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

Francieli Zanon Boito, Guillaume Pallez, Luan Teylo, Nicolas Vidal

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

Guillaume Pallez*, Luan Teylo*, Nicolas Vidal*, Francieli Zanon Boito*
*Inria Bordeaux, LaBRI, University of Bordeaux, France

***Abstract*—One of the main performance issues faced by high-performance platforms is the delay caused by concurrent applications performing I/O. When this happens, applications sharing the I/O bandwidth create congestion, which delays the execution time and affects the platform's overall performance. Several solutions have been proposed to tackle I/O congestion. Among those solutions, I/O scheduling strategies are an essential solution to this problem in HPC systems. Currently, their main drawback is the amount of information needed. In this work, we propose a novel framework for I/O management, IO-SETS. We present the potential of this framework through a simple scheduling solution called SET-10, an I/O scheduling solution that demands minimum information from users and can be easily integrated into HPC environments.**

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 much faster pace. 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 [14]. Twenty years later, in November 2019, the fastest machine was Summit, with 200.8 PFlops (over $62,000\times$ faster) and a PFS capable of 2.5 TB/s ($2,500\times$ faster) [29].

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 [32]. 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 [38]. 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, which means they hold compute resources for longer, hurting the utilization of the platform. Moreover, performance variability increases, since the application's execution time depends not only on what it does but 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, and the topic of this work, is I/O scheduling [9], [13], [18], [23]: it consists in deciding algorithmically which application(s) get priority in this access.

In this work, we focus on I/O scheduling strategies. Precisely, we consider the scenario where applications compete for the parallel file system's bandwidth. Various approaches were proposed for this scenario:

Clairvoyant approaches [12], [13], [42] 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 (the I/O served is that of the application that performed I/O the least recently) [41], [12].

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 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 method is to use *fair-sharing*: the bandwidth is shared equally by all applications in a best-effort fashion. With fair-sharing, when two applications perform I/O concurrently, each one takes twice the time it would take by itself.

We propose a novel approach that is built on the intuition that both exclusive and fair-sharing have benefits. Section II details this intuition. With this respect, we design a two-pronged I/O management approach for HPC systems: first, applications are sorted into *sets*. If applications from the same set try to perform I/O concurrently, that is done 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 systems;
- an instantiation of this method with very simple heuristics that require little information about applications;
- a thorough evaluation of this heuristic that shows its impact and limitations even in extreme scenarios.

The next section presents the intuition behind our method and further motivates it. In Section III, we detail the problem we studied and our platform and application models, so that we can present our method and studied 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. The paper closes with related work in Section VIII and final remarks in Section IX.

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. [18, 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.

We illustrate this with a simple example. Consider two applications, (i) one that has relatively large I/O phases (we will 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 performs periodically: computation during one unit of time and I/O during 0.01 unit of time. We consider two scenarios in Figure 1: Two Large consists of two *large* applications competing for the I/O bandwidth, and Large+Small consists of a *large* and a *small* one. 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 I/O bandwidth between two applications does not have to be done fairly. In Figure 2, we consider the same two application profiles from the previous section, *small* and *large*. If the small I/O phase finishes before the large I/O phase, then giving it more of the available bandwidth improves *locally* (i.e. for this phase) the performance of *small* without delaying the *large* one.

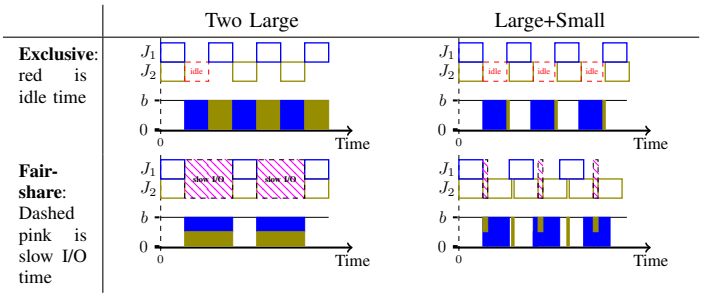


Figure 1: Two different pairs of jobs are shown side by side, and two schedulers are shown one above the other. For each of the four parts of the figure, the top half represents activity on compute nodes, and the bottom half the I/O accesses to the PFS (the height represents the portion of the bandwidth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

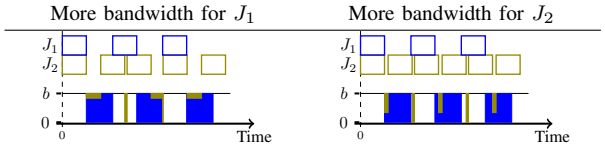


Figure 2: The applications share the bandwidth, but J_2 (the one with small I/O phases) receives more or less of it than J_1.

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

In this work, we assume that we have a parallel platform. This platform possesses 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 [12]. *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 [37]. 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 $p_1, p_2, \ldots, p_k$, then for $i \in \{1, \ldots, k\}$, the i^{th} request is allocated a share x_i of the total bandwidth such that:

$$x_i = \frac{p_i}{\sum_{j=1}^{k} p_j}. \quad (1)$$

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 [12], [17]. 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 I/O bandwidth is equally split between two I/O operations during the duration of their access ($x = 0.5$), those accesses take twice the time they would if they were 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 [33] is able to measure the total volume of I/O 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:

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

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 with 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 $S_i \in \{S_0, S_1, \ldots\}$.
- 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 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.

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

b) FAIR-SHARE*:* in this heuristic, the bandwidth is shared equally among all applications requesting I/O access. This can be modeled by the case where there is one set by application ($S_0, \ldots, S_k$), 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 once a delay equal to the length of the other's I/O phase, 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.

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 easy to obtain, as previously discussed. Hence we define the priority p_i of set S_i (which corresponds to jobs such that $i = \lfloor \log_{10} w_{\text{iter}}^{\text{id}} \rceil$):

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

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).

V. EVALUATION METHODOLOGY

The evaluation in this paper represents a first exploratory work to show the importance of the IO-SETS method. For that, we chose to use simulation because we wanted to test many scenarios, to extensively assess our method's strengths and limitations. For each of these scenarios, we included hundreds of different workloads trying to cover a maximum of combinations of application behaviors. That means the total set of experiments would consume a lot of time and resources (at least 569 days). Moreover, test platforms are somewhat limited in size, but deploying I/O scheduling strategies 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. Later, in Section VII, we discuss the practical applicability of our method and present some proof-of-concept results.

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.com/u693/iosets.

A. Simulating I/O with Simgrid

We use Simgrid v3.30 [6], an open-source simulator of distributed environments that also simulates I/O operations [21]. 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; ii) there is no latency associated with the operation; and iii) fair bandwidth sharing, i.e., by default concurrent I/O from applications receive the same portion of bandwidth (FAIR-SHARE). This model has been validated [21], and Simgrid is a well-established tool used in several projects and publications [37].

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). We create workloads of $N = 60$ parallel applications, and 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, 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) $\alpha_{\text{io}}^{(j)}$ is defined as follows: for each application, we draw a value α_j uniformly at random in $\mathcal{U}[0, 1]$, then we set $\alpha_{\text{io}}(j) = \frac{\omega}{\sum_{k \le N} \alpha_k} \cdot \alpha_j$ to guarantee an I/O Load of ω.

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

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

Using these values, we propose two different protocols to instantiate the individual lengths of each phase.

1) Periodic applications: For periodic applications, we assume that the lengths of all phases are identical. Hence, for all $i \le n$: $t_{\text{cpu}}^{(i)} = t_{\text{cpu}}$ and $t_{\text{io}}^{(i)} = t_{\text{io}}$.

2) Aperiodic applications: We introduce a noise parameter within a range of bound b. Then, for all $i \leq n$, we draw two variables: $\gamma_{\text{cpu}}^{(i)}$ and $\gamma_{\text{io}}^{(i)}$ from a uniform distribution: $\mathcal{U}[-b, b]$. From those:

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

The study of aperiodic applications allows us to take into account variations in application behavior, but also to represent the machine variability (both for processing and I/O performance). That makes our experiments more realistic.

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 [37], and satisfies Equation (1). We have used this to simulate the I/O bandwidth sharing behavior.

D. Measuring performance

When evaluating the performance of an I/O system, we need to consider the global performance of the HPC machine. From several possible metrics [12], [17], two usual ones are:

- system utilization: how much does the system compute per time unit, or equivalently, in our system, 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);
- application stretch: how long does it take for a user to get a result.

Our evaluation methodology aims to measure the performance of scheduling and mapping 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 need to limit our measurement to a time range encompassing only the steady state. Indeed, in our experiments all applications start at the beginning of the simulation. It means that the early stages are not representative in terms of arriving I/O requests. Reciprocally, in the final stages, some applications have finished. At this point, resources are underused, and further measurements represent the workload exhaustion instead of meaningful performance metrics.

To avoid such artifacts, we do our measurements 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: $T_{begin} = 1,000s$ and $T_{end} = 9,000s$. That ensures that it starts after a possible de-synchronization of the different applications and stops before any application finishes. 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.

a) 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.

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

`Utilization` is a system-wide metric representing the platform usage. It has an upper bound of 1 which would mean that jobs are always using efficiently the compute resources.

b) Stretch: From the users' point of view, the most important factor is the speed of their applications. From a fairness perspective, we aim to optimize the worst-case scenario. We define the stretch as the maximum, over all applications, of the ratio between the total time and the effective work executed.

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

`Stretch` is an application-centered metric that represents the maximum slow-down of an application. The lower, the better, with a stretch equal to 1 meaning that all applications run at the same speed as they would in isolation.

VI. EVALUATION AND RESULTS

In this section, we progressively evaluate the IO-SETS method. In a first step (Section VI-A), we consider a favorable scenario where applications are constructed efficiently for the set mapping function of SET-10. We then study the limits and robustness of this heuristic when the distributions of characteristic times are not optimized for SET-10 (Sec. VI-B).

All experiments presented in this section are repeated 200 times.

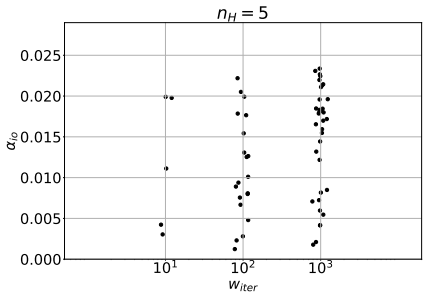


Figure 3: Section VI-A (Figure 4)

A. Validation

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 the heuristic SET-10. By doing that, we temporarily put aside the question of designing an efficient heuristic for IO-SETS.

In Section III, we explained the intuition 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):

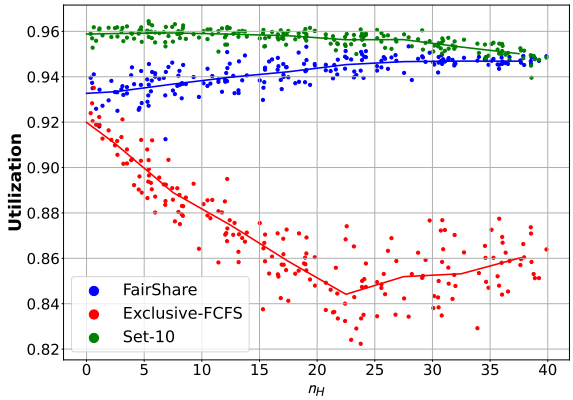
- n_H jobs with characteristic time $w_{\text{iter}} \sim \mathcal{N}(10, 1)$, also called high-frequency jobs in the following (the mean duration of an iteration is 10s);
- n_M medium-frequency jobs with $w_{\text{iter}} \sim \mathcal{N}(100, 10)$;
- n_L low-frequency jobs with $w_{\text{iter}} \sim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

As said in Section V-B, we set $n_H + n_M + n_L = 60$, and specifically we study the impact of the algorithms when $n_M = 20$, $n_H \in \{0, \ldots, 40\}$, and $n_L = 60 - n_H$. An example for the distribution of w_{iter} when $n_H = 5$ is given in Figure 3.

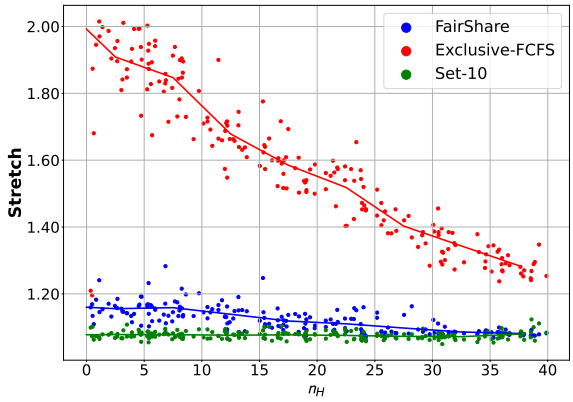
Periodic applications: First we study the case where applications are periodic. The results for this evaluation are presented in Figures 4a and **??**, where we study both `Utilization` and `Stretch`.

Utilization is presented in Figures **??**. FAIR-SHARE and SET-10 strategy have a high and stable platform usage, with an improvement up to 4.41% by SET-10 over FAIR-SHARE when there are very few high frequency jobs. This difference disappears when the number of high frequency jobs increases. On the contrary, EXCLUSIVE-FCFS has rather poor performance compared to the other heuristics, with a higher utilization when jobs are mostly large, confirming the intuition of Section II.

Figures **??** present the `Stretch` objective for each workload. This difference disappears as n_H increases. Here again, EXCLUSIVE-FCFS has the worst results in all cases, confirming its expected shortcomings. Interestingly, the utilization is not correlated to the `Stretch`: indeed, in an exclusive scenario, one of the high-frequency jobs can be penalized many times (which increases its `Stretch`), without hurting



(a) Machine utilization



(b) Maximum stretch

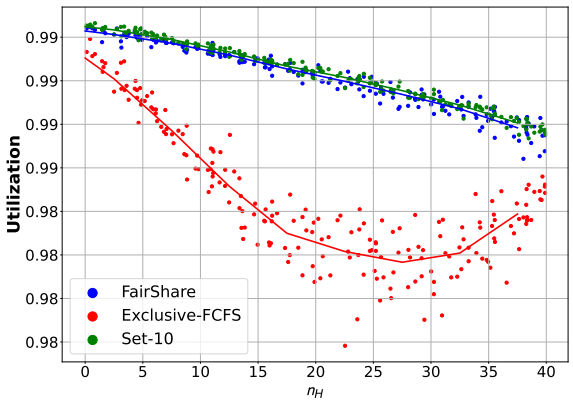
Figure 4: Comparison of scheduling policies EXCLUSIVE-FCFS, FAIR-SHARE, and SET-10, for workloads with sets of different sizes in a saturated system (an I/O stress $\omega = 0.8$). All 200 results obtained with each policy are presented. The y-axis is not the same for both plots, and they do not start at zero.

the `Utilization` significantly. This is confirmed if we look at the individual stretch of each application (Figure 6).

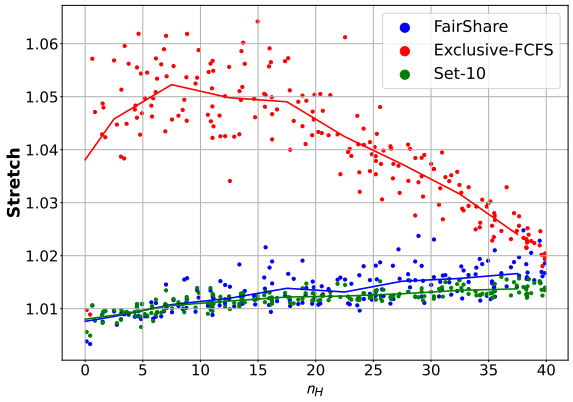
When there are diverse job profiles, EXCLUSIVE-FCFS is not a good strategy for scheduling I/O. While it is important to share the bandwidth, keeping some exclusivity has a positive impact on performance, showing the importance of the IO-SETS method.

Aperiodic applications: 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.

Figures **??** 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



(a) Machine utilization



(b) Maximum stretch

Figure 5: Comparison of scheduling policies EXCLUSIVE-FCFS, FAIR-SHARE, and SET-10, for workloads with sets of different sizes in saturated system (an I/O stress $\omega = 0.2$). All 200 results obtained with each policy are presented. The y-axis is not the same for both plots, and they do not start at zero.

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.

B. *Evaluation of the mapping strategy of* SET-10

In the previous section we have considered jobs generated in a way that translated well into the sets corresponding to the set mapping strategy π defined in Equation (3).

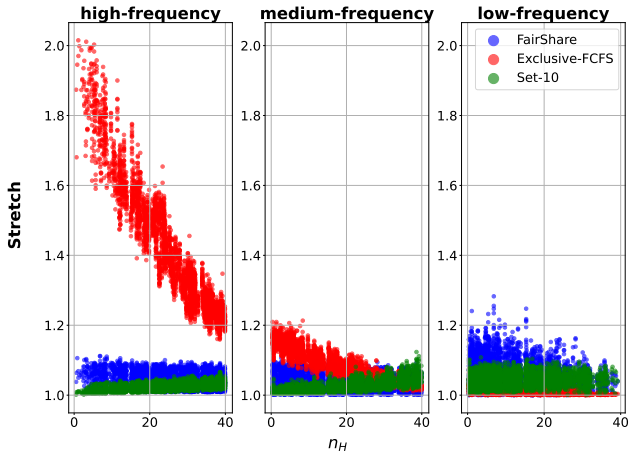


Figure 6: Individual stretch of all applications as a function of the number of high frequency tasks. Each dot corresponds to the stretch of an application. The maximum stretch of EXCLUSIVE-FCFS comes from high frequency set, while that of FAIR-SHARE from low-frequency set.

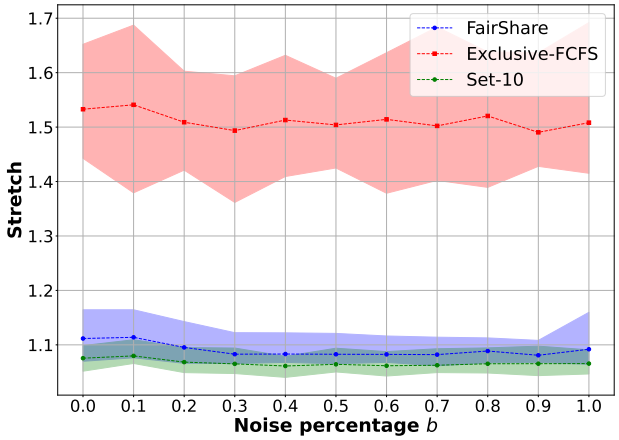


Figure 7: Maximum stretch according to the percentage of noise introduced in phases' length. 200 executions are represented (the line is the mean, the colored area is bounded by the minimum and the maximum values).

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 9); (ii) in the second scenario, we generate jobs that we believe would fit well with another mapping strategy (not SET-10's).

1) Influence of the variability of w_{iter}: For this evaluation, 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 $w_{\text{iter}} \sim \mathcal{N}(100, \eta 100)$

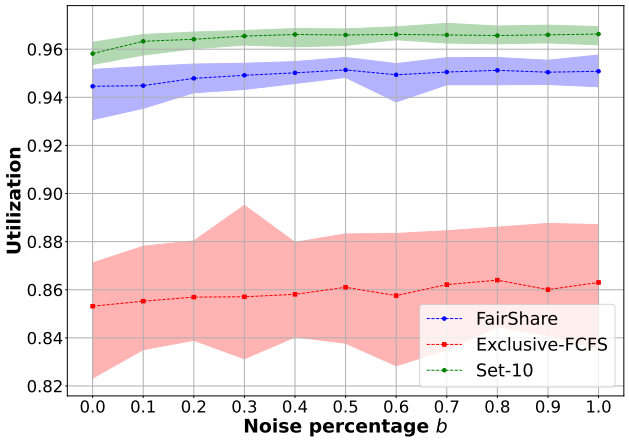


Figure 8: Utilization according to the percentage of noise introduced in phases' length. 200 executions are represented (the line is the mean, the colored area is bounded by the minimum and the maximum values).

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

Results are presented in Figure 10. An example of how the distribution of w_{iter} looks for $\eta = 0.8$ is in Figure 9.

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, Fig. 10 shows that SET-10 keeps a stable performance regardless of the σ. Its relative stretch compared to FAIR-SHARE goes from 0.94 when the sets are well defined ($\sigma/\mu = 0.05$) to 0.98 when characteristics 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.

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

2) 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 above (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.

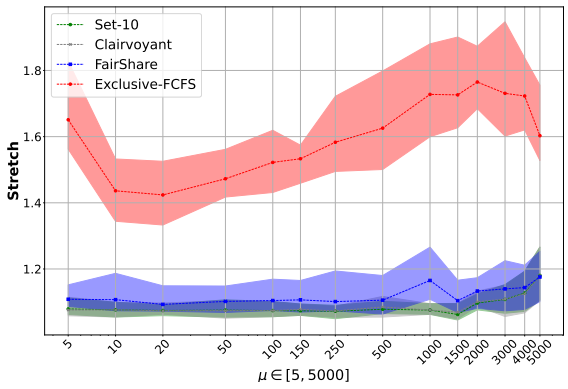


Figure 12: Comparison of the stretch evolution of FAIR-SHARE, EXCLUSIVE-FCFS, SET-10, and *clairvoyant* strategy according to the addition of a fourth application profile (x-axis). The lines represent the mean stretch of each strategy, and the area around the line is its confidence interval.

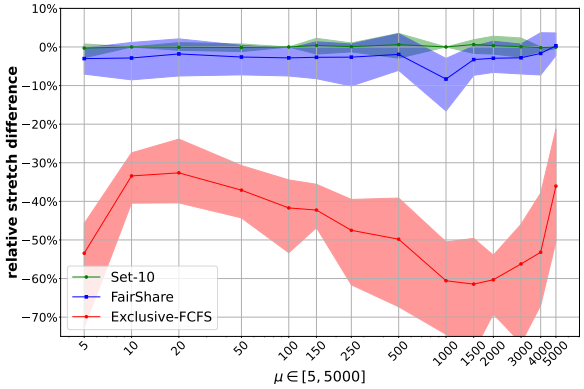
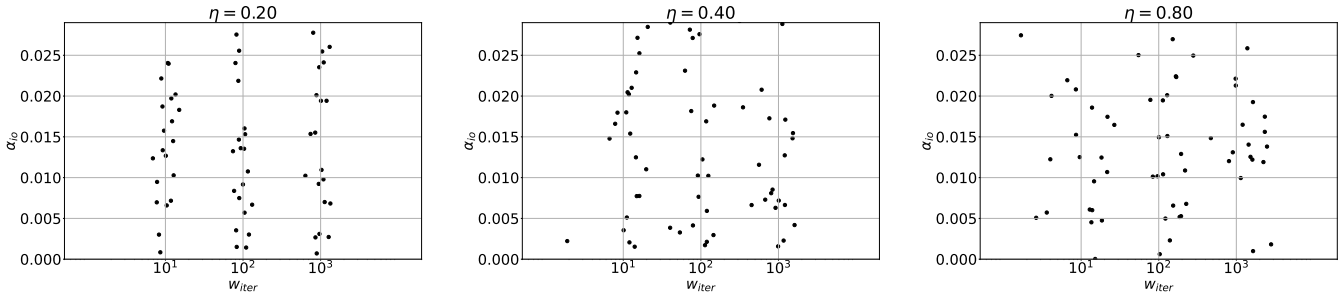


Figure 13: Normalized results of the stretch evolution presented in Fig. 12. The reported scheduling strategies (FAIR-SHARE, EXCLUSIVE-FCFS and SET-10) are normalized according to the *clairvoyant* stretch. In this case, a relative stretch (y-axis) below zero represents the cases where the stretch was worse than the *clairvoyant* strategy.



(a) $\eta = 0.2$ (b) $\eta = 0.4$ (c) $\eta = 0.2$

Figure 9: Workloads from Section VI-B1 : each dot represents characteristic time (w_{iter}) and I/O load (α_{io}) for a job.

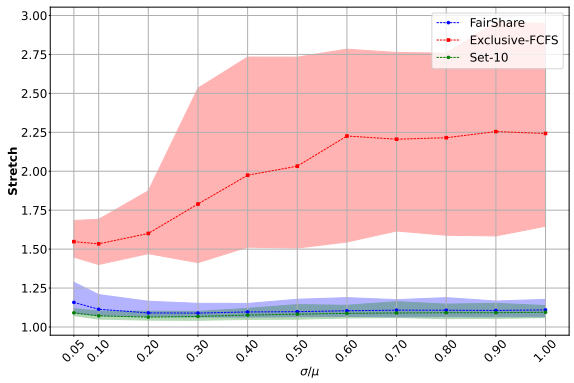


Figure 10: Evaluation of SET-10 for the cases where I/O sets are not well-defined. The graph shows the stretch evolution for SET-10, EXCLUSIVE-FCFS, and FAIR-SHARE according to the increment of the standard deviation used to generate the workloads

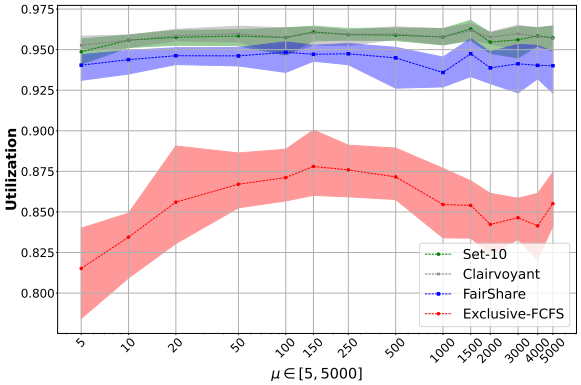


Figure 14: Comparison of the utilization evolution of FAIR-SHARE, EXCLUSIVE-FCFS, SET-10, and *clairvoyant* strategy according to the addition of a fourth application profile (x-axis). The lines represent the mean stretch of each strategy, and the area around the line is its confidence interval.

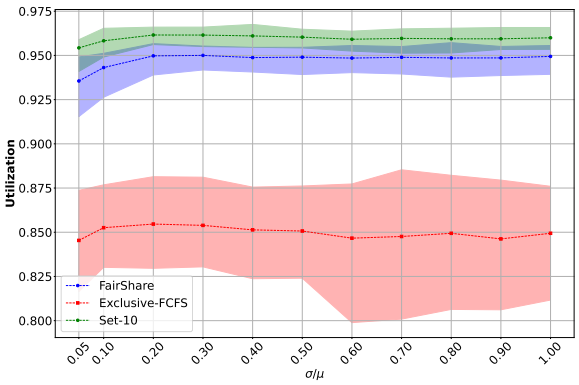


Figure 11: Evaluation of SET-10 for the cases where I/O sets are not well-defined. The graph shows the utilization evolution for SET-10, EXCLUSIVE-FCFS, and FAIR-SHARE according to the increment of the standard deviation used to generate the workloads

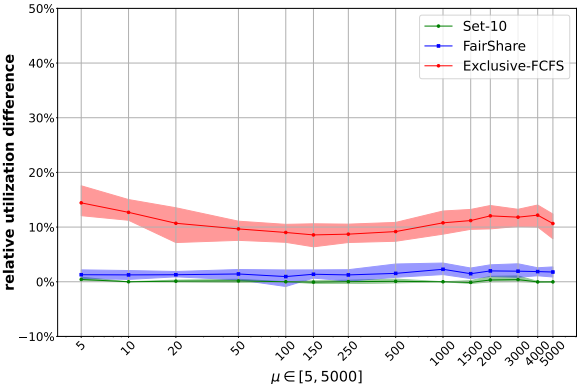


Figure 15: Normalized results of the utilization evolution presented in Fig. 12. The reported scheduling strategies (FAIR-SHARE, EXCLUSIVE-FCFS and SET-10) are normalized according to the *clairvoyant* stretch. In this case, a relative stretch (y-axis) below zero represents the cases where the stretch was worse than the *clairvoyant* strategy.

In addition SET-10, FAIR-SHARE, EXCLUSIVE-FCFS, we use as a baseline a so-called CLAIRVOYANT strategy. CLAIRVOYANT maps each application to a set based on the distribution used for its generation (e.g. all applications based on $\mathcal{N}(10, 1)$ belong in the same set).

Figure 12 shows the `Stretch` of all heuristics. In Figure 13, we show the improvement percentage of the `Stretch` over that of 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 heuristics 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 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 needs the ability of allowing each application to perform or to delay I/O operations. Therefore, a centralized *scheduling agent* is required to implement this control, and applications must contact it *before* each I/O phase.

We believe the best implementation for this would be one that is transparent to applications, i.e. they do not have to handle these interactions explicitly. To achieve that, a *local agent* (which could be implemented as part of the file system's client), running 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.

It is important to notice that this last step should be centralized at the application level to avoid having all compute nodes contacting the scheduler. Indeed, it makes more sense to grant authorization for the whole application and not for individual nodes/processes, 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.

Consequently, an *application agent* would receive the requests from the nodes that are being used by the job and issue a single request/release message to the centralized scheduler.

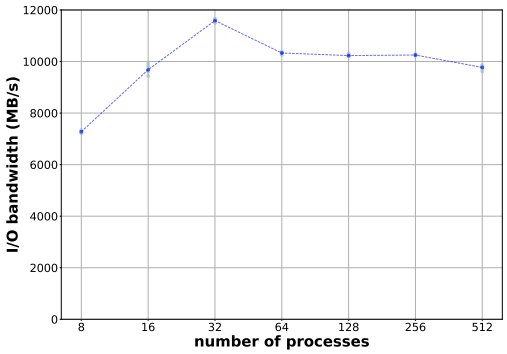


Figure 16: Average write bandwidth (from five runs) measured with IOR for different numbers of processes. In each run, eight nodes are used to write a total of 32 GB to a BeeGFS file system, using 1 MB requests and the file-per-process strategy.

Another possible implementation for the IO-SETS method could involve the I/O forwarding layer. In large HPC machines, all I/O accesses to the PFS must pass through the I/O nodes [3]. Therefore, the I/O nodes 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 easily 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.

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. "Unfair" bandwidth sharing

In addition to access control, another part of our IO-SETS method is the use of priorities to share the bandwidth. We believe this can be easily implemented by adjusting the number or processes used for I/O by applications of each set.

When accessing a PFS, the performance usually increases with the number of processes until reaching a peak. Further increasing it will not improve the bandwidth until some number where contention starts to degrade performance. This behavior can be observed in Figure 16.

In this context, we can implement the bandwidth sharing by defining each set's priority as the number of processes it is allowed to use to contact the PFS (the higher the priority, the more processes it uses). They must be chosen so that the number associated with the lowest priority and the sum of all numbers (i.e. the total number of processes used when there

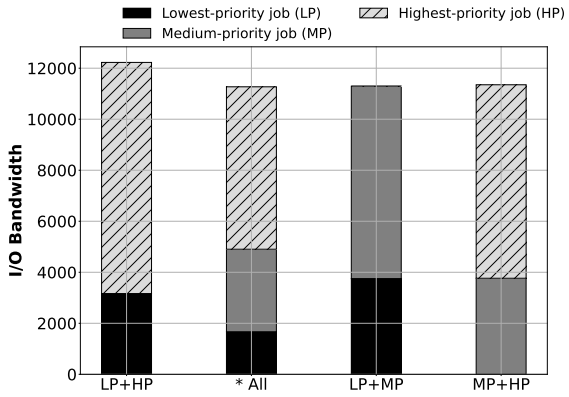


Figure 17: Bandwidth sharing based on priorities. Each bar represents an experiment where multiple benchmarks (generated with IOR) perform I/O using 32, 64, or 128 processes. The write bandwidth values (average of five runs) are stacked, so the total height is the aggregate bandwidth. IOR parameters are the same as in Figure 16, but the amount of data per application is adapted so they have the same duration.

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 16). 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 16, 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.

In our proof-of-concept experiment, shown in Figure 17, 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).

This experiment shows that the priority-based bandwidth sharing that we propose in this paper can be easily implemented in practice. Indeed, knowing a job's set, its priority level is obtained without depending on the other concurrent jobs, and the proportional partitioning of the bandwidth is done "naturally", without the need for further control.

The application agent discussed in Section VII-A could receive this number from the scheduler and redistribute data in order to use the assigned number of processes to actually do I/O (in a similar way to what is done in two-phase collective I/O). Alternatively, if the method is implemented in the I/O nodes, weighted fair queuing (WFQ) request scheduling could be used to enforce the priorities assigned to different jobs.

VIII. Related Work

In HPC machines, the computing nodes are arbitrated between jobs, but there is a shared I/O infrastructure that running applications are usually allowed to access without any control. When they compete for this resource, congestion lowers their I/O performance [39], [20]. This problem is worsened by the fact applications typically perform I/O in bursts [5], [10], [12]. 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)

Architectural solutions such as burst-buffers [35] allow to smooth out the I/O requests over time, reducing the chance of an I/O peak, and allowing for almost asynchronous I/O. However, there still remains an I/O scheduling problem [40] 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

In this paper, we proposed a method for managing applications' accesses to the shared parallel file system in an HPC platform. Other efforts were put into scheduling I/O operations aiming at mitigating interference. Gainaru et al. [12] and Zhou et al. [42] schedule applications' operations at the I/O node level. In a more recent contribution, Gainaru et al. [13] propose a pattern-based schedule relying on the periodic I/O behavior. Dorier et al. [9] analyze the benefits of either delaying or interrupting an application when it is interfering with another one. ASCAR [22] uses controllers on storage clients to detect and reduce congestion using traffic rules. The scheduler AIS [23] identifies offline the I/O characteristics of the application and uses this information to avoid conflicts by issuing I/O aware scheduling recommendations. Alves and al. [2], Snyder and al. [34] and Dorier and al. [8] use different models to predict and avoid I/O interference.

Our work is motivated by the observation that both exclusive and shared access to the bandwidth have advantages, as observed by Jeannot et al. [18]. **To the best of our knowledge, ours is the first work to propose classifying applications into sets and using that for managing the bandwidth.**

Others have improved the batch scheduler to consider I/O as one of the resources that should be arbitrated. Grandl et al. [16] and Tan et al. [36] enrich theoretical scheduling problems with additional dimensions such as I/O requirements. Bleuse et al. [4] attempt to schedule applications requesting both compute and I/O nodes. In their model, Herbein et al. [17] consider that jobs perform I/O proportionally to the number of nodes they need. They show that considering I/O helps reducing performance variability. Our approach is complimentary 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 [5], [10], [12] 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 [28]; a grammar-based model [10] to predict future operations; large scale log analysis [19] or machine learning approaches [25].

In practice, basic information could be obtained at run time from monitoring tools such as TAU [31], Paraver [30], SCALASCA [15], and Paradyn [27]; or from previous executions using I/O profiling tools such as Darshan [5], [33].

In this context, we chose to build our method using information that is easy to obtain and robust to inaccuracies. Our characteristic time metric can be estimated during the execution of the application without imposing a heavy overhead, and can adapt to changes in its behavior.

C. Networks

The priority-based bandwidth sharing part of our work is inspired by networks. Publications from that field include discussions about bandwidth sharing [26], fairness [7], and flow-level models [11]. Low [24] formally showed that the steady-state throughput of network flows are similar to the those obtained by solving a related global optimization problem. Velho et al [37] discuss the granularity needed for the simulation of communication.

IX. Conclusion

In this work we have introduced a novel method for I/O management in HPC systems: IO-Sets, intended to be easy to implement and light in overhead. This method consists in 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 characteristic times. The latter is an average value that corresponds to the mean time between two consecutive I/O phases of an application.

Though 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 the results from this paper open a wide range of avenues for I/O management. From a theoretical standpoint, optimal set mapping and bandwidth partitioning algorithms can be investigated. 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. 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 to manage the access to shared burst buffers.

Acknowledgment

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).

REFERENCES

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

[2] Maicon Melo Alves and Lucia Maria de Assumpçao Drummond. "A multivariate and quantitative model for predicting cross-application interference in virtual environments". *Journal of Systems and Software*, 128:150 – 163, 2017.

[3] Jean Luca Bez, Alberto Miranda, Ramon Nou, Francieli Zanon Boito, Toni Cortes, and Philippe Navaux. Arbitration Policies for On-Demand User-Level I/O Forwarding on HPC Platforms. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 577–586, 2021.

[4] Raphaël Bleuse, Konstantinos Dogeas, Giorgio Lucarelli, Grégory Mounié, and Denis Trystram. Interference-aware scheduling using geometric constraints. In *European Conference on Parallel Processing*, pages 205–217. Springer, 2018.

[5] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[6] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, 2014.

[7] Dah Ming Chiu. Some observations on fairness of bandwidth sharing. In *Proceedings ISCC 2000. Fifth IEEE Symposium on Computers and Communications*, pages 125–131. IEEE, 2000.

[8] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. Using formal grammars to predict I/O behaviors in HPC: The omnisc'io approach. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2435–2449, Aug 2016.

[9] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 155–164. IEEE, 2014.

[10] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. Omnisc'io: a grammar-based approach to spatial and temporal I/O patterns prediction. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 623–634. IEEE, 2014.

[11] S Ben Fred, Thomas Bonald, Alexandre Proutiere, Gwénaël Régnié, and James W Roberts. Statistical bandwidth sharing: a study of congestion at flow level. *ACM SIGCOMM Computer Communication Review*, 31(4):111–122, 2001.

[12] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[13] Ana Gainaru, Valentin Le Fèvre, and Guillaume Pallez. I/O scheduling strategy for periodic applications. *ACM Transactions on Parallel Computing*, 2019.

[14] S. Garg. Tflops pfs: Architecture and design of a highly efficient parallel file system. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 2–2, 1998.

[15] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and computation: Practice and experience*, 22(6):702–719, 2010.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, August 2014.

[17] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable I/O-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–80, 2016.

[18] Emmanuel Jeannot, Guillaume Pallez, and Nicolas Vidal. Scheduling periodic I/O access with bi-colored chains: models and algorithms. *Journal of Scheduling*, 24(5):469–481, 2021.

[19] Sunggon Kim, Alex Sim, Kesheng Wu, Suren Byna, Yongseok Son, and Hyeonsang Eom. Towards HPC I/O performance prediction through large-scale log analysis. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 77–88, 2020.

[20] Chih-Song Kuo, Aamer Shah, Akihiro Nomura, Satoshi Matsuoka, and Felix Wolf. How file access patterns influence interference among cluster applications. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 185–193, 2014.

[21] Adrien Lebre, Arnaud Legrand, Frédéric Suter, and Pierre Veyre. Adding storage simulation capacities to the simgrid toolkit: Concepts, models, and api. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 251–260. IEEE, 2015.

[22] Y. Li, X. Lu, E. L. Miller, and D. D. E. Long. ASCAR: Automating contention management for high-performance storage systems. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–16, May 2015.

[23] 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: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829, Nov 2016.

[24] Steven H Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Transactions On Networking*, 11(4):525–536, 2003.

[25] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan Wild. Modeling I/O performance variability using conditional variational autoencoders. In *2018 IEEE international conference on cluster computing (CLUSTER)*, pages 109–113. IEEE, 2018.

[26] Laurent Massoulié and James Roberts. Bandwidth sharing: objectives and algorithms. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 3, pages 1395–1403. IEEE, 1999.

[27] Barton P Miller, Mark D. Callaghan, Jonathan M Cargille, Jeffrey K Hollingsworth, R Bruce Irvin, Karen L Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[28] James Oly and Daniel A Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155, 2002.

[29] Sarp Oral, Sudharshan S. Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, and Veronica Vergara Larrea. End-to-end I/O portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31. Citeseer, 1995.

[31] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[32] Siddhesh Pratap Singh and Edgar Gabriel. Parallel I/O on compressed data files: Semantics, algorithms, and performance evaluation. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 192–201, 2020.

[33] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. Modular HPC I/O characterization with darshan. In *2016 5th workshop on extreme-scale programming tools (ESPT)*, pages 9–17. IEEE, 2016.

[34] Shane Snyder, Philip Carns, Robert Latham, Misbah Mubarak, Robert Ross, Christopher Carothers, Babak Behzad, Huong Vu Thanh Luu, Surendra Byna, and Prabhat. Techniques for modeling large-scale HPC I/O workloads. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, PMBS '15, pages 5:1–5:11, New York, NY, USA, 2015. ACM.

[35] Hanul Sung, Jiwoo Bang, Chungyong Kim, Hyung-Sin Kim, Alexander Sim, Glenn K. Lockwood, and Hyeonsang Eom. BBOS: Efficient HPC Storage Management via Burst Buffer Over-Subscription. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 142–151, 2020.

[36] Tony T. Tran, Meghana Padmanabhan, Peter Yun Zhang, Heyse Li, Douglas G. Down, and J. Christopher Beck. Multi-stage resource-aware scheduling for data centers with heterogeneous servers. *Journal of Scheduling*, 21(2):251–267, April 2018.

[37] Pedro Velho, Lucas Mello Schnorr, Henri Casanova, and Arnaud Legrand. On the validity of flow-level tcp network models for grid and cloud simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 23(4):1–26, 2013.

[38] Zhe Wang, Pradeep Subedi, Matthieu Dorier, Philip E. Davis, and Manish Parashar. Staging based task execution for data-driven, in-situ scientific workflows. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 209–220, 2020.

[39] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 750–759, 2016.

[40] Benbo Zha and Hong Shen. Improved probabilistic I/O scheduling for limited-size burst-buffers deployed HPC. *Parallel Computing*, 101:102708, 2021.

[41] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[42] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-aware batch scheduling for petascale computing systems. In *2015 IEEE International Conference on Cluster Computing*, pages 254–263. IEEE, 2015.