Arbitration Policies for On-Demand User-Level I/O Forwarding on HPC Platforms

Jean Luca Bez[1], Alberto Miranda[2], Ramon Nou[2], Francieli Zanon Boito[3], Toni Cortes[2,4], Philippe Navaux[1]

[1]Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS) — Porto Alegre, Brazil
[2]Barcelona Supercomputing Center (BSC) — Barcelona, Spain
[3]LaBRI, University of Bordeaux, Inria, CNRS, Bordeaux-INP – Bordeaux, France
[4]Polytechnic University of Catalonia — Barcelona, Spain

{*jean.bez, navaux*}*@inf.ufrgs.br*, {*alberto.miranda,ramon.nou*}*@bsc.es*,
francieli.zanon-boito@u-bordeaux.fr, toni@ac.upc.edu

***Abstract*—I/O forwarding is a well-established and widely-adopted technique in HPC to reduce contention in the access to storage servers and transparently improve I/O performance. Rather than having applications directly accessing the shared parallel file system, the forwarding technique defines a set of I/O nodes responsible for receiving application requests and forwarding them to the file system, thus reshaping the flow of requests. The typical approach is to statically assign I/O nodes to applications depending on the number of compute nodes they use, which is not always necessarily related to their I/O requirements. Thus, this approach leads to inefficient usage of these resources. This paper investigates arbitration policies based on the applications I/O demands, represented by their access patterns. We propose a policy based on the Multiple-Choice Knapsack problem that seeks to maximize global bandwidth by giving more I/O nodes to applications that will benefit the most. Furthermore, we propose a user-level I/O forwarding solution as an on-demand service capable of applying different allocation policies at runtime for machines where this layer is not present. We demonstrate our approach's applicability through extensive experimentation and show it can transparently improve global I/O bandwidth by up to 85% in a live setup compared to the default static policy.**

***Index Terms*—I/O forwarding, allocation policy, MCKP**

1. Introduction

The increasing input and output (I/O) demands of applications from distinct domains stress the existing shared storage infrastructure of High-Performance Computing (HPC) facilities. Furthermore, the increasing heterogeneity of the workloads running in HPC installations, from the traditionally compute-bound scientific simulations to Machine Learning applications and I/O bound Big Data workflows, pose new challenges. As systems grow in the number of compute nodes to accommodate larger applications and more concurrent jobs, the shared storage powered by Parallel File Systems (PFS) is not able to keep providing performance due to concurrency and interference [1], [2], [3].

To mitigate this issue, the I/O forwarding technique [4] seeks to reduce the number of nodes concurrently accessing the PFS servers by creating an additional layer between the compute nodes and the data servers. Thus, rather than applications accessing the PFS directly, the I/O forwarding technique defines a set of *I/O nodes* that are responsible for receiving I/O requests from applications and forwarding them to the PFS in a controlled manner, allowing the application of optimization techniques such as request scheduling and aggregation. Moreover, its presence on an HPC system is transparent to applications and file system agnostic. Due to these benefits, the forwarding technique is applied by Top 500 machines[1] (Table 1).

The forwarding layer is traditionally physically deployed on special nodes, and the mapping between clients and I/O nodes is static. Consequently, a subset of compute nodes will only forward requests to a single fixed I/O node, which ends up forcing applications to use I/O forwarding with a statically pre-defined number of I/O nodes, even if that decision might not be in the best interest for a given workload. Though this setup seeks to distribute I/O nodes between compute nodes evenly, it lacks the flexibility to adjust to applications' I/O demands, and it can even cause the misallocation of forwarding resources and an I/O load imbalance, as demonstrated by Yu et al. [8] on the Sunway TaihuLight and Bez et al. [9] on MareNostrum 4. Moreover, the number of I/O nodes given to an application directly impacts its performance, and the achieved bandwidth also depends on the workload characteristics of the application [9].

In this paper, we argue in favor of a dynamic, on-demand allocation of I/O nodes that considers an application's workload characteristics. Accordingly, given a set of

TABLE 1. SOME OF TOP 500 MACHINES THAT ARE KNOWN TO USE THE I/O FORWARDING TECHNIQUE (JUNE 2020).

Rank	**Supercomputer**	**Compute Nodes**	**I/O Nodes**
4	Sunway TaihuLight [5]	40, 960	240
5	Tianhe-2A [1]	16, 000	256
10	Piz Daint [6]	6, 751	54
11	Trinity [7]	19, 420	576

[1] November 2020 TOP500: https://www.top500.org/lists/2020/06/.

applications ready to run and a fixed number of forwarding resources, solving their allocation problem would consist in determining how many I/O nodes each of them should receive to maximize the aggregated global bandwidth. The allocation policy should be invoked before new applications start to run, and when the set of running jobs has changed.

However, due to the static nature of traditional I/O forwarding infrastructures and the inherent limitations involved in running a production supercomputer, it is not always possible for system administrators to explore different I/O allocation strategies without negatively impacting user jobs. Thus, a research/exploration solution is required that allows both I/O researchers and system administrators to obtain an overview of the benefits or drawbacks of different I/O forwarding configurations under different access patterns. For such a solution to be useful, it should be portable, allow existing applications to run without modifications to their source-code and, if possible, run as a user-level service to simplify deployment. As we advocate for dynamic allocation, such a solution should also allow changing the number of I/O nodes assigned to an application during its execution without disrupting it. With these goals in mind, our contributions are thus three-folded:

- We evaluate different I/O forwarding allocation policies and demonstrate that a dynamic allocation can improve overall global bandwidth and system usage, while efficiently using the available I/O nodes.
- We propose a forwarding allocation policy based on the Multiple Choice Knapsack Problem (MCKP) to arbitrate I/O nodes between applications.
- We present an I/O forwarding service called GekkoFWD that acts as an on-demand forwarding layer and implements the MCKP allocation policy. GekkoFWD builds on top of a user-level ad-hoc file system, enriching it to allow exploring different forwarding deployments. It does not require application modifications and it is simple to run in production.

This paper is organized as follows. Section 2 further motivates this work. Section 3 addresses the arbitration of I/O nodes and details our allocation policy based on the Multiple-Choice Knapsack Problem. Section 4 presents our I/O forwarding solution at user-level. We discuss our results in Section 5. Related work is reviewed in Section 6. Finally, we conclude this paper in Section 7.

2. Motivation

To demonstrate how *application-perceived* I/O performance can change in response to different forwarding configurations, we executed several experiments on the MareNostrum 4 (MN4) supercomputer to get the performance curves of different access patterns with varying numbers of I/O nodes. We adopted a simple tool implemented in user-space named FORGE (I/O Forwarding Explorer) that allows replaying application I/O profiles in different forwarding configurations [9]. MareNostrum has 3, 456 Lenovo ThinkSystem SD530 compute nodes on 48 racks. Each node has two Intel Xeon Platinum 8160 24C chips with 24 processors each at 2.1 GHz which totals to 165, 888 processes and 390 TB of main memory. A 100 Gb Intel Omni-Path Full-Fat Tree is used for the interconnection network and a total of 14 PB of storage capacity is offered by IBM's GPFS (7 data servers and 2 metadata servers). Using FORGE, we covered 189 scenarios with:

- 8, 16, and 32 compute nodes;
- 12, 24, and 48 client processes per compute node;
- File layout: file-per-process or shared-file;
- Spatiality: contiguous or 1D-strided;
- Operation: writes with `O_DIRECT` enabled to account for caching effects present in the system;
- Request sizes of 32KB, 128KB, 512KB, 1MB, 4MB, 6MB, and 8MB synchronously issued until a given total size is transferred or 1s has passed.

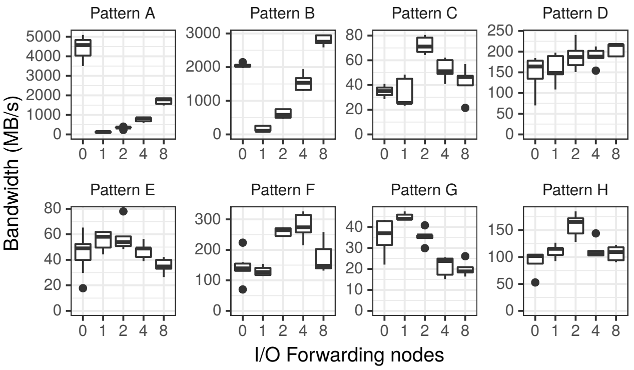

Figure 1. I/O bandwidth of distinct write access patterns with a varying number of I/O forwarding nodes in the MareNostrum 4 supercomputer.

TABLE 2. Details of the access patterns shown in Figure 1.

#	Nodes	Processes	File Layout	Request Spatiality	Request (KB)
A	32	1536	File-per-process	Contiguous	1024
B	32	1536	File-per-process	Contiguous	128
C	32	1536	Shared	Contiguous	1024
D	16	192	Shared	1D-strided	128
E	8	192	Shared	1D-strided	1024
F	16	384	Shared	Contiguous	128
G	32	384	Shared	1D-strided	512
H	8	384	Shared	Contiguous	4096

Figure 1 depicts the achieved bandwidth computed from the execution time (makespan), measured at client-side, when multiple clients issue their requests following an access pattern and taking into account the number of available I/O nodes (0, 1, 2, 4, and 8). Each experiment was repeated at least 5 times, in random order, and spanning different days and periods of the day. Table 2 describes each depicted pattern. The complete evaluation of the 189 experiments is available at the paper's companion repository[2].

The optimal number of I/O nodes for each of the 189 scenarios, considering the available choices of I/O nodes, is different. For 12 (6%), 83 (44%), 15 (8%), and 17 (9%)

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

scenarios, the largest bandwidth is achieved by using 1, 2, 4, and 8 I/O nodes respectively. Whereas, for 62 scenarios (33%), not using forwarding is instead the best alternative.

As expected, there does not seem to be a simple rule regarding the number of I/O nodes to fit all applications and system configurations, which is to be expected given the complexity of factors that can influence I/O performance. Furthermore, some patterns seem to benefit the most from having access to more I/O nodes than others. Consequently, a static mapping of I/O nodes to compute nodes without considering an application's workload does not always result in the best performance [9]. Hence, the need for appropriate allocation policies that take into account these issues to maximize *globally-perceived* I/O performance.

3. On I/O Node Arbitration

Since no simple rules allow to allocate I/O forwarding resources to best suit all applications, I/O node arbitration needs to be considered as an optimization problem. The I/O node allocation may informally be thought of as the following: given a set of jobs to run and a fixed number of I/O nodes, determine how many forwarding nodes each of them should receive to maximize the aggregated global bandwidth. Thus, every time the set of running applications changes, the decisions have to be reevaluated. Based on these requirements, this section describes our proposed solution to the allocation problem. To prove that we are on the right track, we evaluate its pre-implementation by measuring the achieved I/O performance through simulation, comparing it to different baselines.

3.1. The MCKP Allocation Policy

The Multiple-Choice Knapsack Problem (MCKP) is an optimization problem, derived from the 0-1 Knapsack, where the items are subdivided into k classes, each having N_i items. The binary choice of taking an item in the 0-1 problem is replaced by selecting exactly one item from each class. Formally, the problem is described by (1). Where the variables x_{ij} take on value 1 if and only if item j is chosen in class N_i. While the problem is $\mathcal{NP}$-hard, the time complexity of its Dynamic Programming solution, which is pseudo-polynomial, is $O(W\sum_{i=1}^{k} N_i)$.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{k} \sum_{j \in N_i} p_{ij} x_{ij} \\ \text{subject to} \quad & \sum_{i=1}^{k} \sum_{j \in N_i} w_{ij} x_{ij} \leq W, \\ & \sum_{j \in N_i} x_{ij} = 1, \forall i \in \{1, ..., k\} \\ & x_{ij} \in \{0, 1\}, \forall i \in \{1, ..., k\}, \forall j \in N_i. \end{aligned} \tag{1}$$

We chose to model our problem after the MCKP as a global goal (i.e., the aggregated bandwidth) needs to be maximized based on a set of options available to choose from (i.e., the number of I/O nodes an application could use). We focus on optimizing how many forwarding nodes an application should use rather than where it should run.

For the I/O node allocation policy, each class represents an application, and the items of a class denote the number of I/O nodes that the application could use. These items can be different for each class, as long as the number of compute nodes used by the application is divisible by the number of I/O nodes. This constraint is to improve load balancing. Furthermore, it is limited by the total number of I/O nodes. The weight w_i of each item represents the number of I/O nodes, and the value p_i the bandwidth. We must pick one choice for each application, seeking to maximize the global bandwidth, taking into account a pool of available I/O forwarding nodes, represented in the problem by the capacity W.

We assume that we have information about an application's I/O performance using different numbers of forwarding nodes. One can obtain it from exploratory executions, though this information can also be extracted from Darshan [10] traces, which are already transparently collected at many supercomputers. These traces can be used to identify the base access patterns (e.g., file approach, spatiality, and request sizes), the number of processes making I/O requests, and the total transferred data volume. Combined with performance metrics of short benchmark runs using those base patterns with different number of I/O nodes, they allow us to estimate the complete application's I/O performance. Hence profiling runs with different forwarding setups for each application are not required. Details of such an approach are described in [11], as here we focus on the allocation policy. When no such application data is available, i.e., on its first execution, MCKP is provided with the default number of I/O nodes the application would receive for that particular system setup, hence avoiding a negative impact on performance, and future runs could make better decisions based on the collected data.

In our experiments, we allow applications to not use forwarding, which implies no need for sharing I/O nodes as the number of available forwarding resources is always enough. An additional option could be given to the applications when sharing is inevitable due to system deployment: using a system-wide shared I/O node. Nevertheless, we seek to avoid that where possible as it could bring performance interference. When considering sharing, we could use a naive estimation based on the bandwidth of using a single node (for that application) divided by the total number of running applications. There are two caveats to this approach: (I) estimating the impact of interference is not simple and (II) the number of applications sharing the I/O node will be smaller than the number of running applications. Nonetheless, such an estimate does not present an issue as it would be a low-bandwidth option that the policy will take only for the least-performant applications. The remaining $(N - 1)$ I/O nodes could then be given to MCKP to arbitrate.

3.2. Evaluation of MCKP Applicability

From the 189 patterns executed at the MN4 machine, as described in Section 2, we randomly sampled sets of 16 to simulate each policy considering N available I/O nodes. For this experiment, each pattern is an application that is ready to run. We generated $10,000$ sets, to cover multiple combinations of those patterns running at the same time, having up to 128 forwarding nodes to allocate among the 16 applications – eight per application, the maximum I/O node number for which we have results. In those sets, the median number of compute nodes used by all applications was 256, with a minimum of 88 and a maximum of 512 nodes. Results are then obtained by Equation 2, taking the sum of the 16 applications' bandwidth. The W and R in the equation represent the total transferred size by write and read operations for each application a.

$$aggregate\ BW = \sum_{a=1}^{16} \left(\frac{W_a + R_a}{runtime_a} \right) \qquad (2)$$

We compare our MCKP solution to alternative policies:

- **ZERO** and **ONE Policies**: each application is assigned zero or one I/O nodes. These policies demonstrate the initial impact of using I/O forwarding.
- **STATIC Policy**: the total number of I/O nodes is divided between the applications based on the number of compute nodes each one requires (C_a). The number of I/O nodes assigned to application a is given by $ceil(\frac{C_a}{R})$, where $R = \frac{C}{F}$. C and F are the total numbers of compute and I/O nodes in the system. This is the policy used by some supercomputers that have forwarding.
- **SIZE** and **PROCESS Policies**: the I/O nodes are proportionally divided between the running applications based on their sizes s_a (number of compute nodes or processes). The number of I/O nodes assigned to application a is $round\left(F \times \frac{s_a}{\sum_{i=0}^{A} s_i}\right)$, where F is the total number of forwarding nodes, and A the number of applications. The main difference to the STATIC policy is that even when not all compute nodes are in use, all I/O nodes are.
- **ORACLE Policy**: each application is assigned the number of I/O nodes that achieved the highest bandwidth, obtained from our performance evaluation (Figure 5). This is a *fictitious policy* that disregards the limited number of I/O nodes in the system. It is intended to provide an upper bound for the gains.

If we consider the ONE policy, where we allocate a single non-shared I/O node to each application and compare it to not using forwarding (ZERO), in our simulations, we observed a median slowdown of 82.11%. As the majority of the tested workloads benefit from using more than a single I/O node, or not using I/O forwarding at all, this policy is not well suited. On the other hand, if we compare the ZERO to the ORACLE policy, it is possible to better grasp the potential improvements of using forwarding. In this comparison, we observed a minimum performance boost of 0.83% and a maximum of 121.68%. The median improvement was of 25.63%, obtained by using forwarding correctly.

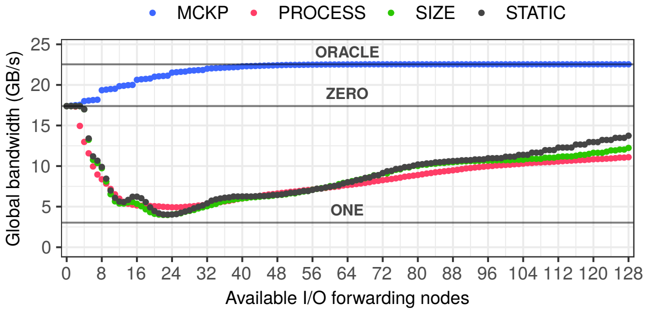

Figure 2. Median global bandwidth observed in the $10,000$ sets of 16 randomly selected applications from the 189 scenarios collected at MN4 supercomputer, as described in Section 2.

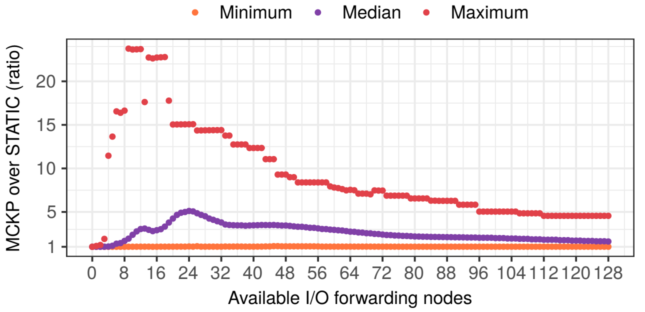

Figure 3. Improvements of MCKP over the STATIC arbitration policy.

Figure 2 compares the median aggregated bandwidth, computed by Equation 2, of the $10,000$ experiments for each number of I/O forwarding nodes using the arbitration policies. The MCKP policy achieves the same aggregated bandwidth as the ORACLE policy (which is not limited by the number of available I/O nodes) when 56 forwarding nodes are allocated between the 16 random running applications. These results also demonstrate that allocating I/O nodes solely based on application size (i.e., number of required compute nodes or processes) is not the best solution. Compared to the STATIC policy, when the optimal number of 56 I/O nodes is available, the MCKP policy achieved a minimum performance boost of 4.08% and a maximum of 739.22%. The median was 211.38%. Figure 2 also highlights the importance of considering the applications' I/O demands when fewer than the optimal number of I/O nodes are available to be arbitrated.

Figure 3 complements the results by depicting the distribution — minimum, median, and maximum — of the improvements in aggregate bandwidth shown by MCKP when compared to the STATIC alternative used by many supercomputers today. The highest median improvement of 5.11× is attained when 24 I/O nodes are available, which gives a ratio of I/O nodes per compute nodes of 1:20, considering the median number of compute nodes the sets of applications have. With this pool size, the MCKP policy improved bandwidth between 1.03× and 15.06× over the STATIC policy. When fewer I/O nodes are available to arbitrate among the running applications, the allocation decision has more impact on global bandwidth. As we are considering sets of 16 random applications (drawn from 189), the difference between MCKP over STATIC can be highlighted (red line in Figure 3), depending on the characteristics and number of compute nodes of each application.

Most interestingly, MCKP never impacts bandwidth negatively when compared to the STATIC policy. Furthermore, MCKP provides, on average, 2.6× the aggregated bandwidth of the existing solution and reaches up to 23.75× the aggregated bandwidth of the STATIC policy. The difference between these two policies tends to reduce as more I/O nodes are available, but MCKP still outperforms STATIC by 1.6× to 2.7×, for 64 I/O nodes (1:8) up to 128 (1:4) I/O nodes available in the system.

4. GekkoFWD: On-Demand I/O Forwarding

Although FORGE allows replaying I/O profiles from applications to rapidly explore different I/O forwarding deployments, it lacks the support to actually run applications themselves. Besides, once an I/O node mapping is selected, it is impossible to dynamically change the number of allocated I/O nodes at runtime. Therefore, we propose a full-fledged user-level I/O forwarding solution that is adequate and easy to run in production machines. To achieve this goal, we enriched an existing ad-hoc file system called GekkoFS [12], [13] with a *forwarding mode*. GekkoFS creates a temporary file system on compute nodes using their local storage capacity as a burst-buffer to alleviate I/O peaks. It ranked 4th in the overall 10-node challenge of IO500[3] in November 2019, as well 2nd concerning metadata performance in the same challenge.

GekkoFS uses the local storage available on compute nodes to provide a global namespace accessible to all participating nodes. The **GekkoFWD** extension mode modifies it to use the shared PFS (e.g., Lustre, GPFS) for storage instead. Moreover, data operations in GekkoFS are typically distributed across all nodes using the Mercury HPC RPC framework. Once an I/O operation is intercepted, the client forwards that request to the responsible server, determined by hashing the file's path. To achieve a balanced data distribution, each file is split into equally sized chunks by the client and distributed among the servers. Conversely, GekkoFWD enables GekkoFS servers to act as intermediate I/O nodes between the compute nodes and the PFS data servers. To achieve this, we leverage the system call interception infrastructure in each GekkoFS client to transparently capture all application I/O requests in each compute node. Then GekkoFWD forwards those requests to a single server, which will now act as an I/O node as determined by a pre-defined allocation policy. To conform to such a policy, we included a thread in the GekkoFS client that checks for mapping updates and responds to any modification.

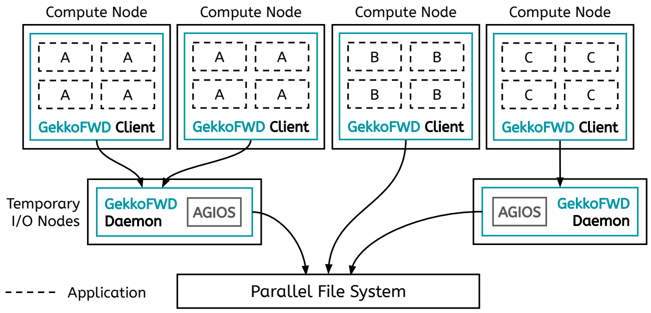

Figure 4. GekkoFWD deployment uses an interception library at the client side and a daemon on the nodes that will act as temporary I/O nodes.

Since forwarding layers are transparent to applications, they usually are a perfect target to implement I/O optimizations such as file-level request scheduling [14], [15], [16]. For that reason, we integrated the AGIOS [17] scheduling library into GekkoFWD. AGIOS provides several schedulers, giving GekkoFWD the flexibility to prototype new scheduling solutions. Once a request is received by a GekkoFWD I/O node, it is fed to AGIOS to determine when it should be processed. Once scheduled, it is then dispatched to the PFS and executed following the normal flow of requests in GekkoFS. Figure 4 depicts a deployment of GekkoFWD, where some applications use forwarding and others do not, depending on the pre-selected policy. GekkoFWD is open source and is available in the official GekkoFS repository[4].

5. Experimental Evaluation

Since we needed fine control on allocation decisions, our evaluation was conducted on the Grid 5000 (G5K) platform, a large-scale testbed for experiment-driven research. Our experiments used two clusters from the Nancy site: Grimoire (8 nodes) and Gros (124 nodes). Grimoire nodes are powered by an Intel Xeon E5-2630 v3 processor (Haswell, 2.40 GHz, 2 CPUs per node, 8 cores per CPU) and 128 GB of memory. The Lustre parallel file system servers deployed on Grimoire nodes use a 600 GB HDD SCSI Seagate ST600MM0088. Gros nodes are powered by an Intel Xeon Gold 5220 processor (Cascade Lake-SP, 2.20 GHz, 1 CPU/node, 18 cores/CPU) and 96 GB of memory. Each node of Gros is connected to two switches with 2 × 10Gbps Ethernet links. The two switches are connected to another one with 2 × 40Gbps links each. The latter is connected to Grimoire's nodes with 4 × 10Gbps Ethernet to each node.

5.1. Applications

To demonstrate the applicability of MCKP under mixed I/O workloads, we ran five different application kernels and the IOR[5] micro-benchmark on top of GekkoFWD.

- The **S3D I/O Kernel** [18] performs N checkpoints (five in our case) at regular intervals, where it writes three and four-dimensional arrays of doubles into

[3] https://www.vi4io.org/io500/list/19-11/10node

[4] https://storage.bsc.es/gitlab/hpc/gekkofs/-/releases

[5] https://github.com/hpc/ior

TABLE 3. Setup and I/O characteristics of the applications.

Label	Application	Operation	File Approach	Write (GB)	Read (GB)	Total (GB)	Nodes	Processes
BT-C	NAS BT-IO (Class C)	write / read	Single shared file	6.3	6.3	12.6	32	128
BT-D	NAS BT-IO (Class D)	write / read	Single shared file	126.5	126.5	253.0	64	512
HACC	HACC-IO	write	File-per-process	1.8	0	1.8	8	64
IOR-MPI	IOR (MPI-IO)	write / read	Single shared file	16.0	16.0	32.0	16	128
POSIX-S	IOR (POSIX)	write / read	Single shared file	16.0	16.0	32.0	16	128
POSIX-L	IOR (POSIX)	write / read	File-per-process	32.0	32.0	64.0	64	512
MAD	MADBench2	write / read	Single shared file	16.2	16.2	32.4	32	64
SIM	S3DSIM	write	Single shared file	19.6	0	19.6	16	16
S3D	S3D-IO	write	Multiple shared files	33.7	0	33.7	64	512

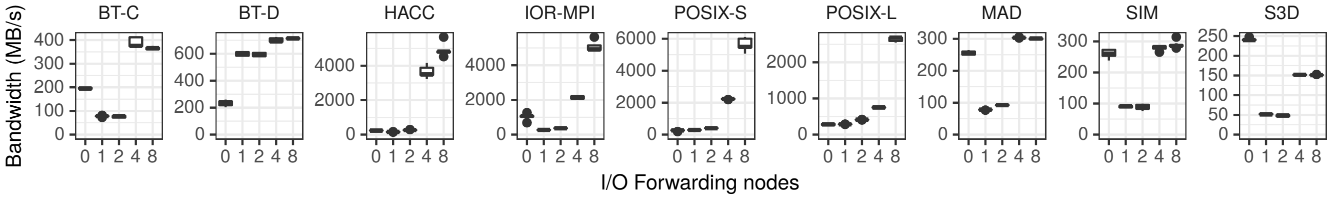

Figure 5. I/O bandwidth, measured at client-side, of five repetitions of each application described in Table 3. The x-axis represents the number of I/O forwarding nodes exclusively used by the job. The y-axis is not the same for each plot.

a newly created file. All three-dimensional arrays are partitioned among the MPI processes, whereas the fourth dimension (the most significant one) is not partitioned. We configured it to use PnetCDF nonblocking APIs, where each checkpoint has four nonblocking write calls, a wait, and a flush [19].

- **MADBench2** [20] has three I/O component functions, each with different access patterns, named *S*, *W*, and *C*. The *S* component consists of writes by a subset of the processes. In *W*, that data is read back, and a smaller subset writes new data. Finally, in component *C*, that data is read back. MADBench2 uses the MPI-IO interface to issue its I/O operations synchronously to a single shared file.
- **HACC-IO**[6] is an I/O kernel of the HACC application used to simulate collision-less fluids in space using N-Body. In our tests, we used $N = 100$K particles. Each process writes $N \times 38$ bytes and a 24 MB header to its own file through POSIX.
- **S3aSim** [21] uses a parallel programming model with database segmentation, which mimics the mpiBLAST access pattern. Given input query sequences, it divides up the database sequences into fragments. Workers request a query and fragment information from the master and search the query against the database fragment assigned. The results are sent to the master to be sorted and then written to a single shared file. We configured S3aSim to issue 100 queries varying from 1 to 10 to a database sequence of sizes ranging from 6 to $45,088,768$, both using a simple uniform random distribution and 128 fragments. Each query writes from ≈ 4MB to 328MB of data, $\approx$ 100MB on average. This application uses individual I/O operations, without synchronizing after writing every query.
- **NAS BT-IO**[7] is based on the Block-Tridiagonal (BT) problem of the NAS Parallel Benchmarks (NPB). After every five time steps, the entire solution, consisting of five double-precision words per mesh point, must be written to file. In the end, all data belonging to a single time step must be stored in the same file and must be sorted by vector component, x, y, and z-coordinate. We used the BT-IO MPI version with collective buffering, where data scattered among the processors is collected on a subset of the participating processors and rearranged before written to file in order of increasing granularity. The C class with 128 processes issues MPI-IO requests of 1.34MB and POSIX requests of 5.23MB, according to Darshan logs. The D class with 512 processes issues larger requests of 5.35MB and 12.31MB, for MPI-IO and POSIX, respectively.

Table 3 labels each application based on its configuration. We detail the parameters used to run each one of them in our companion repository. We compute the bandwidth for each application by measuring the application's execution time at the client-side (i.e., the makespan). Figure 5 confirms FORGE's results in Figure 1, where in neither case there is a single allocation solution that best fits all applications.

As we executed all applications using C compute nodes and P processes, where both C and P are a power of two, we also considered the number of available I/O nodes each application can use as powers of two. For our evaluation, the policies can choose between 0, 1, 2, 4, and 8 I/O nodes for each application. In practice, these options would comprise numbers divisible by the number of compute nodes used by each application to improve load balancing.

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

[7] https://www.nas.nasa.gov/publications/npb.html

5.2. Allocation Decisions

In this section, we investigate the allocation of I/O nodes with a subset of the applications described in Section 5.1. The aggregated bandwidth is computed as the sum of the bandwidth achieved by each application when using the number of I/O nodes allocated to it by the arbitration policy. We focus on a set of jobs composed of **BT-C**, **BT-D**, **IOR-MPI**, **POSIX-L**, **MAD**, and **S3D**. In total, these applications require 72 compute nodes. A complete experiment with all applications, dynamically changing the allocated I/O nodes, is presented in Section 5.3.

Figure 6 details the results. The x-axis represents the number of available I/O nodes to arbitrate, and each box groups a policy. The first group represents direct access to Lustre, whereas the second is the ONE policy. The last box represents the ORACLE policy, as detailed in Section 3. As demonstrated by the results with FORGE at MN4, the ONE policy represents a global slowdown (39.17%) compared to directly accessing the PFS servers, even though some applications such as **S3D** would benefit from this choice. The STATIC, SIZE, and PROCESS (the latter not depicted) cannot achieve the same aggregated bandwidth as the MCKP policy that is $4.59\times$, $4.59\times$, and $4.1\times$ better than the alternatives. MCKP achieves the same performance of the ORACLE (the upper bound) when 36 nodes are available to be arbitrated among the 6 running applications (Figure 6).

Regarding the number of allocated I/O nodes, the STATIC and SIZE policies distribute the I/O nodes in a non-optimal way. Under the constraint of 12 available I/O nodes, for instance, applications **BT-C**, **MAD**, and **S3D** should not use forwarding, as detailed by Table 4. Instead, **IOR-MPI** should receive more I/O nodes as it can achieve a bandwidth that is $18.96\times$ higher when using eight forwarders instead of one. The MCKP policy does not give any I/O nodes for **S3D** as the direct access to the PFS is the best option.

We analyzed the penalty to the performance of individual applications caused by our MCKP policy, which aims at maximizing the *global* bandwidth, in Figure 7. For each total number of available I/O nodes (the boxes), we show the performance of each application (x-axis) with the assigned number of I/O nodes, compared to the best possible result for that application running alone under the same number of I/O nodes. With four I/O nodes, applications **IOR-MPI** and **S3D** manage to achieve the same performance they would attain when running alone under this constraint. For both, choosing between 1, 2, or 4 I/O nodes, the latter is always the best choice. However, for the remaining applications, such as **BT-C** or **BT-D**, where 4 is also the best choice, they reach only 50% and 33% of the bandwidth they could achieve if running alone under that constraint. When running with other applications, especially **IOR-MPI** and **S3D**, they are not prioritized by the policy because they do not gain performance as the first two when using more I/O nodes.

TABLE 4. ALLOCATED FORWARDERS AND ACHIEVED BANDWIDTH USING THE STATIC, SIZE, AND MCKP POLICIES AND 12 I/O NODES.

	STATIC		**SIZE**		**MCKP**	
	I/O Nodes	BW (MB/s)	I/O Nodes	BW (MB/s)	I/O Nodes	BW (MB/s)
BT-C	1	77.6	1	77.6	0	195.7
BT-D	2	594.2	2	594.2	1	597.2
IOR-MPI	1	268.4	1	268.4	8	5089.9
POSIX-L	2	411.9	2	411.9	2	411.9
MAD	1	77.8	1	77.8	0	255.9
S3D	2	48.1	2	48.1	0	241.3

In Figure 8, we depict the bandwidth differences between the STATIC and MCKP policies for each application. Positive values mean that the MCKP was able to yield improvements, whereas negatives indicate that the STATIC policy was a better alternative for that particular application. Improving global bandwidth might often come from impairing specific applications. For instance, the MCKP policy sacrifices **BT-D** by giving fewer I/O nodes than what the STATIC policy would allocate to it. The reason is that **BT-D** has a lower bandwidth, and the increase in performance for the remaining applications is higher than what is lost by **BT-D**, if observed individually.

5.3. Dynamic Allocation Policy

In this section, we use GekkoFWD with MCKP to dynamically arbitrate the I/O nodes between the changing set of running applications in the G5K platform. We split the Gros cluster nodes into two groups: 96 compute nodes and 12 I/O nodes. We deployed Lustre in the Grimoire cluster with one MGS/MDS node and two OSS with one OST of 500GB each. Lustre was configured with a stripe size of 1MB and striping over the available OSTs. We do not consider directly accessing the PFS, i.e., not using forwarding for this test to mimic platforms with this restriction.

In this experiment, we have a predefined queue of jobs to be executed following a strict FIFO order. Once one or more applications are scheduled, MCKP is invoked to choose the number of I/O nodes each should use contained to the number of these resources in the system. The decision considers all running jobs and may change the number of I/O nodes used by some of them. The policy is also invoked when jobs finish, but new ones cannot be scheduled as there are not enough compute nodes yet. The only exception is the STATIC policy, which is invoked but will *not reallocate* resources for already running applications. Notice that we do not need to consider all jobs in the queue, just those already scheduled as ready to execute, as the forwarding resources will only be arbitrated among the running jobs.

We generated random queues of jobs using the applications described in Section 5.1. We selected one queue whose metrics indicate a high number of concurrently running jobs to observe the arbitration of forwarding resources and the decisions' impact. The source code of the queue generator is available at `doi.org/10.5281/zenodo.3875176`.

The selected queue has at least one job of each application, in the following order: **HACC**, **IOR-MPI**, **SIM**, **IOR-MPI**, **IOR-MPI**, **POSIX-S**, **POSIX-L**, **BT-C**, **MAD**, **MAD**, **S3D**, **HACC**, **HACC**, and **BT-D**. Figure 9 illustrates

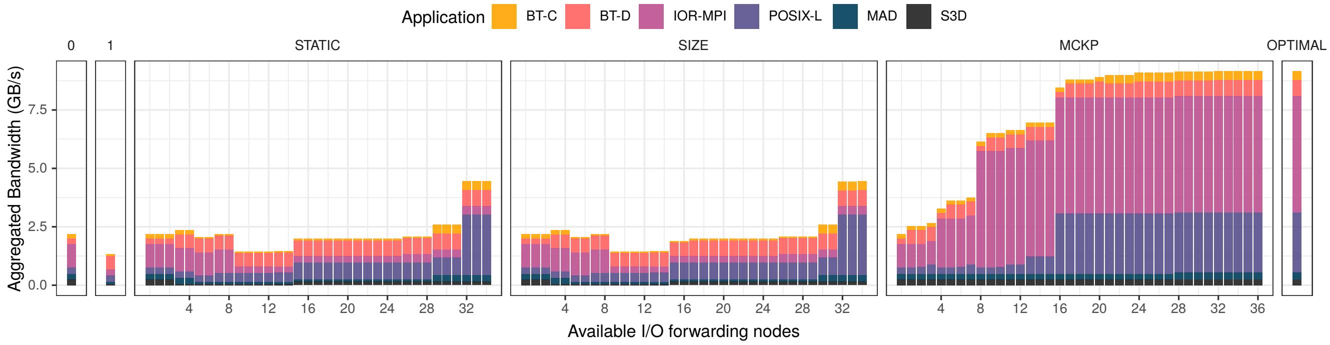

Figure 6. Global aggregated bandwidth (Equation 2) of the six applications under different I/O policies. Colors differentiate the applications.

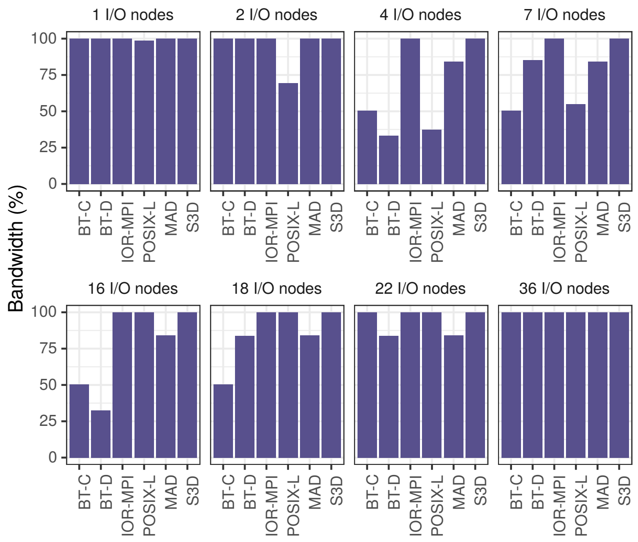

Figure 7. Bandwidth achieved by individual applications using the assigned number of I/O nodes by our MCKP policy, compared to each application running alone under the same I/O node number constraint.

the bandwidth achieved by each application (and the aggregated bandwidth given by Equation 2) under the ONE, STATIC, SIZE, and MCKP policies.

The first job of the **HACC** application is given 1 I/O node by the STATIC policy due to its size. In contrast, MCKP initially allocates 8 I/O nodes and then reduces to 4 as new jobs from **IOR-MPI** and **SIM** applications begin to execute. From the application's perspective, increasing the number of I/O nodes here translates into a bandwidth that is 3.9× higher than on the STATIC allocation (from 987.3MB/s to 3850.7MB/s). For **POSIX-L**, the STATIC policy allocated 8 I/O nodes reaching 1963.9MB/s, whereas MCKP only allows the application to use 8 I/O nodes during 9.7% of the time, and 2 I/O nodes for 90.3% of the time, which limits the bandwidth to 391.7MB/s. For the **POSIX-L** application, using 4 I/O nodes (MCKP) instead of 2 (SIZE), bandwidth is improved by 5.8×, from 180.5MB/s to 1049.9MB/s with MCKP. It is possible to see that the latter prioritizes applications that can reach high bandwidth by giving them more I/O nodes. Moreover, if we compare the STATIC solution to our dynamic MCKP arbitration policy, the latter improves global performance by 1.9× in this scenario — from 8.41GB/s to 16.02GB/s.

The dynamic remapping of I/O nodes to the compute nodes does not require any synchronization between the nodes of the forwarding layer, which could impact performance. The policy solver runs on a separate node, possibly the same used by a job manager (e.g., SLURM). Once the set of running jobs change, the policy should be reapplied to arbitrate all the I/O nodes in this new scenario. In the experiment presented in this section, the time to compute the solution was 399 µs. The time will vary based on the number of running jobs and the number of I/O nodes in the system. For system a with 512 concurrently running jobs and 256 I/O nodes, that would take 2.7 s. The trade-off in performance gains compensates for this overhead. The policy solver generates a new mapping file with allocation decisions. GekkoFWD clients check whether the mapping changed periodically (every 10 s by default). Thus, there might be a brief period where I/O nodes are shared by more than one application, especially if compute node clocks are not synchronized. We believe this should not pose an issue as jobs run in higher orders of magnitude.

6. Related Work

The I/O forwarding layer has been the focus of multiple research efforts to improve its performance and transparently benefit applications. Vishwanath et al. [14] improved the I/O performance of an IBM Blue Gene/P supercomputer by up to 38% by including asynchronous operations in the I/O nodes and a simple request scheduler to coordinate accesses from the multiple threads. Ohta et al. [15] implemented a FIFO, and the quantum-based HBRR request schedulers for the IOFSL framework. The latter aims at reordering and aggregating requests. TWINS [16], a novel scheduler proposed for the forwarding layer, aims at coordinating accesses to the data servers to avoid contention.

Yu et al. [8] address the load imbalance problem of the I/O forwarding layer. They argue that the bursty I/O traffic of HPC applications and the commonly rank 0 I/O pattern make the I/O nodes highly unbalanced. As some I/O nodes become hot spots, they hinder performance. Thus, they propose to recruit idle I/O nodes to alleviate this problem

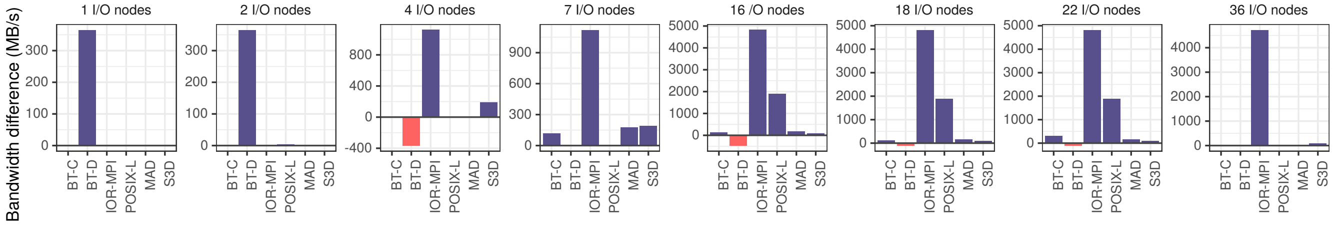

Figure 8. Bandwidth difference between applications running under STATIC and MCKP. Positive means MCKP was faster than STATIC. The y-axis is not the same in all the plots. The complete plot using other values for I/O nodes is available at our companion repository.

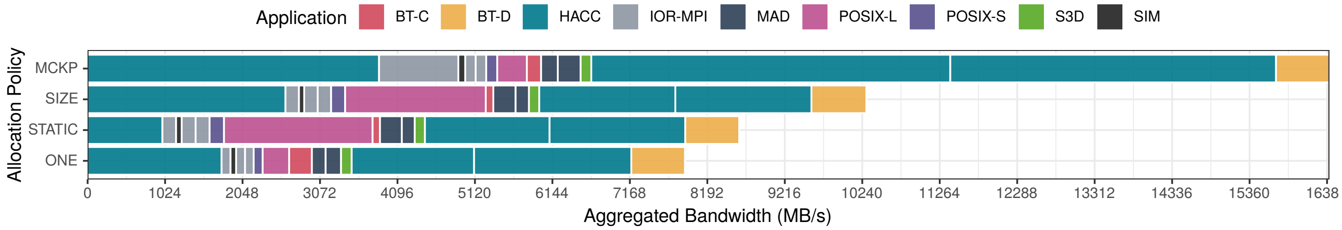

Figure 9. Dynamic allocation of I/O nodes for the running applications on the G5K platform using GekkoFWD. The x-axis displays the bandwidth achieved by each application under a given allocation policy. Colors identify the applications.

by giving additional nodes to the applications. The mapping of additional I/O nodes and primary I/O nodes are exclusive for an application, which adds some flexibility to the default static solution. However, the I/O nodes temporarily allocated to aid others might be required by their subset of compute nodes. In that case, global performance would be impacted. Furthermore, once the mapping is done for an application, it cannot be changed to accommodate new jobs that would benefit from additional nodes. Differently, we propose a dynamic approach to the problem, reviewing I/O node allocations as new jobs start or end their executions.

The Tianhe-2 (Milkyway-2) supercomputer has a hybrid hierarchy storage system named H^{2}FS, that merges the local storage of I/O nodes and the disks in the object storage servers [1]. This supercomputer can be configured with as few as one I/O per 64 compute nodes, or as many as one I/O node per $eight$ compute nodes. The H^{2}FS has an I/O path manager that maps compute nodes to a group of I/O nodes. Two mapping modes are supported. The first is determined by network topology (system deployment) when the system is initialized, or the applications specify it. The latter selects the I/O path based on the real-time overhead of the DPUs, seeking to reduce congestion, allocating DPUs for every file dynamically. Conversely, our approach does not focus on such a small granularity, i.e., file level, but rather on the whole application behavior. Moreover, our policy is not fixed for a given application, but instead, it takes into account concurrent jobs, adapting based on the workload.

Ji et al. [22] propose a dynamic forwarding resource allocation (DRFA), which estimates the number of forwarding nodes needed by a certain job based on its I/O history records. Their approach leverages automatic and online I/O subsystem monitoring and performance data analysis to make such decisions. DFRA works by remapping a group of compute nodes to other than their default I/O node assignments. They either grant more forwarding nodes (for capacity) or unused forwarding nodes (for isolation). Nonetheless, their allocation remains fixed once the job starts and do not adapt or allow a remapping when new applications start or finish to run. Conversely, we argue for a dynamic policy that can evaluate the set of running jobs to arbitrating I/O nodes. Moreover, their strategy relies on an over-provisioning of I/O nodes, and on the assumption that there are idle resources to satisfy all allocation upgrades.

In a previous work [9], we proposed FORGE, a simple forwarding layer implemented in user-space capable of replaying application's I/O profiles to rapidly explore a large number of different I/O forwarding deployments. However, FORGE is not a full-fledged solution. Furthermore, it is impossible to dynamically change the number of I/O nodes while executing an application's I/O profile.

In light of recent trends in storage system design using node local storage, we believe our approach is complementary. Local storage is often temporary and eventually needs to be flushed to the PFS, flowing through the forwarding layer (if present). If used solely as a cache for reads or writes (burst buffer), it would still need to eventually reach the PFS, once more flowing through the forwarding layer. In those scenarios where an application does not issue I/O requests directly to the PFS, MCKP would not allocate nodes for it.

7. Conclusion

In this paper, we argued in favor of a dynamic on-demand allocation of I/O nodes considering the application's I/O characteristics. We demonstrate that the forwarding layer's global deployment combined with the existing static allocation policy based solely on application size is not suitable to accommodate the increasingly heterogeneous workloads entering HPC installations. Instead, an application's I/O characteristics should also be considered when arbitrating forwarding resources among concurrently running applications to improve global performance.

We presented a user-level I/O forwarding solution named GekkoFWD that does not require application modifications and allows a dynamic remapping of forwarding resources to compute nodes. GekkoFWD is simple to run in production machines, where this layer is not present, targeting applications that would benefit from it. We proposed a novel I/O forwarding allocation policy based on the Multiple-Choice Knapsack Problem. We demonstrated our dynamic MCKP allocation policy's applicability to arbitrate on the available I/O forwarding resources through extensive evaluation and experimentation. We showed it can transparently improve global I/O bandwidth by up to 23× compared to the existing static policy, though improving global bandwidth might often come from impairing specific applications. Furthermore, we observed improvements of up to 85% in a live experiment using GekkoFWD and a queue of nine different applications.

As future work, we plan to test our approach in a larger production machine and expand the technique to supercomputers where forwarding is not yet deployed, recruiting idle compute nodes to act as temporary I/O nodes.

All data, source-code, technical implementation details, and analysis scripts used in this research are available at `jeanbez.gitlab.io/forwarding-arbitration`.

Acknowledgments

This study was financed by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. It has also received support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Brazil. It is also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grants PID2019-107255GB; and the Generalitat de Catalunya under contract 2014–SGR–1051. The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, J. Xing, and Y. Yuan, "Hybrid hierarchy storage system in MilkyWay-2 supercomputer," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 367–377, 2014.

[2] 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 Int. Parallel and Distributed Processing Symposium (IPDPS)*. Chicago, IL, USA: IEEE, 2016, pp. 750–759.

[3] J. Yu, G. Liu, X. Li, W. Dong, and Q. Li, "Cross-layer coordination in the I/O software stack of extreme-scale systems," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 10, 2018.

[4] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. Castanos, L. Ceze *et al.*, "An overview of the Blue Gene/L system software organization," in *Euro-Par 2003 Parallel Processing*, Euro-Par 2003 Conference. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 543–555.

[5] 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.

[6] S. Gorini, M. Chesi, and C. Ponti, "CSCS Site Update," http://opensfs.org/wp-content/uploads/2017/06/Wed11-GoriniStefano-LUG2017_20170601.pdf, 2017, [Online; Accessed 7-January-2020].

[7] A. Team, "Trinity Platform Introduction and Usage Model," 2015, los Alamos National Laboratories, number LA-UR-15-26834.

[8] 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)*. Chengdu, China: IEEE, 2017, pp. 2424–2428.

[9] 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 *Int. Parallel Data Systems Workshop*. IEEE, Nov. 2020.

[10] 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 Trans. Storage*, vol. 7, no. 3, Oct. 2011.

[11] F. Z. Boito, "Estimation of the impact of I/O forwarding on application performance," Inria, Research Report RR-9366, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02969780

[12] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - a temporary distributed file system for HPC applications," in *IEEE Int. Conference on Cluster Computing (CLUSTER)*. Belfast, UK: IEEE, Sep. 2018, pp. 319–324.

[13] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS—A temporary burst buffer file system for HPC applications," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 72–91, 2020.

[14] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. E. Papka, R. Ross, and K. Yoshii, "Accelerating I/O Forwarding in IBM Blue Gene/P Systems," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. USA: IEEE Computer Society, 2010, pp. 1–10.

[15] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization Techniques at the I/O Forwarding Layer," in *2010 IEEE International Conference on Cluster Computing*. Heraklion, Crete: IEE, 2010, pp. 312–321.

[16] J. L. Bez, F. Z. Boito, L. M. Schnorr, P. O. A. Navaux, and J.-F. Méhaut, "TWINS: Server Access Coordination in the I/O Forwarding Layer," in *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. St. Petersburg, Russia: IEEE, March 2017, pp. 116–123.

[17] F. Z. Boito, R. V. Kassick, P. O. A. Navaux, and Y. Denneulin, "Automatic I/O scheduling algorithm selection for parallel file systems," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 8, pp. 2457–2472, 2016.

[18] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, "Terascale direct numerical simulations of turbulent combustion using s3d," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001, jan 2009.

[19] W.-k. Liao and A. Choudhary, "Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols," in *2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Austin, Texas: IEEE Press, 2008.

[20] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of Leading HPC I/O Performance Using a Scientific-Application Derived Benchmark," in *2007 ACM/IEEE Conference on Supercomputing*, ser. SC'07. New York, NY, USA: ACM, 2007.

[21] A. Ching, Wu-chun Feng, Heshan Lin, Xiaosong Ma, and A. Choudhary, "Exploring I/O Strategies for Parallel Sequence-Search Tools with S3aSim," in *15th Int. Conference on High Performance Distributed Computing*. Paris, France: IEEE, June 2006, pp. 229–240.

[22] 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. USA: USENIX Association, 2019, pp. 265–279.