I/O performance of multiscale finite element simulations on HPC environments

Francieli Boito, Antonio Tadeu A. Gomes, Louis Peyrondet, Luan Teylo

I/O performance of multiscale finite element simulations on HPC environments

Francieli Boito[1], Antonio Tadeu A. Gomes[2], Louis Peyrondet[1], Luan Teylo[1*]
[1]*Univ. Bordeaux, CNRS, Bordeaux INP, INRIA, LaBRI, UMR 5800, F-33400 Talence, France*
{francieli.zanon-boito, louis.peyrondet, luan.gouveia-lima}@inria.fr
[2]*Laboratório Nacional de Computação Científica (LNCC), Av Getúlio Vargas 333, 25651-075, Petrópolis-RJ, Brazil*
atagomes@lncc.br

***Abstract*—In this paper, we present MSLIO, a code to mimic the I/O behavior of multiscale simulations. Such an I/O kernel is useful for HPC research, as it can be executed more easily and more efficiently than the full simulations when researchers are interested in the I/O load only. We validate MSLIO by comparing it to the I/O performance of an actual simulation, and we then use it to test some possible improvements to the output routine of the MHM (Multiscale Hybrid Mixed) library.**

***Index Terms*—High-performance computing, parallel I/O, numeric library, I/O kernel, mini-app, multiscale simulation**

I. Introduction

In high-performance computing (HPC) systems, the supercomputers, parallel file systems (PFS) allow for accessing persistent data in the form of files. Such a file system is deployed on a set of dedicated machines, usually with access to arrays of storage devices, and accessible to the compute nodes through the network. When reading or writing from/to a file, applications will generate requests to the PFS.

It is well known that the performance applications observe when accessing a PFS depends on the way they generate these requests, which is called their *access pattern*: how many files are accessed, are they accessed from beginning to end, contiguously or randomly, in requests of what size, etc. The access pattern is important because it determines the success of caching and prefetching techniques, the performance of accessing the storage devices, the importance of the costs of generating requests through the network, metadata performance, the number of servers that need to be contacted, etc [1]–[6].

A lot of research has focused on improving the I/O stack — file system, I/O forwarding software, I/O libraries, etc — to transparently improve performance, i.e. without requiring any modifications in the applications [7]–[12]. Still, it remains important that applications perform I/O in an adequate way in order to achieve the highest performance from the PFS. That means that in some cases, for applications that are important (because they are executed very often, or for a long time), the effort of studying and optimizing their I/O routines is justified.

In this paper, we study the output performance of multiscale simulations. More specifically, we consider the class of simulations based on multiscale finite element (FE) methods. From a mathematical viewpoint, these methods are formulated in terms of: (i) a global problem defined in a coarse mesh; and (ii) a collection of local problems, one for each element of the coarse mesh, driven by the physics of the phenomenon being simulated [13]. In terms of software, this formulation implies an architectural style in which local problems are embarrassingly parallel, and the global problem is assembled and solved based on input coming from the solution of all the local problems.

Even though computing the local problems can easily benefit from massively parallel HPC systems, persisting the solution of the local problems may incur considerable overhead to the PFS, because of the simultaneous write operations being requested from a potentially huge number of threads. This situation may happen because of three different potential bottlenecks in a PFS: (i) threads within a same node synchronizing on system calls for writing to the file system; (ii) threads from all nodes asking the metadata servers for file creation/information; and (iii) threads from all nodes synchronizing on writing to the file servers when striping is not judiciously configured. Running actual multiscale simulations to study the behavior of the PFS is prohibitive, because the amount of computation is not directly related to the number and volume of write operations.

In this paper, we discuss the development of an I/O *kernel*—a benchmarking tool that mimics the actual application behavior while focusing only on its I/O routines—targeted at multiscale simulations. The so-called MSLIO tool bypasses expensive computing and inter-process communication operations and reduces the software dependencies, thereby allowing us to evaluate ideas for performance improvements faster and cheaper. Another advantage of such tool is that we are able to evaluate its performance when increasing application parameters beyond what we would be capable of testing in our infrastructure (due to memory or time constraints). **The MSLIO I/O kernel represents an important contribution of this work, as it is publicly available** and can be used by others when evaluating their I/O optimization techniques, such as I/O scheduling, improvements to file system or I/O libraries, etc.

We discuss the I/O behavior of multiscale simulations, and

[*]The authors are presented in alphabetical order. Author contributions are detailed at the end of this document.

we illustrate the usefulness of our I/O kernel by using it to propose and evaluate different access pattern adaptations aiming at improving their I/O performance. As a case study, we used the family of Multiscale Hybrid Mixed (MHM) FE methods, which have been used for modeling and simulating several different multiscale physical phenomena [14]–[18]. Our study was conducted in two systems, which use two widely popular PFS: BeeGFS[1] and Lustre[2].

The remainder of this paper is organized as follows. The next section details the multiscale simulations we study, more specifically the MHM library we considered, and Section III presents the MSLIO I/O kernel. The following Section IV discusses the different optimizations implemented on MSLIO. Then, Section V discusses the experimental methodology, and results are presented in Section VI. Finally, Section VII presents related work, and Section VIII states conclusions and future work perspectives.

II. The MHM library for multiscale simulations

In this section, we discuss the I/O behavior of multiscale simulations, using the software library that implements the MHM methods as a frame of reference. This software library is currently based on two parallelization technologies: MPI and OpenMP. A typical execution using this library starts with a reading phase where each MPI process reads data such as the input mesh, the physical coefficients, and the numerical parameters associated with MHM (polynomial approximation degrees, level of local problem refinement, etc).

During the initialization phase, folders are created for each local problem that will be solved. The number of local problems follows the number of elements in the input mesh. In each of these folders, three files will be created logging information about that local problem: the geometry of the element in the input mesh, the numerical parameters, and a local refined mesh generated automatically by the local problem over the geometry of the element in the input mesh. Then, at every N timesteps (or only once in the case of a stationary problem), an output routine will write an additional file in each folder containing the solution found so far for that local problem. All activity on these per-local problem folders is done in parallel by multiple OpenMP threads, with each file being accessed only by a single thread.

Those solution files, written during the output phase, are valuable for researchers, which will be interested in rendering visualizations of the solution using tools such as Paraview or Ensight. **In this paper, we chose to focus on the I/O performed by the output phase**, because it is the one with the most potential for impact on these simulations: the larger the input mesh and the local refined meshes generated by the local problems, the longer the time spent on I/O in the output phase. Also, for a transient problem, which is solved in a number of timesteps, the output phase will be repeated multiple times, while the other I/O operations happen only once, during initialization.

[1]https://www.beegfs.io/

[2]https://www.lustre.org/

To study the library, we considered the two-dimensional numerical simulation of fluid flows in porous media with multiscale behavior due to heterogeneous diffusion coefficients [14].

III. The MSLIO I/O Kernel

We developed an I/O kernel to mimic the I/O behavior of multiscale simulations, called MSLIO. More specifically, it is based on the MHM library, discussed in the previous section. Having this I/O kernel greatly facilitates the task of studying and improving the MHM library. It is also very useful for the HPC community, as it provides an easy and efficient way to generate a realistic workload that would be caused by executing these simulations. MSLIO is open-source and publicly available at https://gitlab.inria.fr/lpeyrond/mslio.

MSLIO is an MPI+OpenMP application that mimics a single output phase of the library by writing randomly-generated data. It receives two arguments: *ccross* and the *number of sub-elements*. *ccross* determines the number of local problems, and consequently of generated solution files, as discussed in the previous section. The number of files is given by 4^{ccross}. The number of sub-elements represents the level of refinement of the local mesh generated by each local problem, and it impacts the amount of data sent to each file. Table I illustrates the impact of these parameters in the number of files and amount of written data.

Just like in the MHM library, the local problems are distributed among the MPI processes in chunks (each rank is responsible for the same number of "contiguous" local problems): for example, if we had ten local problems in two processes, process one would be responsible for local problems 0 to 4 and process two for 5 to 9. An OpenMP *parallel for* iterates over the local problems of each process.

Data written to the solution files come from a previously-filled array in memory (filled with random doubles in the case of MSLIO). In this array, data is separated by solution point, each point being represented by values in multiple dimensions which are stored contiguously in memory. However, the solution file is separated in per-dimension portions. That means the data layout in memory does not match the data layout in the file, which complicates the output routine. We did not explore the possibility of rearranging data because it would require deeper modifications in the compute phases of the application, which may increase their execution time. Moreover, these simulations require large amounts of RAM memory, which makes copying the array prohibitive.

The output files are in text format (due to compatibility with the currently used workflow), and each value is written to the file by a *fprintf* call (which formats it into 13 characters). At the end, the code reports the elapsed time in the output phase, and also the cumulative time spent on open, write, and close operations (the sum across calls and threads). The time to create the folders is not included in the time reported by MSLIO, because they are created in the initialization phase of the actual simulations.

TABLE I: Number of files and total amount of data written in each of the parameter combinations used in this study. For a given number of sub-elements, the amount of data written to each file is the same. Only stdio write operations are performed by MSLIO, each of them for 14 bytes (except when buffering). Each file is opened and closed only once.

ccross	**3**		**4**				**5**				**6**			
sub-elements	**64**	**128**	**32**	**64**	**128**	**1024**	**32**	**64**	**128**	**512**	**32**	**64**	**128**	**256**
total data (MB)	1.7	6.6	1.8	6.8	26.6	1668.9	7.1	27.2	106.4	1673.8	28.5	108.9	425.8	1683.5
number of files	64		256				1024				4096			

ccross	**7**			**8**			**9**
sub-elements	**32**	**64**	**128**	**32**	**64**	**128**	**32**
total data (MB)	113.9	435.7	1703.2	455.8	1742.8	6815.7	1823.2
number of files	16384			65536			262144

IV. Studied optimizations

To illustrate the usefulness of our MSLIO I/O kernel, after an initial study of the behavior of the output routine of the MHM library, we proposed the following optimizations, which were developed into the MSLIO code. Results obtained with them will be discussed in Section VI

A. MPI-IO single-file version

As discussed in Section II, at every output phase a solution file is written for each local problem. The number of local problems increases exponentially with the *ccross* parameter, meaning that it is usual to have tens to hundreds of thousands of different files being created at this point. This induces a heavy metadata load which may impair I/O performance [19].

Therefore, we aimed at combining all solution files into a single shared one. Indeed, when using the MHM library, users typically employ a post-processing script to combine the multiple files into a single one before visualizing results.

For a first implementation, in order to evaluate its impact on performance, we used MPI-IO to access the shared file, with each original file corresponding to a contiguous chunk in the shared one. If this is successful in improving I/O performance, the next step would be to rewrite the output to follow some format that is adequate for visualization, such as HDF5.

B. Buffering

We observed the application performs I/O by calling *fprintf* once per double. That call comes from the C library, which employs a buffer before actually sending data to the file. Additional caching may be performed by the PFS client. Still, this behavior may result in an access pattern of small requests, which tend to perform poorly because of fixed costs associated with each request issued to the PFS [2], [3], [19].

We wanted to test more aggressive buffering, and to have it even when replacing the *fprintf* calls by MPI-IO calls (for the single-file version). In this optimization, the kernel allocates a buffer of configurable size, and then uses it to accumulate data to be written on a per-file basis—or per-local problem basis, in the case of a shared file.

V. Experimental Methodology

Experiments were conducted in two different systems. The first one is the Bora cluster from PlaFRIM[3], which is a platform from the Inria research center in Bordeaux, France. Each Bora node is powered by two 18-core Intel Xeon processes, 192 GB of RAM and runs CentOS7.6.1810 (kernel v3.10.0-957.el7.x86 64). All PlaFRIM nodes share access to a BeeGFS v.7.2.3 deployed over two storage servers, each with four Object Storage Targets (OST) and one MetaData Target (MDT). The total available data storage of the system is 131 TB, where each OST uses 12 Toshiba AL15SEB18EQY HDDs of 1.8 TB and 10.0000 RPM (organized in RAID-6). Bora nodes are connected to the file system through a 100 GBit/s Omnipath network. A recent paper [4] describes this system's I/O infrastructure performance in detail. As recommended by the authors, we used striping across all eight storage targets for all accessed files, and eight compute nodes were used to run the simulations, with two processes per node and 18 threads per process (one thread per physical core).

We also used the Santos Dumont (SDumont) supercomputer[4], from the LNCC, in Brazil. The used nodes are from the *cpu* partition, which are each powered by two Intel Xeon E5-2695v2 Ivy Bridge processors at 2.4 GHz and 24 cores (twelve per processor). Each of these nodes has 64 GB DDR3 RAM, and they are interconnected through Infiniband FDR (56 Gbps) on a fat-tree topology. The Lustre PFS version 2.1 is deployed with one MDS (Metadata Server) and ten OSS (Object Storage Servers), each with one OST (Object Storage Target), for a total storage capacity of 1.7 PB. Clients use the version 2.4.3 of the Lustre client. This machine's I/O infrastructure was studied by Bez et al. [20]. By default, its Lustre deployment does not stripe files (each file is fully stored in a single storage target). However, in our experiments we configured striping on four OSTs (keeping the default stripe size of 1 MB) to parallelize accesses to the single file (in that version of MSLIO) and used four compute nodes to run the simulation, with two processes per node and 12 threads per process (one per physical core).

It is important to notice that MSLIO requires no special configuration to be used in different systems. It simply has to be compiled, and its only dependency is MPI. In both systems, OpenMPI 3.1.4 was used. MSLIO was compiled using GCC 9.3.0 and the same optimization flag as the library: $-O3$. To measure time spent on the output routine of the MHM library, it was modified to obtain and display this information, always

[3]https://www.plafrim.fr/

[4]https://www.top500.org/system/178569/

making sure not to generate extra I/O operations during the profiled phenomena. Each experiment was repeated multiple times, and the arithmetic mean is presented with error bars representing the standard deviation. During each experimental campaign, a list of multiple different experiments was generated and then randomized before execution, aiming at minimizing possible ordering effects and impact from external activity. This was important because we cannot completely eliminate interference from activity by other users of these systems. Only one experiment is executed at once, and a wait time of at least two minutes is kept between consecutive tests to avoid possible "warm-up" effects.

In the following, all results are presented through plots. The numbers can be found in the Appendix.

VI. Performance Evaluation

In the previous sections, we detailed how we developed the MSLIO I/O kernel that mimics the output phase of the MHM library, picked as a representative of multiscale simulations. In the next section, we start by validating MSLIO by comparing its performance to an actual simulation. Then, in the following sections, we illustrate its usefulness by using it to evaluate the optimizations discussed in Section IV.

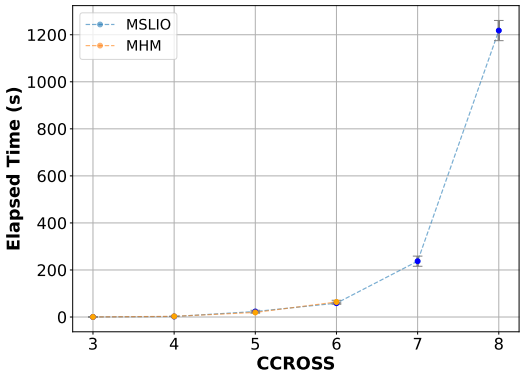
A. Validation of MSLIO

Figures 1 and 2 present the output phase duration in the actual simulation and in MSLIO for different *ccross* values, considering two numbers of sub-elements: 64 and 128. **We can see the output phase performance is very similar between the two, indicating the MSLIO I/O kernel successfully mimics the simulation**.

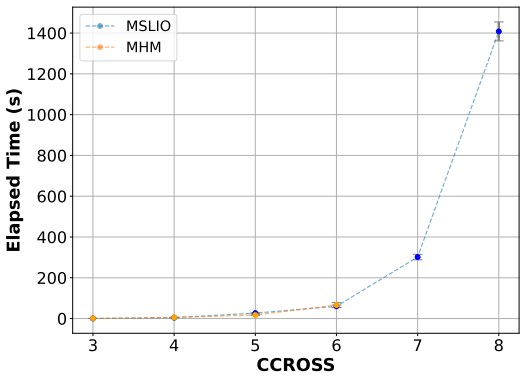
We can also see that I/O time increases exponentially with *ccross*, since so does the number of files and the amount of data, because the file size is determined by the number of sub-elements only. Due to the size of the simulation and its total execution time, we did not run it for *ccross* values of 7 and 8. That further illustrates the usefulness of MSLIO, because one can explore larger I/O loads. The number of files and amount of data handled by these experiments was detailed in Table I.

Figure 3 further characterizes the I/O behavior of both MSLIO and the actual simulation by breaking down the cumulative time spent on different calls during the same experiment. This cumulative time is the sum of the time spent on each call (it is important to notice calls were made in parallel by multiple threads so this is not real elapsed time, which was presented in Figure 1). We can see that both the I/O kernel and the real simulation spend a considerable portion of their time opening (creating) the files, especially with the smallest number of sub-elements, because in that case the amount of data being written is smaller (so the time spent on open calls has a higher importance).

The plots in Figure 3 show that in general open calls represent a higher portion of the time spent by the MSLIO than by the real simulation, which is mostly compensated by write calls representing a smaller portion (because, as seen in



(a) 64 sub-elements

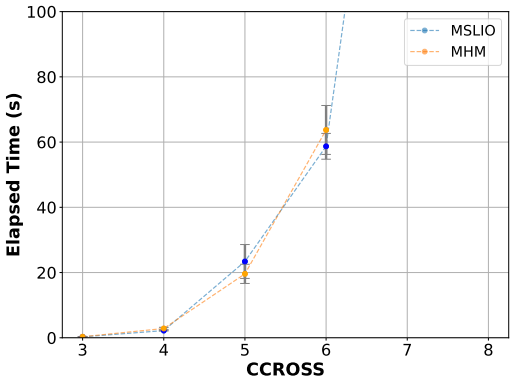


(b) 128 sub-elements

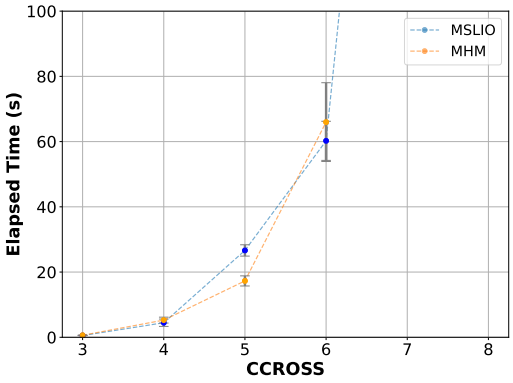
Fig. 1: Time spent writing the solution files by the real simulation and by the MSLIO I/O kernel for different values of *ccross* and numbers of sub-elements. These results were obtained on PlaFRIM. The y-axis is different in each plot, and lines are used to visualize tendencies.

Figure 1, the real elapsed time is very similar for the simulation and for the MSLIO). We believe this happens because the I/O phase of the simulation starts after a compute phase without any barrier, i.e. load imbalance between the processes and threads causes open calls to not be all simultaneous. On the other hand, because the I/O kernel does not have a compute phase, its threads are more likely to call open at the same time. That increases the metadata load and causes these calls to take longer. However, the same phenomenon can be beneficial to write operations because of caching effects.

Still, these differences are not very large, and, as seen in Figure 2, the time spent on the output routine is very similar for both codes. We have also verified through tracing that MSLIO generates the exact same number of open, write, and close calls than the simulation, and with the same size (in the case of writes). Therefore, we conclude the I/O kernel represents well the real multiscale simulation. Finally, the large portion of I/O time spent on open calls indicates decreasing

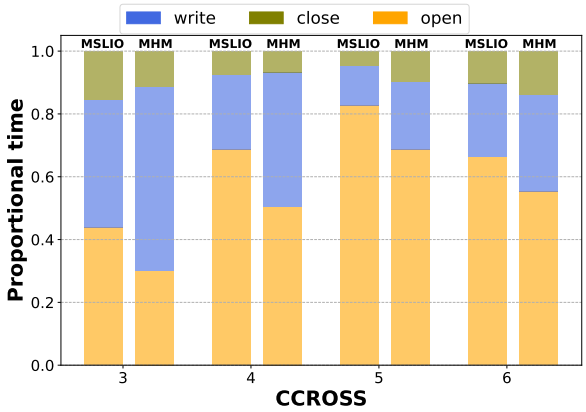


(a) 64 sub-elements

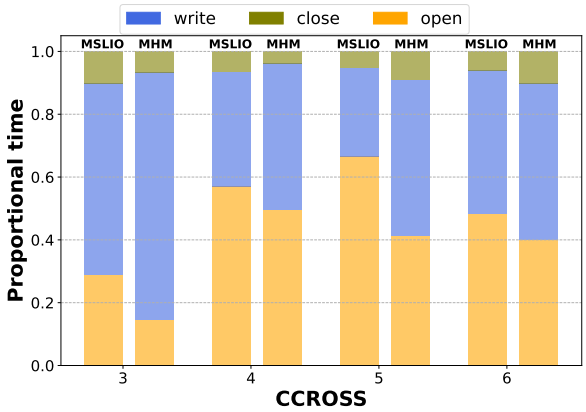


(b) 128 sub-elements

Fig. 2: The same results as shown in Figure 1, but zooming in for a better comparison between MSLIO and the real simulation, denoted by "MHM".



(a) 64 sub-elements



(b) 128 sub-elements

Fig. 3: Percentage of time spent in each type of I/O call by the I/O kernel and by the real simulation (denoted by "MHM"). These are the same executions shown in Figure 1, but cumulative time was used instead of elapsed time, i.e. the sum of time spent on different calls by parallel threads.

the number of files (as discussed in Section IV-A), and thus reducing the metadata load, could improve performance.

B. Buffering of small requests

To evaluate the buffering optimization strategy, we conducted experiments considering three buffers sizes: 128 KB, 512 KB, and 2 MB. These values were chosen around the stripe size of the platforms (512 KB in PlaFRIM and 1 MB in SDumont). Figures 4 and 5 present the results for different combinations of *ccross* and sub-elements on the two platforms.

As can be seen, compared with the case without buffers (original), the buffering strategy did not improve performance and actually increased the I/O time in most of the cases. These results suggest that the buffers are more useful when the amount of files and data increase, and depending on the machine. The only case where buffering improved performance — $\approx 33\%$ using 128 KB buffers, *ccross* of 9 and 32 sub-elements — was on SDumont, with the same experiment not showing the same result in PlaFRIM.

A difference in results between the two platforms could be explained by: i) the nodes in PlaFRIM having significantly more RAM memory than the ones from SDumont, which allows for more client-side caching by the file system (making application-level caching redundant); ii) the number of nodes/processes/threads being smaller in the experiments conducted on SDumont, meaning each node/process/thread accesses more data, which makes caching more difficult; and iii) specific behaviors of the used parallel file system (BeeGFS on PlaFRIM and Lustre on SDumont).

Nonetheless, although we plan on further exploring this optimization option in the future, for now we can conclude that the strategy does not seem to be promising for our case. **These results illustrate the importance of the I/O kernel for developing optimizations: it avoids spending time changing the actual library with ineffective optimizations.**

C. Single solution file

As discussed in Section IV-A, we have implemented an option in MSLIO where instead of creating one file per local problem, all data is sent to a single shared file using MPI-IO independent write calls. Figure 6 presents the results obtained

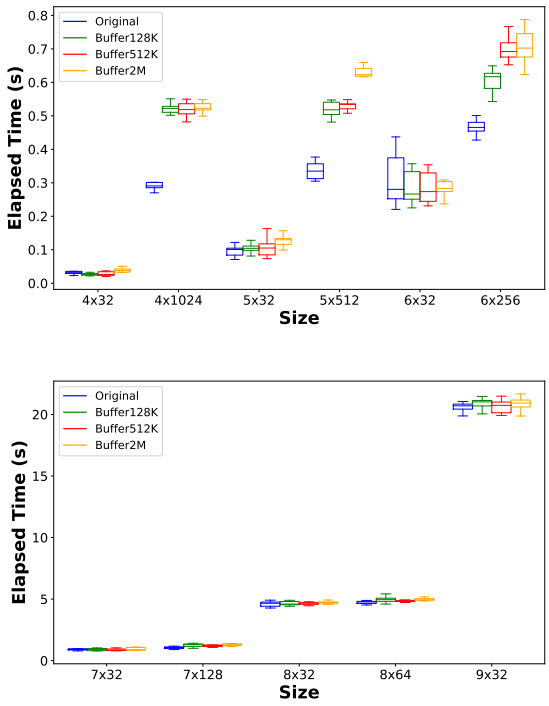


Fig. 4: Time spent by MSLIO to write the solution files for different combinations of *ccross* × sub-elements on PlaFRIM. Four versions of MSLIO, three using buffering, are compared. Two plots are presented to allow for comparisons, because of disparities in the values. The y axis is different in each plot.

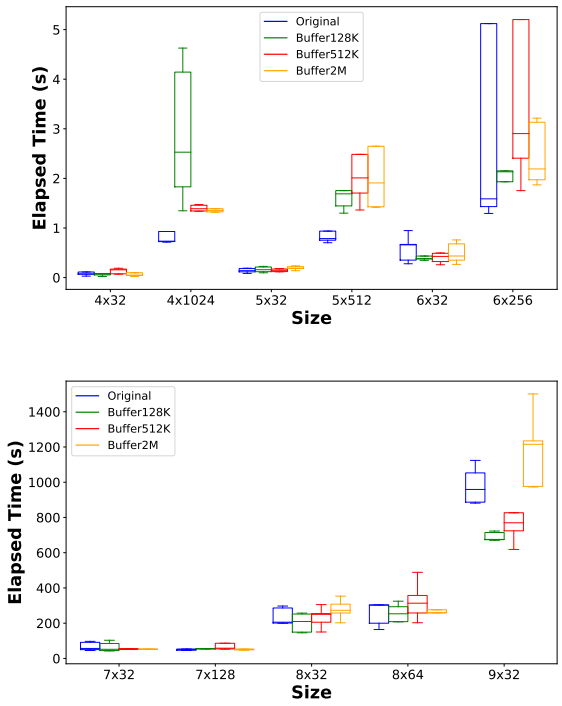


Fig. 5: Time spent by MSLIO to write the solution files for different combinations of *ccross* × sub-elements on SDumont. Four versions of MSLIO, three using buffering, are compared. Two plots are presented to allow for comparisons, because of disparities in the values. The y axis is different in each plot.

with this version. In PlaFRIM, the single-file version severely decreased I/O performance in all cases. On the other hand, in SDumont, it decreased the I/O time in up to $\approx 90\%$ for the largest tested cases. As the number of local problems (and therefore of files being created by the original version) increases, the benefits of a single-file version appear. Another promising observation from Figure 6b is that the time of the single-file version does not seem to increase as fast as the original version does as we increase the parameters.

An explanation for this disparity is the fact that Santos Dumont uses Lustre, which is known to have low metadata performance. Moreover, file-per-process access patterns were reported to perform better than single-file ones in BeeGFS [3].

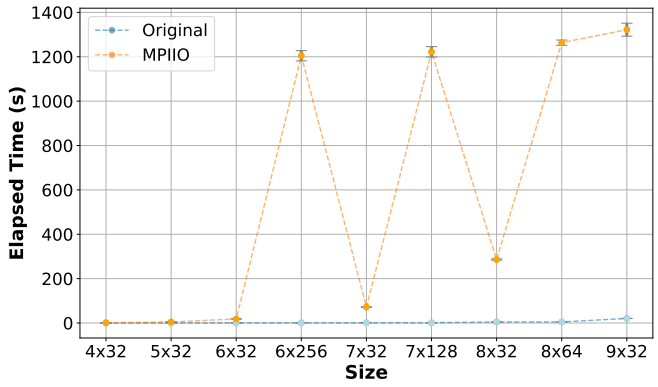
VII. Related Work

The impact of applications' access pattern on performance has been widely studied and reported: the I/O performance observed when accessing a parallel file system depends strongly on the way this access is done [1]–[6]. Many research efforts focus on improving the I/O infrastructure (the parallel file system or an I/O library, for instance) in a way that transparently benefits all applications running on a system [7]–[12]. Still, reaching peak I/O performance often still depends on application tuning [21], which is a more labor-intensive task because of the huge number of different applications that exist. That effort is justified for applications that are considered important because they are executed for long periods of time, or very often, hence its performance improvements result in better utilization of HPC resources [22].

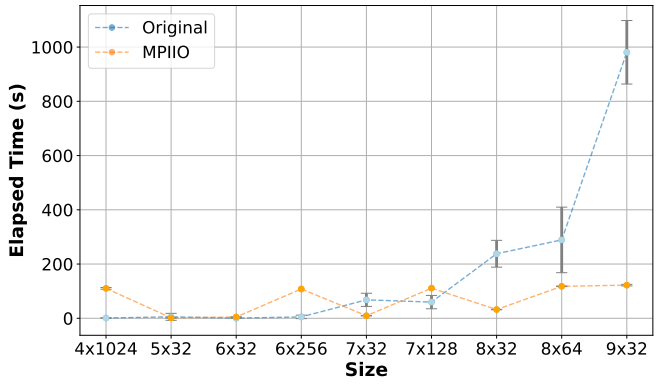
Qiao et al. [23] dynamically adapted the amount of data their applications — data analytics — use to avoid accessing the PFS when it is predicted to be under a heavy load from other jobs. Such application optimizations are of course very specific to the studied cases and not necessarily easily generalized to other applications, despite the general ideas still being helpful.

Other papers describe similar efforts into buffering small requests. Rettenberg and Baden [24] worked with seismic simulations modeled using large unstructured meshes, and Yu et al. [25] with cosmology applications that use adaptive refinement trees.

Boito et al. [26] studied and improved the I/O performance of the Ondes3D seismic wave simulation. They modified the application to buffer small requests, and reached improvements of up to 25% with buffers of 64 to 256 KB in a situation where ≈ 95 MB are written. As discussed in Section VI-B, we did not observe significant performance improvement by application-level buffering. As newer systems tend to have large amounts of RAM memory, in many cases system-provided caching may decrease the importance of request size on write performance.



(a) PlaFRIM



(b) Santos Dumont

Fig. 6: Time spent by MSLIO to write the solutions for different combinations of *ccross* × sub-elements in PlaFRIM and SDumont. The single-file version (here called "MPIIO") is compared to the original one. The y axis is different in each plot and the lines are used only to show tendencies.

A. Benchmark tools and I/O kernels

Benchmarks are valuable tools for experimental research. For parallel I/O, IOR[5] is arguably the most popular one: together with mdtest, it is used to rank I/O systems in the IO-500 list[6]. Its main advantage is the possibility of configuring the access pattern that is used to access the PFS (number of files, request size, etc.). Nonetheless, in many cases, we are interested in creating a "more realistic" scenario, with distinct I/O behaviors that really reflect what is seen among real applications. Compared to I/O benchmarks, real applications are usually less tailored for peak I/O performance [2].

In order to build such a realistic scenario, one needs to run real applications. However, i) they may be labor-intensive to compile and run because of many dependencies, ii) we may need to know what parameters reflect a realistic usage of that application, which is not always trivial, and iii) during their execution they use time and compute resources (e.g. GPUs) that we do not care about if we are trying to optimize I/O performance. For those reasons, I/O kernels are sometimes created and made available to the community. Well known examples are HACC-IO[7] and BT-IO[8]. Our MSLIO extends this list by adding a multiscale simulation I/O kernel.

VIII. Conclusion and Future Work

In this paper, we have presented MSLIO, an I/O kernel that mimics the I/O behavior of multiscale simulations. It has been developed using as a base the MHM library, used for these simulations. Such a kernel is a useful resource for i) researchers that want to generate realistic workloads to evaluate I/O systems, as running only the output routines is faster and easier than running a real application; and ii) for studying and trying to improve the performance of these simulations, as optimization ideas can be more easily and quickly prototyped and tested. MSLIO is publicly available at https://gitlab.inria.fr/lpeyrond/mslio.

The usefulness of MSLIO was illustrated by using it to test two optimization ideas: buffering of small requests and using a single shared file for solution data. We conducted extensive experiments to evaluate these optimizations in two systems: PlaFRIM (with I/O being performed to BeeGFS) and Santos Dumont (using Lustre). The buffering optimization was motivated by the knowledge that small requests tend to reach lower performance than larger requests, but it was not successful in our case. We believe this happened because both systems have nodes with a large amount of RAM memory, so system-provided client-side caching seems to be enough to hide the "bad" access pattern. Moreover, the fact that each file is very small limits what is achievable by buffering.

The other tested optimization — creating a single shared file instead of one per local problem — significantly *decreased* I/O performance in PlaFRIM but reached performance improvements of up to $\approx 90\%$ in SDumont. On the one hand, using multiple files increases the metadata load, but on the other hand using a shared file may cause problems because of locking mechanism, since data sizes are not aligned to the stripe size.

Both cases illustrate one of the advantages of having an I/O kernel: these optimizations were implemented and tested considerably easier than it would have been if the actual library was modified, and that also allowed for more possible parameter combinations to be tested. As future work, we plan on further exploring the single-file optimization and implementing it in the MHM library as an option. It will be important to provide users a guide of when to use or not to use this version, so they can reach the best possible performance in their system. Furthermore, we plan on using MSLIO to try a variation of this optimization where accesses are aligned to the stripe size, which could improve results on BeeGFS.

[5]https://github.com/hpc/ior
[6]https://io500.org/
[7]https://www.vi4io.org/tools/benchmarks/hacc_io
[8]https://www.vi4io.org/tools/benchmarks/npb

AUTHOR STATEMENT

Louis Peyrondet was in charge of studying the MHM library, software design and implementation. He had help from Antonio Tadeu Gomes, who is the main architect of the MHM library. Louis Peyrondet and Francieli Boito were responsible for experimentation. Luan Teylo was responsible for data curation and visualization. All authors participated in the conceptualization of the work, methodology, and analysis of the results. Francieli Boito, Antonio Tadeu Gomes, and Luan Teylo wrote the paper. All authors read and approved the manuscript.

ACKNOWLEDGMENT

The authors would like to thank Alexis Bandet for his help. Some of the experiments were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr). The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper (see http://sdumont.lncc.br). This work was supported in part by the French National Research Agency (ANR) in the frame of DASH (ANR-17-CE25-0004), by the Project Région Nouvelle Aquitaine 2018-1R50119 "HPC scalable ecosystem" and by the "Adaptive multitier intelligent data manager for Exascale (ADMIRE)" project, funded by the European Union's Horizon 2020 JTI-EuroHPC Research and Innovation Programme (grant 956748). This work was conducted in the context of the HPCProSol joint team between Inria and the LNCC https://team.inria.fr/hpcprosol/.

REFERENCES

[1] L. Pottier, R. F. da Silva, H. Casanova, and E. Deelman, "Modeling the performance of scientific workflow executions on hpc platforms with burst buffers," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, 2020, pp. 92–103.

[2] T. Wang, S. Byna, G. K. Lockwood, S. Snyder, P. Carns, S. Kim, and N. J. Wright, "A zoom-in analysis of i/o logs to detect root causes of i/o performance bottlenecks," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 102–111.

[3] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/o characterization and performance evaluation of beegfs for deep learning," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337902

[4] F. Boito, G. Pallez, and L. Teylo, "The role of storage target allocation in applications' I/O performance with BeeGFS," in *CLUSTER 2022 - IEEE International Conference on Cluster Computing*, Heidelberg, Germany, Sep. 2022. [Online]. Available: https://hal.inria.fr/hal-03753813

[5] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai, W. Liu, and W. Xue, "End-to-end I/O monitoring on a leading supercomputer," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/yang

[6] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applications," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 185–193.

[7] M. Agarwal, D. Singhvi, P. Malakar, and S. Byna, "Active learning-based automatic tuning and prediction of parallel i/o performance," in *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, 2019, pp. 20–29.

[8] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari, "Toward managing hpc burst buffers effectively: Draining strategy to regulate bursty i/o behavior," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2017, pp. 87–98.

[9] F. Tessier, V. Vishwanath, and E. Jeannot, "Tapioca: An i/o library for optimized topology-aware data aggregation on large-scale supercomputers," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 70–80.

[10] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, "Automatic, Application-Aware I/O forwarding resource allocation," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 265–279. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/ji

[11] J. L. Bez, A. Miranda, R. Nou, F. Z. Boito, T. Cortes, and P. Navaux, "Arbitration policies for on-demand user-level i/o forwarding on hpc platforms," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 577–586.

[12] H. Sung, J. Bang, C. Kim, H.-S. Kim, A. Sim, G. K. Lockwood, and H. Eom, "Bbos: Efficient hpc storage management via burst buffer over-subscription," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 142–151.

[13] Y. Efendiev and T. Y. Hou, *Multiscale Finite Element Methods*. Springer, 2009.

[14] C. Harder, D. Paredes, and F. Valentin, "A family of Multiscale Hybrid-Mixed finite element methods for the Darcy equation with rough coefficients," *Journal of Computational Physics*, vol. 245, pp. 107–130, 2013.

[15] ——, "On a multiscale hybrid-mixed method for advective-reactive dominated problems with heterogeneous coefficients," *Multiscale Modeling & Simulation*, vol. 13, no. 2, pp. 491–518, 2015.

[16] R. Araya, C. Harder, A. H. Poza, and F. Valentin, "Multiscale hybrid-mixed method for the Stokes and Brinkman equations – The method," *Computer Methods in Applied Mechanics and Engineering*, vol. 324, pp. 29–53, 2017.

[17] S. Lanteri, D. Paredes, C. Scheid, and F. Valentin, "The Multiscale Hybrid-Mixed method for the Maxwell Equations in Heterogeneous Media," *Multiscale Modeling & Simulation*, vol. 16, no. 4, pp. 1648–1683, 2018.

[18] T. Chaumont-Frelet and F. Valentin, "A multiscale hybrid-mixed method for the helmholtz equation in heterogeneous domains," *SIAM Journal on Numerical Analysis*, vol. 58, no. 2, pp. 1029–1067, 2020.

[19] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin, "A checkpoint of research on parallel i/o for high-performance computing," *ACM Comput. Surv.*, vol. 51, no. 2, mar 2018. [Online]. Available: https://doi.org/10.1145/3152891

[20] J. Bez, A. Carneiro, P. Pavan, V. Girelli, F. Boito, B. Fagundes, C. Osthoff, P. Leite da Silva Dias, J.-F. Méhaut, and P. O. Navaux, "I/O Performance of the Santos Dumont Supercomputer," *International Journal of High Performance Computing Applications*, pp. 1–17, 2019. [Online]. Available: https://hal.inria.fr/hal-02270908

[21] L. Wan, A. Huebl, J. Gu, F. Poeschel, A. Gainaru, R. Wang, J. Chen, X. Liang, D. Ganyushin, T. Munson, I. Foster, J. Vay, N. Podhorszki, K. Wu, and S. Klasky, "Improving i/o performance for exascale applications through online data layout reorganization," *IEEE Transactions on Parallel amp; Distributed Systems*, vol. 33, no. 04, pp. 878–890, 2022.

[22] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476181

[23] Z. Qiao, Q. Liu, N. Podhorszki, S. Klasky, and J. Chen, "Taming i/o variation on qos-less hpc storage: What can applications do?" in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[24] S. Rettenberger and M. Bader, "Optimizing i/o for petascale seismic simulations on unstructured meshes," in *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, ser. CLUSTER '15. USA: IEEE Computer Society, 2015, p. 314–317. [Online]. Available: https://doi.org/10.1109/CLUSTER.2015.51

[25] Y. Yu, D. H. Rudd, Z. Lan, N. Y. Gnedin, A. Kravtsov, and J. Wu, "Improving parallel io performance of cell-based amr cosmology applications," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 933–944.

[26] F. Z. Boito, J. L. Bez, F. Dupros, M. A. R. Dantas, P. O. A. Navaux, and H. Aochi, "High performance i/o for seismic wave propagation simulations," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2017, pp. 31–38.

APPENDIX

This appendix complements the paper by detailing all results (arithmetic mean and standard deviation), which were only presented through plots. The results used to validate MSLIO, presented in Figures 1 and 2, are detailed in Table II, and Table III shows the results presented in Figure 3. Tables IV and V present the results shown in Figures 4 and 5, respectively, obtained with the version of MSLIO that does buffering of small requests on the two used platforms (optimization discussed in Section IV-B). Finally, the results presented in Figure 6 — with the version of the I/O kernel that writes all data to a single file — are detailed in Tables VI and VII.

TABLE II: Arithmetic mean and standard deviation of time spent to write the solution files for different combinations of *ccross* and sub-elements on PlaFRIM, comparing the real library to the MSLIO I/O kernel. Results presented in Figure 1 (and 2).

Sub-elements	ccross	MHM Library		MSLIO	
		Mean	Std. dev.	Mean	Std. dev.
64	3	0.322	0.039	0.279	0.013
	4	2.844	0.366	2.207	0.129
	5	19.597	2.94	23.394	5.194
	6	63.717	7.502	58.702	3.955
	7	—	—	237.322	21.591
	8	—	—	1217.808	42.974
128	3	0.621	0.039	0.532	0.045
	4	5.267	0.903	4.462	1.126
	5	17.284	1.551	26.63	1.737
	6	65.979	12.074	60.231	5.995
	7	—	—	301.608	13.404
	8	—	—	1408.053	46.372

TABLE III: Arithmetic mean and standard deviation of percentage of cumulative time spent by the threads on different I/O operations, for different combinations of *ccross* and sub-elements on PlaFRIM, comparing the real library to the MSLIO I/O kernel. Results presented in Figure 3.

64 sub-elements						
ccross:			3	4	5	6
Library	open	Mean	0.295	0.5	0.688	0.55
		Std. dev.	0.053	0.032	0.018	0.106
	write	Mean	0.59	0.434	0.216	0.311
		Std. dev.	0.039	0.046	0.007	0.071
	close	Mean	0.115	0.066	0.096	0.14
		Std. dev.	0.016	0.023	0.014	0.036
MSLIO	open	Mean	0.438	0.685	0.825	0.658
		Std. dev.	0.029	0.013	0.01	0.107
	write	Mean	0.406	0.239	0.129	0.238
		Std. dev.	0.012	0.012	0.012	0.063
	close	Mean	0.156	0.076	0.046	0.104
		Std. dev.	0.028	0.004	0.004	0.044
128 sub-elements						
ccross:			3	4	5	6
Library	open	Mean	0.143	0.483	0.411	0.392
		Std. dev.	0.052	0.087	0.082	0.055
	write	Mean	0.789	0.479	0.498	0.506
		Std. dev.	0.046	0.086	0.045	0.055
	close	Mean	0.068	0.039	0.091	0.102
		Std. dev.	0.008	0.005	0.046	0.009
MSLIO	open	Mean	0.285	0.55	0.663	0.479
		Std. dev.	0.045	0.093	0.059	0.038
	write	Mean	0.611	0.38	0.282	0.46
		Std. dev.	0.035	0.069	0.034	0.034
	close	Mean	0.104	0.069	0.055	0.061
		Std. dev.	0.013	0.027	0.025	0.006

TABLE IV: Arithmetic mean and standard deviation of time spent by MSLIO to write the solution files for different combinations of *ccross* × sub-elements **on PlaFRIM**, comparing the original implementation to the optimization that tries to buffer small write requests. Results presented in Figure 4.

		original	Buffering		
			128 KB	512 KB	2 MB
4x32	Mean	0.035	0.03	0.028	0.041
	Std. dev.	0.014	0.016	0.007	0.011
4x1024	Mean	0.298	0.523	0.518	0.527
	Std. dev.	0.024	0.017	0.022	0.016
5x32	Mean	0.095	0.105	0.106	0.127
	Std. dev.	0.016	0.013	0.027	0.017
5x512	Mean	0.337	0.555	0.53	0.631
	Std. dev.	0.027	0.096	0.024	0.015
6x32	Mean	0.308	0.287	0.286	0.297
	Std. dev.	0.079	0.049	0.046	0.056
6x256	Mean	0.462	0.605	0.698	0.707
	Std. dev.	0.045	0.034	0.036	0.05
7x32	Mean	0.925	1.024	0.902	0.942
	Std. dev.	0.1	0.428	0.115	0.12
7x128	Mean	1.046	1.243	1.2	1.294
	Std. dev.	0.092	0.148	0.074	0.081
8x32	Mean	4.609	4.652	4.742	4.898
	Std. dev.	0.217	0.162	0.372	0.61
8x64	Mean	4.734	4.912	4.826	4.939
	Std. dev.	0.128	0.309	0.108	0.157
9x32	Mean	20.589	20.912	20.648	20.838
	Std. dev.	0.366	0.421	0.547	0.558

TABLE V: Arithmetic mean and standard deviation of time spent by MSLIO to write the solution files for different combinations of *ccross* × sub-elements **on Santos Dumont**, comparing the original implementation to the optimization that tries to buffer small write requests. Results presented in Figure 5.

		original	Buffering		
			128 KB	512 KB	2 MB
4x32	**Mean**	0.095	0.112	0.132	0.135
	Std. dev.	0.061	0.126	0.056	0.185
4x1024	**Mean**	1.002	8.713	1.474	3.159
	Std. dev.	0.513	15.174	0.224	4.055
5x32	**Mean**	5.378	6.905	0.148	0.194
	Std. dev.	12.846	16.546	0.03	0.038
5x512	**Mean**	1.771	1.851	7.578	2.919
	Std. dev.	2.185	0.705	12.725	2.442
6x32	**Mean**	0.582	0.677	6.928	6.239
	Std. dev.	0.27	0.655	16.032	14.196
6x256	**Mean**	5.094	3.699	9.538	2.476
	Std. dev.	6.322	3.927	14.529	0.648
7x32	**Mean**	68.006	65.38	59.828	50.462
	Std. dev.	24.259	27.042	21.38	33.048
7x128	**Mean**	59.658	59.001	79.155	59.818
	Std. dev.	24.618	12.461	39.136	22.323
8x32	**Mean**	237.982	202.698	233.14	278.954
	Std. dev.	49.398	53.748	58.393	56.497
8x64	**Mean**	288.948	257.589	323.99	262.905
	Std. dev.	120.844	51.737	108.928	42.983
9x32	**Mean**	980.802	691.518	804.709	1180.068
	Std. dev.	117.142	25.18	173.985	218.742

TABLE VI: Arithmetic mean and standard deviation of time spent by MSLIO to write the solution files for different combinations of *ccross* × sub-elements **on PlaFRIM**, comparing the original implementation to the optimization that writes all data to a single shared file using MPI-IO. Results presented in Figure 6a.

		original	single-file
5x32	**Mean**	0.095	4.654
	Std. dev.	0.016	0.17
6x32	**Mean**	0.308	18.083
	Std. dev.	0.079	0.275
6x256	**Mean**	0.462	1204.579
	Std. dev.	0.045	23.074
7x32	**Mean**	0.925	71.8
	Std. dev.	0.1	1.053
7x128	**Mean**	1.046	1222.112
	Std. dev.	0.092	23.488
8x32	**Mean**	4.609	286.139
	Std. dev.	0.217	2.912
8x64	**Mean**	4.734	1263.461
	Std. dev.	0.128	11.836
9x32	**Mean**	20.589	1322.045
	Std. dev.	0.366	29.277

TABLE VII: Arithmetic mean and standard deviation of time spent by MSLIO to write the solution files for different combinations of *ccross* × sub-elements **on Santos Dumont**, comparing the original implementation to the optimization that writes all data to a single shared file using MPI-IO. Results presented in Figure 6b.

		original	single-file
5x32	**Mean**	5.378	1.636
	Std. dev.	12.846	0.553
6x32	**Mean**	0.582	4.706
	Std. dev.	0.27	0.514
6x256	**Mean**	5.094	108.034
	Std. dev.	6.322	—
7x32	**Mean**	68.006	9.313
	Std. dev.	24.259	0.322
7x128	**Mean**	59.658	111.05
	Std. dev.	24.618	—
8x32	**Mean**	237.982	32.241
	Std. dev.	49.398	0.748
8x64	**Mean**	288.948	117.852
	Std. dev.	120.844	1.08
9x32	**Mean**	980.802	122.204
	Std. dev.	117.142	2.808