Malleable Scheduling for Flows of Jobs and Applications to MapReduce

Viswanath Nagarajan[*] Joel Wolf[†] Andrey Balmin[‡] Kirsten Hildrum[§]

Abstract

This paper provides a unified family of algorithms with performance guarantees for malleable scheduling problems on flows. A flow represents a set of jobs with precedence constraints. Each job has a speedup function that governs the rate at which work is done on the job as a function of the number of processors allocated to it. In our setting, each speedup function is linear up to some job-specific processor maximum. A key aspect of malleable scheduling is that the number of processors allocated to any job is allowed to vary with time. The overall objective is to minimize either the total cost (minisum) or the maximum cost (minimax) of the flows. Our approach handles a very general class of cost functions, and in particular provides the first constant-factor approximation algorithms for total and maximum weighted completion time. Our motivation for this work was scheduling in MapReduce and we also provide experimental evaluations that show good practical performance.

1 Introduction

MapReduce is a fundamentally important programming paradigm for processing big data [9]. This allows for efficiently processing large-scale tasks in a computing cluster. There are a number of different MapReduce implementations: a very popular open-source implementation is *Hadoop* [19]. An important component of any MapReduce implementation is its *scheduler*, which allocates work to different processors in the cluster. A good scheduler is essential in utilizing available computing resources well.

There has been a significant amount of work [42, 43, 44, 45] on designing practical MapReduce schedulers: however, all these papers focus on *singleton* MapReduce jobs. Indeed, single MapReduce jobs were the appropriate atomic unit of work early on. Lately, however, *flows* of interconnected MapReduce jobs are commonly employed. Each flow corresponds to a directed acyclic graph whose nodes are singleton MapReduce jobs and whose arcs represent precedence constraints. Such a MapReduce flow can result, for example, from a single user-level Pig [16], Hive [39] or Jaql [2] query. In these settings, it is the completion times of the flows that matter rather than the completion times of the individual MapReduce jobs.

In this paper, we model the problem of scheduling flows of MapReduce jobs as a malleable scheduling problem with precedence constraints [12]. Because the terminology of parallel scheduling is not standard, we clarify what we mean by the word malleable now. In malleable scheduling, jobs can be executed on an arbitrary number of processors and this number can vary throughout the runtime of the job. A different parallel scheduling model known as moldable scheduling also involves jobs that can be executed on an arbitrary number of processors:

[*]Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor MI 48109, viswa@umich.edu

[†]IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, jlwolf@gmail.com

[‡]Platfora Inc., abalmin@gmail.com

[§]IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, hildrum@gmail.com

however, the number of processors used for any job *can not* vary over time. Some papers such as [5, 18, 22, 23, 28, 30] refer to moldable scheduling also as malleable: we emphasize that the scheduling model in these papers is different from ours.

We provide a unified family of algorithms for malleable scheduling with provable performance guarantees for minimizing either the sum or the maximum of cost functions associated with each flow. Our approach simultaneously handles a variety of standard scheduling objectives, such as weighted completion time, number of tardy jobs, weighted tardiness and stretch. Apart from the theoretical results, we provide two sets of experimental evaluations comparing our algorithm to other practical MapReduce schedulers: (i) simulations based on our model that compare the objective value of each of these schedulers to the optimum; and (ii) real cluster experiments that compare the relative performance of these schedulers. The experimental results demonstrate good practical performance of our algorithms. Our algorithms have also been incorporated in IBM's *BigInsights* [4].

1.1 The Model

There are P identical *processors* in our scheduling environment. Each *flow* j is described by means of a directed acyclic graph. The nodes in each of these directed acyclic graphs are *jobs*, and the directed arcs correspond to precedence relations. We use the standard notation $i_1 \prec i_2$ to indicate that job i_1 must be completed before job i_2 can begin. Each job i must perform a fixed amount of work s_i (also referred to interchangeably as the job *size* or *area*), and can be performed on a maximum number $\delta_i \in [P]$ of processors at any point in time. (Throughout the paper, for any integer $\ell \geq 1$, we denote by $[\ell]$ the set $\{1, \ldots, \ell\}$.) We consider jobs with linear speedup through their maximum numbers of processors: the rate at which work is done on job i at any time is proportional to the number of processors $p \in [\delta_i]$ allocated to it. Job i is complete when s_i units of work have been performed.

We are interested in *malleable schedules*. In this setting, a schedule for job i is given by a function $\tau_i : [0, \infty) \to \{0, 1, \ldots, \delta_i\}$ where $\int_{t=0}^{\infty} \tau_i(t)\, dt = s_i$. Note that this corresponds to both linear speedup and processor maxima. We denote the *start time* of schedule τ_i by $S(\tau_i) := \arg\min\{t \geq 0 : \tau_i(t) > 0\}$; similarly the *completion time* is denoted $C(\tau_i) := \arg\max\{t \geq 0 : \tau_i(t) > 0\}$. A schedule for flow j (consisting of jobs I_j) is given by a set $\{\tau_i : i \in I_j\}$ of schedules for its jobs, where $C(\tau_{i_1}) \leq S(\tau_{i_2})$ for all $i_1 \prec i_2$. The completion time of flow j is $\max_{i \in I_j} C(\tau_i)$, the maximum completion time of its jobs. Our algorithms make use of the following two natural and standard lower bounds on the minimum possible completion time of a single flow j. (See, for example, [12].)

- Squashed area: $\frac{1}{P} \sum_{i \in I_j} s_i$.
- Critical path: the maximum of $\sum_{r=1}^{\ell} \frac{s_{i_r}}{\delta_{i_r}}$ over all chains $i_1 \prec \cdots \prec i_\ell$ in flow j. (Recall that a *chain* in a directed acyclic graph is any sequence of nodes that lie on some directed path.)

Each flow j also specifies an arbitrary non-decreasing *cost function* $w_j : \mathbb{R}_+ \to \mathbb{R}_+$ where $w_j(t)$ is the cost incurred when flow j is completed at time t. We consider both *minisum* and *minimax* objective functions. The minisum (resp. minimax) objective minimizes the sum (resp. maximum) of the cost functions over all flows. In the notation of [12, 29] this scheduling environment is $P|var, p_i(k) = \frac{p_i(1)}{k}, \delta_i, prec|*$; here *var* stands for malleable scheduling, $p_i(k) = \frac{p_i(1)}{k}$ denotes linear speedup, δ_i is processor maxima and *prec* denotes precedence. The * denotes general cost functions, though we should point out that our cost functions are always based on the completion times of the flows rather than the jobs. We refer to these problems collectively as *precedence constrained malleable scheduling*. Our cost model handles all the commonly used scheduling objectives: weighted average completion time, makespan (maximum completion time), average and maximum stretch, and deadline-based objectives associated with number of tardy jobs, service level agreements (SLAs) and so on. (*Stretch* is a fairness objective in which each flow weight is the reciprocal of the size of the flow. SLA cost functions are sometimes called Post Office metrics.) Figure 1 illustrates 4 basic types of cost functions.

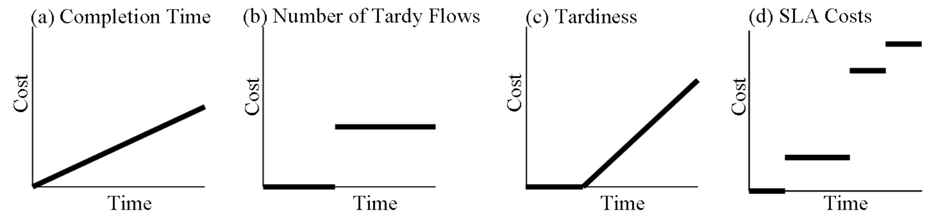

Figure 1: Typical Cost Functions Types.

Recall that a polynomial time algorithm for a minimization problem is said to be an α-*approximation algorithm* (for a value $\alpha \geq 1$) if it always produces a solution with objective value at most α times the optimal. It is known [13] that unless P=NP there are no finite approximation ratios for either the minisum or minimax malleable scheduling problems defined above (even in the special case of chains with length three). In order to circumvent this hardness, we use resource augmentation [24] and focus on *bicriteria* approximation guarantees, defined as follows.

Definition 1 *A polynomial time algorithm for a scheduling problem is said to be an* (α, β)*-bicriteria approximation (for values* $\alpha, \beta \geq 1$*) if it produces a schedule using* β *speed processors that has objective value at most* α *times the optimal (under unit speed processors).*

1.2 Our Results

For minisum objectives we have the following:

Theorem 1 *The precedence constrained malleable scheduling problem admits a* $(2, 3)$*-bicriteria approximation algorithm for general minisum objectives. Hence we obtain (i)* 6*-approximation algorithm for total weighted completion time (which includes total stretch) and (ii)* $(3 \cdot 2^{1/p})$*-approximation algorithm for* ℓ_p*-norm of completion times.*

This approach also provides a smooth tradeoff between the approximation ratios in the two criteria: cost and speed. For any value $\alpha \in (0, 1)$ we obtain a $\left(\frac{1}{1-\alpha}, 1 + \frac{1}{\alpha}\right)$-bicriteria approximation algorithm. By optimizing the parameter α, we can obtain a slightly better 5.83-approximation for weighted completion time.

For minimax objectives we have:

Theorem 2 *The precedence constrained malleable scheduling problem admits a* $(1, 2)$*-bicriteria approximation algorithm for general minimax objectives. Hence we obtain a* 2*-approximation algorithm for maximum weighted completion time (which includes makespan and maximum stretch).*

Our minisum algorithm in Theorem 1 requires solving a minimum cost network flow problem. Although minimum cost flow can be solved in polynomial time and there are theoretically efficient exact [8] as well as approximation algorithms [14], these algorithms are too complex for our implementation in the MapReduce setting. Therefore, we provide an alternative simpler approximation algorithm for minisum scheduling that does not rely on a network flow solver.

Theorem 3 *There is a simple* $(1 + o(1), 6)$*-bicriteria approximation algorithm for precedence constrained malleable scheduling with any minisum objective.*

By modifying a parameter used in this algorithm, we can obtain a better guarantee of 5.83 in the speedup required; for simplicity we focus mainly on the slightly weaker result in Theorem 3. The approximation guarantee in Theorem 3 is incomparable to that in Theorem 1. We note however that for weighted completion time, both algorithms provide the same approximation ratio (even after optimizing their parameters).

An interesting consequence of Theorem 3 is for the special case of *uniform* minisum objectives, where each flow has the same cost function $w : \mathbb{R}_+ \to \mathbb{R}_+$. Examples of such objectives include total completion time and the sum of p^{th} powers of the completion times. We show that our algorithm finds a "universal" schedule that is simultaneously near-optimal for *all* uniform minisum cost functions w.

Theorem 4 *There is an algorithm for precedence constrained malleable scheduling under uniform minisum objectives that given any instance, produces a single schedule which is simultaneously a* $(1 + o(1), 6)$*-bicriteria approximation for all objectives.*

The minimax algorithm (Theorem 2) and the simpler minisum algorithm (Theorem 3) are implemented in our MapReduce scheduler. We provide two types of experimental results under various standard scheduling objectives. Both experiments compare our scheduler to two other commonly used MapReduce schedulers, namely *FIFO* (which schedules flows in their arrival order) and *Fair* (which essentially divides processors equally between all flows). We note that these experiments may be somewhat unfair to *FIFO* and *Fair* since they are each agnostic with respect to particular objective functions. Nevertheless, we think this comparison is useful since *FIFO* and *Fair* are the most commonly employed practical MapReduce schedulers. (They are both implemented in Hadoop.) Moreover, as Theorem 4 shows, for a subclass of objectives our algorithm is also agnostic to the specific objective.

The first set of experiments is based on random instances and compares the performance of each of these algorithms relative to lower bounds on the optimum, which we compute during the algorithm. On most minisum objectives, the average performance of our algorithm is within 52% of these lower bounds. And for most minimax objectives, the average performance of our algorithm is within 12% of these lower bounds. In all cases, our algorithm performs much better than *FIFO* and *Fair*.

The second set of experiments is based on an implementation in a real computer cluster. So this does not rely on any assumptions of our model. The input here consists of MapReduce jobs from an existing benchmark [17] which are represented as flows by adding random precedence constraints. We tested the three schedulers (ours, *FIFO* and *Fair*) on the following four standard objectives: average completion time, average stretch, makespan and maximum stretch. The performance of our algorithm was at least 50% better than both *FIFO* and *Fair* on all objectives except makespan. (For makespan, all three schedulers produced schedules of almost the same objective.)

We note that some of these results (Theorems 1 and 2) were reported without full proofs in [33]. That paper focused on MapReduce system aspects rather than theory. Other than providing detailed proofs of Theorems 1 and 2, this paper provides the following new results. (i) A faster and simpler approximation algorithm for minisum objectives (Theorem 3), (ii) a near-optimal universal schedule for all uniform minisum objectives (Theorem 4), and (iii) an improved approximation ratio of 5.83 for total weighted completion time (Corollary 1).

1.3 Related Work

In order to place our scheduling problem in its proper context, we give a brief, somewhat historically oriented overview of theoretical parallel scheduling. There are essentially three different models in parallel scheduling: *rigid*, *moldable* and *malleable*.

The first parallel scheduling results involved *rigid* jobs. Each such job runs on some fixed number of processors and each processor is presumed to complete its work simultaneously. One can thus (with a slight loss of accuracy) think of a job as corresponding to a rectangle whose height corresponds to the number of processors p, whose width corresponds to the execution time t of the job, and whose area $s = p \cdot t$ corresponds to the work performed by the job. Early papers, such as [1, 7, 15, 37], focused on the makespan objective and obtained constant-factor approximation algorithms.

Subsequent parallel scheduling research took a variety of directions. One such direction involved *moldable* scheduling: each job here can be run on an arbitrary number p of processors, but with an execution time $t(p)$ which is a function of the number of processors. (One can assume without loss of generality that this function is nonincreasing.) Thus the height of a job is turned from an input parameter to a decision variable. And the rectangle is moldable in the sense that pulling it higher also has the effect of shrinking its width. Clearly rigid scheduling is a special case of moldable scheduling. The first approximation algorithms for moldable scheduling with a makespan objective appeared in [30, 40].

In a different direction, [36] found the first approximation algorithm for both rigid and moldable scheduling problems with a (weighted) average completion time objective.

The notion of *malleable* scheduling is more general than moldable. Here the number of processors allocated to a job is allowed to vary over time. However, each job must still perform a fixed total amount of work. In its most general variant, there is a speedup function (as in the moldable case) which governs the rate at which work is done as a function of the number of allocated processors; so the total work completed is the integral of these rates over time. However, this general problem is very difficult, and so the literature to date [10, 11, 12, 29, 41] has focused on the special case where the speedup function is linear through a given maximum number of processors, and constant thereafter. It turns out that malleable scheduling with linear speedup and processor maxima captures the MapReduce paradigm very well. See the discussion in Subsection 1.4. This is the setting considered in our paper as well.

In the presence of precedence constraints, there are a large number of papers, eg. [5, 18, 22, 23, 28], dealing with *moldable* jobs and the makespan objective. None of these results are directly applicable to *malleable* jobs considered here.

Aside from the negative result in [13], the literature on malleable scheduling is sparse, even in this special case of linear speedup functions and processor maxima. (The problem without processor maxima reduces to the case of single processor scheduling, for which the literature is well-known [34].) The problem of minimizing maximum lateness with linear speedup, processor maxima and release dates was solved in polynomial time in [41] via an iterative maximum flow algorithm based on guesses for the lateness. (The complexity can be improved using techniques of [27].) A polynomial time algorithm for the online problem of minimizing makespan with linear speedup and processor maxima appears in [10, 11].

Again, see [12, 29] for many more details on rigid, moldable and malleable scheduling. The literature on the last is quite limited, and thus this paper is a contribution.

Commonly Used MapReduce schedulers. To the best of our knowledge, all previous schedulers were designed for *singleton* MapReduce jobs. The first scheduler was the ubiquitous *First In, First Out (FIFO)* which prioritizes jobs in their arrival order. *FIFO* is obviously very simple to implement, but it causes large jobs to starve small jobs that arrive even a small time later. This unfairness motivated the second scheduler, called *Fair* [44]. The idea in *Fair* was essentially to allocate the cluster resources as equally as possible among *all* the current jobs. The third scheduler called *Flex* [42, 43] is closest to our work, and is also based on the malleable scheduling model used here. We note that our work handles a much more general setting with *flows* of MapReduce jobs. Moreover we obtain algorithms with theoretical performance guarantees; previously such results were not known even for singleton MapReduce jobs.

1.4 Application to MapReduce

MapReduce [9] is an extensively used parallel programming paradigm. There are many good reasons for the widespread adoption of MapReduce. Most are related to MapReduce's inherent simplicity of use, even when applied to large applications and installations. For example, MapReduce work is designed to be parallelized automatically. It can be implemented on large computer clusters, and it inherently scales well. Scheduling (based on a choice of plugin schedulers), fault tolerence and necessary communications are all handled automatically, without direct user assistance. The MapReduce paradigm is sufficiently generic to fit many big data problems. Finally, and perhaps most importantly, the programming of MapReduce applications is relatively straight-forward, and thus appropriate for less sophisticated programmers. These benefits, in turn, result in lower costs.

MapReduce jobs, consist, as the name implies, of two processing phases: Map and Reduce. Each phase is broken into multiple independent tasks, the nature of which depends on the phase. In the Map phase the tasks consist of the steps of scanning and processing (extracting information) from equal-sized blocks of input data. Each block is typically replicated on disks for availability and performance reasons. The output of the Map phase is a set of key-value pairs. These intermediate results are also stored on disk. There is a shuffle step in which all relevant data from all Map phase output is transmitted to the Reduce phase. Each Reduce task corresponds to a partitioned subset of keys (from the key-value pairs). The Reduce phase consists of a sort step and finally a processing step, which may consist of transformation, aggregation, filtering and/or summarization.

Why does scheduling in MapReduce fit the theory of malleable scheduling with linear speedup and processor maxima so neatly? One reason is that there is a natural decoupling of MapReduce scheduling into an *Allocation* Layer followed by an *Assignment* Layer. In the Allocation Layer, quantity decisions are made, i.e. the number of processors assigned to each MapReduce job. The Assignment Layer then uses the allocation decision as a guideline to assign individual Map/Reduce tasks to processors. The scheduling algorithms discussed in this paper reside in the Allocation Layer. The Assignment Layer works locally at each processor in the cluster. Whenever a task completes on a processor, the Assignment Layer essentially determines which job is most underallocated according to the Allocation Layer schedule, and assigns a new task from that job to this processor. (Occasionally, the Assignment Layer overrides this rule due to data locality concerns. But the deviation from the allocation decision is usually small. We do not discuss these details here since our model and algorithms are for the Allocation Layer.)

Second, in MapReduce clusters each processing node is partitioned into a number of *slots*, typically a small constant times the number of cores in the processor. Processors with more compute power are given more slots than those with less power, the idea being to create slots of roughly equal capabilities, even in a heterogeneous environment. In MapReduce, the slot is the atomic unit of allocation. So the word processor in the theoretical literature corresponds to the word slot in a MapReduce context. (We will continue to talk of processors here.)

Third, both the Map and Reduce phases are composed of *many small, independent* tasks.

Because they are independent they do not need to start simultaneously and can be processed with any degree of parallelism without significant overhead. This, in turn, means that the jobs will have nearly linear speedup. Because the tasks are many and small, the decisions of the scheduler can be approximated closely.

One final reason is that processor maxima constraints occur naturally, either because the particular job happens to be small (and thus have only few tasks), or at the end of a normal job, when only a few tasks remain to be allocated.

Other Models for MapReduce We note that a number of other models for MapReduce have been considered in the theoretical literature. Moseley et al. [32] consider a "two-stage flexible flow shop" [38] model. Berlinska and Drozdowski [3] use "divisible load theory" to model a single MapReduce job and its communication details. Theoretical frameworks for MapReduce computation have been proposed in [25, 26]. Compared to our setting, these models are at a finer level of granularity, that of individual Map and Reduce tasks. Our model, as described above, decouples the quantity decisions (allocation) from the actual assignment details in the cluster. We focus on obtaining algorithms for the allocation layer, which is abstracted as a precedence constrained malleable scheduling problem.

1.5 Paper Outline

The rest of the paper is organized as follows. Section 2 contains our algorithmic framework for both minisum and minimax objectives. Section 3 describes our simpler algorithm for minisum objectives. Section 4 provides results from our experimental evaluations. We conclude in Section 5.

2 Algorithms for Minisum and Minimax Scheduling

In this section, we provide our general algorithms for minisum and minimax objectives (Theorems 1 and 2). In Subsection 2.1 we give a high-level overview of the algorithms, which consist of three main stages. The following three subsections then contain the details of each of these stages.

2.1 Technical Overview

Our algorithms for both minisum and minimax objectives are based on suitable "reductions" to the problem with a *deadline-based* objective. In the deadline-based scheduling problem, every flow j is associated with a deadline $d_j \in \mathbb{R}_+$ such that its cost function is:

$$w_j(t) = \begin{cases} 0 & \text{if } t \le d_j \\ \infty & \text{otherwise} \end{cases} .$$

Notice that in this case both minisum and minimax versions coincide. Even the *deadline-based* malleable scheduling problem is NP-hard and hard to approximate. (See Subsection 2.3 for more details.) So we focus on obtaining a bicriteria approximation algorithm. In particular we show that a simple greedy scheme has a $(1, 2)$-bicriteria approximation ratio.

The reduction from minisum objectives to deadline-based objectives is based on solving a *minimum cost flow* relaxation and "rounding" the optimal flow solution, which incurs some loss in the approximation ratio. The reduction from minimax objectives to deadlines is much simpler and uses a "guess and verify" framework that is implemented via a bracket and bisection search.

Our algorithms have three sequential stages, described at a high level as follows.

1. *Converting general precedence to chains.* First we consider each flow j separately, and convert its precedence constraints into *chain* precedence constraints. (Recall that a chain precedence on elements $\{e_i : 1 \leq i \leq n\}$ is just a total order, say $e_1 \prec e_2 \prec \cdots \prec e_n$.) In order to do this, we create a *pseudo-schedule* for each flow that assumes an infinite number of processors, but respects precedence constraints and the bounds δ_i on jobs i. Then we partition the pseudo-schedule into a chain of *pseudo-jobs*, where each pseudo-job k corresponds to an interval in the pseudo-schedule with uniform processor usage. Just like the original jobs, each pseudo-job k specifies a size s_k and a bound δ_k describing the maximum number of processors on which it can be executed. We note that (unlike jobs) the bound δ_k of a pseudo-job may be larger than P.

2. *Scheduling chains of pseudo-jobs.* Next we design bicriteria approximation algorithms for the malleable scheduling problem when each flow is a chain. This stage relies on the above-mentioned reductions from general minisum/minimax objectives to a deadline-based objective. Then we have a malleable schedule for the pseudo-jobs, satisfying the chain precedence within each flow as well as the bounds δ_k.

3. *Converting the pseudo-schedule into a valid schedule for jobs.* The final stage transforms the malleable schedule of each pseudo-job k into a malleable schedule for the (portions of) jobs i that comprise it. This step also ensures that the original precedence constraints and bounds δ_i on jobs are satisfied. The algorithm used here is a generalization of an old scheduling algorithm [31].

2.2 General Precedence Constraints to Chains

We now describe a procedure to convert an arbitrary set of precedence constraints on jobs into a chain constraint on "pseudo-jobs". Consider any flow with n jobs where each job $i \in [n]$ has size s_i and processor bound δ_i. The precedence constraints are given by a directed acyclic graph on the jobs. In the algorithm, we make use of the squashed area and critical path lower bounds on the minimum completion time of a flow.

Construct a *pseudo-schedule* for the flow as follows. Allocate each job $i \in [n]$ its maximal number δ_i of processors, and assign job i the smallest *start time* $b_i \geq 0$ such that for all $i_1 \prec i_2$ we have $b_{i_2} \geq b_{i_1} + \frac{s_{i_1}}{\delta_{i_1}}$. The start times $\{b_i\}_{i=1}^n$ can be easily computed by dynamic programming. The pseudo-schedule runs each job i on δ_i processors, between time b_i and $b_i + \frac{s_i}{\delta_i}$. Given an infinite number of processors the pseudo-schedule is a valid schedule satisfying precedence constraints.

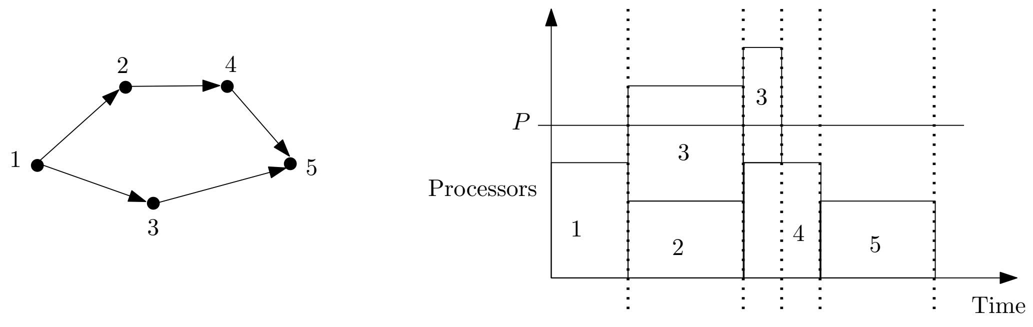

Figure 2: Converting flows into chains.

Next, we construct *pseudo-jobs* corresponding to this flow. Let $T = \max_{i=1}^n (b_i + \frac{s_i}{\delta_i})$ denote the completion time of the pseudo-schedule; observe that T equals the critical path bound of the flow. Partition the time interval $[0, T]$ into maximal intervals $I_1, \dots, I_h$ so that the set of jobs processed by the pseudo-schedule in each interval stays fixed. Note that $\sum_{k=1}^{h} |I_k| = T$. For each $k \in [h]$, if r_k denotes the total number of processors being used during I_k, define pseudo-job k to have processor bound $\delta(k) := r_k$ and size $s(k) := r_k \cdot |I_k|$, which is the total work done by the pseudo-schedule during I_k. Note that a pseudo-job consists of portions of work from multiple jobs; moreover, we may have $r_k > P$, since the pseudo-schedule is defined independent of P. Finally we enforce the chain precedence constraint $1 \prec 2 \prec \cdots \prec h$ on pseudo-jobs. Notice that the squashed area and critical path lower bounds remain the *same* when computed in terms of pseudo-jobs instead of jobs. Clearly, the total size of pseudo-jobs $\sum_{k=1}^{h} s(k) = \sum_{i=1}^{n} s_i$, the total size of the jobs. Moreover, there is only one maximal chain of pseudo-jobs, which has critical path $\sum_{k=1}^{h} \frac{s(k)}{\delta(k)} = \sum_{k=1}^{h} |I_k| = T$, the original critical path bound. Note that pseudo-jobs can be easily constructed in polynomial time, and the number of pseudo-jobs resulting from any flow is at most the number of original jobs in the flow.

See Figure 2 for an example. On the left is the directed acyclic graph of a particular flow, and on the right is the resulting pseudo-schedule along with its decomposition into maximal intervals.

2.3 Malleable Scheduling with Chain Precedence Constraints

Here we consider the malleable scheduling problem on P parallel processors with *chain* precedence constraints and general cost functions. Each chain $j \in [m]$ is a sequence $k_1^j \prec k_2^j \prec \cdots \prec k_{n(j)}^j$ of *pseudo-jobs*, where each pseudo-job k has size $s(k)$ and specifies a maximum number $\delta(k)$ of processors on which it can be run. We note that the $\delta(k)$s may be larger than P. Each chain $j \in [m]$ also specifies a non-decreasing *cost function* $w_j : \mathbb{R}_+ \to \mathbb{R}_+$ where $w_j(t)$ is the cost incurred when chain j is completed at time t. The objective is to find a malleable schedule on P identical parallel processors that satisfies precedence constraints and minimizes the total or maximum cost. Recall that in the original malleable scheduling problem, each flow corresponds to a set of jobs with arbitrary precedence constraints: the above chain of pseudo-jobs is obtained as a result of the transformation in Subsection 2.2. Our algorithm here relies on the chain structure.

Malleable schedules for pseudo-jobs (resp. chains of pseudo-jobs) are defined identically to jobs (resp. flows) as in Subsection 1.1. To reduce notation, we denote a malleable schedule for *chain* j by a sequence $\tau^j = \langle \tau_1^j, \dots, \tau_{n(j)}^j \rangle$ of schedules for its pseudo-jobs, where τ_r^j is a malleable schedule for pseudo-job k_r^j for each $r \in [n(j)]$. Note that chain precedence implies that for each $r \in \{1, \dots, n(j) - 1\}$, the start time of k_{r+1}^j, $S(\tau_{r+1}^j) \geq C(\tau_r^j)$, the completion time of k_r^j. The completion time of this chain is $C(\tau^j) := C(\tau_{n(j)}^j)$.

Even very special cases of this problem do not admit any finite approximation ratio:

Theorem 5 ([13]) *Unless P=NP, there is no finite approximation ratio for precedence constrained malleable scheduling, even with chain precedences of length three.*

Proof: This follows directly from the NP-hardness of the makespan minimization problem called $P|1any1, pmtn|C_{max}$ of [13]. We state their result in our context: each chain is of length three, where the first and last pseudo-jobs have maximum $\delta = 1$, and the middle pseudo-job has maximum $\delta = P$. Then it is NP-hard to decide whether there is a malleable schedule of makespan equal to the squashed area bound (denoted M).

We create an instance of precedence constrained malleable scheduling as follows. There are P processors and the same set of chains. The cost function of each chain j is $w_j : \mathbb{R}_+ \to \mathbb{R}_+$ where $w(t) = 0$ if $t \leq M$ and $w(t) = 1$ if $t > M$. Clearly, the optimal cost of this malleable scheduling instance is zero if and only if the instance of $P|1any1, pmtn|C_{max}$ has a makespan M schedule; otherwise the optimal cost is one. Therefore it is also NP-hard to obtain *any* multiplicative approximation guarantee for precedence constrained malleable scheduling. ■

Given this hardness of approximation, we focus on bicriteria approximation guarantees. We first give a $(1, 2)$-bicriteria approximation algorithm when the cost functions are deadline-based. Then we obtain a $(2, 3)$-bicriteria approximation algorithm for arbitrary minisum objectives and a $(1, 2)$-bicriteria approximation algorithm for arbitrary minimax objectives.

2.3.1 Deadline-based Objective

We consider the problem of scheduling chains on P parallel processors under a deadline-based objective. That is, each chain $j \in [m]$ has a *deadline* d_j and its cost function is: $w_j(t) = 0$ if $t \leq d_j$ and ∞ otherwise.

We show that a natural greedy algorithm is a good bicriteria approximation. By renumbering chains, we assume that $d_1 \leq \cdots \leq d_m$. The algorithm schedules chains in non-decreasing order of deadlines, and within each chain it schedules pseudo-jobs greedily (by allocating the maximum possible number of processors). A formal description appears as Algorithm 1.

Algorithm 1 Algorithm for scheduling with deadline-based objective

```
 1: initialize utilization function $\sigma : [0, \infty) \to \{0, 1, \dots, P\}$ to zero.
 2: for $j = 1, \dots, m$ do
 3:   for $i = 1, \dots, n(j)$ do
 4:     set $S(\tau_i^j) \leftarrow C(\tau_{i-1}^j)$ and initialize $\tau_i^j : [0, \infty) \to \{0, \dots, P\}$ to zero.
 5:     for each time $t \geq S(\tau_i^j)$ (in increasing order), set $\tau_i^j(t) \leftarrow \min\left\{P - \sigma(t), \delta(k_i^j)\right\}$, until
        $\int_{t \geq S(\tau_i^j)} \tau_i^j(t)\, dt = s(k_i^j)$.
 6:     set $C(\tau_i^j) \leftarrow \max\{z : \tau_i^j(z) > 0\}$.
 7:     update function $\sigma \leftarrow \sigma - \tau_i^j$.
 8:   set $C(\tau^j) \leftarrow C(\tau_{n(j)}^j)$.
 9:   if $C(\tau^j) > 2 \cdot d_j$ then
10:     instance is *infeasible.*
11:   else
12:     output schedules $\{\tau_j : j \in [m]\}$.
```

Theorem 6 *There is a $(1, 2)$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints and a deadline-based objective.*

Proof: We obtain this result by analyzing Algorithm 1. First, notice that this algorithm produces a valid malleable schedule that respects the chain precedence constraints and the maximum processor bounds. Next, we prove the performance guarantee. It suffices to show that if there is any solution that satisfies deadlines $\{d_\ell\}_{\ell=1}^m$ then $C(\tau^j) \leq 2d_j$ for all chains $j \in [m]$. Consider the utilization function $\sigma : [0, \infty) \to \{0, \dots, P\}$ just after scheduling chains $[j]$ in the algorithm. Let A_j denote the total duration of times t in the interval $[0, C(\tau^j)]$ where $\sigma(t) = P$, i.e. all processors are busy with chains from $[j]$; and $B_j = C(\tau^j) - A_j$ the total duration of times when $\sigma(t) < P$. Note that A_j and B_j consist of possibly many non-contiguous intervals. It is clear that

$$A_j \quad \leq \quad \frac{1}{P} \sum_{\ell=1}^{j} \sum_{i=1}^{n(\ell)} s(k_i^\ell). \tag{1}$$

Since the algorithm always allocates the maximum possible number of processors to each pseudo-job, at each time t with $\sigma(t) < P$ we must have $\tau_i^j(t) = \delta(k_i^j)$, where $i \in [n(j)]$ is the unique index of the pseudo-job with $S(\tau_i^j) \le t < C(\tau_i^j)$. Therefore,

$$B_j \quad \le \quad \sum_{i=1}^{n(j)} \frac{s(k_i^j)}{\delta(k_i^j)}. \tag{2}$$

Notice that the right hand side in (1) corresponds to the squashed area bound of the *first j chains*, which must be at most d_j if there is any feasible schedule for the given deadlines. Moreover, the right hand side in (2) is the critical path bound of chain j, which must also be at most d_j if chain j can complete by time d_j. Combining these inequalities, we have $C(\tau^j) = A_j + B_j \le 2 \cdot d_j$.

Thus, if the processors are run at twice their speeds, we obtain a solution that satisfies all deadlines. This proves the $(1, 2)$-bicriteria approximation guarantee. ■

2.3.2 Minisum Objectives

We consider the problem of scheduling chains on P parallel processors under arbitrary minisum objectives. Recall that there are m chains, each having a non-decreasing cost function $w_j : \mathbb{R}_+ \to \mathbb{R}_+$, where $w_j(t)$ is the cost of completing chain j at time t. The total number of pseudo-jobs is denoted N. The goal in the minisum problem is to compute a schedule of minimum total cost. We obtain the following bicriteria approximation in this case.

Theorem 7 *There is a $(2, 3+o(1))$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under arbitrary minisum cost objectives.*

For each chain $j \in [m]$, define

$$Q_j := \max\left\{\sum_{i=1}^{n(j)} s(k_i^j)/\delta(k_i^j),\ \sum_{i=1}^{n(j)} s(k_i^j)/P\right\} \tag{3}$$

to be the maximum of the critical path and squashed area lower bounds. We may assume, without loss of generality, that every schedule for these chains completes by time $H := 2m \cdot \lceil \max_j Q_j \rceil$. In order to focus on the main ideas, we first provide an algorithm where:

A1. Each cost function $w_j(\cdot)$ has integer valued breakpoints, i.e. times where the cost changes.

A2. The running time is pseudo-polynomial, i.e. polynomial in m, N and H.

We will show later that both these restrictions can be removed to obtain a truly polynomial (in m, N and $\log H$) time algorithm for any set of cost functions.

Our algorithm works in two phases. In the first phase, we treat each chain simply as a certain volume of work, and formulate a *minimum cost flow* subproblem using the cost functions w_js. The solution to this subproblem is used to determine candidate deadlines $\{d_j\}_{j=1}^m$ for the chains. Then in the second phase, we run our algorithm for deadline-based objectives using $\{d_j\}_{j=1}^m$ to obtain the final solution.

Minimum cost flow. Here, we treat each chain $j \in [m]$ simply as work of volume $V_j := \sum_{i=1}^{n(j)} s(k_i^j)$, which is the total size of pseudo-jobs in j. Recall that a network flow instance consists of a directed graph (V, E) with designated source (r) and sink (r') nodes. Each arc $e \in E$ has a capacity u_e and cost w_e (per unit of flow). There is also a demand of ρ units. A *network flow* is an assignment $f : E \to \mathbb{R}_+$ of values to arcs such that (i) for any node

$v \in V \setminus \{r, r'\}$, the total flow entering v equals the total flow leaving v, and (ii) for each arc $e \in E$, $f_e \le u_e$. The value of a network flow is the net flow out of its source. The objective is to find a flow f of value ρ having minimum cost $\sum_{e\in E} w_e \cdot f_e$. It is well known that this problem can be solved in polynomial time.

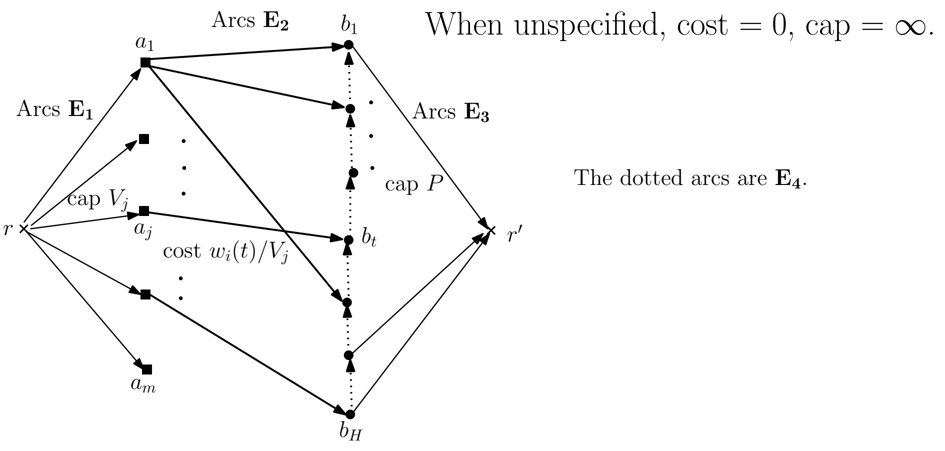

Figure 3: The Minimum Cost Flow Network.

The min-cost flow subproblem. The nodes of our flow network are $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_H\} \cup \{r, r'\}$, where r denotes the source and r' the sink. The nodes a_js correspond to chains and b_ts correspond to intervals $[t-1, t)$ in time. The arcs are $E = E_1 \cup E_2 \cup E_3 \cup E_4$, where:

$$E_1 := \{(r, a_j) : j \in [m]\}, \quad \text{arc } (r, a_j) \text{ has cost 0 and capacity } V_j,$$

$$E_2 := \{(a_j, b_t) \ : \ j \in [m], t \in [H], t \ge Q_j\}, \quad \text{arc } (a_j, b_t) \text{ has cost } \frac{w_j(t)}{V_j} \text{ and capacity } \infty,$$

$$E_3 := \{(b_t, r') : t \in [H]\}, \quad \text{arc } (b_t, r') \text{ has cost 0 and capacity } P, \text{ and}$$

$$E_4 = \{(b_{t+1}, b_t) : t \in [H-1]\}, \quad \text{arc } (b_{t+1}, b_t) \text{ has cost 0 and capacity } \infty.$$

See also Figure 3. We set the demand $\rho := \sum_{j=1}^{m} V_j$, and compute a minimum cost flow $f : E \to \mathbb{R}_+$. We use $\mathcal{I}$ to denote this network flow instance. Notice that, by definition of the arc capacities, any ρ-unit flow must send exactly V_j units through each node a_j ($j \in [m]$).

The next claim relates instance $\mathcal{I}$ to the malleable scheduling instance.

Claim 1 *The minimum cost of a network flow in instance $\mathcal{I}$ is at most the optimal value of the malleable scheduling instance.*

Proof: Consider any feasible malleable schedule having completion time C_j for each chain $j \in [m]$. By definition, Q_j is a lower bound for chain j, i.e. $C_j \ge Q_j$ and hence edge $(a_j, b_{C_j}) \in E_2$ for all $j \in [m]$. We will prove the existence of a feasible network flow of ρ units having cost at most $\sum_{j=1}^{m} w_j(C_j)$. Since the cost functions $w_j(\cdot)$ are monotone, it suffices to show the existence of a feasible flow of ρ units (no costs) in the sub-network N' consisting of edges $E_1 \cup E_2' \cup E_3 \cup E_4$, where $E_2' := \{(a_j, b_t) \ : \ j \in [m], \ Q_j \le t \le C_j\}$. By max-flow min-cut duality, it now suffices to show that the minimum $r - r'$ cut in this network N' is at least ρ. Observe that any finite capacity $r - r'$ cut in N' is of the form $\{r\} \cup \{a_j : j \in S\} \cup \{b_t : 1 \le t \le \max_{j\in S} C_j\}$, where $S \subseteq [m]$ is some subset of chains. The capacity of edges crossing such a cut is:

$$\sum_{j \notin S} V_j \ + \ P \cdot \max_{j\in S} C_j. \tag{4}$$

Notice that all the chains in S are completed by time $T := \max_{j\in S} C_j$ in the malleable schedule: so the total work assigned to the first T time units is at least $\sum_{j\in S} V_j$. On the other hand, the malleable schedule only has P processors: so the total work assigned to the first T time units must be at most $P\cdot T$. Hence $P\cdot \max_{j\in S} C_j \ge \sum_{j\in S} V_j$. Combined with (4) this implies that the minimum cut in N' is at least $\sum_{j=1}^m V_j = \rho$. This completes the proof. ■

Obtaining candidate deadlines Now we round the flow f to obtain deadlines d_j for each chain $j \in [m]$. We define $d_j := \arg\min \left\{t : \sum_{s=1}^t f(a_j, b_s) \ge V_j/2\right\}$, for all $j\in[m]$. In other words, d_j corresponds to the "half completion time" of chain j given by the network flow f. Since $w_j(\cdot)$ is non-decreasing and $\sum_{t\ge d_j} f(a_j,b_t) \ge V_j/2$, we have

$$w_j(d_j) \quad\le\quad 2\cdot \sum_{t\ge d_j} \frac{w_j(t)}{V_j}\cdot f(a_j,b_t), \qquad \forall j\in[m]. \tag{5}$$

Note that the right hand side above is at most twice the cost of arcs leaving node a_j. Thus, if we obtain a schedule that completes each chain j by its deadline d_j, using (5) the total cost $\sum_{j=1}^m w_j(d_j) \le 2\cdot \mathsf{OPT}$. Moreover, by definition of the arcs E_2,

$$d_j \quad\ge\quad Q_j \quad\ge\quad \text{(critical path of chain } j), \qquad \forall j\in[m]. \tag{6}$$

By the arc capacities on E_3 we have $\sum_{j=1}^m \sum_{s=1}^t f(a_j,b_s) \le P\cdot t$, for all $t\in[H]$.

Let us renumber the chains in deadline order so that $d_1 \le d_2 \le \cdots \le d_m$. Then, using the definition of deadlines (as half completion times) and the above inequality for $t = d_j$,

$$\sum_{\ell=1}^{j} V_\ell \quad\le\quad 2\cdot \sum_{\ell=1}^{j}\sum_{s=1}^{d_j} f(a_\ell,b_s) \quad\le\quad 2P\cdot d_j, \qquad \forall j\in[m]. \tag{7}$$

Solving the subproblem with deadlines. Now we apply the algorithm for scheduling with a deadline-based objective (Theorem 6) using the deadlines $\{d_j\}_{j=1}^m$ computed above. Notice that we have the two bounds required in the analysis of Theorem 6:

- The squashed area of the first j chains is $\frac{1}{P}\cdot \sum_{\ell=1}^j V_\ell \le 2\cdot d_j$ for all $j\in[m]$ by (7).
- The critical path bound is at most d_j for all $j\in[m]$, by (6).

By an identical analysis, it follows that the algorithm in Theorem 6 produces a malleable schedule that completes each chain j by time $3\cdot d_j$. So, running this schedule using processors three times faster results in total cost at most $\sum_{j=1}^m w_j(d_j) \le 2\cdot\mathsf{OPT}$.

Handling restrictions A1 and A2. We now show that both restrictions made earlier can be removed, while incurring an additional $1+o(1)$ factor in the processor speed. Recall the definitions of lower bounds Q_j for chains $j\in[m]$, and the horizon $H = 2m\cdot\lceil \max_j Q_j\rceil$. By scaling up sizes, we may assume (without loss of generality) that $\min_j Q_j \ge 1$. So the completion time of any chain in any schedule lies in the range $[1,H]$. Set $\epsilon := 1/m$, and partition the $[1,H]$ time interval as:

$$T_\ell \;:=\; \left[(1+\epsilon)^{\ell-1},\,(1+\epsilon)^{\ell}\right], \qquad \text{for all } \ell = 1,\ldots,\log_{1+\epsilon} H.$$

Note that the number of parts above is $R := \log_{1+\epsilon} H$ which is polynomial. We now define a polynomial size network on nodes $\{a_1,\ldots,a_m\}\cup\{b_1,\ldots,b_R\}\cup\{r,r'\}$, where r denotes the source and r' the sink. The nodes a_js correspond to chains and b_ℓs correspond to time intervals T_ℓs. The arcs are $E = E_1\cup E_2\cup E_3\cup E_4$, where:

$$E_1 := \{(r,a_j) : j\in[m]\}, \quad \text{arc } (r,a_j) \text{ has cost } 0 \text{ and capacity } V_j,$$

$$E_2 := \left\{(a_j,b_\ell)\ :\ j\in[m],\ell\in[R],Q_j\le(1+\epsilon)^\ell\right\},\quad \text{arc } (a_j,b_\ell) \text{ has cost } w_j\left((1+\epsilon)^{\ell-1}\right)/V_j \text{ and capacity } \infty,$$

$$E_3 := \{(b_\ell,r') : \ell\in[R]\},\quad \text{arc } (b_\ell,r') \text{ has cost 0 and capacity } |T_\ell|\cdot P, \text{ and}$$

$$E_4 = \{(b_{\ell+1},b_\ell) : \ell\in[H-1]\},\quad \text{arc } (b_{\ell+1},b_\ell) \text{ has cost 0 and capacity } \infty.$$

Above, $|T_\ell| = \epsilon\cdot(1+\epsilon)^{\ell-1}$ denotes the length of interval T_ℓ. As before, we set the demand $\rho := \sum_{j=1}^m V_j$, and compute a minimum cost flow $f : E\to\mathbb{R}_+$. Notice that any ρ-unit flow must send exactly V_j units through each node a_j ($j\in[m]$). Exactly as in Claim 1 we can show that this network flow instance is a valid relaxation of any malleable schedule. The next two steps of computing deadlines and solving the subproblem with deadlines are also the same as before. The only difference is that the squashed area (7) and critical path (6) lower bounds are now larger by a $1+\epsilon$ factor, due to the definition of intervals T_ℓs.

Thus the algorithm is a $(2,3(1+\epsilon))$-bicriteria approximation, which proves Theorem 7.

Tradeoff between speed and objective. We can use the technique of α-point rounding (see eg. [6]) and choose deadlines in the network flow f based on "partial completion times" other than just the halfway point used above. This leads to a continuous tradeoff between the approximation bound on the speed and objective.

Theorem 8 *For any $\alpha\in(0,1)$ there is a $\left(\frac{1}{1-\alpha},1+\frac{1}{\alpha}\right)$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under any minisum objective.*

This algorithm generalizes that in Theorem 7 by selecting deadlines as:

$$d_j := \arg\min\left\{t\ :\ \sum_{s=1}^{t} f(a_j,b_s)\ge\alpha\cdot V_j\right\},\quad \forall j\in[m].$$

By an identical analysis we obtain

$$w_j(d_j)\quad\le\quad\frac{1}{1-\alpha}\cdot\sum_{t\ge d_j}\frac{w_j(t)}{V_j}\cdot f(a_j,b_t),\qquad \forall j\in[m],$$

in place of (5). And

$$\sum_{\ell=1}^{j} V_\ell\quad\le\quad\frac{1}{\alpha}\cdot\sum_{\ell=1}^{j}\sum_{s=1}^{d_j} f(a_\ell,b_s)\quad\le\quad\frac{1}{\alpha}P\cdot d_j,\qquad \forall j\in[m],$$

in place of (7). As before we also have the bound (6) on critical paths. Combining these bounds with the algorithm for deadlines (Theorem 6) we obtain a malleable schedule with speed $1+\frac{1}{\alpha}$ that has cost at most $\frac{1}{1-\alpha}$ times the optimum. This proves Theorem 8.

We note that in some cases the bicriteria approximation guarantees can be combined.

Corollary 1 *There is a $3+2\sqrt{2}\approx 5.83$ approximation algorithm for minimizing total weighted completion time in malleable scheduling with chain precedence constraints.*

Proof: This follows directly by observing that if a $1+\frac{1}{\alpha}$ speed schedule is executed at unit speed then each completion time scales up by exactly this factor. Therefore, the algorithm from Theorem 8 for any value of α yields a $\left(\frac{1+\alpha}{\alpha-\alpha^2}\right)$-approximation algorithm for weighted completion time. Optimizing for $\alpha\in(0,1)$ gives the result (choosing $\alpha=\sqrt{2}-1$). ■

We also have the following result for ℓ_p-norm objectives.

Corollary 2 *There is a $(3 \cdot 2^{1/p})$-approximation algorithm for minimizing the ℓ_p-norm of completion times in malleable scheduling with chain precedence constraints.*

Proof: Recall that for a schedule with completion times $\{C_j\}_{j=1}^m$, the ℓ_p-norm objective equals $\left(\sum_{j=1}^m C_j^p\right)^{1/p}$. We use the minisum cost function $w_j(t) = t^p$ in Theorem 7. The algorithm in Theorem 7 then gives a $(2, 3)$-bicriteria approximation for the minisum objective $\sum_{j=1}^m C_j^p$. Viewed as a unit speed schedule this is a $(2 \cdot 3^p)$-approximation, and hence for the ℓ_p norm objective we obtain the claimed $(3 \cdot 2^{1/p})$-approximation algorithm. ■

2.3.3 Minimax Objectives

Here we obtain the following result.

Theorem 9 *There is a $(1, 2+o(1))$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under arbitrary minimax cost objectives.*

Proof: The algorithm assumes a bound M such that $M \leq \mathsf{OPT} \leq (1+\epsilon)M$ for some small $\epsilon > 0$ and attempts to find a schedule of minimax cost at most M. The final algorithm performs a bracket and bisection search on M and returns the solution corresponding to the smallest feasible M. (This is a common approach to many minimax optimization problems, for example [20].) As with the minisum objective in Theorem 7, our algorithm here also relies on a reduction to deadline-based objectives. In fact the algorithm here is much simpler:

1. *Obtaining deadlines.* Define for each chain $j \in [m]$, its deadline $D_j := \arg\max\{t : w_j(t) \leq M\}$.
2. *Solving deadline subproblem.* We run the algorithm for deadline-based objectives (Theorem 6) using these deadlines $\{D_j\}_{j=1}^m$. If the deadline algorithm declares infeasibility, our estimate M is too low; otherwise we obtain a 2-speed schedule having minimax cost M.

Setting $\epsilon = 1/m$, the binary search on M requires $O(m \log(w_{max}/w_{min}))$ iterations where w_{max} (resp. w_{min}) is the maximum (resp. minimum) cost among all chains. So the overall runtime is polynomial. ■

As in Corollaries 1 and 2, the bicriteria guarantees can be combined for some objectives, including makespan.

Corollary 3 *There is a 2-approximation algorithm for minimizing maximum weighted completion time in malleable scheduling with chain precedence constraints.*

2.4 Converting Pseudo-Job Schedule into a Valid Schedule

The final stage converts any malleable schedule of chains of pseudo-jobs into a valid schedule of the original instance (consisting of flows of jobs). We convert the schedule of each pseudo-job k separately. Recall that each pseudo-job consists of portions of jobs. We will construct a malleable schedule for these job-portions that has the same cumulative processor-utilization as the schedule for pseudo-job k. The original precedence constraints are satisfied since the chain constraints are satisfied on pseudo-jobs, and the jobs participating in any single pseudo-job are independent.

Consider any pseudo-job k that corresponds to an interval I_k in the pseudo-schedule of some flow. (Recall Subsection 2.2.) Let pseudo-job k consist of portions of the jobs $S \subseteq [n]$; then the processor maximum of pseudo-job k is $r_k = \sum_{i \in S} \delta_i$. See Figure 4. The left side shows an example of a pseudo-job. Consider also any malleable schedule $\sigma : [0, \infty) \to \{0, 1, \cdots, r_k\}$ of pseudo-job k; note that this schedule has area $\int \sigma(t)t = s_k = |I_k| \cdot r_k$. We now describe how to "partition" this schedule σ into a set of schedules for the portions of jobs in S corresponding to I_k.

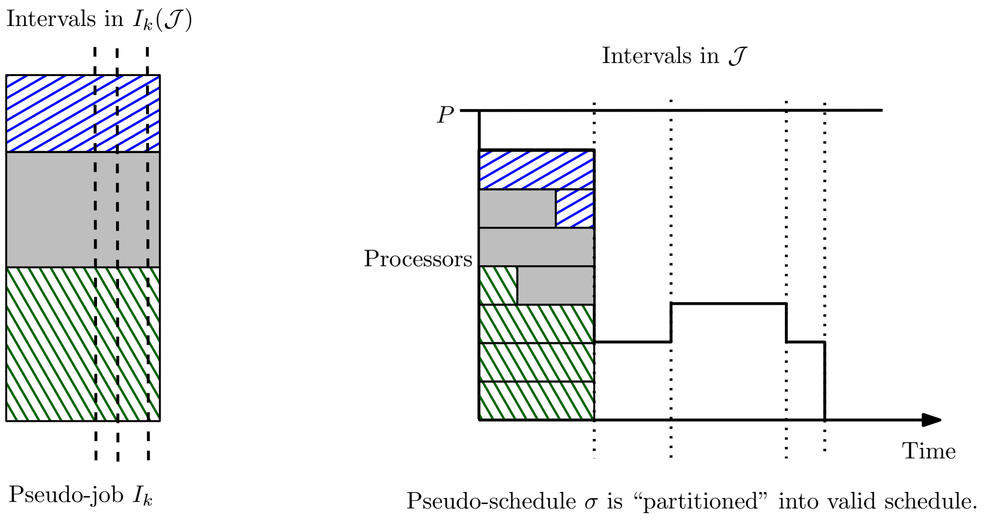
Intervals in $I_k(\mathcal{J})$
Pseudo-job I_k
Intervals in $\mathcal{J}$
P
Processors
Time
Pseudo-schedule σ is "partitioned" into valid schedule.

Figure 4: Converting pseudo-schedule into valid schedule.

The algorithm first decomposes σ into maximal intervals of time $\mathcal{J}$ each of which involves a constant number of processors. For each interval $J \in \mathcal{J}$, we let $|J|$ denote its length and $\sigma(J)$ denote the number of processors used during J. So the work done by σ during $J \in \mathcal{J}$ is $\sigma(J) \cdot |J|$. Based on $\mathcal{J}$ we partition the interval I_k into $\{I_k(J) : J \in \mathcal{J}\}$, where each $|I_k(J)| = \frac{|J| \cdot \sigma(J)}{r_k}$; note that this is indeed a partition as $\sum_{J \in \mathcal{J}} |I_k(J)| = \sum_{J \in \mathcal{J}} \frac{|J| \cdot \sigma(J)}{r_k} = \frac{s_k}{r_k} = |I_k|$. See Figure 4.

Next, the algorithm schedules the work from each interval $I_k(J)$ of the pseudo-job during interval J of schedule σ. Taking such schedules over all $J \in \mathcal{J}$ gives a full schedule for I_k. For each $J \in \mathcal{J}$, we apply McNaughton's Rule [13, 31] to find a valid schedule for $I_k(J)$ which consists of portions of jobs S. This schedule will use $\sigma(J)$ processors for $|J|$ units of time. We consider the jobs in S in arbitrary order, say $i_1, i_2, \cdots, i_{|S|}$. For each job i_ℓ we allocate its work to one processor at a time (moving to the next processor when the previous one has been utilized fully) until i_ℓ is fully allocated; then we continue allocating the next job $i_{\ell+1}$ from the same point. It is easy to see that each job $i \in S$ is allocated at most δ_i processors at any point in time. So we obtain a valid malleable schedule for the job-portions in $I_k(J)$. See the right side of Figure 4 for an example of this valid schedule for the first interval in $\mathcal{J}$.

3 Faster Algorithm for Minisum Scheduling

Consider again the malleable minisum scheduling problem with chain precedence constraints as in Subsection 2.3. There are P parallel processors and m chains. Each chain j has a non-decreasing cost function $w_j : \mathbb{R}_+ \to \mathbb{R}_+$, where $w_j(t)$ is the cost of completing chain j at time t. The goal in the minisum problem is to compute a schedule of minimum total cost. Here we provide a simpler bicriteria approximation algorithm for this problem. The high level idea is to iteratively solve a knapsack-type problem to obtain deadlines for the chains. Then we will apply the algorithm for deadline-based objectives (Theorem 6). For each chain j, recall from (3) that Q_j is a lower bound on its completion time; let R_j and U_j denote the critical path and squashed area bounds, so $Q_j = \max\{R_j, U_j\}$. Let $\ell = \min_{j\in[m]} Q_j$ denote a lower bound on the completion time of *any* chain. We will focus on the following geometrically spaced times:

$$a_i := \ell \cdot 2^i, \quad \text{for all } i \geq 0. \tag{8}$$

We also define for each chain $j \in [m]$, the following incremental costs:

$$\Delta_{ji} := \begin{cases} w_j(a_0) & \text{if } i = 0 \\ w_j(a_i) - w_j(a_{i-1}) & \text{if } i \geq 1 \end{cases}, \quad \forall i \geq 0.$$

Note that $w_j(a_i) = \sum_{k=0}^{i} \Delta_{jk}$ for all chains $j \in [m]$ and $i \geq 0$.

Our algorithm relies on solving many instances of a knapsack-type problem. An instance of the *min-loss knapsack* problem consists of (i) a set T of items where item $j \in T$ has size v_j and profit p_j, and (ii) a budget B. The goal is to select a subset $S \subseteq T$ that has total size $\sum_{j\in S} v_j \leq B$ and minimizes the "unselected" profit $\sum_{j\in T\setminus S} p_j$. This problem can be reduced to the minimization knapsack problem: given items T with profits and sizes as above and some target B', the goal is to select a minimum profit subset S' of items such that the total size in S' is at least B'. The reduction involves an instance of minimum knapsack with items T, profits p_j, sizes v_j and target $B' = \sum_{j\in T} v_j - B$. Every solution $S' \subseteq T$ to the minimum knapsack instance corresponds to a solution $S = T \setminus S'$ to the min-loss knapsack instance with the same objective, and vice-versa. Since there is a *fully polynomial time approximation scheme* (FPTAS) for minimum knapsack [21], we obtain one for min-loss knapsack as well. In particular, we have an $\alpha = 1 + o(1)$ approximation algorithm for min-loss knapsack.

Algorithm 2 is our simpler approximation algorithm for minisum objectives. We will show:

Theorem 10 *If there is an α-approximation algorithm for the minimum knapsack problem then there is an $(\alpha, 6)$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under arbitrary minisum objectives.*

Using the FPTAS for minimum knapsack [21] and the framework in Section 2 which reduces general precedence constraints to chain precedence, this also implies Theorem 3.

Algorithm 2 Simpler Algorithm for Minisum Objectives

```
1: initialize unscheduled chains $T \leftarrow [m]$.
2: for $i = 0, 1, \ldots$ do
3:     set $T_i \leftarrow \{j \in T : R_j \leq a_i\}$, i.e. chains whose critical path bound is at most $a_i$.
4:     define an instance of min-loss knapsack as follows. The items correspond to chains $j \in T_i$
       each of which has size $V_j$ (total size of jobs in $j$) and profit $\Delta_{ji}$. The budget on size is
       $P \cdot a_i$.
5:     let $S_i \subseteq T_i$ be an $\alpha$-approximately optimal solution to this instance of min-loss knapsack.
6:     set $T \leftarrow T \setminus S_i$.
7:     if $T = \emptyset$ then break.
8: run the algorithm from Theorem 6 with deadline $d_j := a_i$ for all $j \in S_i$.
```

Analysis. Fix any optimal solution to the given instance. For any $i \geq 0$ let N_i be the set of chains which are *not* completed in the optimal solution before time a_i.

Claim 2 *The optimal cost* $\mathsf{OPT} \geq \sum_{i\geq 0} \sum_{j\in N_i} \Delta_{ji}$.

Proof: Note that $\sum_{i\geq 0}\sum_{j\in N_i}\Delta_{ji} = \sum_{j\in[m]}\sum_{i:j\in N_i}\Delta_{ji}$. And for any chain $j\in[m]$, its completion time C^*_j is at least $\max\{a_i : j\in N_i\}$. So we have $w_j(C^*_j)\geq\sum_{i:j\in N_i}\Delta_{ji}$. The claim now follows by adding over all $j\in[m]$. ■

Consider any iteration $i\geq 0$ in our algorithm. Recall that S_i (for any $i\geq 0$) is the set of chains with deadline a_i. Let $T'_i = \cup_{k\geq i}S_i$ denote the set of chains which have deadline at least a_i; note that T'_i equals the set T at the start of iteration i. And $T_i\subseteq T'_i$ are those chains whose critical path bounds are at most a_i. The next claim shows that the incremental cost of chains $T'_i\setminus S_i$ that have deadline more than a_i is not much more than that of the chains in N_i (for the optimal solution).

Claim 3 *For each $i\geq 0$, $\sum_{j\in T_i\setminus S_i}\Delta_{ji}\leq\alpha\cdot\sum_{j\in T_i\cap N_i}\Delta_{ji}$; hence $\sum_{j\in T'_i\setminus S_i}\Delta_{ji}\leq\alpha\cdot\sum_{j\in N_i}\Delta_{ji}$.*

Proof: Let $O_i = T_i\setminus N_i$. Since all chains in O_i complete before time a_i in the optimal solution, we have $\sum_{j\in O_i}V_j\leq P\cdot a_i$. Hence O_i is a feasible solution to the min-loss knapsack instance in iteration i. The objective value of solution O_i is $\sum_{j\in T_i\setminus O_i}\Delta_{ji} = \sum_{j\in T_i\cap N_i}\Delta_{ji}$. The first claim now follows since we use an α-approximation algorithm for min-loss knapsack.

To see the second claim, notice that $T'_i\setminus S_i = (T_i\setminus S_i)\cup(T'_i\setminus T_i)$. So

$$\sum_{j\in T'_i\setminus S_i}\Delta_{ji} = \sum_{j\in T_i\setminus S_i}\Delta_{ji}+\sum_{j\in T'_i\setminus T_i}\Delta_{ji}\leq\alpha\cdot\sum_{j\in T_i\cap N_i}\Delta_{ji}+\sum_{j\in T'_i\setminus T_i}\Delta_{ji}\leq\alpha\cdot\sum_{j\in T_i\cap N_i}\Delta_{ji}+\sum_{j\in N_i\setminus T_i}\Delta_{ji}.$$

The last inequality uses $T'_i\setminus T_i\subseteq N_i\setminus T_i$: this is because the completion time (in the optimal schedule) of every chain in $T'_i\setminus T_i$ is at least a_i. Now the second claim follows because $\alpha\geq 1$. ■

Now we bound the completion time of the chains in the algorithm.

Claim 4 *For each $i\geq 0$, the completion time in Step 8 of any chain in S_i is at most $3a_i$. Hence, in a 6 speed schedule, the completion time of all chains in S_i is at most a_{i-1}.*

Proof: Fix any $i\geq 0$ and chain $j\in S_i$. By the definition of the deadlines we know that the critical path of chain j is at most d_j. By the definition of the min-loss knapsack instances, we can bound the total size of chains with deadline at most a_i as follows:

$$\sum_{k=0}^{i}\sum_{j\in S_i}V_j \quad\leq\quad \sum_{k=0}^{i}Pa_k \quad=\quad Pa_i\sum_{k=0}^{i}2^{-k} \quad\leq\quad 2Pa_i.$$

Now we have the two bounds required in the proof of Theorem 6: critical path and the total size of chains with smaller deadlines. So we obtain that the completion time of chain j is at most $a_i+\frac{2Pa_i}{P} = 3a_i$. This completes the proof. ■

We are now ready to bound the cost of our algorithm. Let ALG denote the total cost under a 6 speed schedule. By Claim 4 we have $\mathsf{ALG}\leq\sum_{j\in S_0}w_j(a_0)+\sum_{i\geq 1}\sum_{j\in S_i}w_j(a_{i-1})$. By definition of the incremental costs, we can write

$$\begin{aligned}
\mathsf{ALG} &\leq \sum_{j\in S_0}\Delta_{j0}+\sum_{i\geq 1}\sum_{j\in S_i}\sum_{k=0}^{i-1}\Delta_{jk} = \sum_{j\in S_0}\Delta_{j0}+\sum_{k\geq 0}\sum_{i\geq k+1}\sum_{j\in S_i}\Delta_{jk} &(9)\\
&= \sum_{j\in[m]}\Delta_{j0}+\sum_{k\geq 1}\sum_{i\geq k+1}\sum_{j\in S_i}\Delta_{jk} = \sum_{j\in[m]}\Delta_{j0}+\sum_{k\geq 1}\sum_{j\in T'_k\setminus S_k}\Delta_{jk} &(10)\\
&\leq \sum_{j\in[m]}\Delta_{j0}+\alpha\sum_{k\geq 1}\sum_{j\in N_k}\Delta_{jk} \leq \alpha\sum_{k\geq 0}\sum_{j\in N_k}\Delta_{jk} \leq \alpha\cdot\mathsf{OPT}. &(11)
\end{aligned}$$

The first inequality in (11) is by Claim 3 and the last inequality is by Claim 2. This completes the proof of Theorem 10.

An Improved Approximation. We can obtain an improved $(1+o(1), 5.83)$-bicriteria approximation by a slight variation of this approach. The idea is to use powers of another number $\rho > 1$ (instead of 2) in the definition of the points a_i in (8). In particular we set $a_i := \ell \cdot \rho^i$ for all $i \geq 0$. The rest of the algorithm remains the same. We now outline the changes in the analysis. Only Claim 4 changes, as follows:

- The completion time of chains in S_i in Step 8 is at most $\frac{2\rho-1}{\rho-1} \cdot a_i$. This is because the total size of chains with deadline at most a_i is now bounded by $Pa_i \sum_{k=0}^{i} \rho^{-k} \leq \frac{\rho}{\rho-1} \cdot Pa_i$.
- Therefore, in a $\frac{2\rho^2-\rho}{\rho-1}$ speed schedule, all chains in S_i complete by time a_{i-1}.

Altogether we obtain a $\left(1+o(1), \frac{2\rho^2-\rho}{\rho-1}\right)$-bicriteria approximation. Optimizing over $\rho > 1$, we obtain a $(1+o(1), 3+2\sqrt{2})$-bicriteria approximation ratio (when $\rho = 1 + 1/\sqrt{2}$).

This directly translates to an improved approximation in Theorems 3 and 4.

3.1 Minisum Objective with Uniform Costs

Here we consider a special case of minisum objectives where the cost functions w_j of all chains $j \in [m]$ is *uniform.* We let $w : \mathbb{R}_+ \to \mathbb{R}_+$ denote the common cost function; so the objective value of a schedule that completes chain j at time C_j is $\sum_{j=1}^{m} w(C_j)$. Examples of such objectives include total completion time and the sum of the p^{th} powers of completion times.

An interesting consequence of the algorithm in Theorem 10 is that we obtain a "universal" schedule that is simultaneously near-optimal for *all* cost functions w.

Theorem 11 *There is an algorithm for malleable scheduling with chain precedence constraints and uniform minisum objectives that given any instance, produces a schedule which is simultaneously a $(1+o(1), 6)$-bicriteria approximation for all objectives.*

Again, using the reduction from general precedence constraints to chain precedence in Section 2, this implies Theorem 4.

Algorithm 3 Universal Schedule for Uniform Minisum Objectives

```
1: initialize unscheduled chains $T \leftarrow [m]$.
2: for $i = 0, 1, \ldots$ do
3:     set $T_i \leftarrow \{j \in T : R_j \leq a_i\}$, i.e. chains whose critical path bound is at most $a_i$.
4:     sort the chains in $T_i$ by non-decreasing size $V_j$.
5:     let $S_i \subseteq T_i$ be the maximum prefix in this sorted order with total size at most $P \cdot a_i$.
6:     set $T \leftarrow T \setminus S_i$.
7:     if $T = \emptyset$ then break.
8: run the algorithm from Theorem 6 with deadline $d_j := a_i$ for all $j \in S_i$.
```

The algorithm here is just Algorithm 2 specialized to the case of uniform minisum objectives (setting each $w_j = w$). The key observation is that the resulting algorithm does not depend on the cost function w. Notice that with uniform costs, the incremental costs Δ_{ji} are also uniform, i.e. for each $i \geq 0$ we have $\Delta_{ji} = \Delta_i$ for all chains $j \in [m]$.

Recall that the min-loss knapsack instance in any iteration i of Algorithm 2 is as follows. The items correspond to chains $j \in T_i$ each of which has size V_j and profit $\Delta_{ji} = \Delta_i$; and the budget on size is $P \cdot a_i$. Since all item profits are uniform, an optimal solution to such an instance has no dependence on Δ_i (and the cost function w). Moreover, there is a straightforward exact algorithm for such instances: sort the items in non-decreasing size and select the largest prefix

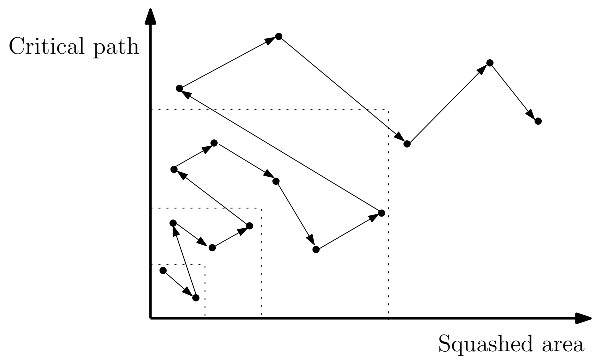

Figure 5: Chains ordered by deadlines in the universal solution of Algorithm 3.

that has total size at most the budget. See Algorithm 3 for a formal description. Figure 5 illustrates Algorithm 3. The axes measure the critical path and squashed area bounds of each chain. The directed path depicts the ordering of the chains by non-decreasing deadlines in Algorithm 3; this is independent of the cost function w.

4 Experimental Results

In this section we evaluate the performance of our family of algorithms via a variety of experiments. We have implemented the faster and simpler minisum algorithm from Section 3, and the minimax algorithm from Section 2.3. The computational tests are in the context of a MapReduce scenario, our motivating practical example. In this context our algorithms are collectively known as *FlowFlex*. We consider its two previously mentioned competitors, *Fair* [44] and *FIFO*. We discuss simulation and real cluster experiments in the next two subsections.

4.1 Simulation Experiments

We begin by describing simulation experiments involving *FlowFlex*, *Fair* and *FIFO*. We will compare the performance of each of these three in terms of the best lower bound that we find. We consider many choices of scheduling objectives, based on completion time, number of tardy jobs, tardiness and SLA step functions. (See Figure 1.) They can be either weighted or non-weighted, and the problem can be to minimize the sum (and hence average) or the maximum over all flows. So, for example, average and weighted average completion time are included for the minisum case. So is average stretch, which is simply completion time weighted by the reciprocal of the amount of work associated with the flow. Similarly, makespan (which is maximum completion time), maximum weighted completion time, and thus maximum stretch is included for the minimax case. Weighted or unweighted numbers of tardy jobs, total tardiness, total SLA costs are included in the minisum case. Maximum tardy job cost, maximum tardiness and maximum SLA cost are included in the minimax case. (A minimax problem involving unit weight tardy jobs would simply be 1 if tardy flows exist, and 0 otherwise, so we omit that objective.) We note that that these experiments are somewhat unfair to both *Fair* and *FIFO*, since both are agnostic with respect to the objective and they focused also on singleton MapReduce jobs rather than flows.

The calculation of the lower bound depends on whether the problem is minisum or minimax. For minisum problems the solution to the minimum cost flow problem provides the lower bound, as does the sum of the critical path objective function values for each chain. For minimax prob-

Algorithm	*FlowFlex*	*Fair*	*FIFO*
Completion Time	1.23/1.46	2.10/2.25	2.07/ 3.00
Stretch	1.22/1.38	3.79/6.32	7.92/21.03
Weighted Completion Time	1.25/1.52	2.42/3.15	2.30/ 5.39
Number of Tardy Jobs	1.42/2.12	1.88/3.97	1.64/ 3.31
Weighted Number of Tardy	1.65/3.06	2.71/9.31	2.14/ 7.08
Tardiness	1.51/3.11	3.51/8.54	3.74/10.55
Weighted Tardiness	1.77/4.11	4.84/8.99	5.25/16.54
Unit SLA	1.62/3.27	2.62/5.19	2.22/ 4.27
SLA	1.52/2.44	2.56/5.31	2.32/ 4.96

Table 1: **Minisum** Simulation Results (Average/Worst Case)

lems the maximum of the critical path objective function values provides a lower bound. But we also improve this bound based on the solution found via a bracket and bisection algorithm. We perform an additional bisection algorithm between the original lower bound and our solution, since we know that the partial sums of the squashed area bounds must be met by the successive deadlines.

Each simulation experiment was created using the following methodology. The number of flows was chosen from a uniform distribution between 5 and 20. The number of jobs for a given flow was chosen from a uniform distribution between 2 and 20. These jobs were then assumed to be in topological order and the precedence constraint between jobs j_1 and j_2 was chosen based on a probability of 0.5. Then all jobs without successors were assumed to precede the last job in the flow, to ensure connectivity. Sampling from a variety of parameters governed the size and processor maxima of different jobs. Weights in the case of completion time, number of tardy jobs and tardiness were also chosen from a uniform distribution between 1 and 10. The one exception was for stretch objectives, where the weights are predetermined by the size of the flow. Similarly, in the case of SLA objectives, the number of steps and the incremental step heights were chosen from comparable distributions with a maximum of 5 steps. Single deadlines for the tardy and tardiness cases were chosen so that it was possible to meet the deadline, with a uniform random choice of additional time given. Multiple successive deadlines for the SLA case were chosen similarly. The number P of processors (slots) was set to 100.

Algorithm	*FlowFlex*	*Fair*	*FIFO*
Makespan	1.01/1.07	1.01/ 1.10	1.02/ 1.08
Stretch	1.03/1.14	4.33/13.42	15.67/55.00
Weighted Completion Time	1.05/1.14	2.22/ 3.81	2.15/ 3.81
Weighted Number of Tardy	1.12/1.17	1.50/10.00	1.47/10.00
Tardiness	1.07/1.35	1.13/ 1.81	1.47/ 4.12
Weighted Tardiness	1.08/1.31	2.69/ 5.28	2.92/ 5.92
Unit SLA	1.26/1.50	1.50/ 3.00	1.42/ 2.00
SLA	1.10/1.43	1.59/ 2.63	1.54/ 2.63

Table 2: **Minimax** Simulation Results (Average/Worst Case)

Table 1 illustrates both average and worst case performance (given 25 simulation experiments each) for 9 minisum objectives. Each row represents the ratio of the *FlowFlex, Fair* or *FIFO* algorithms to the best lower bound available. Each table entry consists of two numbers: the first is the average ratio and the second is the maximum ratio, both taken over the 25 instances. So by definition each ratio must be at least 1. Ratios close to 1 are by definition very good solutions, but, of course, solutions with poorer ratios may still be close to optimal. Note that *FlowFlex* performs significantly better than either *Fair* or *FIFO*, and often is close to optimal. *FIFO* performs particularly poorly on average stretch, because the weights can cause great volatility. *FlowFlex* also does dramatically better than either *Fair* or *FIFO* on the tardiness objectives.

Similarly, Table 2 illustrates the comparable minimax experiments, for those 8 objectives which make sense. Here one sees that makespan is fine for all schemes, which is not particularly surprising. But *FlowFlex* does far better than either *Fair* or *FIFO* on all the others. In all 8 sets of experiments, *FlowFlex* is within 1.26 of "optimal" on average, and generally quite a lot better.

4.2 Cluster Experiments

The cluster experiments were performed using the IBM Platform Symphony MapReduce framework within its BigInsights product [4]. There were 20 processing nodes of 10 slots each. One node was reserved for the scheduler and other MapReduce software. So we had $P = 190$ processors (slots).

We used a workload based on the standard Hadoop Gridmix2 benchmark [17]. For each experiment we ran 10 flows, each consisting of 2 to 10 Gridmix jobs of random sizes, randomly wired into a dependency graph by the same basic procedure we used for our simulation experiments. The experiment driver program submitted a job only when it was *ready.* That is, all of the jobs it depended upon were completed. We ran two sets of experiments: one where all flows arrived at once and another where flows arrived at random intervals chosen from an exponential distribution. For each type of experiment we ran three different random sets of arrival times and job sizes.

In these cluster experiments, the schedulers are running in something more like their native environment. Specifically, they are epoch-based: Every epoch (roughly 2 seconds) they examine the newly revised problem instance. Thus the job sizes for *FlowFlex* change from epoch to epoch. And, of course, flows and jobs arrive and complete. *FlowFlex* then produces a complete schedule that corresponds to allocation suggestions. This is then implemented to the extent possible by the Assignment Layer.

A few comments should be mentioned here. First, we have intentionally not employed sophisticated schemes for estimating the amount of work of each job in the various flows. We do know the number of tasks per job, however, and estimate work for unstarted jobs by using a default work prediction per task. For running jobs we continue to refine our work estimates by extrapolating based on data from the completed tasks. All of this could be improved, for example by incorporating the techniques in [35]. Better estimates should improve the quality of our *FlowFlex* scheduler, but they are orthogonal to the current paper. The second comment is that the Reduce phase is *ready* precisely when some fixed fraction of the Map phase tasks preceeding it have finished. *FlowFlex* simply coalesces all such ready tasks within a single MapReduce job and adjusts the maxima accordingly.

We compared *FlowFlex* to *Fair* and *FIFO* running for submitted jobs. The schedulers were not aware of jobs that were not yet submitted. Figure 6 reports the relative performance improvement of *FlowFlex* for average completion time, makespan, and average and maximum stretch for both sets of experiments. Essentially, we are evaluating the four most commonly used

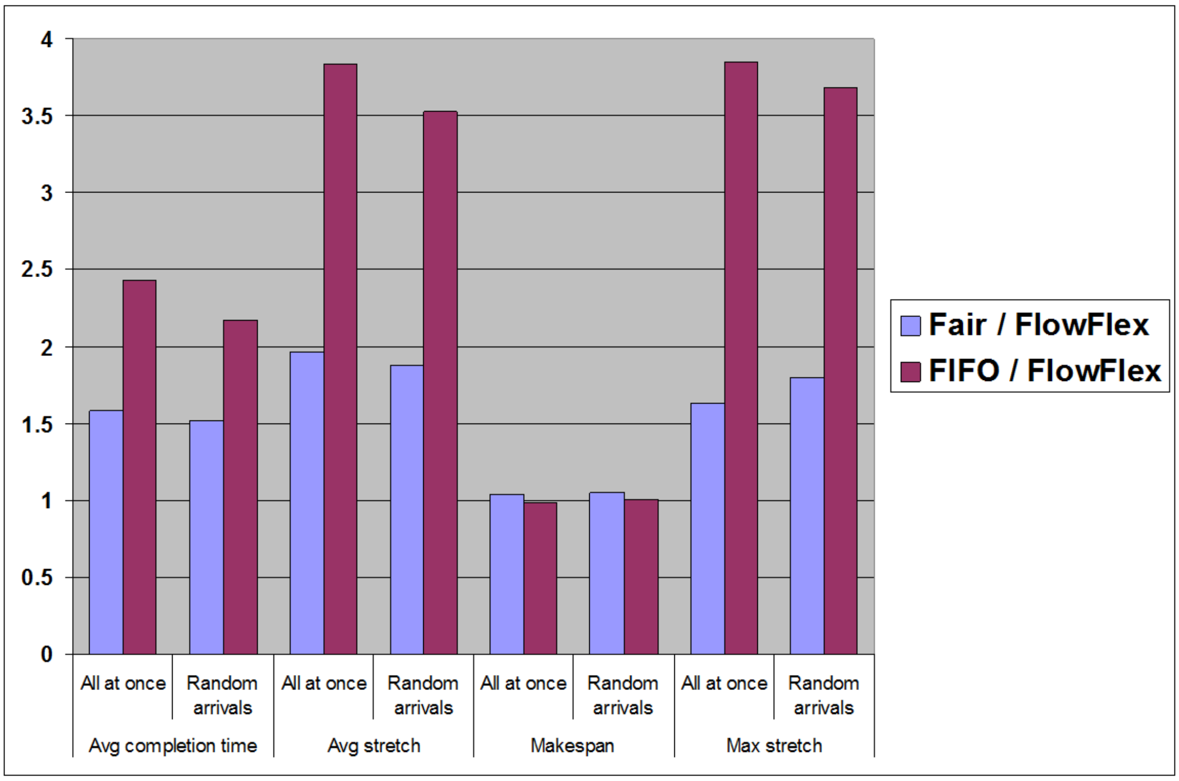

Figure 6: Cluster experiments: relative performance of Fair, FIFO and FlowFlex.

scheduling objectives. Also, all data here for the stretch-based metrics are evaluated based on an analysis of the actual amount of work in each flow. The results in these cluster experiments are more or less comparable to those of the simulations. One notes that the effect of releasing all flows at once or randomly is modest. Makespan, as before, shows all three schemes to be more or less identical in performance. It appears that makespan is a relatively easy problem to solve well in practice. For all other objectives, *Fair* and *FIFO* are at least 50% worse than *FlowFlex*.

We also ran cluster experiments designed to test the robustness of each of the four primary *FlowFlex* algorithms to the three *other* cost functions. Can a *FlowFlex* algorithm for average completion time, for example, also perform well on average stretch, makespan and maximum stretch? Note that the theoretical result (Theorem 4) on universal schedules does not apply to stretch-based metrics, but finding a universal *FlowFlex* algorithm still makes good practical sense. Table 3 describes an additional cluster experiment with 10 flows starting at once, and Table 4 describes an experiment with random arrivals. The four standard algorithms are represented by the rows. The four columns correspond to different objectives. The smallest values in each column should, in theory, be along the diagonal. And this is close to true in most cases. In the average stretch column of Table 3 *FlowFlex* using average stretch is 8% worse than *Flowflex* using average completion time. The reverse is true (by 6%) for the average completion time in Table 4. In the maximum stretch column of Table 4 *FlowFlex* using maximum stretch is 5% worse than *FlowFlex* using maximum stretch. We think the proper way to look at these results is that every one of these four *FlowFlex* with the *exception* of makespan produces close to the best cost function for all four metrics. Why is makespan an outlier? The reason is that makespan, alone among these four cost functions, is primarily specific to the aggregate of all

FlowFlex Algorithm	Average completion time	Average stretch	Makespan	Maximum stretch
Average completion time	211	1.97	780	3.53
Average stretch	223	2.13	776	3.55
Makespan	528	8.08	759	14.49
Maximum stretch	230	2.10	854	2.99

Table 3: Cluster experiments: Algorithmic performance on 4 key cost functions, all at once

FlowFlex Algorithm	Average completion time	Average stretch	Makespan	Maximum stretch
Average completion time	170	1.46	788	2.35
Average stretch	161	1.33	787	2.09
Makespan	386	5.38	757	8.77
Maximum stretch	175	1.38	877	2.20

Table 4: Cluster experiments: Algorithmic performance on 4 key cost functions, random arrivals

the flows rather than the individual flows.

5 Conclusion

We considered the problem of scheduling flows of precedence-constrained jobs in a malleable parallel scheduling setting. Our unified approach handles a wide variety of commonly used minisum and minimax cost functions. Since no standard approximation ratios are possible for our general problem (unless P=NP), we obtained constant-factor bicriteria approximations for both minisum and minimax objectives. We obtained constant-factor (usual) approximation algorithms in some special cases. We also provided experimental analyses which demonstrate good performance relative to lower bounds on the optimum (obtained as byproducts of our algorithms), as well as to other commonly used MapReduce schedulers.

References

[1] B. Baker, E. Coffman and R. Rivest: Orthogonal Packings in Two Dimensions, SIAM Journal on Computing, 9(4): 846-855, 1980.

[2] K. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan and E. Shekita: Jaql: A Scripting Language for Large Scale Semistructured Data Analysis, Proceedings of VLDB, 4(12), 1272-1283, 2011.

[3] J. Berlinska and M. Drozdowski, Scheduling Divisible MapReduce Computations. Journal of Parallel and Distributed Computing, 71, 450–459, 2011.

[4] BigInsights: www-01.ibm.com/software/data/infosphere/biginsights/

[5] Chi-Yeh Chen and Chih-Ping Chu, A 3.42-Approximation Algorithm for Scheduling Malleable Tasks under Precedence Constraints, IEEE Trans. Parallel Distrib. Syst., 24(8), 1479-1488, 2013.

[6] C. Chekuri, R. Motwani, B. Natarajan and C. Stein, Approximation Techniques for Average Completion Time Scheduling, SIAM J. Comput, 31(1), 146-166, 2001.

[7] E. Coffman, M. Garey, D. Johnson and R. Tarjan: Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms, SIAM Journal on Computing, 9(4): 808–826, 1980.

[8] S.I. Daitch and D.A. Spielman, Faster approximate lossy generalized flow via interior point algorithms, Proceedings of the 40th Annual ACM Symposium on Theory of Computing, 451–460, 2008.

[9] J. Dean, J. and S. Ghemawat: Mapreduce: Simplified Data Processing on Large Clusters, ACM Transactions on Computer Systems, 51(1):107–113, 2008.

[10] M. Drozdowski: New Applications of the Munz and Coffman Algorithm, Journal of Scheduling, 4(4):209-223, 2001.

[11] M. Drozdowski: Real-Time Scheduling of Linear Speedup Parallel Tasks, Information Processing Letters, 57(1):35-40, 1996.

[12] M. Drozdowski, Scheduling for Parallel Processing, Springer, 2009.

[13] M. Drozdowski and W. Kubiak: Scheduling Parallel Tasks With Sequential Heads and Tails, Annals of Operations Research, 90:221–246, 1999.

[14] L. Fleischer and K. Wayne, Fast and simple approximation schemes for generalized flow, Mathematical Programming, 91(2):215–238, 2002.

[15] M. Garey and R. Graham: Bounds for Multiprocessor Scheduling with Resource Constraints. SIAM Journal on Computing, 4(2):187-200, 1975.

[16] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan and U. Srivastava, Building a High-Level Dataflow System on Top of MapReduce: The Pig Experience, Proceedings of VLDB, 2(2), 1414-1425, 2009.

[17] Gridmix: https://hadoop.apache.org/docs/r1.2.1/gridmix.html

[18] Elisabeth Günther, Felix G. König, Nicole Megow, Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width, J. Comb. Optim., 27(1), 164-181, 2014.

[19] Hadoop: https://hadoop.apache.org

[20] D.S. Hochbaum and D.B. Shmoys: A Unified Approach to Approximation Agorithms for Bottleneck Problems, Journal of the ACM, 33(3): 533-550, 1986.

[21] O.H. Ibarra and C.E. Kim, Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems, J. ACM, 22(4), 463-468, 1975.

[22] Klaus Jansen and Hu Zhang, Scheduling malleable tasks with precedence constraints, J. Comput. Syst. Sci., 78(1), 245-259, 2012.

[23] Klaus Jansen and Hu Zhang, An approximation algorithm for scheduling malleable tasks under general precedence constraints, ACM Trans. Algorithms, 2(3), 416-434, 2006.

[24] B. Kalyanasundaram and K. Pruhs, Speed is as Powerful as Clairvoyance, Journal of the ACM, 47(4):617-643, 2000.

[25] H. Karloff, S. Suri and S. Vassilvitskii, A Model of Computation for MapReduce, In SODA, 938-948, 2010.

[26] P. Koutris and D. Suciu, Parallel evaluation of conjunctive queries, In PODS, 223-234, 2011.

[27] J. Labetoulle, E. Lawler, J. Lentra and A. Rinnoy Kan, Preemptive Scheduling of Uniform Machines Subject to Release Dates, W. Pulleyblank, editor, Progress in Combinatorial Optimization, 245-261, Academic Press, New York, 1984.

[28] Renaud Lepére, Denis Trystram, Gerhard J. Woeginger, Approximation Algorithms for Scheduling Malleable Tasks under Precedence Constraints, ESA 2001, 146-157.

[29] J. Leung, Handbook of Scheduling, Chapman and Hall/CRC, 2004.

[30] W. Ludwig and P. Tiwari: Scheduling Malleable and Nonmalleable Parallel Tasks, Symposium on Discrete Algorithms, Arlington, VA, 167-176, 1994.

[31] R. McNaughton, Scheduling with Deadlines and Loss Functions, Management Science, 6(1):1–12, 1959.

[32] B. Moseley, A. Dasgupta, R. Kumar and T. Sarlós, On scheduling in Map-Reduce and Flow-Shops, Symposium on Parallel Algorithms and Architectures, San Jose, CA, 2011.

[33] V. Nagarajan, J. Wolf, A. Balmin and K. Hildrum, Flowflex: Malleable Scheduling for Flows of MapReduce Jobs, International Middleware Conference, Beijing, China, 103-122, 2013

[34] M. Pinedo, Scheduling: Theory, Algorithms and Systems, Prentice Hall, 1995.

[35] A. Popescu, V. Ercegovac, A. Balmin, M. Branco and A. Ailamaki, Same Queries, Different Data: Can We Predict Runtime Performance?, International Conference in Data Engineering, Washington, DC, 275-280, 2012.

[36] U. Schwiegelshohn, W. Ludwig, J. Wolf, J. Turek and P. Yu, Smart SMART Bounds for Weighted Response Time Scheduling, SIAM Journal on Computing, 28(1):237–253, 1999.

[37] D. Sleator, A 2.5 Times Optimal Algorithm for Packing in Two Dimensions, Information Processing Letters 10(1):37-40, 1980.

[38] P. Schuurman and G.J. Woeginger, A Polynomial Time Approximation Scheme for the Two-Stage Multiprocessor Flow Shop Problem, Theor. Comput. Sci., 237(1-2), 105-122, 2000.

[39] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N.Zhang, S. Anthony, H. Liu and R. Murthy, Hive - a Petabyte Scale Data Warehouse using Hadoop, International Conference on Data Engineering, Long Beach, CA, 996-1005, 2010.

[40] J. Turek, J. Wolf and P. Yu: Approximate Algorithms for Scheduling Parallelizable Tasks, Symposium on Parallel Algorithms and Architectures, San Diego, CA, 323-332, 1992.

[41] V. Vizing, Minimization of the Maximum Delay in Servicing systems with Interruption, USSR Computational Mathematics and Methematical Physics, 22(3):227-233, 1982.

[42] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.-L. Wu and R. Vernica, On the Optimization of Schedules for MapReduce Workloads in the Presence of Shared Scans, VLDB Journal, 21(5):589-609, 2012.

[43] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, R. Kumar, S. Parekh, K.-L. Wu and A. Balmin, FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads, International Middleware Conference, Bangalore, India, 1-20, 2010.

[44] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Schenker and I. Stoica, Job Scheduling for Multi-User MapReduce Clusters, UC Berkeley Technical Report EECS-2009-55, 2009.

[45] M. Zaharia, D. Borthakur, J. Sarma, K. Elmeleegy, S. Shenker and I. Stoica: Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling, European Conference on Computer Systems, Paris, France, 265-278, 2010.