FlowFlex: Malleable Scheduling for Flows of MapReduce Jobs

Viswanath Nagarajan, Joel Wolf, Andrey Balmin, Kirsten Hildrum

FlowFlex: Malleable Scheduling for Flows of MapReduce Jobs

Viswanath Nagarajan[1], Joel Wolf[1], Andrey Balmin[2⋆], and Kirsten Hildrum[1]

[1] {viswanath, jlwolf, hildrum}@us.ibm.com, IBM T. J. Watson Research Center
[2] andrey@graphsql.com, GraphSQL

Abstract. We introduce *FlowFlex*, a highly generic and effective scheduler for flows of MapReduce jobs connected by precedence constraints. Such a flow can result, for example, from a single user-level Pig, Hive or Jaql query. Each flow is associated with an arbitrary function describing the cost incurred in completing the flow at a particular time. The overall objective is to minimize either the total cost (minisum) or the maximum cost (minimax) of the flows. Our contributions are both theoretical and practical. Theoretically, we advance the state of the art in malleable parallel scheduling with precedence constraints. We employ resource augmentation analysis to provide bicriteria approximation algorithms for both minisum and minimax objective functions. As corollaries, we obtain approximation algorithms for total weighted completion time (and thus average completion time and average stretch), and for maximum weighted completion time (and thus makespan and maximum stretch). Practically, the *average* case performance of the *FlowFlex* scheduler is excellent, significantly better than other approaches. Specifically, we demonstrate via extensive experiments the overall performance of *FlowFlex* relative to optimal and also relative to other, standard MapReduce scheduling schemes. All told, *FlowFlex* dramatically extends the capabilities of the earlier *Flex* scheduler for singleton MapReduce jobs while simultaneously providing a solid theoretical foundation for both.

1 Introduction

MapReduce [8] is a fundamentally important programming paradigm for processing big data. Accordingly, there has already been considerable work on the design of high quality MapReduce schedulers [26, 27, 25, 1, 24]. All of the schedulers to date have quite naturally focused on the scheduling of collections of *singleton* MapReduce jobs. Indeed, single MapReduce jobs were the appropriate atomic unit of work early on. Lately, however, we have witnessed the emergence of more elaborate MapReduce work, and today it is common to see the submission of *flows* of interconnected MapReduce jobs. Such a MapReduce flow can result, for example, from a single user-level Pig [11], Hive [20] or Jaql [4] query. Each flow can be represented by a directed acyclic graph (DAG) in which the

⋆ Work performed while at IBM Almaden Research Center.

nodes are singleton Map or Reduce phases and the directed arcs represent precedence. Significantly, flows have become the basic unit of MapReduce work, and it is the completion times of these flows that determines the appropriate measure of goodness, not the completion times of the individual MapReduce jobs.

This paper introduces *FlowFlex*, a scheduling algorithm for flows of MapReduce jobs. *FlowFlex* can attempt to optimize an arbitrary metric based on the completion times of the flows. Common examples include makespan, average completion time, average and maximum *stretch*[3] and metrics involving one or more deadlines. Any given metric will be appropriate for a particular scenario, and the precise algorithmic variant *FlowFlex* applies will depend on that metric. For example, in a batch environment one might care about makespan, to ensure that the batch window is not elongated. In an interactive environment users would typically care about average or maximum completion time, or about average or maximum stretch. There are also a variety of metrics associated with hard or soft deadlines. To the best of our knowledge scheduling schemes for flows of MapReduce jobs have never been considered previously in the literature.

Our contributions are both theoretical and practical. We advance the theory of *malleable* parallel scheduling with precedence constraints. Specifically, we employ *resource augmentation* analysis to provide *bicriteria approximation algorithms* for both minisum and minimax objective functions. As corollaries, we obtain *approximation algorithms* for *total weighted completion time* (and thus average completion time and average stretch), and for *maximum weighted completion time* (and thus makespan and maximum stretch). We also produce a highly generic and practical MapReduce scheduler for flows of jobs, called *FlowFlex*, and demonstrate its excellent average case performance experimentally.

The closest previous scheduling work for MapReduce jobs appeared in [25]. The *Flex* scheduler presented there is now incorporated in IBM BigInsights [5]. (See also the *FlexSight* visualization tool [7].) *Flex* schedules to optimize a variety of metrics as well, but differs from the current work in that it only considers singleton MapReduce jobs, not flows. Architecturally, *Flex* sits on top of the *Fair* MapReduce scheduler [26, 27], essentially overriding its decisions while simultaneously making use of its infrastructure. The *FlowFlex* scheduling problem is clearly a major generalization of the *Flex* problem. With modest caveats, it can also be said that the *FlowFlex* algorithms significantly generalize those of *Flex*. They are also much more theoretically grounded. See also the special purpose schedulers [1] and *CircumFlex* [24], built to amortize shared Map phase scans.

There are fundamental differences between *Fair* and the three schedulers in the *Flex* family: *Fair* makes its decisions based on the current moment in time, and thus considers only the resource (slot) dimension. *Fair* is indeed fair in the sense of instantaneous *progress*, but does not directly consider completion time. On the other hand, *Flex, CircumFlex and FlowFlex* think in two dimensions, both resource and time. And they optimize towards completion time metrics. It is our contention that completion time rather than instantaneous progress determines the true quality of a schedule.

[3] Stretch is a fairness metric in which each flow weight is the reciprocal of its size.

The rest of this paper is organized as follows. Section 2 gives preliminaries and describes the good fit between the theory of malleable scheduling and the MapReduce enviornment. Section 3 introduces the scheduling model and lists our formal theoretical results. The *FlowFlex* scheduling algorithms are described in Section 4, and the proofs of the performance guarantees are outlined there. Space limitations prevent us from detailing all of these proofs, but interested readers can find them in [2]. In Section 5 we compare *FlowFlex* experimentally with *Fair* and *FIFO*, both naturally extended in order to handle precedence constraints. We also explain the practical considerations associated with implementing *FlowFlex* as an epoch-based scheduler. Conclusions appear in Section 6.

2 Preliminaries

The theory of malleable scheduling fits the reality of the MapReduce environment well. To understand this we give a brief, somewhat historically oriented overview of theoretical parallel scheduling and its relation to MapReduce.

The first parallel scheduling implementations and theoretical results involved what are today called *rigid* jobs. These jobs run on a fixed number of processors and are presumed to complete their work simultaneously. One can thus 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 [6], focused on the makespan metric, providing some of the very first *approximation* algorithms. (These are polynomial time schemes with guaranteed performance bounds.)

Subsequent parallel scheduling research took a variety of directions, again more or less mirroring real scenarios of the time. One such direction involved what has now become known as *moldable* scheduling: Each job can be run on an arbitrary number of processors, but with an execution time which is a monotone non-increasing function of the number of processors. Thus the height of a job is turned from an input parameter to a decision variable. The first approximation algorithm for moldable scheduling with a makespan metric appeared in [23]. Later, [22] found the first approximation algorithm for both rigid and moldable scheduling problems with a (weighted) average completion time metric.

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 its fixed amount of work. One can consider the most general problem variant in which the rate at which work is done is a function of the number of allocated processors, so that the total work completed at any time is the integral of these rates through that time. However, this problem is enormously difficult, and so the literature to date [9, 16] has focused on the special case where the speedup function is linear through a given maximum number of processors, and constant thereafter. Clearly, malleable schedules can only improve objective function values relative to moldable schedules. On the other hand, malleable scheduling problems are even harder to solve well than moldable scheduling problems. We will concentrate on malleable scheduling problems with

linear speedup up to some maximum, with flow precedence constraints and any of several different metrics on the completion times of the flows. See [9, 16] for more details on both moldable and malleable scheduling. The literature on the latter is quite limited, and this paper is a contribution.

Why does MapReduce fit the theory of malleable scheduling with linear speedup and processor maxima so neatly? There are multiple reasons.

1. *MapReduce and malleable scheduling are about allocation:* 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, and that is where any mathematical complexity resides. The Assignment Layer then implements these allocation decisions (to the extent possible, given locality issues [27] and such) in the MapReduce cluster. *Fair, Flex, CircumFlex* and our new *FlowFlex* scheduler reside in the Allocation Layer. The malleable (as well as rigid and moldable) scheduling literature is also about allocation rather than assignment.[4]
2. *MapReduce work exhibit roughly linear speedup, and maximum constraints occur naturally:* 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: Remember that linear speedup is a statement about the rate at which work is completed. Maximum constraints in either the Map or Reduce phase occur because they happen to be small (and thus have few tasks), or when only a few tasks remain to be allocated.
3. *MapReduce fits the malleable model well:* Assuming the tasks are many and small, the decisions of the scheduler can be approximated closely. To understand this, consider Figure 1, which depicts the Assignment Layer implementing the decisions of the Allocation Layer. The Allocation Layer output is a hypothetical malleable schedule for three jobs. The Assignment Layer works locally at each node in the cluster. Suppose a task on that node completes, freeing a slot. The Assignment Layer simply determines which job is the most relatively underallocated according to the Allocation Layer schedule. And then, within certain practical constraints, it acts greedily, assigning a new task from that job to the slot. Examining the figure, the tasks are represented as "bricks" in the Assignment Layer. The point is that the large number and small size of the tasks makes the right-hand side a close approximation to the left-hand side. That is, Assignment Layer reality will be an excellent approximation to Allocation Layer theory.

The model is not perfect, of course. For example, if the number of tasks in a MapReduce job is modest, the idealized scenario depicted in Figure 1 will be less than perfect. Furthermore, there is a somewhat delicately defined notion[5]

[4] In MapReduce, the atomic unit of allocation is called a *slot*, which can be used for either Map or Reduce tasks. So "processor" in the theoretical literature corresponds to "slot" in a MapReduce context.

[5] This will be clarified in Subsection 5.2.

of precedence between Map and Reduce tasks that is not cleanly modeled here. Recall that we model a single MapReduce job as a Map phase followed by a Reduce phase. In reality, some Reduce tasks can begin before all the Map tasks complete. But recall that *Flex* suffers from precisely the same issues, and has been used successfully in IBM BigInsights [5] for several years.

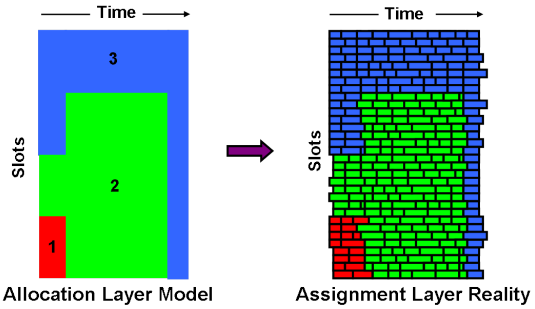

Fig. 1: MapReduce and Malleable Scheduling.

Practically speaking, *FlowFlex* and its predecessors are epoch-based. In each epoch *FlowFlex* wakes up, considers the current version of the scheduling problem, produces a hypothetical malleable schedule and outputs the *initial* allocations to be implemented by the assignment layer. Thus the size data as well as the flows and jobs themselves are each updated for each *FlowFlex* run, making the scheduler more robust.

Other Models for MapReduce. We briefly mention some other MapReduce models that have been considered in the literature. Moseley et al. [18] consider a "two-stage flexible flow shop" [21] model, and give approximation and online algorithms for total completion time of independent MapReduce jobs. 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 [14, 15].

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.

3 Formal Model and Results

As discussed, we model the MapReduce application as a parallel scheduling problem. There are P identical *processors* that correspond to resources (slots) in the MapReduce cluster. Each *flow* j is described by means of a directed acyclic graph. The nodes in each of these DAGs 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 as the job *size*), and can be performed on a maximum number $\delta_i \in [P]$ of processors at any point in time.[6] 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 $p \in [\delta_i]$ assigned 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 satisfies 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, [9].)

- Total load (or *squashed area*): $\frac{1}{P} \sum_{i \in I_j} s_i$.
- Critical path: maximum of $\sum_{r=1}^{\ell} \frac{s_{i_r}}{\delta_{i_r}}$ over all chains[7] $i_1 \prec \cdots \prec i_\ell$ in flow j.

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 job 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 [9, 16] this scheduling environment is $P|var, p_i(k) = \frac{p_i(1)}{k}, \delta_i, prec|*$.[8] We refer to these problems collectively as *precedence constrained malleable scheduling with linear speedup*. Our highly general cost model can solve all the commonly used scheduling objectives: weighted average completion time, makespan (maximum completion time), average and maximum stretch, and deadline-based metrics associated with number of tardy jobs, service level agreements (SLAs) and so on. Figure 2 illustrates 4 basic types of cost functions.

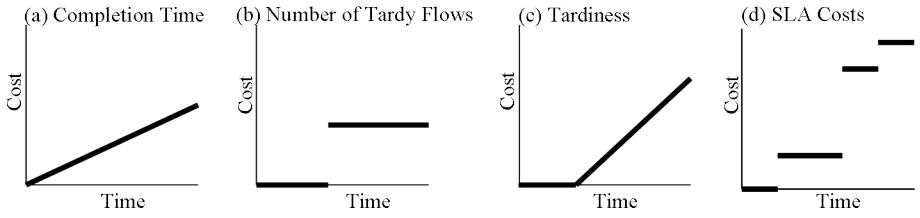

Fig. 2: Typical Cost Functions Types.

[6] Throughout the paper, for any integer $\ell \geq 1$, we denote by $[\ell]$ the set $\{1, \ldots, \ell\}$.

[7] Chains are a special case of flows in which precedence is sequential.

[8] Here *var* stands for malleable scheduling, $p_i(k) = \frac{p_i(1)}{k}$ denotes linear speedup, δ_i is processor maxima, *prec* stands for precedence, and $*$ is for *any* objective function.

The objective functions we consider are either *minisum* or *minimax*: Minisum scheduling problems involve the minimization of the (possibly weighted) sum of individual flow metrics, or, equivalently, their (weighted) average. On the other hand, minimax scheduling problems involve the minimization of the maximum of individual metrics, an indication of worst case flow performance.

Definition 1. *A polynomial time algorithm is said to be an α-approximation if it produces a schedule that has objective value at most $\alpha \geq 1$ times optimal.*

We would like to provide approximation algorithms for the above malleable scheduling problems. But as shown in [10], even under very special precedence constraints (chains of length three) the general minisum and minimax problems admit no finite approximation ratio unless P=NP. Hence we use resource augmentation [13] and focus on *bicriteria* approximation guarantees.

Definition 2. *A polynomial time algorithm is said to be an (α, β)-bicriteria approximation if it produces a schedule using $\beta \geq 1$ speed processors that has objective value at most $\alpha \geq 1$ times optimal (under unit speed processors).*

Our main result is that we can find approximation algorithms in some cases and bicriteria approximation algorithms in all others.

Theorem 1. *The precedence constrained malleable scheduling problem with linear speedup admits the following guarantees.*

- $(2, 3)$*-bicriteria approximation algorithm for general minisum objectives.*
- $(1, 2)$*-bicriteria approximation algorithm for general minimax objectives.*
- 6*-approximation algorithm for total weighted completion time (including total stretch).*
- 2*-approximation algorithm for maximum weighted completion time (including makespan and maximum stretch).*

The first two results on general minisum and minimax objectives imply the other two as corollaries. The main idea in our algorithms (for both minisum and minimax) is a reduction to strict *deadline metrics*, for which a simple greedy scheme is shown to achieve a good bicriteria approximation. The reduction from minisum objectives to deadline metrics is based on a *minimum cost flow* relaxation, and "rounding" the optimal flow solution. The reduction from minimax objectives to deadlines is much simpler and uses a bracket and bisection search.

4 The *FlowFlex* Scheduling Algorithm

Our scheduling algorithm has three sequential stages. See Figure 3 for an algorithmic overview. In a little more detail, the stages may be described as follows.

1. First we consider each flow j separately, and convert its (general) precedence constraint into a *chain* (total order) precedence constraint. 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 bound δ_k of the maximum number of processors it can be run on. We note that (unlike jobs) the bound δ_k of a pseudo-job may be larger than P. An important property here is that the squashed-area and critical-path lower bounds of each chain equal those of its original flow.

2. We now treat each flow as a chain of pseudo-jobs, and obtain a malleable schedule consisting of pseudo-jobs. This stage has two components:
   a. We first obtain a bicriteria approximation algorithm in the special case of metrics based on *strict* deadlines, employing a natural *greedy* scheme.
   b. We then obtain a bicriteria approximation algorithm for general cost metrics, by reduction to deadline metrics. For minisum cost functions we formulate a *minimum cost flow* subproblem based on the cost metric, which can be solved efficiently. The solution to this subproblem is then used to derive a deadline for each flow, which we can use in the greedy scheme. For minimax cost metrics we do not need to solve an minimun cost flow problem. We rely instead on a bracket and bisection scheme, each stage of which produces natural deadlines for each chain. We thus solve the greedy scheme multiple times.

   These performance guarantees are relative to the squashed-area and critical-path lower bounds of the chains, which, by Stage 1, equal those of the respective original flows. We now have a malleable schedule for the pseudo-jobs satisfying the chain precedence within each flow as well as the bounds δ_k.
3. The final stage combines Stages 1 and 2. We transform the malleable schedule of pseudo-jobs into a malleable schedule for the original jobs, while respecting the precedence constraints and bounds δ_i. We refer to this as *shape shifting*. Specifically, we convert the malleable schedule of each pseudo-job k into a malleable schedule for the (portions) of jobs i that comprise it. The full set of these transformations, over all pseudo-jobs k and flows j, produces the ultimate schedule.

1: **for** $j = 1, \ldots, m$ **do**
2: Run Stage 1 scheme on flow j, yielding pseudo-schedule for *chain* of pseudo-jobs.
3: Stage 2 scheme begins.
4: **if** minsum objective **then**
5: Run algorithm in Figure 6.
6: **else**
7: Run minimax algorithm in Figure 8.
8: Stage 2 scheme ends.
9: Run Stage 3 shape shifting algorithm using Stages 1 and 2 output.

Fig. 3: High Level Scheme *FlowFlex* Overview

4.1 Stage 1: General Precedence Constraints to Chains

We now describe a procedure to convert an arbitrary precedence constraint 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.

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 computed in $O(n^2)$ time using 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.

Next, we will 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 $I_1, \ldots, I_h$ so that the set of jobs processed by the pseudo-schedule in each interval stays fixed. 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. (We employ this subtle change of notation to differentiate chains from more general precedence constraints.) 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 $1 \prec 2 \prec \cdots h$ 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.[9]

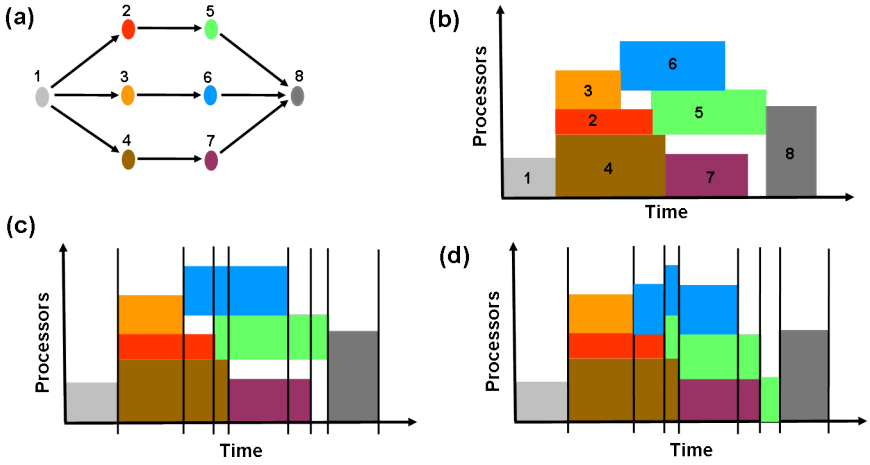

Fig. 4: *FlowFlex* Stage 1.

[9] Clearly, the total size of pseudo-jobs $\sum_{k=1}^h s_k = \sum_{i=1}^n s_i$ the total size of 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.

Figure 4(a) illustrates the directed acyclic graph of a particular flow. Figure 4(b) shows the resulting pseudo-schedule. Figures 4(c) and (d) show the decomposition into maximal intervals.

4.2 Stage 2: Scheduling Flows with Chain Precedence Constraints

In this section, 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 $k_1^j \prec k_2^j \prec \cdots k_{n(j)}^j$ of *pseudo-jobs*, where each pseudo-job k has a size $s(k)$ and specifies a maximum number $\delta(k)$ of processors that it can be run on. 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 cost.

Malleable schedules for pseudo-jobs (resp. chains of pseudo-jobs) are defined identically to jobs (resp. flows). To reduce notation, we denote a malleable schedule for *chain* j by a sequence $\tau^j = \langle \tau_1^j, \ldots, \tau_{n(j)}^j \rangle$ 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 $r \in \{1, \ldots, n(j) - 1\}$, 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 $C(\tau^j) := C(\tau_{n(j)}^j)$.

Unfortunately, as shown in [10], this problem does not admit any finite approximation ratio unless P=NP. Given this hardness of approximation, we focus on bicriteria approximation guarantees. We first give a $(1, 2)$-approximation algorithm when the cost functions are based on strict deadlines. Then we obtain a $(2, 3)$-approximation algorithm for arbitrary minisum metrics and a $(1, 2)$-approximation algorithm for arbitrary minimax metrics. Importantly, all these performance guarantees are based on the squashed-area and critical-path lower bounds of the chains. Since the Stage 1 transformation (flows to chains) maintains these same lower bounds, the guarantees in Stage 2 are relative to the lower bounds of the original flows. So the objective value incurred in Stage 2 is a good approximation to the optimum of the scheduling instance under the original flows.

Scheduling with Strict Deadlines We consider the problem of scheduling chains on P parallel processors under a strict deadline metric. 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.

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

Proof Sketch. By renumbering chains, we assume that $d_1 \leq \cdots \leq d_m$. The algorithm schedules chains in increasing 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 Figure 5. The utilization function $\sigma : \mathbb{R}_+ \to \{0, 1, \ldots, P\}$ denotes the number of processors being used by the schedule at each point in time.

1: Initialize utilization function $\sigma : [0, \infty) \to \{0, 1, \ldots, P\}$ to zero.
2: **for** $j = 1, \ldots, m$ **do**
3: **for** $i = 1, \ldots, n(j)$ **do**
4: Set $S(\tau_i^j) \leftarrow 0$ if $i = 1$ and $S(\tau_i^j) \leftarrow C(\tau_{i-1}^j)$ otherwise.
5: Initialize $\tau_i^j : [0, \infty) \to \{0, \ldots, P\}$ to zero.
6: For each time $t \geq S(\tau_i^j)$ in increasing order, set

$$\tau_i^j(t) \;\leftarrow\; \min\left\{P - \sigma(t)\,,\, \delta(k_i^j)\right\}, \tag{1}$$

 until the area $\int_{t \geq S(\tau_i^j)} \tau_i^j(t)\, dt \;=\; s(k_i^j)$ the size of pseudo-job k_i^j.
7: Set $C(\tau_i^j) \leftarrow \max\{z : \tau_i^j(z) > 0\}$.
8: Update utilization function $\sigma \leftarrow \sigma - \tau_i^j$.
9: Set $C(\tau^j) \leftarrow C(\tau_{n(j)}^j)$.
10: **if** $C(\tau^j) > 2 \cdot d_j$ **then**
11: Instance is *infeasible*.

Fig. 5: Algorithm for Scheduling Chains with Deadline Metric

Notice that this algorithm produces a valid malleable schedule that respects the chain precedence constraints and the maximum processor bounds. To prove the performance guarantee, we show that if there is any solution that meets all deadlines $\{d_\ell\}_{\ell=1}^m$ then the algorithm's schedule satisfies $C(\tau^j) \leq 2 \cdot d_j$ for all chains $j \in [m]$. The main idea is to divide the time $C(\tau^j)$ taken to complete any chain j into two types of events according to Equation (1), namely times where all P processors are fully utilized (i.e. $\tau_i^j(t) = P - \sigma(t)$) and times where a pseudo-job is fully run (i.e. $\tau_i^j(t) = \delta(k_i^j)$). The first event is bounded by the total area in the earliest j chains and the second by the critical path of chain j, each of which is at most d_j. So chain j's completion time $C(\tau^j) \leq 2 \cdot d_j$.

Minisum Scheduling We now consider the problem of scheduling chains on P parallel processors under arbitrary minisum metrics. 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 goal in the minisum problem is to compute a schedule of minimum total cost. Let opt denote the optimal value of the given minisum scheduling instance.

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

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

$$Q_j \quad := \quad \max \left\{ \sum_{i=1}^{n(j)} \frac{s(k_i^j)}{\delta(k_i^j)} \ , \ \frac{1}{P} \sum_{i=1}^{n(j)} s(k_i^j) \right\}, \tag{2}$$

the maximum of the critical path and area lower bounds. Note that the completion time of each chain j (even if it is scheduled in isolation) is at least Q_j. So the optimal value $\mathsf{opt} \geq \sum_{j=1}^{m} w_j(Q_j)$.

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 assume here that (i) each cost function $w_j(\cdot)$ has integer valued breakpoints (i.e. times where the cost changes) and (ii) provide an algorithm with runtime polynomial in H. In the full version, we show that both these assumptions can be removed. Before presenting the algorithm, we recall:

Definition 3 (Minimum cost flow). *The input is a network given by a directed graph (V, E) with designated source/sink nodes and demand ρ, where each arc $e \in E$ has a capacity α_e and cost (per unit of flow) of β_e . A* flow *satisfies arc capacities and node conservation (in-flow equals out-flow), and the goal is to find a flow of ρ units from source to sink having minimum cost.*

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 metrics using $\{d_j\}_{j=1}^m$ to obtain the final solution. The algorithm is described in Figure 6, followed by a high-level proof sketch (the details can be found in the full version).

1: Set volume $V_j \leftarrow \sum_{i=1}^{n(j)} s(k_i^j)$ for each chain $j \in [m]$.
2: Define network N on nodes $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_H\} \cup \{r, r'\}$, where r is the source and r' the sink.
3: Define arcs $E = E_1 \cup E_2 \cup E_3 \cup E_4$ of N as follows (see also Figure 7).

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

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

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

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

4: Compute minimum-cost flow f in N of $\rho := \sum_{j=1}^{m} V_j$ demand from r to r'.
5: Set deadline $d_j \leftarrow \arg\min \left\{ t : \sum_{s=1}^{t} f(a_j, b_s) \geq V_j/2 \right\}$, for all $j \in [m]$.
6: Solve this deadline metric instance using Algorithm 5.

Fig. 6: Algorithm for Scheduling Chains with Minisum Metric

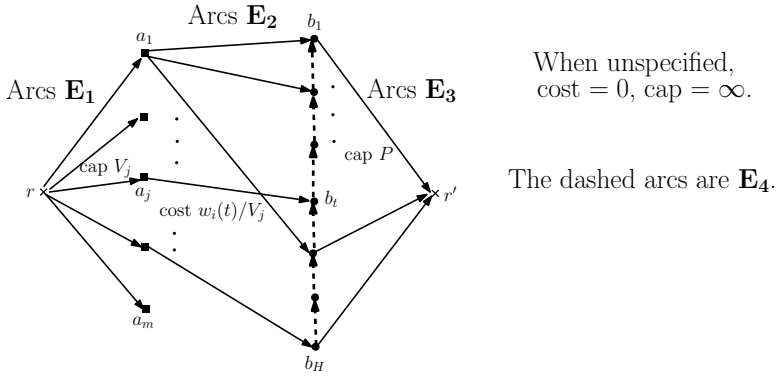

Fig. 7: The Minimum Cost Flow Network.

Proof Sketch of Theorem 3. In the first phase of our algorithm (Steps 1-4) we treat each chain $j \in [m]$ as work of volume V_j, which is the total size of pseudo-jobs in j. The key property of this phase is that the network flow instance on N is a *relaxation* of the original scheduling instance, i.e. the minimum cost flow f is at most opt. This property relies on the construction of N, where the nodes a_js correspond to chains and b_ts correspond to intervals $[t-1, t)$ in time. The arcs E_1 (together with the demand $\rho = \sum_{j=1}^{m} V_j$) enforce that V_j amount of flow is sent to each a_j, i.e. V_j work is done on each chain j. The arcs E_2 ensure that flow from a_j can only go to nodes b_t with $t \geq Q_j$, i.e. chain j can complete only after Q_j. See (2). These arcs also model the minisum cost objective. Finally, arcs E_3 and E_4 correspond to using at most P processors at any time.

In the second phase of the algorithm (Steps 5-6) we use the min-cost flow solution f to obtain a feasible malleable schedule for the m chains. The candidate deadlines $\{d_j\}_{j=1}^{m}$ correspond to times when the chains are "half completed" in the solution f. Since the costs $w_j(\cdot)$ are non-decreasing, the cost $\sum_{j=1}^{m} w_j(d_j)$ of completing chains by their deadlines is at most $2 \cdot \mathit{cost}(f) \leq 2 \cdot \mathsf{opt}$. Then, using the definition of network N, we show that for each chain j, its critical path is at most d_j and the squashed area of earlier-deadline chains is at most $2 \cdot d_j$. These two bounds combined with the analysis of the deadline metric algorithm (Theorem 2) imply that each chain $j \in [m]$ is completed by time $3 \cdot d_j$.

In practice we could round the flow based on multiple values, not just the single halfway point described above. This would yield, in turn, multiple deadlines, and the best result could then be employed.

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

Corollary 1. *There is a 6-approximation algorithm for minimizing weighted completion time in malleable scheduling with chain precedence constraints, including average stretch.*

Proof. This follows directly by observing that if a 3-speed schedule is executed at unit speed then each completion time scales up by a factor of three.

Minimax Scheduling Here we consider the problem of scheduling chains on P parallel processors under minimax metrics. Recall that there are m chains, each having a non-decreasing cost function $w_j : \mathbb{R}_+ \to \mathbb{R}_+$. The goal is to compute a schedule that minimizes the maximum cost of the m chains.

Theorem 4. *There is a $(1, 2+o(1))$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under minimax cost metrics.*

Recall the definition of Q_js (maximum of critical path and area lower bounds) from (2); and $H = 2m \cdot \lceil \max_{j=1}^{m} Q_j \rceil$ an upper bound on the length of any schedule. The algorithm given below is based on a bracket and bisection search that is a common approach to many minimax optimization problems, for example [12]. It also relies on the algorithm for deadline metrics. See Figure 8.

1: Set $lastsuccess \leftarrow \max_{j=1}^{m} w_j(H)$ and $lastfail \leftarrow 0$.
2: **while** $lastsuccess - lastfail > 1$ **do**
3: Set $L \leftarrow (lastsuccess + lastfail)/2$.
4: **for** $j = 1, \ldots, m$ **do**
5: Compute deadline $D_j := \arg\max\{t : w_j(t) \leq L\}$ for each chain $j \in [m]$.
6: Solve this deadline-metric instance using Algorithm 5.
7: **if** schedule is feasible with 2-speed **then**
8: Set $lastsuccess \leftarrow L$.
9: **else**
10: Set $lastfail \leftarrow L$.
11: Output the schedule corresponding to *lastsuccess*.

Fig. 8: Algorithm for Scheduling Chains with Minimax Objective

Proof. Let opt denote the optimal minimax value of the given instance. Clearly, $0 \leq \mathsf{opt} \leq \max_j w_j(H)$. (We assume that the cost functions w_js are integer valued: this can always be ensured at the loss of a $1 + o(1)$ factor.)

Observe that for any value $L \geq \mathsf{opt}$, the deadline instance in Step 5: the optimal schedule itself must meet the deadlines $\{D_j\}$. Combined with the deadline-metric algorithm (Theorem 2), it follows that our algorithm's schedule for any value $L \geq \mathsf{opt}$ is feasible using 2-speed. Thus the final *lastsuccess* value is at most opt, which is also an upper bound on the algorithm's minimax objective.

As in Corollary 1, the bicriteria guarantees can be combined for some metrics.

Corollary 2. *There is a 2-approximation algorithm for minimizing maximum completion time in malleable scheduling with chain precedence constraints, including makespan and maximum stretch.*

4.3 Stage 3: Converting Pseudo-Job Schedule into Valid Schedule

The final stage combines the output of Stages 1 and 2, converting any malleable schedule of pseudo-jobs and chains into a valid schedule of the original jobs and

flows. We consider the schedule of each pseudo-job k separately. Using a generalization of McNaughton's Rule [17], we will construct a malleable schedule for the (portions of) jobs comprising 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 interval I_k in the pseudo-schedule (recall Stage 1), during which jobs $S \subseteq [n]$ are executed in parallel for a total of $r_k = \sum_{i \in S} \delta_i$ processors. Consider also any malleable schedule of pseudo-job k, that corresponds to a histogram σ (of processor usage) having area $s_k = |I_k| \cdot r_k$ and maximum number of processors at most r_k.

We now describe how to *shape shift* the pseudo-schedule for S in I_k into a valid schedule given by histogram σ. The idea is simple: Decompose the histogram σ into intervals $\mathcal{J}$ of constant numbers of processors. For each interval $J \in \mathcal{J}$, having height (number of processors) $\sigma(J)$, we will schedule the work from a time $\frac{|J| \cdot \sigma(J)}{r_k}$ sub-interval of I_k; observe that the respective areas in σ and I_k are equal. Since $\sum_{J \in \mathcal{J}} |J| \cdot \sigma(J) = s_k = |I_k| \cdot r_k$, taking such schedules over all $J \in \mathcal{J}$ gives a full schedule for I_k. For a particular interval J, we apply McNaughton's Rule to schedule the work from its I_k sub-interval. This rule was extended in [10] to cover a scenario more like ours. It has linear complexity. McNaughton's Rule is basically a wrap-around scheme: We order the jobs, and for the first job we fill the area *horizontally*, one processor at a time, until the total amount of work involving that job has been allotted. Then, starting where we left off, we fill the area needed for the second job, and so on. All appropriate constraints are easily seen to be satisfied.

Figure 9(a) shows a Stage 1 pseudo-schedule and highlights the *first* pseudo-job (interval I_1). The lowest histogram σ of Figure 9(b) illustrates the corresponding portion for this pseudo-job in the Stage 2 malleable greedy schedule; the constant histogram ranges are also shown. The equal area sub-intervals in I_1 are shown as vertical lines in Figure 9(a). Applying McNaughton's Rule to the first sub-interval of I_1 we get the schedule shown at the bottom-left of Figure 9(b). The scheme then proceeds with subsequent sub-intervals.

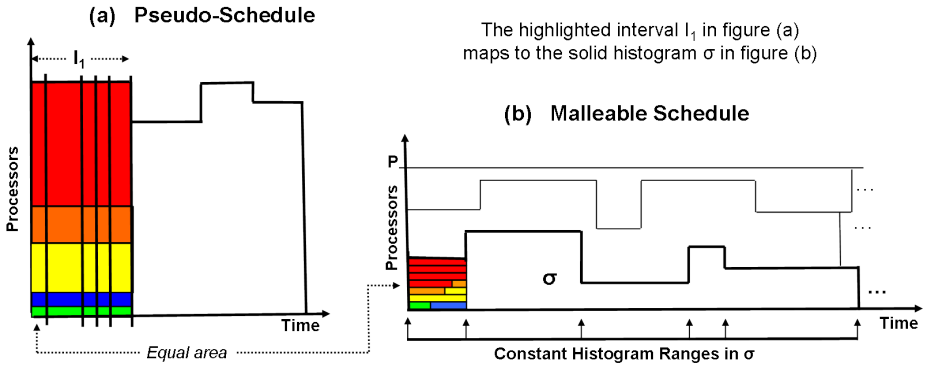

Fig. 9: *FlowFlex* Stage 3: Shape Shifting.

5 Experimental Results

5.1 Simulation Experiments

In this section we describe the performance of our *FlowFlex* algorithm via a variety of simulation experiments. We consider two competitors, *Fair* [26] and *FIFO*. We will compare the performance of each of these three in terms of the best lower bounds we can find for these NP-hard problems. (There is no real hope of finding the true optimal solutions in a reasonable amount of time, but these lower bounds will at least give pessimistic estimates of the quality of the *FlowFlex, Fair* and *FIFO* schedules.) We will consider nearly all combinatorial choices of scheduling metrics, from five basic types. They are based on either completion time, number of tardy jobs, tardiness and SLA step functions. (See Figure 2.) 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 metric.) We note that that these experiments are somewhat unfair to both *Fair* and *FIFO*, since both are completely agnostic with respect to the metrics we consider. But they do at least make sense, when implemented as ready-list algorithms. (In other words, they simply schedule all ready jobs by either *Fair* or *FIFO* rules, repeating as new jobs become ready.) We chose not to compare *FlowFlex* to *Flex*, because that algorithm *does* optimize to a particular metric, and it is not at all obvious how to "prorate" the flow-level metric parameters into a set of per job parameters.

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 a bound. For minimax problems the maximum of the critical path objective function values provides a lower bound. But we can also potentially improve this bound based on the solution found via the 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 form a variety of parameters

governed whether the flow itself was "big" in volume, and also whether the jobs in that flow were "tall" and/or "wide" (that is, having maximum height equal to the number of slots). 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 metrics, where the weights are predetermined by the size of the flow.) Similarly, in the case of SLA metrics, the number of steps and the incremental step heights were chosen from similar distributions with a maximum of 5 steps. Single deadlines for the tardy and tardiness cases was 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 of slots was set to 25.

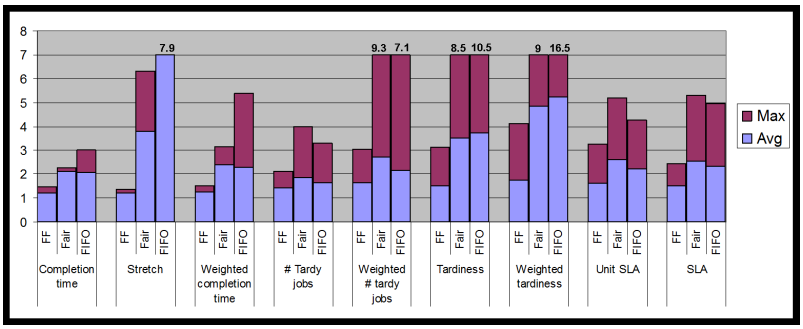

Fig. 10: Minisum Simulation Results: Average and Worst Case.

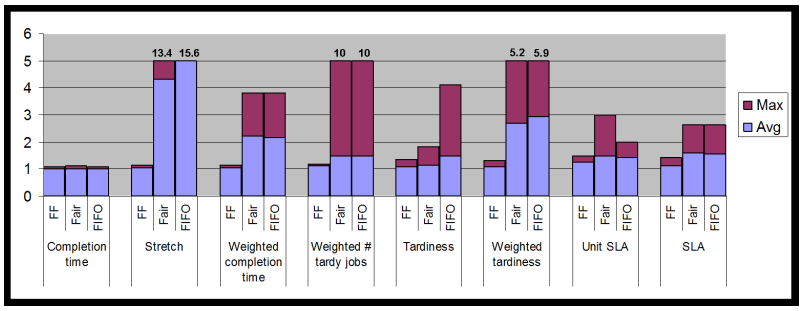

Fig. 11: Minimax Simulation Results: Average and Worst Case.

Figure 10 illustrates both average and worst case performance (given 25 simulation experiments each) for 9 minisum metrics. Each column represents the ratio of the *FlowFlex, Fair* or *FIFO* algorithm to the best lower bound available.[10] Thus 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

[10] To deal with lower bounds of 0, which is possible for some metrics, we added 1 to both the numerator and denominator. The effect is typically quite modest.

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 metrics. Similarly, Figure 11 illustrates the comparable minimax experiments, for those 8 metrics 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, and some of these are *very* difficult problems. Again, some of the *Fair* and *FIFO* ratios can be quite bad. In all 8 sets of experiments, *FlowFlex* is within 1.26 of "optimal" on average, and generally quite a lot better.

5.2 Cluster Experiments

We have prototyped *FlowFlex* on the IBM Platform Symphony MapReduce framework with IBM's BigInsights product [5].

We used a workload based on the standard Hadoop Gridmix2 benchmark. 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 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.

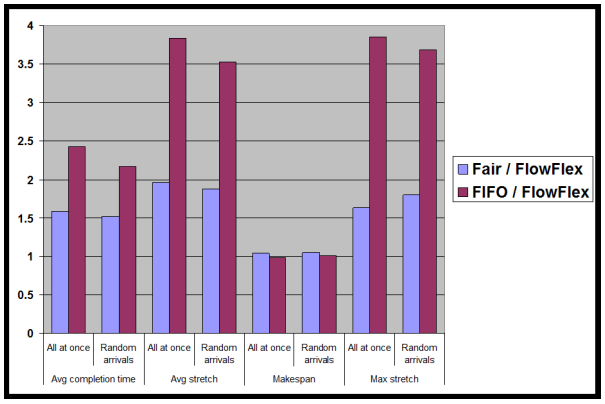

Fig. 12: Cluster Experiments, Gridmix2-based Live Benchmark.

In these cluster experiments, the schedulers are running in something more like their natural 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 complete (theoretical) schedules for internal consumption, but also, more importantly, initial allocation suggestions. This is then implemented to the extent possible by the Assignment Layer.

A few comments should be mentioned here. First, we have not yet focused on integrating 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 can be improved in the future, for example by incorporating the techniques in [19]. Better estimates should improve the quality of our *FlowFlex* scheduler. The second comment is that we are using a slight variant of *FlowFlex* for minisum problems. This variant avoids the minimum cost flow problem. It is faster and approximately as effective. The third 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. (They were not aware of jobs that were not yet submitted.) Figure 12 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 scheduling metrics.

6 Conclusion

In this paper we have introduced *FlowFlex*, a MapReduce scheduling algorithm of both theoretical and practical interest. Theoretically, we have extended the literature on malleable parallel scheduling with precedence constraints, linear speedup and processor maxima. We have provided a unified three-stage algorithm for any scheduling metric, and given worst-case performance bounds. These include approximation guarantees where possible, and bicriteria approximation guarantees where not. Practically, *FlowFlex* is the natural and ultimate extension of *Flex*, a MapReduce scheduler for singleton jobs already in use in IBM's BigInsights. We have evaluated *FlowFlex* experimentally, showing its excellent average case performance on all metrics. And we have shown the superiority of *FlowFlex* to natural extensions of both *Fair* and *FIFO*.

References

1. P. Agrawal, D. Kifer and C. Olston: Scheduling Shared Scans of Large Data Files, Proceedings of VLDB, 2008.
2. A. Balmin, K. Hildrum, V. Nagarajan and J. Wolf, Malleable Scheduling for Flows of MapReduce Jobs, Research Report RC25364, IBM Research, 2013.
3. J. Berlinska and M. Drozdowski, Scheduling Divisible MapReduce Computations. Journal of Parallel and Distributed Computing, 71, 450–459, 2011.
4. 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, 2011.
5. BigInsights: www-01.ibm.com/software/data/infosphere/biginsights/
6. 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.

7. W. De Pauw, J. Wolf and A. Balmin, Visualizing Jobs with Shared Resources in Distributed Environments, IEEE Working Conference on Software Visualization, Eindhoven, Holland, 2013.
8. J. Dean, J. and S. Ghemawat: Mapreduce: Simplified Data Processing on Large Clusters. ACM Transactions on Computer Systems, 51(1),107–113, 2008.
9. M. Drozdowski, Scheduling for Parallel Processing, Springer, 2009.
10. M. Drozdowski and W. Kubiak, Scheduling Parallel Tasks With Sequential Heads and Tails, Annals of Operations Research, 90, 221–246, 1999.
11. 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, 2009.
12. D.S. Hochbaum and D.B. Shmoys, A Unified Approach to Approximation Algorithms for Bottleneck Problems, J. ACM, 33(3), 533-550, 1986.
13. B. Kalyanasundaram and K. Pruhs, Speed is as Powerful as Clairvoyance, J. ACM, 47(4), 2000, 617-643.
14. H. Karloff, S. Suri and S. Vassilvitskii, A Model of Computation for MapReduce, In SODA, 938-948, 2010.
15. P. Koutris and D. Suciu, Parallel evaluation of conjunctive queries, In PODS, 223-234, 2011.
16. J. Leung, Handbook of Scheduling, Chapman and Hall/CRC, 2004.
17. R. McNaughton, Scheduling with Deadlines and Loss Functions, Management Science, 6(1), 1–12, 1959.
18. B. Moseley, A. Dasgupta, R. Kumar and T. Sarlós, On Scheduling in Map-Reduce and Flow-Shops, In SPAA, 289-298, 2011.
19. A. Popescu, V. Ercegovac, A. Balmin, M. Branco and A. Ailamaki, Same Queries, Different Data: Can We Predict Runtime Performance?, In ICDE Workshops 2012: 275-280
20. 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, in ICDE 2010: 996-1005
21. 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.
22. 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.
23. J. Turek, J. Wolf and P. Yu: Approximate Algorithms for Scheduling Parallelizable Tasks, in SPAA, 1992.
24. 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), 2012.
25. 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, in Middleware, 2010.
26. 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.
27. 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, in EuroSys, 2010.