# Collective I/O Performance on the Santos Dumont Supercomputer

André Ramos Carneiro, Jean Luca Bez, Francieli Zanon Boito, Bruno Alves Fagundes, Carla Osthoff, Philippe Navaux

# Collective I/O Performance on the Santos Dumont Supercomputer

André Ramos Carneiro[1], Jean Luca Bez[2], Francieli Zanon Boito[3],
Bruno Alves Fagundes[1], Carla Osthoff[1], Philippe O. A. Navaux[2]
[1]National Laboratory for Scientific Computing (LNCC) — Petrópolis, Brazil
{andrerc, brunoaf, osthoff}@lncc.br
[2]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS) — Porto Alegre, Brazil
{jlbez, navaux}@inf.ufrgs.br
[3]Grenoble Informatics Laboratory (LIG), INRIA — Grenoble, France
francieli.zanon-boito@inria.fr

***Abstract*—The historical gap between processing and data access speeds causes many applications to spend a large portion of their execution on I/O operations. From the point of view of a large-scale, expensive, supercomputer, it is important to ensure applications achieve the best I/O performance to promote an efficient usage of the machine. In this paper, we evaluate the I/O infrastructure of the Santos Dumont supercomputer, the largest one from Latin America. More specifically, we investigate the performance of collective I/O operations. By conducting an analysis of a scientific application that uses the machine, we identify large performance differences between the available MPI implementations. We then further study the observed phenomenon using the BT-IO and IOR benchmarks, in addition to a custom microbenchmark. We conclude that the customized MPI implementation by Bull (used by more than 20% of the jobs) presents the worst performance for small collective write operations. Our results are being used to help the Santos Dumont users to achieve the best performance for their applications. Additionally, by investigating the observed phenomenon, we provide information to help improve future MPI-IO collective write implementations.**

## I. Introduction

Applications that execute in high-performance computing (HPC) infrastructures — large-scale clusters or supercomputers — often need to input or output data. This is usually accomplished by performing I/O operations to a parallel file system (PFS), such as Lustre [1] or GPFS [2].

The parallel file system is deployed over a set of dedicated machines that act as metadata and/or data servers. Files are separated into fixed-size chunks and distributed across data servers through an operation called "data striping". I/O operations represent a bottleneck for an increasing number of applications due to the speed difference between computation and data access, as the latter depends on slower components like disks and the network.

Performance observed when accessing a PFS is highly dependent on the way this access is performed, i.e. on the application's access pattern [3], [4]. For instance, higher performance is achieved when accessing sequentially positioned portions of a file in large requests as opposed to accessing small sparse portions. For this reason, many optimization techniques ([5], [6], [7]) have been proposed aiming at adapting the access pattern to achieve the best performance from the file system.

The most popular of such techniques is the use of *collective I/O operations*, proposed in the 90s to the ROMIO implementation of the MPI-IO [8] API. The traditional implementation of this technique is called "two-phase I/O" [9]. A set of processes that intend to read/write from/to the same file make a global call, and a subset of them — the "aggregators" — are chosen to perform the operation on behalf of the others. In the first phase processes send their data to aggregators, and in the second one aggregators write data to the PFS (the description is symmetrical for read operations). The advantage of this method is that fewer requests are generated to the PFS, and they are less sparse and larger in size, hence performance is usually increased by using collective operations.

In this paper, we evaluate the I/O infrastructure of the *Santos Dumont* supercomputer (also called "SDumont") regarding collective operations. SDumont is a Bull/Atos machine, located at the National Laboratory for Scientific Computing (LNCC) in Brazil. With a total of 18,144 cores, it is the largest supercomputer from Latin America, with an investment from the Brazilian government of approximately 60 million dollars [10].

Since I/O performance is a limiting factor for many scientific applications, the importance of such evaluation and optimization work is clear. Considering the high financial costs associated with buying and maintaining a supercomputer, its administrators must promote an efficient use of the machine.

We start by studying a real scientific application that is executed in the SDumont: the Ocean-Land-Atmosphere Model (OLAM) [11]. The I/O performance evaluation of this atmospheric simulation indicated the need to investigate the MPI collective I/O operations on the machine. Then we use existing and custom benchmarks to characterize and explain performance.

We compare different MPI implementations that are available in the SDumont and show large collective I/O performance differences between them. This work provided valuable information in the form of guidelines so SDumont users can achieve higher I/O performance. It can also be applied to other similar machines. Finally, by investigating the reported phenomenon, we provide information that can be used to improve future MPI-IO implementations. To the best of our knowledge, ours is the first work to document and investigate this behavior.

The remainder of this paper is organized as follows: the next section describes the Santos Dumont supercomputer. Section III presents the performance evaluation conducted with the OLAM application and discuss its results. Section IV further investigates the behavior observed in Section III using benchmarks to confirm hypotheses and explain results. Related work is discussed in Section V, and Section VI presents conclusions and discusses future work.

## II. The Santos Dumont Supercomputer

The SDumont supercomputer, located at the LNCC in Brazil, has a total of 18,144 CPU cores. Every compute node has two Intel Xeon E5-2695v2 Ivy Bridge 2.4GHz 12-core processors, 64GB DDR3 RAM memory, and one 128GB SSD. There are three types of nodes:

- 504 B710 (regular) compute nodes;
- 198 B715 nodes with two K40 GPUs each;
- 54 B715 nodes with two Xeon Phi KNC co-processors each.

Compute, login and storage nodes are connected through Infiniband FDR on a fat-tree full-nonblocking topology. The Lustre parallel file system version 2.1 is deployed through the Xyratex/Seagate ClusterStor v1.5.0, with one MDS (Metadata Server) and 10 OSS (Object Storage Service), each with one OST (Object Storage Target), for a total storage capacity of 1.7 PB. Clients use the version 2.4.3, and mount the file system with the *flock* option.

All experiments discussed in this paper were conducted using the default stripe size of 1 MB and a stripe count of 10, i.e., files are distributed among all 10 Lustre data servers (OSTs).

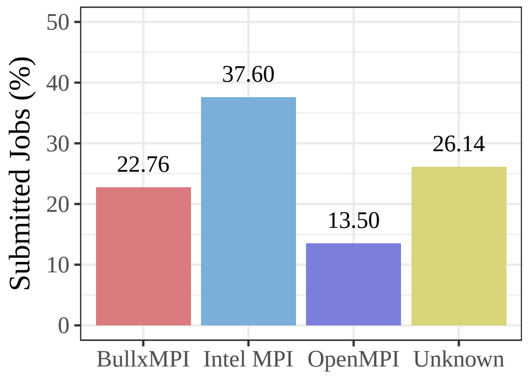


Fig. 1: MPI usage among jobs submitted to SDumont

SDumont users can choose between the three MPI implementations available: BullxMPI v1.2.8.4, based on OpenMPI, IntelMPI v5.1.3 build 20160120, based on MPICH, and OpenMPI versions 1.10, 2.0, and 2.1. Fig. 1 shows the percentage of jobs using each MPI implementation, as observed from June 15 to July 22, 2017. The last bar accounts for jobs that do not use the expected command to select an implementation.

## III. OLAM Performance Evaluation

In this section, we discuss the I/O performance evaluation of the Ocean-Land-Atmosphere Model (OLAM), one of the main scientific applications executed in the SDumont. We first describe the application in Section III-A, then the experimental methodology in Section III-B. Results are discussed in Section III-C for two OLAM configurations. It is important to notice these are not the last results from this paper, as the observed phenomenon is further investigated and discussed in Section IV.

### A. The Ocean-Land-Atmosphere Model

OLAM [11] is a global numerical simulation model that provides the advantages of regional models through the execution of a global model. This is achieved using a global grid, which can be refined for specific regions. Using this technique, regional phenomena can be properly simulated without the necessity of previous executions of global models with a coarser grid, since border conditions are already generated during the execution of OLAM.

The model essentially consists of a global triangular-cell grid mesh with local refinement capability, the full compressible non-hydrostatic Navier-Stokes equations, a finite volume formulation of conservation laws for mass, momentum, and potential temperature, and numerical operators that include time splitting for acoustic terms. The global domain greatly expands the range of atmospheric systems and scale interactions that can be represented in the model, which was the primary motivation for developing OLAM.

OLAM was developed in Fortran 90 and parallelized with MPI under the Single Program Multiple Data (SPMD) model. OLAM is an iterative model, where each timestep may result in the output of data as defined in its parameters. Its workflow is illustrated in Fig. 2.

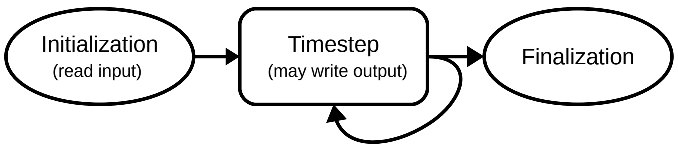


Fig. 2: OLAM's interactive organization

Each process completely reads the initialization files. Typical input files are global initial conditions at a certain date and time and global maps describing topography, soil type, ice-covered areas, Olson Global Ecosystem (OGE) vegetation dataset, depth of the soil interacting with the root zone, sea surface temperature and Normalized Difference Vegetation

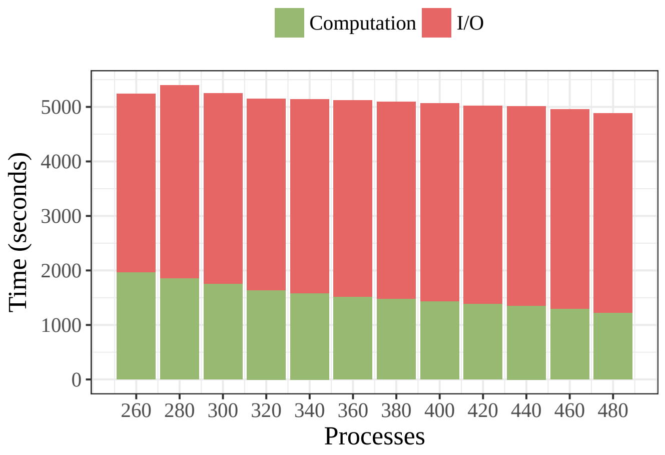


Fig. 3: Initial results with OLAM in the SDumont

Index (NDVI). Next, the processing and data output phases are executed alternately: during each processing phase, OLAM simulates a number of timesteps, evolving the atmospheric conditions on time-discrete units. After each timestep, processes exchange messages with their neighbors to keep the atmospheric state consistent.

After executing a number of timesteps, the variables representing the atmosphere (atmospheric pressure, temperature, wind speed, precipitation, etc) are written to a history file. During this phase, processes use the HDF5 [12] library to write to the shared file, and the library generates MPI collective I/O operations. These output history files can have from a few MB to many GB, depending on the grid definition and refinement.

Initial OLAM executions in SDumont indicated performance is compromised by I/O operations, as shown in Fig. 3. Bars stack time spent in I/O (in red) and computation (in green) for different numbers of processes. Up to 73% of the execution time was spent with I/O operations. Furthermore, despite computation time decreasing as the scale is increased, I/O impaired the application's scalability.

## B. *Experimental Methodology*

New experiments were conducted in SDumont with OLAM version 4.10 (r544), modeling the northwest region of the São Paulo state, in Brazil ($21°00'00.0''S$ $51°00'00.0''W$). The configuration for these simulations has six vertical levels. Two days are simulated, using timesteps of ten seconds and writing output every simulated hour. A total of 49 files are generated, with approximately 1.1 GB each.

OLAM was compiled with Intel Parallel Studio XE 2017 update 1 and HDF5 version 1.8.18. The Darshan tool [13] version 3.1.4 was used to profile executions.

The results presented in this section are the medians of five executions on 10 compute nodes (sdumont[5004–5013]) using the 24 cores per node, for a total of 240 cores. No additional hints were passed to HDF5 or MPI-IO, and all MPI implementations use the same parameters.

## C. *Results*

*1) OLAM with 7 grids:* The first set of experiments that we will discuss compare BullxMPI and OpenMPI 1.10, and motivate the rest of this work. OLAM was executed with seven grids (one global and six refined) of resolutions 200km (global), 100km, 50km, 25km, $12.5km$, 6.25km, and 3.125km. Results, as reported by Darshan, can be seen in Fig. 4. The first graph, in Fig. 4a show time spent in I/O (in red) and computation (in green). We can see a large decrease in execution time when using OpenMPI 1.10, due to a much shorter I/O time (computation time was similar to both implementations).

Since we could **not** conclude all these results follow a normal distribution, we have used the Wilcoxon-Mann-Whitney test [14] to compare them, and compare the medians instead of the means. The statistical test has indicated that computation time for the two MPI implementations is **not** significantly different, but I/O and total execution time are.

Fig. 4b separates time spent in I/O operations per API — POSIX and MPI-IO. It shows most of the I/O time is spent in MPI-IO operations (through HDF5), and the performance difference between the MPI implementations comes mostly from this part. In Fig. 4c the number of MPI-IO write calls is presented separated by type, and we can see most of the write operations are collective.

Table I shows the most usual sizes for I/O requests generated by OLAM. Requests are rather small, with most operations being for approximately 1300 bytes.

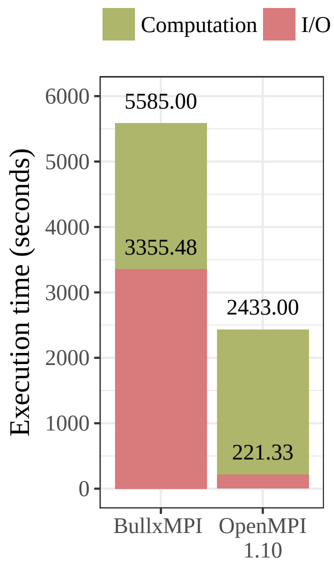
TABLE I: Size of I/O operations generated by OLAM, as reported by Darshan.

| **API** | **BullxMPI** | | **OpenMPI-1.10** | |
|---|---|---|---|---|
| | Size (bytes) | Count | Size (bytes) | Count |
| **POSIX** | 8192 | 8585280 | 8192 | 8585280 |
| | 8190 | 6785760 | 8190 | 6785760 |
| | 512 | 126186 | 512 | 126186 |
| | 1048576 | 46011 | 1048576 | 46011 |
| **MPI-IO** | 1308 | 94521 | 1308 | 110397 |
| | 1312 | 55713 | 1312 | 94521 |
| | 1316 | 44394 | 1316 | 63798 |
| | 59040 | 43904 | 58860 | 59584 |

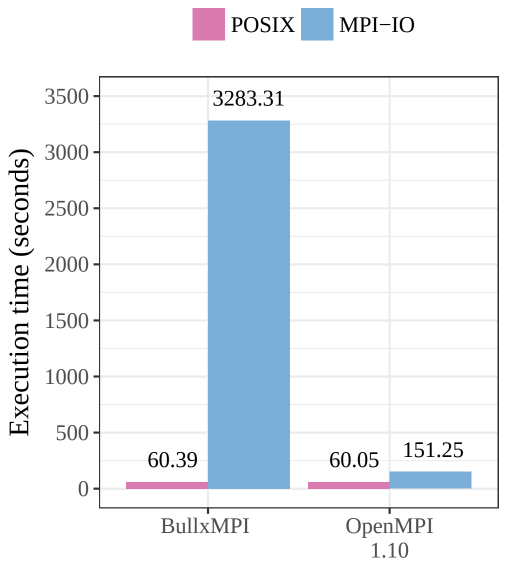
*2) OLAM with 4 grids:* The large performance difference between the MPI implementations was unexpected, and thus we have conducted more comprehensive experiments, including all MPI implementations that are available at the machine. To decrease the processing time in the supercomputer while keeping the same I/O behavior we executed an OLAM configuration with four grids of resolution 200km (global), 100km, 50km, and 25km.

Because of incompatibilities between Darshan and some MPI implementations, we have modified the OLAM source code to measure and report execution and I/O time internally.

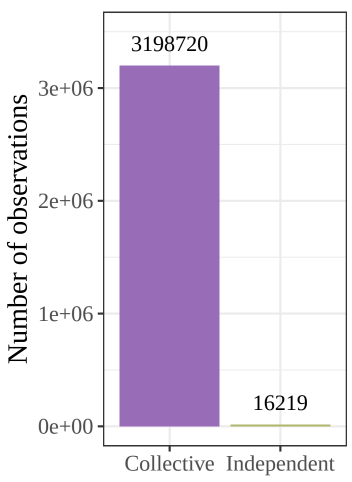
Results are presented in Fig. 5, and show one more time that BullxMPI is the alternative that causes OLAM to spend the most time on I/O operations, 87% of the execution time with four grids. The large difference to OpenMPI 1.10 per-



(a) Execution time



(b) I/O Time separated by API



(c) MPI-IO write operations

Fig. 4: Results for OLAM with 7 grids comparing BullxMPI and OpenMPI 1.10, reported by Darshan.

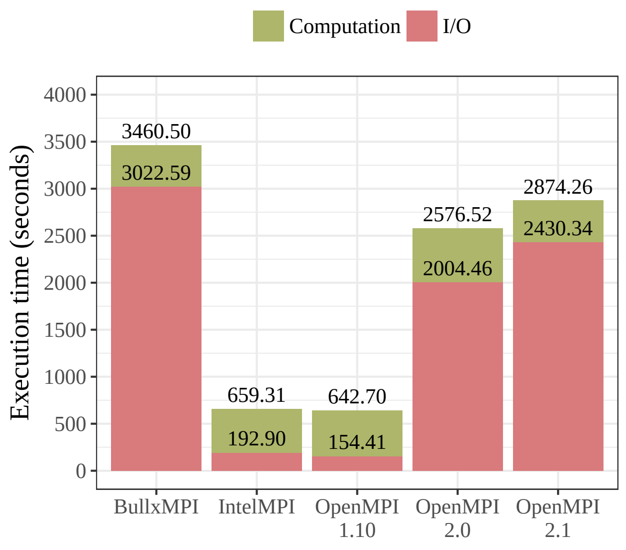


Fig. 5: Results for OLAM with 4 grids

formance was still present, IntelMPI presented similar results to OpenMPI 1.10, and OpenMPI 2.0 and 2.1 were better than BullxMPI, but worse than OpenMPI 1.10 and IntelMPI.

Similarly to the OLAM configuration with seven grids, we have not concluded all the sets of results follow a normal distribution, and thus used a non-parametric test to compare them. The Dunn test [15] could **not** conclude results for IntelMPI are significantly different from results with OpenMPI 1.10 (not for I/O or for total execution time). Similarly, results for BullxMPI are not statistically different from results for OpenMPI 2.1. Finally, results for OpenMPI 2.0 and 2.1 are not significantly different.

## IV. Collective I/O Performance

In the previous section, experiments with the OLAM application indicated large performance differences between MPI implementations. Because of the application's characteristics, reported by Darshan, they were believed to be due to MPI collective write operations.

### A. *Experiments with the BT-IO benchmark*

To confirm that it was not something specific to the application, we conducted experiments with the BT-IO benchmark from the NPB [16], the second most used benchmark in the parallel I/O research field, as pointed by [17]. We used the D class, which generates a file of approximately 132.6 GB and yields an execution time in order of minutes. This benchmark generates MPI-IO collective write calls. These experiments were executed over eight nodes (sdumont[5004–5011]), using 18 cores per node, for a total of 144 cores.

Results are presented in Fig. 6, and are median values from 5 repetitions. The Dunn test was used to compare all sets of results, and indicated that results for BullxMPI are significantly different from IntelMPI and OpenMPI 1.10, and results for OpenMPI 1.10 are different from OpenMPI 2.0 and OpenMPI 2.1. Nonetheless, we can see this difference is quite small if compared to what was observed before. Studying the information provided by Darshan, we observed the size of requests generated by the benchmark was approximately 18 MB, much larger than requests generated by OLAM.

### B. *Experiments with the IOR benchmarking tool*

The fact we observed large performance difference between different MPI implementations with OLAM, but not with the BT-IO benchmark, indicates this difference does not happen for large collective write requests. To confirm it happens for

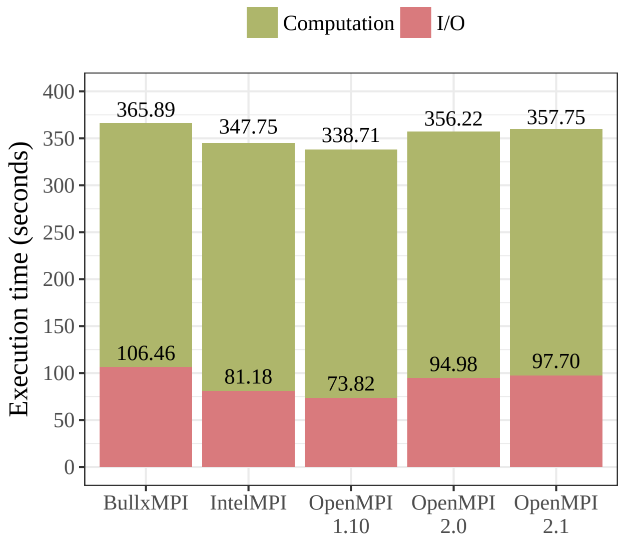


Fig. 6: Results for BT-IO class D

small requests (that it was not something specific related to OLAM), we conducted more experiments using the IOR benchmarking tool [18].

IOR experiments were executed with all the MPI implementations available, including the OMPIO implementation of MPI-IO, available for OpenMPI versions. We only used ROMIO for previous results because of incompatibilities between OMPIO and OLAM. IOR was configured to perform collective read and write tests, generating a file of 1.5 GB using MPI-IO and HDF5 (the latter also used by OLAM, as discussed in Section III). Tested request sizes were 1024, 1312, 58864, and 65536 bytes. Request sizes of 1312 and 58864 bytes are the most common access sizes used by OLAM, as seen in Table I. They generate misaligned access on the Lustre parallel file system that is configured with a stripe size of 1 MB. To verify the impact of accesses that are not aligned with the stripe size, we also included requests of 1024 and 65536 bytes, which do not lead to misaligned accesses. Table II details the IOR parameters used for these experiments.

TABLE II: Parameters of IOR experiments

| Transfer Size | Block Size | Segment Size | Segment Count |
|---:|---:|---:|---:|
| 1024 | 1024 | 245760 | 6554 |
| 1312 | 1312 | 314880 | 5116 |
| 58864 | 58864 | 14127360 | 116 |
| 65536 | 65536 | 15728640 | 104 |

Results are presented in Fig. 7, and are the median values of five executions on 10 compute nodes (sdumont[5004–5013]) using 24 cores per node, for a total of 240 cores.

For the small request sizes (1024 and 1312 bytes), shown in Fig. 7a, we can see performance differences between MPI implementations (for both read and write tests) that are very similar to what was observed for OLAM in Fig. 5. These differences are smaller and show different behavior for the large request sizes (58864 and 65536 bytes), shown in Fig. 7b and 7c. This confirms the differences observed in Section III are **not** specific to OLAM, but happen when small requests (of up to approximately 1 KB) are generated.

The Dunn statistical test was used to compare all sets of results. Write/read time obtained with IntelMPI and OpenMPI 1.10 with ROMIO were **not** significantly different. Results for OpenMPI 1.10 with OMPIO were **not** significantly different from the other two sets in small tests with HDF5 and small write tests with MPI-IO.

Time obtained with BullxMPI was significantly different from OpenMPI 1.10 and IntelMPI in most small tests, except small read tests with MPI-IO, where it was **not** significantly different from OpenMPI 1.10 with OMPIO. For large tests, BullxMPI and OpenMPI 1.10 were similar in most cases — except BullxMPI and OpenMPI 1.10 with OMPIO in read tests with HDF5 and 64 KB requests, and in read tests with MPI-IO and requests of 58864 bytes.

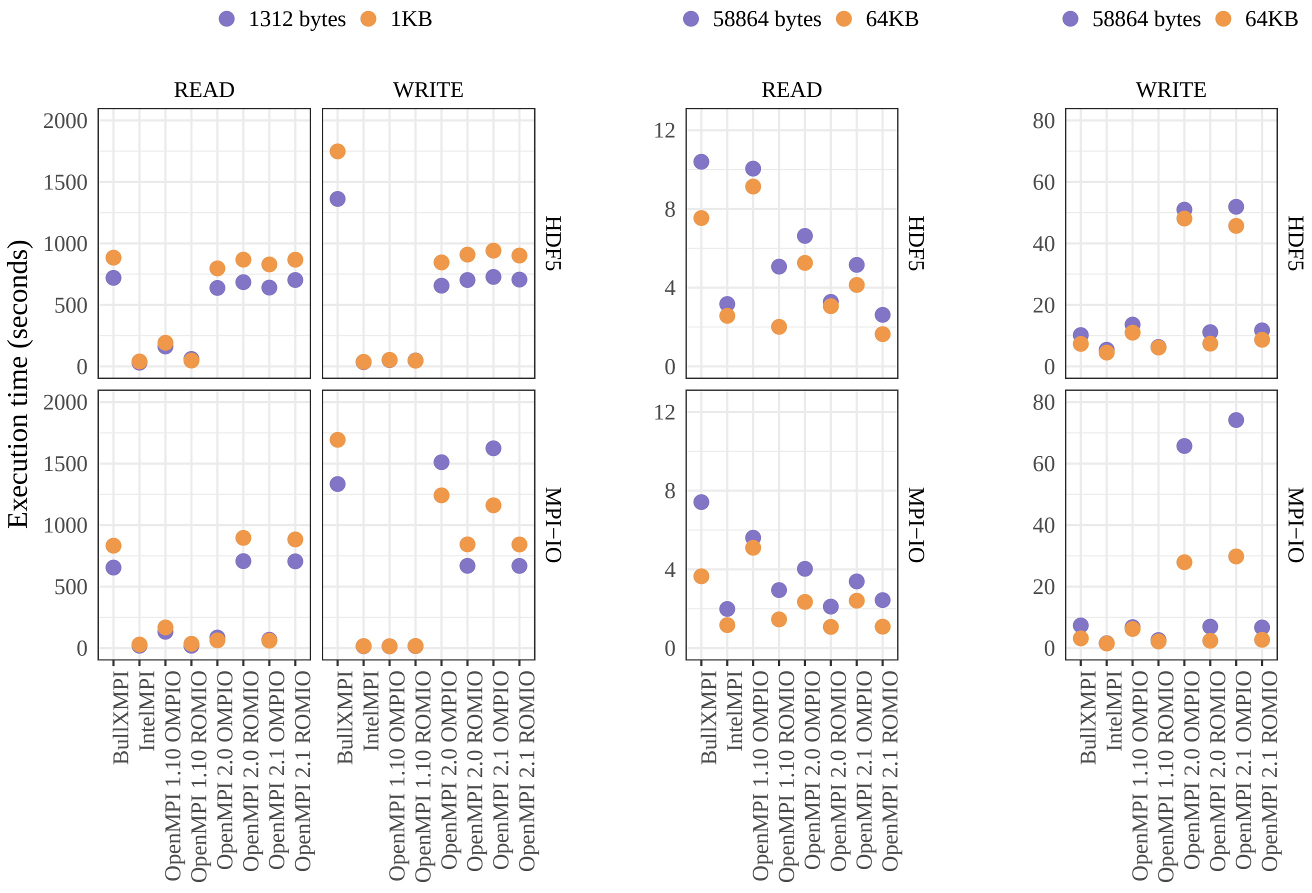
Comparing results for BullxMPI to the ones for OpenMPI 2.0 and 2.1, they are different when using ROMIO for small read experiments and large write experiments — except OpenMPI 2.0 with ROMIO in the tests with MPI-IO and requests of 58864 bytes. In small write experiments, BullxMPI results are significantly different from the ones for OpenMPI 2.0, except using MPI-IO with OMPIO for 1 KB requests and with both ROMIO and OMPIO for 1312 bytes requests, and from the ones for OpenMPI 2.1 with ROMIO, except using MPI-IO with requests of 1312 bytes. Finally, in large read experiments, results for BullxMPI are **not** significantly different from results for OpenMPI 2.0 and 2.1 with OMPIO in tests using MPI-IO with 64 KB requests.

As expected, larger requests lead to higher performance from all MPI implementations (all differences were confirmed with the Wilcoxon-Mann-Whitney test). The misaligned access only presented a large negative impact on versions 2.0 and 2.1 of OpenMPI with OMPIO, using the MPI-IO API and large request sizes. Nonetheless, the difference was **not** confirmed by the statistical test for read experiments with large requests, using MPI-IO through OpenMPI 2.1 with OMPIO.

Regarding the I/O APIs, HDF5 presented an inferior performance than MPI-IO, due to its added overhead. This difference was **not** confirmed by the Wilcoxon-Mann-Whitney test for large write tests with BullxMPI, write tests with requests of 58864 bytes with OpenMPI 1.10 with ROMIO, small read tests with OpenMPI 2.1 with ROMIO, read tests with 1 KB requests with IntelMPI, read tests with 58864 bytes requests with IntelMPI, OpenMPI 1.0 with ROMIO, OpenMPI 2.0 with ROMIO, and OpenMPI 2.1; and read tests with 64 KB requests with OpenMPI 1.10 with ROMIO and OpenMPI 2.1 with OMPIO.

## C. *Experiments with a custom microbenchmark*

After detecting the performance difference between MPI implementations in the OLAM experiments, and confirming though benchmarks that it happens for small collective I/O



(a) Small requests (b) Large read requests (c) Large write requests

Fig. 7: IOR results. It is important to notice the scale is **not** the same in all graphs.

requests, we developed a microbenchmark to further investigate this phenomenon. It was developed based on the two-phase collective write operations as implemented by ROMIO. The same MPI calls internally used by ROMIO are used to implement the different steps, and the microbenchmark reports time spent on each step. Table III details these steps. The source code is freely available at https://github.com/francielizanon/pretend_coll .

This experiment was configured to issue collective write requests of 1312 bytes, using 10 compute nodes (sdumont[5004–5013]) and all 24 cores per node, for a total of 240 cores. Fig. 8 shows results, which are median values from five executions. It is possible to notice that most of the time was spent in Step 3, where processes communicate request information (offset and size) to their aggregators. Most of the difference in performance between the MPI implementations comes from this step. There is some difference in step 4 as well, but on a much smaller scale.

We applied the Dunn test in each step to compare results for different MPI implementations. In step 3, BullxMPI results are significantly different from results obtained for OpenMPI 1.10, and IntelMPI results are different from OpenMPI 1.0, OpenMPI 2.0 with OMPIO, and OpenMPI 2.1 with ROMIO. In step 4, BullxMPI results are different from results with OpenMPI 1.10, and results with IntelMPI are different from OpenMPI 2.0 and OpenMPI 2.1.

These are surprising results, because, as detailed in Table III,

TABLE III: Steps of the custom microbenchmark that mimics two-phase collective write operations

| | |
|---|---|
| **Step 1** | Exchange messages between all processes so that every one knows the start and end offsets of the whole requested portion (`MPI_Allgather`). |
| **Step 2** | Exchange messages between all processes so that every one knows the number and size of original requests (`MPI_Allreduce`). |
| **Step 3** | Exchange messages between aggregators and processes to communicate the offsets and access sizes (`MPI_Isend` and `MPI_Irecv`). |
| **Step 4** | Exchange messages between aggregators and processes so that every aggregator obtain the data to perform the write operation (`MPI_Isend` and `MPI_Irecv`). |
| **Step 5** | Aggregators execute the I/O operation to the parallel file system (`MPI_File_write_at`). |

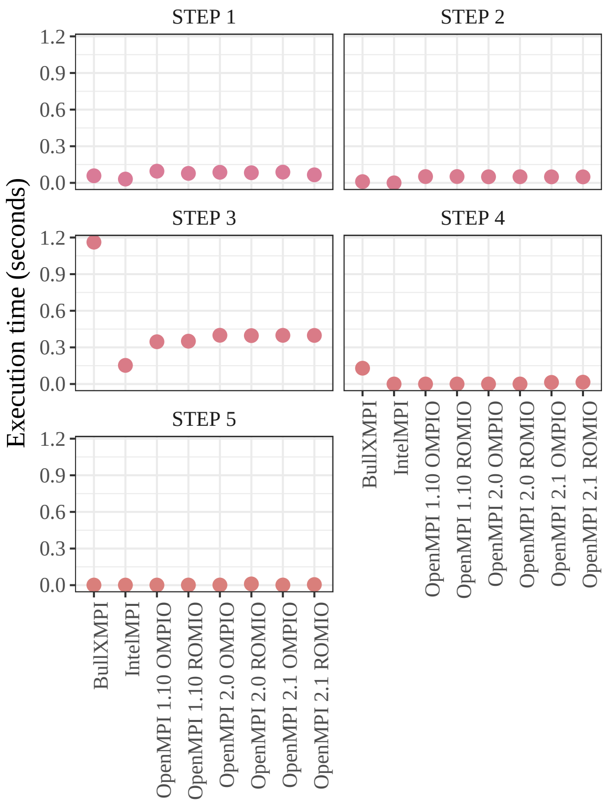


Fig. 8: Time on each step of the collective write operation

steps 3 and 4 use the same MPI calls. The main difference between them is that step 3 sends much smaller messages (just two numbers). None of the tested implementations seem to handle this situation well, and in the case of BullxMPI it causes the worst impact on performance.

It is also interesting to observe how performing the operation to the remote parallel file system, which could be expected to be the most important step for performance, does not account for most of the time. Instead, most of the time spent on such small collective write operations is used for coordinating processes and exchanging data. We have not observed differences for large collective calls in Sections IV-A and IV-B because their steps 4 and 5 will be longer, decreasing the impact of step 3.

## V. Related Work

A wide range of factors can negatively impact I/O performance in the HPC context. Understanding and characterizing a platform can provide insights on how the applications should perform I/O operations to obtain the best performance.

Using Darshan I/O logs on thousands of applications at the Intrepid, Mira and Edison supercomputers, the work conducted in [19] aimed at identifying behaviors that impacted performance to guide future optimizations. It is also pointed out that improving the I/O performance of the top used applications can yield great benefits for the system as a whole. In [20], the authors present a characterization of the storage performance of the Cray XK6 Jaguar supercomputer while examining the implications of those results for application performance. They observe and quantify limitations from competing traffic, concurrency, interference and stragglers of write operations on shared files. Our work, on the other hand, delves deeply into the use of collective I/O operations and the performance impact related to the choice of MPI implementation.

It is fairly common that vendors prepare customized MPI implementations for use in their supercomputers and large-scale clusters. These commercial solutions often claim improved performance over open-source alternatives. Some studies seek to evaluate such implementations considering distinct workloads and applications.

In [21] the authors evaluate three MPI implementations, two open-source (MPICH and LAM-MPI) and one proprietary (MESH-MPI). They employ benchmarks such as the NPB – NAS Parallel Benchmarks – and show the commercial implementation is significantly faster than the open-source alternatives. Additionally, they demonstrate that the customized solution yields much better performance for small collective communication operations. Closely related is the evaluation conducted in [22] for the Cray Red Storm computing platform. The vendor-supported MPICH2 implementation (MPICH2-0.97) is compared to two other solutions based on MPICH (MPICH-1.2.6 and MPICH-1.2.6 using SHMEM [23]). They demonstrate that the first is slightly outperformed by an open-source alternative in terms of latency and bandwidth. However, they do not take into account in their evaluation the use of collective I/O operations.

Many HPC applications issue collective I/O operations and their performance problems justify the considerable work that has been conducted on improving them. In [24], the authors propose an initial implementation of nonblocking collective I/O, as introduced by the MPI 3.1 standard. Their motivation is to satisfy the need to overlap computation and I/O and to hide the synchronization cost imposed by standard blocking collective I/O operations. In [25] a set of MPI-IO hints extension is proposed so users can harness locally attached SSDs to boost collective I/O performance by increasing parallelism and reducing global synchronization impact in the ROMIO implementation. In [26], the communication phase of the collective I/O operations is tuned for a Blue Gene/L supercomputer by exploring the specific network features of the machine. They modified the implementation of those collective I/O operations to use MPI directives that yield most performance in that architecture when exchanging messages between aggregators and processes.

To the best of our knowledge, no other work has investigated in such depth the performance differences between MPI implementations for small collective I/O operations on a large-scale cluster. Such investigation, motivated by issues observed with one of the main applications from the system, provides information to guide users to obtain higher I/O performance on the shared machine. Moreover, by reporting these findings, we hope to help improve future MPI implementations.

## VI. Conclusion and Future Work

In this paper, we have evaluated the performance of collective I/O operations in the Santos Dumont, the largest supercomputer in Latin America. Such an analysis is important to ensure the efficient use of the machine, as many applications spend a significant portion of their execution time in I/O operations. Initial results with OLAM, one of the main scientific applications executed in the machine, pointed a large performance difference between the MPI implementations available for users. We have further investigated this difference by conducting experiments with the BT-IO benchmark, the IOR benchmarking tool and with a custom benchmark.

Results have pointed that the observed difference happens for small requests (of approximately 1 KB), and comes from the step of the collective operation where processes exchange small messages to communicate request information to aggregators. With this work, we provide valuable information that can be used to improve future versions of collective write implementations. To the best of our knowledge, ours was the first work to show and investigate this phenomenon.

Furthermore, the most concrete contribution of this work is advice to be given to SDumont users, as it was observed over $20\%$ of jobs use BullxMPI, the implementation that presented the worst results in our analysis. By helping users achieve better performance for their applications, we promote a better usage of the machine.

Future work includes further optimizing the OLAM code to issue larger request sizes, and to benefit from the local SSD devices, using them as burst buffers [27]. Regarding the Santos Dumont supercomputer, we plan to collect more detailed information about applications that use it and their I/O requirements. This information will help further characterizing and optimizing its I/O infrastructure.

## Acknowledgment

This research has received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4e Project, grant agreement n$^\circ$ 689772. It has been partially supported by CAPES Brazil and by the EU H2020 EoCoE project (http://www.eocoe.eu/). We thank RICAP, partially funded by CYTED, Ref. 517RT0529. 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. URL: http://sdumont.lncc.br

## References

[1] SUN, "High-performance storage architecture and scalable cluster file system," Tech. Rep., 2007. [Online]. Available: http://www.csee.ogi.edu/~zak/cs506-pslc/lustrefilesystem.pdf

[2] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings...*, ser. FAST '02, 1st USENIX Conference on File and Storage Technologies. USENIX, 2002.

[3] 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)*. IEEE, Sept 2014, pp. 185–193.

[4] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: Reading patterns for extreme scale science IO," in *Proceedings...*, ser. HPDC '11, 20th International Symposium on High Performance Distributed Computing. ACM, 2011, pp. 49–60.

[5] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/O acceleration with pattern detection," in *Proceedings...*, ser. HPDC '13, 22nd International Symposium on High-Performance Parallel and Distributed Computing. ACM, 2013, pp. 25–36.

[6] Y. Yin, J. Li, J. He, X.-H. Sun, and R. Thakur, "Pattern-direct and layout-aware replication scheme for parallel i/o systems," in *Proceedings...*, 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS). IEEE, May 2013, pp. 345–356.

[7] Z. Wang, X. Shi, H. Jin, S. Wu, and Y. Chen, "Iteration based collective I/O strategy for parallel I/O systems," in *Proceedings...*, ser. CCGrid'14, May 2014, pp. 287–294.

[8] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong, "Overview Of The MPI-IO Parallel I/O Interface," 1995.

[9] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved Parallel I/O via a Two-phase Run-time Access Strategy," *ACM SIGARCH Computer Architecture News*, vol. 21, no. 5, pp. 31–38, Dec. 1993.

[10] "Atos supports Brazil in the race to High-Performance Computing with the installation of a Bull supercomputer in Petropolis (RJ)." [Online]. Available: https://atos.net/en/2015/press-release/deals-contracts-press-releases_2015_09_07/pr-2015_09_07_01

[11] OLAM, "Ocean Land Atmosphere Model," https://sourceforge.net/projects/olam-model/, accessed: November 2017.

[12] The HDF Group, "Hierarchical Data Format, v5," 1997-2016, /HDF5/.

[13] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access Through Continuous Characterization," *Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011.

[14] N. Feltovich, "Nonparametric tests of differences in medians: Comparison of the wilcoxon–mann–whitney and robust rank-order tests," *Experimental Economics*, vol. 6, no. 3, pp. 273–297, 2003.

[15] O. J. Dunn, "Multiple comparisons among means," *Journal of the American Statistical Association*, vol. 56, no. 293, pp. 52–64, 1961.

[16] NASA, "Nas Parallel Benchmarks (NPB)," https://www.nas.nasa.gov/publications/npb.html, accessed: November 2017.

[17] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. Dantas, and Y. Denneulin, "A Checkpoint of Research on Parallel I/O for High Performance Computing," Tech. Rep., 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01591755

[18] H. Shan and J. Shalf, "Using ior to analyze the i/o performance for hpc platforms," in *In: Cray User Group Conference (CUG'07)*, 2007.

[19] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of i/o behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 33–44.

[20] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky, S. Oral, and N. Podhorszki, "Characterizing output bottlenecks in a supercomputer," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[21] B. Vinter, J. M. Bjrndalen, O. J. Anshus, and T. Larsen, "A comparison of three mpi implementations," in *in Proc. of Communicating Process Architectures (CPA 2004)*. Morgan Kaufman, 2004, pp. 127–136.

[22] R. Brightwell, *A Comparison of Three MPI Implementations for Red Storm*. Berlin, Heidelberg: Springer, 2005, pp. 425–432.

[23] ——, *A New MPI Implementation for Cray SHMEM*. Berlin, Heidelberg: Springer, 2004, pp. 122–130.

[24] S. Seo, R. Latham, J. Zhang, and P. Balaji, "Implementation and evaluation of mpi nonblocking collective i/o," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1084–1091.

[25] G. Congiu, S. Narasimhamurthy, T. S, and A. Brinkmann, "Improving Collective I/O Performance Using Non-volatile Memory Devices," in *2016 IEEE International Conference on Cluster Computing*, Sept 2016, pp. 120–129.

[26] H. Yu, R. K. Sahoo, C. Howson, G. Almasi, J. G. Castanos, M. Gupta, J. E. Moreira, J. J. Parker, T. E. Engelsiepen, R. B. Ross, R. Thakur, R. Latham, and W. D. Gropp, "High performance file I/O for the Blue Gene/L supercomputer," in *The 12th International Symposium on High-Performance Computer Architecture, 2006.*, Feb 2006, pp. 187–196.

[27] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012, pp. 1–11.