Scheduling Jobs and Batches Based on Historical Data

Richárd Kápolnai, Imre Szeberényi

***Abstract*—** This paper considers the problem of minimizing the makespan when scheduling jobs of unknown length on identical machines, meaning there is no *a priori* information on the job lengths. However, as the user of a parallel computational infrastructure is assumed to repeat the execution of the jobs multiple times and is able to measure individual machine completion times, a historical database is proposed to aid the approximation of the optimal schedule. We limit the size of this database so any set of jobs assigned to a machine has to be briefly described with limited data so an arbitrary feasible schedule may not be suitable.

Two job models are studied: the *Parameter Sweep Application (PSA)* model which can be regarded as a set of jobs without dependency or inter-communication requirements, and the *Batches of PSAs (BPSA)* model where the jobs are executed in batches to be preceded by a sequence independent setup work.

We propose an iterative framework which repeats computing an approximate schedule, executing the PSA or BPSA and updating the historical database according to the machine completion times. After each iteration, the approximation algorithm further improves some upper bound on the makespan until a 2-approximation is reached in case of PSA, 3-approximation in case of BPSA. The scheduling algorithm always assigns consecutive jobs called *chains* to machines keeping the historical database and the machine assignment descriptions brief.

***Keywords*—**approximation, chain partitioning, historical data, batch scheduling, uncertainty, workload balancing

I. Introduction

We investigate the problem of scheduling a *Parameter Sweep Application (PSA)* on a parallel computational infrastructure. In a PSA, the same computation has to be executed for many different values of a parameter, which is common in engineering or scientific computations. To each parameter value, we assign a non-interruptible *job*. We call the domain of all possible parameter values the *parameter space*, corresponding to the set of jobs. Every job requires a *processing time* to complete called its *length*, and a machine's *completion time* is the total length of work assigned to it. We consider the cost of a schedule the *makespan*, i.e. the maximal completion time. Hence the underlying problem is the widely known NP-hard optimization problem of scheduling independent jobs to identical parallel machines while minimizing the makespan, often denoted as $P||C_{\max}$, except that we have no *a priori* knowledge of the job lengths. To maintain tractability, we focus on a 2-approximation, where an α-*approximation* is a polynomial time algorithm yielding a solution of cost at most α times the optimal cost and α is called the *approximation ratio*.

Our inspiration was a supposedly common usage scenario. When a user develops a PSA for a project, sometimes executes it several times during the project, as observed by others [1], because of different reasons: testing, fixing bugs, adding new features such as more detailed output, improved precision etc. To exploit a parallel infrastructure, the user has to partition somehow the parameter space by creating a *schedule*, i.e. a mapping from the parameter space to the set of machines. We call the work of a machine, which is the set parameters assigned to it, an *assignment*. In many PSAs, the processing times of the parameters may be significantly different and hard to predict, so the first partition may result in a makespan far from optimal. So after the execution, assuming the measured machine completion times are available, the user may notice that some assignments took extremely short or long time to complete. To minimize the makespan, the user should adjust the partition intuitively or systematically in order to shorten the longest assignments at the next execution. If the makespan still turns out to be too high, further adjustments are made until the makespan meets the requirements.

Our most essential assumption is that most users tend to repeat executing their PSAs. Naturally not to optimize the makespan, but for various reasons such as executing later a revised version of PSA. This kind of user behaviour is assumed throughout this paper:

Conjecture 1 ([1]). *The user submits for execution a PSA multiple times.*

It also is important to notice that the process above is validated by another strong assumption: the parameter space and processing times of the parameters remain constant between the executions, nevertheless, the user may do changes to the PSA (e.g. fix a program bug). The lack of these ideal conditions could clearly influence both the cost of schedules and the convergence properties of this process.

The main goal of this paper is proposing a generic, iterative framework to help automating this process. In each iteration the framework updates its historical database with the measured completion times of the execution of the previously computed schedule. If it is not clear whether the approximation ratio of 2 has already been reached, then splits the long assignments into parts estimating proportionally their lengths, contracts the small ones, and re-computes the schedule for the next execution. We prove that, under all assumptions above, after a finite number of iterations the approximation ratio

Authors are with the Department of Control Engineering and Information Technology, Budapest University of Technology and Economics, Hungary, email: kapolnai@iit.bme.hu.

The authors acknowledge the use of resources provided by the European Grid Infrastructure. For more information, please reference the EGI-InSPIRE paper (http://go.egi.eu/pdnon).

of 2 is always reached for any PSA. For a demonstration, we implemented a preliminary version of the framework as a Saleve client [2]. The demonstration includes a simple synthetic example as well as a real, previously published PSA. We believe this framework can be incorporated into existing PSA execution supporting tools or even scheduling techniques.

According to Conjecture 1, the PSA is not repeated because our framework needs more iteration. It is repeated regardless of the framework because this is the wish of the user. So the framework does not cause additional execution costs, instead it gradually decreases the execution costs.

To keep in sight easy implementation and small historical database, let us suppose the parameter space is an ordered set. Our scheduling algorithm only assigns a set of consecutive jobs called a *chain* to any machine, which keeps the machine assignment descriptions brief so the historical database. So in a nutshell, based on historical data, the framework splits the long chains and contracts the small ones in each iteration.

It is worth clarifying that we have two different approximation ratios in the process that are not necessarily equal. We call the *framework ratio* the cost of the schedule after the final iteration of the framework divided by the cost of the optimal *a priori* schedule of the original jobs. Within an iteration of the framework, we create a synthetic scheduling problem with known or estimated job lengths: these synthetic jobs are the previously measured or temporarily estimated chains consisting of the original jobs of unknown length. To schedule the synthetic jobs, we use some approximation algorithm, and we call its approximation ratio the *iteration ratio*, which is the cost of the computed schedule of the synthetic jobs divided by the optimal cost of scheduling the synthetic jobs. Obviously the iteration ratio is a lower bound on the framework ratio.

The iteration ratio of 2 cannot be further improved unless we give up our restriction of scheduling only chains of jobs. Hence we say the *price of chaining* is 2, inspired by Papadimitriou's definition of the "price of anarchy" [3]. So if the limited size of the database compensates this price of chaining, then we insist on applying it because of its adequacy to our problem.

We also extend the iterative framework for an application model more general than PSA we call *Batches of PSAs (BPSA)*, in which the jobs are partitioned into *families*. Families are divided into *batches* in a schedule, where a batch is a subset of a family scheduled on one machine. The difference between a BPSA and a PSA is that a machine can execute a batch only if it already executed a *setup* for that family which also has processing time. We suppose the setup lengths are *known a priori*, but the job lengths are still unknown. A number of applications follow the BPSA model. Some common examples for setup work can be downloading and preprocessing the input data such as images, and the different families of jobs operate on different input sets, or, installing a specific software or hardware environment such as a virtual machine or some physical equipment before starting the effective work.

Organization of this paper: Section II enumerates some related work. Section III states the underlying problem formally, and presents the iterative framework to solve it. Section IV proves that the framework meets the requirements which is illustrated in Section V with a synthetic and a real-world PSA. Section VI extends the framework for batches with setups. Finally, the paper is concluded by Section VII.

II. Related Work

The classical scheduling problem $P||C_{\max}$ has received much attention through the past decades. We refer to [4, 5] for a survey of the first approximation algorithms including limited approximations schemes for fixed number of machines. There also exist simple 2-approximation algorithms, e.g. Graham's list scheduling [6] or the more general one of Archer and Tardos [7] which rounds a fractional schedule. However, these algorithms cannot be used in the iterations of the framework because of incompatibilities shown in Section IV, therefore we present yet another 2-approximation for $P||C_{\max}$ which schedules chains. Many approximations for scheduling batches (the BPSA model) are surveyed in [8], and we use an idea from [9] to compute the synthetic schedule. Setups can be both software of hardware tasks [10, 11].

Scheduling chains is usually referred to in the literature as *one dimensional array partitioning* or *chain-on-chain partitioning*, or, in the case of identical machines, equivalently *chain partitioning*. In our setting, the optimal chain partitioning can be found in polynomial time [12], and exact solutions are surveyed in [13, 14]. Despite the efficiency of exact solutions, we are not interested in them because even an optimal chain partitioning is just another 2-approximation for the original problem $P||C_{\max}$. This is a direct consequence of the price of chaining, as detailed in Section III.

There are a number of approaches to design scheduling techniques when uncertainty arises. Some surveys characterize techniques as proactive (more robust to unplanned changes) and reactive (less committed to existing plan) [15, 16]. For instance, a proactive approach could be utilizing the average and worst-case execution times [17]. The well-known list scheduling could be considered as a reactive method when job lengths are unknown *a priori*: when a machine is idle because of finishing its temporary assignment, it gets one new job. This setup is also known as self-scheduling. While self-scheduling is a good approximation in theory, obviously effects an enormous overhead in a system. Guided self-scheduling and factoring algorithms offer a compromise: they form chunks of jobs (similar to our chains) to reduce the overhead. Initially these algorithms dispatch large chunks, then the chunk size will gradually decrease to improve balancing [18].

However, reactive behaviour usually implies the algorithm intervenes *during* the execution of the schedule based on some quasi real-time feedback of the system state, aiming to improve *one* schedule. Instead, our long-term goal is simpler: we aim to collect historical data to improve the next schedules. We note that getting real-time feedback may be very expensive in distributed systems. Our framework uses only a historical database which is not real-time, thus we believe it is more generally applicable and easier to implement.

The problem of unknown jobs had been attacked with numerous methods. A straightforward idea to require the users to provide the processing times [19]. AppLeS [20] and GrADS [21] require the user to provide explicit performance

model of the application to predict processing times. Probabilistic estimation was also studied, assuming the lengths are independent random variables [22]. Uncertainty can concern not only the application attributes (the number of jobs, their lengths etc.) but also the resource performances (number and speed of machines etc.) [23], although the latter case is beyond the scope of this paper.

Frequently, the application properties could be learnt by building a historical database from measurements from previous executions. Probabilistic estimation could be backed up by a database to estimate the average of the distribution [24]. A database can support machine learning techniques to automatically develop performance models from the application source code [25], or to predict processing times using records of *similar* jobs where similarity is defined by some distance function in the attribute space considering e.g. user and job identification information, requested resources [26, 27, 28].

Some work focus on collecting processing times of the whole application [29, 30] in order to optimize resource performance such as utilization. Others, as well as this work, measure the parts of an application to optimize application running time [31, 32, 33, 34]. Some of them model the processing time to be depending on the input data size so a time can be predicted by projecting a previously measured time on the same resource for another data set (e.g. [34, 30]). In our case, we cannot use this prediction system for the chain lengths because then every job (or parameter) would have uniform processing time, contradicting to our setting. In [32] and [33], the goal is determining the optimal number of machines to achieve the desired speed-up and efficiency. MARS [31] collects both application-specific and system-specific information to balance loads, but it was designed for more general parallel applications than a PSA, and it does not exploits that the tasks may be divisible as our chains.

III. The Proposed Iterative Framework

In this section, after introducing the notations and the basic properties of chain partitioning, we formalize our objective and present the framework.

A. Approaching the problem with chain partitioning

We wish to schedule n non-interruptible jobs of length $p_1, \ldots, p_n$ onto m identical machines. A schedule f is a mapping $f : [1 \ .. \ n] \mapsto [1 \ .. \ m]$, and we say f is a chain partitioning iff either $f(j+1) = f(j)$ or $f(j+1) = f(j)+1$ holds for all $1 \leq j < n$, so each assignment is a chain. A chain c is a subinterval within $[1 \ .. \ n]$, its cardinality is its *size*, its processing time is its *length*, denoted by $\ell(c)$. The completion time of machine i is the total length of its assignment: $C_i(f) = \sum_{f(j)=i} p_j$. A chain containing only one job is called a *singleton*, its size is 1. Let $C^*_{\max}$ denote the makespan of the optimal schedule (NP-hard to compute), and $C^{1D*}_{\max}$ the makespan of the optimal chain partitioning (computable in polynomial time). We call the *price of chaining* the supremum of the quotient $C^{1D*}_{\max}/C^*_{\max}$ over all valid scheduling problems. To demonstrate the price of chaining is 2, we have

Example 1. *Let $n = 2m$, and the processing times $p_1 = \ldots = p_m = m$, $p_{m+1} = \ldots = p_{2m} = 1$.*

Clearly, for this input $C^*_{\max} = m+1$ and $C^{1D*}_{\max} = 2m$, so if the price of chaining exists, it is at least 2. On the other hand, as the 2-approximation framework presented by this paper is a chain partitioning as well, we have $C^{1D*}_{\max} \leq 2 \cdot C^*_{\max}$. We note that a slight alteration of the algorithm of [7] also admits a 2-approximation chain partitioning.

According to our assumption on uncertainty, after executing a schedule f, all measured completion times $\mathbf{C}(f)$ become known and can be used to optimize future schedules. $\mathbf{C}(f)$ stands for the vector $C_1(f), \ldots, C_m(f)$. It is easy to see that any chain partition f can be unambiguously described in $O(m \log n)$ space: the m assignments are intervals of integers, so the description needs m integer numbers, each number given in $\log n$ digits. The framework uses a historical database containing less than $2m$ chains, as presented in the next subsection.

B. Outline of the Framework

The working of the framework is outlined in Algorithm 1, details are elaborated in this section. Given m, n, the historical database contains initially an arbitrary initial schedule f_0, and its measured, accurate completion times $\mathbf{C}(f_0)$. f_0 can be obtained e.g. by splitting the parameter space into equally sized parts. In the qth iteration, starting with $q = 1$, the schedule f_q has to be computed based on the database. After the execution of f_q, the measurements $\mathbf{C}(f_q)$ are merged into the database.

Algorithm 1 *Iterative framework*

```
1:  procedure PSAPartition(m, n, f_0, C(f_0))
2:      q ← 0, calculate T_LB
3:      while s(q) > 1 do
4:          q ← q + 1
5:          P ← a synthetic scheduling problem of the chains
            of the database and their length
6:          for all original chain c in P s.t. ℓ(c) > 2T_LB do
7:              Split chain c in half
8:              Estimate the length of the half chains
9:              Replace chain c with the half chains in P
10:         f_q ← FrugallySchedule(P, T_LB)
11:         Execute schedule f_q, measure C(f_q)
12:         Merge C(f_q) into the database
13:         Recalculate T_LB
```

We define the indicator of exit condition $s(q)$ to be the maximum size of the chains in the database of length more than $2T_{\text{LB}}$ at the end of the qth iteration, and 0, if every length is under $2T_{\text{LB}}$. The framework exits when every long chain is a singleton, i.e. $s(q) = 1$ or $s(q) = 0$.

We define *the chains of the database after merging* $\mathbf{C}(f_q)$ in line 12 as follows, which we rely in lines 5 and 13 on. A chain assignment c of f_q may or may not contain a chain c^* of estimated length as a subset. If it does not, then we say c is a chain of the database with is its measured length (some completion time). Otherwise it is easy to verify that $c \setminus c^*$ is

also a chain because c^* is either prefix or suffix subset of c. In addition, its length $\ell(c \setminus c^*)$ was already known as stated later by Proposition 2 (previously measured or calculated), so finally after executing f_q the *exact* length of c^* can be calculated from measurements: $\ell(c^*) = \ell(c) - \ell(c \setminus c^*)$. So in this case we say c^* and $c \setminus c^*$ are chains of the database. Hence for each assignment c, we have either one or two chains in the database, so the database consists of less than $2m$ chains. We note that the chains of the database are disjoint and cover $[1 \mathrel{..} n]$, i.e. form a partition. We also note that it is crucial to gather as much information as possible on the half chains of estimated length as they are the longest ones exercising the most influence on the makespan. This is the reason we need to store more information than the last schedule f_q and its completion times $\mathbf{C}(f_q)$.

The goal of the framework is to assure that $C_{\max} \le 2C^*_{\max}$ after a finite number of iterations. For this purpose, a lower bound on the optimal makespan $C^*_{\max}$ is determined in each iteration: the average completion time $T_{\text{LB}} := \sum_{j=1}^{n} p_j/m$. We note that we could omit the recalculation of T_{LB} in every iteration (line 13), but it provides the framework some adaptivity: if the processing times change because of some alteration in the PSA, the recalculation of T_{LB} restores the convergence.

In each iteration, a synthetic scheduling problem is prepared from the chains of the database after an adjustment: the longest chains (longer than $2T_{\text{LB}}$) are split in half. The length of the half chains are unknown, so they are estimated temporarily as the half of the original length. Each chain corresponds to a synthetic job of the same length, where a synthetic length is either provided by the database or estimated as above. The synthetic jobs are scheduled by the algorithm called FRUGALLYSCHEDULE, presented in Algorithm 2, which is a simple array partitioning algorithm for jobs of known length, with important features though. FRUGALLYSCHEDULE returns a schedule f' which can be trivially interpreted also as a schedule f of the original jobs of unknown length. We mention that in the earlier conference version of this paper [35] this algorithm also contained an optional heuristics to move to the next machine if its workload reached T.

Algorithm 2 *Subroutine: schedules frugally*

Require: P: n' synthetic jobs of known length $p'_1, \ldots, p'_{n'}$
Ensure: schedule f' is a mapping $f' : [1 \mathrel{..} n'] \mapsto [1 \mathrel{..} m]$

1: **function** FRUGALLYSCHEDULE(P, T)
2: $\quad i \leftarrow 1$
3: $\quad C_1 \leftarrow 0, \ldots, C_m \leftarrow 0$
4: $\quad$ **for all** job j **do**
5: $\quad\quad$ **if** $C_i > 0$ and $C_i + p'_j > 2T$ **then** $i \leftarrow i + 1$
6: $\quad\quad$ **if** $i > m$ **then ABORT!** T is too small
7: $\quad\quad$ Assign job j to machine i, i.e. $f'(j) := i$
8: $\quad\quad$ Update C_i
9: $\quad$ **return** the computed schedule f'

The next section presents the sketch of a formal proof that the framework meets the requirements, while Section V strengthens it via examples.

IV. ANALYSIS OF THE FRAMEWORK

Although the framework and the scheduling subroutine look primitive, together they successfully manage some non-trivial issues. Most important one is the convergence (finite number of iteration), achieved by the framework by keeping decreasing the sizes of the longest chains in the database until every non-singleton chain is under $2T_{\text{LB}}$. To assist in that, the scheduling subroutine should not map more than one synthetic job of estimated size ("half chain") to a machine. If each estimation is specified with exact measurements by the end of the iteration, long chains will keep getting smaller and shorter. Even an optimal chain partitioning may map more than one estimated chain to a machine as well as other approximations such as the one in [7], so the existence of FRUGALLYSCHEDULE (Algorithm 2) is justified.

Another aid in maintaining convergence is that the database stores more than just the last measured completion times, as illustrated below. The notation $f[a \mathrel{..} b] = i$ stands for $f(j) = i$ for all $j \in [a \mathrel{..} b]$.

Example 2. *Let* $m = 3$ *and* $n = 4$ *with* $p_1 = 20, p_2 = 70, p_3 = 20, p_4 = 10$. *Let the initial schedule* f_0 *be the following mapping:* $f_0[1] = 1, f_0[2 \mathrel{..} 3] = 2$ *and* $f_0[4] = 3$.

So the lengths of the chains of the database are $\ell([1]) = C_1(f_0) = 20$, $\ell([2 \mathrel{..} 3]) = C_2(f_0) = 90$ and $\ell([4]) = C_3(f_0) = 10$. As $T_{\text{LB}} = 40$, the chain $[2 \mathrel{..} 3]$ is split by the framework, and the half chains $[2]$ and $[3]$ both gets estimated length of 45. Then the next computed schedule f_1 contracts the first two chains, so the measurements $\mathbf{C}(f_1)$ are: $\ell([1 \mathrel{..} 2]) = 90, \ell([3]) = 20$ and $\ell([4]) = 10$. If the previous data $C_1(f_0)$ was already unavailable, then there would not be any significant improvement on the quality of the database: the size and the length of longest chain would be the same (2 and 90). However, using both $\ell([1]) = C_1(f_0)$ and $\ell([1 \mathrel{..} 2]) = C_1(f_1)$ the framework can conclude that $\ell([2]) = 70$.

As the difficulties above shows, the framework and the subroutine has to interact to succeed. We begin the discussion of the framework with the properties of FRUGALLYSCHEDULE.

Proposition 1. *Given the number of machines* m *and* n' *jobs of length* $p'_1, \ldots, p'_{n'}$ *by the problem* P, *let* $p'_{\max} := \max_{1 \le j \le n'}\{p'_j\}$. *For the schedule* f' *returned by* FRUGALLYSCHEDULE(P, T) *in Algorithm 2, we have the following:*

(i) if $T \ge \sum_{j=1}^{n'} p'_j/m$, *then the algorithm does not abort and* f *is a valid chain partitioning,*
(ii) if the algorithm does not abort, then $C_{\max}(f') \le \max\{2T, p'_{\max}\}$,
(iii) if the algorithm does not abort, no machine gets two jobs longer than T, *even if* $C_{\max}(f') > 2T$.

Proof: For part (i), first we prove by induction that for any $k < m$, if the first k machines did not get all the jobs, then either $\sum_1^k C_i \ge k \cdot T$ or $\sum_1^{k+1} C_i \ge (k+1) \cdot T$ holds. For any machine k, there are two possible cases if the algorithm has not run out of unassigned jobs i.e. there are still jobs for machine $(k + 1)$. In the first case, machine k gets normal amount of work: $C_k \ge T$, proving the case trivially. In the

other case, machine k gets less work: $C_k < T$, but this also implies $C_{k+1} > 2T$, because a job longer than $2T$ comes next in line 4 of Algorithm 2, and the machine $(k+1)$ gets this long job. Thus $C_k + C_{k+1} > 2T$, proving this case. Consequently the last machine could get a work of length at most $\sum_{j=1}^{n'} p'_j - (m-2)T$, proving (i).

For both parts (ii) and (iii), observe that a machine i gets more than $2T$ amount of work only if its assignment consists of one single job, as line 4 assures that each job longer than $2T$ is assigned to a dedicated machine. ■

According to part (ii), FRUGALLYSCHEDULE is not an ordinary 2-approximation. It is of importance, because while $p'_{\max}$ is a lower bound on the optimal makespan of the synthetic scheduling problem P, it is usually not a lower bound for the original scheduling problem, as the longest synthetic job may not correspond to a singleton chain. The algorithm is frugal because it does not allow the makespan to reach $2p'_{\max}$, as an ordinary approximation would.

Proposition 2 enables us to finish the analysis of the framework.

Proposition 2.

(i) In any schedule executed by the framework, the assignment of a machine can contain at most one half chain of estimated length, so the length of the complement of the estimated chain is known.

(ii) $s(q+1) \leq \lceil s(q)/2 \rceil$ *for every iteration* q.

(iii) If $s(q) = 1$, *then* $C_{\max}(f_q) = p_{\max}$, *if* $s(q) = 0$, *then* $C_{\max}(f_q) \leq 2T_{LB}$.

Proof: Part (i) follows from part (iii) of Proposition 1 and the fact that each estimated length is more than T (see line 8 in Algorithm 1). Part (ii) follows from that all long, non-singleton chains in the database are split in half in each iteration. Part (iii) follows from part (ii) of Proposition 1. ■

Finally, we summarize the results in

Theorem 1. *The* PSAPARTITIONER *framework presented in Algorithm 1 yields a makespan at most* $2C^*_{\max}$ *using a database of size* $O(m \log n)$, *after at most* $\lceil \log n \rceil$ *iterations, starting from any initial schedule* f_0.

We note that as the initial schedule can be arbitrary, so if the processing times would change between two iterations because of some change, the converging process is automatically restarted from the current schedule.

V. DEMONSTRATION OF THE FRAMEWORK

In this section we present two PSAs to test the framework with: a simple artificial one (Example 3), and a graph generation program (Example 4).

Example 3. *Let* $m = 3$, $n = 12$ *and the job lengths* $p_{1..12} = 93, 1, 1, 1, 102, 50, 25, 25, 1, 1, 1, 1$. *The initial schedule* f_0 *assigns uniform, 4-sized chains to the machines.*

This example focuses on illustrating the virtual chains of the database, which are more than the last measurement. The average load remains $T_{\text{LB}} = 100$ throughout the iterations. Before the first iteration, initially we have the schedule $f_0[1 \,..\, 4] = 1, f_0[5 \,..\, 8] = 2, f_0[9 \,..\, 12] = 3$ and its completion times $\mathbf{C}(f_0) = (96, 202, 4)$.

As $C_{\max} = 202 > 2T_{\text{LB}} = 200$, the first iteration has to come. The chain $[5 \,..\, 8]$ is split in half, both $\ell([5 \,..\, 6])$ and $\ell([7 \,..\, 8])$ are estimated to be 101, so FRUGALLYSCHEDULE is called with the synthetic jobs $p'_{1..4} = 96, 101, 101, 4$. It returns the schedule of the synthetic jobs $f'_1[1 \,..\, 2] = 1, f'_1[3 \,..\, 4] = 2$, which is interpreted as a schedule (chain partitioning) of the original jobs: $f_1[1 \,..\, 6] = 1, f_1[7 \,..\, 12] = 2$. This is executed, and the completion times are measured as $\mathbf{C}(f_1) = (248, 54)$.

However, at the end of the 1. iteration, the "chains of the database" and their length are: $\ell([1 \,..\, 4]) = 96$, $\ell([5 \,..\, 6]) = 152$, $\ell([7 \,..\, 8]) = 50$, $\ell([9 \,..\, 12]) = 4$, derived from $\mathbf{C}(f_1)$ and $\mathbf{C}(f_0)$. In the 2. iteration no chain of the database needs to be split, as the previous long chain $[1 \,..\, 6]$ is represented already as two chains of the database. So the final schedule is $f_2[1 \,..\, 4] = 1, f_2[5 \,..\, 6] = 2, f_2[7 \,..\, 12] = 3$, the completion times are $\mathbf{C}(f_2) = (96, 152, 54)$, and the framework stops as realises that $C_{\max} = 152 < 2T_{\text{LB}}$.

Example 4 ([36]). *The processing times of the* $n = 49566$ *jobs of a plane graph generation PSA [36] are shown on Fig. 1 (top), and* $m = 12$.

The whole process is illustrated on Fig. 1. The first diagram (on top) visualizes the processing times, which spread from 4 to 8125 (in hundredth seconds). It also shows that the first 10000 jobs ($j \leq 10000$) are longer than the others.

The second diagram of Fig. 1 shows the initial schedule f_0 and its completion times $\mathbf{C}(f_0)$. The initial schedule assigns the uniform-sized chains to the 12 machines and the assignment intervals (chains) are separated by vertical lines, and their machine indices are also displayed. The machine index assigned to a job index j is $f_0(j)$, so $C_{f_0(j)}(f_0)$ is the completion time of the machine that gets job j. There are 12 rectangles on the second diagram, and the width of the ith rectangle corresponds to the *size* (number of jobs) of the assignment of machine i, while its height corresponds to the *length* of this assignment. Obviously the makespan of f_0 is the height of the highest rectangle, which is $C_{\max}(f_0) = 760177$.

The average completion time is $T_{\text{LB}} = 194851$, so the first iteration splits the first and the second chains, estimate their sizes (not shown on the figure) and computes the next schedule f_1, executes it and measures its completion times. f_1 and $\mathbf{C}(f_1)$ are shown on the third diagram. In this iteration, FRUGALLYSCHEDULE only assigned jobs to 8 machines, and the machine index i is only displayed for $5 \leq i \leq 8$. Again, we have 8 rectangles corresponding to 8 machine assignments. It is worth observing that the short chains (lowest rectangles) from the previous schedule f_0 were contracted into bigger chains (wider rectangles), while the longest ones (highest rectangles) were split into smaller (narrower rectangles) chains. The makespan is still $C_{\max} = 433898$, so the next iteration comes.

The second iteration splits only the third chain of f_1, computes and executes the schedule f_2, updates the database with $\mathbf{C}(f_2)$, shown on the fourth, last diagram of Fig. 1. The only difference from the previous iteration is that the one chain split in half is dispatched to two distinct machines. Finally, the

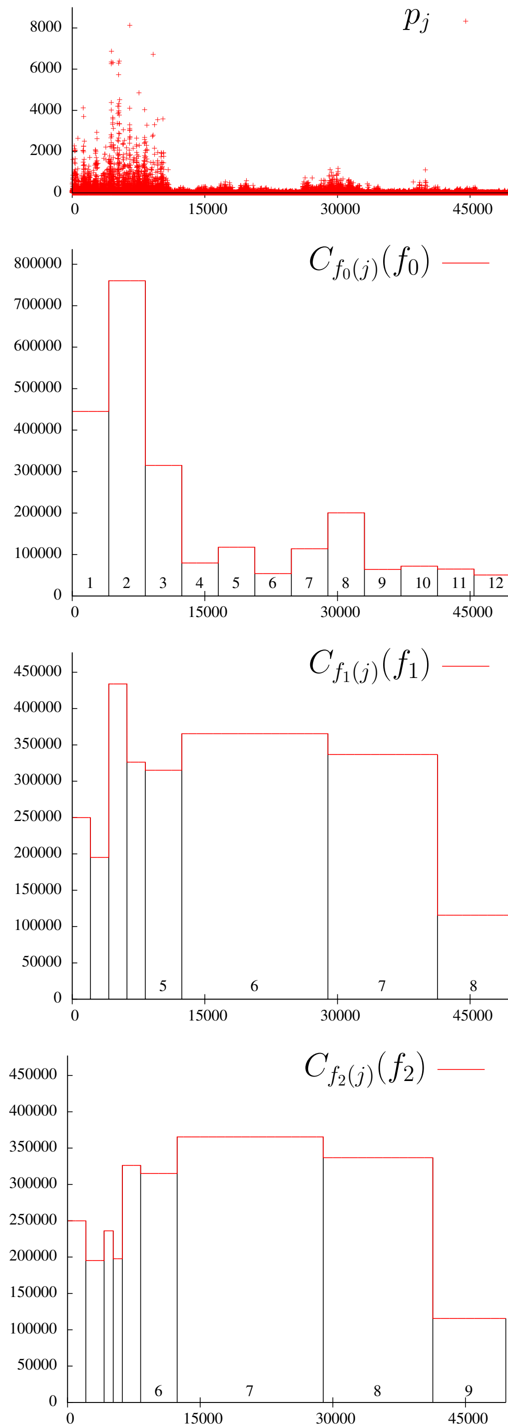

Figure 1. Plane graph generation PSA processing times p_j and machine completion times $\mathbf{C}(f_q)$ of iterations $q = 0, 1, 2$. All horizontal axes correspond to the job number j. Vertical axes denote time.

makespan is $C_{\max} = 365471$, so the framework stops with the satisfying schedule f_2.

VI. Scheduling Batches with Setups

This section extends the framework for the more general application model we call *Batches of PSAs (BPSA)*. A BPSA consists of n jobs similarly as a PSA, though the jobs are partitioned into g *families*. We assume every family consists of jobs of consecutive indices, i.e. the families are chains. Each family $h \in [1 \,..\, g]$ is divided into one or more *batches* [8] in a schedule, preceded by the corresponding setup of length u_h, where u_h depends only on the family. So the work of a machine consists of batches of jobs and corresponding setups, and its completion time has to be redefined as the total length of the jobs and setups assigned. The more batches are divided a family into, the more additional setups are contributed to the total work. As chain partitioning keeps family members close, it might be an economical approach to mitigate costs from having extra setup work. We suppose the setup lengths $u_1, \ldots, u_g$ (denoted by the vector $\mathbf{u}$) are *known a priori*, but the job lengths are still unknown.

The extended framework presented in Algorithm 3 is a generalization of the iterative framework shown by Algorithm 1 in Section III. In each iteration, first it calculates a preliminary schedule (chain partitioning), which may not be feasible yet because it misses some setups (to be inserted later). Let us pretend for a moment that we have only a single machine, i.e. $m = 1$. Then the only feasible chain partitioning is that we put the setup of the first family of length u_1, then every job of the first family, then continue with a setup and the jobs of the second family etc. We regard this machine assignment, i.e. the sequence of $(n + g)$ jobs and setups as a *fictitious PSA*, and the framework computes the chain partitioning of the fictitious PSA first. We call this partitioning $f : [1 \,..\, n + g] \mapsto [1 \,..\, m]$ the *preliminary schedule*, which is not a feasible schedule if a family is split up between machines. To fix it, we need to simply insert the corresponding *additional setup* into each assignment not starting with a setup, then we have the feasible *fixed schedule*, illustrated on Fig. 2. A similar idea was used in [9].

The chains of the database now form a partition of $[1 \,..\, n + g]$ and used in the preliminary schedule. However, the measured machine completion times are feedback on the fixed schedule instead, so a conversion is needed: the length of the chain of the database is equal to the completion time minus the length of the additional setup. This easy conversion is done when initializing the database from completion times (line 1) and when merging the new completion times (line 13). We mention an implementation issue: there may be a redundant setup in the end of assignments but since we know the setup lengths, we are not required to execute the redundant setup, instead, we can simply add its length to the measured completion time.

Finally, we need to adjust the former lower bound T_{LB} and the maximum size of the long chains s, hence we introduce $T^u_{\text{LB}} := T_{\text{LB}} + \sum_{h=1}^{g} u_h / m$ which is now the average completion time of the fictitious PSA, and the definition of $s^u(q)$ is

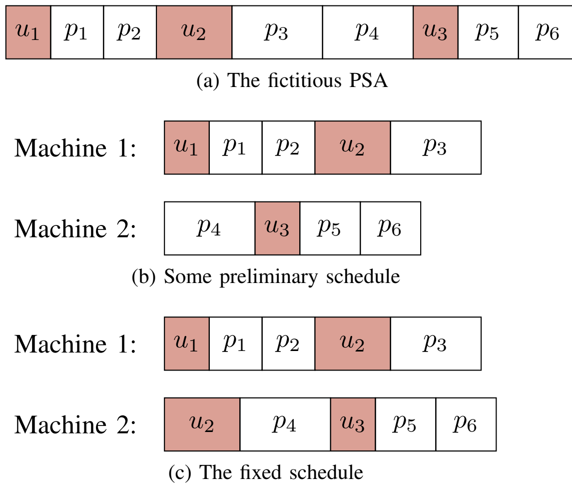

Figure 2. Fixing a schedule with additional setups, $g = 3$, $n = 6$, $m = 2$.

Algorithm 3 *Iterative framework for batches*

1: **procedure** BATCHPARTITIONER$(m, n, \mathbf{u}, f_0, \mathbf{C}(f_0))$
2: $q \leftarrow 0$, calculate T^u_{LB}
3: **while** $s^u(q) > 1$ **do**
4: $q \leftarrow q + 1$
5: $P \leftarrow$ a synthetic scheduling problem of the chains of the database and their length
6: **for all** original chain c in P s.t. $\ell(c) > 2T^u_{\text{LB}}$ **do**
7: Split chain c in half
8: Estimate the length of the half chains
9: Replace chain c with the half chains in P
10: $f_q \leftarrow$ FRUGALLYSCHEDULE(P, T^u_{LB})
11: Fix f_q: add the corresponding setups
12: Execute f_q, measure $\mathbf{C}(f_q)$
13: Merge $\mathbf{C}(f_q)$ into the database
14: Recalculate T^u_{LB}

similar to $s(q)$ but uses T^u_{LB} as a threshold instead of T_{LB}, and counts the setups in as well.

The extended framework copes with the constraint of setups, and in the special case when $\mathbf{u} \equiv 0$ it behaves exactly like the previous framework. However, for the batch case it admits only a 3-approximation as shown by

Theorem 2. *The* BATCHPARTITIONER *framework presented in Algorithm 3 yields a makespan at most* $3C^*_{\max}$ *using a database of size* $O(m \log n)$*, after at most* $\lceil \log n + g \rceil$ *iterations, starting from any initial schedule* f_0.

Proof: Proposition 1 is still applicable in addition to the claims (i) and (ii) of Proposition 2. So the makespan of the preliminary schedule in the last iteration is either $\max\{p_{\max}; u_{\max}\}$ if $s^u(q) = 1$ or less than $2T^u_{\text{LB}}$ if $s^u(q) = 0$, i.e. the preliminary schedule is a 2-approximation, although infeasible.

We base the rest of the proof on the observation that in order to fix the preliminary schedule, at most one setup has to be inserted into each machine assignment in line 11 in Algorithm 3. It is the consequence of the construction of the fictitious PSA that when a machine switches from one family to another, there already is a corresponding setup in the preliminary schedule. Thus the completion times of the fixed schedule are made longer than the preliminary completion times with at most $u_{\max}$, respectively. In conclusion, as the preliminary is a 2-approximation, the fixed schedule is a 3-approximation. ■

VII. OUTLOOK AND CONCLUDING REMARKS

We presented a framework to schedule independent jobs of unknown length *a priori*. The framework iteratively learns the necessary length information to achieve an approximation by repeatedly adjusting an initial scheduling, executing it and updating its historical database. In each iteration the framework approaches to its goal, the information on the most dense parts of the parameter space is refined, and an upper bound on the schedule ($p'_{\max}$) monotone decreases. We showed that the framework after $\log n$ iterations (execution of schedules) yields a schedule with a makespan at most 2 times the optimal makespan, using only $O(m \log n)$ space for the historical database. The framework was tested with a PSA and was considered an adequate tool to automatically aid the user to reduce the cost of schedule. We also mention that a small perturbation in the time characteristics of the PSA probably does not destroys the convergence.

To the best knowledge of the authors, it is an open problem whether this framework can be strictly improved in the sense that neither the framework approximation ratio (2), nor the convergence rate ($\log n$) and the space used ($m \log n$) degrades.

A batch setting where the setups would also be unknown a priori seems to be rather a calling open problem.

Another direction of possible future work could be generalizing this really simple model towards a heterogeneous environment or more complex job interactions. However, both the chain partitioning for the case of heterogeneous machines [37], and the application of using multi-dimensional chain partitioning [38] is NP-hard.

REFERENCES

[1] A. B. Downey and D. G. Feitelson, "The elusive goal of workload characterization," *SIGMETRICS Performance Evaluation Review*, vol. 26, no. 4, pp. 14–29, 1999.

[2] P. Dóbé, R. Kápolnai, A. Sipos, and I. Szeberényi, "Applying the improved Saleve framework for modeling abrasion of pebbles," in *Int. Conf. on Large-Scale Scientific Computing*, ser. LNCS, vol. 5910. Springer, 2010, pp. 467–474.

[3] C. Papadimitriou, "Algorithms, games, and the internet," in *ACM Symposium on Theory of Computing*. ACM, 2001, pp. 749–753.

[4] R. Graham, E. Lawler, J. Lenstra, and A. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," in *Discrete Optimization II*, ser. Annals of Discrete Mathematics, E. J. P.L. Hammer and B. Korte, Eds. Elsevier, 1979, vol. 5, pp. 287–326.

[5] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems theoretical and practical results," *J. ACM*, vol. 34, no. 1, pp. 144–162, 1987.
[6] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical J.*, vol. 45, no. 9, pp. 1563–1581, 1966.
[7] A. Archer and E. Tardos, "Truthful mechanisms for one-parameter agents," in *IEEE Symposium on Foundations of Computer Science*, 2001, pp. 482–491.
[8] A. Allahverdi, C. Ng, T. Cheng, and M. Y. Kovalyov, "A survey of scheduling problems with setup times or costs," *European J. Operational Research*, vol. 187, no. 3, pp. 985–1032, 2008.
[9] T. Kis and R. Kápolnai, "Approximations and auctions for scheduling batches on related machines," *Operation Research Letters*, vol. 35, no. 1, pp. 61–68, January 2007.
[10] T.-P. Hong, P.-C. Sun, and S.-D. Li, "A heuristic algorithm for the scheduling problem of parallel machines with mold constraints," in *Int. Conf. on Applied Computer an Applied Computational Science*. WSEAS, 2008, pp. 242–247.
[11] E. K. Dimitris Dranidis, "A production scheduling strategy for an assembly plant based on reinforcement learning," in *WSES Int. Conf. on Circuits, Systems, Communications and Computers*. WSEAS, 2001.
[12] S. Bokhari, "Partitioning problems in parallel, pipeline, and distributed computing," *IEEE Transactions on Computers*, vol. 37, no. 1, pp. 48–57, 1988.
[13] S. Khanna, S. Muthukrishnan, and S. Skiena, "Efficient array partitioning," in *Int. Colloquium on Automata, Languages and Programming*, ser. LNCS. Springer, 1997, vol. 1256, pp. 616–626.
[14] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1D partitioning," *J. Parallel and Distributed Computing*, vol. 64, no. 8, pp. 974–996, 2004.
[15] W. Herroelen and R. Leus, "Project scheduling under uncertainty: Survey and research potentials," *European J. Operational Research*, vol. 165, no. 2, pp. 289–306, 2005.
[16] H. Aytug, M. A. Lawley, K. McKay, S. Mohan, and R. Uzsoy, "Executing production schedules in the face of uncertainties: A review and some future directions," *European J. Operational Research*, vol. 161, no. 1, pp. 86–110, 2005.
[17] X. Guo, M. Boubekeur, J. McEnery, and D. Hickey, "Acet based scheduling of soft real-time systems: An approach to optimise resource budgeting," *Int. J. of Computers and Communications*, vol. 1, no. 3, pp. 82–89, 2007.
[18] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in heterogeneous systems via weighted factoring," in *ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1996, pp. 318–328.
[19] D. A. Lifka, "The ANL/IBM SP scheduling system," in *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, ser. LNCS. Springer, 1995, vol. 949, pp. 295–303.
[20] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, 2003.
[21] H. Dail, F. Berman, and H. Casanova, "A decoupled scheduling approach for grid application development environments," *J. Parallel and Distributed Computing*, vol. 63, no. 5, pp. 505–524, 2003.
[22] R. L. Daniels and J. E. Carrillo, "β-robust scheduling for single-machine systems with uncertain processing times," *IIE Transactions*, vol. 29, pp. 977–985, 1997.
[23] F. Dong and S. G. Akl, "Scheduling algorithms for grid computing: State of the art and open problems," School of Computing, Queen's University, Kingston, Ontario, Tech. Rep. 2006-504, 2006.
[24] M. Chtepen, B. Dhoedt, F. De Turck, P. Demeester, F. Claeys, and P. Vanrolleghem, "Adaptive checkpointing in dynamic grids for uncertain job durations," in *Int. Conf. on Information Technology Interfaces*, 2009, pp. 585–590.
[25] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2009, pp. 75–84.
[26] T. N. Minh and L. Wolters, "Using historical data to predict application runtimes on backfilling parallel systems," in *Euromicro Conf. on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 246–252.
[27] K. Gao, M. Liu, K. Chen, N. Zhou, and J. Chen, "Sampling-based tasks scheduling in dynamic grid environment," in *Int. Conf. on Simulation, Modeling and Optimization*. WSEAS, 2005, pp. 25–30.
[28] K. Gao, Z. Chen, and W. Zhong, "Evaluating performance of grid based on soft computing," in *Int. Conf. on E-ACTIVITIES*. WSEAS, 2007.
[29] D. Tsafrir, Y. Etsion, and D. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 6, pp. 789–803, 2007.
[30] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron, "Dynamic heterogeneous scheduling decisions using historical runtime data," in *Workshop on Applications for Multi- and Many-Core Processors*, San Jose, CA, 2011.
[31] J. Gehring and A. Reinefeld, "MARS—A framework for minimizing the job execution time in a metacomputing environment," *Future Generation Computer Systems*, vol. 12, no. 1, pp. 87–99, 1996.
[32] E. Heymann, M. Senar, E. Luque, and M. Livny, "Adaptive scheduling for master-worker applications on the computational grid," in *IEEE/ACM Int. Workshop on Grid Computing*, ser. LNCS. Springer, 2000, vol. 1971, pp. 214–227.
[33] G. Wrzesinska, J. Maassen, and H. E. Bal, "Self-adaptive applications on the grid," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2007, pp. 121–129.

[34] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *IEEE/ACM Int. Symposium on Microarchitecture*. ACM, 2009, pp. 45–55.

[35] R. Kápolnai, I. Szeberényi, and B. Goldschmidt, "Approximation of repeated scheduling chains of independent jobs of unknown length based on historical data," in *Recent Advances in Computer Science*. WSEAS Press, 2013, pp. 41–46.

[36] R. Kápolnai, G. Domokos, and T. Szabó, "Generating spherical multiquadrangulations by restricted vertex splittings and the reducibility of equilibrium classes," to appear in Periodica Polytechnica Electrical Engineering.

[37] A. Pınar, E. Kartal Tabak, and C. Aykanat, "One-dimensional partitioning for heterogeneous systems: Theory and practice," *J. Parallel and Distributed Computing*, vol. 68, no. 11, pp. 1473–1486, 2008.

[38] M. Grigni and F. Manne, "On the complexity of the generalized block distribution," in *Int. Workshop on Parallel Algorithms for Irregularly Structured Problems*, ser. LNCS. Springer, 1996, vol. 1117, pp. 319–326.