Periodic I/O scheduling for super-computers

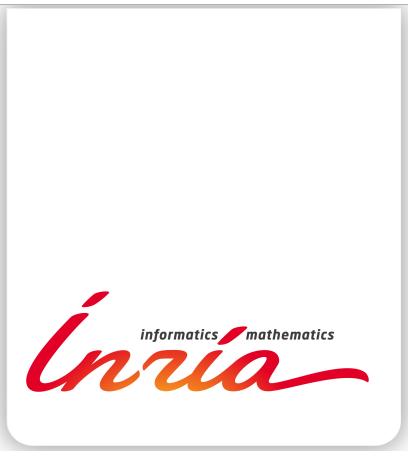
Guillaume Aupy, Ana Gainaru, Valentin Le Fèvre

Periodic I/O scheduling for super-computers

Guillaume Aupy, Ana Gainaru, Valentin Le Fèvre

RESEARCH REPORT
N° 8721
February 2017
Project-Team Tadaam

ISSN 0249-6399 ISRN INRIA/RR--8721--FR+ENG



Periodic I/O scheduling for super-computers

Guillaume Aupy*, Ana Gainaru†, Valentin Le Fèvre‡

Project-Team Tadaam

Research Report n° 8721 — February 2017 — 29 pages

Abstract: With the ever-growing need of data in HPC applications, the congestion at the I/O level becomes critical in super-computers. Architectural enhancement such as burst-buffers and pre-fetching are added to machines, but are not sufficient to prevent congestion. Recent online I/O scheduling strategies have been put in place, but they add an additional congestion point and overheads in the computation of applications.
In this work, we show how to take advantage of the periodic nature of HPC applications in order to develop efficient periodic scheduling strategies for their I/O transfers. Our strategy computes once during the job scheduling phase a pattern where it defines the I/O behavior for each application, after which the applications run independently, transferring their I/O at the specified times. Our strategy limits the amount of I/O congestion at the I/O node level and can be easily integrated into current job schedulers. We validate this model through extensive simulations and experiments by comparing it to state-of-the-art online solutions, showing that not only our scheduler has the advantage of being de-centralized and thus overcoming the overhead of online schedulers, but also that it performs better than these solutions, improving the application dilation up to 13% and the maximum system efficiency up to 18%.

Key-words: Scheduling; models; I/O; HPC; bandwidth; congestion.

* Inria & University of Bordeaux
† Mellanox Technologies, Oak Ridge, USA
‡ École Normale Supérieure de Lyon, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement périodique d'entrées/sorties pour super-ordinateurs

Résumé : Dans cet article, nous nous intéressons à des techniques de gestion d'entrées-sorties dans les super-ordinateurs. La nouveauté de ce travail est la prise en compte de certaintes caractéristiques et arguments structurels sur les applications haute performance, leur périodicité, dans la conception de nos algorithmes. Nous nous comparons à des solutions récentes et montrons un gain en efficacité système atteignant 18% et en dilation atteignant 13%.

Nous montrons comment facilement intégrer ces solutions sur des super-ordinateurs.

Mots-clés : Ordonnancement; modèles; entrées-sorties; HPC; bande passante; congestion.

1 Introduction

In the race to larger supercomputers, the most commonly used metric is is the computational power. However supercomputers are not simply computers with billions of processors. One of the reason why Sunway TaihuLight (the world fastest supercomputer as of Nov 2016 [1]), reaches 93 PetaFlops on HPL (a performance benchmark based on dense linear algebra), but struggles to reach 0.37 PetaFlop on HPCG, a recent benchmark based on actual HPC applications [10] is data movement. Nowadays, a supercomputing application creates or has to deal with TeraBytes of data. This is true in all fields, from medical research (Brain initiatives), to astrophysics (HACC [16], Enzo [5], HOMME [24]), including meteorology (CM1 [4]) and fusion plasma (GTC [13]). In 2013, Argonne upgraded its house supercomputer: moving from Intrepid (peak performance: 0.56 PFlops; peak I/O throughput: 88 GB/s) to Mira (peak performance: 10 PFlops; peak I/O throughput: 240 GB/s). While both criteria seem to have improved considerably, the reality behind is that for a given application, its I/O throughput scales linearly (or worse) with its performance, and hence, what should be noticed is a downgrade from 160 GB/PFlop to 24 GB/PFlop! On Intrepid, it was shown that I/O congestion could cause up to a 70% decrease to the I/O throughput [14].

To help with the ever growing amount of data created, architectural improvement such as burst buffers [21] have been added to the system. Work is being done to transform the data before sending it to the disks in the hope of reducing the I/O sent [11]. However, even with the current I/O footprint burst buffers are not able to completely hide congestion. Moreover, the data used is always expected to grow. Recent works [14] have started working on novel online, centralized I/O scheduling strategies at the I/O node level. However one of the risk noted on these strategies is the scalability issue caused by potentially high overheads (between 1 and 5% depending on the number of nodes used in the experiments) [14]. Moreover, it is expected this overhead to increase at larger scale since it need centralized information about all applications running in the system.

In this paper, we present a decentralized I/O scheduling strategy for supercomputers. We show how to take known HPC application behaviors (namely their periodicity) into account to derive novel static algorithms. The periodicity of HPC applications has been well observed and documented [6, 14, 12]: HPC applications alternate between computation and I/O transfer, this pattern being repeated over-time. Furthermore, fault-tolerance technique (such as periodic checkpointing [9]) also add to this periodic behavior. Using this periodicity property, we compute a static periodic scheduling strategy, which provides a way for each applications to know when they should start transferring their I/O (i) hence reducing potential bottlenecks either due to I/O congestion, and (ii) without having to consult with I/O nodes every time I/O should be done and hence adding an extra overhead. The main contributions of this paper are:

- A novel light-weight I/O algorithm that looks at optimizing both application-oriented (dilation or fairness) and platform-oriented (maximum system

efficiency) objectives;
- A set of extensive simulations and experiments that show that this algorithm performs as well or better than current state of the art heavy-weight online algorithms.

Note that the algorithm presented here is done as a proof of concept to show the efficiency of this kind of light-weight techniques. We believe our scheduler can be implemented naturally into a job scheduler and we provide experimental results backing this claim. However, this integration is beyond the scope of this paper.

The rest of the paper is organized as follows: in Section 2 we present the application model and optimization problem. In Section 3 we present our novel algorithm technique as well as a brief proof of concept for a future implementation. In Section 4 we present extensive simulations based on the model to show the performance of our algorithm compared to state of the art. We then confirm the performance on a super-computer to validate the model. We give some background and related work in Section 5. We provide concluding remarks and ideas for future research directions in Section 6.

2 Model

In this section we use the model introduced in our previous work [14] that has been verified experimentally to be consistent with the behavior of Intrepid and Mira, super-computers at Argonne.

We consider scientific applications running at the same time on a parallel platform. The applications consist of series of computations followed by I/O operations. On a super-computer, the computations are done independently because each application uses its own nodes. However, the applications are concurrently sending and receiving data during their I/O phase on a dedicated I/O network. The consequence of this I/O concurrency is congestion between an I/O node of the platform and the file storage.

2.1 Parameters

We assume that we have a parallel platform made up of N identical unit-speed nodes, composed of the same number of identical processors, each equipped with an I/O card of bandwidth b (expressed in bytes per second). We further assume a centralized I/O system with a total bandwidth B (also expressed in bytes per second). This means that the total bandwidth between the computation nodes and an I/O node is $N \cdot b$ while the bandwidth between an I/O node and the file storage is B, with usually $N \cdot b \gg B$. We have instantiated this model for the Intrepid platform on Figure 1.

We have K applications, all assigned to independent and dedicated computational resources, but competing for I/O. For each application $\text{App}^{(k)}$ we define:
- Its size: $\text{App}^{(k)}$ executes with $\beta^{(k)}$ dedicated processors;

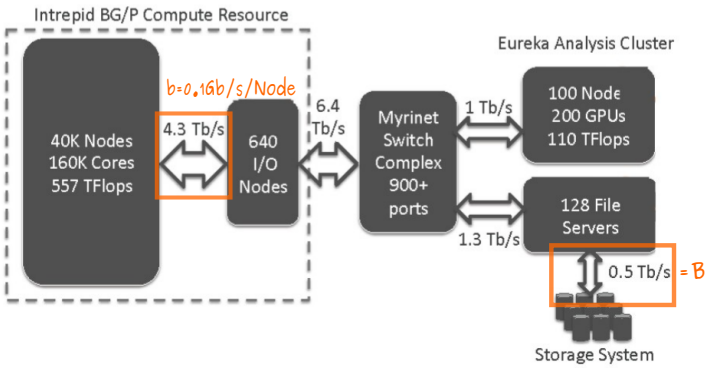


Figure 1: Model instantiation for the Intrepid platform [14].

- Its pattern: $\text{App}^{(k)}$ obeys a pattern that repeats over time. There are $n_{\text{tot}}^{(k)}$ *instances* of $\text{App}^{(k)}$ that are executed one after the other. Each instance consists of two disjoint phases: computations that takes a time $w^{(k)}$, followed by I/O transfers for a total volume $\text{vol}_{\text{io}}^{(k)}$. The next instance cannot start before I/O operations for the current instance is terminated.

We further denote by r_k the time when $\text{App}^{(k)}$ is released on the platform and d_k the time when the last instance is completed. Finally, we denote by $\gamma^{(k)}(t)$, the bandwidth used by a node on which application $\text{App}^{(k)}$ is running, at instant t.

2.2 Execution Model

As the computation resources are dedicated, we can always assume w.l.o.g that the next computation chunk starts right away after completion of the previous I/O transfers, and is executed at full (unit) speed. On the contrary, all applications compete for I/O, and congestion will likely occur. The simplest case is that of a single periodic application $\text{App}^{(k)}$ using the I/O system in dedicated mode during a time-interval of duration D. In that case, let γ be the I/O bandwidth used by each processor of $\text{App}^{(k)}$ during that time-interval. We derive the condition $\beta^{(k)}\gamma D = \text{vol}_{\text{io}}^{(k)}$ to express that the entire I/O data volume is transferred. We must also enforce the constraints that (i) $\gamma \leq b$ (output capacity of each processor); and (ii) $\beta^{(k)}\gamma \leq B$ (total capacity of I/O system). Therefore, the minimum time to perform the I/O transfers for an instance of $\text{App}^{(k)}$ is $\text{time}_{\text{io}}^{(k)} = \frac{\text{vol}_{\text{io}}^{(k)}}{\min(\beta^{(k)}b, B)}$. However, in general many applications will use the I/O system simultaneously, whose bandwidth capacity B will be shared among all these applications (see Figure 2).

This model is very flexible, and the only assumption is that at any instant, all processors assigned to a given application are assigned the same bandwidth.

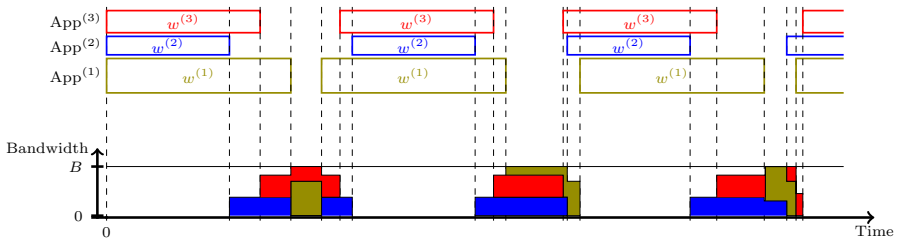


Figure 2: Scheduling the I/O of three periodic applications (top: computation, bottom: I/O).

This assumption is transparent for the I/O system and simplifies the problem statement without being restrictive. Again, in the end, the total volume of I/O transfers for an instance of $\text{App}^{(k)}$ must be $\text{vol}_{\text{io}}^{(k)}$, and at any instant, the rules of the game are simple: never exceed the individual bandwidth b of each processor ($\gamma^{(k)}(t) \leq b$ for any k and t), and never exceed the total bandwidth B of the I/O system ($\sum_{k=1}^{K} \beta^{(k)}\gamma^{(k)}(t) \leq B$ for any t).

2.3 Objectives

We now focus on the optimization objectives at hand here. We use the objectives introduced in [14].

First, the *application efficiency* achieved for each application $\text{App}^{(k)}$ at time t is defined as

$$\tilde{\rho}^{(k)}(t) = \frac{\sum_{i \leq n^{(k)}(t)} w^{(k,i)}}{t - r_k},$$

where $n^{(k)}(t) \leq n_{\text{tot}}^{(k)}$ is the number of instances of application $\text{App}^{(k)}$ that have been executed at time t, since the release of $\text{App}^{(k)}$ at time r_k. Because we execute $w^{(k,i)}$ units of computation followed by $\text{vol}_{\text{io}}^{(k,i)}$ units of I/O operations on instance $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$, we have $t - r_k \geq \sum_{i \leq n^{(k)}(t)} \left(w^{(k,i)} + \text{time}_{\text{io}}^{(k,i)} \right)$. Due to I/O congestion, $\tilde{\rho}^{(k)}$ never exceeds the optimal efficiency that can be achieved for $\text{App}^{(k)}$, namely

$$\rho^{(k)} = \frac{w^{(k)}}{w^{(k)} + \text{time}_{\text{io}}^{(k)}}$$

The two key optimization objectives, together with a rationale for each of them, are:

- SysEfficiency: where we maximize the peak performance of the platform, namely maximizing the amount of operations per time unit:

$$\text{maximize } \frac{1}{N} \sum_{k=1}^{K} \beta^{(k)} \tilde{\rho}^{(k)}(d_k). \tag{1}$$

- DILATION: where we minimize the largest slowdown imposed to each application (hence optimizing fairness across applications):

$$\text{minimize} \max_{k=1..K} \frac{\rho^{(k)}}{\tilde{\rho}^{(k)}(d_k)}. \tag{2}$$

Note that it is known that both problems are NP-complete, even in an (easier) offline setting [14].

3 Periodic scheduling strategy

In general, for an application App$^{(k)}$, $n_{\text{tot}}^{(k)}$ the number of instances of App$^{(k)}$ is very large and not polynomial in the size of the problem. For this reason, online schedule have been preferred until now. The key novelty of this paper is to introduce *periodic schedules* for the K applications. Intuitively, we are looking for a computation and I/O *pattern* of duration T that will be repeated over time (except for *initialization* and *clean up* phases), as shown on Figure 3a. In this section, we start by introducing the notion of periodic schedules and a way to compute the application efficiency differently. We then provide the algorithms that are at the core of this work.

Because there is no competition on computation (no shared resources), we can consider that a chunk of computation directly follows the end of the transfer of I/O, hence we need only to represent I/O transfers in this pattern. The bandwidth used by each application during the I/O operations is represented over time, as shown in Figure 3b. We can see that an operation can overlap with the one of the previous pattern or the next pattern, but overall, the pattern will just repeat.

To describe a pattern, we use the following notations:

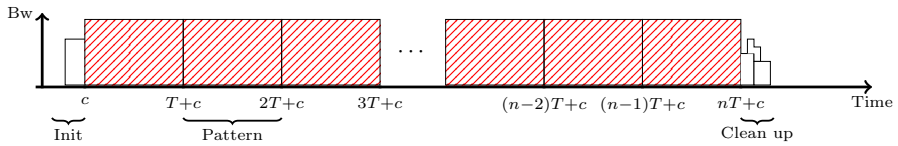
- $n_{\text{per}}^{(k)}$: the number of instances of App$^{(k)}$ during a pattern.
- $\mathcal{I}_i^{(k)}$: the i-th instance of App$^{(k)}$ during a pattern.
- $\texttt{initW}_i^{(k)}$: the time of the beginning of $\mathcal{I}_i^{(k)}$. So, $\mathcal{I}_i^{(k)}$ has a computation interval going from $\texttt{initW}_i^{(k)}$ to $\texttt{endW}_i^{(k)} = \texttt{initW}_i^{(k)} + w^{(k)} \bmod T$.
- $\texttt{initIO}_i^{(k)}$: the time when the I/O transfer from the i-th instance of App$^{(k)}$ starts (between $\texttt{endW}_i^{(k)}$ and $\texttt{initIO}_i^{(k)}$, App$^{(k)}$ is idle). Therefore, we have

$$\int_{\texttt{initIO}_i^{(k)}}^{\texttt{initW}_{(i+1)\%n_{\text{per}}^{(k)}}^{(k)}} \beta^{(k)}\gamma^{(k)}(t)dt = \text{vol}_{\text{io}}^{(k)}.$$

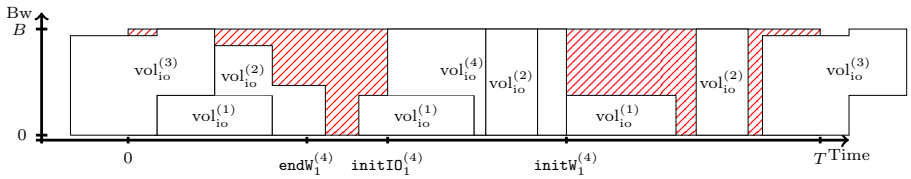
Globally, if we consider the two dates per instance $\texttt{initW}_i^{(k)}$ and $\texttt{initIO}_i^{(k)}$, that define the change between computation and I/O phases, we have a total of $S \leq \sum_{k=1}^{K} 2n_{\text{per}}^{(k)}$ distinct dates, that are called the *events* of the pattern.

We define the periodic efficiency of a pattern of size T:

$$\tilde{\rho}_{\text{per}}^{(k)} = \frac{n_{\text{per}}^{(k)} w^{(k)}}{T}. \tag{3}$$



(a) Periodic schedule (phases)



(b) Detail of I/O in a period/pattern

Figure 3: A schedule (above), and the detail of one of its regular pattern (below), where $(w^{(1)} = 3.5; \mathrm{vol}_{\mathrm{io}}^{(1)} = 240; n_{\mathrm{per}}^{(1)} = 3)$, $(w^{(2)} = 27.5; \mathrm{vol}_{\mathrm{io}}^{(2)} = 288; n_{\mathrm{per}}^{(2)} = 3)$, $(w^{(3)} = 90; \mathrm{vol}_{\mathrm{io}}^{(3)} = 350; n_{\mathrm{per}}^{(3)} = 1)$, $(w^{(4)} = 75; \mathrm{vol}_{\mathrm{io}}^{(4)} = 524; n_{\mathrm{per}}^{(4)} = 1)$.

For periodic schedules, we use it to approximate the actual efficiency achieved for each application. The rationale behind this can be seen on Figure 3. If $\mathrm{App}^{(k)}$ is released at time r_k, and the first pattern starts at time $r_k + c$, that is after an initialization phase, then the main pattern is repeated n times (until time $n \cdot T + r_k + c$), and finally $\mathrm{App}^{(k)}$ ends its execution after a clean-up phase at time $d_k = r_k + c + n \cdot T + c'$. If we assume that $n \cdot T \gg c + c'$, then $d_k - r_k \approx n \cdot T$. Then the value of the $\tilde{\rho}^{(k)}(d_k)$ for $\mathrm{App}^{(k)}$ is:

$$\begin{aligned}\tilde{\rho}^{(k)}(d_k) &= \frac{\left(n \cdot n_{\mathrm{per}}^{(k)} + \delta\right) w^{(k)}}{d_k - r_k} = \frac{\left(n \cdot n_{\mathrm{per}}^{(k)} + \delta\right) w^{(k)}}{c + n \cdot T + c'} \\ &\approx \frac{n_{\mathrm{per}}^{(k)} w^{(k)}}{T} = \tilde{\rho}_{\mathrm{per}}^{(k)}\end{aligned}$$

where δ can be 1 or 0 depending whether $\mathrm{App}^{(k)}$ was executed or not during the clean-up or init phase.

3.1 PerSched: a periodic scheduling algorithm

For details in the implementation, we refer the interested reader to the source code available at `https://github.com/vlefevre/IO-scheduling-simu`.

The difficulties of finding an efficient periodic schedule are three-fold:

- The first one is that the right pattern size has to be determined;
- The second one is that for a given pattern size, the number of instances of each application that should be included in this pattern need to be determined;

- Finally, the time constraint between two consecutive I/O transfers of a given application, due to the computation in-between makes naive scheduling strategies harder to implement.

Finding the right pattern size A solution is to find schedules with different pattern sizes between a minimum pattern size $T_{\min}$ and a maximum pattern size $T_{\max}$.

Because we want a pattern to have at least one instance of each application, we can trivially set up $T_{\min} = \max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)})$. Intuitively, the larger $T_{\max}$ is, the more possibilities we can have to find a good solution. However this also increases the complexity of the algorithm. We want to limit the number of instances of all applications in a schedule. For this reason we chose to have $T_{\max} = O(\max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)}))$. We discuss this hypothesis in Section 4, where we give better experimental intuition on finding the right value for $T_{\max}$. Experimentally we observe (Section 4, Figure 7) that $T_{\max} = 10T_{\min}$ seems to be sufficient.

We then decided on an iterative search where the pattern size increases exponentially at each iteration from $T_{\min}$ to $T_{\max}$. In particular, we use a precision ε as input and we iteratively increase the pattern size from $T_{\min}$ to $T_{\max}$ by a factor $(1 + \varepsilon)$. This allows us to have a polynomial number of iterations. The rationale behind the exponential increase is that when the pattern size gets large, we expect performance to converge to an optimal value, hence needing less the precision of a precise pattern size. Furthermore while we could try only large pattern size, it seems important to find a good small pattern size as it would simplify the scheduling step. Hence a more precise search for smaller pattern sizes. Finally, we expect the best performance to cycle with the pattern size. We verify these statements experimentally in Section 4 (Figure 6).

Determining the number of instances of each application By choosing $T_{\max} = O(\max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)}))$, we guarantee the maximum number of instances of each application that fit into a pattern is $O\left(\frac{\max_k(w^{(k)}+\text{time}_{\text{io}}^{(k)})}{\min_k(w^{(k)}+\text{time}_{\text{io}}^{(k)})}\right)$.

Instance scheduling Finally, our last item is, given a pattern of size T, how to schedule instances of applications into a periodic schedule.

To do this, we decided on a strategy where we insert instances of applications in a pattern, without modifying dates and bandwidth of already scheduled instances. Formally, we call an application schedulable:

Definition 1 (Schedulable)**.** Given an existing pattern

$$\mathcal{P} = \cup_{k=1}^{K}\left(n_{\text{per}}^{(k)}, \cup_{i=1}^{n_{\text{per}}^{(k)}}\{\texttt{initW}_i^{(k)}, \texttt{initIO}_i^{(k)}, \gamma^{(k)}()\}\right),$$

we say that an application $\text{App}^{(k)}$ is schedulable if there exists $1 \leq i \leq n_{\text{per}}^{(k)}$, such that:

$$\int_{\texttt{initW}_i^{(k)}+w^{(k)}}^{\texttt{initIO}_i^{(k)}-w^{(k)}} \min\left(\beta^{(k)}b, B - \sum_l \beta^{(l)}\gamma^{(l)}(t)\right) dt \geq \text{vol}_{\text{io}}^{(k)} \tag{4}$$

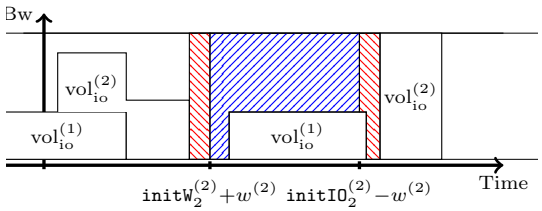


Figure 4: Description of what schedulable means: if we want to insert an instance of $\text{App}^{(2)}$, we need to check that the blue area is greater than $\text{vol}_{\text{io}}^{(2)}$, while the red area is reserved for the computation of $w^{(2)}$.

To understand the integral in Equation (4): we are checking that during the end of the computation of the i^{th} instance ($\texttt{initW}_i^{(k)} + w^{(k)}$), and the beginning of the computation of the $i + 1^{\text{th}}$ instance ($\texttt{initIO}_i^{(k)}$-$w^{(k)}$), there is enough bandwidth to perform at least a volume of I/O of $\text{vol}_{\text{io}}^{(k)}$. We represent it graphically on Figure 4.

With Definition 1, we can now explain the core idea of the instance scheduling part of our algorithm. Starting from an existing pattern, while there exist applications that are schedulable:

- Amongst the applications that are schedulable, we choose the application that has the worse Dilation. The rationale is that even though we want to increase SysEfficiency, we do it in a way that ensures that all applications are treated fairly;
- We insert the instance into an existing scheduling using a procedure Insert-In-Pattern such that (i) the first instance of each application is inserted using procedure Insert-First-Instance which minimizes the time of the I/O transfer of this new instance, (ii) the other instances are inserted just after the last inserted one.

Note that Insert-First-Instance is implemented using a water-filling algorithm [15] and Insert-In-Pattern is implemented as described in Algorithm 1. We use a different function for the first instance of each application because we do not have any previous instance to use the Insert-In-Pattern function. Thus, the basic idea would be to put them at the beginning of the pattern, but it will be more likely to create congestion if all applications are "synchronized" (for example if all the applications are the same, they will all start their I/O phase at the same time). By using Insert-First-Instance, every first instance will be at a place where the congestion for it is minimized. This creates a starting point for the subsequent instances.

Algorithm 1: INSERT-IN-PATTERN

```
1  procedure INSERT-IN-PATTERN(P, App^(k))
2  begin
3      if App^(k) has 0 instance then
4          return INSERT-FIRST-INSTANCE (P, App^(k));
5      else
6          T_min := +∞ ;
7          Let I_k^{i} be the last inserted instance of App^(k);
8          Let E_0, E_1, ..., E_{j_i} the times of the events between the end of
             I_k^{i} + w^(k) and the beginning of I_k^{(i+1) mod l_T(k)};
9          For l = 0 ... j_i − 1, let B_l be the minimum between β^(k) b and the
             available bandwidth during [E_l, E_{l+1}];
10         DataLeft = vol_io^(k);
11         l = 0;
12         sol = [];
13         while DataLeft > 0 and l < j_i do
14             if B_l > 0 then
15                 TimeAdded = min(E_{l+1} − E_l, DataLeft/B_l);
16                 DataLeft -= TimeAdded·B_l;
17                 sol = [(E_l, E_l + TimeAdded, B_l)] + sol;
18             l++;
19         if DataLeft> 0 then
20             return P
21         else
22             return P.addInstance(App^(k),sol)
```

The function `addInstance` updates the pattern with the new instance, given a list of the intervals $(\mathcal{E}_l, \mathcal{E}_{l'}, b_l)$ during which $\text{App}^{(k)}$ transfers I/O between $\mathcal{E}_l$ and $\mathcal{E}_{l'}$ using a bandwidth b_l.

Correcting the period size In Algorithm 2, the pattern sizes under trial are determined by $T_{\min}$ and ε. There is no reason why this would be the right pattern size, and one might be interested in reducing it to fit precisely the instances that are included in the solutions that we found.

In order to do so, once a periodic pattern has been computed, we try to improve the best pattern size we found in the first loop of the algorithm, by trying new pattern sizes, close to the previous best one, let us say T_{opt}. To do this, we add a second loop which now tries $1/\varepsilon$ uniformly distributed pattern sizes from T_{opt} to $T_{\text{opt}}/(1+\varepsilon)$.

With all of this in mind, we can now write PERSCHED (Algorithm 2), our algorithm to construct a periodic pattern. For all pattern sizes tried between $T_{\min}$ and $T_{\max}$, we return the pattern with maximal SYSEFFICIENCY.

Algorithm 2: Periodic Scheduling heuristic: PERSCHED

```
1  procedure PERSCHED(K', ε, {App^(k)}_{1≤k≤K})
2  begin
3      T_min ← max_k(w^(k) + time_io^(k));
4      T_max ← K' · T_min;
5      T = T_min;
6      SE ← 0;
7      T_opt ← 0;
8      P_opt ← {};
9      while T ≤ T_max do
10         P = {};
11         while exists a schedulable application do
12             A = {App^(k) | App^(k) is schedulable};
13             Let App^(k) be the element of A minimal with respect to the
                 lexicographic order (ρ^(k)/ρ̃_per^(k), w^(k)/time_io^(k));
14             P ←INSERT-IN-PATTERN(P, App^(k));
15         if SE < SYSEFFICIENCY(P) then
16             SE ← SYSEFFICIENCY(P);
17             T_opt ← T;
18             P_opt ← P
19         T ← T · (1 + ε);
20     T ← T_opt;
21     while true do
22         P = {};
23         while exists a schedulable application do
24             A = {App^(k) | App^(k) is schedulable};
25             Let App^(k) be the element of A minimal with respect to the
                 lexicographic order (ρ^(k)/ρ̃_per^(k), w^(k)/time_io^(k));
26             P ←INSERT-IN-PATTERN(P, App^(k));
27         if SYSEFFICIENCY(P) = (T_opt/T) · SE then
28             P_opt ← P;
29             T ← T − (T_opt − T_opt/(1+ε))/⌊1/ε⌋
30         else
31             return P_opt
```

We estimate SYSEFFICIENCY of a periodic pattern, by replacing $\tilde{\rho}^{(k)}(d_k)$ by $\tilde{\rho}^{(k)}_{\text{per}}$ in Equation (1)

3.2 Complexity analysis

Finally, in this section we show that our algorithm runs in reasonnable execution time. We detail theoretical results that allowed us to reduce the complexity. We want to show the following result:

Theorem 1. *Let* $n_{\max} = \left(\frac{\max_k(w^{(k)}+time_{io}^{(k)})}{\min_k(w^{(k)}+time_{io}^{(k)})}\right)$, PerSched$(K', \varepsilon, \{App^{(k)}\}_{1\leq k\leq K})$ *runs in*

$$O\left(\left(\left\lceil\frac{1}{\varepsilon}\right\rceil + \left\lceil\frac{\log K'}{\log(1+\varepsilon)}\right\rceil\right)\cdot K^2\left(n_{\max}+\log K'\right)\right).$$

Some of the complexity results are straightforward. The key results to show are:

- The complexity of the tests "`while` *exists a schedulable application*" on lines 11 and 23
- The complexity of computing $\mathcal{A}$ and finding its minimum element on line 13 and 25.
- The complexity of Insert-In-Pattern

To reduce the execution time, we proceed as follows: instead of implementing the set $\mathcal{A}$, we implement a heap $\tilde{\mathcal{A}}$ that could be summarized as

$$\{\text{App}^{(k)} | \text{App}^{(k)} \text{ is not yet known to } \textbf{not} \text{ be schedulable}\}$$

sorted following the lexicographic order: $\left(\frac{\rho^{(k)}}{\tilde{\rho}_{\text{per}}^{(k)}}, \frac{w^{(k)}}{\text{time}_{\text{io}}^{(k)}}\right)$. Hence, we replace the while loops on lines 11 and 23 by the algorithm snippet described in Algorithm 3. The idea is to avoid calling Insert-In-Pattern after each new inserted instance to know which applications are schedulable.

Algorithm 3: Schedulability snippet

```
11 Ã = ∪_k{App^(k)} (sorted by (ρ^(k)/ρ̃^(k)_per, w^(k)/time^(k)_io));
12 while Ã ≠ ∅ do
13     Let App^(k) be the minimum element of Ã;
14     Ã ← Ã \ {App^(k)};
15     Let P' = Insert-In-Pattern(P, App^(k));
16     if P' ≠ P then
17         P ← P';
18         Insert App^(k) in Ã following (ρ^(k)/ρ̃^(k)_per, w^(k)/time^(k)_io);
```

We then need to show that they are equivalent, that is:

- At all time, the minimum element of $\tilde{\mathcal{A}}$ is minimal amongst the schedulable applications with respect to the order $\left(\frac{\rho^{(k)}}{\tilde{\rho}^{(k)}_{\text{per}}}, \frac{w^{(k)}}{\text{time}^{(k)}_{\text{io}}}\right)$ (shown in Lemma 4);
- If $\tilde{\mathcal{A}} = \emptyset$ then there are no more schedulable applications (shown in Corollary 2).

To show this, it is sufficient to show that (i) at all time, $\mathcal{A} \subset \tilde{\mathcal{A}}$, and (ii) $\tilde{\mathcal{A}}$ is always sorted according to $\left(\frac{\rho^{(k)}}{\tilde{\rho}^{(k)}_{\text{per}}}, \frac{w^{(k)}}{\text{time}^{(k)}_{\text{io}}}\right)$.

Definition 2 (Compact pattern)**.** We say that a pattern

$$\mathcal{P} = \cup_{k=1}^{K} \left(n^{(k)}_{\text{per}}, \cup_{i=1}^{n^{(k)}_{\text{per}}} \{\texttt{initW}^{(k)}_i, \texttt{initIO}^{(k)}_i, \gamma^{(k)}()\} \right)$$

is compact if for all $1 \leq i < n^{(k)}_{\text{per}}$, either $\texttt{initW}^{(k)}_i + w^{(k)} = \texttt{initIO}^{(k)}_i$, or for all $t \in [\texttt{initW}^{(k)}_i, \texttt{initIO}^{(k)}_i]$, $\sum_l \beta^{(l)}\gamma^{(l)}(t) = B$.

Intuitively, this means that we can only schedule a new instance for all application $\text{App}^{(k)}$ between $\mathcal{I}^{(k)}_{n^{(k)}_{\text{per}}}$ and $\mathcal{I}^{(k)}_1$.

Lemma 1. *At any time during* PerSched*, $\mathcal{P}$ is compact.*

Proof. For each application, either we use Insert-First-Instance to insert the first instance (so $\mathcal{P}$ is compact as there is only one instance of an application at this step), either we use Insert-In-Pattern which inserts an instance just after the last inserted one, which is the definition of being compact. Hence, $\mathcal{P}$ is compact at any time during PerSched. □

Lemma 2. Insert-In-Pattern$(\mathcal{P}, App^{(k)})$ *returns $\mathcal{P}$, if and only if $App^{(k)}$ is not schedulable.*

Proof. One can easily check that Insert-In-Pattern checks the schedulability of $\text{App}^{(k)}$ only between the last inserted instance of $\text{App}^{(k)}$ and the first instance of $\text{App}^{(k)}$. Furthermore, because of the compacity of $\mathcal{P}$ (Lemma 1), this is sufficient to test the overall schedulability.

Then the test is provided by the last condition $\texttt{Dataleft} > 0$.

- If the condition is false, then the algorithm actually inserts a new instance, so it means that one more instance of $\text{App}^{(k)}$ is schedulable.
- If the condition is true, it means that we cannot insert a new instance after the last inserted one. Because $\mathcal{P}$ is compact, we cannot insert an instance at another place. So if the condition is true, we cannot add one more instance of $\text{App}^{(k)}$ in the pattern.

□

Corollary 1. *In Algorithm 3, an application $App^{(k)}$ is removed from $\tilde{\mathcal{A}}$ if and only if it is not schedulable.*

Lemma 3. *If an application is not schedulable at some step, it will not be either in the future.*

Proof. Let us suppose that $\text{App}^{(k)}$ is not schedulable at some step. In the future, new instances of other applications can be added, thus possibly increasing the total bandwidth used at each instant. The total I/O load is non-decreasing during the execution of the algorithm. Thus if for all i, we had

$$\int_{\texttt{initW}_i^{(k)}+w^{(k)}}^{\texttt{initIO}_i^{(k)}-w^{(k)}} \min\left(\beta^{(k)}b, B-\sum_l \beta^{(l)}\gamma^{(l)}(t)\right)dt < \text{vol}_{\text{io}}^{(k)},$$

then in the future, with new bandwidths used $\gamma'^{(l)}(t) > \gamma^{(l)}(t)$, we will still have that for all i,

$$\int_{\texttt{initW}_i^{(k)}+w^{(k)}}^{\texttt{initIO}_i^{(k)}-w^{(k)}} \min\left(\beta^{(k)}b, B-\sum_l \beta^{(l)}\gamma'^{(l)}(t)\right)dt < \text{vol}_{\text{io}}^{(k)}.$$

□

Corollary 2. *At all time,*

$$\mathcal{A} = \{App^{(k)} | App^{(k)} \text{ is schedulable}\} \subset \tilde{\mathcal{A}}.$$

This is a direct corollary of Corollary 1 and Lemma 3

Lemma 4. *At all time, the minimum element of $\tilde{\mathcal{A}}$ is minimal amongst the schedulable applications with respect to the order $\left(\frac{\rho^{(k)}}{\tilde{\rho}_{per}^{(k)}}, \frac{w^{(k)}}{time_{io}^{(k)}}\right)$ (but not necessarily schedulable).*

Proof. First see that $\{\text{App}^{(k)} | \text{App}^{(k)} \text{ is schedulable}\} \subset \tilde{\mathcal{A}}$.

Furthermore, initially the minimality property is true. Then the set $\tilde{\mathcal{A}}$ is modified only when a new instance of an application is added to the pattern. More specifically, only the application that was modified has its position in $\tilde{\mathcal{A}}$ modified. One can easily verify that for all other applications, their order with respect to $\left(\frac{\rho^{(k)}}{\tilde{\rho}_{\text{per}}^{(k)}}, \frac{w^{(k)}}{\text{time}_{\text{io}}^{(k)}}\right)$ has not changed, hence the set is still sorted. □

This concludes the proof that the snippet is equivalent to the while loops. With all this we are now able to show timing results for the version of Algorithm 2 that uses Algorithm 3.

Lemma 5. *The loop on line 23 of Algorithm 2 terminates in at most $\lceil 1/\varepsilon \rceil$ steps.*

Proof. The stopping criteria on line 27 checks that the number of instances did not change when reducing the pattern size. Indeed, by definition for a pattern $\mathcal{P}$,

$$\begin{aligned}\textsc{SysEfficiency}(\mathcal{P}) &= \sum_k \beta^{(k)} \tilde{\rho}^{(k)}_{\text{per}} \\ &= \frac{\sum_k \beta^{(k)} n^{(k)}_{\text{per}} w^{(k)}}{T}.\end{aligned}$$

Denote $\texttt{SE}$ the SysEfficiency reached in T_{opt} at the end of the while loop on line 11 of Algorithm 2. Let $\textsc{SysEfficiency}(\mathcal{P})$ be the SysEfficiency obtained in $T_{\text{opt}}/(1+\varepsilon)$. By definition,

$$\textsc{SysEfficiency}(\mathcal{P}) < \texttt{SE} \qquad \text{and}$$
$$\frac{T_{\text{opt}}}{1+\varepsilon}\textsc{SysEfficiency}(\mathcal{P}) < T_{\text{opt}}\texttt{SE}.$$

Necessarily, after at most $\lceil 1/\varepsilon \rceil$ iterations, Algorithm 2 exits the loop on line 23.

□

Proof of Theorem 1. There are $\lfloor m \rfloor$ pattern sizes tried where $T_{\min} \cdot (1+\varepsilon)^m = T_{\max}$ in the main "while" loop (line 9), that is

$$m = \frac{\log T_{\max} - \log T_{\min}}{\log(1+\varepsilon)} = \frac{\log K'}{\log(1+\varepsilon)}.$$

Furthermore, we have seen (Lemma 5) that there are a maximum of $\lceil 1/\varepsilon \rceil$ pattern sizes tried of the second loop (line 21).

For each pattern size tried, the cost is dominated by the complexity of Algorithm 3. Let us compute this complexity.

- The construction of $\tilde{\mathcal{A}}$ is done in $O(K \log K)$.
- In sum, each application can be inserted a maximum of $n_{\max}$ times in $\tilde{\mathcal{A}}$ (maximum number of instances in any pattern), that is the total of all insertions has a complexity of $O(K \log K n_{\max})$.

We are now interested by the complexity of the different calls to Insert-In-Pattern.

First one can see that we only call Insert-First-Instance K times, and in particular they correspond to the first K calls of Insert-In-Pattern. Indeed, we always choose to insert a new instance of the application that has the largest current slowdown. The slowdown is infinite for all applications at the beginning, until their first instance is inserted (or they are removed from $\tilde{\mathcal{A}}$) when it becomes finite, meaning that the K first insertions will be the first instance of all applications.

During the k-th call, for $1 \leq k \leq K$, there will be $n = 2(k-1)+2$ events (2 for each previously inserted instances and the two bounds on the pattern), meaning

that the complexity of INSERT-FIRST-INSTANCE will be $O(n \log n)$ (because of the sorting of the bandwidths available by non-increasing order to choose the intervals to use). So overall, the K first calls have a complexity of $O(K^2 \log K)$.

Furthermore, to understand the complexity of the remaining calls to INSERT-IN-PATTERN we are going to look at the end result. In the end there is a maximum of $n_{\max}$ instance of each applications, that is a maximum of $2n_{\max}K$ events. For all application $\text{App}^{(k)}$, for all instance $\mathcal{I}_i^{(k)}$k, $1 < i \leq n^{(k)}$, the only events considered in INSERT-IN-PATTERN when scheduling $\mathcal{I}_i^{(k)}$k were the ones between the end of $\texttt{initW}_k^{(i)} + w^{(k)}$ and $\texttt{initW}_{k+1}^{(i)}$. Indeed, since the schedule has been able to schedule $\text{vol}_{\text{io}}^{(k)}$, INSERT-IN-PATTERN will exit the while loop on line 13. Finally, one can see that the events considered for all instances of an application partition the pattern without overlapping. Furthermore, INSERT-IN-PATTERN has a linear complexity in the number of events considered. Hence a total complexity by application of $O(n_{\max}K)$. Finally, we have K applications, the overall time spent in INSERT-IN-PATTERN for inserting new instances is $O(K^2 n_{\max})$.

Hence, with the number of different pattern tried, we obtain a complexity of

$$O\left(\left(\lceil m \rceil + \left\lceil \frac{1}{\varepsilon} \right\rceil\right)\left(K^2 \log K + K^2 n_{\max}\right)\right).$$

□

Note that in practice, both K' and K are small (≈ 10), and ε is close to 0, hence making the complexity $O\left(\frac{n_{\max}}{\varepsilon}\right)$.

3.3 High-level implementation, proof of concept

We envision the implementation of this periodic scheduler to take place at two levels:

1) The job scheduler would know the applications profile (using solutions such as Omnisc'IO [12]). Using profiles it would be in charge of computing a periodic pattern every time an application enters or leaves the system.

2) Application-side I/O management strategies (such as [30, 22, 29]) then would be responsible to ensure the correct transfer of I/O at the right time by limiting the bandwidth used by nodes that transfer I/O. The start and end time for each I/O as well as the used bandwidth are described in input files.

4 Evaluation and model validation

Note that the data used for this section and the scripts to generate the figures are available at `https://github.com/vlefevre/IO-scheduling-simu`.

In this section, we (i) assess the efficiency of our algorithm by comparing it to a recent dynamic framework [14], and (ii) validate our model by comparing theoretical performance (as obtained by the simulations) to actual performance on a real system.

We perform the evaluation in three steps: first we simulate behavior of applications and input them into our model to estimate both DILATION and SYSEFFICIENCY of our algorithm (Section 4.4) and evaluate these cases on an actual machine to confirm the validity of our model. Finally, in Section 4.5 we confirm the intuitions introduced in Section 3 to determine the parameters used by PERSCHED.

4.1 Experimental Setup

The platform available for experimentation is Jupiter at Mellanox, Inc. To be able to verify our model, we use it to instantiate our platform model. Jupiter is a Dell PowerEdge R720xd/R720 32-node cluster using Intel Sandy Bridge CPUs. Each node has dual Intel Xeon 10-core CPUs running at 2.80 GHz, 25 MB of L3, 256 KB unified L2 and a separate L1 cache for data and instructions, each 32 KB in size. The system has a total of 64GB DDR3 RDIMMs running at 1.6 GHz per node. Jupiter uses Mellanox ConnectX-3 FDR 56Gb/s InfiniBand and Ethernet VPI adapters and Mellanox SwitchX SX6036 36-Port 56Gb/s FDR VPI InfiniBand switches.

We measured the different bandwidths of the machine and obtained $b = 0.01$GB/s and $B = 3$GB/s. Therefore, when 300 cores transfer at full speed (less than half of the 640 available cores), congestion occurs.

Implementation of scheduler on Jupiter We simulate the existence of such a scheduler by computing beforehand the I/O pattern for each application and feeding it as input files. The experiments require a way to control the exact moment when all applications perform I/O, use the CPU or stay idle waiting to start their I/O. For this purpose, we modified the IOR benchmark [27] to read the input files that provide the start and end time for each I/O transfer as well as the bandwidth used. Our scheduler generates one such file for each application. Each IOR instance represents one application whose I/O pattern is described in one of the generated scheduling files. The IOR benchmark is split in different sets of processes running independently on different nodes, where each set represents a different application. One separate process acts as the scheduler and receives I/O requests for all groups in IOR. Since we are interested in modeling the I/O delays due to congestion or scheduler imposed delays, the modified IOR benchmarks do not use inter-processor communications.

We made experiments on our IOR benchmark and compared the results between periodic and online schedulers as well as with the performance of the original IOR benchmark without any extra scheduler.

4.2 Applications and scenarios

In the literature, there are many examples of periodic applications. Carns et al. [6] observed with Darshan the periodicity of four different applications (MADBench2 [7], Chombo I/O benchmark [8], S3D IO [25] and HOMME [24]). Furthermore, in our previous work [14] we were able to verify the periodicity

App$^{(k)}$		$w^{(k)}$ (s)	$\text{vol}_{\text{io}}^{(k)}$ (GB)	$\beta^{(k)}$
Turbulence1	(T1)	70	128.2	32,768
Turbulence2	(T2)	1.2	235.8	4,096
AstroPhysics	(AP)	240	423.4	8,192
PlasmaPhysics	(PP)	7554	34304	32,768

Table 1: Details of each application.

Set #	T1	T2	AP	PP
1	0	**10**	0	0
2	0	**8**	**1**	0
3	0	**6**	**2**	0
4	0	**4**	**3**	0
5	0	**2**	0	**1**
6	0	**2**	**4**	0
7	**1**	**2**	0	0
8	0	0	**1**	**1**
9	0	0	**5**	0
10	**1**	0	**1**	0

Table 2: Number of applications of each type launched at the same time for each experiment scenario.

of gyrokinetic toroidal code (GTC) [13], Enzo [5], HACC application [16] and CM1 [4].

Unfortunately, few documents give the actual values for $w^{(k)}$, $\text{vol}_{\text{io}}^{(k)}$ and $\beta^{(k)}$. Liu et al. [21] provide different periodic patterns of four scientific applications: PlasmaPhysics, Turbulence1, Astrophysics and Turbulence2. They were also the top four write-intensive jobs run on Intrepid in 2011. We chose the most I/O intensive patterns for all applications (as they are the most likely to create I/O congestion). We present these results in Table 1. Note that to scale those values to our system, we divided the number of processors $\beta^{(k)}$ by 64, hence increasing $w^{(k)}$ by 64. The I/O volume stays constant.

To compare our strategy, we tried all possible combinations of those applications such that the number of nodes used equals 640. That is a total of ten different scenarios that we report in Table 2.

4.3 Baseline and evaluation of existing degradation

We ran all scenarios on Jupiter without any additional scheduler. In all tested scenarios congestion occurred and decreased the visible bandwidth used by each applications as well as significantly increased the total execution time. We present in Table 3 the average I/O bandwidth slowdown due to congestion for

Set #	Application	BW slowdown	SysEfficiency
1	Turbulence 2	65.72%	0.064561
2	Turbulence 2 AstroPhysics	63.93% 38.12%	0.250105
3	Turbulence 2 AstroPhysics	56.92% 30.21%	0.439038
4	Turbulence 2 AstroPhysics	34.9% 24.92%	0.610826
6	Turbulence 2 AstroPhysics	34.67% 52.06%	0.621977
10	Turbulence 1 AstroPhysics	11.79% 21.08%	0.98547

Table 3: Bandwidth slowdown, performance and application slowdown for each set of experiments

the most representative scenarios together with the corresponding values for SysEfficiency. Depending on the IO transfers per computation ratio of each application as well as how the transfers of multiple applications overlap, the slowdown in the perceived bandwidth ranges between 25% to 65%.

Interestingly, set 1 presents the worst degradation. This scenario is running concurrently ten times the same application, which means that the I/O for all applications are executed almost at the same time (depending on the small differences in CPU execution time between nodes). This scenario could correspond to coordinated checkpoints for an application running on the entire system. The degradation in the perceived bandwidth can be as high as 65% which considerably increases the time to save a checkpoint. The use of I/O schedulers can decrease this cost, making the entire process more efficient.

4.4 Comparison to online algorithms

In this subsection, we present the results obtained by running PerSched and the online heuristics from our previous work [14]. Because in [14] we had different heuristics to optimize either Dilation or SysEfficiency, in this work, the Dilation and SysEfficiency presented are the best reached by *any* of those heuristics. This means that *there are no online solution able to reach them both at the same time*! We show that even in this scenario, our algorithm outperforms these heuristics *for both optimization problems*!

PerSched takes as input a list of applications, as well as the parameters, presented in Section 3, $K' = \frac{T_{\max}}{T_{\min}}$, ε. All scenarios were tested with $K' = 10$ and $\varepsilon = 0.01$.

Simulation results We present in Table 4 all evaluation results. The results obtained by running Algorithm 2 are called PerSched. To go further in our

evaluation, we also look for the best DILATION obtainable with our pattern (we do so by changing line 15 of PERSCHED). We call this result *min* DILATION in Table 4. This allows us to estimate how far the DILATION that we obtain is from what we can do. Furthermore, we can compute an upper bound to SYSEFFICIENCY by replacing $\tilde{\rho}^{(k)}$ by $\rho^{(k)}$ in Equation (1):

$$\text{Upper bound} = \frac{1}{N}\sum_{k=1}^{K}\frac{\beta^{(k)}w^{(k)}}{w^{(k)}+\text{time}_{\text{io}}^{(k)}}. \tag{5}$$

Set	Min DILATION	Upper bound SYSEFF	PERSCHED		Online	
			DILATION	SYSEFF	DILATION	SYSEFF
1	1.777	0.172	1.896	0.0973	2.091	0.0825
2	1.422	0.334	1.429	0.290	1.658	0.271
3	1.079	0.495	1.087	0.480	1.291	0.442
4	1.014	0.656	1.014	0.647	1.029	0.640
5	1.010	0.816	1.024	0.815	1.039	0.810
6	1.005	0.818	1.005	0.814	1.035	0.761
7	1.007	0.827	1.007	0.824	1.012	0.818
8	1.005	0.977	1.005	0.976	1.005	0.976
9	1.000	0.979	1.000	0.979	1.004	0.978
10	1.009	0.988	1.009	0.986	1.015	0.985

Table 4: Best DILATION and SYSEFFICIENCY for our periodic heuristic and online heuristics.

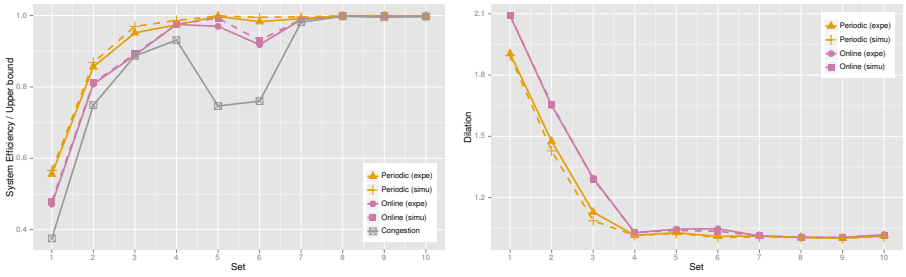
The first noticeable result is that PERSCHED almost always outperfoms (when it does not, matches) both the DILATION and SYSEFFICIENCY attainable by the online scheduling algorithms! This is particularly impressive as these objectives are not obtained by the same online algorithms (hence conjointly), contrarily to the PERSCHED result.

While the gain is minimal (from 0 to 3%, except SYSEFFICIENCY increased by 7% for case 4) when little congestion occurs (cases 4 to 10), the gain is between 9% and 16% for DILATION and between 7% and 18% for SYSEFFICIENCY when congestion occurs (cases 1, 2, 3)!

The value of ε has been chosen so that the computation stays short. It seems to be a good compromise as the results are good and the execution times vary from 4 ms (case 10) to 1.8s (case 5) using a Intel Core I7-6700Q. Note that the algorithm is easily parallelizable, as each iteration of the loop is independent. Thus it may be worth considering a smaller value of ε, but there will be no big improvement on the results.

Model validation through experimental evaluation We used the modified IOR benchmark to reproduce the behavior of applications running on HPC

systems and analyze the benefits of I/O schedulers. We made experiments on the 640 cores of the Jupiter system. Additionally to the results from both periodic and online heuristics, we present the performance of the system with no additional I/O scheduler.



(a) SysEfficiency/ Upper bound SysEfficiency

(b) Dilation

Figure 5: Performance for both experimental evaluation and theoretical (simulated) results. The performance estimated by our model is accurate within 3.8% for periodic schedules and 2.3% for online schedules.

Figure 5 shows the SysEfficiency (normalized using the upper bound in Table 4) and Dilation when using the periodic scheduler in comparison with the online scheduler. For the system efficiency the upper limit and the results when applications are running without any scheduler are also shown. As observed in the previous section, the periodic scheduler gives better or similar results to the best solutions that can be returned by the online ones, in some cases increasing the system performance by 18% and the dilation by 13%. When we compare to the current strategy on Jupiter, the SysEfficiency reach 48%! In addition, the periodic scheduler has the benefit of not requiring a global view of the execution of the applications at every moment of time (by opposition to the online scheduler).

Finally, a key information from those results is the precision of our model introduced in Section 2. The theoretical results (based on the model) are within 3% of the experimental results!

This observation is key in launching more thorough evaluation via extensive simulations and is critical in the experimentation of novel periodic scheduling strategies.

4.5 Discussion on finding the best pattern size

The core of our algorithm is a search of the best pattern size via an exponential growth of the pattern size until $T_{\max}$. As stated in Section 3, the intuition of the exponential growth is that the larger the pattern size, the less needed the precision for the pattern size as it might be easier to fit many instances of each

Set	n_{inst}	$n_{\max}$	Set	n_{inst}	$n_{\max}$
1	11	1.00	6	353	35.2
2	25	35.2	7	81	10.2
3	33	35.2	8	251	31.5
4	247	35.2	9	9	1.00
5	1086	1110	10	28	3.47

Table 5: Maximum number of instances (n_{inst}) per application, ratio between longest and shortest application ($n_{\max}$) in the solution returned by PerSched.

application. On the contrary, we expect that for small pattern sizes finding the right one might be a precision job.

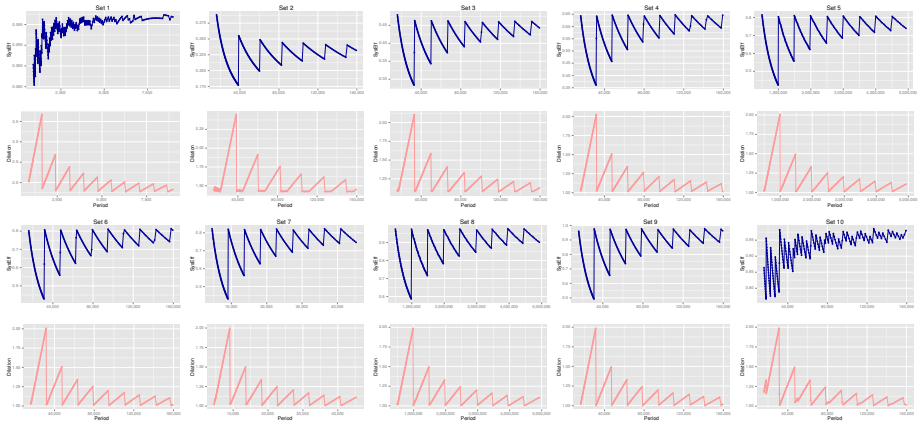


Figure 6: Evolution of SysEfficiency and Dilation when the pattern size increases.

We verify this experimentally and plot on Figure 6 the SysEfficiency and Dilation found by our algorithm as a function of the pattern size T.

Finally, the last information to determine to tweak PerSched is the value of $T_{\max}$. Remember that we denote $K' = T_{\max}/T_{\min}$.

To be able to get an estimate of the pattern size returned by PerSched, we provide in Table 5 (i) the maximum number of instances n_{inst} of any application, and (ii) the ratio $n_{\max} = \frac{\max_k \left(w^{(k)} + \text{time}_{\text{io}}^{(k)}\right)}{\min_k \left(w^{(k)} + \text{time}_{\text{io}}^{(k)}\right)}$. Together along with the fact that the Dilation (Table 4) is always below 2 they give a rough idea of K' ($\approx \frac{n_{\text{inst}}}{n_{\max}}$). It is sometimes close to 1, meaning that a small value of K' can be sufficient, but choosing $K' \approx 10$ is necessary in the general case.

We then want to verify the cost of under-estimating $T_{\max}$. For this evaluation all runs were done up to $K' = 100$ with $\varepsilon = 0.01$. Denote SysEfficiency(K') (resp. Dilation(K')) the maximum SysEfficiency (resp. corresponding Di-

LATION) obtained when running PERSCHED with K'. We plot their normalized version that is:

$$\frac{\text{SYSEFFICIENCY}(K')}{\text{SYSEFFICIENCY}(100)} \left(\text{resp. } \frac{\text{DILATION}(K')}{\text{DILATION}(100)} \right)$$

on Figure 7. The main noticeable information is that the convergence is very

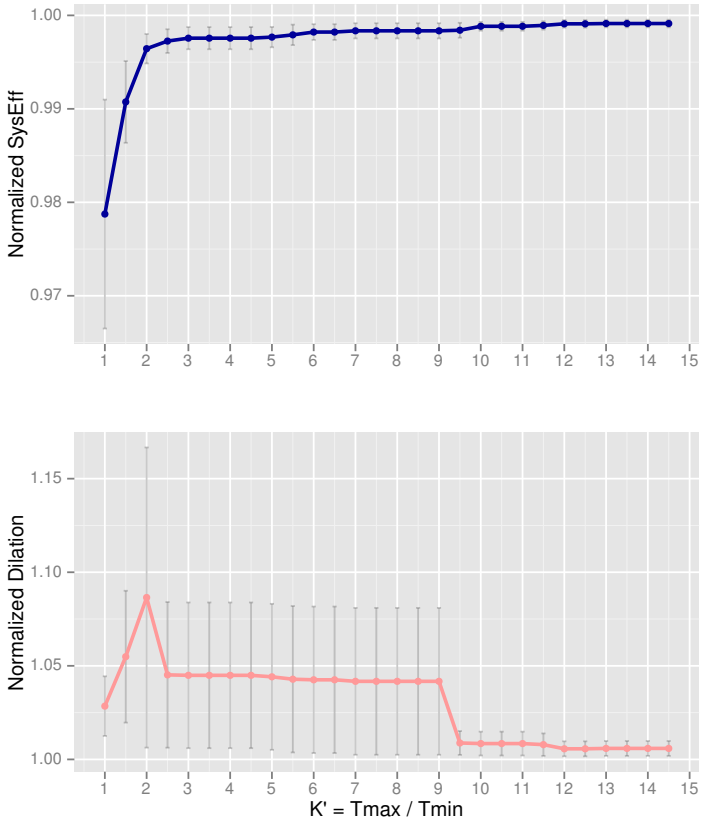


Figure 7: Normalized system efficiency and dilation obtained by Algorithm 2 averaged on all 10 sets as a function of K' (with Standard Error bars).

fast: when $K' = 3$, the average SYSEFFICIENCY is within 0.3% of SYSEFFICIENCY(100), but the corresponding average DILATION is 5% higher than DILATION(100). If we go to $K' = 10$ then we have a SYSEFFICIENCY of 0.1% of SYSEFFICIENCY(100) and a DILATION within 1% of DILATION(100)! Hence validating that choosing $K' = 10$ is sufficient.

5 Related Work

Performance variability due to resource sharing can significantly detract from the suitability of a given architecture for a workload as well as from the overall performance realized by parallel workloads [28]. Over the last decade there have been studies to analyze the sources of performance degradation and several solutions have been proposed. In this section, we first detail some of the existing work that copes with I/O congestion and then we present some of the theoretical literature that is similar to our PERIODIC problem.

The storage I/O stack of current HPC systems has been increasingly identified as a performance bottleneck. Significant improvements in both hardware and software need to be addressed to overcome oncoming scalability challenges. The study in [18] argues for making data staging coordination driven by generic cross-layer mechanisms that enable global optimizations by enforcing local decisions at node granularity at individual stack layers.

While many other studies suggest that I/O congestion is one of the main problems for future scale platforms [3, 23], few papers focus on finding a solution at the platform level. Some paper consider application-side I/O management and transformation (using aggregate nodes, compression etc) [30, 22, 29]. We consider those work to be orthogonal to our work and able to work jointly. Recently, numerous works focus on using machine learning for auto tuning and performance studies [2, 20]. However these solution also work at the application level for IO-scheduling and do not have a global view of the I/O requirements of the system and they need to be supported by a platform level I/O management for better results.

Some paper consider the use of burst buffers to reduce I/O congestion by delaying accesses to the file storage, as they found that congestion occurs on a short period of time and the bandwidth to the storage system is often underutilized [21]. However, the computation power tends to increase faster than the I/O bandwidth, which may cause the bandwidth to be saturated more often and thus decreasing the efficiency of burst buffers. [19] presents a dynamic I/O scheduling at the application level using burst buffers to stage I/O and to allow computations to continue uninterrupted. They design different strategies to mitigate I/O interference, including partitioning the PFS, which reduces the effective bandwidth non-linearly. However, the strategies are basically designed for only 2 applications and their heuristics does not take into account the characteristics of the applications to better optimize the scheduling.

The study from [26] offers ways of isolating the performance experienced by applications of one operating system from variations in the I/O request stream characteristics of applications of other operating systems. While their solution cannot be applied to HPC systems, the study offers a way of controlling the coarse grain allocation of disk time to the different operating system instances as well as determining the fine-grain interleaving of requests from the corresponding operating systems to the storage system.

Closer to this work, online schedulers for HPC systems were developed such as our previous work [14], the study by Zhou et al [31], and a solution proposed

by Dorier et al [11]. In [11], 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.

Our previous study [14] is more general by offering a range of options to schedule each I/O performed by an application. Similarly, the work from [31] also utilizes a global job scheduler to mitigate I/O congestion by monitoring and controlling jobs' I/O operations on the fly. Unlike online solutions, this paper focuses on a decentralized approach where the scheduler is integrated into the job scheduler and computes ahead of time, thus overcoming the need to monitor the I/O traffic of each application at every moment of time.

As a scheduling problem, our problem is somewhat close to the cyclic scheduling problem (we refer to Hanen and Munier [17] for a survey), namely there are given a set of activities with time dependency between consecutive tasks stored in a DAG that should be executed on p processors. The main difference is that in cyclic scheduling there is no consideration of a constant time between the end of the previous instance and the next instance.

6 Conclusion

Performance variation due to resource sharing in HPC systems is a reality and I/O congestion is currently one of the main causes of degradation. Current storage systems are unable to keep up with the amount of data handled by all applications running on an HPC system, either during their computation or when taking checkpoints. In this document we have presented a novel I/O scheduling technique that offers a decentralized solution for minimizing the congestion due to application interference. Our method takes advantage of the periodic nature of HPC applications by allowing the job scheduler to pre-define each application's I/O behavior for their entire execution. Recent studies [12] have shown that HPC applications have predictable I/O patterns even when they are not completely periodic, thus we believe our solution is general enough to easily include the large majority of HPC applications.

We conducted simulations for different scenarios and made experiments to validate our results. Decentralized solutions are able to improve both total system efficiency and application dilation compared to dynamic state-of-the-art schedulers. Moreover, they do not require a constant daemon capable of monitoring the state of all applications, nor do they require a change in the current I/O stack. One particularly interesting result is for scenario 1 with 10 identical periodic behaviors (such as what can be observed with periodic checkpointing for fault-tolerance). In this case the periodic scheduler shows a 30% improvement in SysEfficiency. Thus, system wide applications taking global checkpoints could benefit from such a strategy.

Future work: we believe this work is the initialization of a new set of techniques to deal with the I/O requirements of HPC system. In particular, by showing the efficiency of the periodic technique on simple pattern, we expect to open a door to multiple extensions. We give here some examples that we will consider in the future. The next natural directions is to take more complicated periodic shapes for applications (an instance could be composed of sub-instances) as well as different point of entry inside the job scheduler (multiple IO nodes). This would be modifying the Insert-In-Pattern procedure and we expect that this should work well as well. Another future step would be to study how variability in the compute or I/O volumes impact a periodic schedule or the impact of non periodic applications. Finally we plan to model burst buffers and to show how to use them conjointly with periodic schedules.

Our method is used for minimizing the congestion caused by concurrent I/O accesses. However, the methodology and concepts are general and can be applied to any resource sharing problem. We will continue to investigate the causes for performance degradation in HPC applications and adapt our findings to each case.

Acknowledgement

Part of this work was done when Guillaume Aupy and Valentin Le Fèvre were at Vanderbilt University. The authors would like to thank Anne Benoit and Yves Robert for helpful discussions.

References

[1] T. 500. top supercomputer list. `https://www.top500.org/lists/2016/11/`, 2016.

[2] Behzad et al. Taming parallel I/O complexity with auto-tuning. In *Proceedings of SC13*, 2013.

[3] R. Biswas, M. Aftosmis, C. Kiris, and B.-W. Shen. Petascale computing: Impact on future NASA missions. *Petascale Computing: Architectures and Algorithms*, pages 29–46, 2007.

[4] G. H. Bryan and J. M. Fritsch. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review*, 130(12), 2002.

[5] G. L. Bryan et al. Enzo: An adaptive mesh refinement code for astrophysics. *arXiv:1307.2265*, 2013.

[6] Carns et al. 24/7 characterization of petascale I/O workloads. In *Proceedings of CLUSTER09*, pages 1–10. IEEE, 2009.

[7] J. Carter, J. Borrill, and L. Oliker. Performance characteristics of a cosmology package on leading HPC architectures. In *HiPC*, pages 176–188. Springer, 2005.

[8] P. Colella et al. Chombo infrastructure for adaptive mesh refinement. `https://seesar.lbl.gov/ANAG/chombo/`, 2005.

[9] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *FGCS*, 22(3), 2004.

[10] J. Dongarra and M. A. Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312, 2013.

[11] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. Calciom: Mitigating I/O interference in HPC systems through cross-application coordination. In *Proceedings of IPDPS*, 2014.

[12] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. Omnisc'io: a grammar-based approach to spatial and temporal i/o patterns prediction. In *SC*, pages 623–634. IEEE Press, 2014.

[13] S. Ethier, M. Adams, J. Carter, and L. Oliker. Petascale parallelization of the gyrokinetic toroidal code. *VECPAR*, 2012.

[14] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. Scheduling the i/o of hpc applications under congestion. In *IPDPS*, pages 1013–1022. IEEE, 2015.

[15] R. G. Gallager. *Information theory and reliable communication*, volume 2. Springer, 1968.

[16] S. Habib et al. The universe at extreme scale: multi-petaflop sky simulation on the BG/Q. In *Proceedings of SC12*, page 4. IEEE Computer Society, 2012.

[17] C. Hanen and A. Munier. *Cyclic scheduling on parallel processors: an overview*. Citeseer, 1993.

[18] F. Isaila and J. Carretero. Making the case for data staging coordination and control for parallel applications. In *Workshop on Exascale MPI at Supercomputing Conference*, 2015.

[19] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun. Leveraging Burst Buffer Coordination to Prevent I/O Interference. In *IEEE International Conference on eScience*. IEEE, 2016.

[20] S. Kumar et al. Characterization and modeling of pidx parallel I/O for performance optimization. In *SC*. ACM, 2013.

[21] N. Liu et al. On the role of burst buffers in leadership-class storage systems. In *MSST/SNAPI*, 2012.

[22] J. Lofstead et al. Managing variability in the IO performance of petascale storage systems. In *SC*. IEEECS, 2010.

[23] J. Lofstead and R. Ross. Insights for exascale IO APIs from building a petascale IO API. In *Proceedings of SC13*, page 87. ACM, 2013.

[24] R. Nair and H. Tufo. Petascale atmospheric general circulation models. In *Journal of Physics: Conference Series*, volume 78, page 012078. IOP Publishing, 2007.

[25] Sankaran et al. Direct numerical simulations of turbulent lean premixed combustion. In *Journal of Physics: conference series*, volume 46, page 38. IOP Publishing, 2006.

[26] S. R. Seelam and P. J. Teller. Virtual i/o scheduler: A scheduler of schedulers for performance virtualization. In *Proceedings VEE*, pages 105–115. ACM, 2007.

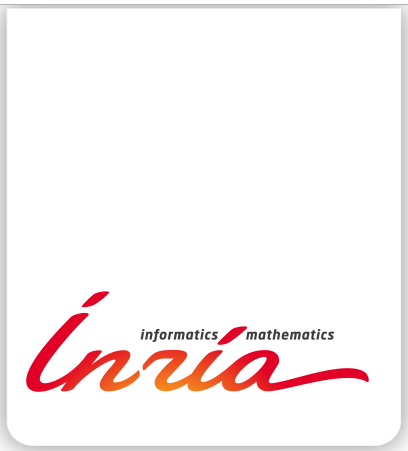
[27] H. Shan and J. Shalf. Using IOR to analyze the I/O performance for HPC platforms. *Cray User Group*, 2007.

[28] D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. *IEEE Workload Characterization Symposium*, pages 137–149, 2005.

[29] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila. Topology-aware data aggregation for intensive i/o on large-scale supercomputers. In *Proceedings of the First Workshop on Optimization of Communication in HPC*, pages 73–81. IEEE Press, 2016.

[30] X. Zhang, K. Davis, and S. Jiang. Opportunistic data-driven execution of parallel programs for efficient I/O services. In *Proceedings of IPDPS*, pages 330–341. IEEE, 2012.

[31] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan. I/o-aware batch scheduling for petascale computing systems. In *2015 IEEE International Conference on Cluster Computing*, pages 254–263, Sept 2015.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399