IO-SETS: Simple and efficient approaches for I/O bandwidth management

Francieli Boito, Guillaume Pallez, Luan Teylo, Nicolas Vidal

IO-SETS: Simple and efficient approaches for I/O bandwidth management

Francieli Boito, Guillaume Pallez, Luan Teylo and Nicolas Vidal*

***Abstract*—One of the main performance issues faced by high-performance computing platforms is the congestion caused by concurrent I/O from applications. When this happens, the platform's overall performance and utilization are harmed. From the extensive work in this field, I/O scheduling is the essential solution to this problem. The main drawback of current techniques is the amount of information needed about applications, which compromises their applicability. In this paper, we propose a novel method for I/O management, IO-SETS. We present its potential through a scheduling heuristic called SET-10, which is simple and requires only minimal information. Our extensive experimental campaign shows the importance of IO-SETS and the robustness of SET-10 under various workloads. In particular in most of the simulated scenarios we improve the I/O slowdown over fairshare by 50%, which corresponds in our scenarios to a platform utilization gain of 2.5%. In the practical scenarios that we did, the utilization gain varies between 10 and 30%. We also provide insights on using our proposal in practice.**

***Index Terms*—I/O Scheduling, Bandwidth Management, I/O performance**

I. INTRODUCTION

As high-performance applications increasingly rely on data, the stress put on the I/O system has been one of the key bottlenecks of supercomputers. Indeed, processing speed has increased at a faster pace than parallel file system (PFS) bandwidth. For example, the first supercomputer in the Top500 list [1] of November 1999 was ASCI Red, with peak performance of 3.2 TFlops and 1 GB/s of PFS bandwidth [2]. Twenty years later, in November 2019, Summit had 200.8 PFlops (over $62,000\times$ faster) and a PFS capable of 2.5 TB/s ($2,500\times$ faster) [3].

Many solutions have been proposed to face the I/O bottleneck. For example, we can trade-off storage for computation: compression will reduce the volume of I/O sent to the PFS at the cost of extra operations [4]. Data staging and in-situ techniques also decrease the amount of data sent to the file system, by using extra cores/nodes (which could instead be used to speed-up computation) to process data while it is generated [5]. Still, either because of limited local storage capacity or because data is meant to be persistent, I/O to the parallel file system cannot be completely avoided.

The I/O bottleneck becomes a bigger issue when multiple applications access the I/O infrastructure simultaneously. Congestion delays their execution, thus they hold compute resources for longer, hurting the utilization of the platform. Moreover, performance variability increases, since the application's execution time depends on the current state of the machine. In this context, solutions were proposed to try to manage the accesses to the shared I/O system. The main way of doing so is I/O scheduling [6], [7], [8], [9]: it consists in deciding algorithmically which application(s) get priority in this access.

We focus on I/O scheduling strategies. Precisely, we consider the scenario where applications compete for the PFS's bandwidth. Various approaches were proposed.

Clairvoyant approaches [10], [7], [11] consider the I/O patterns of each application are known before-hand. In this case, one can compute a schedule that optimizes the right objective functions (often maximum system utilization or a fairness objective between applications). This task can be expensive because applications can have a large number of I/O phases. Moreover, these techniques require information that in practice is often not accurate or even accessible.

Non-clairvoyant schedulers do not know when applications will perform I/O. These accesses are thus simply scheduled given a priority order, the most common being first-come-first-served, or semi round-robin (serve the application that performed I/O the least recently) [12], [10].

In this context, **our goal is to design an I/O scheduling solution that uses some information about applications. Nevertheless, we want to use as little as possible, and our technique should be robust to inaccuracies in this information. Furthermore, we want this solution to be simple in implementation and not intensive in computation**. We believe this makes an important improvement over the state

*Authors in alphabetical order. All parts of the paper were discussed between most authors throughout the work, particularly the design of the evaluation, analysis of the results and writing of the manuscript but some authors had more important contributions in some of the specific tasks. FB: prototype and experiments, supervision. GP: conceptualization, methodology, supervision. LT: simulator, data vizualization, Plafrim trace analysis. NV: methodology.

FB and LT are with Univ. Bordeaux, CNRS, Bordeaux INP, INRIA and LaBRI. GP is with Inria. NV is with Oak Ridge National Laboratory.

e-mail: {francieli.zanon-boito, guillaume.pallez, luan.teylo}@inria.fr, vidaln@ornl.gov

Manuscript received ...; revised ... The authors would like to thank Emmanuel Jeannot and Arnaud Legrand for useful discussions. Experiments presented in this paper 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). 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).

of the art because it can actually be used in practical scenarios, where only limited information is available and more sophisticated algorithms would impose too much overhead.

In all of the approaches discussed above, I/O operations are performed exclusively (i.e. one application at a time), or semi-exclusively when applications cannot use all the available bandwidth by themselves (i.e. some applications run concurrently using as much bandwidth as they can). An alternative is to use *fair-sharing*: the bandwidth is shared equally by all applications in a best-effort fashion, hence when two perform I/O concurrently, each one takes twice the time it would take by itself.

We propose a novel approach called IO-SETS, built on the intuition that both exclusive access and fair-sharing have benefits. Section II details this intuition. With this respect, we design a two-pronged I/O management approach for high performance computing (HPC) systems: first, applications are sorted into *sets*. Applications from the same set do I/O in an exclusive fashion (one at a time). Nonetheless, applications from different sets can do I/O accesses at the same time, and in this case they share the available bandwidth. The main contributions of this paper are the following:

- a novel method for I/O management in HPC;
- an instantiation of this method with simple heuristics that require little information about applications;
- a thorough evaluation of this heuristic that shows its impact and limitations even in extreme scenarios.

To assess the effectiveness of IO-SETS, we conducted evaluations using various simulated scenarios in SimGRID[13], a well-known simulator for distributed systems. In Section VI, we describe several experiments that test different aspects of our solution. Additionally, to validate our simulation results, we developed a prototype using IOR and performed practical evaluations, as detailed in Section VI-A.

The next section presents the intuition behind our method and further motivates it. In Section III, we detail the problem and our platform and application models, so that we can present our method and heuristics in Section IV. Then, in Section V, we discuss the methodology used to evaluate them. Results are presented in Section VI, followed by a discussion on how this method can be used in practice in Section VII and a discussion on its relevance in Section VIII. The paper closes with related work in Section IX and final remarks in Section X.

II. MOTIVATIONAL EXAMPLES

In this Section we present the intuitions behind our proposal. First, we discuss the advantages and drawbacks of techniques that provide exclusive and fair access to the I/O infrastructure. Then, we motivate a new bandwidth sharing approach.

A. Exclusive vs. fair access

In their work, Jeannot et al. [8, Figure 3] have discussed performance of list-scheduling heuristics for exclusive I/O access. They have compared their heuristics to the fair-share algorithm over a wide range of scenarios for I/O. Interestingly, their results indicate that sometimes fair-share behaves better than exclusive heuristics, and sometimes it is the opposite. They have not investigated this aspect further.

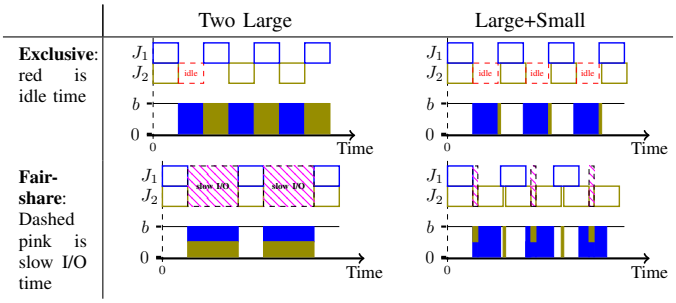


Fig. 1: Two different pairs of jobs are shown side by side, and two schedulers are shown one above the other. For each of the four parts of the figure, the top half represents activity on compute nodes, and the bottom half the I/O accesses to the PFS (the height represents the portion of the bandwidth used).

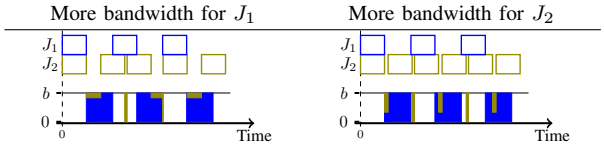


Fig. 2: The applications share the bandwidth, but J_2 (the one with small I/O phases) receives more or less of it than J_1.

We illustrate this with a simple example. Consider two applications, (i) one that has relatively large I/O phases (we call it *large*), which when running in isolation performs periodically: computation during one unit of time and I/O during one unit of time; and (ii) a *small* one, which in isolation periodically does computation during one unit of time and I/O during 0.01 unit of time. We consider two scenarios in Figure 1: two *large* applications competing for the I/O bandwidth, and a *large* with a *small*. In the following, we assume accesses, once started, cannot be interrupted, and that, at the beginning of the first I/O phase, the *large* application's request arrived before. Furthermore, in this paper, we use "application", "task", and "job" without distinction.

As one can see from this figure, fair-sharing can be inefficient: for the TWO LARGE scenario it takes three units of time to perform the work that exclusive does in roughly two units of time (after initialization). However, the opposite can also be true: in the LARGE+SMALL scenario, exclusive takes roughly two units of time to perform the computation of the small application when fair-sharing can do it in roughly one, with almost no extra cost for the *large* application.

B. Bandwidth sharing

We argue that sharing the bandwidth does not have to be done fairly. In Figure 2, we consider the two application profiles from the previous section, *small* and *large*. If the small I/O phase finishes before the large one, then giving it more of the available bandwidth improves *locally* (i.e. for this phase) the performance of *small* without delaying the *large* one.

Of course, improving the performance of the small application may *globally* delay the large one, however the impact

seems negligible. For instance, in Figure 2 (on the right), at every two iterations of J_1, J_2 performs an additional I/O phase. Hence J_1's I/O phase becomes 1% longer. Over its execution, this is a slow-down of less than 0.25%. By comparison, when J_1 receives most of the bandwidth, every three iterations, one *small* I/O phase takes 0.9 units of time instead of 0.01. In this case, three iterations of J_2 take 4.1 units of time instead of 3.2, i.e. a slow-down to the *small* application of roughly 28%.

In the study presented in this paper, we try to confirm experimentally this intuition that encourages assigning more bandwidth to applications with relatively small I/O phases.

III. I/O scheduling in HPC systems

This section describes the platform and application models we use in this study. We also introduce the notion of *characteristic time* (w_{iter}) to represent applications.

A. Platform model

We assume a parallel platform with a separated I/O infrastructure that is shared by all nodes and that provides a total bandwidth B. There are many ways to share the I/O bandwidth between concurrent I/O accesses [10]. *One of the novel ideas that we develop here is to manage I/O bandwidth using a priority-based approach*, which was inspired by a network protocol [14]. It consists in assigning priorities to I/O accesses.

The idea is that, when an application performs an I/O phase by itself, it can use the full bandwidth of the system. However, when there are k concurrent requests for I/O, with respective priorities $p_1, p_2, \dots, p_k$, then for $i \in \{1, \dots, k\}$, the i^{th} request is allocated a share x_i of the total bandwidth such that:

$$x_i = \frac{p_i}{\sum_{j=1}^{k} p_j}. \tag{1}$$

By recomputing priorities at each time step, one could model any bandwidth partitioning strategy. However, here we impose that the priority (not the bandwidth share) for each application must be computed independently from others.

B. Applications' I/O behavior

HPC applications have been observed to alternate between compute and I/O phases [10], [15]. This burstiness of I/O is one of the motivations for I/O scheduling.

Definition 1 (Job model)**.** There are N jobs. For $j \leq N$, job J_j consists of n_j non-overlapping iterations. For $i \leq n_j$, iteration i is composed of a computing phase of length $t_{\text{cpu}}^{(i)}$ followed by an I/O phase of length $t_{\text{io}}^{(i)}$. These lengths (in time units) correspond to when the job is run in isolation on the platform (i.e. when there is no interference from other jobs).

For simplicity, in the following and if there is no ambiguity, we relax some of the indexes and use n for the number of iterations, and t_{cpu} and t_{io} for the length of a compute or I/O phase. We consider here that an I/O operation that would take t_{io} units of time in isolation takes t_{io}/x units of time when using only a share x of the I/O bandwidth. For instance, if the bandwidth is equally split between two operations ($x = 0.5$), they take twice the time they would by themselves.

The precise I/O profile of an application, i.e. all its I/O accesses along with their volumes, start and finish time, can be extremely hard to predict. Obtaining this would involve a detailed I/O profiler that would impose a heavy overhead and generate a large amount of data, and still the profile would suffer from inaccuracy. I/O profilers used in practice focus on average or cumulative data. For instance, Darshan [16] is able to measure the total volume of I/O accessed by an application. In this work, we focus on average parameters:

Definition 2 (Characteristic time, w_{iter})**.** The *characteristic time*, w_{iter}, of an application with n iterations:

$$w_{\text{iter}} \stackrel{\text{def}}{=} \frac{\sum_{i \leq n} t_{\text{cpu}}^{(i)} + t_{\text{io}}^{(i)}}{n}.$$

This characteristic time can be seen as the average time between the beginning of two consecutive I/O phases. In the example from Section II, the characteristic time w_{iter} is 2 units of time for the large application and 1.01 for the small one.

We also define a job's average portion of time spent on I/O:

Definition 3 (I/O ratio, α_{io})**.** The I/O ratio of a job is:

$$\alpha_{\text{io}} = \frac{\sum_{i \leq n} t_{\text{io}}^{(i)}}{\sum_{i \leq n} t_{\text{cpu}}^{(i)} + t_{\text{io}}^{(i)}}$$

Using this, we can define the average *I/O stress* of the platform with N applications at a given time:

$$\omega = \sum_{j \leq N} \alpha_{\text{io}}^{(j)} \tag{2}$$

The intuition behind this value is that it corresponds roughly to the expected average occupation of the I/O bandwidth. We can make the following important remarks:

- These average values do not depend on the number of used nodes or their performance. In practice, an application that runs on one node of 1 GFlops for 20 minutes and performs 20 iterations has the same w_{iter} value (1 minute) than another that runs on 2000 nodes at 2 TFlops for 120 minutes and performs 120 iterations.
- The characteristic time and I/O ratio as defined above are average values that we expect could be evaluated easily, or approximated from previous runs of an application. Due to being averages, they are more robust to variability than individual phases (Ergodic Theorem).

IV. IO-Sets: a novel I/O management method for HPC systems

Motivated by the examples in Section II, *we propose the* IO-Sets *method, which allows exclusive access for some applications, and (not fair) bandwidth sharing for others*.

A. High-level presentation of IO-Sets

Our proposition is a *set-based approach*, described below.

- when an application wants to do I/O, it is assigned to a set $S_i \in \{S_0, S_1, \dots\}$.
- Each set S_i is allocated a bandwidth priority p_i.

- At any time, only one application per set is allowed to do I/O (exclusive access within sets). We use the first-come-first-served (FCFS) scheduling strategy within a set (i.e. we pick the application that requested it the earliest).
- If applications from multiple sets request I/O, they are allowed to proceed and their share of the bandwidth is computed using the sets' priorities and Equation (1).

Proposing a heuristic in the IO-SETS method consists therefore of answering two important questions: (i) how do we choose the set in which an application is allocated, and (ii) how do we define the priority of a set.

B. Reference strategies

We can illustrate the instantiation of the IO-SETS method by using it to represent the two discussed reference strategies.

EXCLUSIVE-FCFS*:* for this heuristic, all applications have exclusive access and are scheduled in a FCFS fashion. This is the case where there is a single set S_1 with any priority (for instance $p_1 = 1$). All applications are assigned to S_1.

FAIR-SHARE*:* bandwidth is shared equally among all applications requesting I/O access. This can be modeled by having one set per application ($S_0, \dots, S_k$), all with the same priority $p_i = 1$. Application id is scheduled in set S_{id}.

C. SET-10 *algorithm*

In the motivational example from Figure 1, when two jobs have the same characteristic time, it seems more efficient to provide them with exclusive I/O access so that they can synchronize their phases. In this case, one of them would pay a delay equal to the length of the other's I/O phase once, but then for the remaining iterations, neither of them would be delayed. However, as discussed in Section II, exclusive access does not bring benefits when the applications' characteristic times are very different. Based on this observation, we propose the SET-10 heuristic, which builds the sets depending on w_{iter}.

Set mapping: Given an application A_{id}, with a characteristic time $w_{\text{iter}}^{\text{id}}$, then the mapping allocation is π:

$$\pi : A_{\text{id}} \mapsto S_{\lfloor \log_{10} w_{\text{iter}}^{\text{id}} \rceil} \tag{3}$$

where $\lfloor x \rceil$ defines the nearest integer value of x.

In other words, an application is assigned to a set that corresponds to its w_{iter} magnitude order: there will be a set for applications with w_{iter} so that $\log_{10} w_{\text{iter}}$ is between 0.5 and 1.5 (w_{iter} between 4 and 31 go into S_1), another for $\log_{10} w_{\text{iter}}$ between 1.5 and 2.5 (w_{iter} between 32 and 316 go into S_2), and so on.

Set priority: To define priorities for the sets, we start with the following observation: if two applications start performing I/O at the same time and share the bandwidth equally, the one with the smallest I/O volume finishes first. Now if we increase the bandwidth of the smallest, it will finish earlier, but the largest one will not be delayed (see Figure 2).

Based on this, we want to provide higher bandwidth to the sets with the smallest I/O accesses. Intuitively, if the characteristic time w_{iter} are of different orders of magnitude, we assume that so are the I/O accesses. An advantage of using w_{iter} instead of the I/O volume is that it is easier to obtain, as previously discussed. Hence we define the priority p_i of set S_i (which corresponds to jobs such that $i = \lfloor \log_{10} w_{\text{iter}}^{\text{id}} \rceil$):

$$p_i = 10^{-i} \tag{4}$$

That means the applications with the smallest w_{iter} receive the highest priority, and therefore most of the bandwidth. Priority decreases exponentially: S_1 receives priority $1/10$, S_2 has $1/100$, and so on.

Definition 4 (SET-10)**.** We define SET-10 the strategy in the IO-SETS method consisting of:

- The list of sets with their priorities $\{\dots, (S_{-1}, 10), (S_0, 1), (S_1, 0.1), (S_2, 0.01), \dots\}$ (where the priorities are computed with Equation (4));
- The π function that maps jobs to sets using Equation (3).

D. Other Set-based heuristics

In order to show the importance of both the set mapping — with the π heuristic — and the priority assigning parts of SET-10, we add two heuristics to our evaluation. In the first one, SET-FAIRSHARE, we use the π mapping strategy with fair sharing of the bandwidth, i.e. for all i, $p_i = 1$ (all sets have the same priority). In the second one, SHARE+PRIORITY, we use the priority heuristic from SET-10 (Equation 4), but each application is in its own individual set (just like FAIR-SHARE), so they can all perform I/O concurrently.

These two heuristics allow us to show that it is important to have a combination of a good priority heuristic with a good set mapping heuristic.

V. EVALUATION METHODOLOGY

The evaluation in this paper represents a first exploratory work for the IO-SETS method. For that, we chose to use simulation because we wanted to test many scenarios, each of hundreds of different workloads trying to cover a maximum of combinations of behaviors. Such experiments would consume a lot of time and resources (at least 569 days). Moreover, test platforms are limited in size, but deploying I/O schedulers in a production system (just to test them) is complex and usually not allowed. Besides the convenience, a simulator brings extra advantages such as reproducibility and scalability. Nevertheless, we also conducted some non-simulated experiments in order to validate our simulation methodology. Later, in Section VII, we discuss the practical applicability of our method.

In Section V-A, we describe Simgrid, the tool used to simulate our environment. We then discuss the workload generation (Section V-B), the way the IO-SETS method was implemented in Simgrid (Section V-C), and finally the performance metrics used to compare the scheduling strategies (Section V-D). All code used for our evaluation is available and documented at https://gitlab.inria.fr/hpc_io/io-sets.

A. Simulating I/O with Simgrid

We use Simgrid v3.30 [13], an open-source simulator of distributed environments that also simulates I/O operations [17]. Those are defined by a volume and the storage device where they will be performed. Each device has a speed and is associated with one host, it can be accessed either from the host (locally) or from other machines through the network (shared access). In both cases, the simulator adopts a coarse-grain model that sees I/O operations as a unique flow under a bandwidth (or bandwidths for the shared access).

Simgrid's I/O model has the following assumptions: i) time increases linearly with the amount of data; and ii) there is no latency associated with the operation. This model has been validated [17], and Simgrid is a well-established tool used in several projects and publications [14].

B. Workload generation protocol

Here we describe the way jobs are created. The instantiation of the introduced variables (μ, σ, b) is done on a per-experiment basis and described in Section VI.

We are interested in a steady state evaluation of the system, and hence consider that all applications execute for a time that is long enough (a horizon h, in practice we set $h = 20,000$s). Each workload was executed considering two scenarios for the target average I/O stress ω (Equation 2): a saturated system ($\omega = 0.8$) and a light I/O stress ($\omega = 0.2$).

As discussed in Definition 1, a job is fully described by its number of iterations n, and compute and I/O phases' length. In order to generate each job J_j, we proceed as detailed below.

1) We decide on a characteristic time w_{iter} for the job. In practice, it is drawn from the normal distribution $\mathcal{N}(\mu, \sigma)$, truncated so that we consider only positive values. μ and σ are selected on an experiment basis.
2) The number of iterations of the job is $n = \frac{h}{w_{\text{iter}}}$.
3) To avoid having all applications synchronized at $t = 0$, which would be detrimental to FAIR-SHARE, we draw a release time Γ in $\mathcal{U}[0, w_{\text{iter}}]$. Γ may be seen as an amount of work left from a previous iteration.
4) α_{io} is defined as follows: for each application, we draw a value α_j uniformly at random in $\mathcal{U}[0, 1]$, then we set $\alpha_{\text{io}} = \frac{\omega}{\sum_{k \leq N} \alpha_k} \cdot \alpha_j$ to guarantee an I/O load of ω.

α_{io} allows us to define the average values t_{cpu} and t_{io}:

$$t_{\text{cpu}} = (1 - \alpha_{\text{io}}) \cdot w_{\text{iter}} \quad \text{and} \quad t_{\text{io}} = \alpha_{\text{io}} \cdot w_{\text{iter}}$$

We introduce a noise parameter, within a range of bound b, to represent machine variability (both for processing and I/O performance). For all $i \leq n$, we draw two variables: $\gamma_{\text{cpu}}^{(i)}$ and $\gamma_{\text{io}}^{(i)}$ from a uniform distribution: $\mathcal{U}[-b, b]$.

$$t_{\text{cpu}}^{(i)} = (1 + \gamma_{\text{cpu}}^{(i)}) \cdot t_{\text{cpu}} \quad \text{and} \quad t_{\text{io}}^{(i)} = (1 + \gamma_{\text{io}}^{(i)}) \cdot t_{\text{io}}$$

The parameter γ makes our experiments more realistic and allows us to cover variations in application behavior.

Remark: In Simgrid, applications are not defined by their time, but rather by the amount of work (in GFlop for compute and GB for I/O) and performance (in GFlops and GB/s). Therefore, in the implementation, we say that each resource used by the applications is capable of 1 GFlops, and that the amount of computation they must perform is $t_{\text{cpu}} \cdot 1$ GFlops. Similarly, we set the I/O bandwidth of the platform to 1 GB/s, and the amount of data they must access to $t_{\text{io}} \cdot 1$ GB/s.

For the same reason, we do not need to formally select the number of cores used by each application because we would get an equivalent behavior: increasing computing performance would simply mean increasing the amount of work they must do to keep the same t_{cpu}. Since we focus on I/O, we can use a single resource to represent any parallelism.

C. Implementing IO-SETS in Simgrid

A simulation in Simgrid has three main components: *hosts*, which have CPUs, disks, and network connections; *activities*, which define communication, processing, read and write operations; and *actors*, user-defined functions that determine when the activities will be executed in the hosts. To evaluate the scheduling strategies considered in this work, each job has two additional parameters: their set identifier, and their priority.

We implemented our method as a Simgrid actor that manages the execution of the jobs. It maintains a set of jobs *performing* I/O, and a queue of jobs *waiting* for access. When a new job requests access, the scheduler checks whether a job from the same set is performing I/O. If there is one, then the new job is put at the end of the *waiting* queue, otherwise it is put in the *performing* set. When an I/O phase finishes, then the scheduler checks the first job from the *waiting* queue that belongs to the same set and moves it to the *performing* set.

Priority-based bandwidth sharing between concurrent I/O jobs is already implemented as a network function in Simgrid [14], and satisfies Equation (1). We have used this to simulate the I/O bandwidth sharing behavior.

D. Measuring performance

When evaluating an I/O system, we need to consider the global performance of the HPC machine. From several possible metrics [10], [15], two usual ones are: i) system utilization: how much does the system compute per time unit, or equivalently, what is the proportion of time loss due to I/O congestion, either because the I/O is slowed down, or because an application is kept idle (Figure 1); and ii) application stretch: how long does it take for a user to get a result.

Our methodology aims to measure the performance of strategies during a steady state. Such observations can then be extrapolated and interpreted as representing the behavior of an open system. In this context, we limit our measurement to a time range encompassing only the steady state. The usual approach is to remove the beginning of the simulation (initialization time), and the end of the simulation (where some applications have left the system), see Figure 3.

That is important because the early stages are not representative in requests. Reciprocally, in the final stages, some applications have finished. At that point, resources are underused,

Fig. 3: Example of execution with release times for the applications (FAIR-SHARE algorithm). The green lines represent the time frame considered to compute the metrics. CPU phases are depicted in blue, and I/O phases in orange.

and further measurements represent the workload exhaustion instead of meaningful performance metrics.

To do so, we follow the usual approach of calculating our metrics within a time frame $[T_{begin}, T_{end}]$ with $T_{begin} > 0$ and $T_{end} < h$. Specifically, with a horizon $h = 20,000$s, we select: $T_{begin} = 6,000s$ and $T_{end} = 14,000s$. Additionally, for each task J_j, we define the effective completed work $e_j = e_j^{cpu} + e_j^{io}$ as the sum of the lengths of all compute and I/O phases executed by this task within the interval, and the effective iterations e_j^{iter} as the number of iterations performed by a task inside the time frame.

`Utilization`: System administrators aim to maximize the amount of work executed on their platform. From their point of view, the most meaningful metric is utilization. We define the (cpu) utilization as the time spent doing useful computation divided by the total time normalized by the number of applications.

$$\texttt{Utilization} = \frac{\sum_j e_j^{cpu}}{N.(T_{end} - T_{begin})}$$

It is a system-wide metric that represents platform usage. It has an upper bound given by $1 - \omega/N$, which would mean that jobs are always using efficiently the compute resources. In other words, the upper bound for utilization comes from the situation where applications never wait for I/O and always use the full bandwidth when doing I/O. In that case, the compute resources will be used all the time except when doing I/O: ω/N gives the average time spent by applications on I/O according to Equation 2.

`IO-slowdown` and `Max Stretch`: From the users' point of view, the most important factor is the speed of their applications. Thus, we define two metrics considering I/O performance and execution time. First, `IO-slowdown` is the geometric mean, over all applications, of the ratio between the time each application spent either waiting for or doing I/O and the time it would take if doing I/O in isolation (using all available bandwidth). We also define the `Max Stretch` as the maximum, over all applications, of the ratio between the elapsed time (time frame size) and the effective work.

$$\texttt{IO-slowdown} = \prod_j \left(\frac{T_{end} - T_{begin} - e_j^{cpu}}{T_{io}.e_j^{iter}} \right)^{\frac{1}{N}}$$

$$\texttt{Max Stretch} = \max_j \frac{T_{end} - T_{begin}}{e_j}$$

These are application-centered metrics. The `Max Stretch` represents the maximum slow-down of an application, and the `IO-slowdown` represents how close from the minimum the actual I/O time is. In both cases, the lower, the better, with 1 meaning that all applications run at the same speed as they would in isolation.

A comprehensive summary of all the variables presented in this section can be found in the Web Supplementary Material [18, Section A].

VI. EVALUATION AND RESULTS

In this section, we evaluate the IO-SETS method. We start by comparing results obtained with simulation and on a real system (Section VI-A), with the goal of validating our simulation strategy. Then, in Section VI-B, we present results from simulated scenarios that are all challenging and progressively unfavorable to IO-SETS, with the goal of studying its limits and robustness.

A. Validating the simulation approach

To validate our simulation strategy, we compare it to an experiment conducted on a real system. For this proof-of-concept implementation, we wrote a centralized scheduler to implement SET-10 and modified the IOR benchmark [19] to communicate with it before and after each I/O phase.

In this scenario, a total of $N = 16$ concurrent IOR instances are created, distributed on two sets: n_H high-frequency applications of $w_{\text{iter}} = 64s$ and $N - n_H$ low-frequency ones of $w_{\text{iter}} = 640s$. All of them have $\alpha_{\text{io}} = 1/16$. The w_{iter} and α_{io} values were chosen according to the total I/O load ω and so that the shortest I/O phases would still write a total of 32 GB, which was experimentally observed to be the enough to reach peak performance in the used platform [20]. The access pattern was also chosen to allow for reaching peak performance: contiguous 1 MB write requests are issued to per-process files. Applications from each set are homogeneous (which favors SET-10 over FAIR-SHARE, contrarily to the rest of the paper which tries to challenge it).

Experiments were executed five times, with a horizon $H = 6400s$ and calculating metrics between $1300s$ and $6300s$. All release times were set to 0 (in practice we observed a difference of 1 second in average between the first and last application to start in each test), and the noise parameter is also set to $b = 0$ since the machine-induced variability is already directly measured.

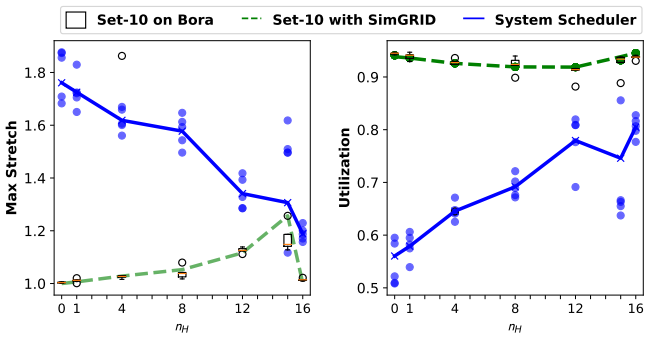


Fig. 4: Comparison of practical (box and whisker) and simulated (dashed lines) executions with SET-10. The blue line and points shows baseline executions without SET-10. Simulated results are deterministic. `Max Stretch` is shown on the left (lower is better) and `Utilization` on the right (higher is better). The y-axes do not start at 0.

The experiments were conducted on the Bora cluster from the PlaFRIM experimental platform. Each of its nodes has two 18-core Intel Xeon processes, 192 GB of RAM and run CentOS7.6.1810 (kernel v3.10.0-957.el7.x86 64). The PFS is BeeGFS v.7.2.3 deployed over two servers, each with four storage targets and one MDT. The servers and compute nodes are connected through a 100 Gbps Omnipath network.

We used eight nodes to run applications and a separate node for the scheduler. Data sizes, numbers of nodes and processes and their placement, and stripe count (all eight targets) were selected so each application would reach peak performance by itself, as recommended by Boito et al. [20] in their study conducted in the same platform. More specifically, each application uses all the eight nodes, with processes equally distributed across nodes.

Moreover, the priority-based bandwidth sharing was implemented by using ten times more processes for each application from one set than to each from the other set (160 vs. 16). These numbers were chosen according to the experiment presented in Figure 11, which shows both numbers of processes allow for roughly the same performance when an application is not sharing the access to the PFS. Furthermore, in the experiment discussed in Section VII (results in Figure 12), we confirm that using different numbers of processes we can assign different portions of the bandwidth for them.

As a baseline, we also evaluated a scenario without our scheduler, which we call the "system scheduler" (which would be close to FAIR-SHARE with additional optimization). For this all applications used the same number of processes (16).

The results presented in Figure 4 show the excellent precision of the simulator. 80% of the experimental results are within 3.5% (resp. 1.5%) of the simulated `Utilization` (resp. `Max Stretch`). There are two outlying scenarios:

- For $n_H = 4$, one repetition had results severely different from the others. We believe this was due to interference from other jobs on the shared platform.
- When $n_H = 15$, most practical results are slightly better than the simulation. In this case most I/O phases are small, hence we attribute this to to system optimizations such as caching. Note that this is not observable at $n_H = 16$ because the `Max Stretch` is equal to 1, leaving no room for optimizations.

Simply running the experiments used in this section necessitates over 62h of machine time (times 2 if we consider the baseline evaluation), while our simulation takes less than 5s on a laptop. The complete set of evaluations in this paper would need tremendous compute time. Overall, we believe this first result demonstrated the validity of using a simulator for the rest of the evaluation, and the difference of cost justifies it.

Finally, a comment on the performance: this scenario is favorable to SET-10 because all applications start at the same time, which hurts FAIR-SHARE-based strategies such as our baseline. This is close to the motivation in Figure 1. In the rest of this section, we use the random release time in order to create scenarios that are less favorable to SET-10.

B. *Performance analysis of* IO-SETS

After validating our simulation methodology in the previous section, we now use it to evaluate our proposal. In Section VI-B1, we consider a scenario where applications are constructed efficiently for the set mapping function of SET-10. Then, in Section VI-B2, the distributions of characteristic times are not optimized for SET-10. We represent in Figure 5 different examples of w_{iter} distributions that were used. Here we focus on results with $\omega = 0.8$, because they are the most interesting: when the I/O load in the system is very low, the impact of I/O scheduling techniques is decreased. Results with $\omega = 0.2$ are presented in the Web Supplementary Material [18, Section C].

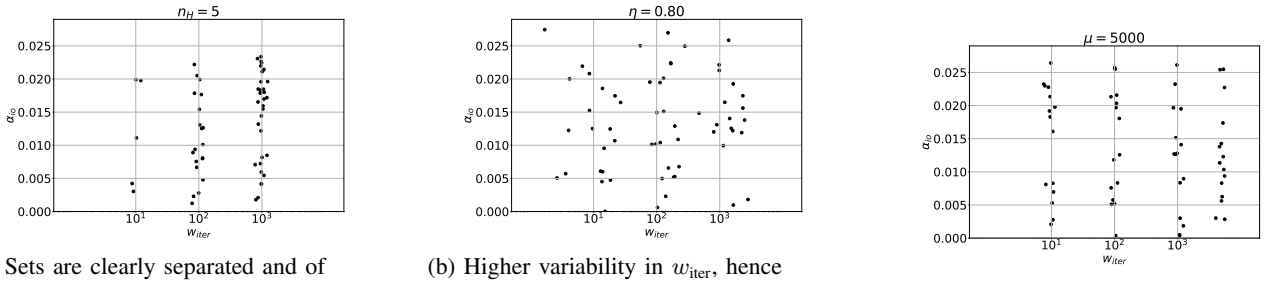
1) Sets correspond to the mapping algorithm: In this section, we aim to show that IO-SETS is indeed a relevant method when compared to the reference strategies EXCLUSIVE-FCFS and FAIR-SHARE. In order to focus on the importance of the use of sets in the IO-SETS method, we design a collection of applications that clearly belong to different groups of SET-10. By doing that, we temporarily put aside the question of designing an efficient heuristic for IO-SETS.

In Section III, we intuited that jobs should be grouped based on their characteristic time (w_{iter}). Therefore, using the protocol presented in Section V-B, we define three different job profiles (i.e. values for μ and σ):

- n_H jobs with characteristic time $w_{\text{iter}} \sim \mathcal{N}(10, 1)$, also called high-frequency jobs in the following (the mean duration of an iteration is 10s);
- n_M medium-frequency jobs with $w_{\text{iter}} \sim \mathcal{N}(100, 10)$;
- n_L low-frequency jobs with $w_{\text{iter}} \sim \mathcal{N}(1000, 100)$;

Here we set $N = n_H + n_M + n_L = 60$. Results for $N = 15$ are presented in the Web Supplementary Material [18, Section C].

We study the impact of the algorithms when $n_M = 20$, $n_H \in \{0, \ldots, 40\}$, and $n_L = 40 - n_H$. This configuration results in a bimodal workload, with only two sets, when n_H is either 0 or 40. For all other values of n_H, we have three sets. An example for the distribution of w_{iter} when $n_H = 5$ is given in Figure 5a. In this Section, we consider $b = 0.1$. Results are in Figure 6.



(a) Sets are clearly separated and of different sizes (Section VI-B1, Figure 6)

(b) Higher variability in w_{iter}, hence sets are less separated (Section VI-B2, Figure 9)

(c) A fourth job profile is added, not a new order of magnitude (Section VI-B2, Figure 10)

Fig. 5: Workloads from some of our experiments. Each dot represents a job: characteristic time (w_{iter}) and I/O ratio (α_{io}).

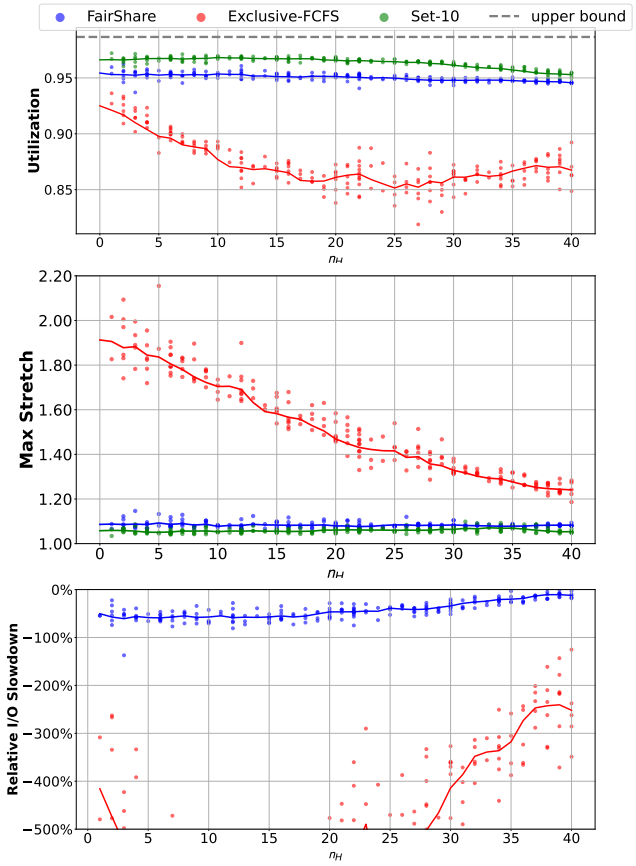


Fig. 6: Comparison of the policies over the studied objectives. The depicted metrics are `Utilization` (the higher, the better), `Max Stretch` (the lower, the better), and the relative `IO-slowdown`, which shows the relative loss of FAIR-SHARE and EXCLUSIVE-FCFS to SET-10 (higher is better).

The first observation is that `Utilization` and `Max Stretch` of both SET-10 and FAIR-SHARE are close to their optimal values (when there is no I/O congestion). This is not surprising given the low volume of I/O *per* application: applications spend on average 1.33% of their time on I/O. In practice, SET-10 is still able to get closer to the bounds, with a gain for both objectives of 2.5% over FAIR-SHARE. When looking closer at the I/O performance (`IO-slowdown`), this corresponds to a steady gain of 50% over FAIR-SHARE!

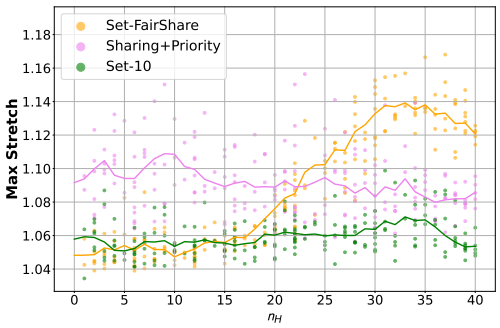


Fig. 7: Studying the respective impact of the set mapping and priority assignment parts of SET-10 (lower is better).

EXCLUSIVE-FCFS has poor results compared to the others, confirming the intuition: when applications have different profiles, exclusivity of I/O accesses should not be considered.

When there are diverse job profiles, EXCLUSIVE-FCFS is not a good strategy for scheduling I/O. While it is important to share the bandwidth, keeping some exclusivity has a positive impact on performance, showing the importance of the IO-SETS method.

The improvement brought by SET-10 is caused by both the set mapping and the priority assigning heuristics. To study the contribution of each of these parts, in Figure 7 we compare SET-10 to the two other policies discussed in Section IV-D: SET-FAIRSHARE that uses the π set mapping heuristic but gives the same priority for all jobs, and SHARE+PRIORITY where all jobs can perform I/O at the same time but with portions of the bandwidth according to SET-10's heuristic. We can see both parts of SET-10 are complementary for the `Max Stretch`: the exclusive access inside groups is essential when there are more low-frequency applications (n_H is low), while giving more of the bandwidth to high-frequency applications is important when there are more of those. From a `Utilization` perspective, the difference between heuristics that use bandwidth priority (SET-10 and SHARE+PRIORITY) are less important.

In a second step, we study the robustness of SET-10 to variability in the phases' duration. This variability can come from the machine (variability in compute and I/O performance) or from the application behavior.

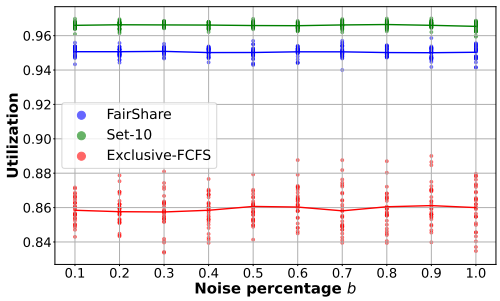


Fig. 8: Utilization (the higher, the better) according to the percentage of noise introduced in phases' length.

Figure 8 presents results obtained by varying the "noise parameter" b (discussed in Section V-B) for the case where $n_H = n_M = n_L = 20$. The main observation is that variability in the iteration length has no impact on any of the heuristics. While this could be expected from FAIR-SHARE and EXCLUSIVE-FCFS, it confirms that an average w_{iter} is enough for the design of our heuristics and that exact information about the length of each phase is not needed.

This robustness shows that the IO-SETS method can be used in more realistic settings. In addition, for the remainder of the evaluation, we can safely focus on periodic applications without loss of generality.

2) (Hardest scenarios) Sets do not match the mapping algorithm, evaluation of the robustness of SET-10*:* In the previous section, we have considered jobs generated in a way that translates well to the set mapping strategy π defined in Equation (3). In this part, we are interested in evaluating the efficiency and limitations of this mapping strategy. In order to do so, we present two evaluations: (i) in the first scenario we increase the standard deviation parameter until the generated jobs' characteristic times are totally spread (Figure 5b); (ii) in the second scenario, we generate jobs that we believe would fit well with another mapping strategy (not SET-10's).

Influence of the variability of w_{iter}: For various values of $\eta = \mu/\sigma$, we generate workloads with:

- Twenty jobs with characteristic time $w_{\text{iter}} \sim \mathcal{N}(10, \eta 10)$
- Twenty jobs with $w_{\text{iter}} \sim \mathcal{N}(100, \eta 100)$
- Twenty jobs with $w_{\text{iter}} \sim \mathcal{N}(1000, \eta 1000)$

Results are presented in Figure 9. An example of how the distribution of w_{iter} looks for $\eta = 0.8$ is in Figure 5b.

The job variability does not impact FAIR-SHARE because that strategy does not rely on jobs' characteristics to make decisions, differently from SET-10 that maps jobs to sets based on their similarities. Still, Figure 9 shows that SET-10 keeps a stable performance regardless of the σ. Its relative stretch compared to FAIR-SHARE remains 0.98 either when the sets are well-defined ($\sigma/\mu = 0.05$) or when characteristic times are more distributed over the space of possibilities ($\sigma/\mu = 1$). Looking at how SET-10 mapped the applications, we see that the variability is handled by moving the applications into different sets of similar orders of magnitude. In the more extreme cases, more sets are created. For instance, with $\sigma/\mu = 1$, SET-10 mapped applications into five distinct sets.

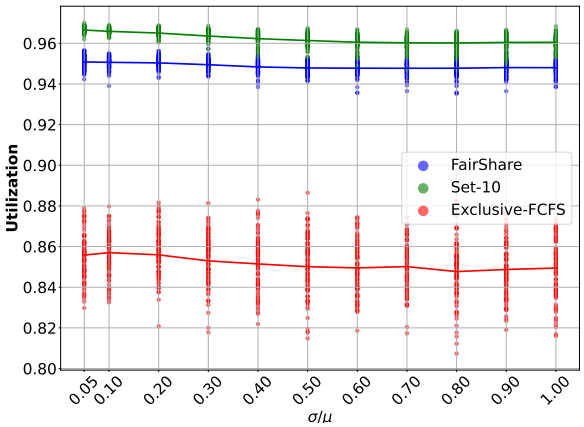


Fig. 9: Evaluation of SET-10 for the cases where sets are not well-defined. The graph shows the utilization (the higher, the better) according to the increment of the standard deviation used for the workloads.

SET-10 not only has excellent performance for jobs with specific characteristic times, but its mapping function can tackle even more heterogeneous workloads.

Comparison to other mapping strategies: The SET-10 strategy relies on the mapping allocation π (see Equation (3)). In previous tests, the workloads are adapted by design to this mapping function. Properly defining the optimal function represents a research topic of its own and is left as future work. For now, we want to explore whether π stays relevant when the workload distribution is not adapted to it.

In this evaluation, we add to the balanced workload defined earlier (3 groups with w_{iter} following respectively $\mathcal{N}(10, 1)$, $\mathcal{N}(100, 10)$, and $\mathcal{N}(1000, 100)$), a fourth profile of $w_{\text{iter}} \sim \mathcal{N}(\mu, 0.1\mu)$, with $\mu \in [5; 5000]$. Each of these groups has 15 applications, so we still have a total of 60 per workload. An example of the job distribution is represented in Figure 5c.

As a baseline to our set mapping strategy, we also compare results to a *fictitious* strategy, here called CLAIRVOYANT. It creates sets based on the way we generated them instead of determining them using w_{iter}, for example all applications based on $\mathcal{N}(10, 1)$ are put in the same set. The CLAIRVOYANT policy is therefore a fictitious oracle, because in practice sets can only be estimated based on observations from the jobs.

Figure 10 shows the percentage increment in `Utilization` over CLAIRVOYANT. FAIR-SHARE and EXCLUSIVE-FCFS do not use a mapping allocation therefore their variations are solely due to workload specifications. SET-10 and CLAIRVOYANT behave consistently in all cases showing that π represents a robust heuristic for set mapping, taking into account the challenge of separating applications with short characteristic time and grouping applications with long ones. It avoids the pitfalls of penalizing high-frequency applications while ensuring that some accesses are still

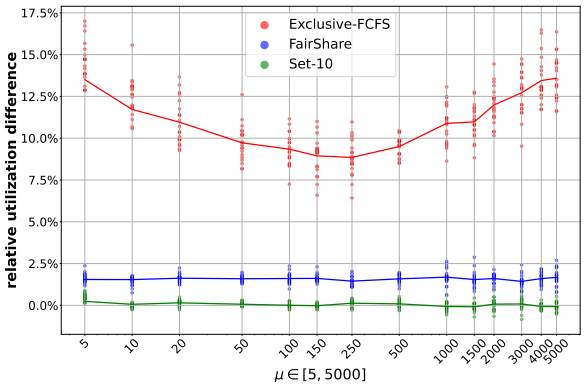


Fig. 10: Comparison of the strategies with the addition of a fourth application profile (x-axis). Results are normalized by the CLAIRVOYANT ones. A value above zero means the utilization was lower (worse) than with CLAIRVOYANT.

exclusive.

This set of experiments seems to confirm the fact that the mapping function used by SET-10 is a good choice: even against an adversary which knows the application sets, its performance is extremely robust.

VII. On the practical applicability of IO-Sets

After using simulation for evaluating our IO-SETS method, here we discuss how it can be implemented in practice.

A. Enforcing I/O scheduling

The method requires a centralized *scheduling agent* to allow or delay each application, and it must be contacted by applications *before* each I/O phase. We believe it would be best to make it transparent to applications, i.e. they do not have to handle these interactions explicitly.

To achieve that, a *local agent* (which could be part of the file system's client) on each compute node could intercept all I/O operations made by the application. Additional logic is required to detect beginning and ending of an I/O phase (which can correspond to multiple write or read requests). Working at the level of phases and not individual requests adds complexity but has the advantage of decreasing overhead. Such a detection could be based on time: requests from the same phase have a certain inter-arrival time (which will be very short), and longer periods of inactivity precede and follow a phase. FTIO [21] is also an option for the detection of I/O phases, as discussed in Section VII-B.

It is important to notice that this last step should be centralized at the application level by having an *application agent* per job. Indeed, it makes more sense to grant authorization for the whole application, and this would also scale better as the number of jobs in a supercomputer is expected to be considerably smaller than the number of compute nodes.

Another possible implementation for the IO-SETS method could involve the I/O nodes [22]. They could coordinate with a centralized agent to allow/delay applications in the same way as described above. This alternative has the advantage of not including any code to be executed in the compute nodes.

B. Estimation of the characteristic time

The application agent (or the centralized one) could keep track of the time between the start of consecutive I/O phases and update an estimation of w_{iter} for each job. That estimation will, of course, not be available for the first phase. In this case, the scheduler could for instance be conservative and temporarily map the application to a low-priority set.

In a recent work, Tarraf et al. [21] proposed FTIO, an approach that includes the use of the Discrete Fourier Transform (DFT) to detect the periodicity of I/O phases. The "period" provided by the tool corresponds to our w_{iter}, as the results showed it approximates really well the average time between I/O phases even for non-periodic applications. FTIO can be used either for post-mortem profiling or during the execution, to detect the current behavior. In both scenarios, it can provide this information needed for IO-SETS.

An idea for future work is to update w_{iter} in a way — for instance, using exponentially weighted moving average — that allows it to adapt to changes in the behavior of the application. With IO-SETS, there is no reason why an application could not change sets and priority during its execution.

C. Weighted bandwidth sharing

In addition to access control, IO-SETS involves the use of priorities to share the bandwidth. Here we discuss two ideas on how to implement that.

1) Number of processes/threads: When increasing the number of processes used to access a PFS, performance usually increases until a peak, remains stable, and then starts to degrade. This behavior can be observed in Figure 11. In this context, we could define each set's priority as the number of processes it is allowed to use for I/O (the higher the priority, the more processes it uses). They must be chosen so that the number given to the lowest priority and the sum of all numbers (i.e. the total number used when there is exactly one application from each set doing I/O) are able to reach the system's peak performance (they are both in that *plateau* in Figure 11). That ensures each application gets a portion of the bandwidth that is proportional to its priority when sharing, and all of the bandwidth when running alone.

In the platform from Figure 11, the lowest priority should receive at least 32 processes, and in total no more than 256 should be used. An adapted heuristic in the IO-SETS method could — assuming three sets but this can easily be extended to another number — assign 32 to the lowest-priority set, 64 to the middle one, and 128 to the highest-frequency, for example.

In our proof-of-concept experiment, shown in Figure 12, concurrent jobs from different sets do I/O at the same time using priority-based numbers of processes. We can see that, for instance, the highest-priority application (HP) always gets roughly twice as much bandwidth as the medium-priority one (MP), and four times as much as the lowest-priority job (LP).

A challenge in this idea would be to adapt the number of processes/threads used by each application for I/O (and changing it if its behavior evolves). That could be done by extending an I/O library or the PFS client that would redistribute data to the right number of processes, similarly

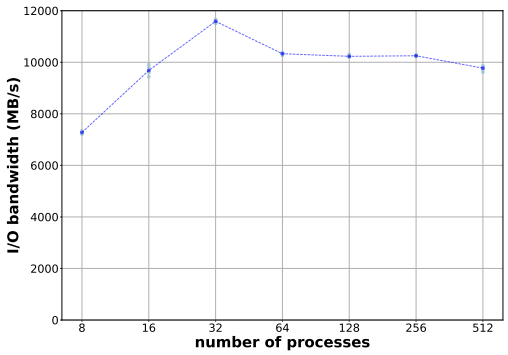


Fig. 11: Average write bandwidth (from five runs) measured with IOR for different numbers of processes. In each run, eight nodes are used to write a total of 32 GB to a BeeGFS file system, using 1 MB requests and the file-per-process strategy.

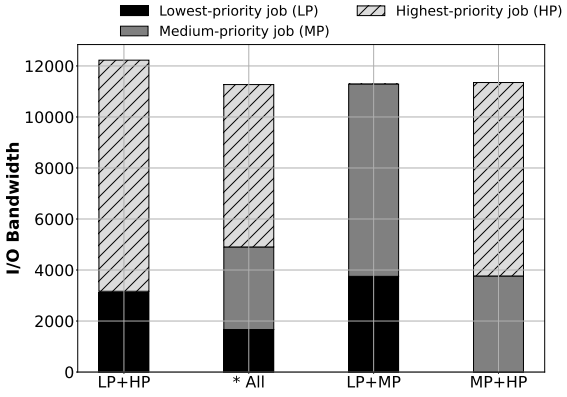


Fig. 12: Bandwidth sharing based on priorities. Each bar represents an experiment where multiple benchmarks (generated with IOR) perform I/O using 32, 64, or 128 processes. The write bandwidth values (average of five runs) are stacked, so the total height is the aggregate bandwidth. IOR parameters are the same as in Figure 11, but the amount of data per application is adapted so they have the same duration.

to how two-way collective operations work [23]; or simply by using more or less threads for I/O. Further work is required to study the overhead that would be imposed by this strategy.

2) Weighted bandwidth sharing: Another hint comes from the fact the notion of sharing a resource according to priorities or weights exists — often using the Weighted Fair Queuing (WFQ) algorithm — in contexts such as networks, where publications discuss bandwidth sharing [24], fairness [25], and flow-level models [26]. It has also been used in disks [27], [28]. It is important to notice our proposal assigns priorities/weights and WFQ is a way of enforcing them, i.e. WFQ is *not* an alternative to IO-SETS.

Qian et al. [29] propose a token bucket filter scheduler for Lustre that limits the number of pending RPC calls a job may have. That is done to respect pre-defined QoS rules including assigning portions of the bandwidth to jobs. The challenge of using that for IO-SETS is that we want each job to have access to the full bandwidth when doing I/O by itself. ASCAR [30] uses fully-decentralized TCP-like congestion control in the compute nodes to try to detect and reduce congestion. Actions taken in this regard depend on "traffic rules", which the authors argue are too difficult to create and thus they are automatically obtained by an unsupervised optimizer that searches for and stores the best rules for each given workload. Their congestion control works by dynamically limiting the number of pending I/O requests, which has the consequence of limiting the used bandwidth. Xu et al. [31] propose vPFS, a mechanism that works on the PFS servers to intercept requests from applications and apply a distributed weighted fair queuing request scheduler, with the goal of enforcing bandwidth sharing rules. Patel et al. [32] propose GIFT, a coupon-based bandwidth allocation approach that aims to maximize the I/O bandwidth utilization while ensuring fairness among concurrent applications on parallel storage systems. The authors demonstrated that the I/O bandwidth could be partitioned between applications by throttling (when necessary) the access of applications at the storage targets level. That approach comes from the observation that in the I/O path (i.e., compute nodes, network, routers and storage servers) of an HPC platform, the storage targets are usually the components with the lowest bandwidth.

These proposals have similarities to the proportional bandwidth sharing part of ours, and in fact could be adapted and used to implement it.

The priority-based bandwidth sharing that we propose can be implemented in different ways, including WFQ server-side request scheduling and by adapting the number of processes applications use for I/O. We showed the latter works through an experiment. Further investigating (and implementing) these strategies in different contexts should be the subject of future work, motivated by our findings of the relevance and impact of the IO-SETS method.

VIII. ON THE RELEVANCE OF IO-SETS

In Section VI, we showed that our method has results equivalent or better than FAIR-SHARE, including in situations that are not favorable, notably sets that are not well separated. In this context, the relevance of IO-SETS and of the SET-10 heuristic depends on the possibility of observing in practice scenarios where concurrent applications belong to different sets. To confirm this happens, we studied one year of job I/O traces generated in our PlaFRIM platform. The Web Supplementary Material [18, Section B] gives more details about this study. The FTIO [21] tool was used to obtain the w_{iter} for each job, and we focused on jobs that achieve at least 100 MB/s and that have more than one I/O phase, a total of 4088 jobs.

When applying the mapping function presented in Equation 3, we have the following percentages of jobs in each set:

- Set S_0: 321 jobs (7.85%)
- Set S_1: 1866 jobs (45.65%)
- Set S_2: 1369 jobs (33.49%)

- Set S_3: 435 jobs (10.64%)
- Set S_4: 88 jobs (2.15%)
- Set S_5: 9 jobs (0.22%)

Hence, based on the traces from our platform, it can be observed that our assumption that applications performing I/O can be classified into distinct sets is correct. Moreover, the data reveals that the majority of the I/O-bound applications fall into either sets 1, 2, or 3, which strengthens the validity of the simulated scenarios presented in Section VI, where we specifically focused on these three sets.

We further separated time into 5-hour windows and found 652 such windows where our 4088 interesting jobs executed. We observed that the jobs inside a window belonged to at least two distinct sets in 46.85% of the cases. Moreover, three sets were found in 27.45% of the executions, while executions with only one set represented 15.64% of the cases. Executions with 4 or more sets were found in 9.17% of the execution cases. Therefore, our assumption that concurrently running applications usually belong to different sets is supported by our data.

IX. Related Work

In HPC machines, the computing nodes are arbitrated between jobs, but they can usually access the shared I/O infrastructure without any control. When they compete for this resource, congestion lowers their I/O performance [33], [34]. This problem is worsened by the fact applications typically perform I/O in bursts [35], [36], [10]. This creates periods of time of high I/O stress in the system and increases the impact of I/O performance on application performance (because these I/O phases are often synchronous). Solutions such as burst-buffers [37] allow to smooth out the I/O requests over time, reducing the chance of an I/O peak. However, there still remains an I/O scheduling problem [38] of selecting when data must be moved to/from the burst-buffers. Furthermore, these devices have a high cost and finite capacity.

A. I/O Scheduling

Gainaru et al. [10], like us, studied centralized online scheduling of accesses to the shared PFS. They proposed heuristics for selecting what jobs can perform I/O while aiming to optimize different metrics. These greedy heuristics must keep track of the current values of these metrics and the impact of different choices on them (which involves information about applications' phases).

Zhou et al. [11] enhanced the batch scheduler to monitor and control jobs' I/O during their execution. At every I/O request generated in the system, their scheduler measures current jobs' requested bandwidth and checks for contention. If it is detected, jobs are selected to be suspended according to heuristics they proposed and similarly to the work by Gainaru et al. In both of these proposals, the scheduler respects the maximum bandwidth capacity of the system when allowing I/O accesses, i.e. they happen with "semi-exclusive" access to the PFS. As we have shown, bandwidth sharing has advantages and often provides better results than exclusive access.

In a more recent contribution, Gainaru et al. [7] proposed a strategy where a schedule is computed once during the batch scheduling phase, giving each job that is being launched the exact times when it can perform I/O. For that, the scheduler requires a complete profile of the applications' phases. In practice, such a detailed profile is i) expensive and difficult to obtain, and ii) not always present or accurate because application behavior may change with input data and the scheduler may not be able to identify the application. Our proposal, on the other hand, uses only the w_{iter} metric which can be calculated during the execution with minimal overhead. Furthermore, it can adapt to changes in I/O behavior, which makes IO-Sets a more robust approach.

Dorier et al. [6] analyzed the benefits of either delaying or interrupting an application when it is interfering with another one. The authors conceptualized a software layer, CALCioM, that allows for inter-application communication and gathers information from all levels of the I/O stack. Such a layer could be used to implement our IO-Sets method. From their analysis, they pointed that the bigger the difference in size between applications, the less effective the exclusive approach is. However, it was found to be the best option when applications have similar I/O requirements.

That observation, that both exclusive and shared access have advantages, is the main motivation behind our work. This has also been documented by Jeannot et al. [8]. Their proposition of an I/O scheduling solution to a flow-shop problem variant is preferable to sharing under certain workload conditions. **To the best of our knowledge, ours is the first work to propose classifying applications into sets and using that for managing the bandwidth.**

1) I/O-aware batch scheduling: Others have improved the batch scheduler to consider I/O as one of the resources that should be arbitrated. Grandl et al. [39] and Tan et al. [40] enrich theoretical scheduling problems with additional dimensions such as I/O requirements. Bleuse et al. [41] attempt to schedule applications requesting both compute and I/O nodes. In their model, Herbein et al. [15] consider that jobs perform I/O proportionally to the number of nodes they need. They show that considering I/O helps reducing performance variability. The scheduler AIS [9] identifies, from server logs, the I/O characteristics of applications and issues recommendations for the batch scheduler (which consist of running or delaying jobs) depending on the current state of the system. Our approach is complementary to those, because we focus on the situation where there already is a set of applications running and requesting I/O.

B. I/O Behavior of Applications

As discussed in Section I, scheduling techniques often require some information about applications. Previous studies [35], [36], [10] reveal that HPC applications often perform computation and data transfer alternately. Moreover these studies show that this behavior can be predicted beforehand. The literature regarding the prediction of I/O patterns is abundant. Notable examples include: the use of I/O traces to build a Markov model representing spatial patterns [42]; a grammar-

based model [36] to predict future operations; large scale log analysis [43] or machine learning approaches [44].

In practice, basic information could be obtained at run time from monitoring tools such as TAU [45], Paraver [46], SCALASCA [47], and Paradyn [48]; or from previous executions using I/O profiling tools such as Darshan [35], [16].

In this context, we chose to build our method using information that is easy to obtain and robust to inaccuracies.

X. Conclusion

In this work we have introduced a novel method for I/O management in HPC systems: IO-Sets, intended to be simple and light in overhead. This method consists of a smooth combination of exclusive and non-exclusive bandwidth sharing. We have proposed the Set-10 algorithm, based on this method, where jobs are categorized depending on the order of magnitude of their mean time between consecutive I/O phases (which we call their "characteristic time" or w_{iter}).

Through an extensive evaluation, we have first shown the importance of IO-Sets, and then the excellent performance of the Set-10 algorithm on various challenging scenarios. Notably, Set-10 was shown to be quite robust to inaccurate information, due to being based on an average estimation instead of precise application information. Finally, we have provided insights on how our method can be implemented.

We believe that our results open a wide range of avenues for I/O management. Notably, they have been evaluated through simulations. This is an important first step, which merits further confirmation. The next step is to develop software to implement this method. We have provided some elements for its design. Furthermore, in a real setting one can expect some hypothesis not to hold anymore (such as the linear scaling of I/O). In addition, from a practical point of view, the implementation of IO-Sets will have to adapt to real applications that have more complex I/O behaviors that change over time. For this, the robustness in the results is auspicious.

From a theoretical standpoint, optimal set mapping and bandwidth partitioning algorithms can be investigated. An important direction that we will consider is that jobs can change sets at each I/O phase. For research on I/O monitoring tools, our results indicate that an average behavior may be enough information to obtain, and also cheaper. Finally, we believe our method could be successful in other parts of the I/O stack, for instance for shared burst buffers.

References

[1] "Top500," https://www.top500.org/.

[2] S. Garg, "Tflops pfs: Architecture and design of a highly efficient parallel file system," in *SC '98*, 1998, pp. 2–2.

[3] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea, "End-to-end I/O portfolio for the summit supercomputing ecosystem," in *SC'19*. New York, NY, USA: ACM, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356157

[4] S. P. Singh and E. Gabriel, "Parallel I/O on compressed data files: Semantics, algorithms, and performance evaluation," in *CCGrid'20*, 2020, pp. 192–201.

[5] Z. Wang, P. Subedi, M. Dorier, P. E. Davis, and M. Parashar, "Staging based task execution for data-driven, in-situ scientific workflows," in *Cluster'20*, 2020, pp. 209–220.

[6] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination," in *IPDPS'14*. IEEE, 2014, pp. 155–164.

[7] A. Gainaru, V. Le Fèvre, and G. Pallez, "I/O scheduling strategy for periodic applications," *ACM ToPC*, 2019.

[8] E. Jeannot, G. Pallez, and N. Vidal, "Scheduling periodic I/O access with bi-colored chains: models and algorithms," *J. of Scheduling*, vol. 24, no. 5, pp. 469–481, 2021.

[9] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems," in *SC'16*, Nov 2016, pp. 819–829.

[10] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC applications under congestion," in *IPDPS'15*. IEEE, 2015, pp. 1013–1022.

[11] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/O-aware batch scheduling for petascale computing systems," in *Cluster'15*. IEEE, 2015, pp. 254–263.

[12] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination," in *SC'10*. IEEE, 2010, pp. 1–11.

[13] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *JPDC*, vol. 74, no. 10, pp. 2899–2917, 2014.

[14] P. Velho, L. M. Schnorr, H. Casanova, and A. Legrand, "On the validity of flow-level tcp network models for grid and cloud simulations," *ACM TOMACS*, vol. 23, no. 4, pp. 1–26, 2013.

[15] S. Herbein, D. H. Ahn, D. Lipari, T. R. Scogland, M. Stearman, M. Grondona, J. Garlick, B. Springmeyer, and M. Taufer, "Scalable I/O-aware job scheduling for burst buffer enabled hpc clusters," in *ACM HPDC'16*, 2016, pp. 69–80.

[16] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright, "Modular HPC I/O characterization with darshan," in *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, 2016, pp. 9–17.

[17] A. Lebre, A. Legrand, F. Suter, and P. Veyre, "Adding storage simulation capacities to the simgrid toolkit: Concepts, models, and api," in *CCGrid'15*. IEEE, 2015, pp. 251–260.

[18] F. Boito, G. Pallez, L. Teylo, and N. Vidal, "Web supplementary material for "io-sets: Simple and efficient approaches for i/o bandwidth management"," PDF, 2023. [Online]. Available: https://gitlab.inria.fr/hpc_io/io-sets

[19] "IOR Benchmark, version 3.3.0," https://github.com/hpc/ior.

[20] F. Boito, G. Pallez, and L. Teylo, "The role of storage target allocation in applications' i/o performance with beegfs," in *Cluster'22*, 2022.

[21] A. Tarraf, A. Bandet, F. Boito, G. Pallez, and F. Wolf, "Ftio: Detecting i/o periodicity using frequency techniques," 2023.

[22] 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 *IPDPS'21*, 2021, pp. 577–586.

[23] G. Congiu, S. Narasimhamurthy, T. SuB, and A. Brinkmann, "Improving Collective I/O Performance Using Non-volatile Memory Devices," in *Cluster'16*. IEEE, sep 2016, pp. 120–129.

[24] L. Massoulié and J. Roberts, "Bandwidth sharing: objectives and algorithms," in *IEEE INFOCOM'99*. IEEE, 1999, pp. 1395–1403.

[25] D. M. Chiu, "Some observations on fairness of bandwidth sharing," in *ISCC'00*. IEEE, 2000, pp. 125–131.

[26] S. B. Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts, "Statistical bandwidth sharing: a study of congestion at flow level," *ACM SIGCOMM*, vol. 31, no. 4, pp. 111–122, 2001.

[27] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *ICMCS'99*. IEEE, 1999, pp. 400–405.

[28] P. J. Shenoy and H. M. Vin, "Cello: A disk scheduling framework for next generation operating systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, pp. 44–55, 1998.

[29] Y. Qian, X. Li, S. Ihara, L. Zeng, J. Kaiser, T. Süß, and A. Brinkmann, "A configurable rule based classful token bucket filter network request scheduler for the lustre file system," in *SC'17*, 2017.

[30] Y. Li, X. Lu, E. L. Miller, and D. D. E. Long, "ASCAR:automating contention management for high-performance storage systems," in *MSST'15*, May 2015, pp. 1–16.

[31] Y. Xu, D. Arteaga, M. Zhao, Y. Liu, R. Figueiredo, and S. Seelam, "vpfs: Bandwidth virtualization of parallel storage systems," in *MSST*, 2012, pp. 1–12.

[32] T. Patel, R. Garg, and D. Tiwari, "{GIFT}: A coupon based {Throttle-and-Reward} mechanism for fair and efficient {I/O} bandwidth man-

agement on parallel storage systems," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 103–119.
[33] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *IPDPS'16*, 2016, pp. 750–759.
[34] C.-S. Kuo, A. Shah, A. Nomura, S. Matsuoka, and F. Wolf, "How file access patterns influence interference among cluster applications," in *Cluster'14*, 2014, pp. 185–193.
[35] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Cluster'09 Workshops*. IEEE, 2009, pp. 1–10.
[36] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'io: a grammar-based approach to spatial and temporal I/O patterns prediction," in *SC'14*. IEEE, 2014, pp. 623–634.
[37] 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 *CCGrid'20*, 2020, pp. 142–151.
[38] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "Sizing and partitioning strategies for burst-buffers to reduce IO contention," in *IPDPS'19*. IEEE, 2019, pp. 631–640.
[39] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, Aug. 2014.
[40] T. T. Tran, M. Padmanabhan, P. Y. Zhang, H. Li, D. G. Down, and J. C. Beck, "Multi-stage resource-aware scheduling for data centers with heterogeneous servers," *J. of Scheduling*, vol. 21, no. 2, pp. 251–267, Apr. 2018.
[41] R. Bleuse, K. Dogeas, G. Lucarelli, G. Mounié, and D. Trystram, "Interference-aware scheduling using geometric constraints," in *Euro-Par'18*. Springer, 2018, pp. 205–217.
[42] J. Oly and D. A. Reed, "Markov model prediction of I/O requests for scientific applications," in *ICS'02*, 2002, pp. 147–155.
[43] S. Kim, A. Sim, K. Wu, S. Byna, Y. Son, and H. Eom, "Towards HPC I/O performance prediction through large-scale log analysis," in *HPDC'20*, 2020, pp. 77–88.
[44] S. Madireddy, P. Balaprakash, P. Carns, R. Latham, R. Ross, S. Snyder, and S. Wild, "Modeling I/O performance variability using conditional variational autoencoders," in *Cluster'18*. IEEE, 2018, pp. 109–113.
[45] S. S. Shende and A. D. Malony, "The tau parallel performance system," *IJHPCA*, vol. 20, no. 2, pp. 287–311, 2006.
[46] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *WoTUG'18*. Citeseer, 1995, pp. 17–31.
[47] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *CCPE*, vol. 22, no. 6, pp. 702–719, 2010.
[48] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

BIOGRAPHIES

Francieli Boito is an associate professor at the University of Bordeaux, working in an Inria team. Her research interests revolve around I/O and storage for HPC, including techniques for contention mitigation and application characterization. See https://www.labri.fr/perso/fzanonboito/ for details.

Guillaume Pallez is a tenured researcher at Inria Rennes. His research interests include algorithm design and scheduling techniques for parallel and distributed platforms (data-aware scheduling, stochastic scheduling etc). Among other, he served as the Technical Program vice-chair for SC'17, Technical program chair for SC'24, and co-general chair for ICPP'22. He was a recipient of the 2019 IEEE TCHPC Early Career researcher award. See http://people.bordeaux.inria.fr/gaupy/ for further information.

Luan Teylo received his PhD in Computer Science in 2021 from Fluminense Federal University (Brazil). He graduated in Computer Science in 2015 and obtained a master's degree in 2017. Since 2021, he has been a postdoc in the TADaaM team at INRIA Bordeaux – Sud-Ouest, where his main research topic is I/O in HPC systems. His research interests include I/O, distributed algorithms, cloud computing, and scheduling problems. See http://luanteylo.com for further information.

Nicolas Vidal received his PhD in Computer Science in 2022 from University of Bordeaux. He is now a postdoc at Oak Ridge National Lab.