Master in High Performance Computing

AP-IO: An Asynchronous I/O Pipeline for CFD code ASHEE.

Supervisor(s):

José Gracia

Candidate:

Miguel Castellano

6th edition

2019–2020

Acknowledgments

I'd like to thank in the first place the lecturers and staff members of the MHPC master for their hard work, dedication and availability during this whole year. Secondly, I want to thank my supervisor José Gracia for his time and support as well as all other members of the SPMT group at HLRS, who were at all times nice to me and helped me whenever needed. They managed to make me feel welcome despite all COVID19 restrictive measures for social distancing. Last but not least, I'd also like to pay special thanks to the secretaries of HLRS and specially to Ms. Arnold, who patiently answered all my e-mails and helped me with all the German paperwork.

On the institutional part, I want to thank the ICTP and the SISSA school for their big commitment to this master program and the effort that they've invested in such an amazing project and also the University of Stuttgart and specially the HLRS for employing me and providing me with all sort of courses, equipment and other perks during my stay. Additionally, I'd like to express my gratitude to the BSC and their support team, which helped me in more than one occasion. Their machine (MareNostrum 4) turned out critical for the completion of this thesis.

To end with, I want to thank the ChEESE center of excellence, which funded this project and many others, with the aim of preparing Solid Earth simulation codes for the Exascale era.

Contents

1 Introduction

ChEESE is a Center of Excellence in Solid Earth science aimed at boosting computational performance of scientific simulations in this domain, with the purpose of preparing existing codes for the exascale. There are currently 10 different codes under development, which can be found on their website [1] and include 4 in computational seismology, 2 in Magnetohydrodynamics (MHD), 2 in physical volcanology, and 2 in tsunami modelling. One of the 2 codes targeting volcanology science is called ASHEE, for "ASH Equilibrium Eulerian" and it was developed for the multi-phase flow modelling of volcanic plumes.

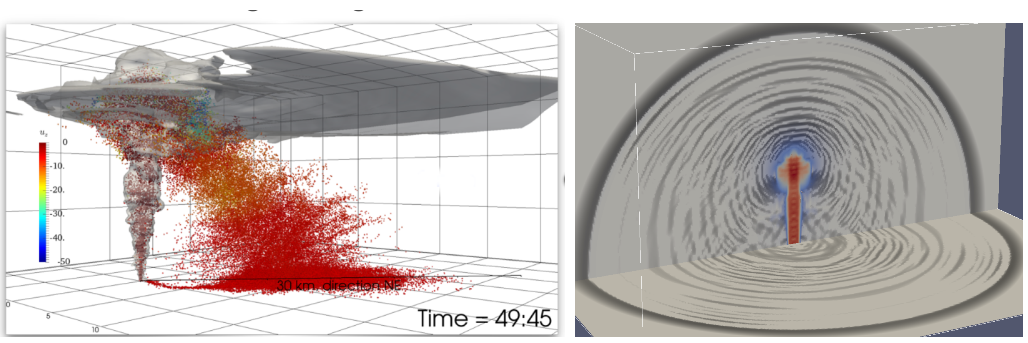


Figure 1: **Left**: Visualization of ASHEE output. **Right**: Mini-app output. Images provided by Istituto Nazionale di Geofisica e Vulcanologia (INGV).

ASHEE [2] is a Computational Fluid Dynamics (CFD) code based on the OpenFOAM open-source C++ toolbox for numerical solvers. It is composed of what we call the mini-app, which constitutes the main solver, plus some extra pieces of code which compute additional fields and other relevant variables. In the following, the focus will be laid onto the mini-app, that is, the core solver of ASHEE, which is a pre-defined OpenFOAM solver for compressible fluids called "rhoPimpleFoam". In particular, it is the input-output (I/O) part of the code that will be optimized or, more specifically, the output part, in charge of writing simulation data to disk.

Input/Output has for some time now been one of the core limiting factors of application performance due to the high latency of I/O operations ($\sim ms$) and the extreme data volumes used in current applications. Over the years, the core performance of most processors has sky-rocketed, but so has the amount of input and output data, shifting the bottleneck from Floating Point Operations per Second (FLOPS) to I/O operations per second (IOPS) for many scientific I/O - intensive applications. In the era of data, the need for an efficient, high-throughput solution for data-intensive applications has become a priority and much are the efforts that have been made in this direction, both at the software and hardware level. Burst buffer technologies like DDN's Infinity Memory Engine (IME) [3], consisting of an intermediate appliance lying between the application and the file system, or high performance I/O libraries like ADIOS [4], are two good examples of these efforts [5].

In this work, however, based on the paper by Ren Xiaoguang and Xu Xinha [6] from the State Key Laboratory of High Performance Computing in China, it is asynchronous execution that is exploited to maximize parallelism and hide I/O latency behind the execution of other instructions, in particular, for a CFD code, those in charge of computing the fields of interest. While the original paper studies three of the most common OpenFOAM solvers (interFoam, icoFoam and pisoFoam), only the rhoPimpleFoam solver is considered for this thesis, along with some extra metrics that are omitted in the cited paper.

1.1 Asynchronous programming

Asynchronous programming or asynchronous execution is related to the flow of instructions in a program. Within a single thread, all instructions are executed synchronously, that is, one after the other. One instruction is executed only after the previous instruction is over. This way, a thread is executed by a single core at a time and the order of instructions complies with the data dependencies established. However, this is not to be confused with instruction pipe-lining, or out-of-order execution, where instructions within a thread can actually be run in parallel or in a different order. In the case of out-of-order execution, instructions are processed following the *data order*, the order in which the data and operands become available in the processor's registers. Nevertheless, the final results are queued and re-ordered at the graduation or retire stage to comply with the original *program order*. This makes it possible to use clock-cycles that would have otherwise been wasted, but the overall execution order within the thread is sequential, thanks to this final reordering. Instruction pipe-lining, on the other hand, allows for the concurrent execution of different instruction stages through the use of different dedicated resources. But again, the final order is the same within the thread.

Having two separate threads, however, we can actually have instructions running asynchronously, which means that one instruction can be left running while the next batch of instructions executes in parallel without waiting for it to complete. This is done in the following way:

A thread is spawned by another thread to work independently on a given instruction. Meanwhile, the original thread continues to execute the next batch of instructions. Put like this, execution is synchronous within each of the threads, but it is asynchronous when the other is taken into account. That is to say, the execution of instructions running on the separate thread is asynchronous when compared to the original thread, since this one doesn't wait for completion of the instructions running on the spawned thread to continue execution, but does it in parallel with them.

Now, How is this different for I/O? Let us first familiarize with the basics of input/output at the HPC level in order to get a better grasp on how asynchronous execution can be better coupled with it.

1.2 Fundamentals of HPC I/O

In all HPC systems or computing clusters, input/output operations (IOPS) are coordinated by the parallel file system, which is a computer appliance combining software and hardware components designed to facilitate the storage of data across multiple networked servers. Parallel file systems allow for the concurrent reading and writing of data to and from distributed storage devices by simultaneously running processes across the cluster. This is possible thanks to the use of multiple I/O channels or controllers, which provides a significant I/O throughput, especially when streaming workloads that involve a large number of computing nodes. There are many different parallel file systems, but the most commonly used are IBM's GPFS (MN) and open-source Lustre (Hawk).

1.2.1 The Lustre parallel file system

Lustre (from Linux + cluster) is an open-source parallel file system which is used in many of the TOP500 [7] HPC systems in the world, including No.1 ranked ”Fugaku” supercomputer at RIKEN Center for Computational Science, Japan. The architecture of a typical Lustre file-system is illustrated in Fig.2. One of its defining features is having separate servers for metadata to relieve pressure from the main data servers. As observed on the figure, all compute nodes are connected through InfiniBand [8] or similar with each other and the I/O servers (Object Storage Servers or OSS). Likewise, the I/O servers are connected to each other and the RAID disks (Object Storage Targets or OST) through an Ethernet network. These I/O servers or OSS take I/O requests from all compute nodes across the cluster and execute them in parallel to the greatest extent possible, given the amount of requests and the number of available servers. This can lead to instabilities on I/O performance caused by different levels of contention over the storage resources, which makes I/O a difficult thing to measure.

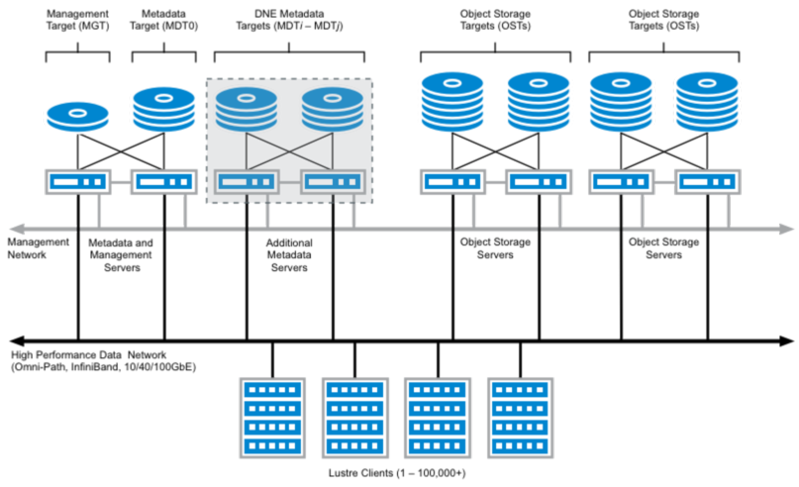

Management Target (MGT)
Metadata Target (MDT0)
DNE Metadata Targets (MDTi – MDTj)
Object Storage Targets (OSTs)
Object Storage Targets (OSTs)
Management Network
Metadata and Management Servers
Additional Metadata Servers
Object Storage Servers
Object Storage Servers
High Performance Data Network (Omni-Path, InfiniBand, 10/40/100GbE)
Lustre Clients (1 – 100,000+)

Figure 2: Lustre parallel file system architecture.

2 The pipeline

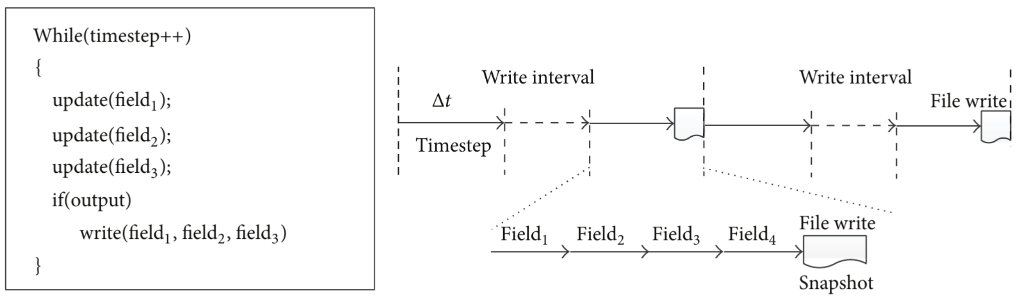
A pipeline is a work-chain where multiple specialized units work in parallel on a different production stage for a given product. When talking about software, these stages are generally different functions acting on some data and the product is the resulting data. In this particular case, we want to build a pipeline with two stages: A computational stage and a writing stage, both acting on the field variables of interest for the simulation. Moreover, the writing stage will be performed asynchronously, hence the name of Asynchronous Pipeline I/O (AP-IO) or Asynchronous I/O Pipeline.

2.1 Concept

The main goal of asynchronous execution is, as we have already learned, to leave independent tasks running in the background, on different threads, thereby overlapping the execution of multiple sets of instructions. The idea of having an asynchronous I/O pipeline is to hide the I/O latency of the code by spawning a service thread that runs in the background, in charge of writing data asynchronously while the main thread executes the rest of the code in parallel.

In most simulation codes, there is always a loop that runs for a certain number of iterations until the total simulation time has been attained. Every n time-steps or iterations, the code "dumps" or writes all the data from the simulation at that particular time-step, that is, all the values of the fields involved, probes, and other variables. This way, we can post-process it and analyse it, generate visualizations, etc.

In the original OpenFOAM solver that ASHEE uses, this is done right at the end of a loop iteration, just before the next iteration starts, and it's blocking or synchronous, meaning that execution is stopped to perform the writing output operation and it's not resumed until this one is finished, which can take a significant share of total run-time, depending on the volume of data to write.



```
While(timestep++)
{
  update(field_1);
  update(field_2);
  update(field_3);
  if(output)
      write(field_1, field_2, field_3)
}
```

Figure 3: **Left**: Pseudo-code for the original solver. **Right**: Illustration of iterative execution workflow. Extracted from [6].

However, with this alternative approach, what we do is, every time a field or any other variable is ready to be written, it is sent to write by another thread, in parallel, while the main thread keeps executing the next batch of instructions, without stopping and waiting for completion of the writing operation, as illustrated in the figure below.

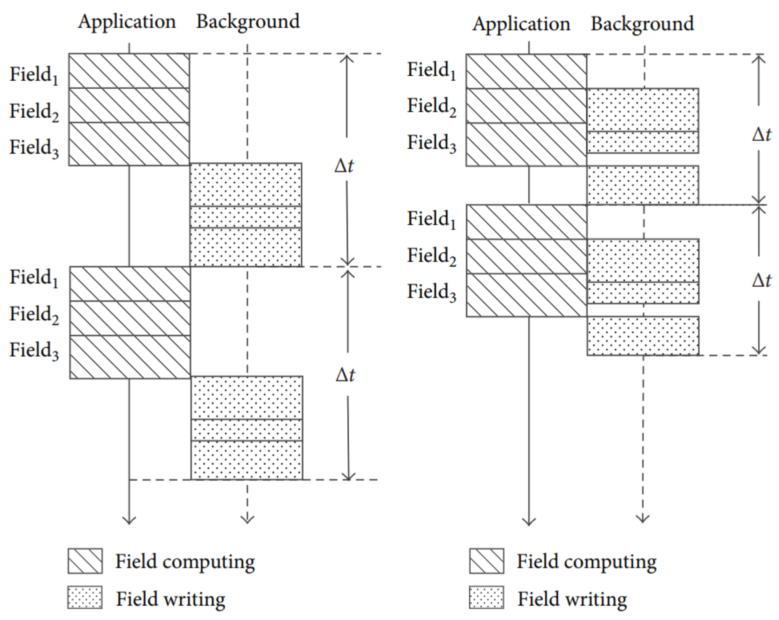


Figure 4: **Left**: Original execution scheme. **Right**: AP-IO execution scheme. Extracted from [6].

Doing this, we're able to partially hide the output latency of writing operations behind field-computation instructions charged with solving the equations for the different fields. Partially, because there is a limit to how much of the I/O workload we're able to hide and thus overlap with computational operations, and this is given by the Potential Shield Time (PST). The PST is, for every field or variable in the simulation, the maximum time available to perform the output operation before the variable needs to be updated again, as presented in Fig. 5. If the actual time it takes to write a field is greater than its PST, the main thread will have to wait until the field has been written to memory before it can update it again. Otherwise the field would be updated in the middle of its writing operation, which would mess up the output file.

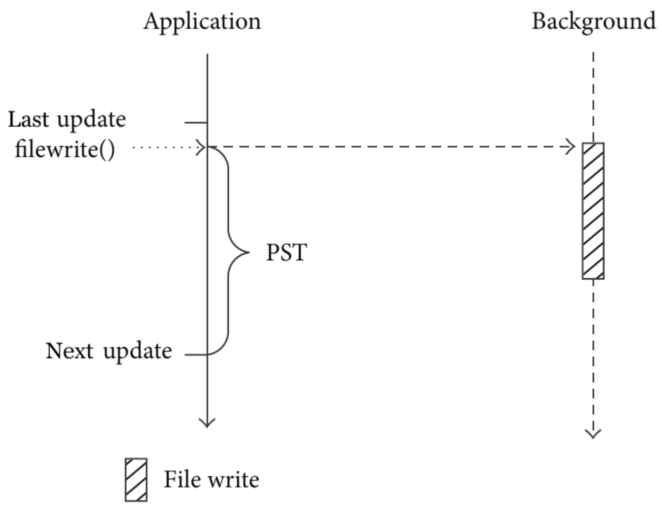


Figure 5: Potential Shield Time (PST). It is the maximum amount of time there is to hide output latency before a field needs to be updated again. Extracted from [6].

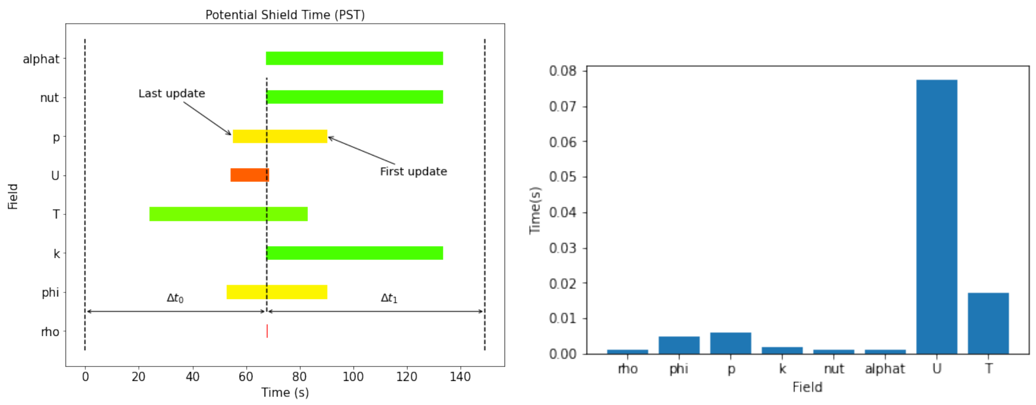


Figure 6: **Left**: PST for each and every field computed for the ASHEE code. **Right**: I/O times for every field. 32 million cells on a single node of MareNostrum (48 cores).

2.2 Implementation

For the implementation of this pipeline and in particular, for the management of parallel execution, I decided to use POSIX threads through the C++ `pthread` library, which allow for a fine-grained control of threads and are perfectly suited for simple problems with a small number of them. Unlike other shared memory, parallel programming frameworks like OpenMP, POSIX threads offer a low-level tuning that grants total control over the scheduling of threads and the synchronization between them, making them perfect for our pipeline.

The parallel scheme is pretty simple: At the beginning of the simulation, a service or writer thread is spawned, which does nothing other than taking writing jobs from a queue and executing them one after the other. In order to exploit the ability of most current processors to accommodate multiple threads (usually two) simultaneously on different logical CPUs (simultaneous multi-threading technology or SMT), this spawned thread is mapped and pinned to the spare logical CPU in every core, hoping to better spread the workload. This is done by setting the CPU affinity of the writer threads with `pthread_setaffinity_np()` to the CPU set containing only the spare logical CPU in every core. Knowing that these are assigned the last indices within every node, the set reads, for every core inside a N-core node, as `CPU_SET(rank % N + N, & cpuset)`, using C++ `CPU_SET` function (See 5. Code Snippets), where `rank` denotes the index of the logical CPU containing the main thread, which ranges from 0 to N-1. The extra logical CPUs, reserved for the writer threads, take the indices going from N to 2N-1.

Since the service thread will only be performing system calls over and over again, it won't compete for CPU resources with the main rank. However, since the heavy part of all I/O operations is taken care of by the I/O controller in the background, this doesn't really show any effect on performance. In other words, the one actually writing to disk is the I/O server, as seen on Fig. 2 for the Lustre file-system, and the CPU only sends requests to these servers for writing and reading operations. Therefore, since the time for performing these system calls is almost negligible compared to the workload of the main thread, whether the service thread is run on the same core using SMT or not, it doesn't really make a noticeable difference, though the idea in principle makes sense.

For the synchronization between threads, condition variables were used. A condition variable is an object that can be used to block or resume the execution of a thread through a system of locks and notifications. So instead of polling for a given result to be available, a thread can be notified ("woken up") by another thread when it should resume, saving CPU resources. Similarly, a thread can also block its execution and wait until notified by another thread simply using a lock over a mutex object. Just to be clear, condition variables are the ones that get notified, mutual exclusion objects protect condition variables from being accessed concurrently by different threads and locks are put on mutex objects for this purpose.

In this case, one condition variable was used for the worker thread to wait whenever the job queue is empty and be notified whenever a job is added to it (See Fig. 7 Right) plus an extra condition variable for every field, which sends the main thread to sleep (`cv.wait()`) when the field is still being written and gets notified (`cv.notify_one()`) when the output operation is over and the field can be updated again (See Fig. 7 Left). This requires the use of two separate `mutex` or mutual exclusion objects: One is used to lock the job queue to the writing thread when this one is empty through the `cv_worker` condition variable and the other locks the field variables to the main thread through their own `field_handler.cv` to prevent it from updating them while they're being written to disk.

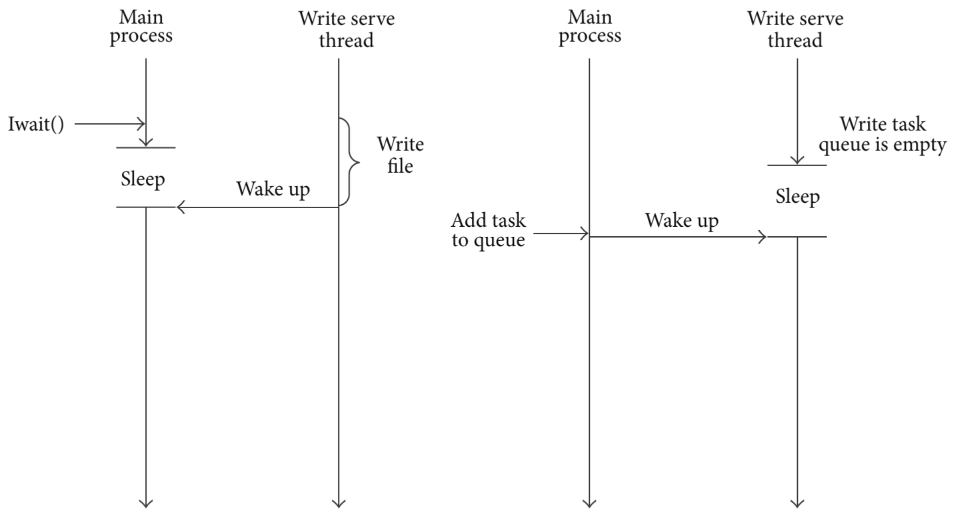


Figure 7: **Left**: writer thread has finished writing the field and notifies the main thread to resume execution (update field). **Right**: main thread notifies the "sleeping" (empty queue) writer thread that a new field is ready to be written. Extracted from [6].

3 Performance analysis

A performance analysis was carried out on two different machines, the new Hawk super-computer at HLRS, Stuttgart and the long-standing MareNostrum 4 (MN4) at BSC, in Barcelona. The Hawk machine is based on the AMD EPYC 7742, which is a 64-bit 64-core x86 server microprocessor. Every node contains two 64-core processors per node, giving a total of 128 cores per node. MN4, on the other hand, runs on Intel Skylake, in particular, Xeon Platinum 24-core processors, giving a total of 48 processors per node. Through a set of test simulations alternating both versions of the code, it was possible to calculate an approximate average for the Output Overlap (OO) attained with the AP-IO pipeline, which is given by:

$$OO = \frac{(OT - AT) \times 100}{OT} \quad \% \tag{1}$$

where OT denotes the time spent on output by the original code and AT the one spent by the AP-IO implementation, that is, not the time spent by the service thread in writing data but the time spent by the main thread in waiting for the data to be written, which corresponds to the share of all I/O time that is not hidden or overlapped. Therefore, by subtracting the non-overlapped portion of I/O to the total I/O time of the original code, we're left with the overlapped fraction, or Output Overlap (OO), as a percentage of total I/O, which is equal to the speedup on the I/O part only.

3.1 Measuring I/O

Due to the stochastic nature of I/O traffic across the network, which results in a high variability [9], measuring I/O becomes a difficult task, not obvious to deal with. At times when one or multiple users are running very I/O intensive jobs on several compute nodes, contention on the file system will increase substantially, slowing down all I/O operations. On the contrary, when there is very little I/O traffic and the I/O servers are not saturated with requests, I/O will run smoothly and performance will go up. Fig. 8. illustrates the problem. Let's imagine we have a profile for I/O traffic that looks like the one on the figure, with a large volume of I/O requests during the day that relaxes over the night and bounces back up again during day-time, totally fictitious. We can associate a frequency to this I/O pattern of one oscillation per day. Now let's say we alternate the runs for the AP-IO and the original code. If we run the AP-IO version of the code during the day and the original during night-time, we might see no performance benefit at all or even a negative effect on performance from using AP-IO. However, this has nothing to do with the pipeline implementation and concerns only the effect of disk contention. Since I/O is slower during the day, all runs doing I/O operations at daytime take longer and are thus biased with respect to those taking place at night. Nonetheless, if we were to run both versions during the day or during the night, the bias would cancel out. On Fig.8, we see four different plannings or ways of alternating the runs (A, B, C, D). Planning A is completely biased, since the alternating frequency resonates with the I/O traffic. On the other hand, planning D alternates smaller jobs, which prevents the I/O load from changing significantly from one run to the next.

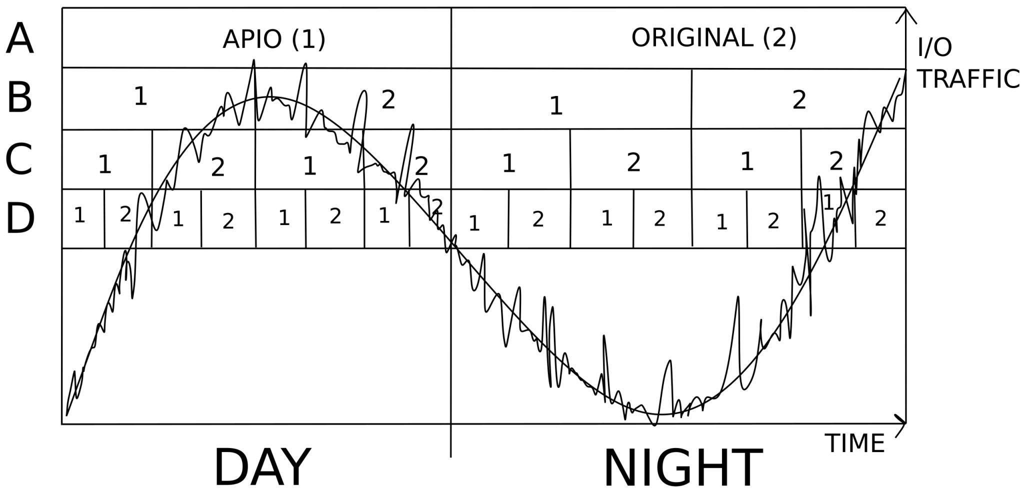


Figure 8: Made-up hypothetical example of I/O traffic over the network as a biasing factor affecting performance comparison between the AP-IO and original versions of the code.

3.2 Test simulation setup

In order to compare both versions of the code so as to calculate the speedup of using the AP-IO pipeline, several tests were carried out, of different size (number of cells) and with different number of processors across multiple nodes. Like any other finite element code, or in particular, finite-volume code like OpenFOAM, the simulation domain is transformed into a mesh made up of cells. The equations for all fields are solved for each cell and therefore, a field value is calculated for every single cell. Moreover, the blockMesh utility of OpenFOAM allows for the creation of blocks of cells, which facilitate the creation of complex domain shapes. The employed simulation setup for these tests is composed of a 9-block mesh including a central column block meant to contain the volcanic plume, featuring an inlet at the bottom and an outlet at the top, plus all the other 8 surrounding blocks to the north (N), south (S), east (E), west (W) and corners (NE, NW, SE, SW). Total system size in cells is then given by the addition of all cells within these 9 blocks, which was set to 32, 64, 128 and 256 million by choosing an initial size for the blocks and refining twice. This helped a lot to speed up the creation of the mesh. Although a parallel blockmesh generation tool is available, it only works for single-block meshes. However, refinement can easily be executed in parallel for meshes with multiple blocks.

To execute both runs under the same conditions and avoid measuring false performance differences, both versions were run on the same nodes, in an alternating fashion. Concerning the duration of the tests, it is important to choose a long enough simulated time so that the simulation reaches a stable time-step and computational time per step. Otherwise we could be getting false or unrealistic metrics. Nevertheless, to avoid the effect of I/O traffic frequency that we presented earlier, runs should not be too long either so that we can gather enough data and the change in I/O traffic from one run to another is small enough to be neglected.

3.3 Performance metrics on Hawk (HLRS)

Most of the tests were conducted on the Hawk system at HLRS. Hawk being a new system, it took some time to get it running smoothly and some of the performance metrics data is not reliable and contains outliers. This was due to an over-load of the file system. Basically, all jobs for all the different combinations of system size and number of nodes were run simultaneously, what eventually saturated the file system and led to a major slow-down of I/O performance caused by the massive contention over the storage resources. Still, under these conditions of extreme stress, some good speedups were observed.

3.3.1 Single-node

On a single-node, darshan profiler reveals an I/O writing share that reaches almost 30% of total run-time (See Fig. 9). On the standard I/O part, corresponding to file operations and metadata, the load is very light since the number of files is not significant.

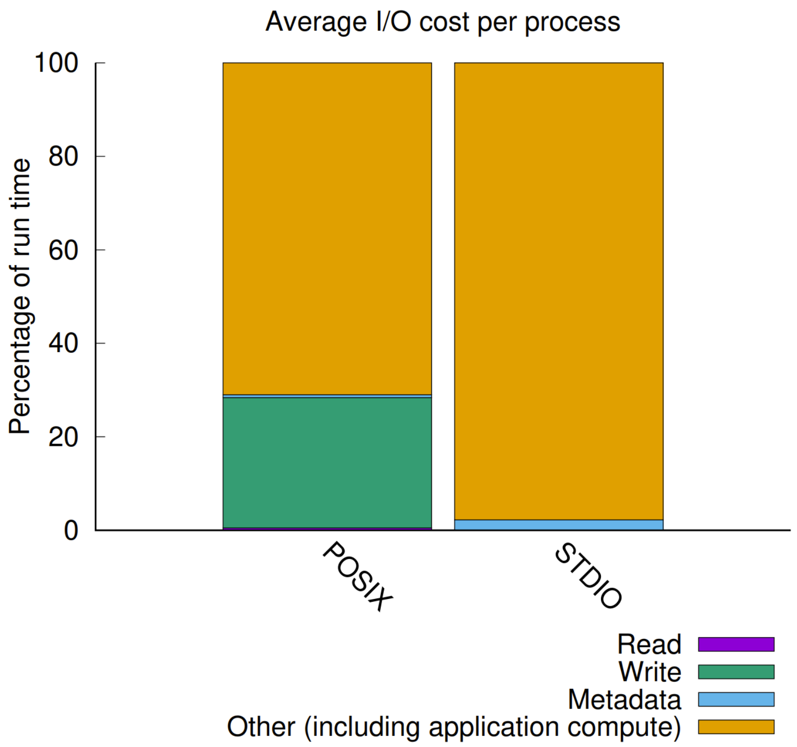


Figure 9: I/O percentage of run-time per processor for a single-node run on Hawk as given by darshan I/O profiler. I/O accounts for almost 30% of total run-time, with some variability from one run to another.

As observed on Fig. 10, 9 jobs were run simultaneously on different nodes, alternating AP-IO and original runs. The resulting timings for the AP-IO runs were much more stable than the originals, showing almost no variability from one node to the other. The speedups on total execution time lie around 19%, which, for an I/O load taking 30% of total run-time, means output overlap is 63% (19% is the speedup for total execution time, while 63% is the speedup on the I/O). An I/O speedup or output overlap of 100% would be equivalent to a 30% speedup on total run-time, equal to hiding the total share of I/O.

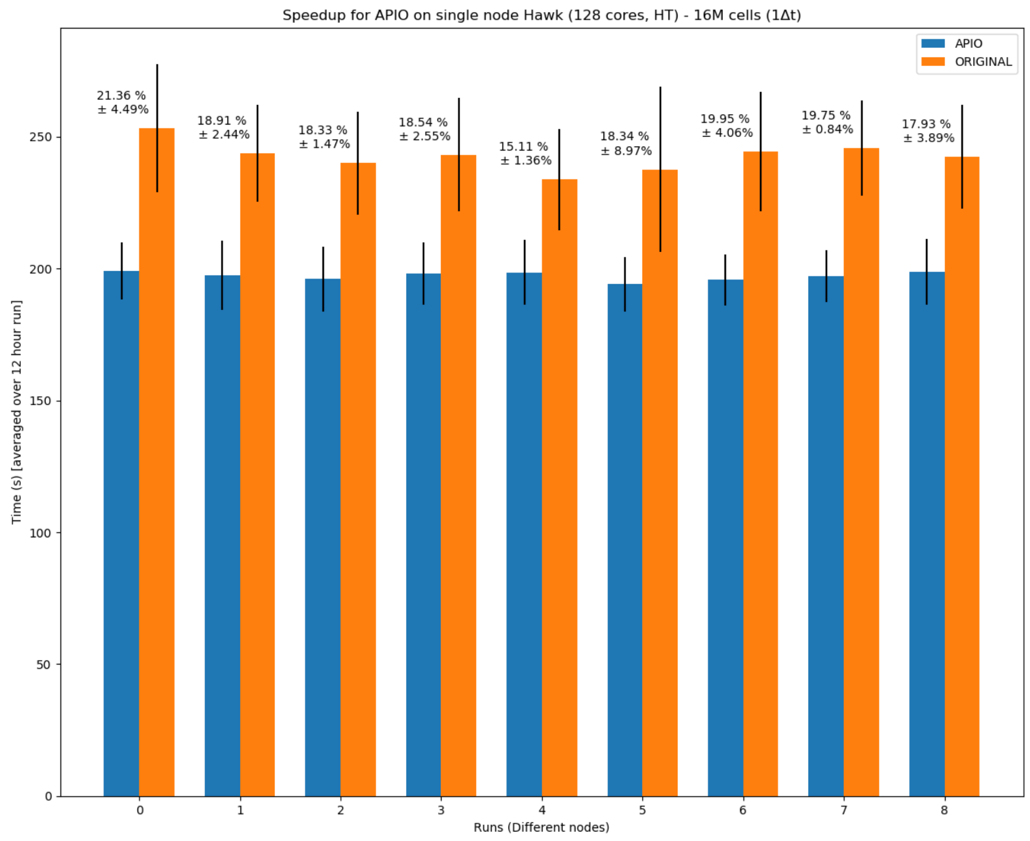


Figure 10: AP-IO / Original total execution times for single-node runs (128 procs) of 16 million cells on Hawk. Average speedup is around 19%, which means overlap is close to 63%.

On a different set of experiments, only the overlap was measured, that is, only the main thread's waiting time for the AP-IO implementation and the I/O time for the original code. Results can be checked on Fig. 11. It is noticeable that, for some reason, measurements are pretty stable up to the 8th run and then slow down, surely due to an increase in demand for the file system over the network. Nonetheless, we still observe a constant overlap of almost 50%.

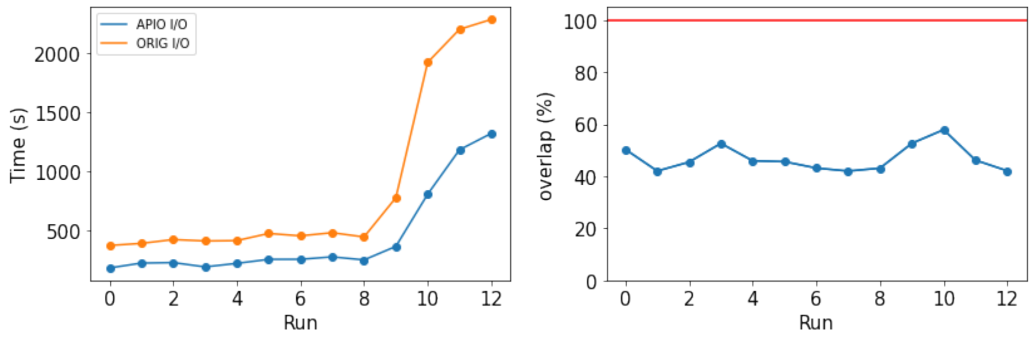


Figure 11: 32 million cells on single node of Hawk (128 procs).
Overlap: [mean: 46.84%, std: 4.82%, max: 57.86% ,min: 41.98%]

3.3.2 Multi-node

Multi-node tests were the ones most hardly hit by the over-saturation of the file system. However, we can see, as commented above, good speedups that come around 40%, discarding the outliers. As can be understood from Figs. 12 to 22, the tests were conducted in two batches: A first batch that covers all the runs up to the 10th on Fig. 12 and a second batch that takes the rest. We can observe that I/O time rapidly increases with time run after run. This is due to what was described earlier: All the jobs are put in the queue of the job scheduler, then the smaller jobs get scheduled first and the rest are gradually added as more time frames and nodes become available. Since these are all very I/O intensive jobs, the more and bigger jobs get scheduled and are running, the more pressure there is on the file system and the slower all the I/O operations are for all nodes in the cluster. This is the reason for the rapid increase in I/O time on the figures. The presence of distant outliers could have been caused by the sudden scheduling of one of the very big jobs, which might have triggered a peak in I/O activity, and its following cancellation after some iterations due to failure.

2 NODES (256 CORES)

The 32 million cells 256-core run exhibits two clear outliers in the data, one on the AP-IO for the 6th test-run and another on the original version for the 14th run. When removed from our dataset, we're left with a 39.71% average overlap.

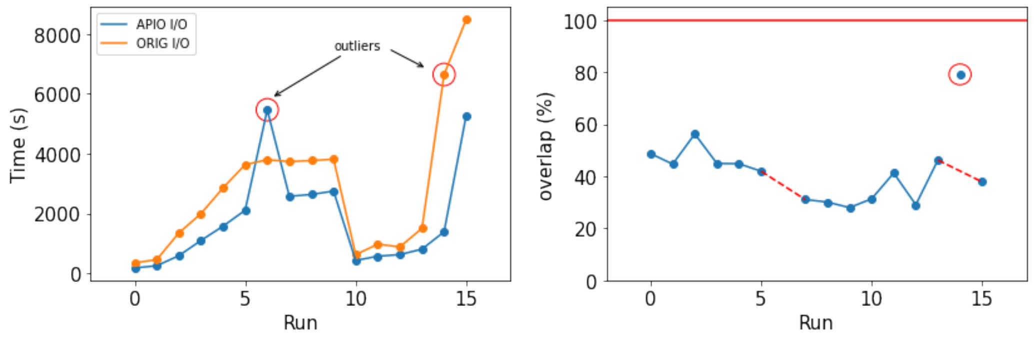


Figure 12: 32 million cells on 2 nodes of Hawk (256 procs).
Overlap: [mean: 39.71%, std: 8.34%, max: 56.23% ,min: 27.94%]

4 NODES (512 CORES)

This test suffered the effect of the same event that triggered the anomaly in the data, resulting in both the same outliers for the two batches. Overlap is a bit higher but falls within standard deviation.

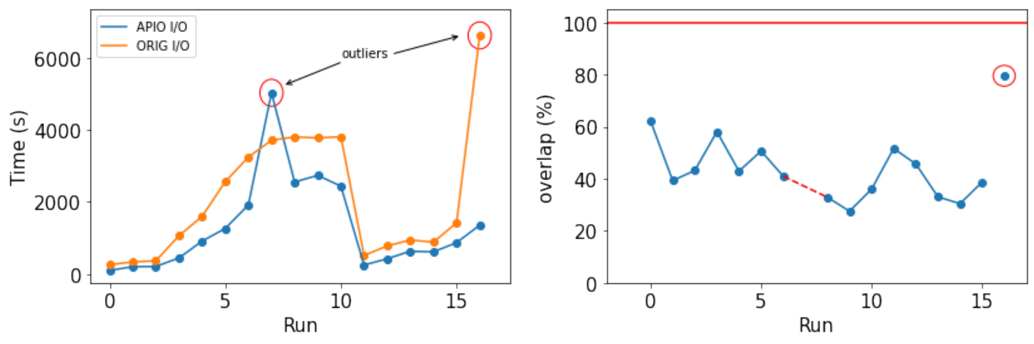


Figure 13: 32 million cells on 4 nodes of Hawk (512 procs).
Overlap: [mean: 42.21%, std: 9.69%, max: 62.15% ,min: 27.53%]

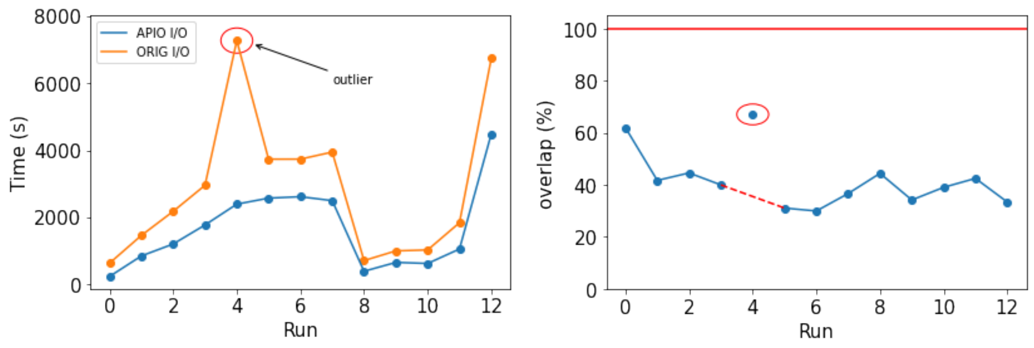


Figure 14: 64 million cells on 4 nodes of Hawk (512 procs).
Overlap: [mean: 39.91%, std: 8.17%, max: 61.90% ,min: 29.92%]

8 NODES (1024 CORES)

No outliers are visible for the 32 million cells test. However, both the other tests running on 1024 processors suffered from the same anomalies seen in the previous examples, affecting mostly one test in the first batch and another in the second, as illustrated by Fig. 17. A different kind of anomaly is present on the 128 million cells test, though, visible on run 6, where both versions exhibit a similar performance, giving a 0% overlap. Here's a possible explanation: The original version is always run after the AP-IO, so probably, run number 6 of the original code, which was executed last, ran right after some other job had finished, removing its share of pressure from the file system which, in return, accelerated I/O for that particular run, resulting in a fake better performance. It's always hard to know exactly the reason behind these crooked data points, but this kind of phenomena was observed to occur in many occasions.

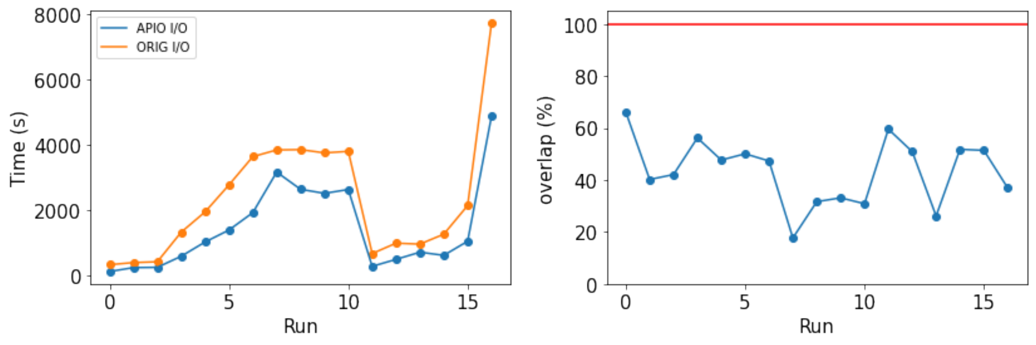


Figure 15: 32 million cells on 8 nodes of Hawk (1024 procs).
Overlap: [mean: 43.51%, std: 12.42%, max: 66.16% ,min: 17.70%]

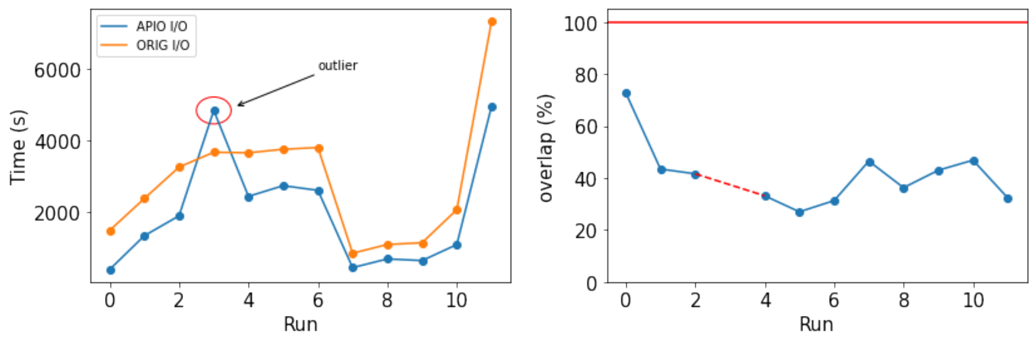


Figure 16: 64 million cells on 8 nodes of Hawk (1024 procs).
Overlap: [mean: 41.29%, std: 11.79%, max: 72.74% ,min: 26.98%]

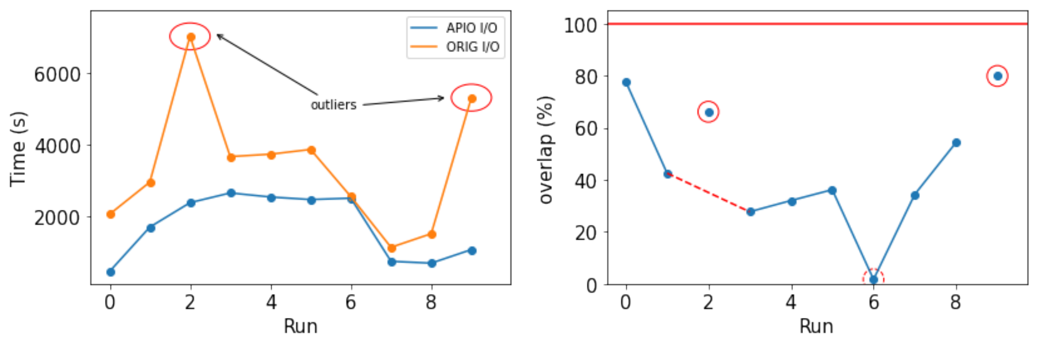


Figure 17: 128 million cells on 8 nodes of Hawk (1024 procs).
Overlap: [mean: 43.55%, std: 16.05%, max: 77.67% ,min: 27.69%]

16 NODES (2048 CORES)

The big tests like this one could only run for a limited number of times, which resulted in less data points. Both the 64 and 128 million cells tests gave a similar speedup, with exception of the 256 million cells test, which exhibits a 10% drop, perhaps not too reliable since the data looks pretty irregular.

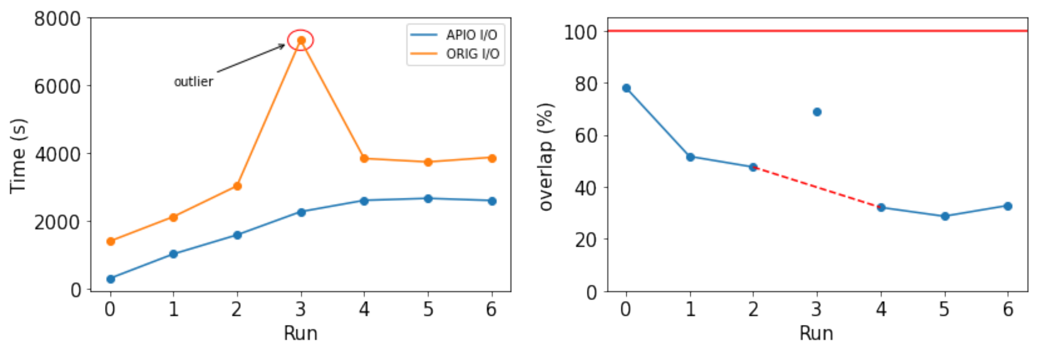


Figure 18: 64 million cells on 16 nodes of Hawk (2048 procs).
Overlap: [mean: 45.16%, std: 16.98%, max: 78.13% ,min: 28.68%]

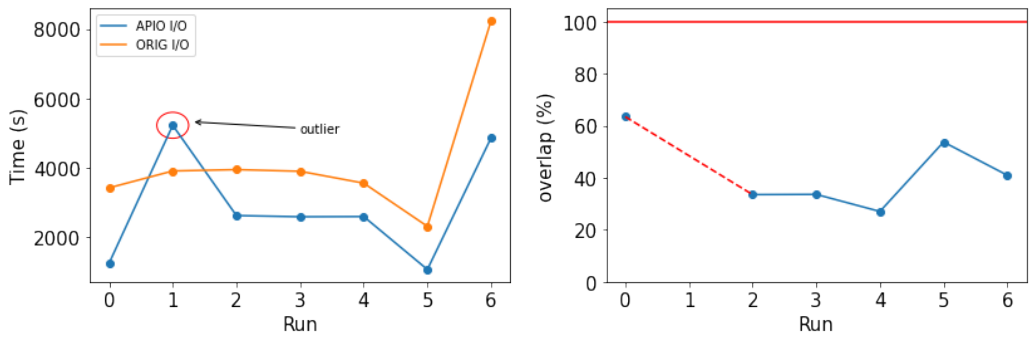


Figure 19: 128 million cells on 16 nodes of Hawk (2048 procs).
Overlap: [mean: 42.05%, std: 12.67%, max: 63.42% ,min: 27.05%]

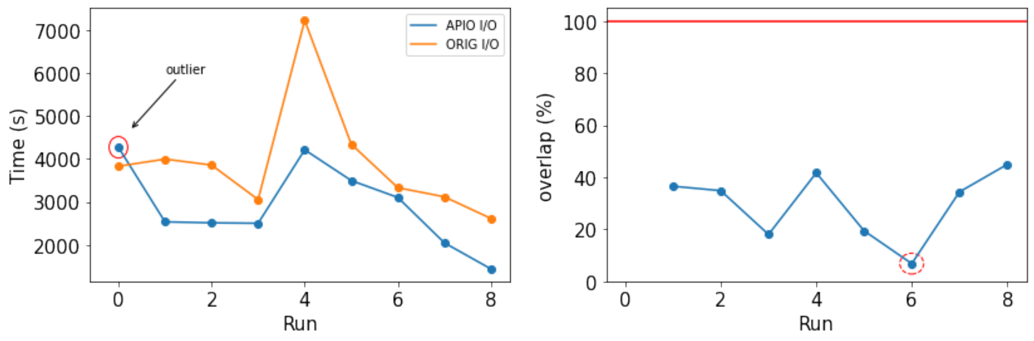


Figure 20: 256 million cells on 16 nodes of Hawk (2048 procs).
Overlap: [mean: 32.83%, std: 9.58%, max: 44.92% ,min: 18.02%]

32 NODES (4096 CORES)

This is the maximum number of nodes that could be used without failure. The test simulation containing 512 million cells was not possible to conduct either due to system failure. Still, we observe a similar speedup that gets degraded though for the 256 million cells system, just like it did on 16 nodes too.

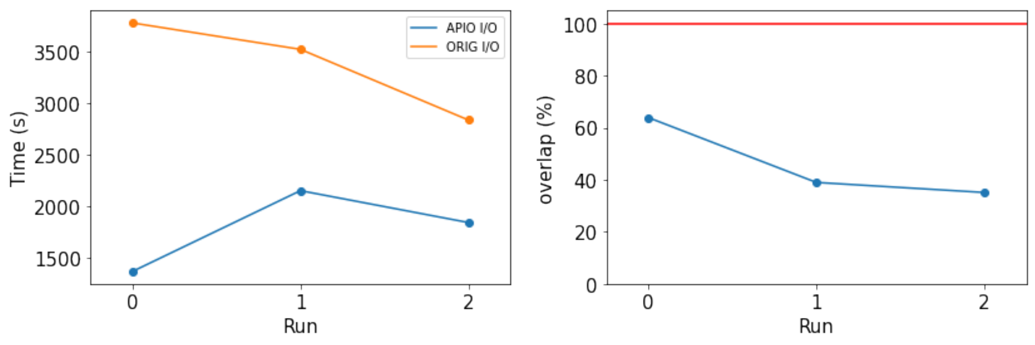


Figure 21: 128 million cells on 32 nodes of Hawk (4096 procs).
Overlap: [mean: 45.98%, std: 12.75%, max: 63.88% ,min: 35.11%]

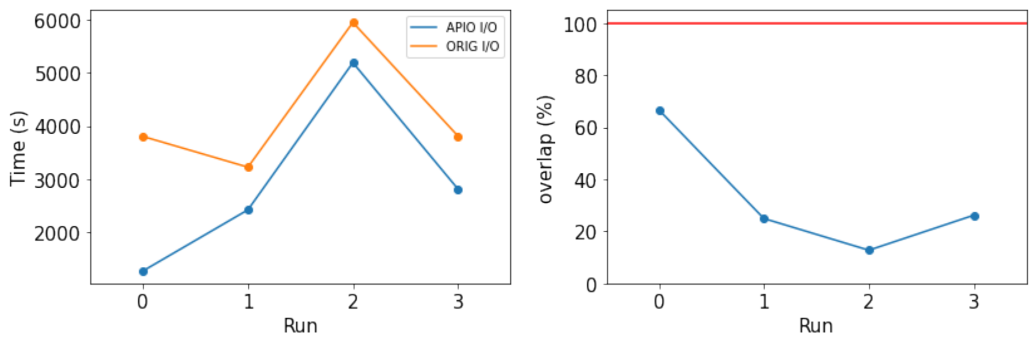


Figure 22: 256 million cells on 32 nodes of Hawk (4096 procs).
Overlap: [mean: 32.63%, std: 20.34%, max: 66.65% ,min: 12.74%]

3.4 Performance metrics on MareNostrum (BSC)

All tests conducted on MareNostrum 4 resulted in very stable, clean metrics. However, being an external machine from a different super-computing center (BSC in Barcelona), CPU hours were very limited and massive tests were too expensive. Also, SMT (Simultaneous Multi-Threading) is disabled in all nodes except for only four and later just three, which were the only ones used for this work.

3.4.1 Single-node

Single node performance on MN4 is extremely high, reaching almost a 100% speedup for the I/O, that is, a complete overlap. Nothing surprising though, if we look at Fig. 6, right at the beginning of this thesis, where PST and I/O were calculated for 32 million cells runs on a single node of MN4. From this figure, we can deduce that there will be no waiting time at all, since the PST for every field is large enough to hide all writing operations completely. The data for PST and writing times by field was not collected for the multi-node setups, but it's something worth studying analytically in later work to calculate theoretical overlaps. Turning to Fig. 23, it is surprising to see how small is the I/O percentage of total run-time this time. On Hawk, we were looking at a painful 30% of I/O, and now it's come to only around 5% for MareNostrum. Now, a possible cause for this huge difference might come from the file system. As commented above, the Hawk machine is a new system that has just been installed recently, but the file-system wasn't upgraded, which might have caused a mismatch between the new system and the old file-system, conceived for a smaller number of processors. So perhaps the most reasonable explanation for this is that the file-system is not large enough to manage the requests of the many times more processors per node compared to the old Cray machine that was installed at HLRS, and this is why I/O is slower for Hawk and thus way faster for MareNostrum 4, in comparison [10]. Standard I/O and metadata takes almost no time for the same reason Hawk did, the number of files is insignificant.

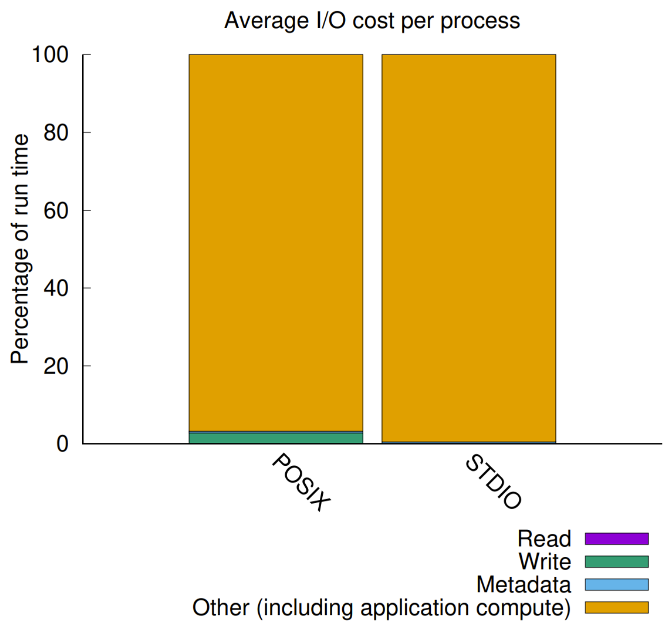


Figure 23: I/O percentage of run-time per processor for the single-node run on MN as given by darshan I/O profiler. I/O accounts for only 5% of total run-time.

Looking at total execution time, we see an average speedup of 5%, which amounts for the totality of I/O, meaning that it's achieving an almost 100% overlap. At this stage of the work, it was possible to run on 4 nodes simultaneously, since these were the only hyper-threaded nodes available, then it went down to 3, which is why the next overlap tests were only taken up to 3 nodes.

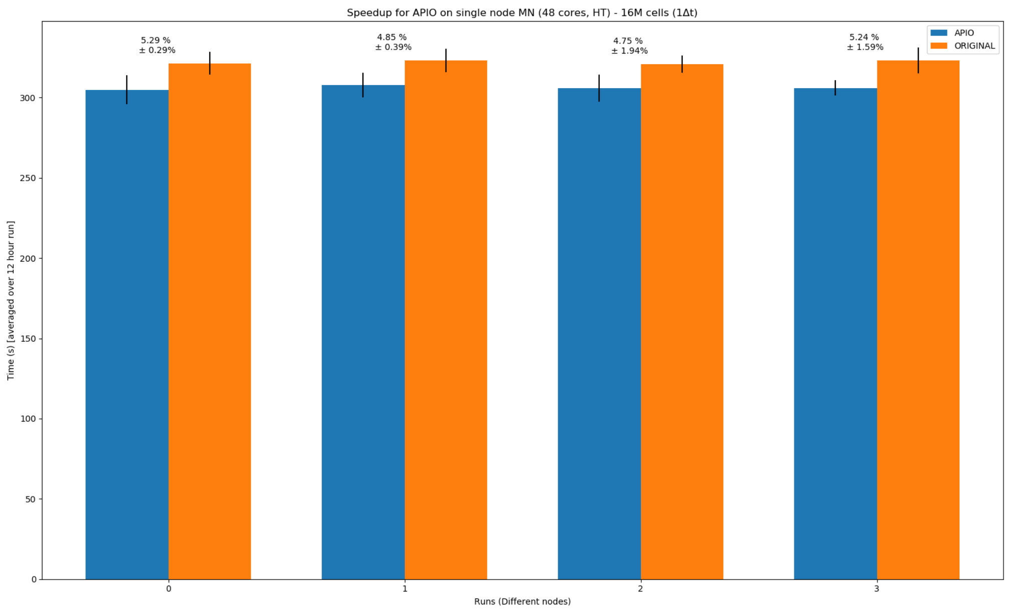


Figure 24: AP-IO / Original total execution times for single-node runs (48 procs) of 16 million cells on MN. Average speedup is around 5%, which means overlap is close to 100%.

When studying only the I/O times for single-node, we corroborate our results on total execution time, obtaining an overlap of 98%, very close to 100%.

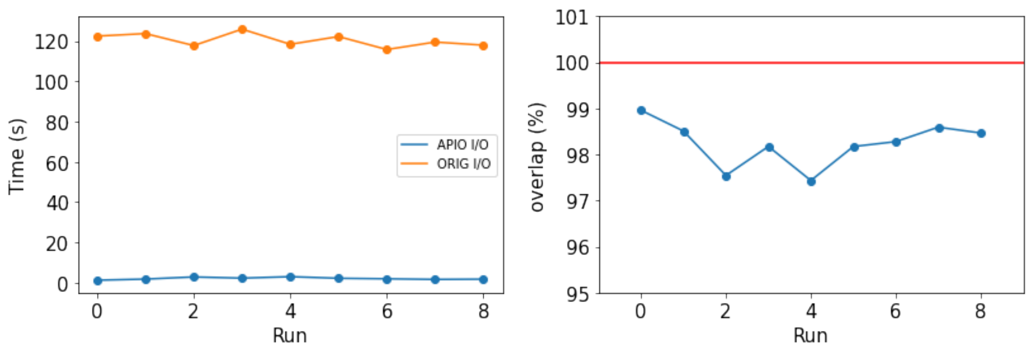


Figure 25: 32 million cells on single node of MareNostrum (48 procs).
Overlap: [mean: 98.24%, std: 0.46%, max: 98.96% ,min: 97.43%]

3.4.2 Multi-node

Multi-node metrics for MareNostrum reveal a sustained stability over many runs, which suggests that the file system is providing a good service and responds properly to the network's demand.

2 NODES (96 CORES)

Both bi-nodal test runs for 32 and 64 million cells on MN4 yield a performance gain on the I/O equal to 49%, equivalent to halving the time spent on I/O, just like they did on Hawk, which looks very promising. The flat curves from Fig. 26 might be the result of a night-time job, while the times on Fig. 27 may have run at day-time, when the network is busier and the I/O throughput is hurt, causing a 5 fold increase on the I/O latency but maintaining the same speedup.

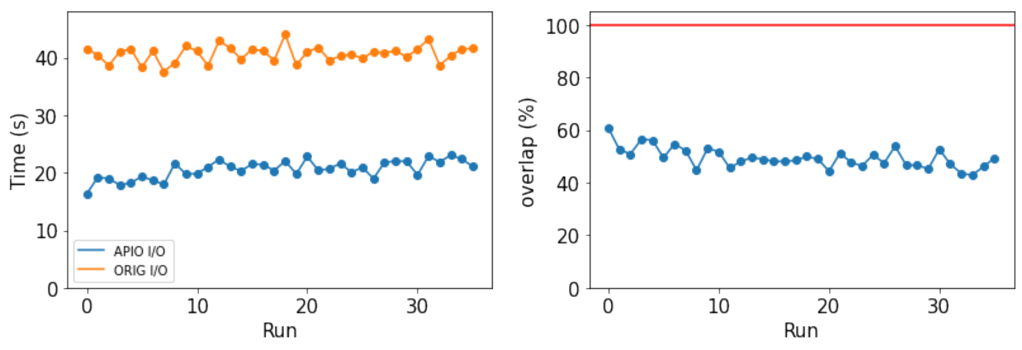


Figure 26: 32 million cells on two nodes of MareNostrum (96 procs).
Overlap: [mean: 49.41%, std: 3.87%, max: 60.76% ,min: 42.80%]

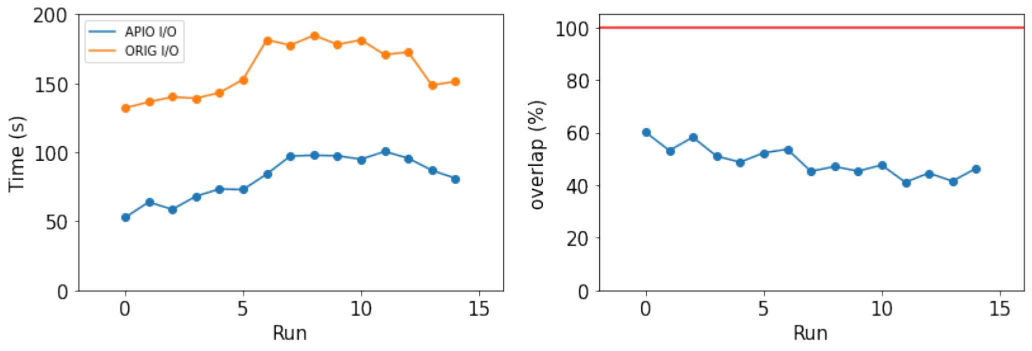


Figure 27: 64 million cells on two nodes of MareNostrum (96 procs).
Overlap: [mean: 49.04%, std: 5.41%, max: 60.09% ,min: 41.07%]

3 NODES (144 CORES)

A 6% increase in speedup is observed for the 3 node test, which falls almost within the standard deviation, but could also be explained by a reduction in the number of cells per processor.

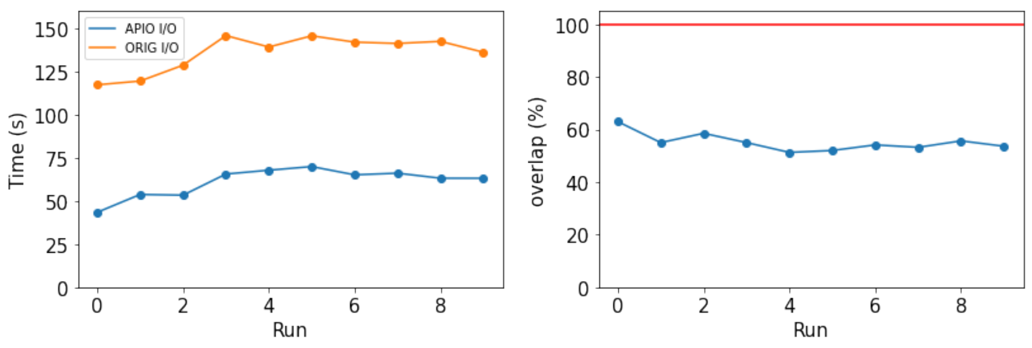


Figure 28: 64 million cells on three nodes of MareNostrum (144 procs).
Overlap: [mean: 55.14%, std: 3.24%, max: 62.98% ,min: 51.27%]

4 Conclusions

From the performance analysis, we conclude that the AP-IO pipeline actually succeeds in hiding a significant part of the output latency by executing all I/O operations in an asynchronous manner. Even under extreme stress conditions for the file-system, we observe overlaps that reach an average of 35 to 45 % for both multi and single-node runs on Hawk.

On the MareNostrum 4 system, however, single-node tests give an almost 100% overlap, while multi-node runs settle at around 50%. Measured performance metrics for AP-IO look generally independent of system size and number of processors, though overlap is slightly reduced for a large number of processors. This could be due to the massive number of files and metadata generated, which grows linearly with the number of computing cores used.

Regarding the future work, it would be interesting, as discussed in section 3.4.1, to do an analytical study of theoretical overlap by looking at the PST and writing times of every field for a multi-node setup. Basically, calculate an estimation of the accumulated waiting time based on the the PST available for writing and the actual writing time for every field. Other than that, it'd also be interesting to follow the directives presented in [6] to change the order in which fields are computed so as to maximize the total PST. This was considered impossible for this case, or at least too complicated to bother, but could potentially lead to great improvements. Moreover, it'd be very useful to make this pipeline compatible with other I/O tools in OpenFOAM like the collated format or the adiosFoam module [11].

5 Code snippets

```
field_handler f_rho("rho"),f_rho_0("rho_0"), f_k("k"),f_k_0("k_0"),
    f_p("p"),f_p_0("p_0"),f_phi("phi"),f_phi_0("phi_0"), f_U("U"),
    f_U_0("U_0"),f_T("T"),f_nut("nut"),f_alphat("alphat");

std::queue<field_handler*> queue;
std::condition_variable cv_worker;
std::atomic<bool> running {true};

std::thread writer (writeFromQueue,rank,std::ref(running),std::ref(
    queue),std::ref(cv_worker),std::ref(mesh));

cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(rank%48 +48, &cpuset);
pthread_setaffinity_np(writer.native_handle(),sizeof(cpu_set_t), &
    cpuset);
```

Snippet 1: `apio_init.H`

```
#ifndef APIO_H
#define APIO_H

#include <iostream>
#include <queue>
#include <string>
#include <condition_variable>
#include <mutex>
#include <thread>
```

```
#include <atomic>
#include <vector>
#include <mpi.h>

std::mutex mtx1,mtx2;

double startio,endio;
double iotime=0;
double alliotime;

struct field_handler
{
    word f;
    std::string state;
    std::condition_variable cv;
    field_handler(const char * field) : state("WRITE_FINISHED"),f(
    word(field)){}
};

void writeFromQueue(int rank, std::atomic<bool>& running,std::queue<
    field_handler*>& queue,std::condition_variable& cv_worker,
    objectRegistry& mesh)
{
     while(running)
     {
          if(!queue.empty())
          {
               field_handler* field = queue.front();
        queue.pop();
```

```
      field->state="WRITING";
      std::lock_guard<std::mutex> lck(mtx2);
      mesh.getObjectPtr<regIOobject>(field->f,false)->write();
      field->state="WRITE_FINISHED";
      field->cv.notify_one();
        }
        else
        {
          std::unique_lock<std::mutex> lck(mtx1);
            cv_worker.wait(lck,[&]{return !queue.empty()||!running
    ;});
        }
    }
}

void Iwrite(std::vector<field_handler*> list, std::
    condition_variable& cv_worker, std::queue<field_handler*>& queue)
    {
     startio=MPI_Wtime();
     for (field_handler* f : list)
     {
         f->state="WAIT_WRITE";
         std::lock_guard<std::mutex> lck(mtx1);
         queue.push(f);
         cv_worker.notify_one();
     }
     endio=MPI_Wtime();
     iotime+=endio-startio;
}
```

```

void Iwait(std::vector<field_handler*> list)
{
    startio=MPI_Wtime();
    for (field_handler* f : list)
    {
        std::unique_lock<std::mutex> lck(mtx2);
        f->cv.wait(lck,[&]{return f->state=="WRITE_FINISHED";});
    }
    endio=MPI_Wtime();
    iotime+=endio-startio;
}

#endif
```

Snippet 2: `apio.H`

```
running=false;
cv_worker.notify_one();
writer.join();
```

Snippet 3: `apio_terminate.H`

References

[1] *ChEESE*. URL: `https://cheese-coe.eu/`.

[2] M. Cerminara, T. E. Ongaro, and L. Berselli. "ASHEE: a compressible, equilibrium-Eulerian model for volcanic ash plumes". In: *Geoscientific Model Development* 9 (2015), pp. 697–730.

[3] *IME*. URL: `https://www.ddn.com/products/ime-flash-native-data-cache/`.

[4] William F. Godoy et al. "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management". In: *SoftwareX* 12 (2020), p. 100561. ISSN: 2352-7110. DOI: `https://doi.org/10.1016/j.softx.2020.100561`. URL: `http://www.sciencedirect.com/science/article/pii/S2352711019302560`.

[5] Ning Liu et al. "On the Role of Burst Buffers in Leadership-class Storage Systems". In: Apr. 2012. DOI: `10.1109/MSST.2012.6232369`.

[6] Ren Xiaoguang and Xu Xinhai. "AP-IO: asynchronous pipeline I/O for hiding periodic output cost in CFD simulation". In: *TheScientificWorldJournal* 2014 (2014), p. 273807. ISSN: 2356-6140. DOI: `10.1155/2014/273807`. URL: `https://europepmc.org/articles/PMC3997917`.

[7] *TOP500*. 2020. URL: `https://www.top500.org/lists/top500/list/2020/11/`.

[8] *InfiniBand*. URL: `https://www.nvidia.com/en-us/networking/products/infiniband/`.

[9] J. Lofstead et al. “Managing Variability in the IO Performance of Petascale Storage Systems”. In: *SC ’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–12. DOI: `10.1109/SC.2010.32`.

[10] Yuan Wang et al. “Performance Evaluation of A Infiniband-based Lustre Parallel File System”. In: *Procedia Environmental Sciences* 11 (Dec. 2011), pp. 316–321. DOI: `10.1016/j.proenv.2011.12.050`.

[11] Karl Meredith, Mark Olesen, Norbert Podhorszki. *Integrating ADIOS into OpenFOAM for Disk I/O*. esi-group. 2016. URL: `https://openfoam.com/documentation/files/adios-201610.pdf`.