Scheduling bi-colored chains

Nicolas Vidal

Scheduling Bi-colored Chains

Nicolas Vidal*

December 5, 2018

1 Introduction

Until now, the performance of a supercomputer was mainly measured by its computational power. However, as platforms grow larger and the amount of data involved increases, we encounter new issues. On large scale platform, I/O movement is critical. and recent benchmarks started to measure data movement. Algorithm design is shifting focus from raw computational power to handle the bottleneck due to data management.

To tackle this problem, some approach aim at reducing the amount of data by compressing or preprocessing it. The solution we explore adopts a very different point of view and can be complementary to these approaches. We aim at managing the I/O data in the system.

The current I/O behavior of HPC machines is "best effort". This strategy does not anticipate congestion, creating slowdown in the performance of applications even though additional enhancements, burst-buffers are added to I/O nodes.

Observations show that some HPC applications periodically alternate between (i) operations (computations, local data-accesses) executed on the compute nodes, and (ii) I/O transfers of data and this behavior can be predicted before-hand. Taking this structural argument, along with HPC-specific applications facts (there are in general very few applications running concurrently on a machine, and the applications run for many iterations with similar behavior) the goal is to design new algorithms for I/O scheduling. The novelty of this class of algorithms for I/O management is that we intend them to be computed statically.

To design a static algorithm, we emphasize on the periodic behavior of most applications. Scheduling the I/O volume of the different applications is repeated over time. This is critical since often the number of application runs is very high.

In the following report, we try do develop a formal background for periodic scheduling approach. First, we define a model, bi-colored chain schedul-

*ENS de Lyon - Inria BSO

ing,then we go through related results existing in the literature and explore the complexity of this problem variants. Finally, to match the HPC context, we study periodical sub-problems.

2 Model

Our aim is to schedule different applications on an HPC platform with a focus on limiting the congestion. Consider that a given application alternates computation phases and I/O phases. In a HPC platforms, there are specifically dedicated nodes for I/O. Computation and data handling are managed by different nodes. We can therefore model an application as a chain of dependent tasks, alternatively being computation and I/O operations. We call this problem bi-colored chain scheduling.

2.1 Bi-colored chains

For m two-stage jobs $J_1, ..., J_m$ to be processed on machines A and B, verifying the properties below :

- each job consists of a chain of alternating A-operations and B-operations;
- There are no precedence constraints among different jobs;
- Two machines, A and B, of bandwidth δ_a and δ_b can run in parallel and can process respectively the A-operations and the B-operations of the job.
- An A-operation (resp. B-operation) cannot start unless the B-operation (resp. A-operation) immediately preceding it on the chain is completed;
- Preemption is not allowed.

We can remark that when all $n_i = 1$, this problem is a 2-Flowshop [18].

The intuition beyond this model is that A-tasks represent computation and B-tasks data management. (δ_a is the computational power of machine A, and δ_b is the I/O performance)

In addition, as we focus on the I/O restrictions, we can consider that we can compute an arbitrary number of rigid jobs on A and focusing solely on machine B.

Our aim is mainly to minimize the overall execution time of the applications.

Notations We write chain job J_i as $J_i = \prod_{j=1}^{n_i}(a_{i,j}, b_{i,j})$ where A-type task $a_{i,j}$ and B-type task $b_{i,j}$ must be completed before a-type $a_{i,k}$ and b-type $b_{i,k}$ for all $k \geqslant j$.

$a_{i,j}$ models the j^{th} compute phase of application i and $b_{i,j}$ its j^{th} I/O phase.

We write n_i the length (aka number of A-tasks and B-tasks pairs) of job J_i, $\alpha_{i,j}$ (resp. $\beta_{i,j}$) the starting time of task $a_{i,k}$ (resp. $b_{i,j}$).

2.2 Single processor scheduling with latencies

In the case, where machine A has no limitation ($\delta_A = \infty$), the problem is the same as single processor-scheduling with latency (or minimum delay [17]) on a single machine. The problem is the following: Given a set of chains applications, and a single machine P.

- each job consists of a chain of tasks $b_{i,j}$ to be performed on P.
- each task $b_{i,j}$ is followed by a latency $a_{i,j}$, the time between the end of $b_{i,j}$ and the start of $w_{i,j+1}$ cannot be lower than $a_{i,j}$
- There are no precedence constraints between different jobs
- Preemption is not allowed

$J_i = \prod_{j}^{n_i}(b_{i,j}, a_{i,j})$

From any instance of the bi-colored chain problem, we can construct an equivalent instance of single processor with latencies.

To do so, we identify B-operations (from bi-colored chains) with tasks and A-operations with latencies and reverse the order of the chains obtained.

This approach of the problem has been discussed in [17] and is easier to study as it focus only on one processor. In the following, we study this form of the problem.

3 Problem presentation

3.1 Objectives

Given a set of jobs with latencies: $J_i = \prod_{j}^{n_i}(b_{i,j}, a_{i,j})$

Definition 1. *In our context, a schedule $\mathcal{S}$ is a function that assigns a start time to each B-task such that:*

- $\mathcal{S}(b_{i,j}) + b_{i,j} + a_{i,j} \leqslant b_{i,k}$ *for all i and j<k.*
- $\forall i_1, i_2, j_1, j_2,$ *either* $\beta_{i_1,j_1} + b_{i_1,j_1} \leqslant \beta_{i_2,j_2}$ *or* $\beta_{i_2,j_2} + b_{i_2,j_2} \leqslant \beta_{i_1,j_1}$

In other words, $\mathcal{S}$ respects precedence and latency constraint and two B-tasks cannot be run simultaneously.

Notations. *We write $\beta_{i,j} = \mathcal{S}(b_{i,j})$ the starting time of task $b_{i,k}$ and $\alpha_{i,j} = \mathcal{S}(b_{i,j}) + b_{i,j}$ its end of execution (start of latency period).*

A *scheduling problem* consist to find an schedule $\mathcal{S}$ that optimizes a given objective.

Makespan We first define the *makespan* of a schedule as the total execution time.

Definition 2 (Makespan)**.** $MS(\mathcal{S}) = \max_i(\alpha_{i,n_i} + a_{i,n_i})$

Definition 3 (Makespan problem)**.** *Given a set of job with latencies, $J_i = \prod_j^{n_i}(b_{i,j}, a_{i,j})$ Is there an algorithm that provides an schedule $\mathcal{S}$ such that $MS(\mathcal{S})$ is minimal ?*

This objective is the "easiest" one, and the first classically used in scheduling problems.

Whereas the first objective focus on the global performance of the platform, in real life, different actors can run applications at the same time. We might also consider fairness as an additional objective. Dilation is a user-oriented objective which focus on relative progress rates of applications.

Dilation Given a schedule $\mathcal{S}$, the dilation is the largest slowdown imposed to each application.

Definition 4 (Dilation)**.** $dil(\mathcal{S}) = \max_{i=1\ldots m} \frac{\sum_k (a_{i,k}+b_{i,k})}{alpha_{i,n_i}+a_{i,n_i}}$

Definition 5 (Dilation problem)**.** *Problem: Given a set of job with latencies, $J_i = \prod_j^{n_i}(b_{i,j}, a_{i,j})$ Is there an algorithm that provides an schedule $\mathcal{S}$ such that $dil(\mathcal{S})$ is minimal ?*

4 Complexity and Algorithms

We have previously seen that schedule to reduce I/O congestion can be model by a 2-stage bi-colored chain scheduling problem. This problem is known to be NP-hard. In this part, we present some restrictions of this problem and ways to solve them.

4.1 Chains of length 1

To begin with, we consider the sub problem where all chains contain only one two-stage element. In this particular case, bi-colored chains scheduling is a classic two stages flow-shop problem.

To lighten the notation, $b_{i,j}$ can become b_j and we relabel the j to match the scheduling order (ie i<j if and only if b_i is executed before b_j).

As there are no precedence constraints, all tasks b can be run whenever convenient on machine B. Hence the following result:

Lemma 4.1 ([17],[18]). *For problem $J_i = (b_i, a_i)$, $C_{max} = \sum_{j \leqslant i} b_j + a_i$ and an optimal schedule is reach by greedily executing the largest a_i first (Johnson order).*

4.2 Chains of length at least 2

In this part, we discuss the difficulty of finding an optimal schedule with chains of equal length.

In order to do so define the problem duplicated subset sum as follow.

Definition 6 (Duplicated subset sum). *Given an integer set $\mathcal{A} = \{x_1, x_2, ..., x_t\}$ and a integer T. Are there disjoints subset S_1 and S_2 such that:*

- $S_1, S_2 \subseteq \mathcal{A}$
- $S_1 \cap S_2 = \emptyset$
- $\sum_{x_i \in S_1} x_i + 2 \sum_{x_j \in S_2} x_j = T$

Theorem 4.2. *Duplicated subset sum is NP-complete.*

Proof. Reduction from 3-sat. The classical reduction from 3-sat to subset sum as defined in [14] can be adapted for duplicated subset-sum. See Appendix A.

□

Theorem 4.3. *2-chains is NP-complete.*

Proof. Let $\mathcal{A} = x_1, x_2, ..., x_t$, T be an instance of duplicated subset sum. Consider that $2\Sigma x_i > T$

Build an instance of 2-chains:

- $J_i = (x_i, 1)^2$ for $i = 1...t$
- $J_{t+1} = J_{t+2} = (T+1, T+1)^2$

The idea is to build an optimal solution of the 2-chains problem by scheduling the jobs J_{t+1} and J_{t+2}. Then fill the lateness at the end of the schedule (of length T+1) with as many tasks as possible.

□

These two proofs can be generalized for any integer k.

Corollary 4.3.1. *k-subset sum is NP complete.*

Theorem 4.4. *k-chains is NP-complete.*

4.3 All tasks of all chains are identical :

In this section, we consider the case where all tasks are identical but spread into different chains. Minimizing the makespan of this sub-problem can be done in polynomial time.

Consider a set of m Jobs with the form $J_i = (b, a)^{n_i}$ sorted by non-increasing n_i.

We want to find a schedule $\mathcal{S}$ that minimize the makespan $MS(\mathcal{S})$. Given a sequence, we write b_i^t, the tth occurrence of task b_i

Algorithm 1 Hierarchical Round-robin

```
1: procedure HRR(J_i = (b, a)^{n_i})
2:     Sort jobs by non-increasing n_i
3:     Initialize n_0 blocks containing b_0
4:     while ∃J_i of length n_0 do
5:         Schedule one of its task within each block
6:     i ← 0
7:     while ∃J_j remaining, n_j < n_0 its length do
8:         for t = 0, t < n_j − 1 do
9:             Insert b_j^t at the end of block i + t mod n-0-1
10:        i ← i + n_j mod n_0
```

Theorem 4.5. H*ierarchical Round-robin is optimal for* $J_i = (b, a)^{n_i}$

The proof follows the steps below:

1. We define of a cost function C such that $\forall \mathcal{A},\ MS_{\mathcal{A}} \geqslant C(\mathcal{A})$
2. $\forall \mathcal{A}$ algorithm, we show $C_{min}(\mathcal{A}_{opt}) \geqslant C(HRr)$
3. We conclude by $MS_{HRr} = C(HRr)$

In the following, we consider jobs to be sorted by non increasing n_i.

Lemma 4.6. *There is an optimal schedule which starts with* J_0.

Proof. Consider an optimal schedule, b_i its first task. If $i \neq 0$, then consider the first b_0^t such that there is no b_i between b_0^t and b_0^{t+1} Such a task exists because $n_i \leqslant n_0$. Write $b_i^{\tilde{k}}$ the last occurrence of b_i before b_0^k As all tasks are identical, we can permute $b_i^{\tilde{k}}$ and b_0^k for k<t without increasing the makespan. □

We introduce the notion of block :

Definition 7. *Given a schedule S, the block B_t^S is the subsequence S starting from b_0^t (included) until b_0^{t+1}. $B_{n_i}^S$ the subsequence starting from b_0^t until the end of the schedule. Its length $|B_t^S|$ is its number of tasks. Define its cost: $C(B_{n_0}^S) = |B_{n_0}^S|b + a$ and $C(B_t^S) = max(a + b, |B_t^S|b)$ if $t \neq n_0$*

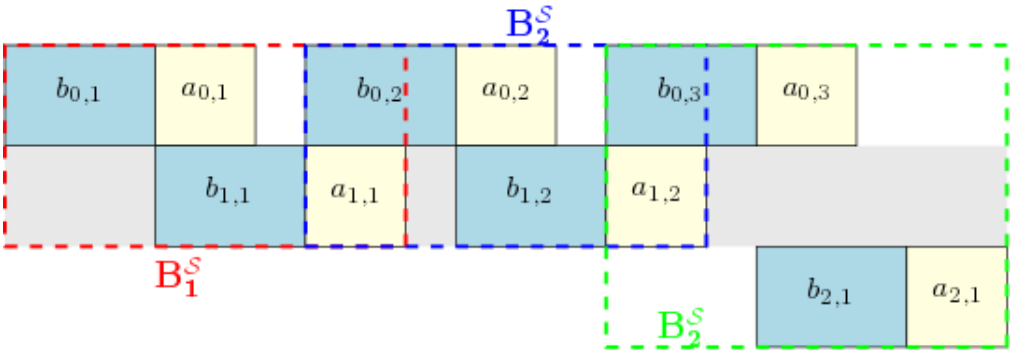

Figure 1: Example of a schedule $\mathcal{S}$
Each line of the figure represent the execution of a job $(J_1 = (b, a)^3, J_2 = (b, a)^2, J_3 = (b, a)^1$ The B-tasks are in blue, A-tasks in yellow.
Blocks are delimited by dashed boxes.

Proposition 1. *$B_t \mapsto C(B_t)$ is an increasing function of $|B_t|$*

Definition 8. *Given a schedule S, define $C(S) = \sum_i C(B_i^{\mathcal{S}})$*

As we work given a fixed schedule, we write abusively B_t for B_t^S.

Proposition 2. *For any $\mathcal{S}$, $MS_{\mathcal{S}} \geqslant C(\mathcal{S})$.*

Proof. Write $I_{\mathcal{S}}$ the total idle time of the processor during the full execution of $\mathcal{S}$, $I_{\mathcal{S}}^{J_0}$ the idle time that occurs immediately before a task of J_0
$MS_{\mathcal{S}} = \sum_{i,t} b_i^t + I_{\mathcal{S}}$
with $I_{\mathcal{S}} \leqslant I_{\mathcal{S}}^{J_0}$
By definition of blocks $I_{\mathcal{S}}^{J_0} = \sum_t \left(C(B_t^{\mathcal{S}}) - |B_t^{\mathcal{S}}|.b\right)$ Hence the expected inequality. □

Definition 9. *For $k \in \{1, 2..., n_0\}$ define $q_k = \left\lfloor \frac{N-n_0-k}{n_0-1} \right\rfloor$, $r_k = (N - n_0 - k)$ mod $(n_0 - 1)$. Let $\mathcal{S}_{min}$ be a schedule such that its first r_k blocks have size $q_k + 1$, the $n_0 - 1 - r_k$ following have size q_k, and the last has size k. $C_{min}(k) = C(\mathcal{S}_{min})$*

Proposition 3. *$\forall \mathcal{S}$, $C(\mathcal{S}) \geqslant C_{min}(|B_n^{\mathcal{S}}|)$*

Proof. $\mathcal{S}$ an arbitrary schedule. To lighten a little the notation $q = q_{|B^{\mathcal{S}}_{n_0}|}$, $r = r_{|B^{\mathcal{S}}_{n_0}|}$, By recurrence on the smallest size p of a $\mathcal{S}$ interior block. $p = \min_{t<n_O} \left|B^{\mathcal{S}}_t\right|$.

Initialization: If p=q, If there is $n_0 - 1 - r_{|B^{\mathcal{S}}_{n_0}|}$ blocks of size q, then $C(\mathcal{S}) = C_{min}(|B^{\mathcal{S}}_{n_0}|)$ by identity. Assume that if there is at most k blocks of size q, then $C(\mathcal{S}) \geqslant C_{min}(|B^{\mathcal{S}}_{n_0}|)$. If there is k+1 blocks of size q, then there exists B_i and B_j two blocks such that $|B_i| = q$, $|B_j| = p_{max}$ is maximal. Create blocks $\tilde{B}_i$ and $\tilde{B}_j$ by moving task b, $b \in B_j,\ b \notin B_i$, from B_j to B_i.

- if $(a+b) \geqslant p_{max}b$ then $C(B_j) + C(B_i) - C(\tilde{B}_j) + C(\tilde{B}_i) = 0$
- if $p_{max}b \geqslant (a+b) \geqslant (p_{max}-1)b$ then $C(B_j)+C(B_i)-C(\tilde{B}_j)+C(\tilde{B}_i) = p_{max}b - (a+b)$
- if $p_{max}-1)b \geqslant (a+b) \geqslant (q+1)b$ then $C(B_j)+C(B_i)-C(\tilde{B}_j)+C(\tilde{B}_i) = b$
- $(q+1)b \geqslant (a+b) \geqslant qb$ then $C(B_j) + C(B_i) - C(\tilde{B}_j) + C(\tilde{B}_i) = qb - a$
- if $qb \geqslant (a+b)$ then $C(B_j) + C(B_i) - C(\tilde{B}_j) + C(\tilde{B}_i) = 0$

In all cases give $(|\tilde{B}_i| + |\tilde{B}_j|) \leqslant (|B_i| + |B_j|)$
Hence, if there is k+1 blocks of size q, then $C(\mathcal{S}) \geqslant C_{min}(|B^{\mathcal{S}}_{n_0}|)$. By recurrence, if the smallest block in $\mathcal{S}$ has size q, then $C(\mathcal{S}) \geqslant C_{min}(|B^{\mathcal{S}}_{n_0}|)$.

Hypothesis: For an arbitrary p_0, if the smallest block in $\mathcal{S}$ has size at least p_0, then $C(\mathcal{S}) \geqslant C_{min}(|B^{\mathcal{S}}_{n_0}|)$.

Consider that the smallest block in $\mathcal{S}$ has size $p = p_0$,

Then there exists B_i and B_j two blocks such that $|B_i| = p$, $|B_j|$ is maximal. Create blocks $\tilde{B}_i$ and $\tilde{B}_j$ by moving task b, $b \in B_j,\ b \notin B_i$, from B_j to B_i.

With similar cases as above: $|\tilde{B}_i| + |\tilde{B}_j| \leqslant (|B_i| + |B_j|)$
Hence if the smallest block in $\mathcal{S}$ has size at $p = p_0 - 1$, then $C(\mathcal{S}) \geqslant C_{min}(|B^{\mathcal{S}}_{n_0}|)$.

By recurrence, $C(\mathcal{S}) \geqslant C_{min}(|B^{\mathcal{S}}_{n_0}|)$ for all p. □

Proposition 4. *$l = /\{J_i$, length of J_i is $n_0\}/$, $\mathcal{S}_{opt}$ an optimal schedule then $|B^{\mathcal{S}_{opt}}_{n_0}| \geqslant l$.*

Proof. Using the same argument as in lemma 4.6 □

Proposition 5. $MS_{HRr} = C_{min}(l)$

Proof. Blocks obtained by HRr have the same shape as blocks of $C_{min}(l)$.

By construction, all tasks but those from J_0 verify their precedence constraints as soon as they appear in the schedule. Hence $MS_{HRr} = \sum_t C(B^{HRr}_t)$ □

proof of the theorem. We have seen that in proposition 4 that $|B_{n_0}^{\mathcal{S}_{opt}}| \geqslant l$. Thus $C(\mathcal{S}_{opt}) \geqslant C_{min}(l)$ by proposition 3. This bound is attained by Hierarchical Round-robin hence the optimality. □

4.4 Approximations

We have seen that finding a schedule of jobs $J_i = (b_{i,i}, a_i)^{n_i}$ with an optimal makespan is NP-complete. Therefore, we try to find algorithms that provide an acceptable approximation.

Definition 10. *A λ-approximation algorithm is an algorithm whose execution time is polynomial in the instance size and that returns an approximate solution guaranteed to be, in the worst case, at a factor λ away from the optimal solution.*

Proposition 6. *Any list-heuristic provides a 2-approximation for $J_i = (b_i, a_i)^{n_i}$*

Proof. Given a schedule obtained by a list-based heuristic, by definition, if the B-processor is idle, it means that no B-task is available. This total idle time is $t_{idle} = MS_{list} - \sum_i n_i b_i$ All jobs are currently over or performing a A-task, hence the following bound with J_i the last executing job. This gives $t_{idle} \leqslant \sum_j a_{ij} \leqslant \sum_j (a_{ij} + b_{ij}) \leqslant opt$ □

Lemma 4.7. *This bound is tight.*

Proof. The straightforward heuristic would be to prioritize jobs with the most work remaining on processor B. This heuristic provides a approximation factor arbitrary close to 2 on the instance: $J_1 = (\epsilon, \epsilon)^k \to (\epsilon, 1-\epsilon)$ $J_2 = (1, 0)$ and ϵ small enough. □

5 Periodic Algorithms

In this section, we study periodic restriction of the problem. Indeed, in HPC, applications tend to have periodic or pseudo-periodic in the I/O behavior. HPC applications alternate between computation and I/O transfer, this pattern being repeated over-time. In addition, fault-tolerance techniques (such as periodic checkpointing) also add to this periodic behavior.

Late publications such that [1] and [8] propose I/O scheduling strategy with periodic approach.

Some of our objectives were to provide theoretical backgrounds to these algorithm and in a second time, to discuss their pertinence.

5.1 n-chains: $J_i = (b_i, a_i)^n$

We have proved that this problem is NP-hard (theorem 4.5)

As all chains length are equals and we want to ensure fairness between the different jobs. Let's consider a schedule built by ordering one task from each job in a periodic sequence and repeating this period n times. We call this kind of policy "periodic schedule" (Periodic), First, we discuss the way of ordering tasks within a period and then discuss the performance of such scheduling algorithms.

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

The overall idle time of the periodic schedule is:
$t_i(\text{Periodic}) = (n-1).\max_i \left(a_i - \sum_{j \neq i} b_j\right) + \max_k \left(a_k - \sum_{k \prec j} b_j\right)$
The order within a period does not change the overall idle time, therefore we can sort tasks by non-increasing A-task length in a period with gives:
$t_i(\text{Periodic}) \leq (n-1).\max_i \left(a_i - \sum_{j \neq i} b_j\right) + \max_k (a_k)$

Given an optimal schedule, the idle time is:
$t_i(\text{Opt})\max_i \left(na_i - n\sum_{j \neq i} b_j + \sum_{j \prec i} b_j\right) \geq n.\left(\max_k \left(a_k - \sum_{j \neq k} b_j\right)\right)$
where i is the last task running on A and i_1 is its first iteration.

Therefore, using straightforward bounds, the difference between these periodic and opt is at most:
$n.\max_i (a_i) - n\left(\max_k \left(a_k - \sum_{j \neq k} b_j\right)\right) \leq n.\max_i (a_i)$

The optimal makespan is at least $n.\max_i (a_i + b_i)$ Hence the following proposition.

Definition 11. *Define Periodic as the schedule obtain by running in each period one task of each job following an order between jobs (for example Johnson order see Lemma 4.1).*

Proposition 7. *Periodic is a ρ-approximation for $J_i = (b_i, a_i)^n$. With $\frac{11}{10} \leqslant \rho \leqslant 2$*

Remark. *The development above ensure that periodic is at worst a 2-approximation. However it may not be intuitive that Periodic does not provide an optimal schedule. The following example provides the other bound:*

- *3 chains of length 2*

- $J_i = (b_i = 2, a_i = 2)$ *for i=1,2*
- $J_3 = (b_3 = 1, a_3 = 0)$ *for i=3*

The best periodic algorithms give a makespan of 11. (1-2-3-1-2-3)
An optimal ordering has makespan 10. (1-2-1-2-3-3)

Lemma 5.1. *Periodic is asymptotically optimal for* $J_i = (b_i, a_i)^n$

Proof. Write T a period, $nT \leqslant MS_{\mathrm{P}eriodic} \leqslant nT + a_{max}$

By construction : $T = \sum_j b_j + \max_i(a_i - \sum_{j \neq i} b_j)$ $T = \max_i(a_i + b_i, Sigma b_j)$

$n * \max_i(a_i + b_i, \sum_j b_j) \leqslant MS_{opt} \leqslant MS_{\mathrm{P}eriodic} \leqslant n * \max_i(a_i + b_i, \sum_j b_j) + \max_i(a_i)$

Which gives : $\frac{MS_{\mathrm{P}eriodic}}{MS_{\mathrm{OPT}}} \leqslant 1 + \frac{1}{n}$

□

5.2 Periodic repetition: $J_i = [(b_i, a_i)^{k_i}]^n$

This problem is equivalent to the general case. However, as discussed before, the notation emphases the method of resolution. We aim to build a period containing k_i tasks of each job i using list scheduling and to repeat it n times.

This method provides at least a 2-approximation.

Using the same argument as in lemma 6

6 Model Evaluation

Experimental measurement The HPC-oriented team of Inria Bordeaux are working with ATOS. In the scope of this collaboration, we received a collections of data gathered by monitoring the execution of different applications on ATOS platform.

The application are rather short to suit a periodical behavior with execution time between 2h40 and 5 minutes. Given the complete log of I/O operation, we started to gather relevant timestamps (those who show significant amounts of operations) into phases. Then, we output the amount of such phases, their mean volume standard deviation, and their mean period and standard deviation. Additionally, we also compute the percentage of time taken by the first I/O phase and the one elapsed before the last in order to emphasized startup and shutdown behavior. For the applications that have more than 8 I/O phases, we also used the python function stats.normaltest to measure the distance with a normal distribution. It appears that these applications have a nearly periodical comportment with regular I/O volumes. see Appendix for extended view on the data.

7 Collaboration with the CLARISSE project

CLARISSE is a middleware designed to enhance data-staging coordination and control in the HPC software storage I/O stack [9]. It first use an prediction tool to anticipate I/O behavior, then when congestion is anticipated, it allocates inactive nodes of the platform to application to accelerate their computation and desynchronize their I/O.

CLARISSE currently uses first come - first served policy to choose which applications to speed-up. Using the prediction tool to produce an instance of our problem, the heuristics proposed in section 5.1 (prioritize jobs that have the largest computational requirements) would provide a schedule (ie an order of jobs). CLARISSE can then use this order in place of the current policy.

An other approach could be that our algorithm provides time frames where each I/O can be executed. Any I/O operations submitted outside this time frame would be delayed.

Our periodic schedule theoretically provides better performance than FIFO. Experiments with CLARISSE would help quantify this amelioration, validate our model and open new perspectives for algorithm design.

In practice, to be discussed

INPUT: The expected behavior of jobs: computation and I/O durations

- Our algorithm computes a schedule and provides two possible kind of output

OUTPUT:

1. an order between jobs to use in place of FIFO
2. for each I/O phase, a "release date" which is the moment when it can start.
3. for each I/O phase, a timeframe in which it should be triggered. I/O requests that arrive outside these dates are delayed.

8 Related Work

Surprisingly enough, periodic problems is not so abundant among the scheduling literature despite their interest.

Most of them are very application-oriented so the model is not easily exportable. We are not an exception, as our model is designed to represent I/O issues of HPC applications.

Periodic scheduling are also studied in the scope of cyclic scheduling ie considering that the tasks being scheduled have periodical release date ([4],[16], [5]) which does not cover the latency issue.

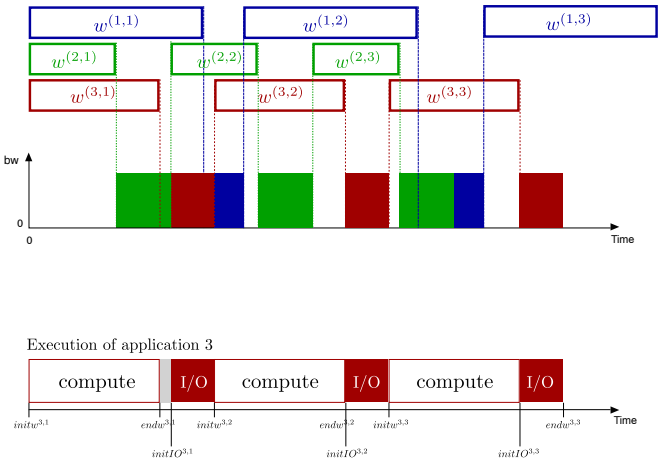

Figure 2: Scheduling three applications

In this figure, we show an example of three applications competing for I/O bandwidth. On the top part, we see the applications doing computations without any constraint. However at the end of their computations, all applications need to transfer some volume of I/O. The bandwidth is therefore used, in turn, by each application. The order of these turn is what we intend to provide with our algorithms.

The following table is a non-exhaustive survey of problem that approach ours:

Problem	Solution	Reference	Remark
$J_i = (a_i, b_i)$	Polynomial	[17] [18]	
$J_i = (a, b)^{n_i}$	Polynomial	Round-Robin	
$J_i = (a_i, b_i)$ $1 \leqslant i < m$, $J_m = (a_i, b_i)^2$	NP-complete	[17]	
$J_i = (a_i, b_i)^2$	NP-complete		
$J_i = (a_i, b_i)^n$	NP-complete	Theorem 4.4	Periodic [(1+1/n)-approx]
$\rightarrow J_i = [(a_i, b_i)^{k_i}]^n$	NP-complete	Open	2-approximation
$J_i = \prod_{j=0}^{n_i} (a_{i,j}, b_{i,j})$	NP-complete	list-scheduling	2-approx
P|2FL|C_{max}	Polynomial	[18]	
P_m|2FL|C_{max}	Pseudo-polynomial	[18]	
1|r_j|S_{max}	NP-complete	[2]	3-partition
Periodic C_{max}/S_{max}	NP-complete	[7]	3-partition
F_2|chains|C_{max}	NP-complete	[11]	
F_2|chains,pmnt|C_{max}	NP-complete	Lenstra	cf.[3]
F_m|$p_{i,j}$=1,chains|$\Sigma w_i C_i$	NP-complete	[13]	
F|$p_{ij} = 1; prec$|C_{max}	NP-hard	[12, 15]	
O_2|chains|C_{max}	strongly NP-hard	[13]	
O_2|pmnt|$\Sigma w_i C_i$	strongly NP-hard	Lenstra	
1|$min\ delays, k\ 1 - chains$|$C_{max}$	P	[17]	
1|$min\ delays, k\ 2, 1...1 - chains$|$C_{max}$	NP-complete	[17]	
1|$min\ delays, k\ 2, 2 - chains$|$C_{max}$	strongly NP-complete	[17]	
1|$min\ delays, k\ n_1, 1...1 - chains$|$C_{max}$	strongly NP-complete	[17]	
1|$min\ delays, k\ 1 - chains$|$\Sigma w_i C_i$	NP-complete	[17]	

References

[1] Guillaume Aupy, Ana Gainaru, and Valentin Le Fèvre. Periodic i/o scheduling for super-computers. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 44–66. Springer, 2017.

[2] Michael A Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA*, volume 98, pages 270–279, 1998.

[3] Peter Brucker and P Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[4] Rainer E Burkard. Optimal schedules for periodically recurring events. *Discrete Applied Mathematics*, 15(2-3):167–180, 1986.

[5] Wolfgang Dauscha, Heinz D Modrow, and Alexander Neumann. On cyclic sequence types for constructing cyclic schedules. *Zeitschrift für Operations Research*, 29(1):1–30, 1985.

[6] Jianzhong Du, Joseph YT Leung, and Gilbert H Young. Scheduling chain-structured tasks to minimize makespan and mean flow time. *Information and Computation*, 92(2):219–236, 1991.

[7] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the i/o of hpc applications under congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1013–1022. IEEE, 2015.

[8] Thomas Herault, Yves Robert, Aurélien Bouteiller, Dorian Arnold, Kurt Ferreira, George Bosilca, and Jack Dongarra. *Optimal Cooperative Checkpointing for Shared High-Performance Computing Platforms*. PhD thesis, INRIA, 2017.

[9] Florin Isaila, Jesus Carretero, and Rob Ross. Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 346–355. IEEE, 2016.

[10] Arnaud Legrand, Alan Su, and Frédéric Vivien. Minimizing the stretch when scheduling flows of divisible requests. *Journal of Scheduling*, 11(5):381–404, 2008.

[11] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Ann. of Discrete Math.*, 1:343–362, 1977.

[12] J.Y.-T. Leung, O. Vornberger, and J.D. Witthoff. On some variants of the bandwidth minimization problem. *SIAM J. Comput.*, 13(3):650–667, 1984.

[13] V Tanaev, W Gordon, and Yakov M Shafransky. *Scheduling theory. Single-stage systems*, volume 284. Springer Science & Business Media, 2012.

[14] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to algorithms*. The MIT Press, 2 edition, 2001.

[15] V.G. Timkovsky. Identical parallel machines vs. unit-time shops and preemptions vs. chains in scheduling complexity. *European J. Oper. Res.*, 149(2):355–376, 2003.

[16] A Vince. Scheduling periodic events. *Discrete Applied Mathematics*, 25(3):299–310, 1989.

[17] Erick D. Wikum, Donna C. Llewellyn, and George L. Nemhauser. One-machine generalized precedence constrained scheduling problems. *Operations Research Letters*, 16(2):87 – 99, 1994.

[18] Guangwei Wu, Jianer Chen, and Jianxin Wang. On scheduling two-stage jobs on multiple two-stage flowshops. *arXiv preprint arXiv:1801.09089*, 2018.

A K-subset sum

Adapted from [14]

To show that k-SUBSET-SUM is in NP, for an instance $< \mathcal{A}, B >$ of the problem, we let the subsets S_1 and S_2 be the certificate. Checking whether $B = \sum_{a_i \in S_1} a_i + 2 \sum_{a_j \in S_2} a_j$ can be accomplished by a verification algorithm in polynomial time/

We now show that 3-SAT can be reduced to SUBSET-SUM. Given a 3-SAT formula /phi over variables $x_1, x_2, ..., x_n$ with clauses $C_1, C_2, ..., C_k$, each containing exactly three distinct literals, the reduction algorithm constructs an instance $< \mathcal{A}, B >$ of the duplicated subset-sum problem such that ϕ is satisfiable if and only if there are subset S_1 and S_2 of $\mathcal{A}$such that $B = \sum_{a_i \in S_1} a_i + 2 \sum_{a_j \in S_2} a_j$.

Without loss of generality, we make two simplifying assumptions about the formula ϕ. First, no clause contains both a variable and its negation, for such a close is automatically satisfied by any assignment of values to the variables. Second, each variable appears in at least one clause, for otherwise it does not matter what value is assigned to the variable.

The reduction creates two numbers (and their duplicate if need be) in set $\mathcal{A}$for each variable x_i and two numbers in $\mathcal{A}$for each clause C_j. We shall create numbers in base 10, where each number contains n+k digits and each digit corresponds to either one variable or one clause. Base 10 has the property we need of preventing carries from lower digits to higher digit. We construct set $\mathcal{A}$and target B as follows. We label each digit position by either a variable or a clause. The least significant k digits are labeled by the clauses, and the most significant n digits are labeled by variables.

- The target t has a 1 each digit labeled by a variable and a 3 in each digit labeled by a clause.
- For each variable x_i, there are two integers, v_i and v'_i in $\mathcal{A}$. Each has A in the digit labeled by x_i and O's in the other variable digits. If literal x_i appears in clause C_j, then the digit labeled by C_j in v_i contains a 1. If literal $\neg x_i$ appears in clause C_j, then the digit labeled by C_j in v'_i contains a 1. All other digits labeled by clauses in v_i and v'_i are 0.

  So far all v_i and v'_i values in set $\mathcal{A}$are unique. Why? For $l \neq i$, no v_l or v'_l values can equal v_i and v'_i in the most significant n digits. Furthermore, by our simplifying assumptions above, no v_i and v'_i can be equal in all k least significant digits. If v_i and v'_i were equal, then x_i and $\neg x_i$ would have to appear in exactly the same set of clauses. But we assume that no clause contains both x_i and $\neg x_i$ and that either x_i or $\neg x_i$ appears in some clause, and so there must some clause C_j for which v_i and v'_i differ.

- For each clause C_j, there are two integers, s_j and s'_j in S. Each has 0's in all digits other than the one labeled by C_j. s_j and s'_j have a 1 in the C_j digit. These integers are "slack variables", which we use to get each clause-labeled digit position to add to the target value of 3.

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	1	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	1	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	1	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	1
t	=	1	1	1	3	3	3	3

Figure 3: Example: The reduction of 3-SAT to SUBSET-SUM. The formula is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3), C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3), C_3 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ and $C_4 = (x_1 \vee x_2 \vee x_3)$. A satisfying assignment of ϕ is $< x_1 = 0, X_2 = 0, X_3 = 1 >$. The set S produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, S = 1001001, 1000110, 100001, 101110, 10011, 11100, 1000,1000, 100,100, 10,10, 1,1. The target t is 1113333. The subset $S' \subset S$ contains v'_1, v'_2, and v_3, corresponding to the satisfying assignment. It also contains slack variables $s_1, s'_1, s'_2, S_3, s_4$ and s'_4 to achieve the target value of 3 in the digits labeled by C_1 through C_4.

Note the the greatest sum of digits in any one digit position is 8, which occurs in the digits labeled by clauses (six 1's from the v_i and v'_i and their duplicates, plus two 1 for s_j and s'_j). Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits. The reduction can be performed in polynomial time. The set $\mathcal{A}$contains 2n+2k values, each of which has n+k digits, and the time to produce each digit is polynomial in n+k. The target t has n+k digits, ans the reduction produces each in constant time.

We now show that the 3-SAT formula ϕ is satisfiable if and only if there are subsets S_1 and S_2 of $\mathcal{A}$ whose sum is B. First, suppose that ϕ has a satisfying assignment. For i = 1, 2,... ,n, if $x_i = 1$ in this assignment, then include v_i, in S_1. Otherwise, include v'_i. In other words, we include in S_1 exactly the v_i and v'_i values that correspond to literals with the value 1 in the satisfying assignment. Having included either v_i or v_i, but not both, for all i, and having put 0 in the digits labeled by variables in all s_j and s'_j, we see that for each variable-labeled digit, the sum of the values of S_1 must be 1, which matches those digits of the target t. Because each clause is satisfied, there is some literal in the clause with the value 1. There- Therefore, each

digit labeled by a clause has at least one 1 contributed to its sum by a v_i or v_i' value in S_1. In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause-labeled digit has a sum of 1, 2, or 3 from the v_i and v_i' values in S_1'. We achieve the target of 3 in each digit labeled by clause C_j by including in S_1 the appropriate nonempty subset of slack variables s_j, s_j'. Since we have matched the target in all digits of the sum, and no carries can occur, the values of S_1 sum to B.

Now, suppose that there are subsets S_1 and S_2 of $\mathcal{A}$such that $B = \sum_{a_i \in S_1} a_i + 2 \sum_{a_j \in S_2} a_j$. The subset S_1 must include exactly one of v_i and v_i' for each i = 1,2,...,n, for otherwise the digits labeled by variables would not sum to 1. For the same reasons S_2 must be empty. If $v_i \in S_1$, we set $x_i = 1$. Otherwise, $v_i' \in S_1$, and we set $x_i = 0$. We claim that every clause C_j, for j = 1,2 k, is satisfied by this assignment. To prove this claim, note that to achieve a sum of 3 in the digit labeled by C_j, the subset S_1 must include at least one v_i or v_i' value that has a 1 in the digit labeled by C_j, since the contributions of the slack variables s_j and s_j' together sum to at most 2. If S_1 includes a v_i that has a 1 in that position, then the literal x_i appears in clause C_j. Since we have set $x_i = 1$ when $v_i \in S_1$, clause C_j is satisfied. If S_1 includes a v_i' that has a 1 in that position, then the literal $\neg x_i$ appears in C_j. Since we have set $x_i = 0$ when $v_i' \in S_1$, clause C_j is again satisfied. Thus, all clauses of ϕ are satisfied, which completes the proof.

Remark. *The very same proof can be adapted for k-subset-sum.*

B Traces

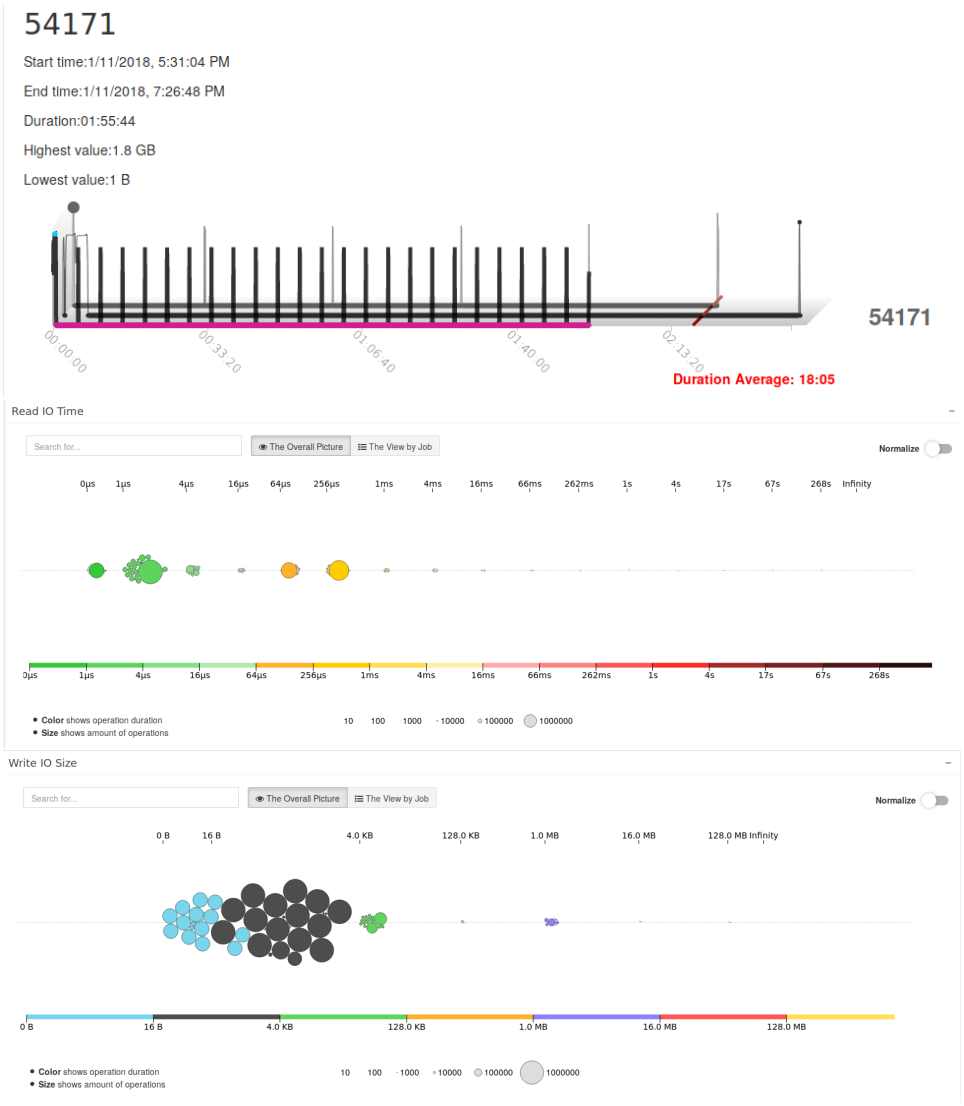

Figure 4: Sample of data provided by ATOS

id	total_duration	total_volume	phases	starting_IO	terminating_IO	mean (period)	sd (period)	pvalue	mean (vol)	sd (vol)	pvalue
10246	335000	1472542576	2	95,52%	100,00%	1,50E+04			7,36E+08	7,11E+08	
4799	280000	1472541830	3	10,71%	100,00%	1,25E+04	2,50E+03		4,91E+08	4,53E+08	
###	495000	306596	6	0,00%	81,82%	1,00E+04			5,11E+04	5,54E+04	
54171	6940000	108571914971	26	0,50%	99,93%	2,72E+05	4,98E+04	5,91E-14	4,18E+09	1,36E+09	8,66E-07
10256	270000	1472542576	3	11,11%	98,15%	1,25E+04	2,50E+03		4,91E+08	4,53E+08	
10235	325000	1472542576	3	13,85%	100,00%	1,25E+04	2,50E+03		4,91E+08	4,53E+08	
51682	1615000	40886112341	19	2,79%	99,07%	8,22E+04	7,19E+04	2,63E-01	2,15E+09	5,43E+09	9,93E-09
10248	330000	1472541830	3	15,15%	100,00%	1,50E+04	5,00E+03		4,91E+08	4,53E+08	
10260	335000	1472542576	2	94,03%	98,51%	1,50E+04			7,36E+08	7,11E+08	
47579	5775000	22627216337	115	0,95%	99,57%	4,90E+04	3,58E+04	6,04E-24	1,97E+08	1,95E+09	1,57E-54
47818	2145000	103103082	65	0,23%	99,07%	3,30E+04	4,64E+03	1,62E-04	1,59E+06	8,86E+06	2,17E-24
10258	250000	1472542576	2	94,00%	100,00%	1,50E+04			7,36E+08	7,11E+08	
47458	9550000	17302692653	8	0,16%	99,84%	1,27E+06	3,08E+06		2,16E+09	2,93E+09	
48565	2145000	124731940002	1	100,00%	0,00%				1,25E+11		
55618	3125000	8732332828	8	0,96%	100,00%	4,39E+05	1,75E+05		1,09E+09	1,77E+09	
10253	335000	1472542576	2	94,03%	98,51%	1,50E+04			7,36E+08	7,11E+08	
54376	4620000	295325749	1	100,00%	0,00%				2,95E+08		
6012	8365000	80611214443	6	0,24%	99,70%	1,65E+06	1,17E+04		1,34E+10	6,91E+09	
10261	335000	1472542576	2	95,52%	100,00%	1,50E+04			7,36E+08	7,11E+08	
10259	255000	1472541830	2	94,12%	100,00%	1,50E+04			7,36E+08	7,11E+08	
47752	1295000	7269439695	6	1,54%	99,23%	2,45E+05	1,92E+05		1,21E+09	7,92E+08	
10255	270000	1472541830	2	92,59%	100,00%	2,00E+04			7,36E+08	7,11E+08	
47512	720000	61203503852	7	0,69%	98,61%	1,08E+05	1,64E+05		8,74E+09	1,11E+10	
10247	495000	8192000000	1	100,00%	0,00%				8,19E+09		
10251	40000	8192000000	1	100,00%	0,00%				8,19E+09		
10233	325000	982034745	1	100,00%	0,00%				9,82E+08		
10252	340000	1472542576	3	16,18%	98,53%	1,25E+04	2,50E+03		4,91E+08	4,53E+08	
10250	380000	1472542576	3	13,16%	98,68%	1,25E+04	2,50E+03		4,91E+08	4,53E+08	
55535	1685000	8733177895	7	2,97%	100,00%	2,73E+05	2,50E+03		1,25E+09	2,89E+09	

Table 1: Statistical analysis of ATOS data.
duration in ms, operations in occurrence number