Estimation of the impact of I/O forwarding on application performance

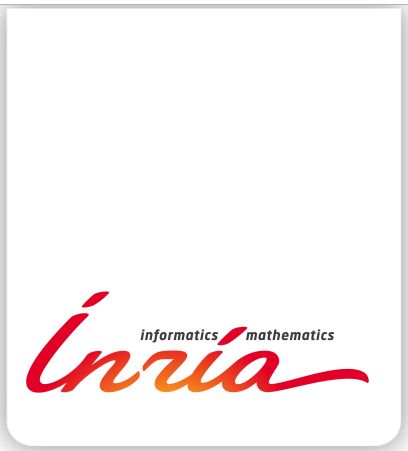
Francieli Zanon Boito

Estimation of the impact of I/O forwarding on application performance

Francieli Zanon Boito

RESEARCH REPORT
N° 9366
October 2020
Project-Team TADaaM

ISSN 0249-6399 ISRN INRIA/RR--9366--FR+ENG



Estimation of the impact of I/O forwarding on application performance

Francieli Zanon Boito

Project-Team TADaaM

Research Report n° 9366 — October 2020 — 20 pages

Abstract: In high performance computing architectures, the I/O forwarding technique is often used to alleviate contention in the access to the shared parallel file system servers. Intermediate I/O nodes are placed between compute nodes and these servers, and are responsible for forwarding requests. In this scenario, it is important to properly distribute the number of available I/O nodes among the running jobs to promote an efficient usage of these resources and improve I/O performance. However, the impact different numbers of I/O nodes have on an application bandwidth depends on its characteristics. In this report, we explore the idea of predicting application performance by extracting information from a coarse-grained aggregated trace from a previous execution, and then using this information to match each of the application's I/O phases to an equivalent benchmark, for which we could have performance results. We test this idea by applying it to five different applications over three case studies, and find a mean error of approximately 20%. We extensively discuss the obtained results and limitations to the approach, pointing at future work opportunities.

Key-words: high performance computing, parallel I/O, tracing, performance prediction

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Estimation du impact de l'utilisation d'I/O forwarding sur les performances des applications

Résumé : Dans les plate-formes pour calcul hautes performances, la technique d'I/O forwarding est souvent utilisée pour atténuer les conflits d'accès aux serveurs du système de fichiers parallèle, qui sont partagés par les applications. Les noeuds d'I/O intermédiaires sont placés entre les noeuds de calcul et ces serveurs et sont responsables de la transmission des demandes. Dans ce scénario, il est important de répartir correctement le nombre de noeuds d'I/O disponibles parmi les jobs en cours d'exécution pour promouvoir une utilisation efficace de ces ressources et améliorer les performances d'I/O. Cependant, l'impact de différents nombres de noeuds intermédiaires sur la bande passante d'une application dépend de ses caractéristiques. Dans ce rapport, nous explorons l'idée de prédire les performances de l'application en extrayant des informations d'une trace agrégée à gros grain d'une exécution précédente, puis en utilisant ces informations pour faire correspondre chacune des phases d'I/O de l'application à un benchmark équivalent, pour lequel on pourrait avoir des résultats de performance. Nous testons cette idée en l'appliquant à cinq applications différentes sur trois études de cas, et trouvons une erreur moyenne d'environ 20%. Nous discutons longuement les résultats obtenus et les limites de l'approche, en indiquant des opportunités pour travaux futures.

Mots-clés : calcul hautes performances, E/S parallèles, I/O forwarding, traces, prediction de performance

Contents

1 Introduction

In high-performance computing (HPC) architectures, parallel file systems (PFS) provide persistent storage to applications though a set of dedicated data and metadata servers. The storage infrastructure, illustrated in Figure 1, often includes a set of intermediate *I/O nodes* that receive requests from the clients and forward them to the PFS. This approach is called I/O forwarding, and has the advantage of alleviating contention in the access to the shared servers. Moreover, it provides optimization opportunities such as request reordering, aggregation, and scheduling. [6, 8]

The assignment of I/O nodes to applications is often made statically, i.e. for N processing and M I/O nodes, each I/O node is connected to N/M processing nodes, which are not able to communicate with other I/O nodes. Hence the I/O nodes that an application will use depend on its placement. Nonetheless, other alternatives are possible. In some systems such as Tianhe-2 and Sunway TaihuLight, it is possible to configure how many and which I/O nodes each job accesses [10, 4].

Using a variable number of I/O nodes per application, instead of simply choosing it based on the used compute nodes, is useful because the processing needs of the application (which affects the number of processors) are not necessarily correlated to the data access needs (which affects the utilization of I/O nodes). Static approaches result in load imbalance and wastes resources (the I/O nodes) that could be used to improve the performance of other concurrent applications [12].

Related work has focused on proposing better allocation strategies that try to assign more I/O nodes to data-intensive applications, and that try not to co-locate jobs that could impose too much interference to each other [4]. Nonetheless, their approach assumes having more I/O nodes always translates in better application I/O performance if it imposes a heavy-enough workload. However, as we show in this report, the I/O performance an application will have with different numbers of I/O nodes depend not only on its intensitivity, but also on its access pattern — aspects such as the number of accessed files, the spatiality of these accesses, the size of requests, etc. Figure 2 illustrate this using as example three applications that will appear later in this report as case studies.

That means we need better allocation strategies that take the applications' characteristics into consideration in order to promote a better usage of the I/O forwarding layer and improve I/O performance across the machine. Nevertheless, such a technique requires the knowledge of the performance each application would experience with different numbers of I/O nodes. Therefore, **in this report, we focus on the first step towards better allocation strategies for the I/O forwarding layer of HPC architecture: on estimating, to an application, its performance as a function of the number of used I/O nodes**.

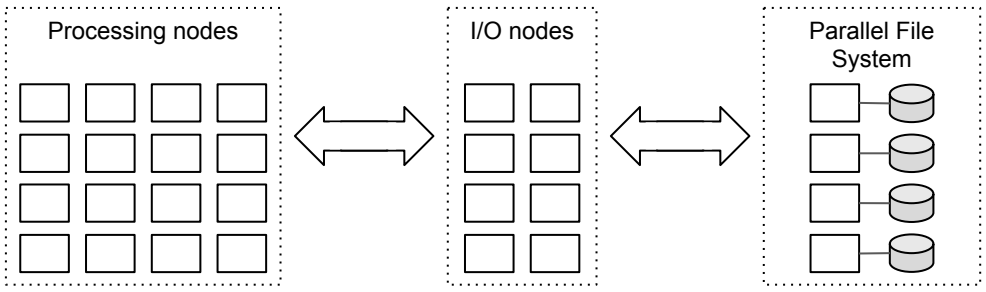
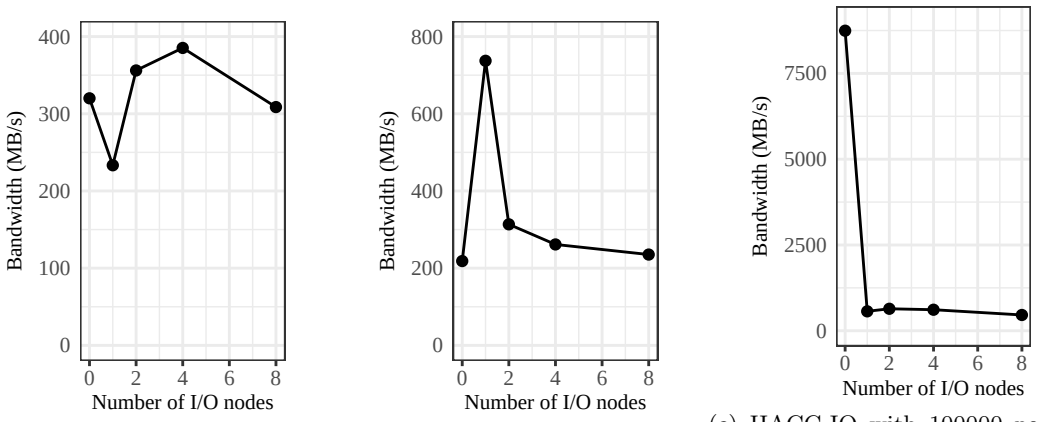


Figure 1: High-level view of the storage infrastructure of HPC architectures



(a) SINGLE-A (Section 3) (b) PERIODIC-A (Section 4) (c) HACC-IO with 100000 particles per process (Section 5)

Figure 2: Three applications, executed on the environment described in the Appendix B, are impacted differently when changing the number of I/O nodes. Lines are added to aid the visualization of the tendencies. The y axis in different for each plot.

The challenge here is that jobs submitted to run on a supercomputer only contain the information of expected execution time and number of processors, not about I/O activity. This information is also not easily obtained from the application binary or even from the users. Moreover, even with a description of what is done by the application, today there are not good, comprehensive, accurate performance models that would allow us to simply achieve such an estimation (and for good reason: building such models is a complex task because performance is affected by too many factors). A solution could be to run the application with different numbers of I/O nodes (multiple times, because of variability) and use this result for future submissions. Nonetheless, requiring such profiling runs would have a high cost, and any allocation technique based on this information would only be able to benefit applications after they were executed many times.

In this report, we test the hypothesis that we could estimate application performance by extracting its I/O characteristics from a coarse-grained aggregated trace and then leveraging results observed for benchmarks with similar characteristics. The use of benchmark results would decrease the cost of obtaining the required information, as benchmarks run faster than the application (they don't have the processing and communication parts) and be useful to multiple applications that present similar access patterns.

We assume a database of I/O traces from previously executed applications is available, similarly to what was done in other research efforts [4]. For our case studies, we use Darshan traces [3], because they are generated by default in many supercomputers[1]. Still, our solution is not dependent on the use of Darshan and could be used with other monitoring tools. In this situation, **when a job is submitted, we would find traces from previous executions of the same application** (which will often be possible because HPC machines are used to run the same applications multiple times), **extract the different I/O phases' characteristics, identify similar benchmarks to each I/O phase, and then use the observed bandwidth of these benchmarks to estimate the I/O performance of the application**.

This report is organized as follows. Section 2 discusses how we estimated I/O phases from

[1] `https://www.mcs.anl.gov/research/projects/darshan/`

Darshan traces in the presented case studies. Then, in Sections 3 to 5 **our hypothesis is tested through a sequence of case studies** of increasing complexity. Section 6 summarizes the proposed technique, and its advantages and limitations observed during the case studies, and identifies future work opportunities. Section 7 closes this report with final considerations.

2 Application I/O phases

Many supercomputers activate some monitoring tools by default for all their jobs, as is often the case for Darshan. In its traditional usage, this I/O monitoring tool intercepts I/O operations and keeps per-node counters during the execution of the job. At the end of the execution, the first rank aggregates all information into statistics that are compressed and written to a per-job file, called a *Darshan trace*. These counters include the number of reads and writes and of bytes accessed through different interfaces, timestamps of the first and the last accesses to each file, etc. Although the generation of more detailed finer-grained traces being possible with Darshan, the traditional aggregated coarse-grained traces are preferred because they minimize overhead and because they are smaller in size and hence easier (and cheaper) to store and handle.

In this work, we want to leverage information from the aggregated coarse-grained traces to predict how the application's I/O performance would be impacted by the use of different numbers of intermediate I/O nodes. We choose to use these traces because they are already generated at many supercomputers, so our prediction would not impose further overhead to the applications or require changes in the infrastructure.

From the set of counters in the Darshan trace of a job, we generate a set of *I/O phases*. Each I/O phase is characterized by:

- timestamps of start and end;
- operation type (read or write);
- involved ranks (the rank if a single one, or the number of ranks in case of multiple ones);
- amount of data;
- time spent on I/O (which may be smaller than $end - start$);
- API;
- number of requests;
- spatiality (consecutive, sequential, or random for POSIX, collective or unknown for MPI-IO);
- number of files and access fashion (file-per-process or shared file).

These fields were chosen based on the hypothesis that they allow for performance prediction, and limited by what is available from the Darshan counters. Details on how the I/O phase characterization is estimated from the Darshan traces using the SummarIO tool are presented in the Appendix A. This strategy was inspired by the one proposed by Bez. et al [1], but expanded to include more information about the phases and to group similar phases to present a more concise view of the application. The SummarIO tool is freely available at `https://gitlab.inria.fr/hpc_io/summario` .

A limitation of the obtained phases is that the timestamps of start and end of the phase come from the first and last accesses to a file, hence do not necessarily represent the start and end of

an I/O-intensive part of the application execution. Indeed, multiple "logical" application phases could appear as a single phase if they access the same file. Indeed, the use of high-level aggregated traces means the identified application I/O phases are estimations, as detailed information could have been lost.

It is important to notice that the performance estimation part of this work does **not** depend on the phase estimation. Therefore, **our proposal is not dependent on the use of Darshan traces**. Indeed, information of the I/O phases of an application could be obtained from another monitoring tool, including a more detailed one.

3 Case study I — Single-phase synthetic applications

The first case study is the simplest possible scenario: the application is composed of a single POSIX I/O phase, without any processing to be done. That is the simplest case because more information is available about POSIX phases (notably the spatiality) and because the single phase of "concentrated" I/O activity will be easier to estimate. To represent this scenario, we use two benchmarks obtained from the benchmarking tool IOR[2], SINGLE-A and SINGLE-B. They are described in the Table 1. The choice of parameters for this report was arbitrary, while seeking to represent multiple realistic patterns and performance "curves" across the case studies.

Table 1: Description of SINGLE-A and SINGLE-B applications. Both write data using POSIX.

	Nodes	**Processes**	**Files**	**Spatiality**	**Amount of data (total)**	**Request size**
SINGLE-A	16	128	shared-file	contiguous	16 GB	2 MB
SINGLE-B	64	512	file-per-process	contiguous	32 GB	4 MB

The two codes were executed with the goal of obtaining their Darshan traces, which were then provided to the SummarIO code. The result is presented in Table 2. A single phase was identified to each application. For SINGLE-A we know that 128 processes wrote contiguous requests to a shared-file. Request size is easily obtained by dividing *bytes* by *#requests*. For SINGLE-B, we know all processes were involved in phases where a single process accessed its own file (therefore a file-per-process phase). We can estimate the number of processes from the number of files.

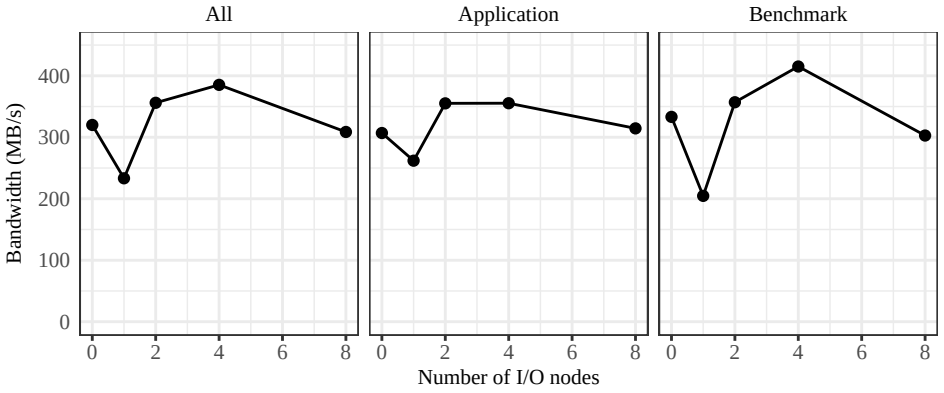
Table 2: I/O phases from applications SINGLE-A and SINGLE-B, estimated with SummarIO.

job	**start**	**end**	**type**	**rank**	**bytes**	**time**
SINGLE-A	0.026459	71.625603	write	all	17179869184	47.51776984375
	api	**#requests**	**spatiality**	**files**	**#files**	**mode of #processes**
	POSIX	8192	contiguous	shared	1	128
job	**start**	**end**	**type**	**rank**	**bytes**	**time**
SINGLE-B	0.064315	99.164715	write	all	34359738368	99.092545
	api	**#requests**	**spatiality**	**files**	**#files**	**mode of #processes**
	POSIX	8192	contiguous	unique	512	1

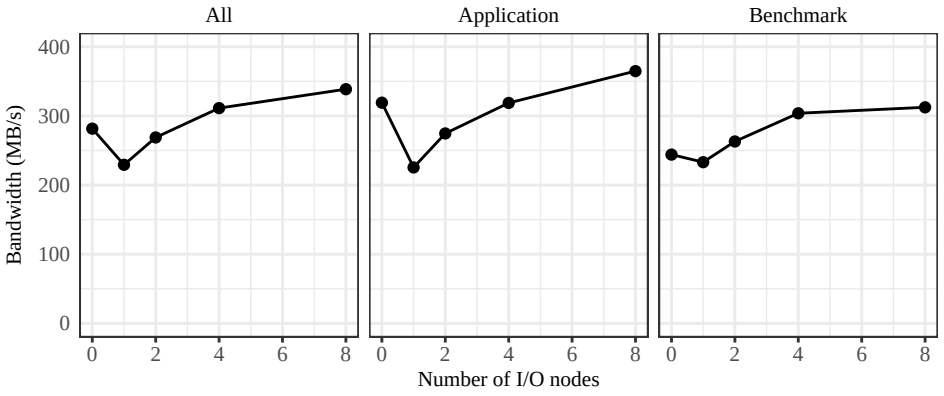
[2] `https://github.com/hpc/ior`

The estimated phases match therefore the characteristics presented in Table 1. That means that, using IOR as the benchmarking tool, in this particular case the benchmarks generated to mimic these phases will be the applications themselves. Therefore, to evaluate our approach in this case study, we executed the SINGLE-A and SINGLE-B applications 10 times each, and then randomly marked half of the executions as "benchmark" and half as "application". Results are presented in Figure 3. All experiments presented in this report were conducted using the Grid'5000 platform [2]. Details of the experimental methodology are presented in the Appendix B.

Although the three curves in each plot are obtained from repetitions of the same code, they present some differences. That happens because of the variability observed in these executions. Figure 4 illustrated this variability by plotting the bandwidth of all ten repetitions of each application with each number of I/O nodes. The performance of SINGLE-A with 4 I/O nodes varied from 286 to 512 MB/s, and the performance of SINGLE-B with 0 I/O nodes from 217 to 365 MB/s. The standard variation was between 58 and 99 MB/s for SINGLE-A and between 21 and 67 MB/s for SINGLE-B. Variability in I/O performance is a well-known phenomenon [9, 7],



(a) SINGLE-A



(b) SINGLE-B

Figure 3: Performance (mean of multiple executions) of SINGLE-A and SINGLE-B applications as a function of the number of I/O nodes. The ten executions were randomly separated into "application" and "benchmark". "All" presents the mean of all ten results. The lines are added to aid the visualization of tendencies.

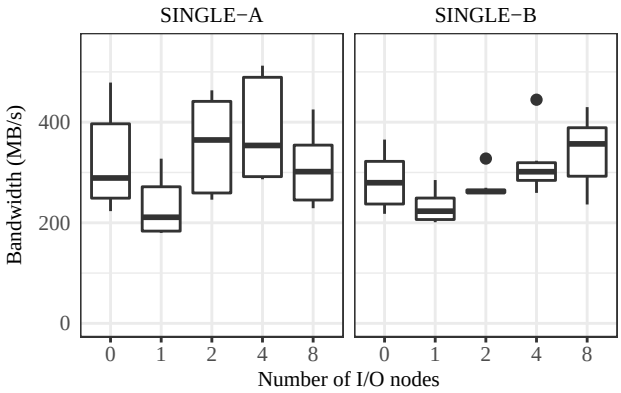


Figure 4: Boxplots of the ten results obtained for each number of I/O nodes, separated by application. The line in the middle represents the median, and the lower and upper hinges show the first and third quartiles. The whiskers extend to other values at a distance from the hinges of up to 1.5×IQR, which is the distance between the first and third quartiles. Points beyond that distance are "outliers" and plotted as points.

and is has been shown to depend on the application access pattern [5].

The pairs of samples to each number of I/O nodes were compared using the Dunn statistical test . The p-value has higher than $\alpha/2$ for all pairs (except for 0 I/O nodes in SINGLE-B), meaning we cannot say the application and benchmark executions are significantly different. Indeed, the differences between the "Application" and "Benchmark" curves are due to luck in the random labeling of the repetitions. It is possible a higher number of repetitions would result in more similar curves, as larger samples would represent better the distribution they come from.

We calculated the error we would cause when estimating application performance by the mean bandwidth observed for the benchmark. The error is calculated to each of the five application repetitions with each number of I/O nodes as $|B_w - B_a|/B_a$, where B_w is the performance of the benchmark and B_a of the application. The distribution of the 25 obtained values for each application are described in Table 3. The error observed for SINGLE-B was smaller than for SINGLE-A because the former presented less variability in the results of different executions, as showed in Figure 4.

Both estimations allow for identifying the best number of I/O nodes (four for SINGLE-A and 8 for SINGLE-B). However, some differences are "deformed".

4 Case study II — Periodic synthetic applications

The second case study consists also of two synthetic applications, generated with the IOR benchmarking tool. These applications, PERIODIC-A and PERIODIC-B, are composed of eight peri-

Table 3: Distribution of error when using the mean of benchmark executions as the estimate for each application execution.

	minimum	**mean**	**median**	**maximum**
SINGLE-A	0.1	0.25	0.26	0.45
SINGLE-B	0.0	0.14	0.16	0.33

odic I/O phases, separated by 10-second compute phases. Their details are presented in Table 4.

Table 4: Description of applications PERIODIC-A and PERIODIC-B. Both write data via POSIX in eight regular phases, separated by 10-second intervals (representing a compute phase). In PERIODIC-A, a new shared file is created at each phase.

	Nodes	**Processes**	**Files**	**Spatiality**	**Amount of data (total)**	**Request size**
PERIODIC-A	24	24	shared-file	1D-strided	24 GB	512 KB
PERIODIC-B	24	24	file-per-process	contiguous	24 GB	4 KB

Similar to what was done in the previous Section, to the first case study, applications were executed at first to obtain the traces, which were parsed with the SummarIO tool to generate an estimation of their I/O phases. Eight identical phases (differing only on start and end timestamps and slightly on duration) were obtained for each application. The first phase of each application is presented in Table 5.

Table 5: **First I/O phase** of PERIODIC-A and PERIODIC-B, estimated with SummarIO.

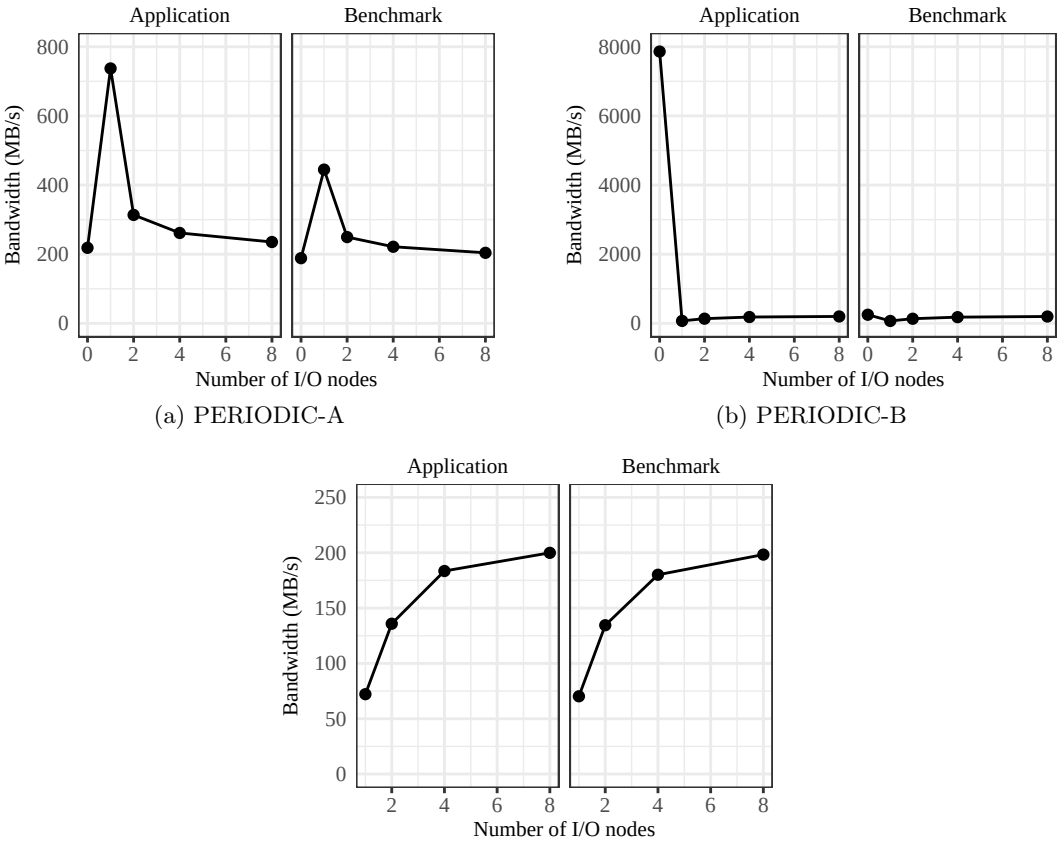
job	**start**	**end**	**type**	**rank**	**bytes**	**time**
PER.-A	10.031531	24.02671	write	all	3221225472	11.1388105416667
	api	**#requests**	**spatiality**	**files**	**#files**	**mode of #processes**
	POSIX	6144	sequential	shared	1	24
job	**start**	**end**	**type**	**rank**	**bytes**	**time**
PER.-B	10.030157	20.391011	write	all	3221225472	10.334366
	api	**#requests**	**spatiality**	**files**	**#files**	**mode of #processes**
	POSIX	786432	contiguous	unique	24	1

From this information we can identify that:

- During each phase from PERIODIC-A, all 24 processes performs a write to a shared file in 512 KB requests for a total of 3 GB of data. The "sequential" spatiality means not contiguous, but also not random. Considering HPC applications tend to output structured data, **we assume "sequential" spatiality represents the 1D-strided access pattern**.
- During each PERIODIC-B's phase, all ranks are involved in the access to 24 per-process files to write a total of 3 GB in 4KB requests.

We therefore generated, with IOR, the benchmarks that correspond to these two phases. Because the eight phases of each application are identical, we use directly the bandwidth of the benchmark to estimate for the application. Details about the experimental methodology are discussed in the Appendix B. Figure 5 compares the performance observed for application and benchmark. A third graph was added to compare PERIODIC-B results excluding the ones with 0 I/O nodes, because the very high point made the visualization of the other points difficult.

We can see in Figure 5a that the two curves for PERIODIC-A are similar, with the benchmark consistently underestimating performance. A large difference was observed only when using a



(a) PERIODIC-A

(b) PERIODIC-B

(c) PERIODIC-B (number of I/O nodes > 0)

Figure 5: Performance (mean of five executions) of PERIODIC-A and PERIODIC-B applications as a function of the number of I/O nodes. The lines are added to aid the visualization of tendencies.

single I/O node (737 vs. 444 MB/s). These experiments did not present a high variability, with standard deviations between 2 and 36 MB/s. However, if we investigate the performance of each I/O phase of the application when running with 1 I/O node, it varies from 513 to 1195 MB/s. Still, the corresponding benchmark did not present such a variation. A similar phenomenon can was observed for PERIODIC-B (Figures 5b and 5c) when using zero I/O nodes (7862 vs. 250 MB/s): the performance of each application phase varied between 280 MB/s and 12 GB/s, while the equivalent benchmark presented a stable lower performance.

Such a difference in behavior is **not** explained by differences in the codes of application and benchmark, because they were both generated with IOR using the same parameters (except for number of phases and time between them for the application). The only distinction is that the benchmark repetitions were **not** executed in sequence, but randomly in a mix of multiple experiments, as detailed in the Appendix B. We conclude thus that **in some situations periodic phases present a high performance variability that seems to comes from the fact of being executed in sequence, benefiting from a "warmed up" system**.

Table 6 presents the error obtained when using the benchmark average performance to es-

timate each of the five application executions. Despite the divergent cases discussed above, we observed a 20% mean error, similarly to what was observed for the first case study (Section 3). The estimated curves allow for the identification of the optimal number of I/O nodes to be used for each application.

Table 6: Distribution of error when using the mean of benchmark executions as the estimate for each application execution.

	minimum	**mean**	**median**	**maximum**
PERIODIC-A	0.10	0.20	0.16	0.44
PERIODIC-B	0.0	0.2	0.2	0.98

5 Case study III — HACC-IO

In the previous sections, synthetic applications were used to explore situations where the application is composed of a single or multiple periodic I/O phases. In this third case study, we use HACC-IO[3], which is the I/O kernel of a real scientific application called HACC. It was executed will default options, in the write-only version, using 8 processes on 8 nodes and 100,000 particles per process. The only phase obtained from the Darshan trace with the SummarIO tool are presented in Table 7.

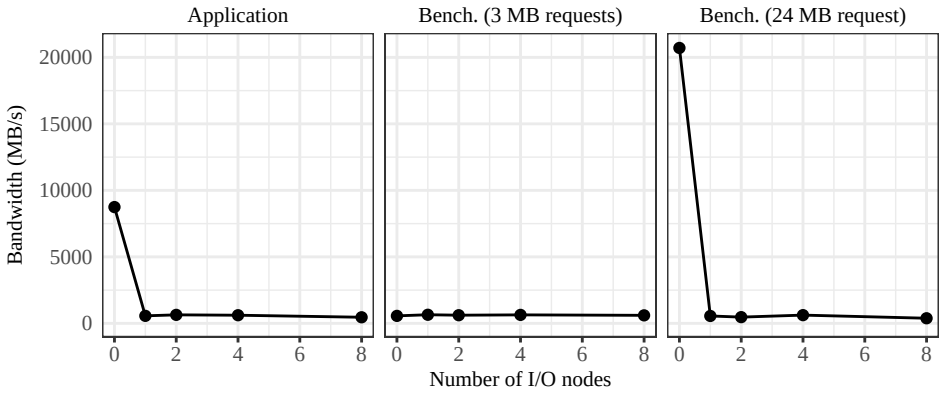
Table 7: I/O phases from the HACC-IO application, estimated with SummarIO.

job	**start**	**end**	**type**	**rank**	**bytes**	**time**
HACC-IO	0.032241	2.386244	write	all	1853812736	2.337255
	api	**#requests**	**spatiality**	**files**	**#files**	**mode of #processes**
	POSIX	640	contiguous	unique	64	1

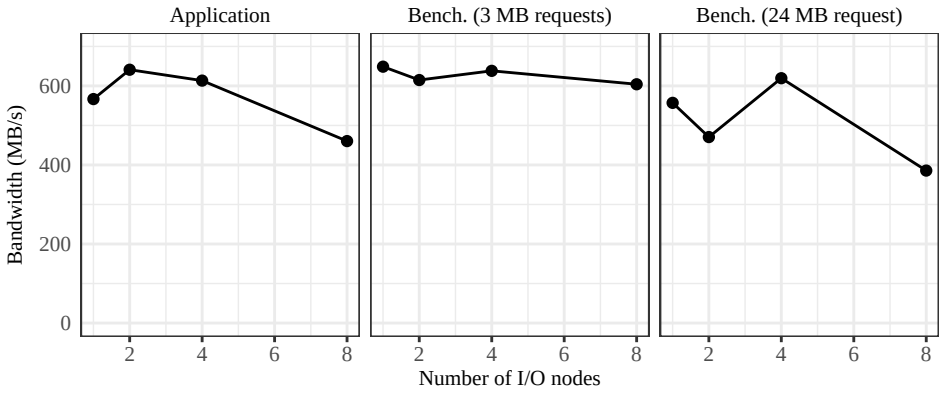
From these phases, we can conclude that HACC-IO has a single file-per-process write where 64 processes issue 2.8 MB requests for a total of 27.6 MB per file. These exact values not being accepted by IOR, we estimate performance for this application using a benchmark with these characteristics, but where each process accesses 30 MB in 3 MB requests. The first two boxes in the plots of Figures 6a and 6b compares the performance obtained for the application and for this first benchmark. The second plot is the same as the first, but removing the point for 0 I/O nodes to allow for the visualization of the other points. The main difference between the two curves was for 0 I/O nodes (8745 vs. 564 MB/s). Notably, in this case best number of I/O nodes for the benchmark (1, followed by 4) was not the same as for the application (0, followed by 2).

Studying the code of the application, one can find out that it actually does not generate ten 3 MB requests. Each process generates 10 requests of different sizes — 9 to write the contents of 9 different arrays, for a total of approximately 3.6 MB (which depends on the number of particles), and a last one to write a 24 MB header. We cannot identify that from the phase presented in Table 7, as that is a limitation imposed by the use of a coarse-grained aggregated Darshan trace — detailed information about individual requests is lost. To have an idea of the estimate we could get with more detailed information, we generated with IOR a file-per-process benchmark where each process generates a single request of 24 MB, emulating therefore the portion of HACC that handles the most data. The performance observed for this benchmark

[3] `https://github.com/glennklockwood/hacc-io/`



(a) 0–8 I/O nodes



(b) Number of I/O nodes > 0

Figure 6: Performance (mean of five executions) of HACC application and estimation benchmarks as a function of the number of I/O nodes. Processes executing the benchmark in the second box of each plot access 30 MB of data in 3 MB requests. The benchmark in the third box of each plot generates a single 24 MB request per process. The lines are added to aid the visualization of tendencies.

is presented in the third box of the plots in Figure 6. Although still quite different from the application curve, with this benchmark we would be able to find the best number of I/O nodes is zero, and that among the other options 8 is the one with the worst performance.

Table 8 summarizes the error obtained by using the average performance of each benchmark to estimate the bandwidth of each of the five HACC-IO executions. The relative better quality of the estimation obtained with the benchmark that generates a single 24 MB request per process does not translate into a lower mean or median error, in part because of the large difference in performance for 0 I/O nodes (8745 vs. 20711 MB/s).

Table 9 lists the performance observed for the application and the second benchmark when running without I/O forwarding (zero I/O nodes). Both seem to present two groups of results: one in the hundreds of MB/s, and another at tens of GB/s. If that is the case, then the large difference in average bandwidth would be coming from the small number of repetitions (we could think it was due to luck that the application had a single high-performance execution while the

Table 8: Distribution of error when using the mean of benchmark executions as the estimate for each application execution, separated by benchmark. In the first one, each process generates 10 requests of 3 MB each, and in the second a single 24 MB request is generated by process. The benchmarks are identical regarding other parameters. The first one is what would be used in our approach for I/O performance estimation.

	minimum	**mean**	**median**	**maximum**
Bench. (10 x 3 MB requests)	0.10	0.22	0.12	0.99
Bench. (a 24 MB request)	0.1	7.77	0.15	61.47

Table 9: Bandwidth (MB/s) of each repetition of the application HACC-IO and of the second estimation benchmark (where each process generates a single 24 MB request) with 0 I/O nodes. The repetition number of only to identify them and is not related to the order in which they were executed.

Repetition	**1**	**2**	**3**	**4**	**5**
Application	488.3	526.2	331.5	41971.4	409.7
Bench. (a 24 MB request)	370.1	287.1	52145.5	50370.5	383.0

benchmark had two).

6 Estimation of the impact of I/O forwarding on application performance

In this report, we aim at testing the hypothesis that it is possible to estimate application performance as a function of the number of I/O nodes by using an aggregated trace from a previous execution and benchmark results.

For a given application, the tested approach consists of estimating its I/O phases from the Darshan trace using the SummarIO tool. Then, **a benchmark is identified to each I/O phase** with the following characteristics:

1. In shared-file phases, the number of processes come from the *mode of #processes*. In file-per-process phases, it comes from the number of files.
2. Operation (read or write), API, and file strategy (shared or file-per-process) are taken directly from corresponding fields.
3. The amount of data per process is estimated by dividing the total amount of data by the number of processes, and the request size by dividing by the number of requests.
4. Consecutive spatiality is represented by a contiguous benchmark, sequential by 1D-strided (as discussed in Section 4), and unknown MPI-IO phases are supposed to be contiguous.
5. Finally, the number of nodes is not available in the Darshan trace, so it cannot be obtained by SummarIO. Still, we consider that information can be easily obtained from the resource manager of the machine.

As discussed in the first item above, the number of processes of file-per-process phases is estimated from the number of files. That could lead to an incorrect quantity if each process accesses multiple files one after the other (with a short time between them), because these

consecutive file-per-process phases could be aggregated in the third step of SummarIO, as detailed in Appendix A. As an improvement, we could imagine obtaining the number of processes used by the application from the command line used to execute it, if available. In that case, the number of running processes would be an upper limit to the number of involved in an I/O phase.

The amount of data accessed by process and the size of each request is supposed to be homogeneous, which would be incorrect for applications with load imbalance among the processes, or that make requests of different sizes during the same I/O phase. The latter is the case of HACC with 100,000 particles per process, as discussed in Section 5. Both the amount of data and request size are crucial for a good performance estimation, and therefore worse results are expected to these cases then to the homogeneous ones. As future work, we plan to improve request size estimation by using the histogram of the five most frequent request sizes provided among the Darshan counters for each phase. That is expected to improve results for these situations, as illustrated in the Case Study III when we added results with a benchmark closer to the application characteristics.

If an application has multiple I/O phases, then their bandwidths will be used to calculate an estimation of the I/O time (based on how much data each I/O phase will access), and then the application bandwidth will be calculated from this estimated I/O time.

In the periodic applications studied in Section 4, we observed that in some scenarios the phases present a higher variability than an equivalent benchmark, with higher performance due to the phases being executed one after the other. That limits the effectivity of our approach, unless we were to detect periodic phases (which is possible with the obtained information from traces) and then design periodic benchmarks to estimate their performance. Nonetheless, the main goals of using benchmarks in this case are to use the same results for multiple applications and to decrease profiling time (because the benchmark will have shorter execution than the whole application). This type of specification would go against our goals. Despite the fact that the benchmark underestimated performance in these situations, it still produced a curve that allowed for the identification of the best number of I/O nodes.

Among some results for the HACC-IO application, notably the ones presented in Table 9, we observed an "extreme" variability, with results apparently divided into two groups: one of much higher bandwidth than the other. For such scenarios, the arithmetic mean seems inappropriate to represent and compare their performance. As future work, we plan on exploring other options to represent (and estimate) performance distributions.

Another curious observation from the Case Study III was that the error metric, used in this report, indicated that the second benchmark was worse for estimation of application performance than the first, whereas the observation of the produced curves indicates the contrary. In fact, for future work we plan on finding and using a metric that quantifies how well the "curve" is estimated. That will be important for an eventual allocation policy that uses these estimations because it must be able to identify the best number of I/O nodes for each application.

Finally, in the estimations discussed in the case studies, we used benchmarks matching the application I/O phase characteristics. However, in a real-life application, it would be more efficient and practical to use approximate values for request size and amount of data. For instance, we could try representing all data amounts between 1 and 10 GB by one of these extremes, or by 5 GB. Nonetheless, finding these values is not an easy task because performance is very sensitive to the amount of accessed data (and in some cases to the request size), specially when dealing with small sizes. That happens because the amount of data impacts the success of caching and prefetching techniques. As a future work, we plan on exploring this idea.

7 Conclusion

In this report, we test the hypothesis that we can estimate the performance of an application as a function of the number of used I/O nodes by using benchmarks that mimic its I/O phases, with these phases being estimated from a coarse-grained aggregated trace. Such an estimation would be useful to be used by allocation strategies that assign I/O nodes to applications seeking to maximize performance (individual or global).

To test our hypothesis, we developed the SummarIO tool to estimate I/O phases from Darshan aggregated traces, and proposed a technique to match the description of an I/O phase to a corresponding benchmark. We applied our approach to five applications (four synthetic and a real one) over three case studies of increasing complexity. We found our technique resulted in a mean error of approximately 20% when using the benchmark performance to estimate it for the application.

This result indicate the approach is promising, with some improvements expected as future work. The observed error means that, when proposing an allocation policy that uses this approach for application performance estimation, it is important to make it robust to a certain degree of estimation error.

In addition to the future work listed in Section 6, we would like to expand our analysis to more applications, with more different behaviors, to enrich our conclusions. Furthermore, most of the observed drawbacks of our proposal come from limitations in the information available from Darshan traces. Therefore, we would like to look for other monitoring tools to obtain the desired metrics, or possibly propose a new one.

All results presented in this report, as well as code used to generate and analyze them, are available at the companion repository: `https://gitlab.inria.fr/hpc_io/iofwd_perf_impact`.

Acknowledgements

Experiments presented in this paper were carried out using the Gridâ5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

[1] J. L. Bez, A. R. Carneiro, P. J. Pavan, V. S. Girelli, F. Z. Boito, B. A. Fagundes, C. Osthoff, P. L. da Silva Dias, J.-F. Méhaut, and P. O. Navaux. I/o performance of the santos dumont supercomputer. *The International Journal of High Performance Computing Applications*, 34(2):227–245, 2020.

[2] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, page 8 pp., Seattle, WA, USA, 2005. IEEE.

[3] 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. *ACM Transactions on Storage*, 7(3):1–26, oct 2011.

[4] 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)*, pages 265–279, Boston, MA, 2019. USENIX Association.

[5] G. K. Lockwood, W. Yoo, S. Byna, N. J. Wright, S. Snyder, K. Harms, Z. Nault, and P. Carns. UMAMI: A recipe for generating meaningful metrics through holistic I/O performance analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage Data Intensive Scalable Computing Systems - PDSW-DISCS '17*, pages 55–60, New York, New York, USA, 2017. ACM Press.

[6] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa. Optimization techniques at the I/O forwarding layer. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 312–321, 2010.

[7] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. Tiwari, S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, A. S. Bland, O. Ridge, L. Computing, and O. Ridge. Best Practices and Lessons Learned from Deploying and Operating Large-Scale Data-Centric Parallel File Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis on (SC)*, 2014.

[8] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii. Accelerating I / O Forwarding in IBM Blue Gene / P Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis on (SC)*, number November, 2010.

[9] B. Xie, Z. Tan, P. Carns, J. Chase, K. Harms, J. Lofstead, S. Oral, S. S. Vazhkudai, and F. Wang. Applying Machine Learning to Understand Write Performance of Large-scale Parallel Filesystems. In *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*, pages 30–39, 2019.

[10] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, J. Xing, and Y. Yuan. Hybrid hierarchy storage system in MilkyWay-2 supercomputer. *Frontiers of Computer Science*, 8:367–377, 2014.

[11] 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 2019*, pages 379–394, Boston, MA, USA, 2019. USENIX Association.

[12] J. Yu, G. Liu, W. Dong, X. Li, J. Zhang, and F. Sun. On the load imbalance problem of I/O forwarding layer in HPC systems. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, volume 2018-Janua, pages 2424–2428. IEEE, dec 2017.

Appendix A — Estimation of I/O phases from Darshan traces

This section describes the process of estimating an application's I/O phases from a Darshan trace. For all case studies presented in this report, Darshan version 3.2.4 was used. We developed the SummarIO tool, which is available at `https://gitlab.inria.fr/hpc_io/summario`.

Counters in a trace are separated by API (POSIX, MPI-IO, etc), accessed file and rank that generated the access. In some situations, instead of a single rank the value -1 means

that all ranks participated in that access. The set of counters presented to each (handle, rank) pair depends on the API (for instance, MPI-IO include counters of collective operations, while POSIX counters allow for the detection of spatiality), but they always include a number of bytes read/written and timestamps for the first and last accesses. The SummarIO tool receives such Darshan traces as input and produces an output csv file containing a set of I/O phases with the set of fields listed in Section 2. This generation happens in three steps, listed below and detailed in the following.

1. The counters for each present (file handle, API, operation, rank) combination are read and parsed. Each such combination will result in a single phase which will be either to a shared-file (with all ranks being involved) or to a single file by a single rank.

2. If phases for the same (file handle, API, operation) combination but with different ranks are found, they are combined to generate a single phase. The goal of this step if to merge shared-file phases that were not reported as such by Darshan. That happens notably when multiple ranks, but not all of them, share a file.

3. The whole set of phases is then sorted by start timestamp. Phases with similar characteristics that happen close in time are combined into a single one. The first goal of this step is identifying file-per-process phases, which will be reported once to each involved rank. A secondary goal is to gather similar consecutive phases (for instance if all ranks access two shared-files) to present a more concise view of the application activity.

Step 1 — parsing the counters

For the initial set of phases most information is obtained directly from the counters, such as amount of data, number of requests, start, and end. At this point, a phase can either be a shared-file phase where all ranks are involved, or a file-per-process phase of a single process.

For POSIX phases, the counters provide a number of *consecutive* and *sequential* accesses, which can be compared to the total number of accesses. Consecutive means the start offset of the request in the file is the same as the end offset of the previous request, i.e. consecutive requests are contiguous. On the other hand, sequential requests have increasing start offsets, meaning the file is accessed from beginning to end but not contiguously.

To decide the phase spatiality, we use a configurable *threshold* parameter, which defines the percentage of the requests inside the phase that can differ in classification from the others. In other words, $1 - threshold$ gives how many of the requests must be consecutive/sequential for the whole phase to be considered consecutive/sequential. If the phase cannot be classified as consecutive or sequential, it is considered *random*. A similar strategy is used for MPI-IO phases to decide between *collective* or *unknown* spatiality. For the results discussed in this report, *threshold* was set to 0.2.

Step 2 — combining similar phases from different ranks to the same file

After having parsed the counters to obtain a single phase for each present (file handle, API, operation, rank) combination, phases that differ only by rank are combined in this second step. Such "duplicates" will only happen for phases corresponding to single-rank accesses.

- The generated phase starts at the shortest start timestamp and ends and the highest end timestamp.

- The number of requests is the sum of the requests generated in the sub-phases, similarly to the amount of data.

- The number of involved ranks is calculated by creating a set of the ranks involved in the combined phases.
- The number of requests per type is calculated to re-estimate the spatiality of the phase in the same way as discussed in Step 1.
- The I/O time of the phase is the longest among the different phases' duration. In other words, we assume they correspond to a parallel phase.

Step 3 — combining similar phases to different files

We traverse the list of I/O phases, ordered by start timestamp, and compare each phase to all the ones that start up to *time_ distance* after its end, where the amount of *time_ distance* is a configurable parameter of the tool. If two phases happen this close in time and are deemed "compatible", the end timestamp is updated (with the max between the two) to continue searching for more phases. Phases are considered compatible if they use the same API, operation (read or write), spatiality, and file strategy (shared or unique). For the results presented in this report, the used *time_ distance* parameter was of 1 second.

After traversing the list, we will possibly have identified sets of phases that should be combined. That will be done following the same approach as described in Step 2.

This approach works to identify file-per-process phases. However, a limitation is that when other types of similar consecutive phases are aggregated (for instance if an application accesses multiple shared files one after the other), the I/O time estimation will be wrong, because we take the maximum I/O time among the combined phases (and in the second case the phases are not parallel, we should sum their duration). This is a limitation of using the coarse-grained aggregated traces, because there is no other way of identifying a file-per-process phase, even if this is one of the most popular access patterns among HPC applications [11]. We sacrifice therefore the precision on the phase estimation duration in exchange for being able to identify these file-per-process phases.

Appendix B — Experimental methodology

This section details the experimental methodology used for all presented results. All tests were executed in the Nancy site of the Grid'5000 platform [2], using clusters Grimoire and Gros[4].

Three nodes from Grimoire were used as Lustre servers (an MDS and two OSS+OST). This cluster has 8 nodes, each of them powered by two eight-core Intel Xeon E5-2630 v3 and 128 GB of RAM memory. For storage, the MDS used a 200 GB SSD SCSI Toshiba PX02SSF020 SSD, and the OSTs used a 600 GB HDD SCSI Seagate ST600MM0088 HDD each. Up to Gros nodes were used as compute and I/O nodes. It has 124 nodes, each containing an 18-core Intel Xeon Gold 5220 and 96 GB of RAM. Gros nodes are interconnected by 25 Gbps Ethernet links to two switches, which are connected to the Grimoire switch by two 40 Gbps links each. The third switch is connected to Grimoire nodes by four 10 Gbps links.

Lustre version 2.12.5 was used. Servers run a CentOS 7.7.1908 , and clients a Centos 8. The file system was installed with default parameters, and then configured to stripe files across the two data servers, with stripe size of 1 MB. All contents of the file system were erased between consecutive experiments.

[4] `https://www.grid5000.fr/w/Nancy:Hardware`

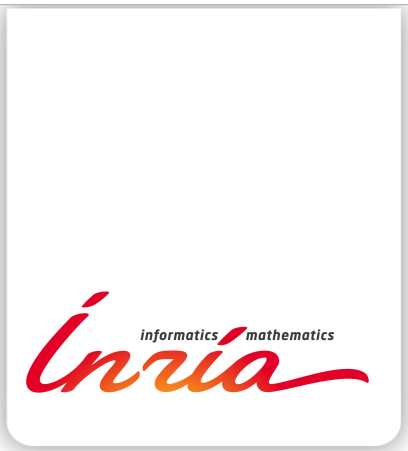
To emulate the use of I/O forwarding in this setup, we dedicated up to 8 Gros nodes to work as I/O nodes, and used GekkoFWD[5] to intercept applications' I/O requests and forward them to these I/O nodes, which used the locally mounted Lustre file system for storage. Applications were equally distributed among the I/O nodes, and this setup was completely reinstalled before each experiment.

Experiments that were executed with the goal of obtaining a Darshan trace used version 3.2.4, and other experiments (with the goal of measuring performance) did not use Darshan.

The whole set of experiments (five repetitions of each) was executed in random order over multiple days. That was done to minimize the impact of temporal system slow-downs and particular testing orders in our results. Moreover, the whole clusters were reserved for the experiments, even when not all nodes were used, to avoid network interference from other applications.

For each experiment, the bandwidth is calculated by dividing the total amount of data accessed by the application by the parallel I/O time (the time of the slowest process). All results and scripts used to generate and analyze them are available at the companion repository: `https://gitlab.inria.fr/hpc_io/iofwd_perf_impact`.

[5] https://jeanbez.gitlab.io/forwarding-arbitration/



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399