IO-aware Job-Scheduling: Exploiting the Impacts of Workload Characterizations to select the Mapping Strategy

Emmanuel Jeannot, Guillaume Pallez, Nicolas Vidal

Journal Title
XX(X):1–13

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

I/O-aware Job-Scheduling: Exploiting the Impacts of Workload Characterizations to select the Mapping Strategy

Emmanuel Jeannot[1], Guillaume Pallez[1] and Nicolas Vidal[1,2]

Abstract

In high performance, computing concurrent applications are sharing the same file system. However, the bandwidth which provides access to the storage is limited. Therefore, too many I/O operations performed at the same time lead to conflicts and performance loss due to contention. This scenario will become more common as applications become more data intensive. To avoid congestion, job schedulers have to play an important role in selecting which application run concurrently. However I/O-aware mapping strategies need to be simple, robust and fast. Hence, in this paper, we discuss two plain and practical strategies to mitigate I/O congestion. They are based on the idea of scheduling I/O access so as not to exceed some prescribed I/O bandwidth. More precisely, we compare two approaches: one grouping applications into packs that will be run independently (i.e pack scheduling), the other one scheduling greedily applications using a predefined order (i.e. list scheduling).

Results show that performances depend heavily on the I/O load and the homogeneity of the underlying workload. Finally, we introduce the notion of characteristic time, that represent information on the average time between consecutive I/O transfers. We show that it could be important to the design of schedulers and that we expect it to be easily obtained by analysis tools.

Keywords

I/O scheduling, Congestion management, job scheduler

Introduction

One of the central problems of supercomputing is the allocation of jobs with different requirements on the shared resource. Research on Resource and Job* Management Systems (RJMS, aka Batch Schedulers) is extremely active for HPC as the smallest loss in system utilization can cost millions of dollars. Nowadays, for most RJMS, the mapping of applications onto HPC resources is mostly based on two inputs: the amount of computing resources required by the application and the duration of the reservation of such resources.

However, HPC centers face a significant shift of their workloads with the increase of Big Data, analytics and Machine Learning applications: the amount of data in scientific computation is continuously increasing. Generally, HPC applications perform alternatively computational and storage (I/O) phases† in the course of their execution Gainaru et al. (2015); Zhou et al. (2015). Applications share the storage system and it has been observed performance degradation due to concurrent access to such storage system Yildiz et al. (2016). Hence, to optimize such I/O intensive application, it is important to mitigate I/O congestion. In this work, we focus on how a job scheduler can fulfill part of this objective. However, as stated above, standard job schedulers are focused on computational needs. The goal of this work is therefore to discuss the impact of data-awareness on the design of resource management algorithms. Specifically, we focus on the incorporation of the I/O needs of applications, one of the current major bottlenecks in HPC systems, into the batch scheduling algorithms.

Among the different possible solutions to implement I/O-aware job scheduling, we must select strategies that are compatible with resources-centric ones and that, in order to be adopted, should have the following characteristics :

- scalable (thus have a low complexity);
- simple (easy to implement and understand);
- robust (do not require complex input to be effective).

For this reason, we focus this study on the following paradigms of the literature: pack-scheduling and list-scheduling (see Figure 1). *Pack-Scheduling* is a strategy that maps applications by packs, i.e. sets of applications that start at the same time. The next pack cannot start as long as the last application of the previous pack has not finished

[1]Inria, Univ Bordeaux, LaBRI, Talence, France
[2]Oak-Ridge National Laboratory, USA

Corresponding author:
Emmanuel Jeannot, Inria
Email: emmanuel.jeannot@inria.fr

*Throughout the paper we will use the term "*job*" or "*application*" interchangeably

†In this manuscript we call access to the storage, an Input/Output (I/O) phase where date are read or written to a remote parallel file system

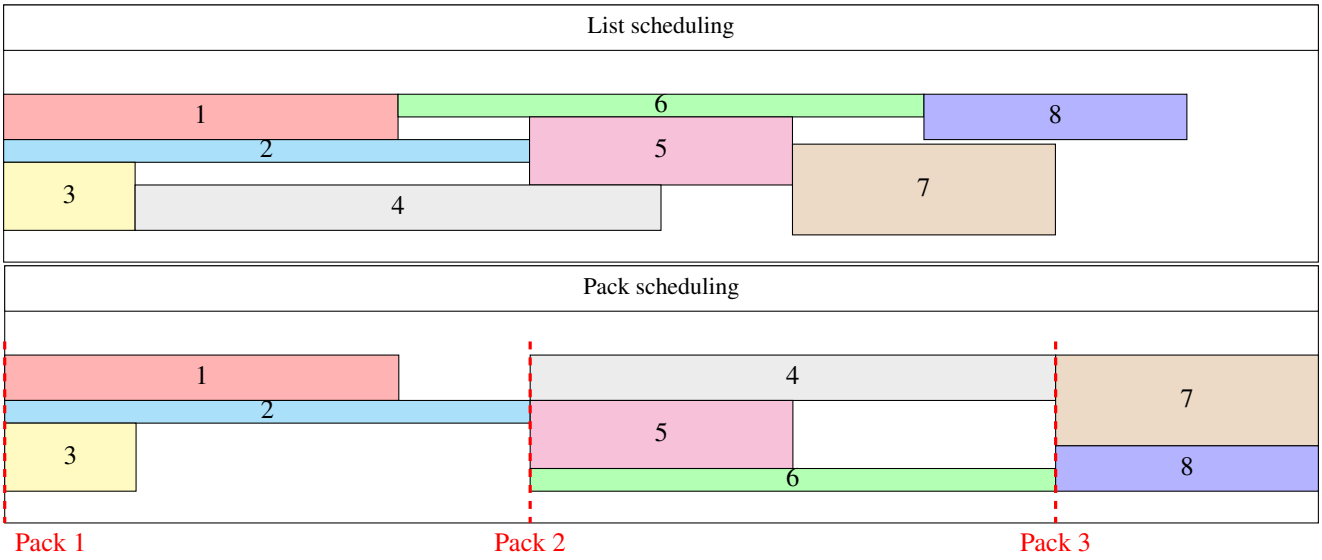

Figure 1. An Gantt-chart example of list scheduling and pack scheduling for the same set of tasks (packs are separated by dotted red lines). x-axis is time, y-axis is number of resources

its execution Sun et al. (2018); Carretero et al. (2020). *List-Scheduling* does not impose this constraint: it sorts the tasks given a priority order (typically, First-Come-First-Served as in several batch schedulers pri; Jackson et al. (2001)), and schedules them as soon as there are enough compute nodes available. In this work, we design, discuss and compare I/O-aware versions of these two paradigms. To the best of our knowledge, such study has never been conducted.

There are generally two objectives to account for: a platform-oriented objective (maximizing the utilization of the machine, i.e, the number of Flops), and a user-oriented objective (fairness). Hence, to implement a multi-dimensional scheduling (taking into account I/O needs as well as processor needs) and a multi-criteria scheduling (utilization vs. fairness), we consider a two-pronged approach:

- The batch-scheduler uses accumulated I/O information instead of exact behavior (which can be collected using tools such as Darshan Snyder et al. (2016)) to map the application onto the resources,
- At runtime, an I/O scheduling middleware (such as Clarisse Isaila et al. (2016)) then schedules the concurrent I/O using online heuristics.

In this paper, we make the following important contributions:

- we provide and compare an I/O-aware version of the pack and list-scheduling strategies;
- while list-scheduling is usually preferred as a mapping heuristic for batch-schedulers, we show that in case of I/O intensive workloads, an I/O-aware pack scheduling strategy may be more efficient;
- for Pack-scheduling, we show that a *characteristic time*, i.e. the order of magnitude of I/O transfers, can be taken into account to provide fairer policies. We provide intuition and discuss the implications of this.
- we underline the strong relation between the workloads and the scheduling policy relevance and provide insight on how to choose an adequate one.

Related work

Architecture and I/O optimization The I/O bottleneck is an issue that can cause a significant performance loss Gainaru et al. (2015) in High-Performance Computers. Several orthogonal solutions to deal with it have been explored. The first one is to add additional hardware (typically Burst-Buffers Liu et al. (2012); Kougkas et al. (2016), multi-layer memory architectures Boito et al. (2013)) to mitigate the congestion occurring at the I/O bandwidth level. The architectural enhancements are aiming to fluidify I/O requests. It is often accompanied by the development of data middleware, such as the design of intermediate aggregating layer which enables collective operations Tessier et al. (2017); Singh et al. (2007). An orthogonal question is that of optimizing the I/O usage of a given set of applications on a specific platform. This is a problem with numerous dimensions that are often hard to configure properly. Some researchers have advocated the use of Machine Learning strategies Mao et al. (2019) to do so. These previous approaches are often a means to optimize the I/O usage of a given set of applications on a specific platform.

In the presence of competing applications, other solutions try to use the elasticity of applications and resources. Recently Singh and Carretero Singh and Carretero (2019) proposed a middleware that would allow using the malleability of an application to shift forward or backward the time when an application is supposed to do its I/O movements. The solution AHPI/OS Isaila et al. (2008) allows using elastic partitions that can scale up or down the number of storage resources available to an application. This is similar to what has been recently done in cloud computing with the concept of elastic filesystem Xu et al. (2014).

I/O scheduling A complementary approach, has focused on the design of algorithms to deal with the I/O of several co-running applications. Generally, the idea is to choose which application to delay or, on the opposite, which to prioritize. Based on this, several approaches have been considered. Dorier et al. Dorier et al. (2014a) provided Calciom, a solution designed to let two applications

communicate together to decide the order at which they should perform their I/O. Strategies for more applications use a more centralized approach with a middleware (such as Clarisse Isaila et al. (2016)) in charge of taking the decision based on parameters of concurrent applications. Online strategies Gainaru et al. (2015); Zhou et al. (2015) typically present the impact of different priority orders on several conflicting objectives, from system utilization (i.e. maximizing the number of Flops) to user fairness (i.e. making sure that there are not some users who suffer from considerable delay). Gainaru et al. Gainaru et al. (2019) showed that one could observe a periodic I/O pattern in many HPC applications, and have proposed efficient but computationally intensive strategies to optimize the scheduling of I/O transfers. Aupy et al. Aupy et al. (2018) have proposed to model I/O behaviors statistically: at each time unit, the authors assumed that a job had a given probability to be performing I/O. Given a FIFO strategy to deal with I/O, they have shown how one can use Markov Chains to find the right dimensions for burst-buffers. The authors then extended their results to deal with the possibility of distributed buffers and solved it optimally via a linear program Aupy et al. (2019). However in this second work, their approach is very computationally intensive as it requires the entire execution profile of an application.

I/O impact in batch scheduling The last element which is the core of the problem studied in this work is the problem of scheduling the applications on the compute node, taking into account both the constraints linked to the mapping process, and adding a constraint on I/O. There is little work yet on this front. The two most notable are that of Herbein et al. Herbein et al. (2016) and recently in Carretero et al. Carretero et al. (2020). They are the motivation of our work. Herbein et al. Herbein et al. (2016) have worked on making batch scheduling I/O-aware. They have focused on the simple list-scheduling heuristic FCFS (First-Come First-Served) and have added an I/O constraint to guarantee that the I/O bandwidth is not overloaded by the concurrent applications. On the other hand, Carretero et al. Carretero et al. (2020) have used a pack-scheduling strategy to co-schedule applications, also respecting an I/O constraint. They have not considered backfilling strategies. Both works conclude that the main advantage of these approaches is the gain in job performance variability. In addition Carretero et al. Carretero et al. (2020) have shown that this gain allows taking better decisions in the presence of a distributed I/O system (several I/O nodes).

While without the I/O constraint, it is in general expected that list-scheduling strategies behave better than pack-scheduling ones, in our work, we provide comparison between these two approaches. We also discuss an important improvement to the pack-scheduling approach by considering the notion of *characteristic time*.

We will conclude this related work section by discussing on what is done to obtain information that can be used to instantiate the different algorithmic strategies or middleware. There are many tools that allow collecting the I/O profile of an application. One of the most famous is probably Darshan Snyder et al. (2016) which allows collecting at the end of the execution of an application cumulated information on I/O behavior. In addition, system administrators are often keen to monitor their platform behavior, providing valuable data on applications, I/O operations, and overall platform performance. For example, Carneiro et al. (2018); Hu et al. (2016) supply I/O measurement and a characterization of these applications in HPC platforms.

Approaches to estimate Job I/O Behavior

Checking whether there is enough I/O bandwidth available is a hard problem in itself to perform at the batch scheduler level. Indeed, generally I/O is burst-based: an application consists of consecutive phases of computation and phases of I/O. This creates obvious scalability issues (the number of events to take into account is too large). Even without scalability issues, predicting exactly the occurrence of these events is unreasonable (indeed, there is often a fluctuation in real behavior due to network congestion, processing unit usage, etc). In order to manage I/O constraints without knowing the exact I/O behavior of each application, different approaches have been proposed.

Existing approaches

The first approach as proposed by Aupy et al. Aupy et al. (2019) is to use all discrete event I/O information to develop the schedule. This is very computationally intensive as there are often a lot of very small I/O transfers (see for instance Paul et al. (2021), where $O(\text{hours})$ jobs include $O(10^7)$ I/O calls of size < 1M, and up to $O(100000)$ I/O calls of size 10M-100M). Less computationally intensive, an alternative approach as proposed by Gainaru et al. Aupy et al. (2017); Gainaru et al. (2019) is to use knowledge about the I/O pattern of applications to develop periodic strategies. Hence, one can focus on the subset of these I/O calls.

A second approach, as used by Aupy et al. Aupy et al. (2018) is to use statistical behavior for scheduling, and model I/O calls with probabilities. This is used for the mapping part, then an online I/O scheduling algorithm (FIFO in this case) is used.

Chosen approach: average I/O bandwidth

A third approach Herbein et al. (2016); Carretero et al. (2020) (the one used in this work), is to use the information on the *average I/O bandwidth of each application*. This is computed by summing all volumes of I/O transfer and dividing by the length of the application execution. This information is easily obtainable using system monitoring tools like Darshan Snyder et al. (2016) or at submission time, by the users, through an adequate batch scheduler interface. Herbein et al. (2016) have argued that the advent of local storage such as Burst-Buffers may help to generalize this behavior.

Dealing with average I/O bandwidth is very robust as it does not require to have a precise temporal view of the application behavior.

Then, the inclusion of this approach in the design of the job schedulers works as follows. Given a job to map,

1. we consider the jobs that are already mapped,
2. we check whether there are enough processors available,

3. we check whether the minimum average bandwidth available during its execution is greater than the average bandwidth needed by the job.

I/O-awareness of Mapping strategies

As explained in the introduction, I/O congestion causes performance degradation. If we let several applications access the storage at the same time, thus creating contention, they all suffer performance degradation. If we schedule I/O access by letting only one application accessing the storage at the same time, sequentially, then only the last application will see its performance degraded. This is why, in this work, to limit as much as possible performance degradation due to contention, applications I/O access are performed exclusively: no two applications are allowed to access the storage system at the same time. To implement this we can couple our approach with a runtime I/O management solution (such as Clarisse Isaila et al. (2016)) in order to provide or prevent access to the I/O system Gainaru et al. (2015); Carretero et al. (2020).

However, to avoid choosing which application is entitled to access the storage at a given time and hence possibly delaying other applications, it is crucial to carefully select the set of applications that are executed at the same time. To do so, we work on two aspects. First, as explained in the previous section, we consider, as the I/O need of a given application, the average I/O bandwidth of this application and not the exact I/O performed at any given time. Second, we have extended two heuristics of the literature (list-scheduling and Pack scheduling) to make them I/O-aware.

I/O-aware List-scheduling (LS)

When compute nodes are available, we try to map applications onto these resources. For the list-scheduling strategy, candidate applications are considered in their order of appearance in the workload on a first-come, first-served basis. They are mapped onto the available resources as soon as both their processor and their bandwidth requirements are met. This means that, as in regular job scheduler we need to have enough computing resources but, in addition, we also need that the I/O requirement of this application (i.e. its average bandwidth), does not exceed the remaining bandwidth (i.e. the storage bandwidth minus the sum of the average bandwidth of already running applications). We call this constraint a *strict constraint*: the sum of the average I/O bandwidth of all applications running at a time cannot exceed the I/O bandwidth of the system.

I/O-aware Pack-Scheduling

Carretero et al. Carretero et al. (2020) showed that making packs of applications starting at the same time and running concurrently can improve the control over congestion. Here, we follow the same procedure. In addition to the standard computing resource constraint, packs are built such that, its average I/O bandwidth does not exceed the I/O bandwidth of the system. This I/O constraint is called *average constraint*: the average I/O bandwidth of a pack is the sum of all I/O operations performed by the pack's application divided by the pack duration. It means that I/O operations can exceed the I/O limit at some points of the execution. To sum up, to add an application in a pack, we check (i) if there are enough available compute nodes; (ii) if the average pack I/O bandwidth with this additional job is not greater than the total bandwidth. If there are no solutions, we create a new pack to be scheduled after the existing packs.

Variants for building packs

The way we sort applications to built pack can have an impact on the pack-scheduling performance. In the following, we propose three different ways to build packs.

1. Without sorting applications (Random), this serves as a baseline. It can also describe a way to build packs in steady-state without an overview of the application pool.
2. Sorting applications following non-increasing execution time (Max), this is the default way to make packs, as scheduling applications with similar duration minimizes the resource unbalance at the pack execution end.
3. Characteristic time (Char) is the duration of a period consisting of one compute phase followed by an I/O phase. Sorting applications by this parameter is a less intuitive approach. As I/O accesses are performed exclusively, there is a delay induced every time an I/O request is blocked by another operation. Based on the periodic behavior of applications, we expect the I/O phases to be delayed in the first iterations. Then, if the period spans are close enough, a *synchronization* effect may occur and I/O can be performed in turn with little to no delays. A simple example is shown in Fig 2.

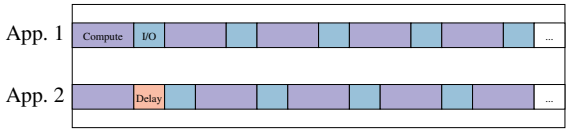

Figure 2. Synchronization example with two identical applications. I/O accesses are performed in turn and no delay occurs past the first iteration.

Backfilling

In pack scheduling, the platform nodes get idle during the execution of each pack as applications terminate. These nodes are not reused until the next pack starts. In a minor extent, idle resources waiting for a future application can occur as well in the list-scheduling case. However this performance loss can be avoided. Indeed, the packs and applications duration can be predicted as well as the amount of nodes involved. Using this information, we can see if an application fits in the free space and modify the schedule accordingly. In the following, such a backfilling strategy is implemented for both pack and list scheduling.

Machine and Application Models

Tampering directly with a production system, impacting the work of users solely for the sake of evaluating prototype strategies is not an option. Evaluating the relevance of the

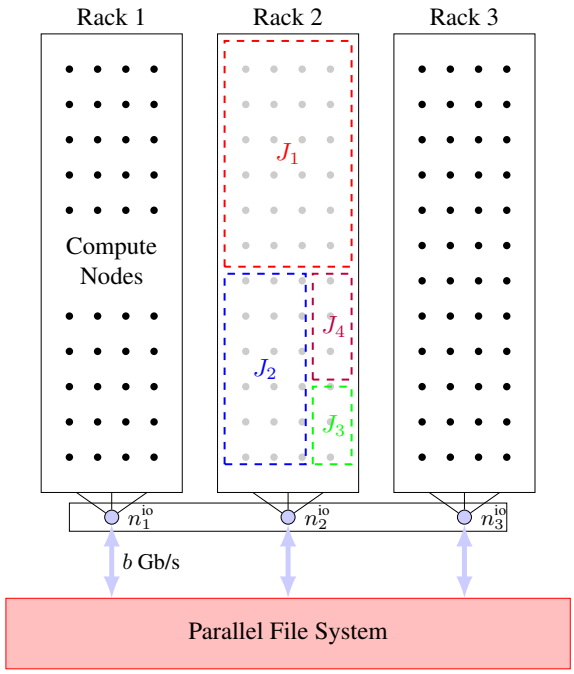

Figure 3. Schematic of the architecture. Jobs J_1, J_2, J_3 and J_4 compete for the bandwidth available on I/O node n_2^{io} Carretero et al. (2020).

aforementioned strategies and their consequences is thus particularly challenging and hence requires, as a first step, the use of heavy simulations to compute statistically significant evaluations.

In previous work, the coherence between the behavior of the simulator that we describe below and that of real machines (Vesta, a development platform for Mira Gainaru et al. (2015), and Jupiter a Cluster at Mellanox Gainaru et al. (2019)) was verified with I/OR benchmarks. Hence for this evaluation we will rely solely on a simulator which we detail below.

Machine Model

In high performance computing, parallel platforms consist of computational resources structured in racks composed of compute nodes. One (or sometimes several) I/O node is available on each rack for the compute nodes to access the parallel file system (PFS). Each I/O node has a fixed, limited *I/O bandwidth* and hence compute nodes share this bandwidth when accessing data on the PFS. In many supercomputers racks are homogeneous: on each rack, the I/O bandwidth is the same as well as the number of compute nodes associated to each I/O node.

Therefore, without loss of generality, in this paper, we will consider only one rack with one I/O node: extending it to several racks (or I/O nodes) Carretero et al. (2020) is straightforward: the proposed solutions will be applicable to an arbitrary number of racks considering the platform homogeneity.

Moreover, for the evaluations, following the trend in current I/O management software (such as Clarisse Isaila et al. (2016)), we consider that simultaneous bandwidth sharing is not allowed (i.e. on a given I/O node, only one application is performing I/O at the same time). Blocking I/O guarantees that at all time the I/O bandwidth is not overloaded, hence we do not need here to model I/O congestion. We discuss I/O blocking and its consequences in the future work section. Additionally, we assume that I/O preemption is not allowed either: once an application has started to perform I/O, it has to finish its transfer. A representation of this architecture model is shown in Figure 3.

Job Model

Following the literature, we consider that running jobs perform a series of consecutive *non-overlapping* phases:

- compute phases (executed on the compute nodes);
- I/O phases (a transfer of a certain volume of I/O using the available I/O bandwidth) which can be either reads or writes.

In order to run properly, jobs must have access to sufficient computational and I/O resources. The amount of processors requested is defined beforehand by the user and can be seen as one of the application parameters. The amount of data to be accessed during the I/O phase is also a parameter that is assumed to be deterministic. We assume that the user is able to give such information.

Formally, we have a set of n jobs $\{J_1, \ldots, J_n\}$. Each job J_i requests Q_i compute nodes (among P available) for its execution. J_i consists of n_i successive, blocking and non-overlapping operations:

- $W_{i,j}$ (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:

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

Workload Generation

A workload is a set of applications. For each application we define its computational parameters (number of nodes and duration) as well as its I/O behavior (amount of I/O operations, number of periods, etc.) as discussed above.

For the evaluation, we do not consider release time for the application but consider them to be released as a single batch. This will have implications in how we evaluate the performance of the algorithms which we discuss in the "relevance of static workloads" section.

In order to perform a significant number of experiments, we designed a generator aiming to produce diverse fitting workloads[‡].

Application sets are generated following two different protocols detailed below: (i) the Mira-based protocol, inspired by data Patel et al. (2020) collected on the eponymous supercomputer; or (ii) a uniform protocol in

[‡]All scripts regarding the generation of applications are available here: https://gitlab.inria.fr/nividal/workload_mapping

which all parameters are independent. For both these protocols, we will use a protocol to generate I/O behavior described next.

But, before describing the workload generation protocol, we need to define an important workload characterization that will be used for the experiments: the I/O-load (or I/O intensity).

Workload characterization

We hypothesize that the impact of I/O restrictions policies depends on the amount of I/O operations performed by the workload.

Therefore, in our study, we tested the impact of I/O scheduling policies as a function of the I/O load of the system. Given a workload, we define the *I/O load* (or *I/O intensity*), α, as an upper-bound on the I/O bandwidth exertion. α is obtained by dividing the total I/O duration (I/O volumes over available bandwith) by a lower bound of the workload execution time (sum of the processor times of all applications over the total number of processors).

$$\alpha = \frac{\sum_{i=1}^{n} \sum_{j \leq n_i} v_{i,j}/b}{\sum_{i=1}^{n} \left(\frac{Q_i}{P} \cdot T_i(b) \right)} \tag{2}$$

I/O behavior generation

Many real-life applications have pseudo-periodic behavior Carns et al. (2009); Dorier et al. (2014b). Since the point of interest is to study I/O conflicts on large workloads, the fluctuation in operation times does not seem particularly relevant. Hence, to simplify the model, each application is periodic (i.e. for all $j \leq n_i$, $v_{i,j} = v_i$ and $w_{i,j} = w_i$) in all generated workloads.

The number of periods are chosen uniformly at random between 10 and 100 iterations, following orders of magnitude from the literature Liu et al. (2012). This number is independent of the length of the application or the total volume of I/O performed by the application. Hence, when there are many iterations for a given execution length, the total volume of I/O gets evenly distributed over the execution as the iteration length decreases.

Data from the literature generally presents cumulative I/O over the life of the applications. It is not straightforward to deduct the amount of time spent performing I/O for each iteration of an application. However, we consider that the amount of time spent for computation is positively correlated by the time spent to do I/O: the more work, the larger the amount of generated data. Here we consider two I/O workload types (both are parametrized by the target I/O load parameter α):

- BN (standing for Bi-Normal): in this protocol we consider two I/O behaviors, applications with few I/O requirements, and applications with large I/O requirements. For each application, we draw the time spent doing I/O in each iteration following two truncated normal distributions (truncated on the interval $[0, 1]$). These normal distributions have a mean of 0.1 (10% of the time of an iteration is spent doing I/O) and 0.9 and a variance of 0.1. Note that because of the truncation, this gives for the resulting truncated normal distributions X_1 and X_2 means $\mu_1 = 0.289$ and $\mu_2 = 0.711$.
  The prevalence of each type of application depends on the I/O load α that we want to obtain for the evaluation.
- No (standing for Normal): in this protocol, for all applications we draw the proportion of time doing I/O according to a single truncated normal distribution of mean $\mu = \alpha$ (the I/O load) and variance $\sigma = 0.1$.

The total I/O volume of each application is then computed using this proportion of time, scaled proportionally by the number of processors of the application using the following formula where $\mathcal{N}$ designates a random variable following one of the aforementioned truncated normal distribution:

$$v_i \leftarrow \mathcal{N}(\mu, 0.1) \cdot w_i \times Q_i/P$$

Mira-based generation protocol

Here, we described a protocol to generate a workload similar to what have been observed in the Mira (a former large supercomuter from Argonne National Lab) traces.

Processor repartition Data found from Mira shows that the required processors per job follow a discrete exponential distribution of parameter $\lambda = 1.35 \cdot 10^{-4}$ with values ranged from 512 to 49152. We use this distribution to generate jobs for this protocol.

Job duration The same source shows that the job duration depends on the amount of processor used.

- For less than 4K nodes, we observe a median job time of 1 hour
- Between 4K and 16K nodes, we observe a median job time 2 hours,
- More than 16K, we observe a median job time of 0.5 hour.

In each case, we use normal distribution with the same median and a variance of 10 percent of the median time. These distributions are obviously truncated to their positive values.

Uniform generation protocol

We study a second protocol where both job duration and processor repartition are generated uniformly at random.

Job duration The duration of compute operations w_i is chosen uniformly at random in the interval [10,100] minutes.

Processor repartition Let $\mathbb{E}(X)$ be the expectation of the proportion of time spent doing I/O as generated by the I/O behavior generation. To instantiate the number of processors Q_i of each application, we use a uniform distribution in the discrete set $\{2^j\}_{j=0\ldots11}$ of mean $\bar{Q}$. This is obtained by replacing in Equation (2) all values by their average value (where α_{gen} is the target I/O load as specified by the workload generator) :

$$\bar{Q} = \frac{P\mathbb{E}(X)}{\alpha_{\text{gen}}(1 + \mathbb{E}(X))}.$$

Relevance of static workloads

For the sake of simplicity, we chose to use static workloads where the amount of work to perform is defined beforehand. However, in real life, applications continue to arrive all the time. Both our model and such dynamic workloads are comparable when in steady-state but ours present a startup and a closure time that are not realistic.

Nonetheless, the core of mapping a scheduling application is mostly relevant when the system is under heavy load, typically during weekdays. A system administrator could choose to use times with low utilization, e.g. nights and week-end, to wait for the completion of all applications submitted, before accepting new one in order to avoid starvation. Alternatively, we can imagine an overlap of several schedule when the platform is not stressed i.e. starting a new schedule with the available resources as soon as the previous one exits steady-state. In our experiments, we have to confine ourselves to these possible scenarios.

Four Types of Workload

To sum up, in the experiments we will deal with four types of workloads. Given the I/O generation (BiNormal or Normal), and job generation (Mira or Uniform), we name four families of workloads: *Mira-BiNormal* (M-BN), *Mira-Normal* (M-No), *Uniform-BiNormal* (U-BN) and *Uniform-Normal uniform* (U-No).

Evaluation criterion

In this work, we seek to design mapping strategies to optimize the usage of compute resources and I/O bandwidth. Hence, we need to solve a scheduling problem composed of two sub-problems at the same time.

First, we have to compute a *job schedule*, solution of the *mapping* problem which consists in choosing for each application the allocation of compute nodes during the execution timeframe of the batch.

Second, we have to compute an *I/O schedule*, which consists in deciding which application acquire the usage of the I/O node to access the PFS.

The relevance of such a schedule can be measured either from the platform administrator point of view or from the user point of view. It is important to notice that these two approaches, although not exactly opposites, can be in conflict.

Therefore, given a schedule, we use two different metrics to evaluate the relevance of our strategies according to these two angles.

1. The platform *utilization* of a schedule at any time is given as the ratio between the total work executed and the time. Typically, the utilization is an objective more *platform* oriented. The evolution of the platform utilization during the execution shows in what extend the resources are used.
2. The *stretch* ρ_i of an application J_i is the ratio between its minimal execution time and its actual execution time. 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 a *user* oriented objective.

To sum up, our general optimization problem is the following: given a batch of jobs running on a platform with an available I/O bandwidth to the PFS, and connected to P compute nodes. Find a schedule that either maximizes the total utilization, or that minimizes the maximum stretch, respecting the resource constraint (available number of nodes and no I/O bandwidth sharing). This reduces to finding the right allocation of I/O for the different applications. We call this problem IO-SCHED.

Evaluation and Results

In this section, we present experimental evaluations of mappings using the I/O-aware pack or list-scheduling (LS) strategies. In each experiment, we compare results with the different pack algorithms introduced in section "Variants for building packs" (Char, Max and Random). We start with an analysis of the algorithm's performance on the different workloads. Then, we discuss these data in the light of resource usage throughout the execution.

We recall that we evaluate 4 strategies (see Sec. for more details):

- list-scheduling, on first-come first serve basis (LS);
- and three pack scheduling variants:
  - Without sorting applications (Random);
  - Sorting applications following non-increasing execution time (Max);
  - Characteristic time, the duration of a period consisting of one compute phase followed by an IO phase (Char);

Algorithm performance

Noise filtering The random factor, intrinsic to the generation leads to some distorted workloads, with pathologically high I/O. For example, in the binomial workload case, the I/O and the processor are not independently chosen in order to fit the target exertion. Consecutive unbalance in the first application defined can end up designing an abnormally loaded workload. To avoid these artifacts, we exclude such data from the generation.

Utilization Utilization measurement throughout the execution for each workload profiles are presented in Fig 4 and 5. The experiments have been performed for different I/O load values ranging from 0.2 to 0.8. Each trace presents the average of ten executions. Studies on a single run have been in made beforehand to ensure representativeness.

For all compute-intensive workloads (low I/O load), list scheduling makes a better use of the platform throughout the execution. However, list scheduling utilization is degrading when the I/O load increases. Especially when dealing with uniform workloads. In all cases, the platform utilization of list scheduling is stable throughout the execution. The small under-utilization at the beginning of some cases is explained by the few critical applications having huge requirement in terms of I/O or processors.

As for pack-based schedule, we can observe in most cases a startup time with high machine utilization at the beginning

and a quick deterioration while the workload is being exhausted. During the steady-state, pack scheduling based on application length can achieve a comparable utilization as list scheduling, and even outperform it when the I/O load is large.

It means that for a dynamic workload which ensures constant application disposability for the scheduler, Pack scheduling may achieve a better platform utilization. However as it depends on the application sorting prior to pack building, this better utilization may also lead to starvation for low-priority applications.

Lesson learned 1

For utilization, List-scheduling exhibits a constant performance throughout the execution no matter the remaining available application. It can be outperformed by Pack-scheduling during the steady-state for high I/O loads (over 0.6).

Stretch Measurement on stretch in function of the I/O load are presented in Fig 6 for the maximum stretch and 7. for the average stretch. Figures are divided depending on the underlying workload. In each trace, every point plots the result of one run on a specific workload. Lines show the average stretch with confidence intervals.

In all scenarios, pack scheduling provide a slight improvement on both the maximum and the mean stretch compared to list scheduling.

Surprisingly packs built without sorting the applications have a lower average stretch on Mira based workloads. Indeed, the correlation between parameters leads to contention when sorting applications based on the execution times.

Pack scheduling based on characteristic time was designed in order to synchronize applications and minimize the stretch. Indeed, it provides a significant improvement for uniform workloads and perform in the same way as other for the M-No scenario.

The same heuristic performs worse than any other algorithm for M-BN (Mira profile and bi-distributed I/O). Bi-distribution creates applications with the same characteristic time that are very different in their I/O behavior and the synchronization is made at loss. Indeed, when it happens, synchronism induces a constant delay for all concurrent applications. This may ensure a better bound for the worse case scenario by dividing the loss equally. However, when this delay is too long, it impacts all applications execution.

In the end, Pack scheduling based on characteristic time is a double-edged sword: it allows synchronization of the I/O phases. Meaning that if pack scheduling is already efficient for the given workload, it enables applications to "take turns" on the bandwidth and has (almost) no delay. However, if synchronization cannot occur in the given workload, it will lead to recurring delay and an increased stretch.

Lesson learned 2

For Stretch, pack-scheduling performs better than list-scheduling. The only case where the pack-scheduling is outperformed by list scheduling is, for characteristic time, when synchronization between application cannot occur in the given workload

One pitfall would be to evaluate pack scheduling with uniform workloads before implementing them. It could result in an overestimation of their performance. This case also emphasizes the setup sensibility and the difficulty to design solutions taking into account both the general and the specific case.

Starvation for pack scheduling

In Fig 8, we present three measurements: the average number of applications running, the average amount of processors used and the average portion of used bandwidth throughout the execution of each heuristic. The top three are obtained when scheduling applications from Mira Binormal workloads and the bottom three for Mira Normal Workloads. Comparing the two, we want to explain why Pack Scheduling is less efficient in the Mira Binormal case. For pack scheduling, We see that some application profiles are favored compared to others:

1. The applications that use more processors are scheduled first (the processor usage decreases with time).
2. I/O intensive applications are performed last (the bandwidth utilization increases with time).

However, we do not see such behavior for list-scheduling: processor or bandwidth usage stay roughly constant with time (we see a degradation at the end of the execution due to the fact there is not enough applications to execute).

This means that, for pack scheduling, I/O intensive applications need to wait that compute intensive applications have been executed (while this is not the case for list scheduling). We can therefore face starvation in the case of pack-scheduling.

Another way to see the problem is that list-scheduling exhibits a steady state in terms of resource usage (no type of application is favored) while this is not the case for any of the pack scheduling variants (there is no steady state).

Lesson learned 3

Heuristics without steady-state are symptomatic of starvation (e.g. executions have several phases with I/O intensive applications performed after compute intensive ones). Hence, when choosing a heuristic, it is crucial to check whether it reaches a steady-state or not. For instance, in Fig 8 no pack scheduling heuristic has a steady-state for M-BN workloads hence list scheduling should be favored.

Idle time vs stretch

As explained in the description of the I/O aware strategies, our strategies feature two different kinds of constraints. List

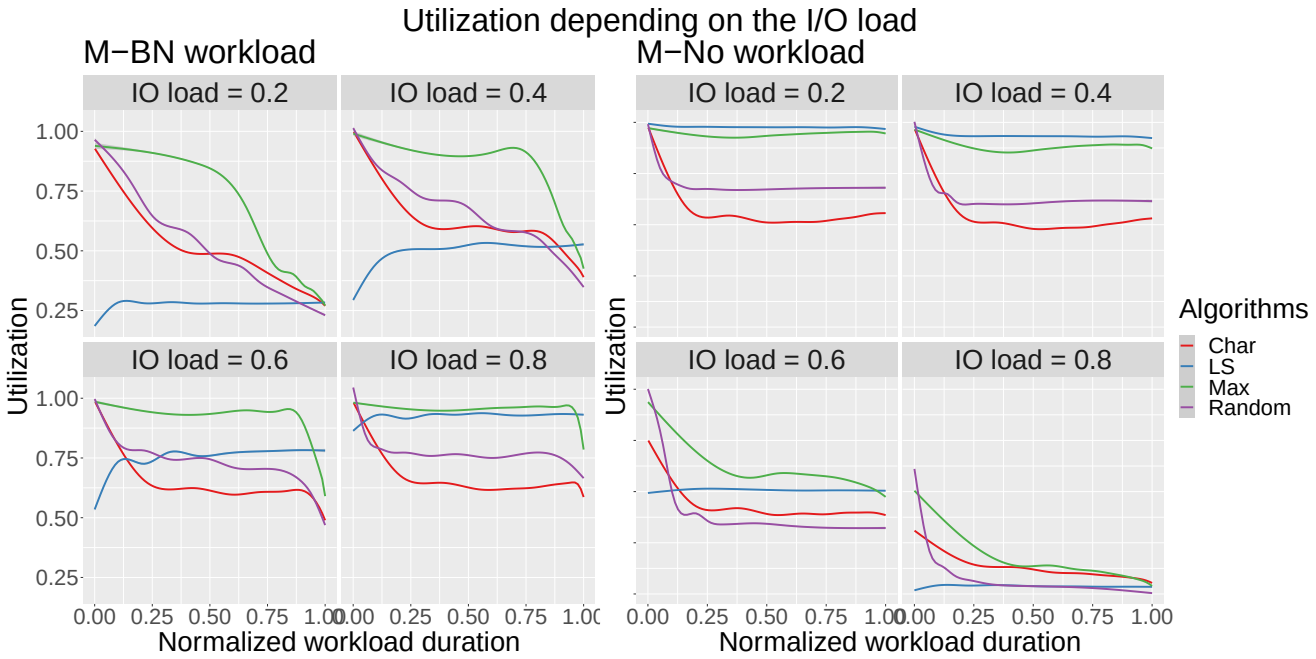

Figure 4. Utilization during the execution for different values of I/O load, for Mira-based workload with Normal or Binormal I/O profiles

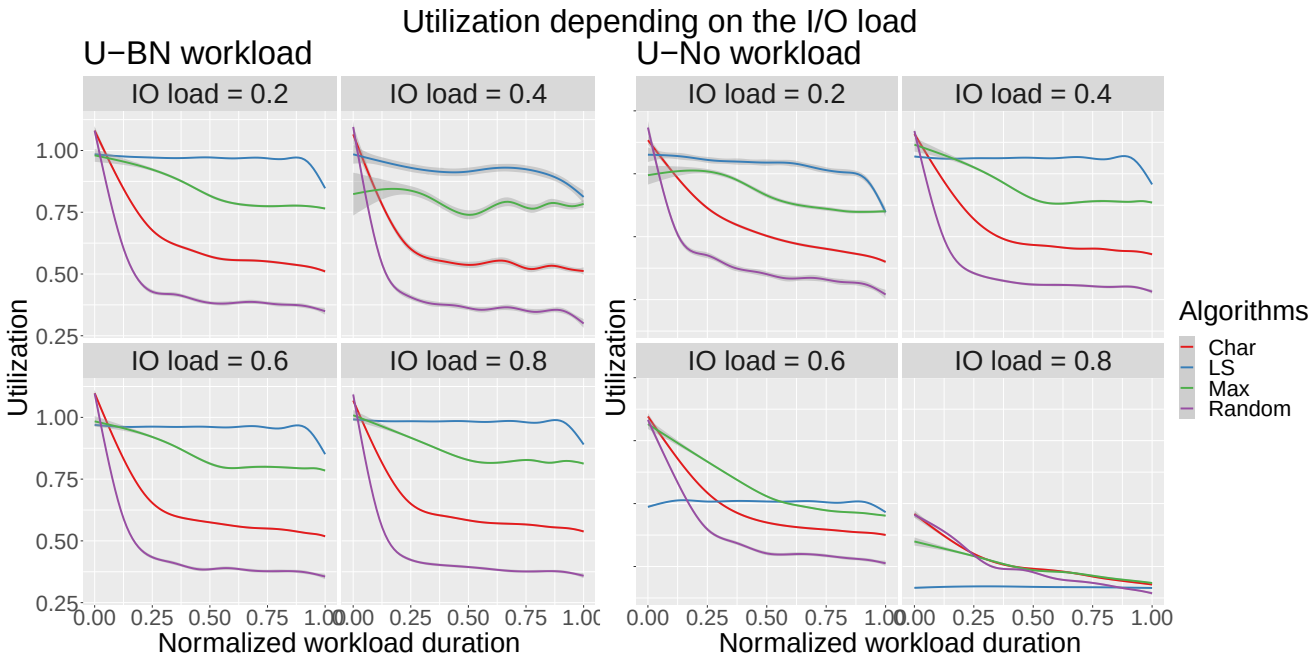

Figure 5. Utilization during the execution for different values of I/O load for Uniform workload with Normal or Binormal I/O profiles

scheduling implement the *strict constraints* which implies that at no time the sum of the bandwidth used by applications can exceed the total system bandwidth. Should an application exceed such limit, it will be paused until enough bandwidth is available. On the contrary pack scheduling (in all variants) features an *average constraints*: packs are built such that the average bandwidth usage does not exceed the total system bandwidth.

In Fig 9, for the four studied strategies, we show the mean stretch as a function of the idle ratio i.e. the cumulated processor-time of non-allocated resources over the total processor-time of the execution. The average of all experiments is drawn with a confidence interval in addition to a point for the result of each execution. Additionally color gradient shows the final average occupation achieved during the workload execution. This figure was obtained while running with M-BN workloads. The same results are obtained with M-No where trends are harder to read in uniform cases.

As expected, we see that the stretch is increasing with the idle ratio. However, the shape of the curve highly depends on the nature of the scheduling heuristic. We see that, except for the characteristic time variant, pack and list scheduling have a comparable Idle/stretch ratio with a slight offset in diminishing idle ratios and increasing stretch for list-scheduling. This offset starts for efficient workload (less than

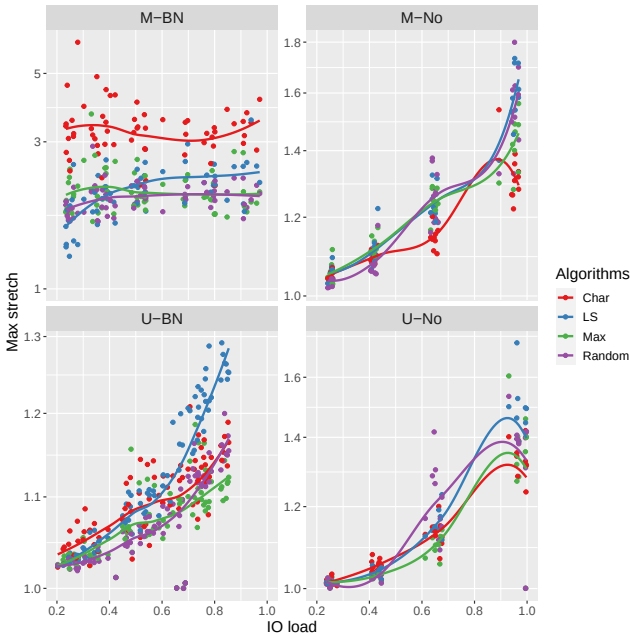

Figure 6. Max stretch for different workloads

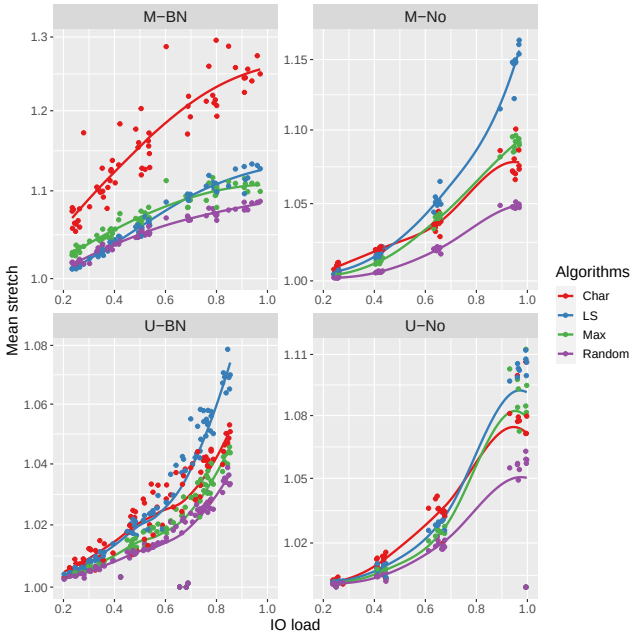

Figure 7. Mean stretch for different workloads

10% idle time, and almost no stretch) and stays constant ever since. Packs built after sorting the application by duration have exactly the same behavior as randomly constructed packs with only a constant gain in regard to stretch. It is more complex to interpret the behavior of packs based on characteristic time. It appears to introduce more idle time and less stretch than other pack building policy in order to gain on the contention. However, this only works for workloads that are already efficient. When the loss increases, the stretch skyrockets. It is likely due to the fact that applications synchronize their I/O, therefore setting a constant delay for all applications running together. For compute intensive workloads and for sets of comparable applications, this delay is close to zero hence the gain but when workloads become data intensive all applications are penalized.

Lesson learned 4

Lesson learned: When I/O increases and conflicts occur, both the idle time and the contention increase. Packs based on characteristic time are advantageous under heavy I/O loads and must be chosen only after rigorous study. As for other heuristics, they keep the same trade-off pace. Then, choosing one over another depends only on the initial offset preference; in addition to the performance discussed before.

Conclusion

With the increasing importance of I/O in HPC, it is required to take into account the I/O behavior of applications when submitted to the batch scheduling system. In this paper, we presented and studied strategies for data-aware mapping. They were chosen for their simplicity and the clear difference on the constraints they implement. The analysis of list scheduling allows studying strict constraints whereas Pack scheduling represents average restrictions. We defined evaluation metrics and design different workloads in order to provide a deep understanding of the strategies behavior. Results (summarized in table 1) show that Pack scheduling can provide a better machine utilization throughout the execution when the I/O exertion is high, in other cases list scheduling is preferable. The parameters defining the workload are also capital with a trade-off between optimized stretch and better platform utilization. For this purpose, we implemented pack based on characteristic time and showed that, if the I/O exertion is not excessive, it can achieve better fairness than both list scheduling and other pack variants in the same scenarios. However, we have seen that performance depends strongly on the workload, pack-scheduling algorithms, in particular the one using characteristic time, perform better with comparable applications (uniform generation, normal repartition of I/O).

The natural direction to open-up as continuation is to discuss over ways to classify applications in order to be more efficient in pack building and have a finer workload appreciation. In a more elaborate way and more importantly, we plan to allow parallel I/O operations instead of exclusive ones. In such a set-up, we would be able to study not only one mapping strategies but combination of several sharing an adequately dedicated portion of the bandwidth.

Collecting I/O information is hard. Analysis tools such as Darshan measure cumulative data which is an important information. With this work we have shown that one additional information may be critical in the design of I/O-aware resource manager: characteristic time. Future work should focus on defining it properly on general sets of applications.

The source code for our implementations is available at https://gitlab.inria.fr/nividal/workload_mapping.

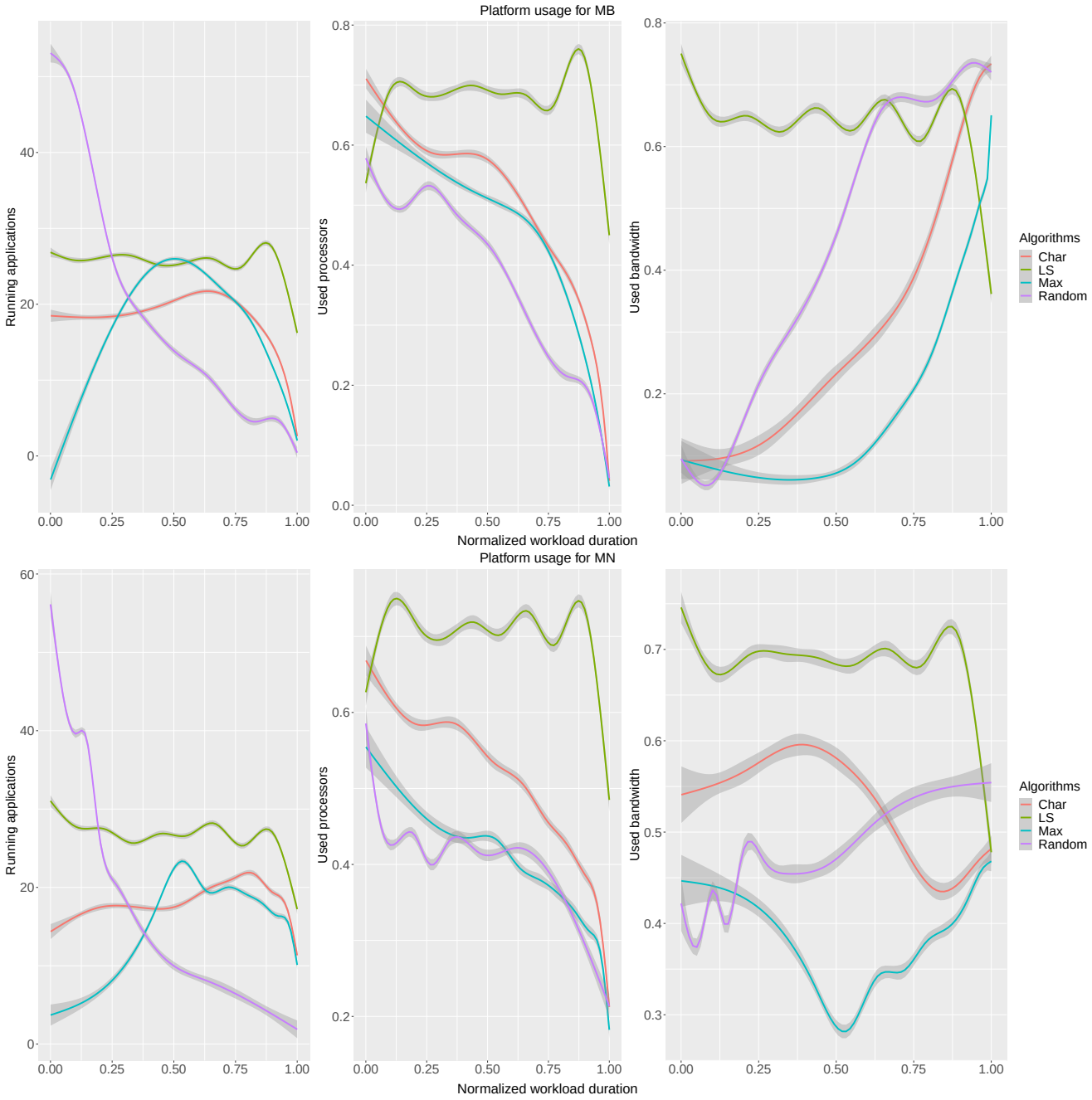

Figure 8. Resource usage (average number of applications running, used processors, used bandwidth on M-BN and M-No workloads

	Execution behavior	Usual advantages	Impact of I/O load	Depending on the workload
List Scheduling	Constant performance	Better use of the platform	Better for compute intensive workloads	Degraded performance high I/O uniform workloads
Pack Scheduling	High performance at the start, exhaustion	Better stretch	Better for I/O intensive workloads	Risk of starvation when packing diverse applications

Table 1. Summary of the findings

Acknowledgements

This work was supported in part by the French National Research Agency (ANR) in the frame of DASH (ANR-17-CE25- 0004) and in part by the Project Région Nouvelle Aquitaine 2018-1R50119

Biography

Emmanuel Jeannot Emmanuel Jeannot is a senior research scientist at Inria. He got his PhD degree in computer science from the Ecole Normale Suprieure de Lyon (France) in 1999. From 2000 to 2005, he was assistant professor at the University Henry Poincaré in

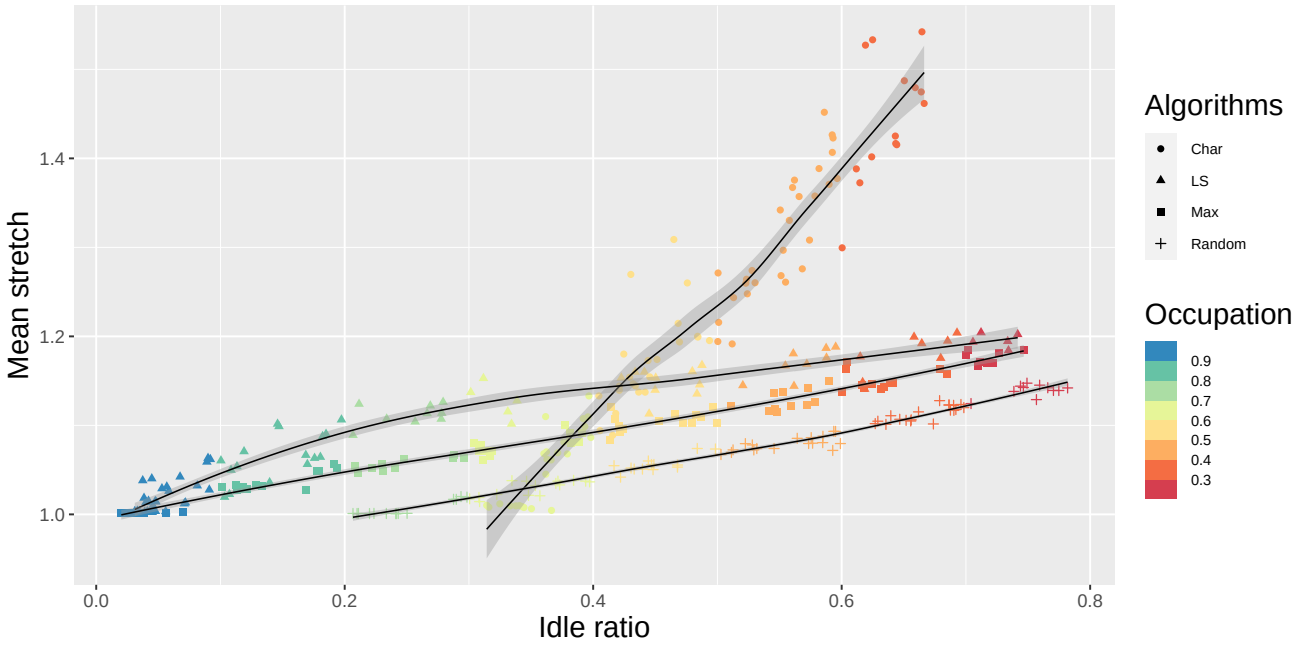

Figure 9. Mean stretch as a function of idle time

Nancy. From 2005 to 2009, he worked for the Nancy Grand-Est Inria research center. Additionally, in 2006 he was a visiting researcher at the University of Tennessee, ICL laboratory. Since 2009, Emmanuel Jeannot is conducting his research at INRIA Bordeaux Sud-Ouest (where he is leading the TADaaM team) and at the LaBRI laboratory of the University of Bordeaux. His main research interests span the vast domain of parallel and high-performance computing and more precisely: runtime systems, processes placement, topology-aware algorithms, scheduling for heterogeneous environments, data redistribution, I/O and storage, algorithms and models for parallel machines, adaptive online compression and programming models.

Guillaume Pallez Guillaume Pallez is a tenured researcher at Inria Bordeaux – Sud-Ouest. His research interests include algorithm design and scheduling techniques for parallel and distributed platforms (data-aware scheduling, stochastic scheduling etc). Among other, he served as the Technical Program vice-chair for SC'17, Technical program chair for SC'24, and co-general chair for ICPP'22. He was a recipient of the 2019 IEEE TCHPC Early Career researcher award. See http://people.bordeaux.inria.fr/gaupy/ for further information.

Nicolas Vidal After graduating in fundamental computer science in 2018, Nicolas Vidal spent four years in the TADaaM team (Topology-Aware System-Scale Data Management for High-Performance Computing applications) in Bordeaux for his PhD. His thesis, defended January 2022, consists in defining strategies from a theoretical perspective to mitigate performance loss in the I/O bottleneck. He is now a working on performance models and data compression as a postdoc for the Workflow Systems group at Oak Ridge National Lab.

References

(????) Slurm Multifactor Priority Plugin. https://slurm.schedmd.com/priority_multifactor.html. Accessed: 2020-10-06.

Aupy G, Beaumont O and Eyraud-Dubois L (2018) What size should your buffers to disks be? In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 660–669.

Aupy G, Beaumont O and Eyraud-Dubois L (2019) Sizing and partitioning strategies for burst-buffers to reduce io contention. In: *Parallel and Distributed Processing Symposium (IPDPS), 2019 IEEE International*. IEEE, pp. 631–640.

Aupy G, Gainaru A and Le Fèvre V (2017) Periodic i/o scheduling for super-computers. In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, pp. 44–66.

Boito FZ, Kassick RV, Navaux PO and Denneulin Y (2013) Agios: Application-guided i/o scheduling for parallel file systems. In: *2013 International Conference on Parallel and Distributed Systems*. IEEE, pp. 43–50.

Carneiro AR, Bez JL, Boito FZ, Fagundes BA, Osthoff C and Navaux PO (2018) Collective i/o performance on the santos dumont supercomputer. In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, pp. 45–52.

Carns P, Latham R, Ross R, Iskra K, Lang S and Riley K (2009) 24/7 characterization of petascale i/o workloads. In: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, pp. 1–10.

Carretero J, Jeannot E, Pallez G, Singh D and Vidal N (2020) Mapping and scheduling hpc applications for optimizing i/o. In: *ICS2020-34th ACM International Conference on Supercomputing*. pp. 1–12.

Dorier M, Antoniu G, Ross R, Kimpe D and Ibrahim S (2014a) Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In: *Parallel and Distributed*

Processing Symposium, 2014 IEEE 28th International. IEEE, pp. 155–164.

Dorier M, Ibrahim S, Antoniu G and Ross R (2014b) Omnisc'io: a grammar-based approach to spatial and temporal i/o patterns prediction. In: *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, pp. 623–634.

Gainaru A, Aupy G, Benoit A, Cappello F, Robert Y and Snir M (2015) Scheduling the i/o of hpc applications under congestion. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, pp. 1013–1022.

Gainaru A, Le Fèvre V and Pallez G (2019) I/o scheduling strategy for periodic applications. *ACM Transactions on Parallel Computing* .

Herbein S, Ahn DH, Lipari D, Scogland TR, Stearman M, Grondona M, Garlick J, Springmeyer B and Taufer M (2016) 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*. pp. 69–80.

Hu W, Liu Gm, Li Q, Jiang Yh and Cai Gl (2016) Storage wall for exascale supercomputing. *Frontiers of Information Technology & Electronic Engineering* 17(11): 1154–1175.

Isaila F, Blas JG, Carretero J, Liao Wk and Choudhary A (2008) Ahpios: An mpi-based ad hoc parallel i/o system. In: *2008 14th IEEE International Conference on Parallel and Distributed Systems*. IEEE, pp. 253–260.

Isaila F, Carretero J and Ross R (2016) Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms. In: *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, pp. 346–355.

Jackson D, Snell Q and Clement M (2001) Core algorithms of the maui scheduler. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, pp. 87–102.

Kougkas A, Dorier M, Latham R, Ross R and Sun XH (2016) Leveraging burst buffer coordination to prevent i/o interference. In: *e-Science (e-Science), 2016 IEEE 12th International Conference on*. IEEE, pp. 371–380.

Liu N, Cope J, Carns P, Carothers C, Ross R, Grider G, Crume A and Maltzahn C (2012) On the role of burst buffers in leadership-class storage systems. In: *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, pp. 1–11.

Mao H, Schwarzkopf M, Venkatakrishnan SB, Meng Z and Alizadeh M (2019) Learning scheduling algorithms for data processing clusters. In: *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. ACM, pp. 270–288.

Patel T, Liu Z, Kettimuthu R, Rich P, Allcock W and Tiwari D (2020) Job characteristics on large-scale systems: long-term analysis, quantification, and implications. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 1–17.

Paul AK, Karimi AM and Wang F (2021) Characterizing machine learning i/o workloads on leadership scale hpc systems. In: *MASCOTS 2021*. pp. 1–8.

Singh DE and Carretero J (2019) Combining malleability and i/o control mechanisms to enhance the execution of multiple applications. *Journal of Systems and Software* 148: 21 – 36. DOI:https://doi.org/10.1016/j.jss.2018.11.006. URL http://www.sciencedirect.com/science/article/pii/S0164121218302425.

Singh DE, Isaila F, Calderón A, Garcia F and Carretero J (2007) Multiple-phase collective i/o technique for improving data access locality. In: *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)*. IEEE, pp. 534–542.

Snyder S, Carns P, Harms K, Ross R, Lockwood GK and Wright NJ (2016) Modular hpc i/o characterization with darshan. In: *2016 5th workshop on extreme-scale programming tools (ESPT)*. IEEE, pp. 9–17.

Sun H, Elghazi R, Gainaru A, Aupy G and Raghavan P (2018) Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. pp. 194–203. DOI:10.1109/IPDPS.2018.00029.

Tessier F, Vishwanath V and Jeannot E (2017) Tapioca: An i/o library for optimized topology-aware data aggregation on large-scale supercomputers. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, pp. 70–80.

Xu L, Cipar J, Krevat E, Tumanov A, Gupta N, Kozuch MA and Ganger GR (2014) Springfs: bridging agility and performance in elastic distributed storage. In: *FAST*. pp. 243–255.

Yildiz O, Dorier M, Ibrahim S, Ross R and Antoniu G (2016) 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*. Chicago, United States, pp. 750–759.

Zhou Z, Yang X, Zhao D, Rich P, Tang W, Wang J and Lan Z (2015) I/O-aware batch scheduling for petascale computing systems. In: *2015 IEEE International Conference on Cluster Computing*. pp. 254–263. DOI:10.1109/CLUSTER.2015.45.