Periodic I/O scheduling for super-computers

Guillaume Aupy[1], Ana Gainaru[2], and Valentin Le Fèvre[13]

[1] Inria & University of Bordeaux, Talence, France
[2] Vanderbilt University, Nashville TN, USA
[3] École Normale Supérieure de Lyon, France

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.
Specifically, we show that not only our scheduler has the advantage of being de-centralized, thus overcoming the overhead of online schedulers, but we also show that on Mira one can expect an average dilation improvement of 22% with an average throughput improvement of 32%! Finally, we show that one can expect those improvements to get better in the next generation of platforms where the compute - I/O bandwidth imbalance increases.

1 Introduction

Nowadays, supercomputing applications create or have to deal with TeraBytes of data. This is true in all fields: as example LIGO (gravitational wave detection) generates 1500TB/year [22], the Large Hadron Collider generates 15PB/year, light source projects deal with 300TB of data per day and climate modeling applications are expected to have to deal with 100EB of data [17]. According to experts "Very few large scale applications of practical importance are not data intensive" (Alok Choudhary, Apr 2012).

Management of I/O operations is critical at scale. However, observations on the Intrepid machine at Argonne National Lab show that I/O transfer can be slowed down up to 70% due to congestion [14]. In 2013, Argonne upgraded its house supercomputer: moving from Intrepid (Peak performance: 0.56 PFlop/s; peak I/O throughput: 88 GB/s) to Mira (Peak performance: 10 PFlop/s; peak

I/O throughput: 240 GB/s). In 2018, the new machine at Argonne, Aurora, is expected to have a Peak performance of 450 PFlops/s and a peak I/O throughput of 1 TB/s. While both criteria seem to continuously improve 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 (Intrepid) to 24 GB/PFlop (Mira) and finally 2.2 GB/PFlop (Aurora)!

With this in mind, to be able to scale, conception of new algorithms has to change paradigm: going from a compute-centric model to a data-centric model.

To help with the ever growing amount of data created, architectural improvement such as burst buffers [23] 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.

Many recent HPC studies have observed independent patterns in the I/O behavior of HPC applications. The periodicity of HPC applications has been well observed and documented [7,14,12]: HPC applications alternate between computation and I/O transfer, this pattern being repeated over-time. Furthermore, fault-tolerance techniques (such as periodic checkpointing [10]) also add to this periodic behavior. Carns et al. [7] observed with Darshan the periodicity of four different applications (MADBench2 [8], Chombo I/O benchmark [9], S3D IO [27] and HOMME [26]). Furthermore, in our previous work [14] we were able to verify the periodicity of gyrokinetic toroidal code (GTC) [13], Enzo [6], HACC application [15] and CM1 [5].

Recently, Hu et al. [18] summed up the four key characteristics of HPC applications observed in the literature:

1. *Periodicity*: Applications alternate between compute phases and I/O phases. Furthermore they do so in a periodic fashion: a regular pattern of computation - I/O is repeated over time.
2. *Burstiness*: In addition to the periodicity observed, sometimes, short I/O bursts occur.
3. *Synchronization*: I/O accesses of an application are performed in a synchronized way between the different parallel processes.
4. *Repeatability*: The same jobs are often run many times with only different input, hence the compute-I/O pattern of an application can be predicted

before it is executed.
The key idea in this project is to take into account those known structural behaviors of HPC applications and to include them in scheduling strategies.

Using this periodicity property, we compute a static periodic scheduling strategy, which provides a way for each application 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 these 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. For the purpose of this paper the applications are already scheduled on the system and are able to receive information about their I/O scheduling. The goal of our I/O scheduler is to eliminate congestion points caused by application interference while keeping the overhead seen by all applications to the minimum. Computing a full I/O schedule over all iterations of all applications is not realistic at today's scale. The process would be too expensive both in time and space. Our scheduler overcomes this by computing a period of I/O scheduling that includes different number of iterations for each application.

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 con-

currently 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, each equipped with an I/O card of bandwidth b (expressed in bytes per second). We further assume having "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.

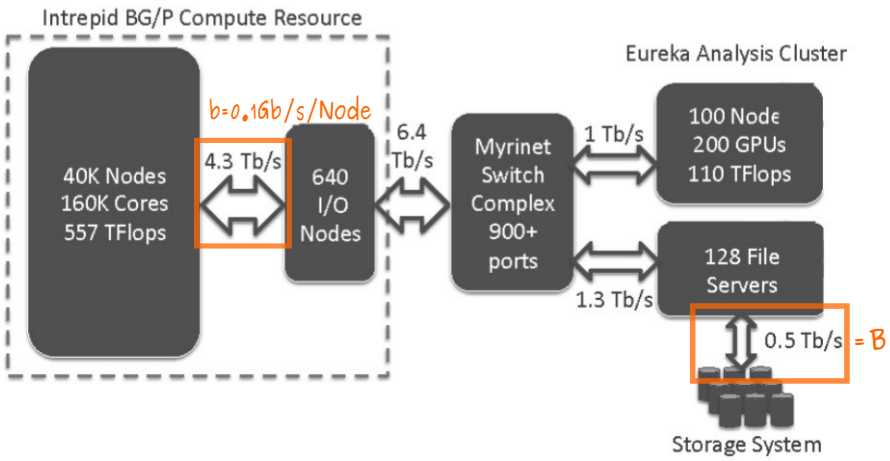


Figure 1: Model instantiation for the Intrepid platform [14].

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 nodes;
- 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 take 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

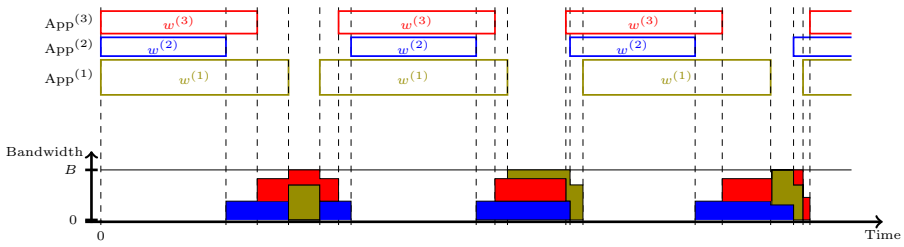


Figure 2: Scheduling the I/O of three periodic applications (top: computation, bottom: I/O).

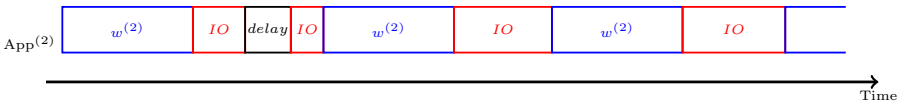


Figure 3: Application 2 execution view

bandwidth used by a node on which application $\text{App}^{(k)}$ is running, at instant t. For simplicity we assume just one I/O transfer in each loop. However, our model can be extended to work with multiple I/O patterns as long as these are periodic in nature or as long as they are known in advance.

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). Scheduling application I/O will guarantee that the I/O network will not be loaded with more than its designed capacity. Figure 2 presents the view of the machine when 3 applications are sharing the I/O system. This translates at the application level to delays inserted before I/O bursts (see Figure 3 for application 2's point of view).

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

sumption 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):

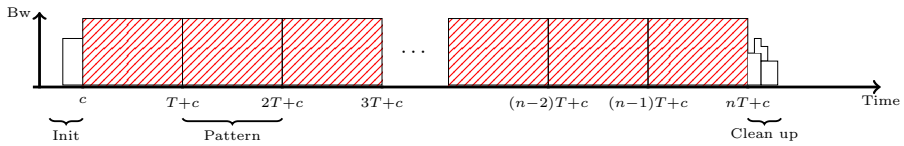
$$\text{minimize } \max_{k=1..K} \frac{\rho^{(k)}}{\tilde{\rho}^{(k)}(d_k)}. \tag{2}$$

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

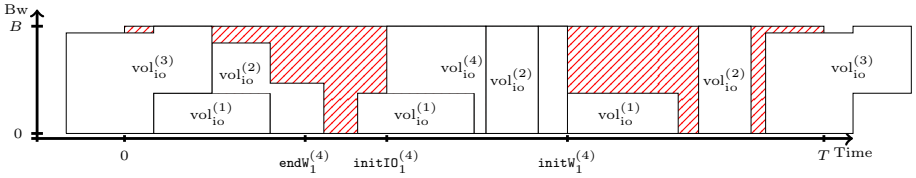
3 Periodic scheduling strategy

In general, for an application $\text{App}^{(k)}$, $n_{\text{tot}}^{(k)}$ the number of instances of $\text{App}^{(k)}$ 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 4a. In this section, we start by introducing the notion of periodic schedule 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 I/O transfer, 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 4b. 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.



(a) Periodic schedule (phases)



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

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

To describe a pattern, we use the following notations:

- $n_{\text{per}}^{(k)}$: the number of instances of $\text{App}^{(k)}$ during a pattern.
- $\mathcal{I}_i^{(k)}$: the i-th instance of $\text{App}^{(k)}$ 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

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

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:

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

For periodic schedules, we use it to approximate the actual efficiency achieved for each application. The rationale behind this can be seen on Figure 4. If 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 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 App$^{(k)}$ is:

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

where δ can be 1 or 0 depending whether 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 (see the companion report [1]) 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 sizes, 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 the companion report [1].

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* $\mathcal{P} = \cup_{k=1}^{K}\left(n_{per}^{(k)}, \cup_{i=1}^{n_{per}^{(k)}}\{\mathit{initW}_i^{(k)}, \mathit{initIO}_i^{(k)}, \gamma^{(k)}()\}\right)$*, we say that an application* $App^{(k)}$ *is schedulable if there exists* $1 \leq i \leq n_{per}^{(k)}$*, such that:*

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

To understand 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 ($\texttt{initIO}_i^{(k)}$-$w^{(k)}$: this will represent the beginning

of computation of the $i+1^{\text{th}}$ instance after the insertion of the new one, but currently it is just some time before the I/O transfer of the i^{th} instance) , 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 5.

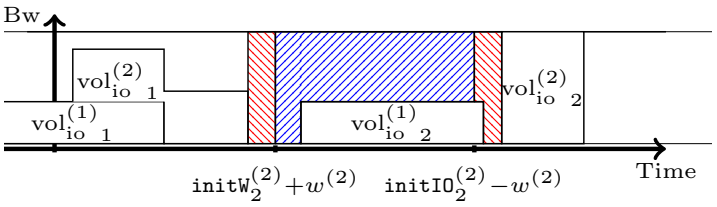


Figure 5: Graphical description of Definition 1: to insert an instance of $\text{App}^{(2)}$, we need to check that the blue area is greater than $\text{vol}_{\text{io}}^{(2)}$ with the bandwidth constraint. The red area is off limit for I/O as it would be used for computations.

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 so that it minimizes its I/O transfer time, (ii) the other instances are inserted just after the last inserted one.

With all of this in mind, we can now write PerSched (Algorithm 1), 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. For space concerns, we present here a simplified version of the real PerSched algorithm used in the simulations. You can find the minor improvement in the companion report [1].

3.2 Complexity analysis

Due to lack of space, we only give the complexity of our algorithm, the proof is in the companion report [1].

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*

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

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

Algorithm 1: 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     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)

3.3 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 application profiles (using solutions such as Omnisc'IO [12]). Using the 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 [33,24,32]) then would be responsible to ensure the correct I/O transfer 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, (i) we assess the efficiency of our algorithm by comparing it to a recent dynamic framework [14], and (ii) we 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. Once the model is validated, we perform extensive simulations.

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 for how long they use the CPU or stay idle waiting to start their I/O in addition to the amount of I/O they are writing to the disk. For this purpose, we modified the IOR benchmark [30] 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. 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. Our modified version of the benchmark reads the I/O scheduling file and adapts the bandwidth used for I/O transfers for each application as well as delaying the beginning of I/O transfers accordingly.

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. [7] observed with Darshan the periodicity of four different applications (MAD-

Bench2 [8], Chombo I/O benchmark [9], S3D IO [27] and HOMME [26]). Furthermore, in our previous work [14] we were able to verify the periodicity of gyrokinetic toroidal code (GTC) [13], Enzo [6], HACC application [15] and CM1 [5].

Unfortunately, few documents give the actual values for $w^{(k)}$, $\text{vol}_{\text{io}}^{(k)}$ and $\beta^{(k)}$. Liu et al. [23] 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 nodes $\beta^{(k)}$ by 64, hence increasing $w^{(k)}$ by 64. The I/O volume stays constant.

$\text{App}^{(k)}$	$w^{(k)}$ (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.

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 the most representative scenarios together with the corresponding values for SysEfficiency. Depending on the I/O 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 consid-

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

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

erably 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 simultaneously these heuristics *for both optimization objectives*!

The results presented in [14] represent the state of the art in what can be achieved with online schedulers. Other solutions show comparable results, with [34] presenting similar algorithms but focusing on dilation and [11] having the extra limitation of allowing the scheduling of only two applications.

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 1 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):

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

The first noticeable result is that PerSched almost always outperforms (when it does not, matches) both the Dilation and SysEfficiency attainable

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

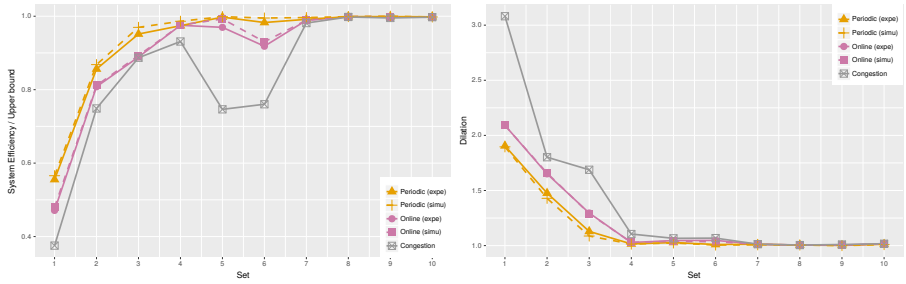
Table 4: Best Dilation and SysEfficiency for our periodic heuristic and online heuristics.

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 6) 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 we expect 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. Figure 6 shows the SysEfficiency (normalized using the upper bound in Table 4) and Dilation when using the periodic scheduler in comparison with the online scheduler. 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).



(a) SysEfficiency/ Upper bound SysEfficiency (b) Dilation

Figure 6: 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.

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.

Synthetic applications The previous experiments showed that our model can be used to simulate real life machines[4]. In this next step, we now rely on synthetic applications and simulation to test extensively the efficiency of our solution.

We considered two platforms (Intrepid and Mira) to run the simulations with concrete values of bandwidths (B, b) and number of nodes (N). The values are reported in Table 5.

Platform	B (GB/s)	b (GB/s)	N	GFlops/node
Intrepid	64	0.0125	40,960	2.87
Mira	240	0.03125	49,152	11.18

Table 5: Bandwidth and number of nodes of each platform used for simulations.

The parameters of the synthetic applications are generated as followed:

- $w^{(k)}$ is chosen uniformly at random between 2 and 7500 seconds for Intrepid (and between 0.5 and 1875s for Mira whose nodes are about 4 times faster than Intrepid's nodes),
- the volume of I/O data $\text{vol}_{\text{io}}^{(k)}$ is chosen uniformly at random between 100 GB and 35 TB.

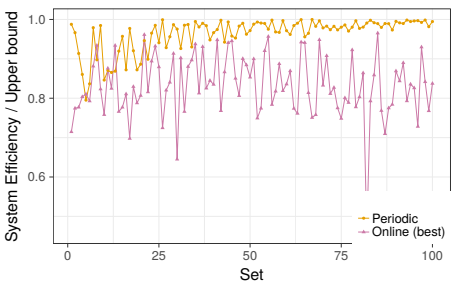
[4] Note that in our previous work [14] we already showed that this model was also fitting Intrepid and Mira

These values where based on the applications we previously studied.

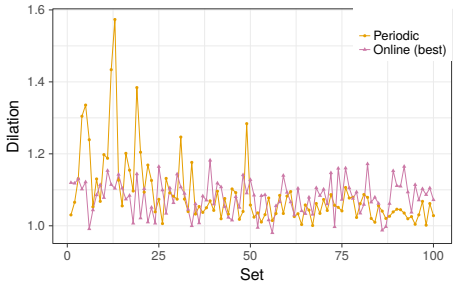
We generate the different sets of applications using the following method: let n be the number of unused nodes. At the beginning we set $n = N$.

1. Draw uniformly at random an integer number x between 1 and $\max(1, \frac{n}{4096} - 1)$ (to ensure there are at least two applications).
2. Add to the set an application $\text{App}^{(k)}$ with parameters $w^{(k)}$ and $\text{vol}_{\text{io}}^{(k)}$ set as previously detailed and $\beta^{(k)} = 4096x$.
3. $n \leftarrow n - 4096x$.
4. Go to step 1 if $n > 0$.

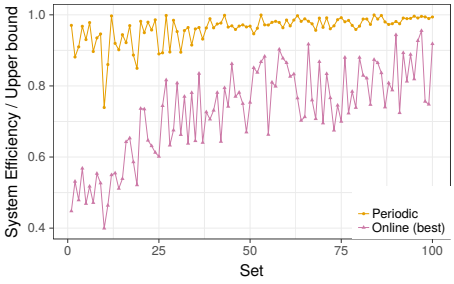
We then generated 100 sets for Intrepid (using a total of 40,960 nodes) and 100 sets for Mira (using a total of 49,152 nodes) on which we run the online algorithms (either maximizing the system efficiency or minimizing the dilation) and PerSched. The results are presented on Figures 7a and 7b for simulations using the Intrepid settings and Figures 7c and 7d for simulations using the Mira settings.



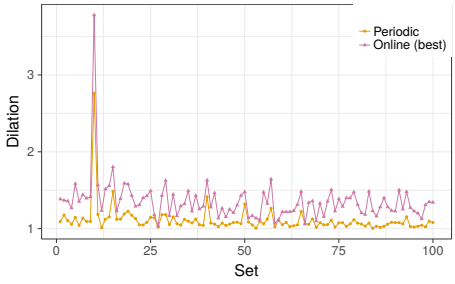
(a) Intrepid - SysEfficiency/ Upper bound SysEfficiency



(b) Intrepid - Dilation



(c) Mira - SysEfficiency/ Upper bound SysEfficiency



(d) Mira - Dilation

Figure 7: Comparison between online heuristics and PerSched on synthetic applications.

We can see that overall, our algorithm increases the system efficiency in al-

most every case. On average the system efficiency is improved by 16% on intrepid (32% on Mira) with peaks up to 116%! On Intrepid the dilation has overall similar values (an average of 0.6% degradation over the best online algorithm, with variation between 11% improvement and 42% degradation). However on Mira in addition to the improvement in system efficiency, PerSched improves on average by 22% the dilation!

The main difference between Intrepid and Mira is the ratio *compute* over *I/O bandwidth*, that is the speed at which data is created/used over the speed at which data is transfered. This ratio increases a lot (and hence incurring more I/O congestion) on Mira. Hence we expect our algorithm to be a lot more efficient on systems where congestion is even more critical.

These two experiments show two things: (i) our algorithm improves a lot the system efficiency compared to the online algorithms, without degrading too much the dilation and (ii) our algorithm is expected to scale extremely well, that is when the computing power increases faster than the bandwidth of the platform, as we can see from the results on Mira.

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 [31]. 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 [19] 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 [4,25], few papers focus on finding a solution at the platform level. Some papers consider application-side I/O management and transformation (using aggregate nodes, compression etc) [33,24,32]. 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 [3,21]. However these solution also work at the application level, 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 papers 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 [23]. Note that because the computation power increases faster than the

I/O bandwidth, this assumption may not hold in the future and the bandwidth may tend to be saturated more often and thus decreasing the efficiency of burst buffers. [20] 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. For now, these strategies are designed for only two applications.

The study from [28] 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 [34], 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 [34] 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 [16] for a survey) and periodic scheduling problems [29,2]. Namely there are given a set of activities with time dependency between consecutive tasks stored in a DAG that should be executed on N nodes. 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. More specifically, if an instance of an application has been delayed, the next instance of the same application is not delayed by the same time. With our model this could be interpreted as not overlapping I/O and computation.

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 by 32% and application dilation by 22% simultaneously 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 points of entry inside the job scheduler (multiple I/O 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 This work was supported in part by the ANR Dash project. Part of this work was done when Guillaume Aupy and Valentin Le Fèvre were in Vanderbilt University. The authors would like to thank Anne Benoit and Yves Robert for helpful discussions.

References

1. G. Aupy, A. Gainaru, and V. Le Fèvre. Periodic I/O scheduling for supercomputers. Research Report 9037, Inria Bordeaux Sud-Ouest, 2017.

2. S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 280–288. IEEE, 1995.
3. Behzad et al. Taming parallel I/O complexity with auto-tuning. In *Proceedings of SC13*, 2013.
4. 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.
5. G. H. Bryan and J. M. Fritsch. A benchmark simulation for moist nonhydrostatic numerical models. *Monthly Weather Review*, 130(12), 2002.
6. G. L. Bryan et al. Enzo: An adaptive mesh refinement code for astrophysics. *arXiv:1307.2265*, 2013.
7. Carns et al. 24/7 characterization of petascale I/O workloads. In *Proceedings of CLUSTER09*, pages 1–10. IEEE, 2009.
8. J. Carter, J. Borrill, and L. Oliker. Performance characteristics of a cosmology package on leading HPC architectures. In *HiPC*, pages 176–188. Springer, 2005.
9. P. Colella et al. Chombo infrastructure for adaptive mesh refinement. `https://seesar.lbl.gov/ANAG/chombo/`, 2005.
10. J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *FGCS*, 22(3), 2004.
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. 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.
16. C. Hanen and A. Munier. *Cyclic scheduling on parallel processors: an overview*. Citeseer, 1993.
17. B. Harrod. Big data and scientific discovery, 2014.
18. W. Hu, G.-m. Liu, Q. Li, Y.-h. Jiang, and G.-l. Cai. Storage wall for exascale supercomputing. *Journal of Zhejiang University-SCIENCE*, 2016:10–25, 2016.
19. 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.
20. 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.
21. S. Kumar et al. Characterization and modeling of pidx parallel I/O for performance optimization. In *SC*. ACM, 2013.
22. A. Lazzarini. Advanced ligo data & computing, 2003.
23. N. Liu et al. On the role of burst buffers in leadership-class storage systems. In *MSST/SNAPI*, 2012.
24. J. Lofstead et al. Managing variability in the IO performance of petascale storage systems. In *SC*. IEEECS, 2010.

25. J. Lofstead and R. Ross. Insights for exascale IO APIs from building a petascale IO API. In *Proceedings of SC13*, page 87. ACM, 2013.
26. R. Nair and H. Tufo. Petascale atmospheric general circulation models. In *Journal of Physics: Conference Series*, volume 78, page 012078. IOP Publishing, 2007.
27. Sankaran et al. Direct numerical simulations of turbulent lean premixed combustion. In *Journal of Physics: conference series*, volume 46, page 38. IOP Publishing, 2006.
28. 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.
29. P. Serafini and W. Ukovich. A mathematical model for periodic scheduling problems. *SIAM Journal on Discrete Mathematics*, 2(4):550–581, 1989.
30. H. Shan and J. Shalf. Using IOR to analyze the I/O performance for HPC platforms. *Cray User Group*, 2007.
31. D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. *IEEE Workload Characterization Symposium*, pages 137–149, 2005.
32. 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.
33. 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.
34. 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.