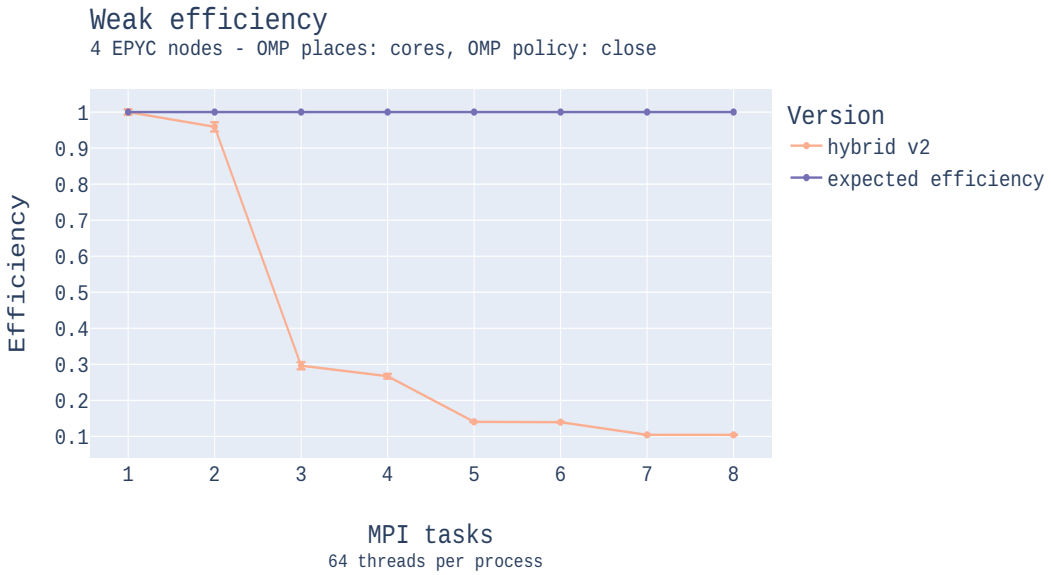

Figure 2.19: Weak efficiency for EPYC nodes as we simultaneously increase the number of MPI tasks and the grid size, keeping the workload per process constant.

The results for the weak efficiency for EPYC nodes is very similar to the case of THIN nodes. However, in this case we have slightly worse efficiency, which could be due to the fact that we have much more thread overhead per task compared to THIN nodes.

2.6 Conclusions

In conclusion, our V2 implementation scaled well when we considered strong scaling, on all architectures, and for various increasing grid sizes. We explored results for strong OMP scaling using grids up to $40k$. For strong MPI scaling, we used grids of size up to $100k$ across many nodes.

In some cases, we obtained superlinear scaling, which was probably caused by better cache usage.

When we considered weak scaling, on both THIN and EPYC nodes, the weak efficiency progressively got worse as we used more MPI processes and started using more nodes. However, this is to be expected.

More importantly, we also observed that it is not always beneficial to implement overly complicated versions, and that sometimes the best solution is the simplest one. This is evidenced by the poor scaling achieved by version V1, which was conceptually more advanced and refined, but required more synchronization, which ultimately killed performance.

Lastly, we talk about some possible improvements. All our experiments were conducted with zero frequency saving. However, in most applications of interest, we are interested in visualizing the pattern as we evolve the grid. Therefore, it would be very interesting to repeat the above experiments using a high frequency of saving, for example s=1. We expect that the scalability will suffer as we increase the number of processes, since more processes will be writing the snapshot image. In fact, according to this[10] source, which we used for MPI IO, one of the biggest bottlenecks of today is caused precisely by the IO operations.

Another very interesting idea would be to implement a version that deals with everything at the bit level, to be more memory efficient. We expect that this version will be slightly slower since we will have more overhead to extract the information from the cells, however, by doing this, we would be able to deal with problem sizes 8 times as big.

2.7 Optional - Results and Discussion: Algorithm 2

This last optional section is devoted to presenting and discussing some results of the implementations for the second algorithm. In particular we compare the serial programs of both algorithms and then

we study the strong MPI scalability of the pure MPI implementation of algorithm 2 and compare it to the hybrid V2 implementation.

Note that this section is completely optional and not the main focus of the assignment.

For simplicity, we only run our experiments on THIN nodes.

As discussed previously, the idea behind algorithm 2 is to reduce the number of unecessary updates by skipping over dead cells with dead neighbors. Therefore, we expect to have the best results when using a very sparse grid. In our analysis, we will consider both a randomly initialized grid as well as a sparse one, and compare the differences.

Of course, our analysis is limited to `static` evolution.

2.7.1 Serial comparison

We begin by comparing the serial versions of both algorithms: the one based on the first algorithm (A1) and the new one based on algorithm 2 (A2).

We use a randomly initialized grid of size $10k$ and a sparse grid of the same size based on the glider pattern[14], which is a famous pattern that will "glide" through the grid indefinitely. In this grid, only 5 cells are alive in any given step of the evolution.

In the random grid, we will have random noise throughout the evolution, so we don't expect much of a speedup. If anything, the additional overhead will slow down the serial A2 implementation compared to the serial A1. However, for the second grid, we expect that the serial A2 will perform better.

Below we show the results of the serial implementations using both grids and doing 200 evolutions, with zero frequency saving. We ran all experiments 10 times to obtain a mean and standard deviation.

Algorithm	Random Grid 10k	Glider 10k
(A1) conway_serial_pgm.out	85.15 ± 0.10 (s)	84.93 ± 0.02 (s)
(A2) algo2_conway_serial_pgm.out	101.03 ± 0.01 (s)	25.15 ± 0.01 (s)
	0.84x slower	3.5x faster

Table 2.2: Comparison of serial versions for both algorithms on a randomly initialized grid and a sparse grid of size $10k$.

As expected, the execution time for serial A1 is the same for both grids since it performs the same number of operations, regardless of the grid. We also observe that, as predicted, for a random grid, the serial A2 is slower than serial A1. This is the result of additional overhead in the algorithm since it is more complex.

However, for a very sparse grid, the serial A2 is 3.5 times faster than the A1 counterpart. This is a notable speedup considering that we only changed the algorithmic perspective.

Next, using the same grids, we fixed `OMP_NUM_THREADS=1` and ran the A1 hybrid V2 implementation as we increased the number of MPI tasks across 3 THIN nodes. We repeated the same procedure with the pure MPI implementation for A2.

Below we show the results of the execution time.

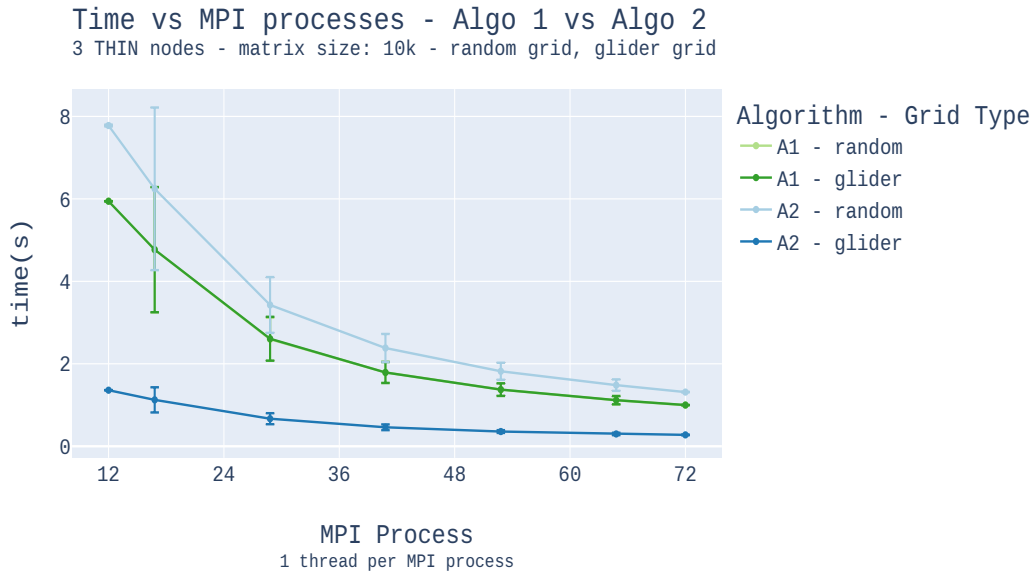

Figure 2.20: Time for A1 hybrid V2 implementation vs pure MPI implementation of A2, using a random grid and a sparse glider grid. When using a sparse grid, we obtain a considerable speedup.

As expected, the execution time for A1 hybrid V2 doesn't change when we use the different grids since we always check *all* the cells. On the other hand, the pure MPI implementation of A2 is slower when we use a randomly initialized grid, which is not very sparse. However, it is much faster when we use the very sparse glider grid. This is because we skip over many cells that do not need to be updated, which reduces considerably the number of instructions used.

Now we show the speedup. We use the smallest execution time for $T(1)$ as the reference value.

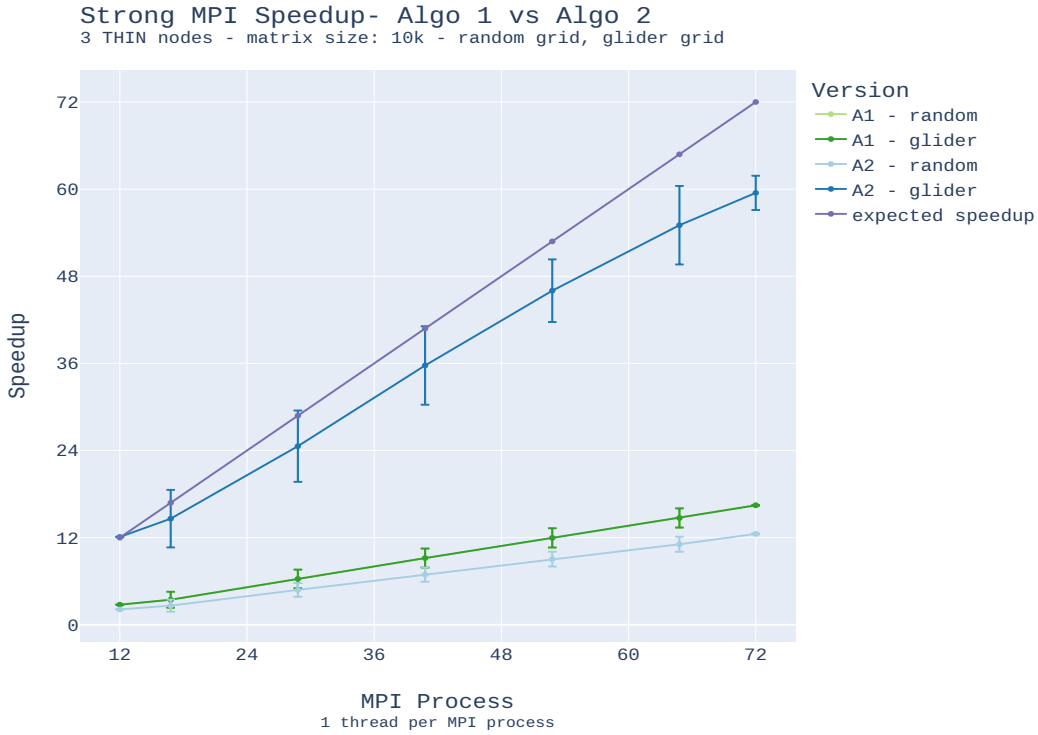

Figure 2.21: Speedup for strong MPI scalability for A1 hybrid V2 vs pure MPI implementation of A2. We use both a random grid and a sparse grid of size $10k$.

In this case, the pure MPI implementation of A2 scales better than the A1 hybrid V2 counterpart when using a sparse matrix. However, when using a random matrix, the scaling is obviously worse.

We conclude therefore that the pure MPI implementation of A2 is advantageous only if we are dealing with sparse grids. This is a neat result that could be useful if one knows a priori that they will be dealing with sparse patterns.

This concludes this optional section.

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/.

[7] Prace. "Advanced mpi io: Mpi io." (), [Online]. Available: https://events.prace-ri.eu/event/176/contributions/59/attachments/170/326/Advanced_MPI_II.pdf.

[8] LifeWiki. "Gosper glider gun." (), [Online]. Available: https://conwaylife.com/wiki/Gosper_glider_gun.

[9] LifeWiki. "P43 snark loop." (), [Online]. Available: https://conwaylife.com/wiki/P43_Snark_loop.

[10] R. Latham. "Introduction to mpi io." (), [Online]. Available: https://www.youtube.com/watch?v=huH_-a3qrfA.

[11] roblatham00. "Hands-on." (), [Online]. Available: https://github.com/radix-io/hands-on/tree/main/mpi2tutorial/examples/life.

[12] W. Gropp. "Advanced mpi: I/o and one-sided communication." (), [Online]. Available: https://www.mcs.anl.gov/research/projects/mpi/tutorial/advmpi/sc2005-advmpi.pdf.

[13] M. Abrash. "Michael abrash's graphics programming black book, special edition." (), [Online]. Available: https://www.jagregory.com/abrash-black-book/%5C#chapter-17-the-game-of-life.

[14] LifeWiki. "Glider." (), [Online]. Available: https://conwaylife.com/wiki/Glider.