Scheduling distributed I/O resources in HPC systems

Alexis Bandet, Francieli Boito, Guillaume Pallez

Scheduling distributed I/O resources in HPC systems

Alexis Bandet, Francieli Boito
Univ. Bordeaux, CNRS, Bordeaux INP, Inria,
LaBRI, UMR 5800, F-33400 Talence, France
{alexis.bandet, francieli.zanon-boito}@inria.fr

Guillaume Pallez
Inria
Rennes, France
guillaume.pallez@inria.fr

***Abstract*—This paper presents a comprehensive investigation on optimizing I/O performance in the access to distributed I/O resources in high-performance computing (HPC) environments. I/O resources, such as the I/O forwarding nodes and object storage targets (OST), are shared between a subset of applications. Each application has access to a subset of them and multiple applications can access the same resources. We propose heuristics to schedule these distributed I/O resources in two steps: for a set of applications, determining the *number* of I/O resources each will use (allocation) and *which* resources they will use (placement). We discuss a wide range of required information about applications' characteristics that can be used by the scheduling algorithms. Despite the fact that a higher level of application knowledge is associated with enhanced performance, our comprehensive analysis indicates that strategic decision-making with limited information can still yield significant enhancements in most scenarios. This research provides insights into the trade-offs between the depth of application characterization and the practicality of scheduling I/O resources.**

***Index Terms*—HPC, parallel I/O, parallel file system, object storage targets, I/O forwarding, scheduling, resource allocation**

I. Introduction

In large high-performance computing (HPC) platforms, applications access persistent data by performing I/O operations to a remote shared parallel file system (PFS), such as Lustre or BeeGFS. These file systems cut files into fixed-size stripes and distribute them across a number of storage targets (OSTs) for parallel access. Depending on the files that they access, all compute nodes may require access to the same OSTs at the same time. Thus, to mitigate contention, a layer of I/O forwarding nodes (or simply "I/O nodes") is often placed between compute nodes and the PFS [1]. In this context, it is important to notice both OST and I/O nodes are potentially shared by running applications. Other shared resources include burst-buffer nodes, present in some systems [2], [3].

When applications access the same I/O resources concurrently, their I/O performance can be slowed down [4], [5], [6], and hence they occupy compute resources for longer. In addition to wasting resources, the fact that I/O performance depends on what others are doing in the system leads to higher performance variability [7], [8], which makes execution time less predictable and, consequently, complicates resource management [9]. Many techniques have been proposed to mitigate contention, mainly on scheduling the accesses to the PFS [10], [11], [12], burst-buffers [13] and I/O-aware batch scheduling [14], [15].

However, these efforts usually see the shared I/O infrastructure as a single resource capable of a certain bandwidth, whereas in practice it is a distributed set of resources from which each application can use a subset. In addition, using X% of the OSTs, for example, does **not** grant a job X% of the PFS' peak performance [16], [17]. Indeed, as we discuss in Section II, depending on their characteristics, each application will be impacted differently by the number of used I/O resources [18], [19], [20].

In this paper, we present a comprehensive study of the problem of scheduling shared I/O resources — I/O nodes, OSTs, etc — to HPC applications with the goal of mitigating contention and improving I/O and system performance. We tackle this problem by proposing heuristics to answer two questions: 1) *how many* resources should we give each application (allocation heuristics), and 2) *which* resources should be given to each application (placement heuristics). These questions are *not* independent, as using more resources often means sharing them. Nonetheless, our two-step approach allows for simpler heuristics that would be usable in practice. Moreover, it allows for studying the impact of these two steps separately.

In addition to overhead, an important aspect that impacts how "implementable" algorithms are is their input regarding applications' characteristics, since this information is often not available or at least imprecise. Therefore, we propose heuristics that use different input and study their robustness to inaccurate information. Our main contributions are:

- We propose allocation and placement algorithms, which work in two steps to schedule I/O resources among concurrent applications.
- We extensively evaluate our approach and compare it to the state of the art using two case studies: scheduling of I/O nodes and of OSTs.
- We discuss the quality of various input parameters and study their impact with the goal of answering what information is important from applications, and how robust the heuristics are to inaccurate information.

The rest of this paper is organized as follows: Section II further motivates this work by providing background on the relationship between the number of I/O resources and per-

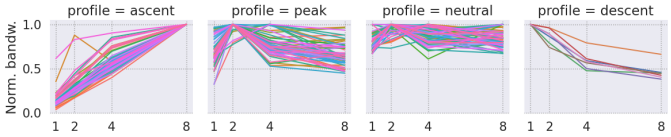


Fig. 1: Normalized bandwidth as a function of the number of I/O nodes for 189 different applications, grouped by behavior. Data from [18].

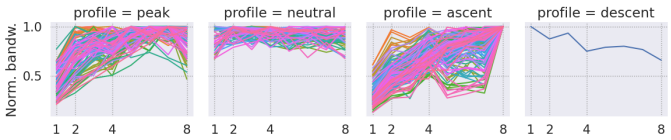


Fig. 2: Normalized bandwidth as a function of the number of OSTs for 301 benchmark configurations, grouped by behavior. Section V-A details how these results were obtained.

formance. Section III formally states the studied problem, and then Section IV discusses the proposed heuristics. The evaluation methodology is detailed in Section V, and results in Section VI. Section VII discusses related work, and Section VIII concludes this paper.

II. Motivation

For both I/O nodes and OSTs, I/O performance depends on the number of I/O resources, and the impact of the latter on the former depends on their access pattern. In the case of I/O nodes, that was shown to be the case by Bez et al. [18]. We plotted data from their work in Figure 1, where we see four different behaviors: increasing the number of I/O nodes can have no impact on performance, improve it, decrease it, or increase it until a point from where having more I/O nodes starts to harm it.

Boito et al. [17] studied the impact of the number of OSTs on performance and concluded that the more, the better. However, their analysis was limited to a single access pattern. A previous study by Chowdhury et al. [16] with the same file system, but using other applications, concluded the number of OSTs had little or no impact. On the other hand, Xu et al. [21] concluded that for collective reads using more OSTs actually *degraded* performance. Figure 2 shows our own results, described in Section V-A.

In practice, file systems are configured with a fixed number of OSTs per file, which is typically rather small to minimize sharing [22], [16]. Moreover, in most systems the number of I/O nodes assigned to a job, either depends on its number of compute nodes, which is not necessarily linked to its I/O behavior, or in some cases is a fixed number whatever the size of the job (it used to be seven on Theta). In all cases, sharing I/O resources is generally considered something to be avoided, as it may harm performance. For example, Bez et al. [23] proposed an algorithm to schedule I/O nodes to jobs which aims at providing exclusive access to the applications with higher I/O performance.

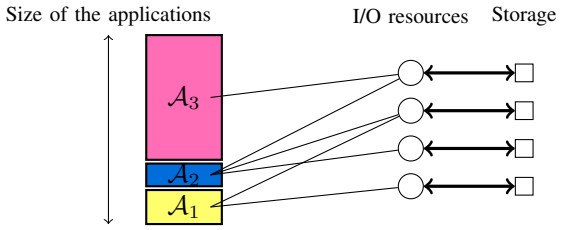


Fig. 3: Representation of the architecture: three applications $\mathcal{A}_1$ (resp. $\mathcal{A}_2$, $\mathcal{A}_3$) use two (resp. three, one) I/O resources. The number of I/O resources is not necessarily correlated to the size (number of compute nodes) of the application.

Nonetheless, over the years studies of the HPC I/O workload have consistently pointed to a bursty behavior, with most applications actually spending only a small portion of their time on I/O operations [24], [25], [4], [26], [27]. That means an exclusive allocation of I/O resources to applications, if made for their whole execution, would not be the most efficient approach. The alternative would be to dynamically change the scheduling of I/O resources whenever application behavior changes. However, that would require a quickly available characterization of current behavior, which is in general not available, and fast implementation of the calculated schedule. Since the behavior may change very frequently for each job, the desired granularity may be only a few milliseconds. For these reasons, in this paper we focus on algorithms (presented in Section IV) that make decisions for entire executions and that do *not* focus on exclusive access, as we believe this approach is more suitable for use in practice.

III. Model

This section describes the platform (shown in Fig. 3) and application models we use in this study.

A. Platform model

We assume that we have a parallel platform composed of compute nodes and remote shared storage. To access this storage, each application must communicate first through distributed I/O resources. There are N I/O resources. We consider that access to compute nodes are exclusive (congestion-free), hence in this work we focus on a performance model for I/O.

Capacity sharing: When multiple applications access concurrently an I/O resource, they share equally its capacity. In other words, the bandwidth of each I/O resource is divided by the number of applications using it at this time.

B. Application model and I/O behavior

K applications run concurrently on the platform. Each application is a series of phases that alternate between computation and I/O subphases [28], [29], [10]. We study the synchronous I/O case where compute and I/O subphases cannot overlap. The key parameters describing an application and used in the rest of this paper are summarized in Table I.

The length of each I/O subphase depends on the number of I/O resources allocated to application $\mathcal{A}_j$. The bandwidth

TABLE I: Key parameters for application $\mathcal{A}_j$.

Q_j	Number of compute resources
p_j	Number of phases (compute then I/O)
$t_{\text{cpu}}^{(i)}$	Length of the compute subphase of phase $i \leq p_j$
$v_{\text{io}}^{(i)}$	Volume of I/O transferred in phase $i \leq p_j$
T_{cpu}^{j}	Accumulated compute time over all compute (sub)phases
V_{io}^{j}	Accumulated volume of I/O over all I/O (sub)phases
b_j	I/O bandwidth as a function of the number of I/O resources

Fig. 4: Independence of I/O transfers with $n = 2$

of $\mathcal{A}_j$ as a function of I/O resources is given by: $n \mapsto b_j(n)$. Thus, if $\mathcal{A}_j$ uses n_j I/O resources, when there is no congestion, its aggregated I/O time (for amount of data V_{io}) is:

$$\text{T\_cf}_{\text{io}}(n) = \frac{V_{\text{io}}}{b_j(n)}$$

In the rest of this paper, for clarity and when there is no ambiguity, we remove the index j when talking about an application variable.

When there is no congestion, during a total time $T_{\text{cpu}}^{j} + \text{T\_cf}_{\text{io}}^{j}(n)$, $\mathcal{A}_j$ occupies n I/O resources during a time $\text{T\_cf}_{\text{io}}^{j}(n)$, and we have:

$$\texttt{I/O-Stress}(i,n) = \frac{n \cdot \text{T\_cf}_{\text{io}}^{i}(n)}{T_{\text{cpu}}^{i} + \text{T\_cf}_{\text{io}}^{i}(n)}$$

1) Characteristic values: For application $\mathcal{A}_j$, we define two characteristic values n_{perf}^{j} and n_{sys}^{j} as:

$$n_{\text{perf}}^{j} = \text{argmin}_n \text{T\_cf}_{\text{io}}^{j}(n) = \text{argmax}_n b_j(n) \quad (1)$$

$$n_{\text{sys}}^{j} = \text{argmin}_n \texttt{I/O-Stress}(j,n) \quad (2)$$

n_{perf}^{j} (Eq. (1)) corresponds to the number of I/O resources that minimizes the I/O transfer time of $\mathcal{A}_j$, and n_{sys}^{j} (Eq. (2)) corresponds to the number of I/O resources that minimizes the stress (`I/O-Stress`) on the system due to $\mathcal{A}_j$.

Lemma 1. *For all applications, $n_{sys} \leq n_{perf}$.*

(The proof is omitted for space limitations.)

2) Independence of I/O transfers: An I/O subphase over n I/O resources can be seen as n independent I/O transfers. This means that if an application is using multiple I/O resources, and its performance is slowed-down on one of those, then the other transfers are *not* slowed down. However, an I/O subphase only ends when all its I/O transfers are over (See Fig. 4).

C. Measuring performance

In this Section, we start by describing a solution (Section III-C1), before presenting how the performance of various solutions are measured (Section III-C2).

1) Defining a schedule: A solution has two elements:

- the number of I/O resources each application uses ($\pi = (n_1, \ldots, n_K)$);
- a mapping of the applications over the I/O resources.

At a given time t, and given a schedule $\pi = (n_1, \ldots, n_K)$, we define its I/O load as:

$$\texttt{I/O-load}(\pi) = \frac{1}{N} \sum_{i=1}^{K} \texttt{I/O-Stress}(i, \pi(i)) \quad (3)$$

Intuitively, this is a lower bound on the expectation of I/O time occupied by π per unit of time. By definition, `I/O-load` is minimized for the schedule $\pi_{\text{sys}} = (n_{\text{sys}}^{1}, \ldots, n_{\text{sys}}^{K})$.

If `I/O-load` > 1, then the system is *saturated*. In such a scenario, typical I/O time needed by applications exceeds system capabilities.

2) Measuring a schedule performance: Given a schedule π, we define several optimization criteria to evaluate it.

`Mean-I/O-SlowDown`: Assume application $\mathcal{A}_j$ transferred an amount of data V in total I/O time T using n_j I/O resources, then we call its `I/O-SlowDown` ρ_j the ratio of time taken to perform its I/O operations compared to the time they take when running in isolation using the number of I/O resources that minimizes this time (n_{perf}^{j}).

$$\rho_j = T / \left(V / b_j(n_{\text{perf}}^{j}) \right)$$

The `Mean-I/O-SlowDown` is the average of these values:

$$\texttt{Mean-I/O-SlowDown} = \frac{\sum_{j \leq K} \rho_j}{K} \quad (4)$$

To have a better qualitative understanding of the `I/O-SlowDown` and `Mean-I/O-SlowDown`, we can separate them into two components: one for the slowdown caused by not using the number of I/O resources that minimizes I/O time (ρ_j^{io}), and another for the slowdown due to congestion (ρ_j^{con}):

$$\rho_j = \rho_j^{io} + \rho_j^{con} \qquad \text{where } \rho_j^{io} = b_j(n_{\text{perf}}^{j}) / b_j(n_j)$$

I/O occupancy: Given a schedule π, the `I/O-res-occ` O_i of an I/O resource i is the measure of the proportion of time this resource is occupied performing I/O operations. Given this value, we can define the spread of `I/O-res-occ`, as it represents the load imbalance over the different I/O resources:

$$\texttt{I/O-spread} = \max_i O_i - \min_i O_i \quad (5)$$

Machine utilization: The previous metrics focus on I/O performance. Nonetheless, from a system administrator perspective, it may be interesting to favor I/O of applications that use more compute nodes/cores in order to improve the utilization of these resources. We define hence `Machine-Idletime` to represent the proportion of time when the compute nodes are *not* being use for computing. Let L^i be the total I/O time of application $\mathcal{A}_i$, in a time interval

$[0, t]$ and exclusively using Q_i of the Q^{cpu} available compute resources. Then:

$$\texttt{Machine-Idletime} = \frac{\sum_{i=1}^{K} L^i.Q_i}{t \cdot Q^{cpu}}$$

IV. Algorithms for scheduling I/O resources

As previously mentioned (Section III-C1), a solution is defined by answers to two questions: the number of I/O resources each application will use and the mapping of applications over the multiple I/O resources. In this section we present heuristics to provide the answers - respectively *allocation* (Section IV-A) and *placement* (Section IV-B) algorithms.

While an application is accurately described by the elements in Table I, in practice this data can be hard to collect and inaccurate. Thus, in Section IV-C we further discuss the input required by the different heuristics.

A. Allocation algorithms

We compare five allocation policies, detailed below.

- **Random**: each application receives a randomly picked number of I/O resources. This serves as a baseline.
- **Static:** application $\mathcal{A}_j$ receives $\frac{Q_j \cdot N}{Q^{cpu}}$ I/O resources, rounded to the closest positive integer. In other words, a number that depends only on the number of used compute resources. For the case of I/O nodes, this policy represents what happens in HPC systems where the mapping from compute nodes to them is static.
- **BestBdw** allocates n^j_{perf} (Eq. (1)) to each $\mathcal{A}_j$, i.e., the number of I/O resources that minimizes its I/O time. On the one hand, this policy minimizes ρ^{io}. However, in some cases it may increase `I/O-load` (Eq. (3)) and make applications share more I/O resources, which leads to more congestion and hence to an increased ρ^{con}.
- **Nsys-Allocator** gives π_{sys}, i.e., it allocates n^j_{sys} (Eq. (2)) to each $\mathcal{A}_j$ to minimize the `I/O-load`.
- **TCPU-Allocator**, detailed in Algorithm 1, starts at π_{sys} and then repeatedly increases the number of I/O resources of the application that maximizes the utilization of compute resources (the sum of `CPULoad`) while respecting the constraint that `I/O-load` must be smaller or equal to 1 (so the I/O system is not saturated).

$$\texttt{CPULoad}(i, n) = Q_i \cdot \frac{T^i_{\text{cpu}}}{T^i_{\text{cpu}} + \text{T\_cf}^i_{\text{io}}(n)}$$

TCPU-Allocator aims at being a compromise between BestBdw and Nsys-Allocator. Note than when the `I/O-load` of π_{perf} is below 1, then it behaves as BestBdw.

B. Placement algorithms

Three placement algorithms are considered.

- **Random-Placement** randomly assigns I/O resources to applications. This policy reflects what happens in practice in many HPC systems, where I/O behavior is not taken into consideration for placement.

Data: K applications
Result: An allocation π

```
π ← initialized as π_sys;
done ← False;
while not done do
    π̃ ← π;
    loadDiff ← an array of size K, filled with -1;
    initIOLoad ← I/O-load(π);
    for i from 1 to K do
        n ← π(i);
        while n ≠ n^i_perf & loadDiff(i) < 0 do
            n ← n + 1;
            if initIOLoad + (I/O-Stress(i, n)/N −
              I/O-Stress(i, π(i))/N) ≤ 1 then
                loadDiff(i) ←
                  CPULoad(i, n) − CPULoad(i, π̃(i)) ;
                π̃(i) ← n;
    idx ← argmax(loadDiff);
    if loadDiff(idx) < 0 then
        done ← True;
    else
        π(idx) ← π̃(idx);
return π;
```

Algorithm 1: TCPU-Allocator

- **Greedy-Non-Clairvoyant** aims at providing a balanced number of applications per I/O resource. It sorts applications by decreasing number of I/O resources (computed by the allocation algorithm), then it places each of them going over the I/O resources in a round-robin fashion.
- **Greedy-Clairvoyant** strives for a balanced load across the I/O resources. It sorts applications by decreasing congestion free I/O ratio $\text{T\_cf}_{\text{io}}/(T_{\text{cpu}} + \text{T\_cf}_{\text{io}})$, then it greedily places them on the I/O resources with the lowest `I/O-res-occ`.

C. On the difficulty of instantiating an algorithm

The eight described algorithms use different information about applications, as summarized in Table II. The colors represent how easy to obtain we consider these values to be:

TABLE II: Heuristics and their input

	Q_j	V^J_{io}	T^J_{cpu}	`argmax` b_j	b_j
Random					
Static	x				
BestBdw				x	
Nsys-Allocator		x	x		x
TCPU-Allocator	x	x	x		x
Random-Placement					
Greedy-Non-Clairvoyant					
Greedy-Clairvoyant		x	x		x

- Easy (green): the number of compute resources used by each application can be obtained from the resource man-

ager. Similarly to the total number of available compute and I/O resources, it is easily obtained and reliable.
- Medium (orange): aggregated information such as the total amount of transferred data and compute time of an application can be obtained from previous runs, for example with profiling tools such as Darshan [25], or provided by the user. In both cases, the actual observed values could vary and this data is only semi-reliable. Note that n_{perf} , used by BestBdw, is considered in this category because it does not require the whole curve and bandwidth values, only the number of I/O resources that maximizes performance.
- Hard (red): for an application $\mathcal{A}_j$, obtaining the bandwidth as a function of the number of I/O resources (b_j) requires multiple previous runs and is naturally sensitive to variability. The system could accumulate this information over time (so it would only be available to *some* of the running applications), or the user could provide it (less reliable).

Still, rather than using the exact b_j for each application, we believe a more realistic alternative is to approximate it by a general behavior (e.g. the profiles seen in Figure 1). Given an application general I/O characteristics, such as read/write ratio, request size, etc. (medium difficulty in our classification above), it could be matched to a benchmark for which the profile is known [30]. We explore this approach, and the robustness of the studied heuristics, in Section VI-B.

V. Evaluation methodology

The evaluation in this paper relies on data from real executions, described in Section V-A, and simulation of HPC systems, detailed in Section V-B. Sections V-C and V-D describe the workload generation and evaluation protocols.

A. Datasets

We used two sets of data, each covering multiple applications and, for each $\mathcal{A}_j$ its b_j, i.e., bandwidth measurements for different numbers of I/O resources. The first, obtained for 189 applications at the MareNostrum supercomputer and shown in Fig. 1, was made publicly available by Bez et al. [18] and is used to cover the use case of scheduling I/O nodes. For the second use case (OST scheduling), we generated a similar dataset by running the IOR benchmark with different configurations in the PlaFRIM experimental platform, using numbers of OSTs for BeeGFS varying from 1 to 8. The I/O infrastructure of this platform has been detailed in [17]. The 301 configurations all write to a shared file, while covering various values for numbers of nodes, processes per node, request size, contiguous vs. 1D-strided file layout, and total amount of data. These results were shown in Fig. 2.

B. Simulator design

We developed a time-based simulator, available at https://gitlab.inria.fr/abandet/shared_IO_simulator along with all the data to reproduce our results. It consists of three phases:

1) the workload generation, described in Section V-C;
2) the execution of the algorithms presented in Section IV. Note that we also implemented MCKP from the state of the art in I/O nodes scheduling [23]. It performs both steps (allocation + placement).
3) The evaluation of the results according to the objectives presented in Section III-C2. We detail how these objectives are measured in Section V-D.

At each unit of time, the simulator examines the status of each application (compute or I/O). When there is a *collision* on an I/O resource (i.e. two or more applications try to do I/O), the colliding applications will alternate in performing I/O for one unit of time each until their I/O subphases are over. This approximates the fair-share I/O scheduling algorithm because all simulations are performed for thousands of units of time.

C. Workload generation protocol

We make the hypothesis that the algorithms' behavior depends on the I/O stress on the system. Hence in our generation of the application sets, we cover various `I/O-load`.

In all the experiments we consider that we have $N = 20$ I/O resources, and $Q^{cpu} = 480$ compute resources. The number of applications K depends on the experiment, so does the target `I/O-load`: Θ. Then, given K and Θ, for each application $\mathcal{A}_j$ we generate it as follows:

- we pick an I/O bandwidth profile uniformly at random in the used data set (Section V-A);
- the number of phases p_j is picked uniformly at random in $\{2, 3, 4, \cdots, 20\}$;
- Q_j is computed so that 10% (resp. 30%, 60%) of applications use 75% (resp. 20%, 5%) of the compute resources (large, medium, and small applications);
- for all applications, we set T^j_{cpu}+T_cf$^j_{\text{io}}(1) = 5000$s (common horizon). Then, we set T^j_{cpu}/T_cf$^j_{\text{io}}(1) = X$, where X is picked uniformly at random in $[0, b]$. The bound b is computed so that $\mathbb{E}\left(\texttt{I/O-Stress}(j, 1)\right) = \frac{\Theta N}{K}$: b is the solution to $\log(1+b) = b\frac{\Theta N}{K}$ (computed analytically). Consequently, we have

$$V^j_{\text{io}} = 5000 \cdot b_j(1)/(1 + X)$$
$$T^j_{\text{cpu}} = 5000\,(1 - 1/(1 + X))$$

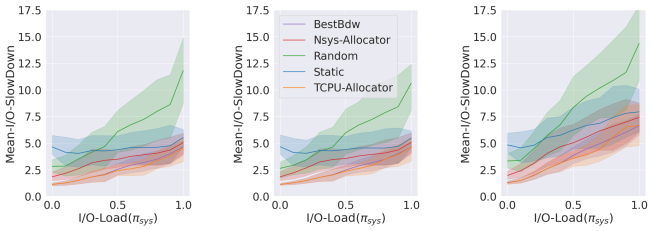
By construction, this generation has the following property:

$$\texttt{I/O-load}(\pi_1) = \frac{1}{N}\sum_{j=1}^{K} \frac{\text{T\_cf}^j_{\text{io}}(1)}{T^j_{\text{cpu}} + \text{T\_cf}^j_{\text{io}}(1)} = \Theta$$

In a second step and for the evaluation, we categorize the sets of application that we have generated by their minimum `I/O-load`, i.e., $\texttt{I/O-load}(\pi_{\text{sys}}) \leq \Theta$.

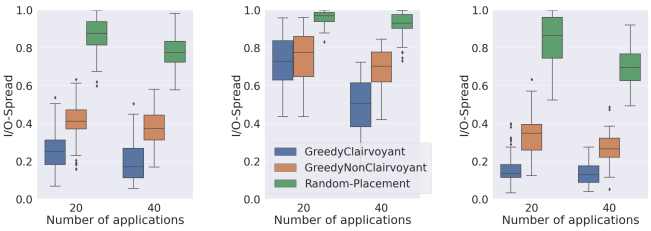
D. Evaluation protocol

Each studied scenario is evaluated with over 100 different application sets. The metrics are measured on an interval where the state of the system is constant (i.e. no application finishes), but of a sufficient length (i.e. each application should have performed at least a complete phase compute+I/O).



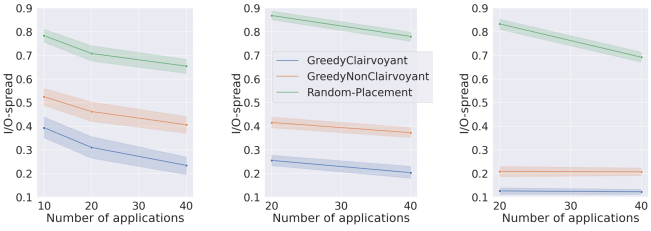
(a) Greedy-Clairvoyant (b) Greedy-Non-Clairvoyant (c) Random-Placement

Fig. 5: `Mean-I/O-SlowDown` for different combinations of algorithms at increasing `I/O-load`(π_{sys}). Lines show the mean value, and the area around them is the percentile interval (10^{th}-90^{th}).



(a) BestBdw (b) Nsys-Allocator (c) Random

Fig. 6: `I/O-spread` for different placement and allocation algorithms when `I/O-load`(π_{sys}) $= 0.5 \pm 0.05$.



(a) `I/O-load`(π_{sys}) $= 0.2 \pm 0.05$ (b) `I/O-load`(π_{sys}) $= 0.5 \pm 0.05$ (c) `I/O-load`(π_{sys}) $= 0.8 \pm 0.05$

Fig. 7: **`I/O-spread`** for different placement algorithms using BestBdw. Lines connect the mean values, and the area around each line shows the 95% confidence interval.

VI. Results

In this Section we provide various elements to compare the different heuristics. We focus the discussion on the impact of the input that each algorithm considers. Specifically:

- in Section VI-A, we compare the algorithms in a setup where we trust their inputs;
- then, in Section VI-B, we loosen the quality of the input information to study the robustness of our algorithms;
- finally, after using the I/O nodes case study in all previous sections, in Section VI-C we show that the results hold with another bandwidth model (the OST case study).

A. Evaluation with accurate input data

In this Section, we evaluate the different solutions based on the optimization criteria presented in Section III-C. As already discussed, the comparison between the algorithms is performed as a function of `I/O-load`(π_{sys}), which corresponds to the level of stress on the I/O system.

1) User observed performance: In Fig. 5, we present `Mean-I/O-SlowDown` (Eq. (4)) for all combinations of placement and allocation algorithms when `I/O-load`(π_{sys}) varies. We can make several key observations:

- In general, the naive algorithm that allocates the number of I/O resources that maximizes I/O bandwidth to each application (BestBdw) is the best allocation strategy.
- On the other hand, when `I/O-load` is large Nsys-Allocator performs better than BestBdw because the latter tends to increase the stress on the system.
- Using the Greedy-Clairvoyant placement strategy has negligible gain on `Mean-I/O-SlowDown` compared to Greedy-Non-Clairvoyant, which requires much less information.

These observations show that choosing either Greedy-Clairvoyant or Greedy-Non-Clairvoyant for allocation has little impact on performance (but they do have an impact compared to Random-Placement). That shows that efficient I/O scheduling can be done with limited information.

Note that Greedy-Clairvoyant and Greedy-Non-Clairvoyant have a different behavior. We can verify this by studying the `I/O-spread`, which represents the load *imbalance* between I/O resources (Eq. (5)). We show in Fig. 6 the `I/O-spread` for the three placement algorithms with several allocation algorithms. This confirms that Greedy-Clairvoyant balances I/O better over the I/O resources. Greedy-Non-Clairvoyant, which balances the number of applications without considering the load/stress they impose, still balances I/O better than Random-Placement. That happens because occupancy is correlated to the number of applications. In Fig. 7, we confirm that this holds for different values of `I/O-load`.

In the rest of this section, we only use Greedy-Non-Clairvoyant as a placement algorithm.

To compare the allocation heuristics, we decompose the `I/O-SlowDown` into their I/O and congestion components (as discussed in Section III-C2) in Fig. 8 and 9. In the first we show two ranges of `I/O-load`(π_{sys}) (low and high), and in the second we show the tendencies when it varies.

In Fig. 8, we can observe that, as by design, BestBdw decreases the portion of I/O time due to I/O resources allocation to its bare minimum, at the cost of a much higher congestion overhead (ρ_{con}) than that of Nsys-Allocator or Static. As `I/O-load`(π_{sys}) increases (Fig. 9), this cost increases linearly and may become a problem at very high `I/O-load` values. On the contrary, the congestion overhead with allocation algorithms that take into account the system `I/O-load` (Nsys-Allocator and TCPU-Allocator) do not vary as much when `I/O-load`(π_{sys}) increases. Nonetheless, that comes at the

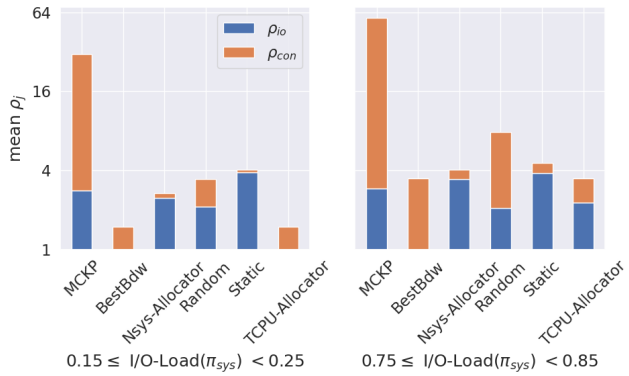


Fig. 8: `Mean-I/O-SlowDown` and `Max-I/O-SlowDown` for allocation algorithms with two ranges for `I/O-load`(π_{sys}). The y axes are in log scale and start at 1 (minimum for ρ_j).

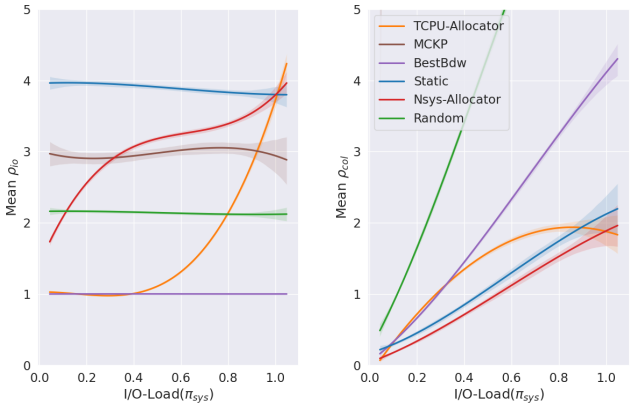


Fig. 9: `I/O-SlowDown` separated into its two causes: I/O resources allocation ρ_{io} and congestion ρ_{con}. The lines show the polynomial regression, and the area around them the 95% confidence intervals.

cost of increased I/O time due to I/O resources allocation. Finally, MCKP performs worse than all the other studied policies for these objectives. Indeed, it seeks to optimize the sum of applications' bandwidths due to I/O resources allocation, and has hence a tendency of favoring a few applications (the highest bandwidth ones) in detriment of others.

2) Machine utilization: Up to now we focused on the average I/O behavior, a user-oriented objective. This disadvantages more unfair algorithms such as Static, which gives more I/O resources to applications that occupy a larger fraction of the compute resources, possibly in detriment of smaller applications that are more numerous (and hence have a higher impact in a quantitative objective like `Mean-I/O-SlowDown`). From a system administrator perspective, it may be more interesting to have an application that occupies a large part of the machine to perform its I/O fast, even if it at the cost of worse performance for smaller applications.

To evaluate this, we plot the `Machine-Idletime` of the compute nodes (i.e. the time when applications are performing I/O) in Fig. 10. We can make two observations:

- At low values of `I/O-load`(π_{sys}), the relative allocation algorithm performance in terms of `Machine-Idletime` are the same as those for `Mean-I/O-SlowDown` (BestBdw and TCPU-Allocator perform the best). This is not surprising: I/O has less impact.
- However, at larger `I/O-load`(π_{sys}), we start to see a real difference of performance between TCPU-Allocator and BestBdw, even though their respective I/O performance were similar, giving an advantage to the heuristic that considers more information.

Static is an interesting heuristic: it uses little information about the applications and still can get good machine utilization results when the `I/O-load` is high.

B. Robustness to the quality of the input

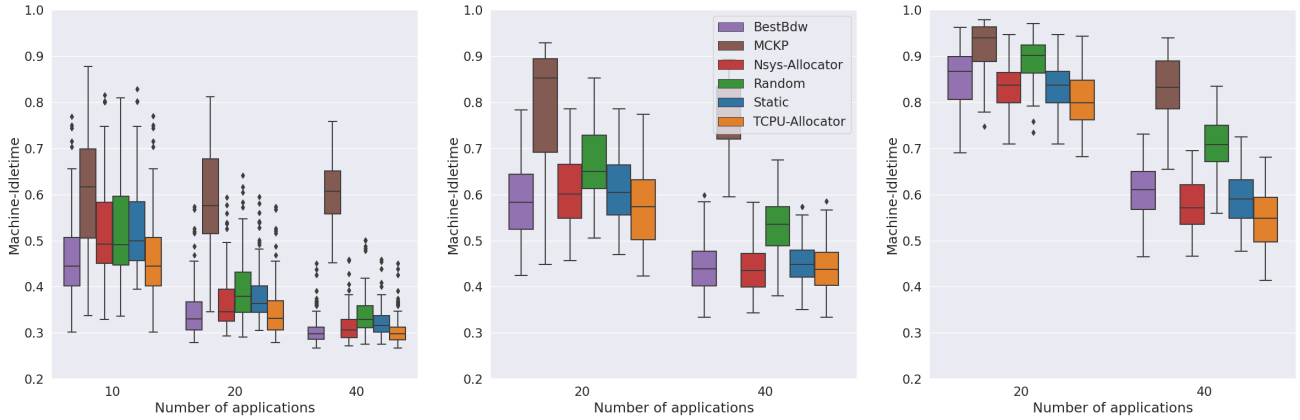
In Section VI-A, we were able to demonstrate the fact that under low `I/O-load`, BestBdw was the most efficient allocation algorithm, whereas under a heavier `I/O-load`, one should rather use TCPU-Allocator which provides the same I/O performance from an application perspective but improves on system utilization.

As shown in Section IV-C, there is however an important difference between these two algorithms: their input. In particular, TCPU-Allocator requires a full I/O profile of the applications whereas BestBdw only requires the number of I/O nodes that provides the maximum bandwidth.

In this section, we run the same simulations as before, but giving algorithms partial (and potentially wrong) information to study how robust they are. Specifically, all applications from a given I/O performance profile (Ascent, Descent, Peak, Neutral) is said to have the same bandwidth function b_j. This function is obtained by interpolating all curves within a profile to the same function (see Fig. 11).

In Fig. 12, we study the performance of three combinations of algorithms (TCPU-Allocator +Greedy-Non-Clairvoyant; BestBdw +Greedy-Non-Clairvoyant; BestBdw +Greedy-Clairvoyant) with poor input accuracy. To do so, we use all generated scenarios, and plot them as a function of `I/O-load`(π_{sys}). Note that Greedy-Non-Clairvoyant is not impacted by the lack of accuracy: its behavior does not change. Similarly, BestBdw is impacted by the lack of accuracy only for some of the applications with a bandwidth profile `Peak` and `Neutral`.

Specifically, in Fig. 12a, we compare their performance to what was observed with accurate information: if the value is 2%, for example, it means that the algorithm with inaccurate information performed 2% worse than with accurate information. We can observe that they have almost identical performance (except in scenarios with extremely low `I/O-load`). As expected, the allocation algorithm where the behavior differs the most is TCPU-Allocator which is the algorithm that requires the most information. Yet this difference is still negligible (up to 4%). With respect to



(a) `I/O-load`$(\pi_{\text{sys}}) = 0.2 \pm 0.05$ (b) `I/O-load`$(\pi_{\text{sys}}) = 0.5 \pm 0.05$ (c) `I/O-load`$(\pi_{\text{sys}}) = 0.8 \pm 0.05$

Fig. 10: `Machine-Idletime` for allocation algorithms at increasing `I/O-load`(π_{sys}). The y axes do *not* start at 0. The lower the better.

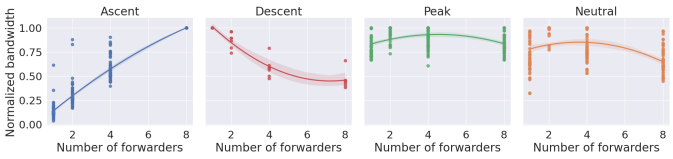


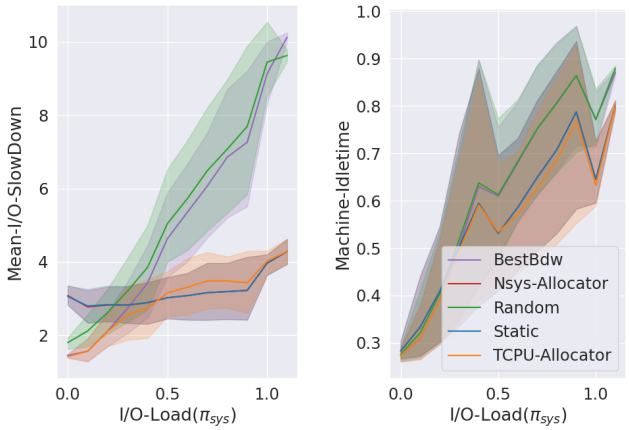
Fig. 11: Polynomial regression over all application profiles.

(a) `Mean-I/O-SlowDown` (b) `Machine-Idletime`

Fig. 12: Performance of three scheduling solutions with partial information about applications (compared to their performance with exact information for `Mean-I/O-SlowDown`).

the mapping algorithm, Greedy-Clairvoyant performs almost identically except in cases with extremely low `I/O-load`, i.e. when the `Mean-I/O-SlowDown` is close to 1, the error in prediction is most impactful.

We confirm their absolute impact by plotting their `Machine-Idletime` (Fig. 12b) which confirms that even with inaccurate information TCPU-Allocator is the best solution. More generally, we observe that the various algorithms are quite robust to some inaccuracy in I/O behavior, and that the main claims of our work with respect to choosing allocation and mapping algorithms hold.



(a) `Mean-I/O-SlowDown` (b) `Machine-Idletime`

Fig. 13: Results for the OST case study. Lines show the mean values, and the area around them the interval between the 10^{th} and 90^{th} percentiles.

C. Scheduling of OSTs

In this final set of experiments, we evaluate our algorithms on a different context: OST scheduling. The main difference with I/O nodes lies on the performance obtained depending on the number of resources allocated to the application (see Fig. 1 and 2).

Due to lack of space, we only show plots for the conclusions that differ from the previous section, i.e. we do not show the comparison between placement strategies and focus solely on the Greedy-Non-Clairvoyant placement algorithm. These results are presented in Figure 13.

Overall the main observation is the same as previously: when there is little I/O stress on the system, then BestBdw performs better than I/O stress aware algorithms such as Nsys-Allocator. Then as the stress increases, its performance quickly degrades. The main difference here is that the cut-off point is much earlier. These results confirm that TCPU-Allocator is an excellent alternative that is able to match the performance of BestBdw when needed, while being I/O stress aware.

There is another important difference with the previous behavior: Static's performance are excellent for this profile. This is interesting because i) for OSTs, Static is *not* what is typically used in practice; and ii) it is quite natural and easy to implement. Unfortunately, the fact that this behavior is highly dependent on application profile (i.e. we did not see the same behavior with the I/O nodes dataset) makes the Static allocation algorithm unreliable.

VII. Related work

A. Scheduling of I/O nodes

Yu et al. [31] document the load imbalance problem of I/O nodes and propose a strategy where they are statically assigned to applications in an exclusive way. Whenever the load of an application's I/O nodes it too high, it is allowed to temporarily use the I/O nodes assigned to others. Differently from us, they do not study the selection of *which* I/O nodes should be shared, hence our work is complementary to theirs.

In the work by Ji et al. [32], less than half of the I/O nodes are statically assigned to applications, and then historical data is used to decide if more nodes, from a pool of available ones, should be used. This decision on the number of I/O nodes each application should use is taken only based on the number of compute nodes that perform I/O operations, while it has since been shown (see Section II) that other characteristics impact b_j in a more complicated way. Their approach also allocates more I/O nodes to applications to avoid sharing them with others of incompatible access patterns. Differently, we focus on more generic placement strategies, that require less information, and our results for Greedy-Non-Clairvoyant prove how effective they can be. Our techniques can be used in the absence of detailed application information and interference models.

Bez et al. [18] propose MCKP, which does allocation and placement by optimizing the sum of applications' bandwidths and favoring exclusive access as much as possible. We argue that exclusive access may be wasteful as many applications are not I/O intensive, and instead further explore the placement aspect. In Section VI, we compare our heuristics to MCKP.

B. Scheduling of OSTs

Yang et al. [33] reduce the problem of placing applications on I/O nodes *and* OSTs to the maximum flow problem, using their I/O load and the current monitored load of the resources. This is similar to what Greedy-Clairvoyant does, with the difference that we do not place applications on all layers of I/O resources at the same time. The strategy by Wang et al. [34] also considers all layers at once. The OST with the lowest-cost path is selected for each of the application's compute nodes. The cost of a path depends on manually-set weights given to layers according to their importance for performance. Moreover, while they focus on improving load balance in the context of each application, our approach considers a global view of all concurrent applications.

The challenge in determining the best number of OSTs for an application is often tackled in the literature by having systems try different configurations over multiple runs. For example, Kim et al. [35] propose DCA-IO to automatically tune PFS parameters. If no information is available for an application, it receives the number of OSTs historically observed to be the best for the number of compute resources it uses. On the other hand, if the application is known, it will receive more OSTs at each execution until a local maximum is found.

Another way of obtaining such information is to train models on aggregated application metrics (more easily obtained). The auto-tuning framework proposed by Behzad et al. [36] adapts the number of OSTs by keeping a database of I/O patterns, extracted from applications, and using non-linear regression models to find the best values. Agarwal et al. [37] use Bayesian optimization and a pre-trained I/O performance model to recommend the best number of OSTs to each application. These approaches could be used together with our proposed heuristics to provide the required information.

VIII. Conclusion

In this work we have investigated the problem of allocating subsets of distributed I/O resources to applications in order to optimize their I/O performance and the platform utilization.

Our contributions include both allocation and placement algorithms. In their design, we have taken into account a trade-off between simplicity and efficiency. To this regard we have shown that the placement algorithm can be quite naive: balancing the absolute number of applications per I/O resource, without considering their I/O load, leads to results as good as I/O-aware placement. This naive algorithm gives more leeway to optimize placement based on other reasons (such as proximity to the applications).

On the contrary, we have shown that the allocation algorithm is more important for I/O performance, and one should use a more fine-tuned algorithm rather than a naive approach such as peak bandwidth or a static approach that allocates a number of I/O resources proportional to the number of compute resources.

An important contribution of our work is the robustness study: indeed, I/O behavior has been shown to be quite volatile and hard to predict. Hence a very efficient heuristic that is not robust to volatility in its input can become quite useless. In this work we have studied different types of input that algorithms could use, and the limits of each algorithm based on these inputs. In addition, we have shown that our presented heuristics are robust to inaccuracy in input information. We believe that this opens avenues in terms of I/O behavior prediction which is a hard problem: indeed one may not need exact prediction for some of the I/O scheduling algorithms.

This should considerably simplify the design of I/O analysis tools.

REFERENCES

[1] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castaños, L. Ceze, P. Crumley, C. C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, "An overview of the blue gene/l system software organization," in *Euro-Par 2003 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 543–555.

[2] J. L. Bez, A. M. Karimi, A. K. Paul, B. Xie, S. Byna, P. Carns, S. Oral, F. Wang, and J. Hanley, "Access patterns and performance behaviors of multi-layer supercomputer i/o subsystems under production load," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 43–55.

[3] A. Lopez, S. Valat, S. Narasimhamurthy, and M. Golasowski, "Ephemeral data access environment: Concept and architecture," IO-SEA project public deliverable, Tech. Rep., 2022.

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

[5] 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 *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 750–759.

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

[7] E. Costa, T. Patel, B. Schwaller, J. M. Brandt, and D. Tiwari, "Systematically inferring i/o performance variability by examining repetitive job behavior," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[8] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. T. S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, and A. S. Bland, "Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 217–228.

[9] A. Gainaru, B. Goglin, V. Honoré, and G. Pallez, "Profiles of upcoming hpc applications and their impact on reservation strategies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1178–1190, 2021.

[10] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications Under Congestion," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 1013–1022, iSSN: 1530-2075.

[11] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, "Calciom: Mitigating i/o interference in hpc systems through cross-application coordination," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 155–164.

[12] F. Boito, G. Pallez, L. Teylo, and N. Vidal, "IO-SETS: Simple and efficient approaches for I/O bandwidth management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 10, pp. 2783 – 2796, Aug. 2023.

[13] G. Aupy, O. Beaumont, and L. Eyraud-Dubois, "Sizing and partitioning strategies for burst-buffers to reduce io contention," in *2019 IEEE international parallel and distributed processing symposium (IPDPS)*. IEEE, 2019, pp. 631–640.

[14] 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.

[15] 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.

[16] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/o characterization and performance evaluation of beegfs for deep learning," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19. New York, NY, USA: Association for Computing Machinery, 2019.

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

[18] J. L. Bez, F. Z. Boito, A. Miranda, R. Nou, T. Cortes, and P. O. A. Navaux, "Towards On-Demand I/O Forwarding in HPC Platforms," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, Nov. 2020, pp. 7–14.

[19] J. Yu, G. Liu, W. Dong, X. Li, J. Zhang, and F. Sun, "On the load imbalance problem of i/o forwarding layer in hpc systems," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, 2017, pp. 2424–2428.

[20] H. Devarajan and K. Mohror, "Extracting and characterizing i/o behavior of hpc workloads," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 243–255.

[21] C. Xu, S. Byna, V. Venkatesan, R. Sisneros, O. Kulkarni, M. Chaarawi, and K. Chadalavada, "Lioprof: exposing lustre file system behavior for i/o middleware," in *2016 Cray User Group Meeting*, 2016.

[22] F. Wang, H. Sim, C. Harr, and S. Oral, "Diving into petascale production file systems through large scale profiling and analysis," in *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, ser. PDSW-DISCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 37–42.

[23] J. L. Bez, A. Miranda, R. Nou, F. Z. Boito, T. Cortes, and P. Navaux, "Arbitration Policies for On-Demand User-Level I/O Forwarding on HPC Platforms," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2021, pp. 577–586, iSSN: 1530-2075.

[24] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, and D. Long, "File system workload analysis for large scale scientific computing applications," in *Proceedings of the Twenty-first Symposium on Mass Storage Systems (MSST)*, 2004.

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

[26] Z. Liu, R. Lewis, R. Kettimuthu, K. Harms, P. Carns, N. Rao, I. Foster, and M. E. Papka, "Characterization and identification of HPC applications at leadership computing facility," in *Proceedings of the 34th ACM International Conference on Supercomputing*. Barcelona Spain: ACM, Jun. 2020, pp. 1–12.

[27] W. Yang, X. Liao, D. Dong, and J. Yu, "A quantitative study of the spatiotemporal i/o burstiness of hpc application," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 1349–1359.

[28] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 Characterization of petascale I/O workloads," in *2009 IEEE International Conference on Cluster Computing and Workshops*. New Orleans, LA, USA: IEEE, 2009, pp. 1–10.

[29] 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*, vol. 27, no. 8, pp. 2435–2449, Aug. 2016.

[30] F. Z. Boito, "Estimation of the impact of I/O forwarding on application performance," p. 24, 2020.

[31] J. Yu, G. Liu, W. Dong, X. Li, J. Zhang, and F. Sun, "On the load imbalance problem of I/O forwarding layer in HPC systems," in *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, Dec. 2017, pp. 2424–2428.

[32] X. Ji, B. Yang, T. Zhang, X. Ma, X. Zhu, X. Wang, N. El-Sayed, J. Zhai, W. Liu, and W. Xue, "Automatic, application-aware i/o forwarding resource allocation," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, ser. FAST'19. USENIX Association, 2019, pp. 265–279.

[33] B. Yang, Y. Zou, W. Liu, and W. Xue, "An end-to-end and adaptive i/o optimization tool for modern hpc storage systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 1294–1304.

[34] F. Wang, S. Oral, S. Gupta, D. Tiwari, and S. S. Vazhkudai, "Improving large-scale storage system performance via topology-aware and balanced data placement," in *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, pp. 656–663.

[35] S. Kim, A. Sim, K. Wu, S. Byna, T. Wang, Y. Son, and H. Eom, "Dca-io: A dynamic i/o control scheme for parallel and distributed file systems," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 351–360.

[36] B. Behzad, S. Byna, Prabhat, and M. Snir, "Optimizing i/o performance of hpc applications with autotuning," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, mar 2019.

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