# Scheduling periodic I/O access with bi-colored chains: models and algorithms

Emmanuel Jeannot, Guillaume Pallez, Nicolas Vidal

**Journal of Sceduling manuscript No.**
(will be inserted by the editor)

# Scheduling periodic I/O access with bi-colored chains: models and algorithms

**Emmanuel Jeannot · Guillaume Pallez (Aupy) · Nicolas Vidal**

Received: date / Accepted: date

**Abstract** Observations show that some HPC applications periodically alternate between (i) operations (computations, local data-accesses) executed on the compute nodes, and (ii) I/O transfers of data and this behavior can be predicted before-hand. While the compute nodes are allocated separately to each application, the storage is shared and thus I/O access can be a bottleneck leading to contention. To tackle this issue, we design new static I/O scheduling algorithms that prescribe when each application can access the storage. To design a static algorithm, we emphasize on the periodic behavior of most applications. Scheduling the I/O volume of the different applications is repeated over time. This is critical since often the number of application runs is very high. In the following report, we develop a formal background for I/O scheduling. First, we define a model, bi-colored chain scheduling, then we go through related results existing in the literature and explore the complexity of this problem variants. Finally, to match the HPC context, we perform experiments based on use-cases matching highly parallel applications or distributed learning framework

**Keywords** High performance computing, complexity, algorithmics, approximations

## 1 Introduction

Until now, the performance of a supercomputer was mainly measured by its computational power. However, as platforms grow larger and the amount of data involved increases, we encounter new issues. Indeed, the way data is allocated, moved or stored takes an increasing part in the performance of these parallel applications. For instance, on large-scale platform, I/O movement is critical as fetching data out of the storage is becoming a growing fraction of the total runtime. Moreover, while the compute nodes are allocated separately to each application, the storage is shared by many applications. It is often seen that the concurrent I/O access to the storage degrades performance [12][22]. There are two main reasons for that. First, I/O access from the compute node uses the storage infrastructure (network, disks, etc.) and hence several concurrent accesses in "best-effort" mode lead to contention on these resources. Such contention is often *over-additive*: due to hardware restrictions, the time spent by each application executed simultaneously is larger than the time that each application would spend without contention if they were executed alone. The second reason is that when applications compete for resources, they are blocked waiting for their request to be completed. This is suboptimal if we compare this to the

E. Jeannot, G. Pallez, N. Vidal
INRIA Bordeaux Sud-Ouest
200, Avenue de la Vielle Tour
33405 Talence Cedex
France E-mail: first.last@inria.fr

case where each application access these resources one at a time: the time spent doing I/O is much more reduced in the latter case. Therefore, we need to design algorithms that shift the focus from raw computational power to handle the bottleneck due to data management.

To tackle this problem, some approaches aim at reducing the amount of data by compressing or pre-processing it [24][7][8]. Moreover, new hardware features, such as burst buffers, are designed to absorb spike in storage access. However, these solutions do not fully address the problem of resource contention: compression does not prevent several applications to access the storage at the same time, and a burst buffer is limited in size and hence, can also suffer from congestion. Here, the solution we explore adopts a very different point of view that is complementary to the reduction of the amount of data that transit. We aim at managing the I/O data in the system, by scheduling the access at the scale of the system

Our solution is based on observations that show that some HPC applications [5, 10, 12] periodically alternate between (i) operations (computations, local data accesses) executed on the compute nodes, and (ii) I/O transfers of data and this behavior can be predicted beforehand. Literature discussing the I/O behavior prediction is abundant and convincing. Including machine learning approaches ([21]). Taking this structural argument, along with HPC-specific applications characteristics (there are in general very few applications running concurrently on a machine, and the applications run for many iterations with similar behavior) the goal is to design new algorithms for scheduling periodic I/O access. In this paper, we study several approaches (namely periodic and list scheduling) that takes into account the different application pattern (computation time, I/O time, number of iterations, etc.), and aim at defining the time when each application has to perform I/O. Based on different sub-cases, we are able to provide optimal algorithms, approximation algorithms or heuristics. We validate these algorithms using use cases from the literature. We show that given some criteria on the instance, we outperform the best-effort strategy. As the I/O schedule is static, we also study its robustness when inputs are subject to error or noise: in this case we show that, in many cases, our strategies still outperform the best effort one even if the characteristic of the applications are not perfectly known in advance.

## 2 Model

In this section we present a formal model to represent HPC applications alternating between compute phases and I/O phases. The model used has been verified experimentally to be consistent with the behavior of Intrepid and Mira, supercomputers at Argonne [12] and Jupiter, a machine at Mellanox [2]. To do this we introduce a more general notion that we call *bi-colored* chains, where the chain consists of two types of operations (e.g. in this case compute and I/O), that need to be run on two different types of machine. One can then choose how to parametrize the machine consistently with the problem under study (here compute nodes and I/O bandwidth). We call HPC-IO the name of the parametrized instance under consideration in this work.

### 2.1 Machine Model

We consider a platform consisting of two types of machines: type $\mathcal{A}$ and type $\mathcal{B}$. Each of these machines can have either a bounded number of resources or an unbounded number of resources as would be the case in a typical scheduling problem.

In the I/O problem under consideration here, we consider that the jobs are already scheduled on the compute nodes (machine of type $\mathcal{A}$) and that there is no competition at this level. Hence, we can assume w.l.o.g an unbounded number of such resources. On the contrary the bandwidth of the Parallel File System (PFS) (machine of type $\mathcal{B}$) is shared amongst the different jobs. Hence, we say that it has a bounded number of resources $B$. In this work we consider $B = 1$. We call this instance of the platform an *I/O platform*.

We give a schematic overview of this model and of jobs executed on this platform in Figure 1.

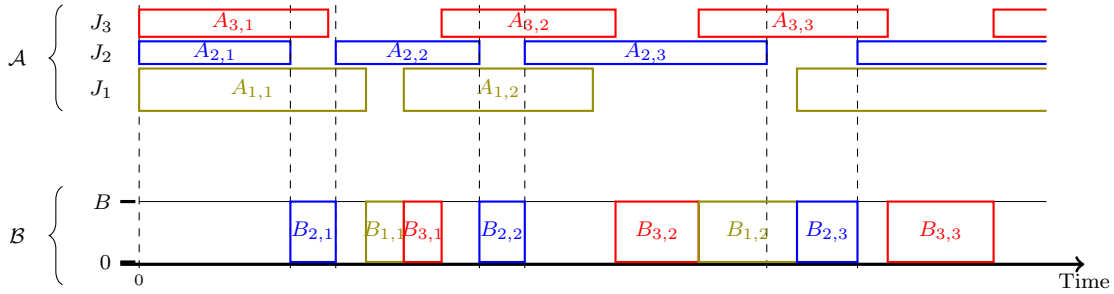



Fig. 1: Schematic overview of three jobs $J_1$, $J_2$, $J_3$ scheduled on a bi-colored platform.

### 2.2 Job Model

We consider scientific applications running simultaneously onto a parallel platform [2,1]. The set of processing resources is already allocated to each application. With respect to I/O, applications consist of consecutive *non-overlapping* phases: (i) a compute phase (executed on machine $\mathcal{A}$); (ii) an I/O phase (executed on machine $\mathcal{B}$) which can be either reads or writes.

Formally, a job $J_i$ consists of $n_i$ successive operation $A_{i,j}$, $B_{i,j}$ ($j \leq n_i$). The dependencies that need to be respected are such that: $A_{i,j+1}$ (resp. $B_{i,j}$) can only start its work when operation $B_{i,j}$ (resp. $A_{i,j}$) is done entirely. We denote by $a_{i,j}$ (resp. $b_{i,j}$) the volume of work of operations $A_{i,j}$ (resp. $B_{i,j}$). In the HPC-IO problem, because there is no constraint on the number of compute nodes allocated to $J_i$, we can assume w.l.o.g that it is equal to 1 and $a_{i,j}$ also corresponds to the execution time of operation $A_{i,j}$. Similarly, when $B_{i,j}$ uses the full I/O bandwidth ($B = 1$), $b_{i,j}$ corresponds to the minimal time to execute operation $B_{i,j}$.

We call such jobs *bi-colored chains* and write them:

$$J_i = (\Pi_{j=1}^{n_i}(A_{i,j}, B_{i,j})) \tag{1}$$

The minimal execution time of $J_i$ is given by the equation:

$$C_i^{\min} = \sum_{j=1}^{n_i} a_{i,j} + b_{i,j} \tag{2}$$

In addition, in this work we consider some specific jobs called *Periodic* jobs. They consist in successions of identical (in volume/time) compute operations and I/O operations. Those are typical patterns in High Performance Computing [5,12,10]. We extend the notation for bi-colored chains to these jobs:

$$J_i = ((A_i, B_i)^{n_i}) \tag{3}$$

### 2.3 Optimization problem

In this Section we detail the HPC-IO optimization problem. In this work, we consider the specific model where the I/O of tasks is rigid: for all applications, the I/O is always performed at full bandwidth and cannot be pre-empted. This model is what is currently implemented in Clarisse [14].

A schedule $\mathcal{S}$ is fully defined by giving an order for the different I/O operations on the machine of type $\mathcal{B}$. Indeed, because there is no competition for the resources of type $\mathcal{A}$:

- $A_{i,1}$ can start immediately;
- $B_{i,j}$ can start as soon as both events are finished: (i) $A_{i,j}$ is finished; (ii) all jobs anterior to $B_{i,j}$ in the schedule on the machine of type $\mathcal{B}$ are finished.

– $A_{i,j+1}$ can start as soon as $B_{i,j}$ is finished.

Hence, we can formally define a schedule:

**Definition 1 (A schedule $\mathcal{S}$)** Given a set of jobs $J_i = (\Pi_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$, a schedule $\mathcal{S}$ is defined by a permutation of the jobs $\left((B_{i,j})_{j\leq n_i}\right)_i$ that satisfies, for all $i, j$, $B_{i,j}$ is before $B_{i,j+1}$

We consider the classical objective function for scheduling problem. It corresponds to the system performance (makespan or execution time). In the future, we may study system fairness as well.

Let $C_i$ be the end of the execution of a job $J_i$ in the schedule $\mathcal{S}$. We define the makespan $C_{\max}^{\mathcal{S}}$ of the schedule $\mathcal{S}$ to be:

$$C_{\max}^{\mathcal{S}} = \max C_i \tag{4}$$

**Definition 2 (MS-HPC-IO)** Given a set of rigid bi-colored chains $J_i = (\Pi_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$, and an I/O platform. Find a schedule that minimizes the makespan $C_{\max}^{\mathcal{S}}$.

## 3 Complexity of HPC-IO

### 3.1 Intractability

In this section we briefly present some intractability results from the literature for MS-HPC-IO.

*MS*-HPC-IO In the literature, several results relate to this problem. The closest to our model is the Precedence Constrained Scheduling problem introduced by Wikum [26], which studies the special case of MS-HPC-IO.

**Theorem 1 ([26, Proposition 2.3])** *MS*-HPC-IO *is NP-complete, even in the simplest case when $n_1 = 2$, and for all jobs $J_i$, $i \neq 1$, $n_i = 1$.*

### 3.2 Polynomial algorithms

In this Section we present some instances where one can compute the optimal solution in polynomial time. We focus here on instances that are important for the HPC-IO problem. Several other specific instances have been studied by Wikum [26].

*Case when $\forall i, n_i = 1$* When for all jobs $J_i$, $n_i = 1$, it is easy to see that any greedy solution that schedules the I/O as soon as they are available is optimal for MS-HPC-IO [26, Proposition 2.1].

*Uniform jobs* We study the case of uniform jobs which is a specific case of periodic jobs. Specifically we consider that there exists $A, B$ s.t., for all $i, j$, $A_{i,j} = A$ and $B_{i,j} = B$. We can then write: $J_i = ((A, B)^{n_i})$. Those jobs can be used to represent some new types of workloads such as hyperparametrization in Machine Learning (see Section 5.3 for more details). In this context, all jobs are part of a bigger job and are released at the same time. Because they are part of a bigger job, we are interested in solving MS-HPC-IO. In this section, w.l.o.g we assume that the jobs $(J_i)_{1\leq i\leq m}$ are sorted by decreasing value of $n_i$.

**Definition 3 (Uniform)** Given a set of jobs $(J_i)_{1\leq i\leq m}$ s.t. $\forall i, r_i = 0, n_i \geq n_{i+1}$ and there exists $A, B$ s.t., for all $i, j$, $A_{i,j} = A$ and $B_{i,j} = B$, UNIFORM is the problem of solving MS-HPC-IO.

**Theorem 2** UNIFORM *can be solved in polynomial time.*

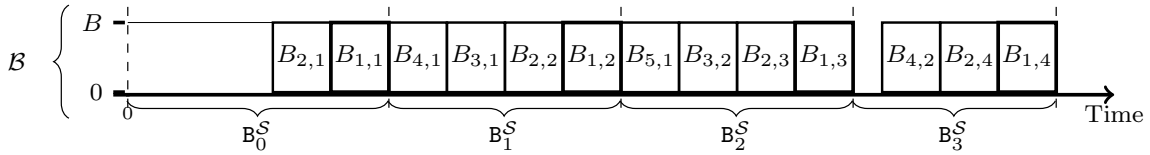



Fig. 2: $J_1 = (2.5, 1)^4$, $J_2 = (2.5, 1)^4$, $J_3 = (2.5, 1)^2$, $J_4 = (2.5, 1)^2$, $J_5 = (2.5, 1)^1$

To show this result, we show that Algorithm 1 (Hierarchical Round-robin) solves the problem in polynomial time. The idea of Hierarchical Round-robin is to structure the schedule around the job with the largest $n_i$.

We start by scheduling each $B$ operation of $J_1$. Then, we schedule before each of those $B$ operations all $B$ operations of jobs such that $n_i = n_1$. Finally, we schedule all remaining jobs in a round-robin fashion between $B_{1,1}$ and $B_{1,n_1}$. We present in Figure 2 an example of a schedule.

**Algorithm 1** Hierarchical Round-robin

**procedure** HRR($J_i = (\Pi_{j \leq n_i} A_{i,j}, B_{i,j})$) ▷ $\forall i, j, n_i \geq n_{i+1}, A_{i,j} = A, B_{i,j} = B$
  Let $S_0, \cdots, S_{n_1 - 1}$ be $n_1$ empty stacks.
  $\mathrm{Id}_b \leftarrow 1$.
  **for** $i = 1$ to $|\{J_i\}|$ **do**
    **if** $n_i = n_1$ **then**
      **for** $j = 1$ to $n_i$ **do**
        Add $B_{i,j}$ to $S_{j-1}$.
    **else** ▷ We do not schedule anymore on $S_0$.
      $\mathrm{Id}_e \leftarrow 1 + (\mathrm{Id}_b + n_i \mod (n_1 - 1))$ ▷ $J_i$ is scheduled from $S_{\mathrm{Id}_b+1}$ to $S_{\mathrm{Id}_e}$
      **if** $\mathrm{Id}_b \leq \mathrm{Id}_e$ **then**
        **for** $j = 1$ to $n_i$ **do**
          Add $B_{i,j}$ to $S_{j+\mathrm{Id}_b}$.
      **else**
        **for** $j = 1$ to $\mathrm{Id}_e$ **do**
          Add $B_{i,j}$ to $S_j$.
        **for** $j = \mathrm{Id}_e + 1$ to $n_i$ **do**
          Add $B_{i,j}$ to $S_{(n_1-1)-(n_i-j)}$.
    $\mathrm{Id}_b \leftarrow \mathrm{Id}_e$.
  **return** $\mathcal{S}_{HRr} = S_0 \cdot S_1 \cdot \cdots \cdot S_{n_1-1}$

We now show formally Theorem 2. To do so:

1. We define of a cost function $\mathcal{C}$ (Def. 5) such that for all schedule $\mathcal{S}$, $C^{\mathcal{S}}_{\max} \geqslant \mathcal{C}(\mathcal{S})$ (Prop. 1);
2. We show that there exists an optimal schedule $\mathcal{S}_{opt}$ such that $\mathcal{C}(\mathcal{S}_{opt}) \geqslant \mathcal{C}(\mathcal{S}_{HRr})$ (where $\mathcal{S}_{HRr}$ is the schedule returned by Hierarchical Round-robin);
3. Finally, we show that $C^{\mathcal{S}_{HRr}}_{\max} = \mathcal{C}(\mathcal{S}_{HRr})$ (Prop. 2), showing the result.

In the rest of this Section, we let $J_i = (0, \Pi_{j=1}^{n_i}(A_{i,j}, B_{i,j}))$ be a set of uniform jobs sorted by decreasing $n_i$. We denote by $a$ (resp. $b$) the execution tasks of tasks $A_{i,j}$ (resp. $B_{i,j}$).

We introduce the notion of block:

**Definition 4 (Block of a schedule $\mathcal{S}$ and its cost)** Given a schedule $\mathcal{S}$, for $k \in [[1, n_1]]$, we define the block $\mathtt{B}_k^{\mathcal{S}}$ to be:

- If $k = 1$, $\mathtt{B}_1^{\mathcal{S}}$ is the set of tasks scheduled to be executed before (including) $B_{1,1}$.
- Otherwise, $\mathtt{B}_k^{\mathcal{S}}$ is the set of tasks scheduled to be executed after (excluding) $B_{1,k-1}$ and before (including) $B_{1,k}$.

We define the cost of a block to be:

$$C(\mathtt{B}_k^{\mathcal{S}}) = \begin{cases} a + |\mathtt{B}_1^{\mathcal{S}}| & \text{if } k = 1 \\ max(a+b, |\mathtt{B}_k^{\mathcal{S}}| \cdot b) & \text{else} \end{cases}$$

We represent the notion of Blocks on Fig. 2

**Definition 5 (Cost of a schedule)** Given a schedule $\mathcal{S}$, its cost is $\mathcal{C}(\mathcal{S}) = \Sigma_{k=1}^{n_1} C(\mathtt{B}_k^{\mathcal{S}})$, where $C$ is the function cost of a block.

**Proposition 1** *For any schedule $\mathcal{S}$, $C_{\max}^{\mathcal{S}} \geqslant \mathcal{C}(\mathcal{S})$.*

*Proof* To obtain this result, one can observe that the blocks partition the schedule until $B_{1,n_1}$, and hence the total makespan is greater than the sum of the makespan of all blocks[1]. Then, we need to show that the makespan of each block is greater than the cost of each block, hence showing the result. This comes naturally from the fact the makespan of a block is necessarily greater than the maximum between (i) the total work that has to be performed during this block ($|\mathtt{B}_k^{\mathcal{S}}| \cdot b$), and (ii) the minimal length imposed by $J_1$ (an execution of $A_{i,j}$ and an execution of $B_{i,j}$). Hence, the makespan of a block is greater than its cost, showing the result.

**Definition 6 (Dominant schedules)** For Uniform, we say that a schedule is *dominant* if:

1. **Prop. (Dom.1)** The last task executed on platform $\mathcal{B}$ is $B_{1,n_1}$;
2. **Prop. (Dom.2)** For all $J_i$, $(n_i - j + i) < n_1$, implies $B_{i,j}$ is executed after $B_{1,1}$.
3. **Prop. (Dom.3)** For all $J_i$ s.t. $n_i = n_1$, $B_{i,1}$ is executed before $B_{1,1}$.

In practice *Dominant Schedules* are schedules that finish by the last operation of $J_1$, and that start by all first operations of *long* jobs and then by $B_{1,1}$.

**Lemma 1** *There exists a dominant schedule which is optimal.*

*Proof* We show the result in three steps:

1. First, we show that there exists an optimal algorithm which ends by the execution of $B_{1,n_1}$;
2. Amongst those optimal algorithms, we show that there exists at least one where for all $J_i$ s.t. $(n_i - j + 1) < n_1$, implies $B_{i,j}$ is executed after $B_{1,1}$;
3. Finally, amongst those, we show that there exists at least one s.t. for all $J_i$ s.t. $n_i = n_1$, $B_{i,1}$ is executed before $B_{1,1}$.

*There exists an optimal algorithm that satisfies Prop. (Dom.1)* We show the result by contradiction. Assume there does not exist an optimal schedule which ends by the execution of $B_{1,n_1}$.

Let $\mathcal{S}$ be an optimal schedule for Uniform that minimizes the number of operations following $B_{1,n_1}$. Let $B_{i,k}$ be the operation directly subsequent to $B_{1,n_1}$ in the schedule.

If $k = n_1$, then because all $A_{i,j}$ are identical, for $1 \leq j \leq k$, we can permute all $B_{i,j}$ operations with $B_{1,j}$ without increasing the makespan, and the number of operations after $B_{1,n_1}$ decreased strictly, contradicting the minimality of $\mathcal{S}$.

Otherwise, necessarily $k < n_1$ (indeed, by definition, for all $i$, $n_i \leq n_1$). In this case, necessarily there exist two consecutive operations of $J_1$ such that there are no operations of $J_i$ between them. Let us call $B_{1,n_1-j_0-1}$ and $B_{1,n_1-j_0}$ those last operations. Then, because all jobs are identical, for $0 \leq j \leq j_0$, we can permute all $B_{i,k-j}$ operations with $B_{1,n_1-j}$ operations without increasing the total makespan. In this new schedule, the number of operations after $B_{1,n_1}$ decreased strictly, hence contradicting the minimality of $\mathcal{S}$.

We denote by $A_{OPT}^1$ the non-empty set of optimal schedules that satisfy Prop. (Dom.1).

[1] Where the makespan of block $\mathtt{B}_k^{\mathcal{S}}$ (resp. $\mathtt{B}_1^{\mathcal{S}}$) is naturally defined as the time between the beginning of the execution of $B_{1,k}$ on platform $\mathcal{B}$ and the beginning of the execution of $B_{1,k+1}$ on platform $\mathcal{B}$.

*There exists a schedule in $A^1_{OPT}$ that satisfies Prop. (Dom.2)* Similarly, we show the result by contradiction. Assume that for all schedules of $A^1_{OPT}$, none satisfy Prop. (Dom.2).

Let $\mathcal{S} \in A^1_{OPT}$ that minimizes the number of operations $B_{i,j}$ that satisfy (i) $B_{i,j}$ is scheduled before $B_{1,1}$; (ii) $n_i - jn_1 - 1$. Let $B_{i,j_0}$ be the last of these operations before $B_{1,1}$ in $\mathcal{S}$.

Then, because $(n_i - j_0 + 1) < n_1$, necessarily there exists $k < n_1$ s.t. there are no operations of $J_i$ between $B_{1,k}$ and $B_{1,k+1}$. Let us denote by $k_0$ the smallest of such $k$. Then, for $j \in \{1, \cdots, k_0\}$we can permute in $\mathcal{S}$ all operations $B_{1,j}$ and $B_{i,j_0-1+j}$ without increasing the schedule length. Indeed, there is no new idle time between any pair of operations $B_{1,j}$ and $B_{1,j+1}$ for $j < k_0$ (because $a_{1,j} = a_{i,j_0-1+j} = a$, nor between $B_{1,k_0}$ and $B_{1,k_0+1}$ because $B_{1,k_0}$ was advanced in time while $B_{1,k_0+1}$ did not move. Similarly, there is no new idle time created in the schedule between $B_{i,j_0-1+j}$ and $B_{i,j_0+j}$. $B_{i,j_0+k_0}$ is scheduled after $B_{1,k_0+1}$ while $B_{i,j_0-1+k_0}$ is scheduled where $B_{i,k_0}$ was scheduled, so the time difference between them is greater than $a$.

Finally, this did not impact either any other jobs because the number of jobs on $\mathcal{B}$ between two occurrences on any other jobs was kept the same.

We can conclude that this transformation did not increase the execution time. In addition, it did not change the schedule after $B_{1,k_0+1}$ where $k_0 + 1 \leq n_1$, hence Prop. (Dom.1) is still respected in this new optimal schedule. There was, however, one fewer job before $B_{1,1}$, contradicting the minimality of $\mathcal{S}$.

We denote by $A^2_{OPT}$ the non-empty set of optimal schedules that satisfy both Prop. (Dom.1) and Prop. (Dom.2).

*There exists a schedule in $A^2_{OPT}$ that satisfies Prop. (Dom.3)* Similarly, we show the result by contradiction. Assume that for all schedules of $A^2_{OPT}$, none satisfy Prop. (Dom.3).

Let $\mathcal{S} \in A^1_{OPT}$ that minimizes the number of operations $B_{i,1}$ that satisfy (i) $B_{i,1}$ is scheduled after $B_{1,1}$; (ii) $n_i = n_1$. Let $B_{i_0,1}$ be the first of these operations after $B_{1,1}$ in $\mathcal{S}$.

By a reasoning very similar to the one used to prove the existence of the set $A^2_{OPT}$, one can show that $\mathcal{S}$ can be chosen such that $B_{i_0,1}$ is the operation directly subsequent to $B_{1,1}$.

Because $n_{i_0} = n_1$, and because $\mathcal{S}$ satisfies Prop. (Dom.1), there exists $j_0 \geq 1$ such that $B_{i_0,j_0}$ and $B_{i_0,j_0+1}$ are scheduled between $B_{1,j_0}$ and $B_{1,j_0+1}$.

Thanks to the property that $\forall i, j, a_{i,j} = a$, we can then create a new schedule whose execution time is not greater than that of $\mathcal{S}$ by permuting for $1 \leq j \leq j_0$, $B_{i_0,j}$ and $B_{1,j}$. This schedule still satisfies Prop. (Dom.1) (we have not modified the location of $B_{1,n_1}$), and Prop. (Dom.2) (the only task that was moved before $B_{1,1}$ is $B_{i_0,1}$), contradicting the minimality of $\mathcal{S}$.

Finally, this concludes the proof that there exists an optimal schedule that is dominant.

**Lemma 2** *Denote by $l_1 = |\{J_i | n_i = n_1\}|$ and $\mathcal{S}_{HRr}$ the solution returned by* Hierarchical Round-robin. *Let $r_1 = (\sum_i n_i - l_1) \mod (n_1 - 1)$, and $q_1 = \lfloor \frac{\sum_i n_i - l_1}{(n_1-1)} \rfloor$ Then, we have the following results:*

- $|\mathcal{B}_1^{\mathcal{S}_{HRr}}| = l_1$,
- *for $j = 2$ to $r_1 + 1$, $|\mathcal{B}_j^{\mathcal{S}_{HRr}}| = q_1 + 1$,*
- *for $j = r_1 + 2$ to $n_1$, $|\mathcal{B}_j^{\mathcal{S}_{HRr}}| = q_1$.*

*Proof* This is a direct consequence from Algorithm 1. One can notice that $\mathtt{B}_k^{\mathcal{S}_{HRr}}$ corresponds to $S_{k-1}$ as returned at the end of the execution.

Hence, $\mathtt{B}_1^{\mathcal{S}_{HRr}}$ only contains the first operation of jobs of length $n_1$ (hence $l_1$ operations), and the rest of the blocks share the remaining operations minus those $l_1$ operations, hence the result.

**Lemma 3** *Given $\mathcal{S}$ a* dominant schedule*, then $\mathcal{C}(\mathcal{S}) \geq \mathcal{C}(\mathcal{S}_{HRr})$.*

*Proof* In this proof we use the definition of $l_1$, $q_1$ and $r_1$ as defined in Lemma 2.

Let $\mathcal{S}$ be a dominant schedule. Denote by $p_{\min} = \min_{k=2}^{n_1}\{|\mathtt{B}_k^{\mathcal{S}}|\}$ (resp. $p_{\max} = \max_{k=2}^{n_1}\{|\mathtt{B}_k^{\mathcal{S}}|\}$, the smallest (resp. largest) block size for all blocks of $\mathcal{S}$ but the first one.

We show the result by recurrence on . By definition of a dominant schedule, we know that $\sum_{k=2}^{n_1} |\mathtt{B}_k^{\mathcal{S}}| = \sum_i n_i - l_1$, hence necessarily $p_{\min} \leq q_1 \leq p_{\max}$.

By definition of $q_1$ and $r_1$, if $p_{\max} - p_{\min} \leq 1$, then $p_{\min} = q_1$ and there are exactly $r_1$ blocks of size $p_{\max}$ and $n_1 - r_1$ blocks of size $p_{\min}$. Hence, $\mathcal{C}(\mathcal{S}) = \mathcal{C}(\mathcal{S}_{HRr})$. In the following we assume that $p_{\max} - p_{\min} > 1$. In particular we have: $p_{\min} \leq q_1 < q_1 + 1 \leq p_{\max}$.

*If $p_{\min} \cdot b \geq a + b$ (resp. $p_{\max} \cdot b \leq a + b$)* Then, we have:

$$\sum_{k=2}^{n_1} C(\mathtt{B}_k^{\mathcal{S}}) = \sum_{k=2}^{n_1} |\mathtt{B}_k^{\mathcal{S}}| \cdot b = b \left( (\sum_i n_i) - l_1 \right) = b \sum_{k=2}^{n_1} |\mathtt{B}_k^{\mathcal{S}_{HRr}}| = \sum_{k=2}^{n_1} \mathcal{C}(\mathtt{B}_k^{\mathcal{S}_{HRr}})$$

(resp. $\sum_{k=2}^{n_1} C(\mathtt{B}_k^{\mathcal{S}}) = \sum_{k=2}^{n_1} a + b = \sum_{k=2}^{n_1} \mathcal{C}(\mathtt{B}_k^{\mathcal{S}_{HRr}})$), meaning that $\mathcal{C}(\mathcal{S}) = \mathcal{C}(\mathcal{S}_{HRr})$.

*Else, $p_{\min} \cdot b < a + b < p_{\max} \cdot b$* In this case, because $|p_{\max} - p_{\min}| \geq 2$, we can show that the cost is strictly greater to the cost of a solution with one element fewer in the largest block, and one more element in the smallest block. This can be done recursively until one of the initialization case as seen above (either $|p_{\max} - p_{\min}| \leq 1$, $p_{\min} \cdot b \geq a + b$, or $p_{\max} \cdot b \leq a + b$) for which we have shown that the cost is equal to $\mathcal{C}(\mathcal{S}_{HRr})$.

Indeed, assume the cost of the smallest block increases by $0 \leq \delta < b$ (resp. cost of the largest bock decreased by $0 < \delta \leq b$). Then, $a + b \leq (p_{\max} - 1) \cdot b$ (resp. $(p_{\min} + 1) \cdot b \leq a + b$), and the cost of the largest block decreased by $b$ (resp. the cost of the smallest block did not increase). Hence, the total cost decreased by $b - \delta > 0$ (decreased by $\delta > 0$).

Again, the path of solutions may not theoretically exist, however this process shows that their cost is indeed greater than that of $\mathcal{S}_{HRr}$.

**Proposition 2** $\mathcal{C}(\mathcal{S}_{HRr}) = C_{\max}^{\mathcal{S}_{HRr}}$

*Proof* We study the stacks $S_0, \cdots, S_{n_1 - 1}$ as returned by Algorithm 1. Note that we have seen that there execution time is necessarily at least equal to their cost because of $J_1$. We now show that this time is enough for a successful execution of the schedule.

The time to execute $S_0$ is exactly $\mathcal{C}(S_0)$, indeed all jobs in this stack are executed for the first time, hence we need to wait for a time $a$, then all I/O operations are ready and we can execute them consecutively (taking a time $|S_0| \cdot b$.

We then show the result on the other stacks by studying the $l^{\text{th}}$ element from the bottom of the stack (the first element of each stack $S_k$ is $B_{1,k+1}$).

Given stack $S_k$, denote by $B_{i,j}$ its $l^{\text{th}}$ element:

- Either $j = 1$, in which case it was ready since $S_0$ and there are no additional time constraints;
- Or $B_{i,j-1}$ was put on stack $S_{k-1}$, then it was at the $l^{\text{th}}$ position of the stack because the stack is balanced. In which case, there are exactly $l - 1$ (resp. $|S_k| - l$) operations on stack $S_{k-1}$ (resp. $S_k$) between those two operations, hence a total time of $(|S_k| - 1) \cdot b$. Hence, we need an idle time at the beginning of the execution of $S_k$ of length $\max(0, a - (|S_k| - 1) \cdot b)$, and an execution time for $S_k$ of $\mathcal{C}(S_k)$ is enough for its successful execution.
- Finally, with the round robin property, $B_{i,j-1}$ could be scheduled on stack $S_{k'}$ where $k' < k - 1$. In this case the time constraint is also respected because $S_{k-1}$ takes by definition more than $a$ units of time.

Hence, the result, we have shown that an execution time equal to the cost for each task was enough to satisfy all the time constraints.

*Proof (Proof of Theorem 2)*

There exists an optimal schedule $\mathcal{S}_{opt}$ to Uniform, such that (i) $C_{\max}^{\mathcal{S}_{opt}} \geq \mathcal{C}(\mathcal{S}_{opt})$ (Prop. 1); (ii) $\mathcal{C}(\mathcal{S}_{opt}) \geq \mathcal{C}(\mathcal{S}_{HRr})$ (Lemma 1 and Lemma 3). Finally, we have seen (Prop. 2) that $\mathcal{C}(\mathcal{S}_{HRr}) = C_{\max}^{\mathcal{S}_{HRr}}$, proving that Hierarchical Round-robin is optimal.

## 4 Approximation algorithms for MS-HPC-IO

We have seen in Section 3 that MS-HPC-IO was in general intractable. A natural question to this is whether there exist efficient approximation algorithms. In this section we show some results on list-scheduling algorithms, then discuss a specific framework of algorithms, periodic algorithms.

**Definition 7 (Approximation algorithm)** For a maximization (resp. minimization) problem $\mathcal{P}$, we say that an algorithm $\mathcal{A}$ is a $\lambda$-approximation algorithm, if for any instance $I \in \mathcal{P}$, $\mathcal{A}(I) \leq \lambda \mathcal{A}_{OPT}(I)$ (resp. $\mathcal{A}(I) \geq \lambda \mathcal{A}_{OPT}(I)$) (where $\mathcal{A}_{OPT}$ is an optimal algorithm for $\mathcal{P}$).

### 4.1 List Scheduling algorithms

We start by considering *list scheduling* strategies (also called *greedy*) which are often considered the most natural algorithms: at all time, either the machine $\mathcal{B}$ is busy or no work of type $\mathcal{B}$ is available. When the machine becomes idle and some multiple operations are available, the machine sorts them (and schedule them) following a priority order.

**Theorem 3** *Any list-scheduling algorithm is a 2-approximation for MS-*HPC-IO *and this ratio is tight.*

*Proof* First, we show that any list-scheduling algorithm is at best a factor two of the optimal for MS-HPC-IO.

We create the instance $I_\varepsilon$: $J_1 = ((A_{1,1} = 0, B_{1,1} = 1))$, $J_2 = ((A_{2,1} = \varepsilon, B_{2,1} = \varepsilon) \cdot (A_{2,2} = 1, B_{2,2} = 0))$. The makespan of any list-scheduling algorithm is: $2+\varepsilon$. Indeed, at $t = 0$, a list-scheduling algorithm has to schedule $B_{1,1}$ because it is the only operation ready. Then, once it is done, it can schedule $B_{2,1}$, which will be followed by the execution of $A_{2,2}$ and $B_{2,2}$.

On the other hand, an optimal schedule waits for $\varepsilon$ units of time so it can schedule $B_{2,1}$ first. Then, it schedules $B_{1,1}$ while $A_{2,1}$ executes. The total execution time is $1+2\varepsilon$. Hence, the approximation ratio is at least:

$$\lambda = \sup_{\varepsilon > 0} \frac{2+\varepsilon}{1+2\varepsilon} = 2$$

We now show that any list-heuristic algorithm is at most a 2-approximation. Given an instance of the problem, let $C_{\max}^{List}$ be the makespan of a list-scheduling algorithm and $C_{\max}^{OPT}$ be the makespan of an optimal algorithm.

Necessarily, $C_{\max}^{OPT} \geq \max_i(\sum_j a_{i,j}+b_{i,j})$ which is the minimal time needed for the longest application $J_i$. We focus on the occupation of platform $\mathcal{B}$. $C_{\max}^{List} = \sum_i \sum_j b_{i,j} + t_{\text{idle}}$, where $t_{\text{idle}}$ is the time where platform $\mathcal{B}$ is waiting for work. Let $B_{i_0,n_{i_0}}$ be the last operation scheduled on $\mathcal{B}$. Then, necessarily, $t_{\text{idle}} \leq \sum_{j=1}^{n_{i_0}} a_{i,j}$.

Hence, we have:

$$C_{\max}^{List} = \sum_i \sum_j b_{i,j} + t_{\text{idle}} \leq C_{\max}^{OPT} + \sum_{j=1}^{n_{i_0}} a_{i,j} \leq 2C_{\max}^{OPT}$$

### 4.2 Periodic algorithms

In this section we focus on periodic applications as defined in Section 2. These applications are very frequent in our target framework, High-Performance Computing (the most common example is that of applications that store their checkpoint at regular intervals for resilience purpose [6]). To tackle them, we study a specific sort of algorithms: *Periodic algorithms*. Indeed it has been shown that those algorithms have many efficient property such as a low memory and compute overhead with excellent performance when the number of operations per jobs is very high [2]. We are interested here in giving some theoretical results that motivates these recent results.

We start by showing that in some context, those algorithms are efficient approximations for the MS-HPC-IO problem.

We define formally a periodic algorithm:

**Definition 8 (Periodic Algorithm)** Given a periodic instance $J = ((a_i, b_i)^{k_i \cdot n})$.

A periodic algorithm $\mathcal{P}$ constructs a *period* which is a schedule of $J = ((a_i, b_i)^{k_i})$: then returns a schedule built by $n$ periodic repetition of the period.

*Periodic algorithms for MS-HPC-IO* We start by considering periodic jobs whose $n_i$ are all equal. In this case HIERARCHICAL ROUND-ROBIN is a periodic algorithm.

**Theorem 4** HIERARCHICAL ROUND-ROBIN *is a* $1 + 1/n$*-approximation algorithm for MS-HPC-IO where all jobs are periodic with the same number of periods (there exists* $n$*, such that* $\forall i,\ J_i = ((A_i, B_i)^n)$*), and the bound is tight.*

*Proof* First, we discuss the way of ordering tasks within a period and then discuss the performance of such scheduling algorithms.

- In the following, I call "idle time" of a schedule $\mathcal{S}$, the time $t_i(\mathcal{S}) = MS_{\mathcal{S}} - \sum_i nb_i$
- In PERIODIC, all jobs have only one task in each period. We can define the order $\prec$: $i \prec j$ if and only if $b_i$ appears before $b_j$ in the period.

The overall idle time of the periodic schedule is:
$t_i(Periodic) = (n-1).\max_i \left(a_i - \sum_{j \neq i} b_j\right) + \max_k \left(a_k - \sum_{k \prec j} b_j\right)$

The order within a period does not change the overall idle time, therefore we can sort tasks by non-increasing A-task length in a period with gives:

$$t_i(Periodic) \leq (n-1).\max_i \left(a_i - \sum_{j \neq i} b_j\right) + \max_k (a_k)$$

Given an optimal schedule $\mathcal{S}_{opt}$, the idle time is:
$t_i(\mathcal{S}_{opt})\max_i \left(na_i - n\sum_{j \neq i} b_j + \sum_{j \prec i} b_j\right) \geq n.\left(\max_k \left(a_k - \sum_{j \neq k} b_j\right)\right)$

where i is the last task running on A and $i_1$ is its first iteration. Therefore, using straightforward bounds, the difference between these periodic and opt is at most:
$n.\max_i (a_i) - n\left(\max_k \left(a_k - \sum_{j \neq k} b_j\right)\right) \leq n.\max_i (a_i)$ The optimal makespan is at least $n.\max_i (a_i + b_i)$

*Remark 1* One can notice that HIERARCHICAL ROUND-ROBIN is asymptotically optimal for MS-HPC-IO when all jobs are periodic with the same number of periods. In addition, one can slightly improve the result by sorting the jobs by decreasing values of $a_i$.

## 5 Evaluation

In this section we present the experimental evaluation of the proposed solutions. To evaluate them we have designed a simulator that implements the model described in section 2 ie a virtual platform with two machine types, with no competition on resource $\mathcal{A}$ and competitive, exclusive accesses on machine $\mathcal{B}$.

### 5.1 Heuristics

For the purpose of evaluation, we compared several heuristics, list-scheduling heuristic as well as very simple periodic algorithms. These heuristics use several *priority order* to choose which task to execute next. One of them is Johnson's priority order

**Definition 9 (Johnson's order)** Given a set of couples $(a_i, b_i)$, divide the values into two disjoint groups $G_1$ and $G_2$ , where $G_1$ contains all couples $(a_i, b_i)$ with $a_i \leqslant b_i$, and $G_2$ contains all couples $(a_j, b_j)$ with $a_j > b_j$ . Order the couples in a sequence such that the first part consists of the values in $G_1$ , sorted in nondecreasing order of $a_i$, and the second part consists of the values in $G_2$ , sorted in nonincreasing order of $b_j$.

The reason why Johnson's order is considered is because if jobs are $J_i = (a_i, b_i)^1$, it is known that the schedule using Johnson order minimizes the completion time of the flowshop. [27].

*List scheduling* In list scheduling policy, as soon as I/O is free, we execute the most critical, available application. We used different orders to define the criticality of a given application:

- **FIFO**: the applications I/O are executed in the order of their request.
- **Johnson**: the application I/Os are executed following Johnson's order (see definition 9)
- **Most Remain**: When scheduling an I/O, pick in priority the application with the most remaining work to do.

*Periodic* We also use a simple variation of periodic heuristics defined in 4.2. Given an instance $J_i = (a_i, b_i)^{n_i}$, each period of the periodic algorithm contains exactly one task for each job until one of the job is completed. The jobs are sorted following the three orders used for list-scheduling heuristics (FIFO, Johnson, Most Remain). The completion of a job or the release of a new one does not change the relative order of the other. Hence, the period holds after such events (with the exception of the completed job who is not running anymore).

*Best effort* With the best effort strategy, there is no schedule of I/O accesses. Instead of waiting their turn to perform I/O operations, concurrent applications accessing the storage system share equally the bandwidth without additional loss. If $k$ applications are performing I/O operations, an application with $b$ amount of I/O will have, after $t$ units of time, $b - \frac{t}{k}$ remaining amount of I/O. The best effort strategy models what happens in real systems when there is no congestion control or I/O scheduling at the level of the applications.

### 5.2 Scenarios/Use-case and instantiation

Applications are modeled by their computation, I/O durations, and their number of periods. An input file describes an *instance* of the problem as a set of $m$ applications and is generated according to table 1. We have two different cases that represent realistic settings.

Table 1: Parameters used for input generation ($u(a, b)$ stands for drawing uniformly in $[a, b]$)

| cases | m | $a_i$ | $b_i$ | $n_i$ | $r_i$ | #instances |
|---|---|---|---|---|---|---|
| General | u(2,15) | u(1,20) | u(0.1,1)$a_i$ | u(5,150) | 0 | 1000 |
| Uniform | u(2,15) | u(1,20) | u(0.1,1)$a_i$ | u(100,200) | 0 | 1000 |

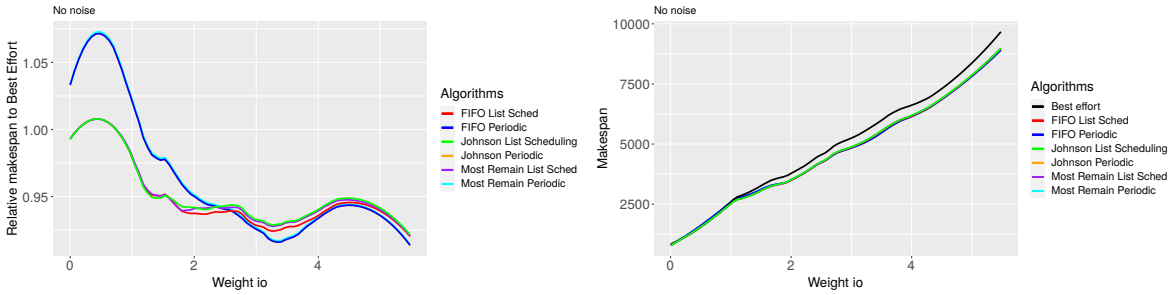
The Uniform case is used for a machine learning multi-parameter training and covers the results of Section 3.2.

### 5.3 Results

In Figure 3, we present the makespan for the general case. The presented graph is the smoothed conditional means on a set of 1000 instances of each case as a function of the weight of I/O, $W$,that accounts for a normalize way of measuring the amount of I/O:

$$W = \sum_i \frac{\sum_j b_{i,j}}{\sum_j a_{i,j} + b_{i,j}}$$

In this Figure, we see that, when the weight of I/O is small, the best effort strategy provides the fastest makespan. This is due to the fact that when there are few I/O, scheduling them is not very useful. However, as soon as the amount if I/O increases, the scheduling strategies improves and outperform the best effort one. Moreover, we see two groups of curves. Periodic schedules and list-scheduling ones. The periodic strategies, *FIFO Periodic*, *Johnson Periodic* and *Most Remain Periodic* are superposed. If we compare these two sets of strategies, we see that when the amount of I/O is small relative to the total of work, list scheduling perform better than periodic strategies and when the weight of I/O increases the periodic strategies are better than the list-scheduling ones. Indeed, when there is few I/O the periodic schedule can force an application to wait for their turn while when there is a high amount of I/O, the short view of the problem by list scheduling algorithm hinder their capacity to handle I/O burst. Whereas all strategies have overall the same makespan evolution, the normalization with best effort (Fig. 3a) accentuate a variation in performance around W=3.This can be explained as follows: after a certain threshold in I/O weight, bandwidth is always busy, therefore all applications have to wait for I/O accesses which mitigate a little the gain. Whereas for low weights, our algorithms reduce the I/O contention, and for high I/O weights it provides a better repartition on I/O operations compared to the best effort strategy. If we look at the absolute tendencies (see Fig. 3b), we observe that the difference bewteen best-effort and the other strategies is globally increasing with the weight of I/O. Some fluctutations for the relative charts are also due to the impact of the input workload.




(a) Relative performance to the Best effort strategy

(b) Absolute performance

Fig. 3: Policies performance comparison on generic inputs for the makespan

*Uncertainty and noise* In our implementation, list scheduling and periodic policies assume that the I/O and computation duration are known in advance. However, in practice these values can never be known with a complete certainty. To model this uncertainty we have added noise to I/O and computation duration. This means that each computation or I/O phase can be subject to a variation in the range of the noise value around the expected, periodical amount. This variation is is determined independently for each phase. It is generated based on a seed that is included with the application specification in order to be reproducible. Indeed, we want this variation to be the same without any concern of the

application order. The generation follows a uniform distribution. We expect this to be a worse scenario than a normal distribution therefore providing more arguments to discuss our policies robustness.

In Fig 4, we present the results with respectively 20% and 50% of noise using the same inputs as for the one in Fig 3.

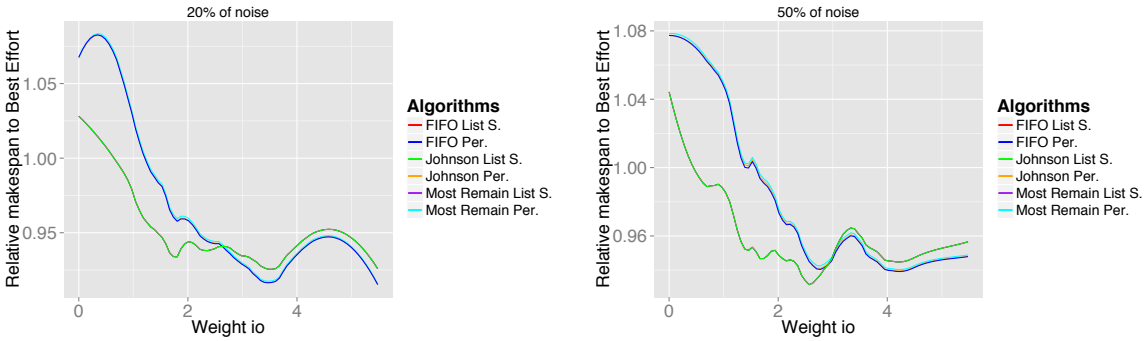



Fig. 4: Policies performance comparison on generic inputs for the makespan relative to the Best effort strategy with uniform noise on the computation or I/O duration

We see that adding noise slightly degrades the performance when the amount of I/O is small compared to the total amount of work. However, when the weight of I/O increases we observe relatively similar performance compared to the case without noise. This means that our strategies are robust to the uncertainty of the duration especially when the amount of I/O is large.

*Machine Learning Use-Case* We describe here a use case where a set of applications is launched at the same time and perform periodic I/O. The goal is to train, in parallel, several deep-learning networks (DLNs) on the same dataset. It works as follows. A set of $m$ nodes of a parallel machine is reserved. $m$ DLNs are generated and trained separately on each node. The goal is to find the best network among the $m$ ones. Therefore, they are trained on the same dataset. Each DLN access a subpart of the dataset from the storage and train itself on this subpart using supervised learning (e.g. with a gradient descent). Then, if the network has not converged it fetches another subpart of the dataset and iterate the learning part. As, for a given DLN, the subpart is of the same size, the IO time (without congestion) and learning time is constant across iterations. However, as each DLN is different (e.g. in terms of topology and meta parameters) the number of iterations is different across DLN. Therefore, according to our nomenclature this use-case fits the Uniform case: $J_i = (A, B)^{n_i}, i \in [1, m]$.

In Figure 5, we compare best effort and the FIFO list scheduling strategies which are both non-clairvoyant (they do not know in advance the number of periods) against the Hierarchical Round-robin for which the closed form of the makespan is given as follows. We are in the Uniform case: the set of jobs is $J_i = (0, (a, b)^{n_i}$. We denote by $n = \max_i n_i$, $l = |\{J_{i_0} | n_{i_0} = n\}|$ (the number of jobs of maximum $n_i$), $q = \frac{(\sum_i n_i) - l}{n-1}$ and $r = ((\sum_i n_i) - l) \mod (n-1)$. Then, the makespan of Hierarchical Round-robin $C_{\max}^{HRr}$ is:

$$C_{\max}^{HRr} = a + l \cdot b + (n - 1 - r) \cdot \max(a + b, qb) + r \cdot \max(a + b, (q + 1)b)$$

According to Theorem 2, Hierarchical Round-robin is asymptotically optimal. Moreover, the FIFO list-scheduling is a 2-approximation algorithm (Theorem 3). For this use case, we see that despite the fact that the FIFO list-scheduling is non-clairvoyant it provides a makespan very close to Hierarchical Round-robin (less than 10% slower). Concerning the best effort strategy, we see that it performs worse than FIFO list-scheduling and up to 60% slower than Hierarchical Round-robin. Indeed, in this case,

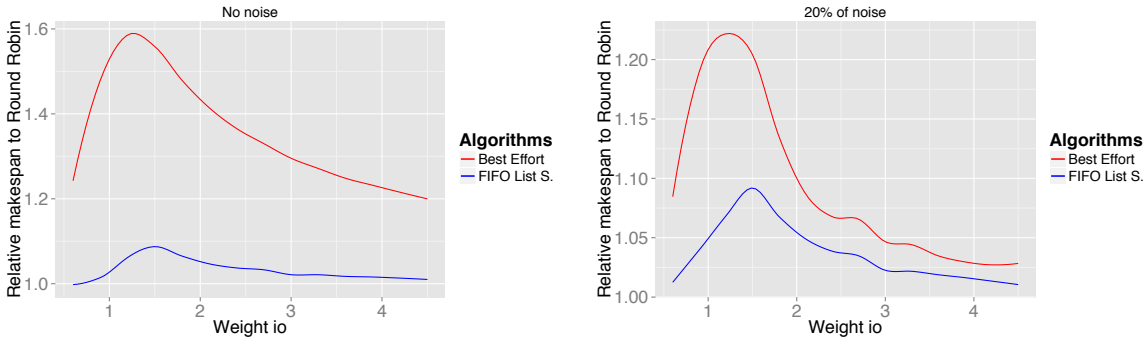



Fig. 5: Policies performance comparison of the ML use case for the makespan relative to HIERARCHICAL ROUND-ROBIN (left no noise, right 20% of uniform noise).

the access of the I/O is synchronized and the best-effort strategy maintain this synchronization and hence the I/O contention during the whole execution of the instance.

To test the case where we can have desynchronization due to uncertainty in computation or I/O execution, we have added 20% of uniform noise on these two costs. The results are presented on the right of Figure 5. In this case, we see that the noise has almost no impact on the FIFO list-scheduling strategy. For the best effort strategy, we see that it has a better performance than without noise but it is still worse than the FIFO list-scheduling. This shows that the best effort strategy does not behave well in case of high congestion of the network.

## 6 Related work

### 6.1 Related theoretical problems

The MS-HPC-IO problem may recall the classical job shop problem (see definition in [18]). In both problems jobs are composed of dependent tasks that have to be performed on specific machines. However, here, we do not have constraints on the computation machine therefore if knowledge of job shop can help to develop insight of solutions, it can not be used straightforwardly for HPC-IO. Variants of job shop and flow shop are abundantly discussed: [17,18,23,4]. Recall that flow shop is a particular case of job shop where the operation sequences do not depend on jobs.

### 6.2 State of the art in I/O management for HPC systems

We are not the first to study performance variability caused by I/O congestion. In this section we will detail some of the existing work and different approaches to understand this issue.

*Data transformation* As contention arises with large amount of data, recent studies propose application-side strategies based on I/O management and transformation. Lofstead et al. [20] study adaptive strategies to deal with I/O variability due to congestion by modifying at certain times both the number of processes sending data , and the size of the data being sent. Tessier and al. [24] focus on the locality of aggregate nodes. These nodes are compute nodes dedicated data sent by other compute nodes during the I/O phase of an application. Those nodes also have the possibility to transform the data being sent (for instance by compressing it [8]). To go further, data can even be compressed in a lossy way [7]. In-situ/intransit analysis developed in recent works [11] try to deal with file systems reaching their

limit. In the past, some workflows used to create the data and to store it on disks before analyzing it as a second step. In-situ/in-transit analysis offers to dedicate some specific nodes to the analysis and to perform it as the data is created. The hope is to reduce the load on the file systems.

We consider that all these solutions occur uphill to our problem and hence can be used conjointly.

*Software to deal with I/O movement* On the application side, the I/O congestion issue can be seen as scheduling problem [20, 29].

Work using machine learning for auto tuning and performance study [3, 16] can be applied for I/O scheduling but do not provide a global view of the I/O requirements of the application. Coupling with a platform level I/O management ensure better results.

Cross-application contention has been studied recently [13, 22, 25]. The study in [13] evaluates the performance degradation in each application program when Virtual Machines (VMs) are executing two application programs concurrently in a physical computing server. The experimental results indicate that the interference among VMs executing two HPC application programs with high memory usage and high network I/O in the physical computing server significantly degrades application performance. An earlier study in 2005 [22] cites application interference as one of the main problems facing the HPC community. While the authors propose ways of gaining performance by reducing variability, minimizing application interference is still left open. In [28], a more general study analyzes the behavior of the center wide shared Lustre parallel file system on the Jaguar supercomputer and its performance variability. One of the performance degradations seen on Jaguar was caused by concurrent applications sharing the filesystem. All these studies highlight the impact of having application interference on HPC systems, without, but they do not offer a solution. Closer to this work, online schedulers for HPC systems were developed such as Aupy et al. [12], the study by Zhou et al [30], and a solution proposed by Dorier et al [9]. In [9], the authors investigate the interference of two applications and analyze the benefits of interrupting or delaying either one in order to avoid congestion. Unfortunately their approach cannot be used for more than two applications. Another main difference with our previous work is the light-weight approach of this study where the computation is only done once. Clarisse [14] proposes mechanisms for designing and implementing cross-layer optimizations of the I/O software stack. The specific implementation of the problem considered here is a naive First Come First Served approach. They, however, provide an excellent opportunity to study our results in a real framework.

*Hardware solutions* Diminishing I/O bottleneck can also be thought at an architectural level. Previous papers [19] noticed that congestion occurs on a short period of time and the bandwidth to the storage is often underutilized. As the computation power used to increase faster than the I/O bandwidth, this observation may not hold in the future. In the meantime, delaying accesses to the system storage can smoothen the I/O request over time and tackle latency. An example of this technique is presented in Kougkas et al [15]. A dynamic I/O scheduling at the application level, using burst buffers, stages I/O and allows computations to continue uninterrupted. They design different strategies to mitigate I/O interference, including partitioning the PFS, which reduces the effective bandwidth non-linearly. Note that for now, these strategies are designed for only two applications, furthermore they are not coupled with an efficient I/O bandwidth scheduling strategy and can only work because they considered an underutilized I/O bandwidth.

## 7 Conclusion

In this report we have studied the problem of scheduling I/O access for applications that alternate computation and I/O. We have formally described the problem as scheduling bi-colored chains. Then, we have studied theoretical results. Despite the fact that the general case is NP-complete, we have provided an optimal algorithm for the Uniform case. Moreover, we have studied two classes of strategies: periodic and list scheduling ones. We have shown that any list-scheduling algorithm is a 2-approximation and

that Hierarchical Round-robin is asymptotically optimal for the periodic case. We have also studied different order for instantiating several heuristics (both periodic and list-scheduling ones).

We have experimentally tested, through simulations, the proposed approaches on realistic cases. We have shown that periodic approaches are the best ones when the relative amount of I/O is high and that the best effort strategy is the worst one. Moreover, we have studied the case where the input is not known with complete certainty but subject to noise. In this case the proposed approaches are shown to be robust. Last, we have studied the case of a distributed learning phase for deep-learning. Results show that the FIFO list-scheduling strategy is very close to the optimal one (despite being non-clairvoyant) and much better than the best effort.

In future work, we want to study several directions. The first one, concern the study of fairness. Indeed, the proposed strategies may favor some applications against others. We would like to devise algorithms that could guarantee that the worst degradation is bounded. We would also like to study the impact of release dates. In this study all the applications start at the same time, which is not realistic. When evaluating the makespan, having release dates makes little sense, however, if we want to study fairness, release dates is a parameter that we will have to take into consideration. Last, we would like to implement strategies based on what we have learned here into an I/O scheduling framework such as Clarisse. We have started a collaboration with the University of Madrid to work in that direction.

**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 Rgion Nouvelle Aquitaine 2018-1R50119 “HPC scalable ecosystem”.

## References

1. Aupy, G., Beaumont, O., Eyraud-Dubois, L.: Sizing and partitioning strategies for burst-buffers to reduce io contention. In: Parallel and Distributed Processing Symposium (IPDPS), 2019 IEEE International. IEEE (2019)
2. Aupy, G., Gainaru, A., Le Fèvre, V.: Periodic i/o scheduling for super-computers. In: International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, pp. 44–66. Springer (2017)
3. Behzad, B., Luu, H.V.T., Huchette, J., Byna, S., Aydt, R., Koziol, Q., Snir, M., et al.: Taming parallel i/o complexity with auto-tuning. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 68. ACM (2013)
4. Brucker, P., Brucker, P.: Scheduling algorithms, vol. 3. Springer (2007)
5. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of petascale i/o workloads. In: Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on, pp. 1–10. IEEE (2009)
6. Daly, J.T.: A higher order estimate of the optimum checkpoint interval for restart dumps. FGCS **22**(3), 303–312 (2006)
7. Di, S., Cappello, F.: Fast error-bounded lossy hpc data compression with sz. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 730–739. IEEE (2016)
8. Dorier, M., Antoniu, G., Cappello, F., Snir, M., Orf, L.: Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In: Cluster Computing (CLUSTER), 2012 IEEE International Conference on, pp. 155–163. IEEE (2012)
9. Dorier, M., Antoniu, G., Ross, R., Kimpe, D., Ibrahim, S.: Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, pp. 155–164. IEEE (2014)
10. Dorier, M., Ibrahim, S., Antoniu, G., Ross, R.: 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, pp. 623–634. IEEE (2014)
11. Dreher, M., Raffin, B.: A flexible framework for asynchronous in situ and in transit analytics for scientific simulations. In: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on, pp. 277–286. IEEE (2014)
12. Gainaru, A., Aupy, G., Benoit, A., Cappello, F., Robert, Y., Snir, M.: Scheduling the i/o of hpc applications under congestion. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, pp. 1013–1022. IEEE (2015)
13. Hashimoto, Y., Aida, K.: Evaluation of performance degradation in hpc applications with vm consolidation. In: Networking and Computing (ICNC), 2012 Third International Conference on, pp. 273–277. IEEE (2012)

14. Isaila, F., Carretero, J., Ross, R.: 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, pp. 346–355. IEEE (2016)
15. Kougkas, A., Dorier, M., Latham, R., Ross, R., Sun, X.H.: Leveraging burst buffer coordination to prevent i/o interference. In: e-Science (e-Science), 2016 IEEE 12th International Conference on, pp. 371–380. IEEE (2016)
16. Kumar, S., Saha, A., Vishwanath, V., Carns, P., Schmidt, J.A., Scorzelli, G., Kolla, H., Grout, R., Latham, R., Ross, R., et al.: Characterization and modeling of pidx parallel i/o for performance optimization. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 67. ACM (2013)
17. Lenstra, J., Rinnooy Kan, A., Brucker, P.: Complexity of machine scheduling problems. Ann. of Discrete Math. **1**, 343–362 (1977)
18. Leung, J., Kelly, L., Anderson, J.H.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA (2004)
19. Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., Maltzahn, C.: On the role of burst buffers in leadership-class storage systems. In: Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on, pp. 1–11. IEEE (2012)
20. Lofstead, J., Zheng, F., Liu, Q., Klasky, S., Oldfield, R., Kordenbrock, T., Schwan, K., Wolf, M.: Managing variability in the io performance of petascale storage systems. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12. IEEE Computer Society (2010)
21. Madireddy, S., Balaprakash, P., Carns, P., Latham, R., Ross, R., Snyder, S., Wild, S.: Modeling i/o performance variability using conditional variational autoencoders. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 109–113. IEEE (2018)
22. Skinner, D., Kramer, W.: Understanding the causes of performance variability in hpc workloads. In: Workload Characterization Symposium, 2005. Proceedings of the IEEE International, pp. 137–149. IEEE (2005)
23. Tanaev, V., Gordon, W., Shafransky, Y.M.: Scheduling theory. Single-stage systems, vol. 284. Springer Science & Business Media (2012)
24. Tessier, F., Malakar, P., Vishwanath, V., Jeannot, E., Isaila, F.: Topology-aware data aggregation for intensive i/o on large-scale supercomputers. In: Proceedings of the First Workshop on Optimization of Communication in HPC, pp. 73–81. IEEE Press (2016)
25. Uselton, A., Howison, M., Wright, N.J., Skinner, D., Keen, N., Shalf, J., Karavanic, K.L., Oliker, L.: Parallel i/o performance: From events to ensembles. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pp. 1–11. IEEE (2010)
26. Wikum, E.D., Llewellyn, D.C., Nemhauser, G.L.: One-machine generalized precedence constrained scheduling problems. Operations Research Letters **16**(2), 87 – 99 (1994). DOI https://doi.org/10.1016/0167-6377(94)90064-7. URL `http://www.sciencedirect.com/science/article/pii/0167637794900647`
27. Wu, G., Chen, J., Wang, J.: On scheduling two-stage jobs on multiple two-stage flowshops. arXiv preprint arXiv:1801.09089 (2018)
28. Xie, B., Chase, J., Dillow, D., Drokin, O., Klasky, S., Oral, S., Podhorszki, N.: Characterizing output bottlenecks in a supercomputer. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 8. IEEE Computer Society Press (2012)
29. Zhang, X., Davis, K., Jiang, S.: Opportunistic data-driven execution of parallel programs for efficient i/o services. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, pp. 330–341. IEEE (2012)
30. Zhou, Z., Yang, X., Zhao, D., Rich, P., Tang, W., Wang, J., Lan, Z.: I/o-aware batch scheduling for petascale computing systems. In: Cluster Computing (CLUSTER), 2015 IEEE International Conference on, pp. 254–263. IEEE (2015)