Scheduling distributed I/O resources in HPC systems

Alexis Bandet, Francieli Boito, Guillaume Pallez

Scheduling Distributed I/O Resources in HPC Systems

Alexis Bandet[1 [0009−0000−3822−4374]], Francieli Boito[1 [0000−0002−1139−0724]], and Guillaume Pallez[2 [0000−0001−8862−3277]]

[1] Univ. Bordeaux, CNRS, Bordeaux INP, Inria,
LaBRI, UMR 5800, F-33400 Talence, France
[2] Inria, Rennes, France
{alexis.bandet, francieli.zanon-boito, 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 by 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 each application, determining *how many* (allocation) and *which* (placement) resources to use. We discuss a wide range of 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 better performance, we demonstrate the robustness of our solutions in scenarios where information is limited or inaccurate. This research provides insights into the trade-offs between the depth of application characterization and the practicality of scheduling I/O resources.

Keywords: HPC, parallel I/O, parallel file system, object storage targets, I/O forwarding, scheduling, resource allocation

1 Introduction and Related Work

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. Because of the gap between processing and I/O speeds, and with processing power ever increasing, many HPC applications spend a large portion of their time on I/O. This access is often synchronous — meaning the application occupies the compute resources while waiting to complete I/O transfer. Therefore, improving I/O performance promotes a more efficient usage of the expensive and power-hungry HPC compute resources.

Parallel 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 sometimes placed between compute nodes and the PFS [1]. Both OSTs and I/O nodes are I/O resources, and it is important to notice both are potentially shared by running applications. Other shared resources include burst-buffer nodes, present in some systems [5,20]. When applications access the same I/O resources concurrently, their I/O performance can be slowed down [24], 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 [13], which makes execution time less predictable and, consequently, complicates resource management [9].

Shared I/O access scheduling Many techniques have been proposed to mitigate contention, mainly PFS access scheduling [17,15,9], burst-buffers [2] and I/O-aware batch scheduling [19,6]. However, these efforts usually see the shared I/O infrastructure as a single resource 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 [12,8]. Indeed, depending on their characteristics, each application is impacted differently by the number of used I/O resources [26,14].

Forwarding node scheduling Xu et al. [23] present an I/O infrastructure where I/O resources can be dynamically selected for each file the applications access. That selection is done based on real-time monitoring data to avoid congestion. Some machines, such as Sunway TaihuLight [18] allow for real-time reconfiguration. From a scheduling perspective, the historical approach to deal with distributed I/O resources was to use a fixed forwarding-node mapping (FFM) [21], where even though the compute nodes are connected to all forwarding nodes, the system uses an exclusive, static mapping between the forwarding and the compute nodes. This led many forwarding nodes to stay idle, while some may be saturated [18]. Recent approaches have provided a mix of exclusive access, and opportunistic use of other forwarding nodes [26,4]. In the algorithm by Bez et al. [4], the allocation and placement are obtained by optimizing the sum of applications' bandwidths and favoring exclusive access as much as possible. We use this algorithm as a comparison to our solutions. Another approach is that proposed by Ji et al. [18]. They assign statically less than half of the I/O nodes to applications. The rest of the I/O nodes are allocated based on historical data (mostly the number of compute nodes performing I/O and I/O volume).

OST scheduling Yang et al. [25] 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. We evaluate a non-exclusive allocation version of this approach. The strategy by Wang et al. [22] 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.

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. Our main contributions are:

- We accurately model the problem of scheduling distributed I/O resources, and propose allocation and placement algorithms.
- An important aspect, which impacts how "implementable" algorithms are, is their requirements input-wise. Indeed this information is often not available or at least imprecise. We discuss the quality of various input parameters and study their impact with the goal of answering questions about the importance of accurate application description, and how robust the heuristics are to inaccurate information.

The rest of this paper is organized as follows: Section 2 formally states the studied problem, and then Section 3 discusses the proposed heuristics. We then evaluate these results in Section 4. We provide concluding remarks in Section 5.

2 Model

This section describes our platform (shown in Fig. 1) and application models.

2.1 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, which can be I/O nodes, OSTs, burst buffer nodes, etc. 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.

2.2 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 [11,16,17]. 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 1.

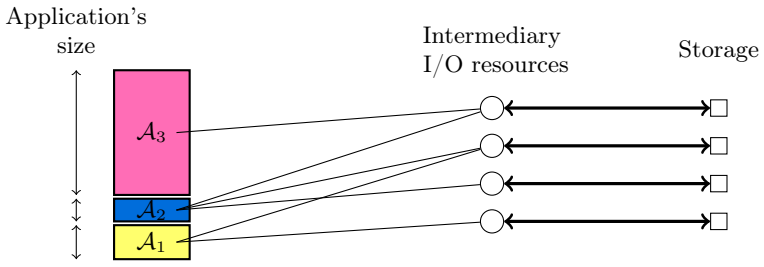


Fig. 1: Representation of the architecture: three applications $\mathcal{A}_1$ (resp. $\mathcal{A}_2$, $\mathcal{A}_3$) use two (resp. three, one) I/O resources. The number of I/O resources is not necessarily correlated to the size (number of compute nodes) of the application.

Table 1: 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

The length of each I/O subphase depends on the number of I/O resources allocated to application $\mathcal{A}_j$. The bandwidth of $\mathcal{A}_j$ as a function of I/O resources is given by: $n \mapsto b_j(n)$. 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: $\text{T\_cf}_{\text{io}}^j(n) = \frac{V_{\text{io}}}{b_j(n)}$.

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:

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

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.

Characteristic values For application $\mathcal{A}_j$, we define two characteristic values:

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

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

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

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. 2).

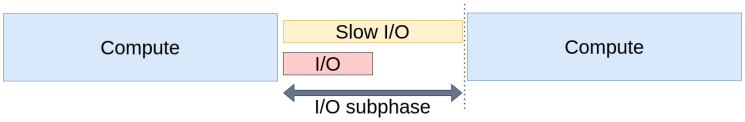


Fig. 2: Independence of I/O transfers with $n = 2$

2.3 Measuring performance

More details on the evaluation objectives are given in the companion report [3].

A solution is described by 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 π, we define its I/O load as:

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

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 $\pi_{\text{sys}} = (n^1_{\text{sys}}, \ldots, n^K_{\text{sys}})$. If `I/O-load` > 1, then the system is *saturated*. In such a scenario, typical I/O time needed by applications exceeds system capabilities.

We define the following optimization criteria to evaluate a schedule π:

I/O performance The first objective, ***Mean-I/O-SlowDown***, measures the I/O performance of the system from an application perspective: it aims at answering the question (*on average, how reduced was my I/O bandwidth?*). To have a better qualitative understanding of the `Mean-I/O-SlowDown`, we measure separately the speed reduction (slowdown) caused by not using the number of I/O resources that minimizes I/O time (ρ^{io}_j), and due to congestion (ρ^{con}_j).

Machine utilization The second objective, ***Machine-Idletime***, measures the proportion of time when the compute nodes are *not* being used for computing. Intuitively, it focuses from a system administrator perspective, where a fair treatment of applications (from an I/O perspective) may forget the fact that not all applications use the machine similarly.

3 Algorithms for scheduling I/O resources

As mentioned in Section 2.3, a solution is defined by 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 3.1) and placement (Section 3.2) algorithms. While an application is accurately described by the elements described in Section 2, in practice this data can be hard to collect and inaccurate. Thus, in Section 3.3 we further discuss the input required by the different heuristics.

3.1 Allocation algorithms

We propose five allocation policies:

- **R*andom***: each application receives a randomly picked number of I/O resources. This serves as a baseline.
- **S*tatic*:** application $\mathcal{A}_j$, running on Q_j compute nodes (out of Q^{cpu} in the system) receives $\frac{Q_j \cdot N}{Q^{cpu}}$ I/O resources, rounded to the closest positive integer. 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.
- **B*estBdw* (BBA)** allocates n^j_{perf} to each $\mathcal{A}_j$, i.e., the number of I/O resources that minimizes its I/O time. While this policy minimizes ρ^{io}, in some cases it may increase `I/O-load` and make applications share more I/O resources, which leads to more congestion and hence to an increased ρ^{con}.
- **N*sys-Allocator* (NSYSA)** gives π_{sys}, i.e., it allocates n^j_{sys} to each $\mathcal{A}_j$ to minimize the `I/O-load`.
- **T*CPU-Allocator* (TA)**, 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).

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

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

3.2 Placement algorithms

Three placement algorithms are considered.

- **R*andom-Placement* (RandP)** 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
$\pi \leftarrow$ initialized as π_{sys};
done $\leftarrow$ False;
while *not done* **do**
  $\tilde{\pi} \leftarrow \pi$;
  loadDiff $\leftarrow$ an array of size K, filled with -1;
  initIOLoad $\leftarrow$ `I/O-load`(π);
  for *i from 1 to K* **do**
    $n \leftarrow \pi(i)$;
    while $n \neq n^i_{perf}$ & *loadDiff*$(i) < 0$ **do**
      $n \leftarrow n + 1$;
      if *initIOLoad* $+ (\texttt{I/O-Stress}(i,n)/N - \texttt{I/O-Stress}(i,\pi(i))/N) \leq 1$ **then**
        loadDiff$(i) \leftarrow \texttt{CPULoad}(i,n) - \texttt{CPULoad}(i,\tilde{\pi}(i))$;
        $\tilde{\pi}(i) \leftarrow n$;
  idx $\leftarrow$ argmax(loadDiff);
  if *loadDiff*$(idx) < 0$ **then**
    done $\leftarrow$ True;
  else
    $\pi(idx) \leftarrow \tilde{\pi}(idx)$;
return π;

Algorithm 1: TCPU-Allocator (TA)

- **G*reedy-Non-Clairvoyant* (GNC)** 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.
- **G*reedy-Clairvoyant* (GC)** 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 the least stressed.

3.3 On the difficulty of instantiating an algorithm

The eight described algorithms use different information about applications, as summarized in Table 2. The colors represent how easy to obtain we consider these values to be:

- Easy (green): the number of compute resources used by each application can be obtained from the resource manager. 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 [10], or provided

Table 2: Heuristics and their input

		Allocation					Placement		
		Random	Static	BBA	NSYSA	TA	RandP	GNC	GC
Easy	Q_j		x			x			
Medium	V_{io}^j				x	x			x
	T_{cpu}^j				x	x			x
	n_{perf}			x					
Hard	b_j				x	x			x

by the user. In both cases, the actual observed values could vary and this data is only semi-reliable. n_{perf}, is considered in this category because it does not require the whole evaluation and bandwidth values.

– 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).

4 Evaluation

4.1 Evaluation methodology

The evaluation in this paper relies on data from real executions. The evaluation is done on a time-based simulator, available at `https://gitlab.inria.fr/hpc_io/ionode_simulator` along with instructions to reproduce our results.

Datasets We use two sets of data with several applications $\mathcal{A}_j$ and b_j, i.e., bandwidth measurements for different numbers of I/O resources.

1. Data on 189 applications on the MareNostrum supercomputer was made publicly available by Bez et al. [4]. We use it for the use case of scheduling I/O nodes.
2. The second dataset (use case of OST scheduling) [7] is generated 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 [8]. 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.

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` values.

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. Given K and Θ, to generate $\mathcal{A}_j$:

- we pick an I/O bandwidth profile uniformly at random in the dataset;
- 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 $T^j_{\text{cpu}} + \text{T_cf}^j_{\text{io}}(1) = 5000$s (common horizon). Then, we set $T^j_{\text{cpu}}/\text{T\_cf}^j_{\text{io}}(1) = X$, 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:

$$V_{\text{io}}[j] = 5000 \cdot b_j(1)/(1 + X); \quad T^j_{\text{cpu}} = 5000\,(1 - 1/(1 + X))$$

By construction, this generation has the property $\texttt{I/O-load}(\pi_1) = \Theta$.

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

Evaluation protocol Each studied scenario is evaluated with over 100 different application sets, randomly generated as described above. 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). In addition to our solutions, we compare to the MCKP algorithm [4].

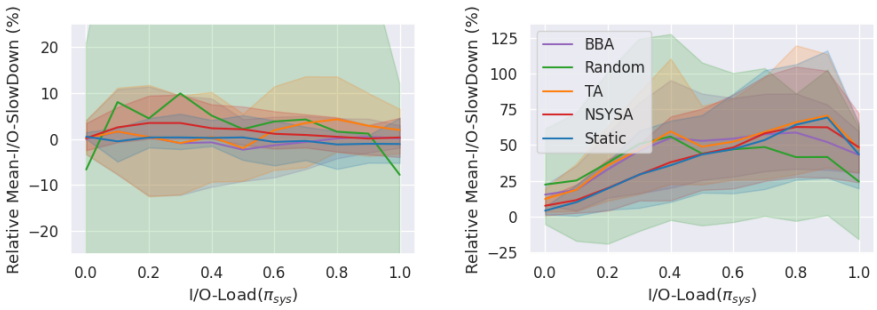
4.2 Results

In this section we provide various elements to compare the different heuristics. We first compare the algorithms in a setup where their inputs are reliable, then we loosen the quality of the input information to study the robustness of our algorithms; finally, we discuss the fact that the results hold with another bandwidth model.

To compare the algorithms, we study the optimization criteria presented in Section 2.3. The comparison between the algorithms is performed when $\texttt{I/O-load}(\pi_{\text{sys}})$ varies. This value corresponds to the level of stress on the I/O system: we hypothesize that this is what impacts the most the performance of the algorithms in relation to each other.

Evaluation with accurate input data In this first set of experiments, we consider that the algorithms have access to precise application profiles. This gives us ideal performance behavior.

As a first step, we study the placement heuristics. In Fig. 3, we present the performance ratio in terms of `Mean-I/O-SlowDown` when using GNC (Fig. 3a) or RandP (Fig. 3b) instead of the more informed heuristic GC for all allocation algorithms. To read Fig. 3a (resp. Fig. 3b), when the TA line is at 2%, that means that on average, TA-GNC (resp. TA-RandP) has a `Mean-I/O-SlowDown` 2% worse than that of TA-GC.



(a) GNC / GC (b) RandP / GC

Fig. 3: Relative `Mean-I/O-SlowDown` for different algorithm combinations when `I/O-load`(π_{sys}) increases. Placement are compared with GC. Lines show the mean value, and the area around them is the percentile interval (10^{th}-90^{th}).

The first key observation that we can make is that choosing either GC or GNC for placement has very little impact on performance (except for Random allocation). Nonetheless, this is not true for RandP, which confirms that placement plays a part in the performance.

By studying the load imbalance between I/O resources (i.e. the difference between the most loaded I/O resources and the least loaded I/O resources), we can show that there is a real difference in behavior between GC and GNC. This difference is even more notable with Random allocation. The fact that the I/O performance is not impacted hints that while we manage to stay below a certain level of I/O stress per I/O resource, improving this equilibrium does not matter. These results are available in the companion report [3]. Hence, we conclude that placement can be done greedily with limited information. *Based on this observation, in the rest of this section, we only use GNC as the placement algorithm.*

User-observed performance To compare the allocation heuristics, we decompose the `I/O-SlowDown` into their I/O and congestion components in Fig. 4, where we show their behavior when `I/O-load`(π_{sys}) varies. By design, BBA decreases the portion of I/O time due to I/O resources allocation (ρ^{io}) to its bare minimum, at the cost of a much higher congestion overhead (ρ^{con}) than that of NSYSA or Static. As `I/O-load`(π_{sys}) increases, 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` (NSYSA and TA) do not vary as much when `I/O-load`(π_{sys}) increases. Nonetheless, that comes at the cost of increased I/O time due to I/O resources allocation. Finally, MCKP [4] performs worse than all the other studied policies for these objectives. Indeed, it seeks to optimize the sum of applications' band-

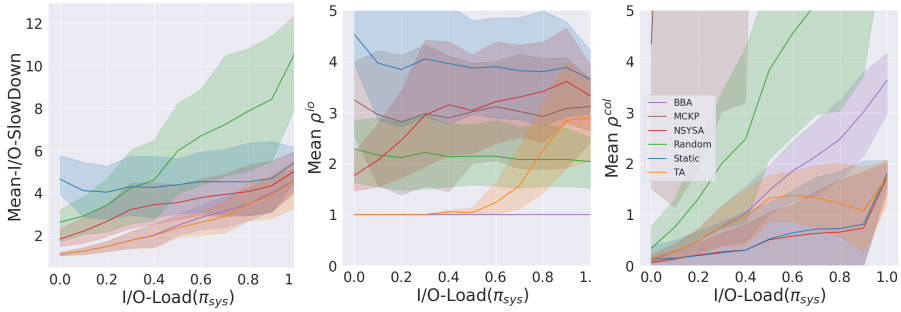
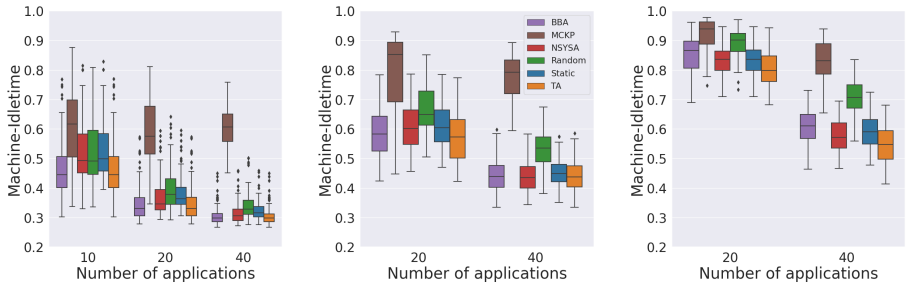


Fig. 4: `Mean-I/O-SlowDown` (left) separated into its two main components: I/O resources allocation ρ^{io} and congestion ρ^{con}. The lines show the mean value, and the area around them is the percentile interval (10^{th}-90^{th}). To improve visibility, MCKP results were omitted from the first plot: they range from 1.30 to 158.97.



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

widths due to I/O resources allocation, and has hence a tendency of favoring a few applications (the highest bandwidth ones) in detriment of others.

Machine utilization Solely looking at `Mean-I/O-SlowDown` tends to encourage equal treatment between all applications (big or small). 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 study the `Machine-Idletime` in Fig. 5. 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` (BBA and TA 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 TA and BBA, even though their respective I/O performance were similar, giving an advantage to the heuristic that considers more information (TA).

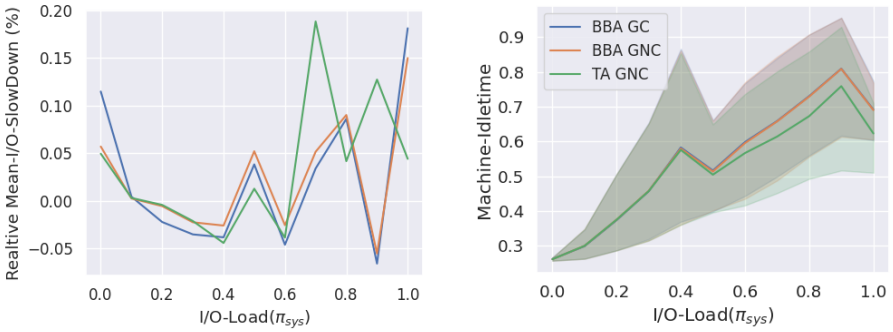
Static is an interesting heuristic: it uses little information about the applications and still can get good machine utilization results when the `I/O-load` is high.

Robustness to the quality of the input In Section 4.2, we were able to demonstrate the fact that under low `I/O-load`, BBA was the most efficient allocation algorithm, whereas under a heavier `I/O-load`, one should rather use TA which provides the same I/O performance from an application perspective but improves on system utilization.

As shown in Section 3.3, there is however an important difference between these two algorithms: their input. In particular, TA requires a full I/O profile of the applications whereas BBA 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. Essentially all applications are classified into four profiles (Ascent, Descent, Peak, Neutral). Then the bandwidth function within a profile is identical for all applications within said profile. Details are provided in the companion report [3].

In Fig. 6, we study the performance of three combinations of algorithms (TA+GNC; BBA+GNC; BBA+GC) 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



(a) `Mean-I/O-SlowDown` compared to performance with exact information (%)

(b) `Machine-Idletime`

Fig. 6: Performance of three scheduling solutions with partial input information

GNC is not impacted by the lack of accuracy: its behavior does not change. Similarly, BBA is impacted by the lack of accuracy only for some of the applications with a bandwidth profile `Peak` and `Neutral`.

Specifically, in Fig. 6a, 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 very similar performance. As expected, the allocation algorithm where the behavior differs the most is TA, which is the one that requires the most information. Yet this difference is still negligible (less than 1%). With respect to the mapping algorithm, GC performs almost identically to GNC 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. 6b) which confirms that even with inaccurate information TA 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 placement algorithms hold.

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.

Overall the main observation [3] is the same as previously: when there is little I/O stress on the system, then BBA performs better than I/O stress-aware algorithms such as NSYSA. 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 TA is an excellent alternative that is able to match the performance of BBA when needed, while being I/O stress aware.

Static's performance are excellent for this use case. 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.

5 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).

In contrast, 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 exact information may not be needed for some of the I/O scheduling algorithms. This should considerably simplify the design of I/O analysis tools.

Acknowledgments. As part of the "France 2030" initiative, this work has benefited from a State grant managed by the French national research agency (*Agence Nationale de la Recherche*) attributed to the Exa-DoST project, and bearing the reference ANR-22-EXNU-0004. It was also supported 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). Experiments 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`).

References

1. Almási, G., Bellofatto, R., Brunheroto, J., Caşcaval, C., Castaños, J.G., al.: An Overview of the Blue Gene/L System Software Organization. In: Euro-Par'23. Springer (2003)
2. Aupy, G., Beaumont, O., Eyraud-Dubois, L.: Sizing and partitioning strategies for burst-buffers to reduce io contention. In: IEEE IPDPS. pp. 631–640. IEEE (2019)
3. Bandet, A., Boito, F., Pallez, G.: Allocation and Placement Algorithms for Scheduling Distributed I/O Resources in HPC Systems. Tech. Rep. hal-04593977, Inria (2024), `https://inria.hal.science/hal-04593977`
4. Bez, J.L., Boito, F.Z., Miranda, A., Nou, R., Cortes, T., Navaux, P.O.A.: Towards On-Demand I/O Forwarding in HPC Platforms. In: PDSW. pp. 7–14 (Nov 2020)
5. Bez, J.L., Karimi, A.M., Paul, A.K., Xie, B., Byna, S., Carns, P., al.: Access patterns and performance behaviors of multi-layer supercomputer i/o subsystems under production load. In: HPDC '22. p. 43–55. New York, NY, USA (2022)
6. Bleuse, R., Dogeas, K., Lucarelli, G., Mounié, G., Trystram, D.: Interference-aware scheduling using geometric constraints. In: Euro-Par. pp. 205–217. Springer (2018)
7. Boito, F.: Write performance with different numbers of OSTs for BeeGFS in PlaFRIM (Jan 2024). `https://doi.org/10.5281/zenodo.10518127`
8. Boito, F., Pallez, G., Teylo, L.: The role of storage target allocation in applications' I/O performance with BeeGFS. In: CLUSTER. Heidelberg, Germany (2022)

9. Boito, F., Pallez, G., Teylo, L., Vidal, N.: IO-SETS: Simple and efficient approaches for I/O bandwidth management. TPDS **34**(10), 2783 – 2796 (Aug 2023)
10. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., et. al.: Understanding and Improving Computational Science Storage Access through Continuous Characterization. ACM Transactions on Storage **7**(3), 8:1–8:26 (Oct 2011)
11. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 Characterization of petascale I/O workloads. In: Cluster. pp. 1–10. IEEE (2009)
12. Chowdhury, F., Zhu, Y., Heer, T., Paredes, S., Moody, A., et. al.: I/o characterization and performance evaluation of beegfs for deep learning. In: ICPP (2019)
13. Costa, E., Patel, T., Schwaller, B., Brandt, J.M., Tiwari, D.: Systematically Inferring I/O Performance Variability by Examining Repetitive Job Behavior. SC '21, ACM (2021)
14. Devarajan, H., Mohror, K.: Extracting and characterizing i/o behavior of hpc workloads. In: CLUSTER. pp. 243–255 (2022)
15. Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S.: Calciom: Mitigating I/O interference in hpc systems through cross-application coordination. In: IEEE IPDPS. pp. 155–164 (2014)
16. Dorier, M., Ibrahim, S., Antoniu, G., Ross, R.: Using Formal Grammars to Predict I/O Behaviors in HPC: The Omnisc'IO Approach. IEEE TPDS **27**(8), 2435–2449 (Aug 2016)
17. Gainaru, A., Aupy, G., Benoit, A., Cappello, F., Robert, Y., Snir, M.: Scheduling the I/O of HPC Applications Under Congestion. In: IEEE IPDPS (May 2015)
18. Ji, X., Yang, B., Zhang, T., Ma, X., Zhu, X., Wang, X., El-Sayed, N., Zhai, J., Liu, W., Xue, W.: Automatic, application-aware i/o forwarding resource allocation. In: USENIX CFST. pp. 265–279. FAST'19, USENIX Association (2019)
19. Liu, Y., Gunasekaran, R., Ma, X., Vazhkudai, S.S.: Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems. In: SC'16. pp. 819–829 (Nov 2016)
20. Lopez, A., Valat, S., Narasimhamurthy, S., Golasowski, M.: Ephemeral data access environment: Concept and architecture. Tech. rep., IO-SEA project (2022)
21. Vishwanath, V., Hereld, M., Iskra, K., Kimpe, D., Morozov, V., Papka, M.E., Ross, R., Yoshii, K.: Accelerating i/o forwarding in ibm blue gene/p systems. In: SC'10. pp. 1–10. IEEE (2010)
22. Wang, F., Oral, S., Gupta, S., Tiwari, D., Vazhkudai, S.S.: Improving large-scale storage system performance via topology-aware and balanced data placement. In: ICPADS'14. pp. 656–663 (2014)
23. Xu, W., Lu, Y., Li, Q., Zhou, E., Song, Z., Dong, Y., Zhang, W., Wei, D., Zhang, X., Chen, H., Xing, J., Yuan, Y.: Hybrid hierarchy storage system in MilkyWay-2 supercomputer. Frontiers of Computer Science **8**(3), 367–377 (Jun 2014)
24. Yang, B., Ji, X., Ma, X., Wang, X., Zhang, T., et. al.: End-to-end I/O monitoring on a leading supercomputer. In: NSDI). USENIX Association (Feb 2019)
25. Yang, B., Zou, Y., Liu, W., Xue, W.: An End-to-end and Adaptive I/O Optimization Tool for Modern HPC Storage Systems. In: IPDPS'22. pp. 1294–1304 (2022)
26. Yu, J., Liu, G., Dong, W., Li, X., Zhang, J., Sun, F.: On the load imbalance problem of I/O forwarding layer in hpc systems. In: ICCC (2017)