Mapping and Scheduling HPC Applications for Optimizing I/O

Jesus Carretero
University Carlos III
Leganés, Spain
jcarrete@arcos.inf.uc3m.es

Emmanuel Jeannot
Inria Bordeaux
Talence, France
emmanuel.jeannot@inria.fr

Guillaume Pallez
Inria Bordeaux
Talence, France
guillaume.pallez@inria.fr

David E. Singh
University Carlos III
Leganés, Spain
desingh@arcos.inf.uc3m.es

Nicolas Vidal
Inria Bordeaux
Talence, France
nicolas.vidal@inria.fr

ABSTRACT

In HPC platforms, concurrent applications are sharing the same file system. This can lead to conflicts, especially as applications are more and more data intensive. I/O contention can represent a performance bottleneck. The access to bandwidth can be split in two complementary yet distinct problems. The mapping problem and the scheduling problem. The mapping problem consists in selecting the set of applications that are in competition for the I/O resource. The scheduling problem consists then, given I/O requests on the same resource, in determining the order to these accesses to minimize the I/O time. In this work we propose to couple a novel bandwidth-aware mapping algorithm to I/O list-scheduling policies to develop a cross-layer optimization solution.

We study this solution experimentally using an I/O middleware: CLARISSE. We show that naive policies such as FIFO perform relatively well in order to schedule I/O movements, and that the important part to reduce congestion lies mostly on the mapping part. We evaluate the algorithm that we propose using a simulator that we validated experimentally. This evaluation shows important gains for the simple, bandwidth-aware mapping solution that we provide compared to its non bandwidth-aware counterpart. The gains are both in terms of machine efficiency (makespan) and application efficiency (stretch). This stresses even more the importance of designing efficient, bandwidth-aware mapping strategies to alleviate the cost of I/O congestion.

KEYWORDS

I/O scheduling, I/O contention, Cross-layer optimizations

1 INTRODUCTION

High-performance computing (HPC) is a key technology for simulating and understanding complex phenomena in many scientific domains including physics, chemistry, biology, life sciences, materials, climate, and geosciences. Along the years, large-scale scientific infrastructures have been mostly designed to maximize the parallel compute efficiency of scientific simulations. The advent of data-intensive computing applications, including high-performance data analytics (HPDA) and deep learning (DL), as well as the huge increase of data in scientific computing currently changes this compute-centric view. This growing demand for data processing is accompanied by disruptive technological progress of the underlying storage technologies. As a result, upcoming exascale HPC systems are transitioning from a simple HPC storage architecture, consisting of a parallel back-end file system and archives often based on tapes, towards a multi-tier storage hierarchy that includes node-local non-volatile main memory (NVMM) with a performance close to DRAM, NVMe-based SSDs inside compute nodes with a bandwidth of many GBytes/s, SSDs on I/O nodes, parallel file systems, campaign storage, and archival storage.

Despite this architectural shift, the usage of the I/O stack is not optimal since end users lack the information about the state of the HPC resources and the I/O accesses of the multiple applications running in a supercomputer. As a result, opportunities for global I/O optimization are missed mostly due to uncoordinated data management, which often leads to redundant data movement, large I/O accesses contention and delayed end-to-end performance. In order to optimize the I/O performance of multiple applications accessing shared I/O resources, we have to address two main problems: First, we need to allocate applications to platform partitions; Secondly, we manage the I/O data in the system by scheduling the accesses that are susceptible of creating I/O conflicts.

Trying to cope with this problem, in a preliminary study we evaluated the inter-application I/O interference in PlaFRIM cluster for three representative use cases. Each compute node of PlaFRIM consists of two 12-core Intel Xeon E5-2680 processors and 128GB RAM. The nodes are connected by Infiniband QDR TrueScale at 40Gb/s, and the filesytem is Lustre configured with one metadata server, and four object storage servers. We evaluated the system I/O performance by executing use-cases under two different configurations: single application and multiple applications. In single application one program was executed exclusively, having full access to complete I/O bandwidth provided by the system. In contrast, with multiple applications, several program instances are executed simultaneously (accessing to different files) and competing for the I/O resources.

Figure 1 shows the I/O bandwidths of each use-case and configurations. Use cases A and B corresponds to an MPI program that writes a distributed matrix in a file using non-collective calls. More precisely, in use case A, each process writes the data consecutively, whereas in use case B a striped write access is performed with a stride size of 195 MB. Use case C is the NAS BTIO simple benchmark, which also uses non-collective MPI calls for accessing the file. In this use-case the I/O has a reduced data granularity. Because of this, use case C has a smaller I/O bandwidth than the other counterparts.

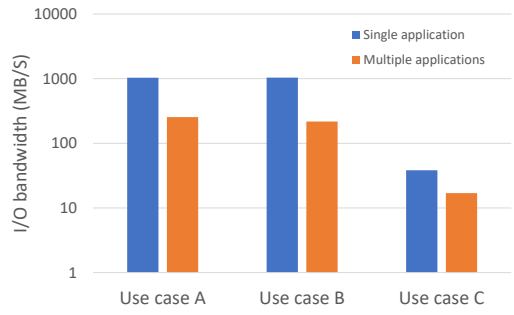


Figure 1: I/O bandwidth comparison for single and multiple applications of the three use cases. In use cases A and B an MPI program that writes a distributed matrix in a file using non-collective calls. Each application uses 96 processes for writing a 47.7GB file. With multiple applications, four instances of the same application are executed simultaneously. Use case C corresponds to the NAS BTIO simple benchmark that operates with an 1.6GB file using 64 processes. With multiple applications, two instances of the same application are executed simultaneously.

Note that for all the use cases the I/O bandwidth degrades when multiple applications are executed. We define this degradation d as the percentage of bandwidth that is lost when multiple applications are being executed. More formally, $d = 1 - \frac{\sum BW_{multi}}{BW_{single}}$ where BW_{single} is the I/O bandwidth for single application configuration and $\sum BW_{multi}$ is the aggregated bandwidth for multiple application configuration. In our experiments we ran 4 applications are the same time and we saw a, degradations of 2%, 16% and 11%, for use cases A, B and C, respectively. There are many reasons [31] for this degradation (that does not always occur) that are related to the complex interaction between the applications and all the levels of the I/O subsystem. One of the main factors occurs when the storage servers receive requests from multiple applications: One is that it is necessary to access to more locations in the hard disk surfaces, reducing the access locality and I/O performance; Another reason is the contention at I/O node-level, given that the raid servers are potentially connected with all the I/O nodes.

Based on these results we conclude that in some contexts, simultaneous inter-application I/O should be avoided. This is the idea behind the CLARISSE I/O scheduling control protocol [16]. Using a publish-subscribe protocol, when multiple applications access to the disk, only one is granted to perform the I/O. The remaining ones are delayed waiting for the I/O completion. In the first contribution of this work, we address this problem by developing and comparing different I/O scheduling policies that include different criteria for selecting the application that performs the I/O first.

Based on these results we conclude that, in some contexts, simultaneous inter-application I/O should be avoided. In this paper we propose a new solution aimed at reducing I/O interference via globally coordinated I/O access operation by studying two problems. The mapping problem consists in selecting the set of applications that are in competition for the I/O resource. The scheduling problem consists then, given I/O requests on the same resource, in determining the order of these accesses to minimize the I/O time. The main contributions of this work are a mathematical model and a packing algorithm to optimize the mapping of applications compute nodes to I/O nodes as well as a thorough experimental study of several I/O scheduling policies to order sequences of I/O operations that must be executed through each I/O node. Then we designed and validated a simulator to perform the evaluation at a larger scale. We then used this simulator to perform additional evaluations of the impact of the mapping strategy. The evaluation results with a single I/O node, show that our cross-layer approach greatly reduces the stretch of the machine while slightly degrading the makespan compared to the standard First-Fit algorithm. Meanwhile, with several partitions (and I/O nodes), our bandwidth-aware strategy performs better for both metrics.

The rest of the paper is organized as follows. Section 2 depict the architecture of the proposed framework. Section 3 reviews the works related to our system. Section 4 presents a mathematical model of the problem considered. Section 5 discusses some strategies for the I/O scheduling and the mapping problems. Section 6 presents experimental evaluations of both solutions. Finally, Section 7 summaries the main conclusions from our work and provides future directions.

2 ARCHITECTURE OVERVIEW

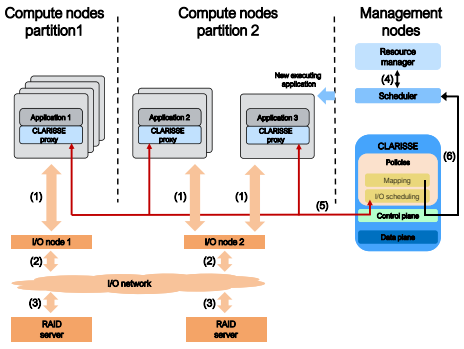


Figure 2: Framework overview.

Figure 2 shows the architecture of the proposed framework. The computational resources are divided into management nodes, that execute the software management tools, and compute nodes, that execute the applications. In this figure, three running applications with 4, 2 and 1 processes are illustrated. Using a similar scheme than in large HPC infrastructures, the compute nodes are divided into partitions (two in the figure). Each partition is associated to one I/O node that is responsible for managing the application I/O accesses (1) and translating them into requests to the storage servers (2) that are subsequently sent to the disks (3).

On the management side, the administration tools include a resource manager for monitoring the system and allocating compute nodes for new applications. The resource manager communicates with the application scheduler (4), responsible for determining what is the next application to be executed. The third component is CLARISSE [16], a middleware for enhancing I/O flow coordination and control in the HPC systems. CLARISSE decouples the policy, control, and data layers of the I/O stack software in order to simplify the task of globally coordinating the parallel I/O on large-scale HPC platforms. In the current implementation of CLARISSE, it incorporates a public-subscribe protocol in the control plane for coordinating the I/O, and several I/O scheduling policies in the policy layer. Each running application includes a CLARISSE proxy that wraps the application I/O calls and communicates with CLARISSE via the control plane (arrow 5). By means of this channel, the I/O scheduling policies can impose a multi-criteria I/O access order to the running application based on different performance metrics.

In addition, the control plane includes a communication line with the scheduler (arrow 6). The policy layer includes mapping techniques that guide the scheduler to allocate the new executing application in certain partitions, with the aim of reducing the number of conflicts accessing the I/O node.

As an illustrative example, let's assume that applications 1 and 2 are initially running in the system, and application 1 is more I/O intensive than application 2. Then, application 3, that is also I/O intensive, is ready to be executed. In order to balance the accesses to both I/O nodes, the mapping policy determines that is better to place application 3 in partition 2, avoiding risk of contention in the I/O node related to application 1. In a second step, when all the applications are being executed, application 2 and 3 compete for the I/O resources in the same partition. Note that conflicts in the access to the second I/O node may arise. Using the new CLARISSE's I/O scheduling policy introduced in this paper, the I/O accesses of both applications are coordinated with the aim of reducing the number of conflicts.

3 RELATED WORK

As exemplified in the introduction, the effect of conflicting accesses on the I/O subsystem is a well-known problem in HPC infrastructures [11, 23, 31]. Recently the load imbalance In this context, the I/O performance is difficult to model due to the complex interaction between different interfering applications. There are different alternatives at different levels of the architecture that reduce the effect of this problem.

The first intuitive solution is to use architectural enhancements such as Burst-Buffers [19]. These solutions have been widely studied to mitigate inter-application I/O interferences [2, 27, 28]. Sizing strategies have been studied [2], as well as buffer placement (shared or distributed) [3, 18]. Tang et al. [27] have studied draining strategies and have shown that the natural reactive strategy to empty the buffer as soon as possible can lead to severe degradation. Aupy et al. [2] have shown theoretically that only emptying the buffer when it is at least 15% full do not lead to significant delays compared to the reactive strategy, however it may mitigate the issues raised by the work of Tang et al. [27] Finally, as was shown in the recent work by Aupy et al. [3], to be efficiently used the buffers still need to be coupled with I/O management strategies.

A second solution is to use the elasticity of applications or file system levels. Singh et al. [25] present a strategy that combines I/O conflict prediction and application malleability. By means of prediction, the future I/O conflicts are forecast. Based on that, the time of the next I/O phase is shifted in order to avoid the I/O conflict. This is carried on by dynamically changing the number of processes of the conflicting application (by means of malleability). AHPIOS [17] is a light-weight ad-hoc parallel I/O system with elastic partitions that can scale up and down with the number of storage resources. Other works such as those of Lim et al. [21] or Cheng et al. [9] present elastic solutions using Hadoop Distributed File System (HDFS). SpringFS [30] presents an elastic filesystem for Cloud computing platforms.

Finally, the solution closest to our work is to schedule the I/O of applications in order to mitigate interference. Several works have tackled this problem. Some approaches [13, 32] use priority function scheduling at the I/O node level: they consider applications already mapped on a machine, sharing some I/O bandwidth. Several priority functions have been proposed by Gainaru et al. [13] to optimize either the equity between applications or the machine throughput. Another recent approach developed by Aupy et al. [4] is to look at structural properties of the applications (such as a periodic behavior in I/O communications) to develop more advanced schedules. In TWINS [7] the access to the I/O nodes is coordinated at the I/O forwarding layer, reducing the contention. Another similar approach is ASCAR [20], that uses traffic controllers on storage clients to detect I/O congestion and introduces traffic rules to reduce it. AIS [22] is an I/O-aware scheduler that performs an offline analysis of the I/O traffic for identifying I/O characteristics of the applications. Based on that, the applications are scheduled avoiding I/O conflicts. Other works that use models to predict and avoid the application I/O interference are [1, 10, 26].

Finally, there has been some recent work trying to incorporate several dimensions (such as I/O needs and compute nodes) into batch schedulers. On the more theoretical side, several work have started to incorporate multiple dimensions to a scheduling problem [14, 29]. Bleuse et al. [8] have considered geometrical constraints where, given some network topologies, they try to schedule applications while respecting their request both on the number of compute nodes and on the number of dedicated I/O nodes. To the best of our knowledge, there is no specific work to consider the scheduling of compute nodes while considering possible interference on I/O. On the practical side, Herbein et al. [15] studied incorporating I/O awareness in batch scheduler strategies. To develop a solution, they considered a simple model where all jobs had an I/O need proportional to the number of nodes needed. Their observation was that taking I/O into account reduced job performance variability.

4 MODEL

In this section we present a mathematical model of the problem considered. The machine model behavior has been verified experimentally to be consistent with the behavior of Intrepid and Mira,

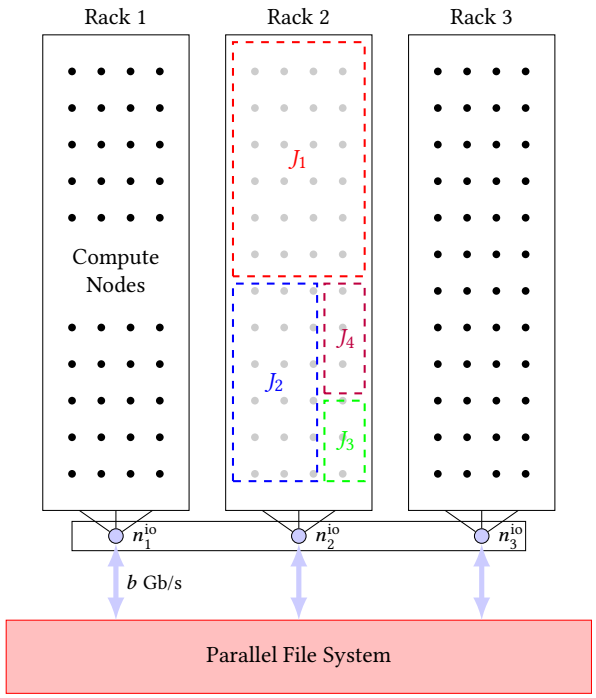


Figure 3: Schematic of the architecture. Jobs J_1, J_2, J_3 and J_4 compete for the bandwidth available on n_2^{io}.

supercomputers at Argonne [13], as well as with a supercomputer at Mellanox [4].

4.1 Machine Model

We consider a parallel platform structured as follows: R I/O nodes $(n_1^{io}, \ldots, n_R^{io})$ are available to perform I/O operations from the compute nodes to the parallel file system. Given $j \in \{1, \ldots, R\}$, each I/O node n_j^{io} has a bandwidth b_j for these operations, which is shared among P_j compute nodes (for a total of $\sum_{j=1}^{R} P_j$ compute nodes).

In this work we assume that the I/O bandwidth is homogeneous, that is, $\forall j, b_j = b$, as well as the number of compute nodes associated to each I/O node ($\forall j, P_j = P$). We represent this architecture model on Figure 3.

4.2 Applications

We consider a batch of scientific applications that need to run simultaneously onto the parallel platform. Applications consists of a series of consecutive *non-overlapping* phases: (i) a compute phase (executed on the compute nodes); (ii) an I/O phase (a transfer of a certain volume of I/O using the available I/O bandwidth) which can be either reads or writes.

Formally, we have a set of n jobs $\{J_1, \ldots, J_n\}$. Each job J_i requests Q_i compute nodes for its execution. J_i consists of n_i successive, blocking and non-overlapping operations: (i) $W_{i,j}$ (a compute operation that lasts for a time $w_{i,j}$); $V_{i,j}$ (an I/O operation that consists in transferring a volume $v_{i,j}$ of data). Therefore, if the bandwidth available to J_i to transfer its I/O to the PFS is equal to b, the time T_i needed for the total execution of J_i is:

$$T_i(b) = \sum_{j \leq n_i} w_{i,j} + \frac{v_{i,j}}{b}. \tag{1}$$

However, in general the I/O bandwidth is shared amongst several applications and it may incur delays to the execution of J_i.

4.3 Optimization problem

In this work, we consider that each job must be scheduled on the compute nodes associated to a single I/O node (contiguity), but that the I/O nodes can be shared amongst several applications. In addition, following the motivational example on I/O interference, we do not allow simultaneous bandwidth sharing amongst applications (i.e. on a given I/O node, only one application is performing I/O at the same time), and we do not allow preemption of I/O (once an application has started to perform I/O, it has to finish its transfer).

A *schedule* is the solution of two allocation problems:

- The *mapping* problem, which consists in choosing for each application the allocation of compute nodes (as depicted in Figure 3);
- The *scheduling* problem, which, for a given I/O node, consists in scheduling the sequences of I/O operations that must be executed through this node (hence of the applications mapped on the compute nodes associated to this I/O node).

We define several objectives. Given a schedule, each job J_i is released at time r_i and finishes its execution at time C_i.

The *stretch* ρ_i of J_i is the ratio between the minimal execution time and the actual execution time:

$$\rho_i = \frac{\sum_{j \leq n_i} w_{i,j} + \frac{v_{i,j}}{b}}{C_i - r_i} \tag{2}$$

(where b is the maximum available I/O bandwidth). A stretch of 1 means that the application is not impacted by the other applications running on the system. A stretch of 2 means that due to I/O contention, the application takes twice as long to execute as it would normally. Typically the stretch is an objective more *user* oriented.

The *makespan* $C_{\max}$ of a schedule is given as the end of the last execution:

$$C_{\max} = \max_i C_i \tag{3}$$

Typically, the makespan is an objective more *platform* oriented: with a fixed amount of work, the work over the makespan is the platform utilization.

Finally, our general optimization problem is the following. Given a set of jobs $J_1, \ldots, J_n$ and a platform with R I/O nodes, each with a bandwidth b to the PFS, and connected to P compute nodes. Find a schedule that minimizes either the total makespan, or that minimizes the maximum stretch ($\max_i \rho_i$). We call the general setup of these problems Hpc-IO, and, depending on the function to optimize: MS-Hpc-IO or ρ-Hpc-IO.

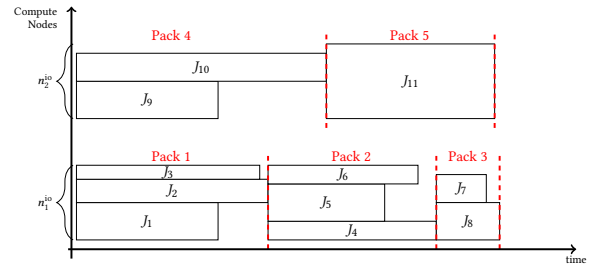
When the number of I/O nodes is equal to 1, this reduces to finding the right allocation of I/O for the different applications. We call this subproblem IO-Sched. Given an allocation to a solution of Hpc-IO, one can compute then independently the I/O scheduling solution using an algorithm to IO-Sched.

Note that both the Hpc-IO problem and the IO-Sched problems are NP-hard: Hpc-IO easily reduces to the multi-processor scheduling problem; IO-Sched has been shown to be NP-hard by Gainaru et al. [13].

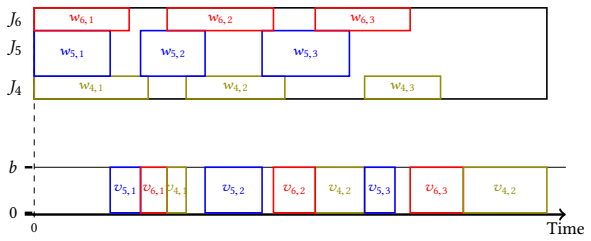
5 PACK SCHEDULING TO SOLVE HPC-IO

In this work, we focus on a special type of solutions to Hpc-IO, specifically *Pack scheduling* algorithms. In Pack scheduling [6], the jobs are partitioned into series of packs, which are then executed consecutively. Tasks within each pack are scheduled concurrently and a pack cannot start until all tasks in the previous pack have completed (see Figure 4).

Pack scheduling has been advocated [12, 24] as it provides an easier and flexible mean of designing and implementing novel algorithms while providing significant savings.



(a) The mapping of 11 applications into packs



(b) The scheduling of I/O operations (bottom) within Pack 2. Computations are allocated on dedicated compute nodes and can start as soon as the I/O is transfered, but I/O operation share the available bandwidth b and can delay applications.

Figure 4: A solution to Hpc-IO of eleven applications on a machine with two I/O nodes.

In this section we discuss some strategies for IO-Sched and Hpc-IO.

5.1 Policies for IO-Sched with a single pack.

Several works have considered the problem IO-Sched. In general there are two approaches list-scheduling heuristics [5, 13], or more involved pattern-based algorithms taking into account structural knowledge of the applications (such as their periodicity) [4].

In this work, we focus on list-scheduling based solutions. *List scheduling* policies consists in scheduling available I/O operations following a priority order as soon as a resource is available. They are an interesting way to solve IO-Sched given the recent results presented by Aupy et al [5]:

Theorem 1 ([5, Theorem 6]). *Any list-scheduling algorithm is a 2-approximation for MS-*IO-Sched *and this ratio is tight.*

Specifically, we consider the following natural priority orders that gives 8 different list-scheduling I/O policies:

(1) Lowest ID: the scheduler picks the application with lowest system ID amongst those which requested I/O.
(2) Longest I/O: the scheduler picks the application which performs the longest I/O phases (resource occupation).
(3) Shortest I/O: the scheduler picks the application which performs the shortest I/O phases (avoid long wait).
(4) Shortest remaining: the scheduler picks the application which is expected to have the least remaining work to do (free the machine ASAP).
(5) Longest remaining: the scheduler picks the application which is expected to have the most remaining work to do (platform occupation).
(6) FIFO: Applications are sorted by increasing I/O request time.
(7) Bandwidth oriented: applications with the lowest ratio between I/O time and execution are advantaged (fairness).
(8) Stretch oriented: applications with the worse stretch are scheduled first (fairness).

5.2 Algorithms for the mapping problem

In this section we provide an algorithm for the mapping problem that takes into account both the need for bandwidth and processor sharing between applications allocated to an I/O node.

The algorithm works in two steps: a first step partition the jobs into packs, while the second step schedules the pack on the different I/O nodes. The intuition of the partitioning algorithm is the following: we sort them by decreasing no contention execution time (as given by Equation 1). Then we create packs following a *Best-Fit* procedure, that is a procedure that schedules greedily the next application in the first pack where it would fit with respect to an I/O constraint and a processor constraint, and otherwise that creates a new pack.

Precisely, to account this constraints, given a pack of jobs Pack, we define:

- $P^{\mathsf{Pack}} = \sum_{i \in \mathsf{Pack}} Q_i$, the number of processors used by the pack;
- $T^{\mathsf{Pack}} = \max_{i \in \mathsf{Pack}} \sum_{j \leq n_i} w_{i,j} + \frac{v_{i,j}}{b}$, the minimal length of the pack;
- $L^{\mathsf{Pack}} = \frac{1}{b \cdot T^{\mathsf{Pack}}} \sum_{i \in \mathsf{Pack}} \sum_{j \leq n_i} v_{i,j}$, the average I/O occupation of the pack.
- S the *I/O sensibility* for pack creation. It is a parameter of the algorithm, by default $S = 1$.

The processor constraint to be respected is: $P^{\mathsf{Pack}} \leq P$. The I/O constraint is $L^{\mathsf{Pack}} \leq S$. Intuitively, this constraint tries to ensure that there is enough I/O bandwidth to perform all I/O operations with minimal delay.

We formalize this in Algorithm 1.

Algorithm 1: Pack-partitioning algorithm.

procedure Make-Pack($J_1, \ldots, J_n$,S)
begin
  Assume the jobs are sorted in decreasing order of $T_i(b) = \sum_{j \le n_i} w_{i,j} + \frac{v_{i,j}}{b}$;
  Let S_P be a set of packs sorted by decreasing value of P^{Pack} (empty initially);
  for $i = 1$ *to* n **do**
    Find the first Pack in S_P such that

$$\sum_{j \le n_i} v_{i,j} \le (S - L^{\text{Pack}}) \cdot bT^{\text{Pack}} \quad (4)$$

$$Q_i \le P - P^{\text{Pack}} \quad (5)$$

    Fit J_i in this pack;
    If there is no such pack, create an empty pack;
  end
  return S_P
end

6 EXPERIMENTAL RESULTS

In this Section we present experimental evaluations of both I/O scheduling and node mapping algorithms.

For the experiments, we have used Tucan cluster consisting of compute nodes with Intel(R) Xeon(R) E7 with 12 cores and 128GB of RAM, interconnected with a 10 Gbps Ethernet. We used the workload depicted in Table 1.

The evaluation is performed in several steps: in Section 6.1 we study the impact of various I/O scheduling policies on IO-Sched when the applications are already mapped on the machine. Then we study the impact of the mapping algorithm in Section 6.2 on Hpc-IO.

6.1 Impact of list-scheduling policies on IO-Sched

6.1.1 Experimental setup. The workload is composed of five synthetic applications based on Jacobi decomposition and having different I/O, computational needs and number of iterations. Each application is configured to perform the CPU and I/O phases with a particular intensity level, that produces a different duration of the phases. The total execution load[1] is the same for all the applications. This means that if each of them would be executed exclusively their duration would be similar. All these applications have been executed in the proposed framework depicted in Figure 2. This means that the application I/O phases are intercepted and coordinated by CLARISSE according to the I/O scheduling policy.

6.1.2 Result analysis and discussion. To see if we attain a steady state in terms of stretch and makespan we have run the workload described in table 1 while increasing its number of *iterations batches*[2]. In Figure 7, we plot the stretch of the different strategies when varying the number of iteration batches. We see that after some iterations the performance converges towards a steady state. In general, applications 3 and 4 are the ones who have the largest stretch. This is explained by the fact that these applications have the lowest computational intensity and hence are more sensible to I/O delay. We can also notice that some policies have a large discrepancy in terms of stretch (for instance "longest I/O" or the "shortest remaining"). They exhibit applications with a stretch close to one (i.e. not delayed) and others with a very high stretch (up to 1.8). This is typical of a starvation situation where some applications are highly favored while others are able to perform I/O only when no other one is requesting access.

[1]Number of iterations × (computational Intensity + I/O Duration)
[2]An iteration batch is a multiplicative factor that increases of reduces the application iteration number. For instance, a value of 2, makes the application run the double number of iterations (for instance, 500 for application 1).

Application id	Matrix size	CPU duration (s)	I/O duration (s)	Iterations
1	5000	32	10	250
2	5000	16	5	500
3	5000	8	2.5	1000
4	5000	8	2.5	1000
5	5000	16	5	500

Table 1: Workload description for the experiment on the Tucan machine (for Figure 7 to 9).

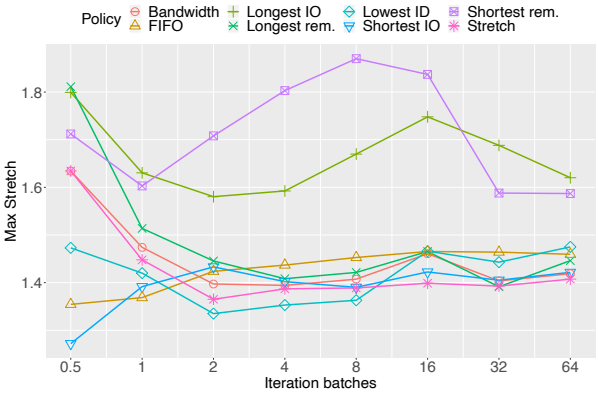


Figure 5: Max Stretch of the different policies when varying the number of iteration batches.

In Figure 5, we plot, for all policies, the maximum stretch of all applications. The maximum stretch measures the worst case for all the applications. We see that the "longest I/O" or "shortest remaining" policies behave the worst as they tend to exhibit starvation behavior. On the other side, several heuristics have a better behavior than the FIFO policy, which is the basic policy for this problem. In particular heuristics favoring applications that have a high stretch or the lowest ratio I/O vs. execution time behave much better than FIFO. Indeed, these two policies tend to greedily improve the fairness among the different applications.

In Figure 8, we plot the makespan of all the policies compared to the default policy (FIFO). Surprisingly, we see that heuristics favoring the maximum stretch are also the ones that exhibit good performance in terms of makespan. Indeed, optimizing the maximum stretch requires to optimize resource utilization in order to

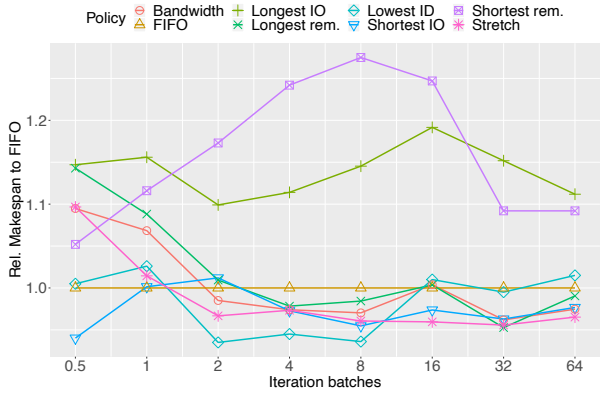


Figure 6: Relative Makespan to FIFO of the different policies when varying the number of iteration batches.

avoid applications to be stalled. In addition, optimizing resource usage of each of them also optimize the time they spent in the system.

At the end, we see that the two policies that tend to sacrifice a resource in the short term behave poorly. Indeed, the "longest I/O" policy favors I/O and "Shortest remaining" favors computation: for all these cases such unbalanced policies lead to very bad fairness among applications. On the other hand, balanced policies that take into account the I/O and the fairness offer very good results compared to the FIFO policy. This is especially the case for the "Stretch oriented", the "Shortest I/O" and the "longest remaining" strategies.

6.2 Analysis of solutions for Hpc-IO

In the previous section, we analyzed several I/O scheduling policies for the IO-Sched problem. We now try to solve the more general Hpc-IO problem by comparing the different mapping algorithms presented in Section 5.2.

First we present the experimental setup (machine simulator and synthetic application generation). Then, the analysis takes place in two steps: first we study deeply the model with a single I/O node, then we extend it on a machine with multiple I/O nodes. Finally, in this section, we consider a machine with $P = 2048$ compute nodes per I/O nodes and one to five I/O nodes. We normalize the I/O bandwidth by setting $b = 1$.

6.2.1 Machine Simulation. To perform the analysis of this section at a larger scale (both in terms of number of applications and size of the target machine) we have designed a simulator able to test larger settings than those presented in Section 6.1.

The simulator tool is integrated in the execution framework, which means that it is connected with the application scheduler as well as CLARISSE. Using the workload provided by the scheduler, the simulator is able to compute the different CPU and I/O phases of each application by means of a fixed-increment time progression. In case of simultaneous I/O accesses, the CLARISSE's scheduling policies are used to determine the I/O access order. By means of this scheme, it is possible to simulate a given workload in the execution framework reusing the same software logic as the one used in the actual workload execution.

In order to validate the simulator, we have rerun the experiments presented in Figure 7. We have compared the results on both cases (simulator vs. real machine: Tucan). We have observed that for all heuristics the order between applications, for the stretch and the makespan, is kept. Moreover quantitatively the values are extremely similar: the largest difference between the simulation and the real execution, for 64 batches is 15%, with a geometric mean less than 5%. Due to lack of space, we do not present all the comparisons, but we give the case for the Longest I/O policy in Figure 9 as it is a heuristic that displays high discrepancy in terms of Stretch.

6.2.2 Synthetic workload generation. To perform the evaluation at a large scale, we propose a protocol to generate different and numerous workloads. We based the design on our protocol on two elements:

- The generated workload needed to be representative of I/O behaviors (hence including applications with high I/O load as well as applications with low I/O load).
- Then, we intuited that the impact of the different algorithm was correlated to a general property of the concurrent applications, namely the *average I/O occupation*. As seen in Section 5.2 (L^{Pack}), this property is linked to the schedule as it takes into account the makespan. For the workload generation, we use a theoretical upper-bound for the average I/O occupation that assumes that there is no gap creating by the schedule. Mathematically, this writes as:

$$\alpha = \frac{P.\sum_{i=1}^{n}\sum_{j\leq n_i} v_{i,j}/b}{\sum_{i=1}^{n}(Q_i \cdot T_i(b))} \tag{6}$$

Based on these two preliminary elements, for each value $\alpha_{\text{gen}} \in \{0.5, 0.75, 1 \ldots 10\}$, we generate 10 workloads in the following way:

- We pick the proportion β of applications with low I/O load at random on $[0, 1]$ (0 meaning all applications have high I/O load, 1 meaning they all have low I/O load).
- We generate the number of applications of each workload uniformly at random in $[25, 100]$.

Then for each application, for simplicity we assume that they are periodic (i.e. for all $j \leq n_i$, $v_{i,j} = v_i$ and $w_{i,j} = w_i$), and:

- Their number of iterations n_i is chosen uniformly at random between 250 and 1000.
- w_i is chosen uniformly at random in $[10, 100]$.
- Applications I/O load (v_i/w_i, and ultimately v_i) is chosen:
  (1) Following a normal distribution of mean $\mu_1 = 0.1$, variance $\sigma_1 = 0.1$ truncated on the interval $[x, y]$ for applications of low I/O load;
  (2) Following a normal distribution of mean $\mu_2 = 0.9$, variance $\sigma_2 = 0.1$ truncated on the interval $[x, y]$ for applications of high I/O load.
- Finally, to compute the number of processors Q_i of each application, we use a distribution in the discrete set $\{2^j\}_{j=0\ldots11}$

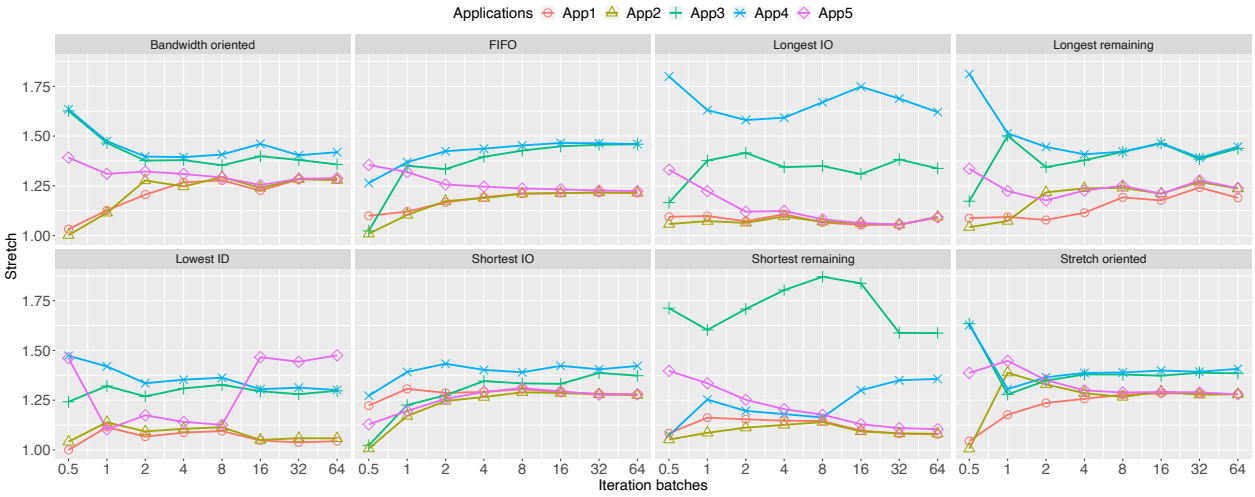


Figure 7: Stretch of the different policies when varying the number of iteration batches.

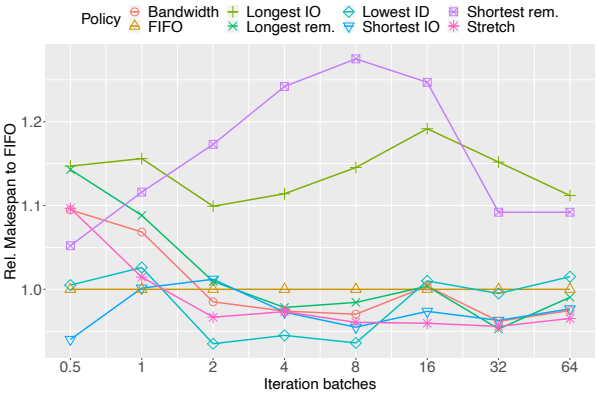


Figure 8: Relative Makespan to FIFO of the different policies when varying the number of iteration batches.

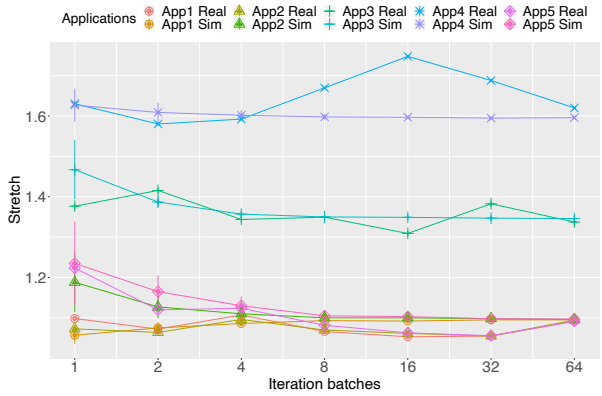


Figure 9: Comparison of Simulated vs real execution (on the Tucan Machine) of the Longest I/O policy

of mean[3]

$$\bar{Q} = \frac{P(\beta\mu_1 + (1-\beta)\mu_2)}{\alpha_{\text{gen}}(1 + \beta\mu_1 + (1-\beta)\mu_2)}.$$

This protocol ensures that we have a various set of workloads, covering different I/O load (α). Note that in the rest of the evaluation, for each workload, we use their actual I/O load as defined by Equation (6), and not the value of α_{gen} used for the generation. The precise code for the generation and execution of the workload is available at https://gitlab.arcos.inf.uc3m.es:8380/desingh/IOscheduling.git.

6.2.3 Evaluation with a single I/O node. In this section we study the overall problem HPC-IO and its solutions consisting in (i) the Make-Pack procedure (Sec. 5.2) including its sensitivity parameter S; (ii) once the packs are done, the different I/O policies.

The experiments are done with comparison to a baseline algorithm: First-Fit [6] for pack creation and FIFO for I/O policy (shown to be the most effective policy in Section 6.1). These two strategies are globally referred to as *First-Fit* in the following. As the baseline algorithm, the pack creation does not take into account the I/O operations of jobs but only their execution time when performed by themselves on the machine. One can observe that essentially this is the Make-Pack procedure when the sensibility $S = \infty$.

In this Section, we study two different pack creation algorithms:

- Make-Pack when the sensibility is set to $S = 1$ (referred to as *Sensibility=1* in the following): intuitively, this strategy tries to minimize the likelihood that there are delays due to I/O.

[3] $\bar{Q}$ is obtained by replacing in Equation (6) all values by their average value, which is not mathematically correct but that we use as a first approximation to generate the workload.

- Make-Pack when the sensibility is set to $S = \alpha$ (referred to as *Sensibility=I/O load* in the following). This is intended as a middle-case behavior that one would obtain if S varied.

Finally, we emulate the execution of the packs using the simulator described in Section 6.2.1 using a defined scheduling policy. As default scheduling policy, we use FIFO for both pack creation algorithms. In addition, we have also performed experiments with *I/O Sensibility=1* and with the "Longest remaining" policy which was proven to be very effective in Section 6.1.

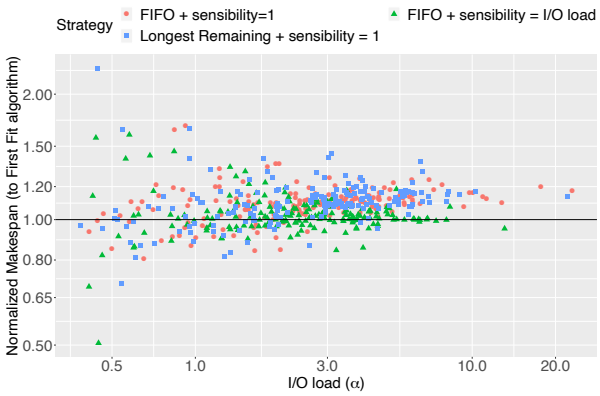


Figure 10: Comparison of makespan for different strategies

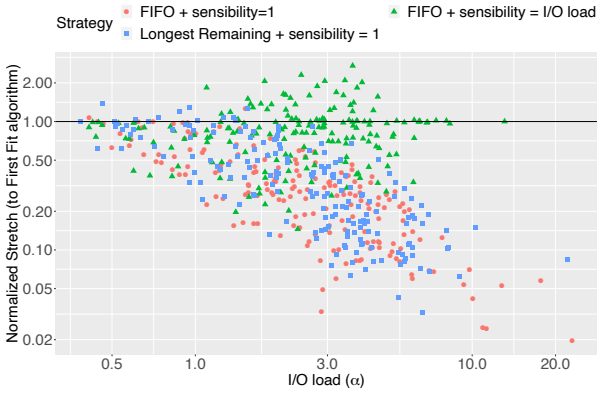


Figure 11: Comparison of stretch for different strategies

Overall performance. To study the performance, we study both makespan (Figure 10) and the stretch[4] (Figure 11) of the different solutions. The results are presented normalized to the respective performance of First-Fit, and we study them as a function of the I/O load α (Equation (6)).

In the standard configuration, with a FIFO scheduling policy and a Pack partitioning sensibility equal to 1, Figure 10 shows an average 10% overhead of our algorithm in terms of makespan while Fig: 11 show significant stretch improvement. The significant stretch improvement could be expected: with the sensibility set to 1 we reduce contention a lot, and intuitively our stretch stays close to 1. On the contrary, First-Fit has an average I/O occupation factor that increases potentially with α, hence increasing the contention and the stretch. Changing I/O scheduling policy does not seem to have an impact on these measures. Hence in the next evaluations we only consider FIFO policy.

[4]In this section, we use the average of the maximum stretch of each pack.

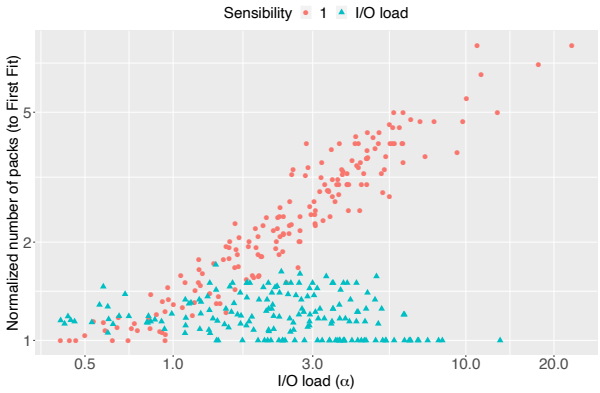


Figure 12: Normalized number of packs produced by the Pack Partitioning algorithm (relative to the First-Fit algorithm) for different sensibility

It was to be expected as bandwidth-aware heuristics produce more packs. Indeed, since we add more constraints on the packs (an I/O constraint, Eq (4)), when this constraint is saturated and if it occurs before the processor constraint (Eq (5)), new packs are created. Fig. 12 shows how many more packs are produced by our algorithm compared to the First-Fit case when varying the I/O load. We see that, for a sensibility of one, the number of packs is increasing with the I/O load. This is due to the fact that when the I/O load is large the Pack Partitioning algorithm creates more pack to avoid I/O contention. When the sensibility is α, the ratio is much smaller and roughly constant because the sensibility determines how contention is avoided. Producing more packs has the advantage, however, of decreasing contention, which explains the improvement in stretch. The downside is increasing the number of packs and unused processors within packs.

As the stretch improvement is noticeable, we may consider that a sensibility of 1 for Pack Partitioning Algorithm is too pessimistic and leads to an unbalanced trade-off. Increasing the bandwidth limit as a function of α provides an alternative compromised, mitigating the makespan loss while maintaining stretch improvement most of the time.

In depth performance. For an in-depth performance evaluation, we focus on the FIFO policies.

We present in Figure 13 the ratio of the execution time as measured to an ideal one that one could predict with no contention would occur (essentially if the global I/O bandwidth was unbounded but keeping the individual I/O bandwidth bounded).

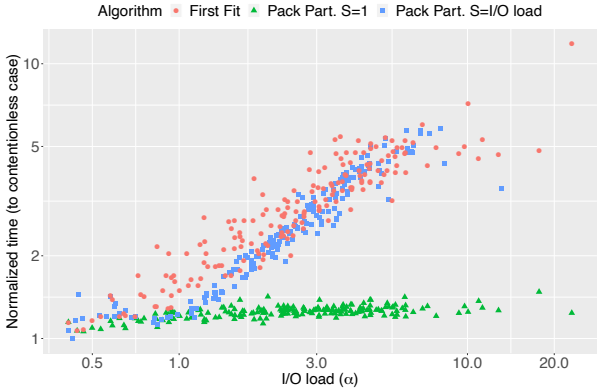


Figure 13: Processor time and robustness: pack algorithm vs. First-Fit

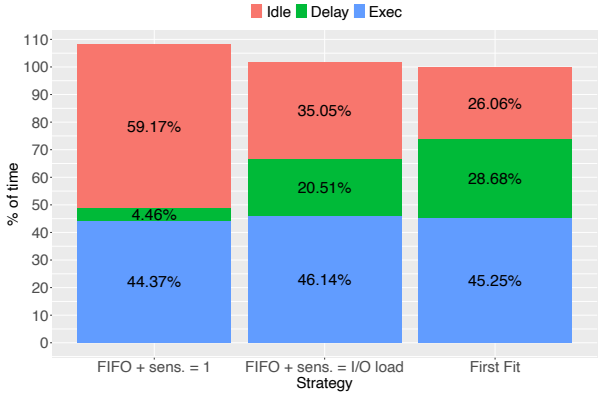


Figure 14: Average execution, idle and delay time (normalized) for different strategies

The interesting observation is that the execution of Pack Part ($S = 1$) is a lot closer to the ideal one (within 20%, while First-Fit can be as bad as 1000%). This provides more control on the execution. This is coherent with the results observed by Herbein et al. [15]. The version of Pack Part with $S = \alpha$ is close in behavior to that of First-Fit as one would expect since the congestion constraint is relaxed.

This results is even more interesting when we study for each compute node the average time that they spend: (i) doing *useful* execution (either they are doing some compute, or the application mapped on them is performing I/O), (ii) being delayed (the application mapped on them is waiting to perform I/O), or (iii) being idle (there is no application mapped on them, or the application mapped on them has finished working and the pack is waiting for some final applications). We plot these average time in Figure 14, normalized with respect to the average time of First-Fit, so that, if the input to all algorithms were identical the *Exec* time (useful execution time) would be identical. Here, the difference in *Exec* time for the three algorithms is an attribute of the randomness in the generation of workloads.

This figure is interesting because again, we observe that for a trade-off of 10% in makespan there is a transfer of 25% of the time from delay to idle. In addition, the additional 10% of time wasted is also moved to idle time. This gives other opportunities (such as turning-off nodes for energy consumption, using available nodes for backfilling operations with applications that do not need I/O etc), while the idle time due to contention is lost. This is another strong argument for bandwidth-aware scheduling policies, even if locally it reduces machine utilization, globally it provides an opportunity to improve it by a lot more than what is wasted.

6.2.4 Multiple I/O nodes. Previous experiments were done using only one partition the compute nodes with one I/O node: packs were executed sequentially. However, in many platforms, several I/O nodes belonging to different partitions/racks are available (see Fig. 3). In order to see the impact of this feature, we then studied the parallel execution of the computed packs.

To do so, given the assumption that I/O nodes do not interact with each other and that they have the same characteristics, we can simply allocate the packs computed by Algorithm 1. In this section we only consider the case where the sensibility $S = 1$, and we use FIFO as the default I/O policy. The pack allocation is done using the Largest Processing Time[5] (LPT) heuristic, where pack duration is the contention less execution time of its longest application.

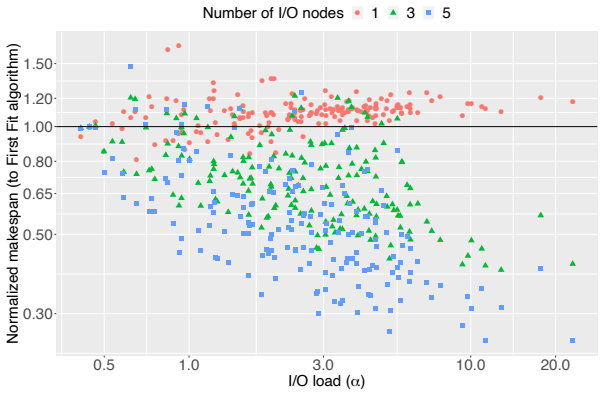


Figure 15: Comparison of the relative makespan of Pack Partitioning Algorithm (with sensibility =1) to the First-Fit algorithm with multiples I/O nodes.

Here, we use the FIFO I/O scheduling policy and the standard configuration of the Pack Partitioning algorithm (sensibility=1). Hence, the one I/O node case is the same as the *sensibility = 1* case of Figure 10.

We see that, when we increase the number of I/O nodes the relative makespan is decreasing. The geometric mean[6] of the 1 (resp. 3 and 5) I/O node(s) case is 1.09 (resp. 0.71 and 0.53). This

[5]We map the longest remaining pack on the I/O node on which it will finish the earliest
[6]We use the geometric mean instead of the arithmetic mean as we are dealing with ratios

means that in the five I/O nodes case our bandwidth-aware solution is, on average, twice as fast as the First-Fit algorithm!

The interpretation of this result is the following. With our Pack Partitioning, we have more but smaller packs than for the First-Fit case (see Fig. 12). Hence, providing a balanced allocation is easier in this case than for the First-Fit case where packs are less numerous but longer. Moreover, we are computing the pack allocation based on the estimated pack duration ignoring the contention. These durations, as shown in Fig. 13, are more precise in the Pack Partitioning case than in the First-Fit case. Hence, the load balancing computed is more accurate with our Pack Partitioning solution: allocation decisions are more robust and hence lead to better solutions.

Last, the I/O nodes are homogeneous and the stretch is a local metric of each individual pack. Therefore, the stretch performance of all algorithms does not depend on the number of I/O nodes and is exactly the same as the one depicted in Figure 11 for a single I/O node.

In conclusion, in a realistic setting where there are multiple I/O nodes, our Pack Partitioning algorithm outperforms the First-Fit algorithm both for the stretch and the makespan.

7 CONCLUSION

This paper addressed the issue of executing concurrent applications in a system with bounded I/O bandwidth. We presented a model, optimization framework as well as simple heuristics for the issue of allocating the applications depending on the I/O resources available on the machine. Through rigorous experiments, our first observation was that simple and fair list-scheduling policies seemed to perform better in the long run when it comes to scheduling I/O access. In addition, we presented a simple strategy to allocate applications together based on an approximation of their resource usage. Our evaluation of this strategy gave interesting results: with a single I/O node, they improved importantly the stretch of the machine while degrading slightly the makespan (or throughput). But a more in-depth study showed that this degradation was a consequence of a much better control of the waste (mostly by having more unoccupied resources instead of resources waiting). We have also studied the case where compute nodes are decomposed in several partitions/racks by increasing the number of I/O nodes. In this case our bandwidth-aware strategy performs better for both metrics.

In the future, several directions open up, both for the mapping of applications and the scheduling of I/O. With respect to the mapping, our natural first step will be to remove the pack constraint and see if we can design an efficient bandwidth-aware strategy. Another direction would be at simply enriching this Pack Partitioning policy with small job backfilling. Then in parallel, moving from exclusive-access I/O policies to policies where the bandwidth can be shared is a direction we intend to take. The difficulty here is to make the middleware ready to evaluate these strategies, maybe through containers.

ACKNOWLEDGEMENTS

This work was supported in part by the French National Research Agency (ANR) in the frame of DASH (ANR-17-CE25-0004). Some of the experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

REFERENCES

[1] Maicon Melo Alves and Lucia Maria de Assumpçao Drummond. A multivariate and quantitative model for predicting cross-application interference in virtual environments. *Journal of Systems and Software*, 128:150 – 163, 2017.
[2] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. What size should your buffers to disks be? In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 660–669. IEEE, 2018.
[3] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. Sizing and partitioning strategies for burst-buffers to reduce io contention. In *Parallel and Distributed Processing Symposium (IPDPS), 2019 IEEE International*. IEEE, 2019.
[4] Guillaume Aupy, Ana Gainaru, and Valentin Le Fèvre. Periodic i/o scheduling for super-computers. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 44–66. Springer, 2017.
[5] Guillaume Aupy, Emmanuel Jeannot, and Nicolas Vidal. Scheduling periodic i/o access with bi-colored chains: models and algorithms. 2019.
[6] Guillaume Aupy, Manu Shantharam, Anne Benoit, Yves Robert, and Padma Raghavan. Co-scheduling algorithms for high-throughput workload execution. *Journal of Scheduling*, 19(6):627–640, 2016.
[7] J. L. Bez, F. Z. Boito, L. M. Schnorr, P. O. A. Navaux, and J. 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)*, pages 116–123, March 2017.
[8] Raphaël Bleuse, Konstantinos Dogeas, Giorgio Lucarelli, Grégory Mounié, and Denis Trystram. Interference-aware scheduling using geometric constraints. In *European Conference on Parallel Processing*, pages 205–217. Springer, 2018.
[9] Zhendong Cheng, Zhongzhi Luan, You Meng, Yijing Xu, Depei Qian, Alain Roy, Ning Zhang, and Gang Guan. Erms: An elastic replication management system for HDFS. In *Cluster Computing Workshops, 2012 IEEE International Conference on*, pages 32–40. IEEE, 2012.
[10] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. Using formal grammars to predict I/O behaviors in HPC: The omnisc'io approach. *IEEE Transactions on Parallel and Distributed Systems*, 27(8):2435–2449, Aug 2016.
[11] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination. In *IPDPS - International Parallel and Distributed Processing Symposium*, pages 155–164, Phoenix, United States, May 2014.
[12] Eitan Frachtenberg, G Feitelson, Fabrizio Petrini, and Juan Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel and Distributed systems*, 16(11):1066–1077, 2005.
[13] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the i/o of hpc applications under congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1013–1022. IEEE, 2015.
[14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *SIGCOMM Comput. Commun. Rev.*, 44(4):455–466, August 2014.
[15] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–80, 2016.
[16] F. Isaila, J. Carretero, and R. Ross. CLARISSE: A middleware for data-staging coordination and control on large-scale HPC platforms. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 346–355, May 2016.
[17] Florin Isaila, Francisco Javier Garcia Blas, Jesus Carretero, Wei keng Liao, and Alok Choudhary. A scalable message passing interface implementation of an ad-hoc parallel I/O system. *The International Journal of High Performance Computing Applications*, 24(2):164–184, 2010.
[18] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan Vazhkudai, and Misbah Mubarak. Evaluating burst buffer placement in hpc systems. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2019.
[19] Anthony Kougkas, Matthieu Dorier, Rob Latham, Rob Ross, and Xian-He Sun. Leveraging burst buffer coordination to prevent i/o interference. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, pages 371–380. IEEE, 2016.
[20] Y. Li, X. Lu, E. L. Miller, and D. D. E. Long. Ascar: Automating contention management for high-performance storage systems. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–16, May 2015.
[21] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*,

pages 1–10. ACM, 2010.

[22] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai. Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829, Nov 2016.

[23] Tirthak Patel, Suren Byna, Glenn K Lockwood, and Devesh Tiwari. Revisiting i/o behavior in large-scale storage systems: the expected and the unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

[24] Manu Shantharam, Youngtae Youn, and Padma Raghavan. Speedup-aware co-schedules for efficient workload management. *Parallel Processing Letters*, 23(02):1340001, 2013.

[25] David E. Singh and Jesus Carretero. Combining malleability and i/o control mechanisms to enhance the execution of multiple applications. *Journal of Systems and Software*, 148:21 – 36, 2019.

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

[27] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari. Toward managing HPC burst buffers effectively: Draining strategy to regulate bursty I/O behavior. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 87–98, Sept 2017.

[28] S. Thapaliya, P. Bangalore, J. Lofstead, K. Mohror, and A. Moody. Managing I/O interference in a shared burst buffer system. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 416–425, Aug 2016.

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

[30] Lianghong Xu, James Cipar, Elie Krevat, Alexey Tumanov, Nitin Gupta, Michael A Kozuch, and Gregory R Ganger. Springfs: bridging agility and performance in elastic distributed storage. In *FAST*, pages 243–255, 2014.

[31] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Robert Ross, and Gabriel Antoniu. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In *IPDPS 2016 - The 30th IEEE International Parallel and Distributed Processing Symposium*, pages 750–759, Chicago, United States, May 2016.

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