RC25364 (WAT1303-075) March 19, 2013
Computer Science

IBM Research Report

Malleable Scheduling for Flows of MapReduce Jobs

Andrey Balmin
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Kirsten Hildrum, Viswanath Nagarajan, Joel Wolf
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Malleable Scheduling for Flows of MapReduce Jobs

Andrey Balmin* Kirsten Hildrum† Viswanath Nagarajan† Joel Wolf†

Abstract

We consider the problem of scheduling MapReduce workloads. A workload consists of multiple independent *flows*, and each flow is itself a set of MapReduce jobs with precedence constraints. We model this as a parallel scheduling problem [5, 9], more specifically as precedence constrained *malleable* scheduling with *linear speedup* and *processor maxima*. Each flow is associated with an arbitrary cost function that describes the cost incurred for 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. We use *resource augmentation* analysis to provide a $(2, 3)$ bicriteria approximation algorithm for general minisum objectives, and a $(1, 2)$ bicriteria approximation algorithm for minimax objectives. We note that (unless P=NP) no finite approximation ratio is possible without resource augmentation. As corollaries of these results, we obtain a 6-approximation algorithm for *total weighted completion time* (and thus average completion time and average stretch), and a 2-approximation algorithm for *maximum weighted completion time* (and thus makespan and maximum stretch).

We also demonstrate via extensive simulation experiments the overall performance of our algorithms relative to optimal and also relative to other, standard MapReduce scheduling schemes.

*IBM Almaden Research Center. Email: abalmin@us.ibm.com

†IBM T.J. Watson Research Center. Email: {hildrum, viswanath, jlwolf}@us.ibm.com

1 Introduction

MapReduce [4] 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 [17, 18, 16, 1, 15, 11]. All of the schedulers to date have quite reasonably 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. Each flow can be represented by a directed acyclic graph (DAG) in which the nodes are singleton MapReduce jobs and the directed arcs represent precedence. Significantly, the 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 a set of scheduling algorithms for flows of MapReduce jobs. We will consider a variety of scheduling metrics, each of which is based on the completion times of the flows. Examples include makespan, average completion time, average and maximum stretch and metrics involving one or more deadlines. Any given metric will be appropriate for a particular scenario, and the algorithm we apply will depend on the choice of metric. For example, in a batch environment one cares about makespan, to ensure that the batch window is not elongated. In an interactive environment users would typically care about completion time. 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 so-called *malleable* parallel scheduling with precedence constraints. At the same time, we produce a highly generic and practical MapReduce scheduler for flows of jobs, which we call *FlowFlex*, and demonstrate its capabilities experimentally. The closest previous scheduling work for MapReduce jobs appeared in [16]. The Flex scheduler presented there is incorporated in IBM BigInsights [2]. 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 [17, 18], essentially overriding its decisions while simultaneously making use of its infrastructure. See also the special purpose schedulers [1] and CircumFlex [15], built to amortize shared Map phase scans.

We note that the theory of malleable scheduling fits the reality of the MapReduce environment quite closely and accurately. In order 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 width corresponds to the number of processors p, whose height 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 [3], focused on the makespan metric, providing some of the original *approximation* algorithms. (These are polynomial time schemes with guaranteed performance bounds.) Makespan was an appropriate metric for the batch workloads of the time. It was also mathematically simpler to analyze than some of the others.

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 width 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 [14]. In a different direction, [12] found the first approximation algorithm for both rigid and moldable scheduling problems with a (weighted) average completion time

metric. This metric was more appropriate for the interactive workloads, but harder to analyze.

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 [5, 9] has focused on the special case where the speedup function is linear through a given maximum number of processors, and constant thereafter. It is a tautology that 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 [5, 9] 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? 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, and that is where the mathematical complexity resides. The Assignment Layer then implements these allocation decisions (to the extent possible) 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. (There is one minor wording issue: In MapReduce, the atomic unit of allocation is called a *slot*, and there are typically some small constant number of MapReduce slots per core in the cluster. So the word processor in the theoretical literature corresponds to the word slot in a MapReduce context.)

Secondly, 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. 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 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.

One final reason is that maximum constraints occur naturally, either because the particular job happens to be small (and thus have few tasks), or at the end of a normal job, when only a few tasks remain to be allocated. We note, by the way, that job minima were a key aspect of the Fair [17] scheduler, because the goal was to avoid the starvation inherent in a simple scheduling scheme like First in, First Out (FIFO). But Flex [16] demonstrates convincingly that the right way to avoid starvation is actually to integrate the slot (resource) and time dimensions rather than to focus only on the former. Although Flex and CircumFlex[15] support job minima, they are not necessary and not considered in this paper.

We note also that Moseley et al. [11] models the MapReduce setting differently (as a "two-stage flexible flow shop" [13]), and gives approximation and online algorithms for total completion time of independent

MapReduce jobs. Compared to our model (which itself builds on [16]), we observe that [11] handles precedence at the MapReduce task level. We model interconnected MapReduce jobs instead, with fully general cost functions. So the results in [11] and ours are incomparable.

1.1 The Model

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.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 $p \in [\delta_i]$ 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, [5].)

- Total load (or *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.

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 [5, 9] this scheduling environment is $P|var, p_i(k) = \frac{p_i(1)}{k}, \delta_i, prec|*$.[1] 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. (*Stretch* is a fairness metric in which each flow weight is the reciprocal of the size of the flow.) Figure 2 illustrates 4 basic types of cost functions.

1.2 Our Results

We provide approximation algorithms for the above malleable scheduling problems. It turns out that even under very special precedence constraints (chains of length three) the general minisum and minimax problems admit no finite approximation ratio. See Theorem 2.1, which is a consequence of [6]. Hence we use resource augmentation [8] and focus on *bicriteria* approximation guarantees: An algorithm is said to achieve an (α, β)-approximation if it produces a schedule using $\beta \geq 1$ speed processors that has objective value at most $\alpha \geq 1$ times the optimal (under unit speed processors).

Theorem 1.1 *The precedence constrained malleable scheduling problem with linear speedup admits a:*

- $(2, 3)$-*bicriteria approximation algorithm for general minisum objectives.*

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

- $(1, 2)$-*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).*
- $(3 \cdot 2^{1/p})$-*approximation algorithm for ℓ_p-norm of completion times.*

The first two results on general minisum and minimax objectives imply the rest as corollaries. Weighted completion time is perhaps the most useful metric in practice, for which we obtain a 6-approximation ratio. For concreteness, one can also focus on this objective while reading the minisum algorithm.

The main idea in our algorithms (for both minisum and minimax) is a reduction to *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 "guess and verify" framework that is implemented via bracket and bisection search. Simulation results in Section 3 show the excellent performance of our algorithms.

2 The Scheduling Algorithm

Our scheduling algorithm has three sequential stages, described at a high level as follows.

1. First we consider each flow j separately, and covert its (general) precedence constraint into a *chain* precedence constraint.[2] 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.
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.

   At this point, we 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.

2.1 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

[2] Recall that a chain precedence on elements $\{e_i : 1 \leq i \leq n\}$ is just a total order, say $e_1 \prec e_2 \prec \cdots \prec e_n$.

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 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.[3]

Subfigure 3(a) illustrates the directed acyclic graph of a particular flow. Subfigure 3(b) shows the resulting pseudo-schedule. Subfigures 3(c-d) show the decomposition into maximal intervals.

2.2 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) as in Subsection 1.1. To reduce notation, we denote a malleable schedule for *chain* j by a sequence $\tau^j = \langle \tau_1^j, \ldots, \tau_{n(j)}^j \rangle$ 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 $C(\tau^j) := C(\tau_{n(j)}^j)$.

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

Theorem 2.1 ([6]) *Unless P=NP, there is no finite approximation ratio for the general malleable scheduling problem, even with chain precedence constraints of length three.*

The proof appears in the appendix. 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.

[3] 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.

2.2.1 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 2.2 *There is a* $(1, 2)$*-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints and a strict deadline metric.*

Proof: 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 4 in the appendix.

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 $\sigma : [0, \infty) \to \{0, \ldots, P\}$ 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

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

which corresponds to the total work in the first j chains. Since the algorithm always assigns 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 $S(\tau_i^j) \leq t < C(\tau_i^j)$. So

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

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 $C(\tau^j) = A_j + B_j \leq 2 \cdot d_j$.

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. ■

2.2.2 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. We obtain the following bicriteria approximation in this case.

Theorem 2.3 *There is a* $(2, 3 + o(1))$*-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under arbitrary minisum cost metrics.*

Proof: For each chain $j \in [m]$, define $Q_j := \max\left\{\sum_{i=1}^{n(j)} s(k_i^j)/\delta(k_i^j),\ \sum_{i=1}^{n(j)} s(k_i^j)/P\right\}$ to be the maximum of the critical path and 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 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 appendix, we show that both these assumptions can be removed.

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.

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/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.

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:

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

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

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

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

See also Figure 5. We set the demand $\rho := \sum_{j=1}^m V_j$, and compute a minimum cost flow $f : E \to \mathbb{R}_+$. Notice that, by definition of arc capacities, any ρ-unit flow must send exactly V_j units through each node a_j ($j \in [m]$). Moreover, this network flow instance is a valid relaxation of any malleable schedule:

$$\text{Total cost of flow } f \ \leq \ \mathsf{opt}, \text{ optimal value of the malleable scheduling instance.} \tag{3}$$

To prove (3) 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. $C_j \geq Q_j$ and hence edge $(a_j, b_{C_j}) \in E_2$ for all $j \in [m]$. We will prove the existence of a feasible 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 $E_2' := \{(a_j, b_t) \ : \ j \in [m],\ Q_j \leq t \leq C_j\}$. By max-flow min-cut duality, it now suffices to show that the minimum $r - r'$ cut in this N' is at least ρ. Observe that any finite capacity $r - r'$ cut in N' is of the form $\{r\} \cup \{a_j : j \in S\} \cup \{b_t : 1 \leq t \leq \max_{j \in S} C_j\}$, where $S \subseteq [m]$ is some subset of chains. The capacity of edges crossing 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 is at most $P \cdot T$. Hence $P \cdot \max_{j \in S} C_j \geq \sum_{j \in S} V_j$, and combined with (4) implies that the minimum cut in N' is at least $\sum_{j=1}^m V_j = \rho$. This completes the proof of (3).

Obtaining Candidate Deadlines Now we round the flow f to obtain deadlines d_j for each chain $j \in [m]$. We define $d_j := \arg\min \left\{t : \sum_{s=1}^t f(a_j, b_s) \geq V_j/2\right\}$, 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 $\sum_{t \geq d_j} f(a_j, b_t) \geq V_j/2$, we have

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

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 $\sum_{j=1}^m w_j(d_j) \leq 2 \cdot \mathsf{opt}$. Moreover, by definition of the arcs E_2,

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

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

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

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

Solving the Deadline-metric Subproblem Now we apply the algorithm for scheduling with deadline metrics (Theorem 2.2) using the deadlines $\{d_j\}_{j=1}^{m}$ computed above. Notice that we have the two bounds required in the analysis of Theorem 2.2:

- Squashed area: $\frac{1}{P} \cdot \sum_{\ell=1}^{j} V_j \leq 2 \cdot d_j$ for all $j \in [m]$ by (7).
- 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 2.2 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 $\sum_{j=1}^{m} w_j(d_j) \leq 2 \cdot \mathsf{opt}$. ■

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 the case of weighted completion time, the "bicriteria" guarantees can be combined.

Corollary 2.4 *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. ■

2.2.3 Minimax Scheduling

By a similar algorithm we obtain the following result for minimax metrics:

Theorem 2.5 *There is a $(1, 2 + o(1))$-bicriteria approximation algorithm for malleable scheduling with chain precedence constraints under arbitrary minimax cost metrics.*

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 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 [7].) As with the minisum objective in Theorem 2.3, our algorithm here also relies on a reduction to the deadline metric. In fact the algorithm 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-metrics (Theorem 2.2) 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. We start the algorithm with a value M corresponding to the critical path bound.

Setting $\epsilon = 1/m$, the binary search on M requires $\log_2(n \cdot w_{max})$ iterations which is polynomial. ■

As in Corollary 2.4, the bicriteria guarantees can be combined for some metrics.

Corollary 2.6 *There is a 2-approximation algorithm for minimizing maximum completion time in malleable scheduling with chain precedence constraints, including makespan and maximum stretch.*

2.3 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 [10], 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 width (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 [6] 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 *vertically*, 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.

Subfigure 6(a) highlights the *first* pseudo-job from the *last* Stage 1 pseudo-schedule. The right-most histogram of Subfigure 6(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 are shown in Subfigure 6(a). Applying McNaughton's Rule to the first sub-interval we get the schedule shown at the bottom of Subfigure 6(b). The scheme then proceeds with subsequent sub-intervals.

3 Simulation Results

In this section we describe the performance of our FlowFlex algorithm via a variety of simulation experiments. We consider two competitors, Fair [17] and FIFO (first in, first out). 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 width 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.

Table 1 illustrates both average and worst case performance (given 25 simulation experiments each) for 9 minisum metrics. Each row represents the ratio of the FlowFlex, Fair or FIFO algorithm to the best lower bound available.[4] 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 metrics. Similarly, Table 2 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.

4 Conclusion

We introduced a model for scheduling flows of interconnected MapReduce jobs in a malleable parallel scheduling setting. Since no standard approximation ratios are possible for our general problem (unless P=NP), we used resource augmentation to obtain constant-factor bicriteria approximations for both minisum and minimax objectives. We evaluated our algorithms extensively on random problem instances, demonstrating excellent performance relative to lower bounds on the optimum (obtained as byproducts of our algorithms), as well as to other prevalent MapReduce scheduling schemes.

[4] To deal with lower bounds of 0, which is possible for some metrics, we added 1 to both the numerator and denominator. Generally, the effect of this technical fix will be quite modest.

References

[1] P. Agrawal, D. Kifer and C. Olston: Scheduling Shared Scans of Large Data Files, Proceedings of VLDB, 2008.

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

[3] 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.

[4] J. Dean, J. and S. Ghemawat: Mapreduce: Simplified data processing on large clusters. ACM Transactions on Computer Systems, 51(1),107–113, 2008.

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

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

[7] D.S. Hochbaum and D.B. Shmoys, A unified approach to approximation algorithms for bottleneck problems, J. ACM, 33(3), 533-550, 1986.

[8] B. Kalyanasundaram and K. Pruhs, Speed is as powerful as clairvoyance, J. ACM, 47(4), 2000, 617-643.

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

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

[11] B. Moseley, A. Dasgupta, R. Kumar and T. Sarlós, On scheduling in map-reduce and flow-shops, In SPAA, 289-298, 2011.

[12] 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.

[13] 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.

[14] J. Turek, J. Wolf and P. Yu: Approximate Algorithms for Scheduling Parallelizable Tasks, in SPAA, 1992.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

A Missing Proofs

Proof of Theorem 2.1. This follows directly from the NP-hardness of the makespan minimization problem called $P|1any1, pmtn|C_{max}$ due to Drozdowski and Kubiak [6]. 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 total load bound (denoted M).

We create an instance of general cost 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 $t \le M$ 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.

Removing assumptions in proof of Theorem 2.3. We will show that both assumptions 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 $H = 2m \cdot \max_j Q_j$. 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 $N := \log_{1+\epsilon} H$ which is polynomial. We now define the poly-sized flow network on nodes $\{a_1, \ldots, a_m\} \cup \{b_1, \ldots, b_N\} \cup \{r, r'\}$, 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:

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

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

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

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

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 (3) 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 deadline-metric subproblem are also same as the proof in Theorem 2.3. 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.

Minimizing ℓ_p-norm objectives. Our general minisum algorithm (Subsection 2.2.2) is also a $(3 \cdot 2^{1/p})$-approximation algorithm for minimizing the ℓ_p*-norm* of completion times, for any $p \ge 1$. 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 2.3. The algorithm in Theorem 2.3 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.

B Figures And Tables

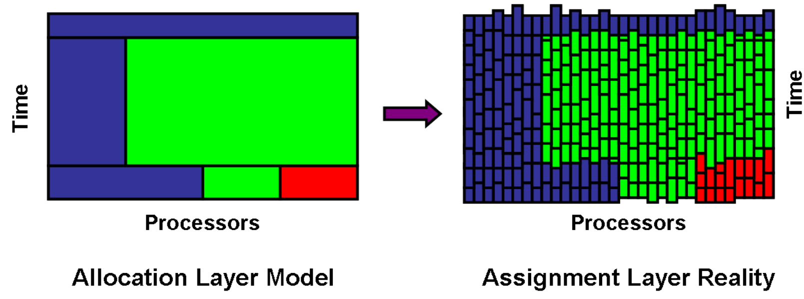

Figure 1: MapReduce and Malleable Scheduling.

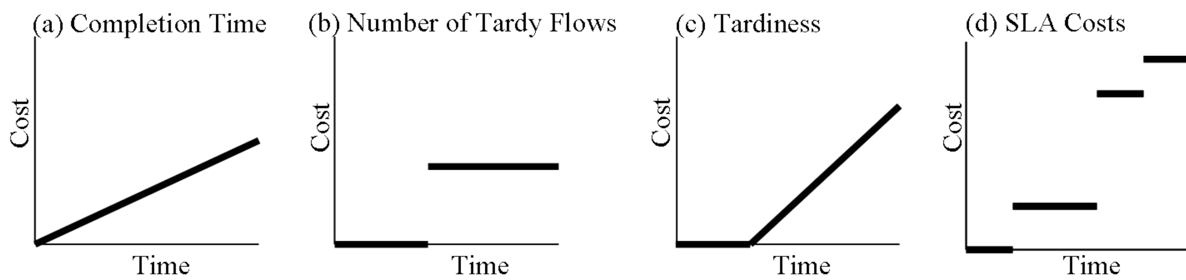

Figure 2: Typical Cost Functions Types.

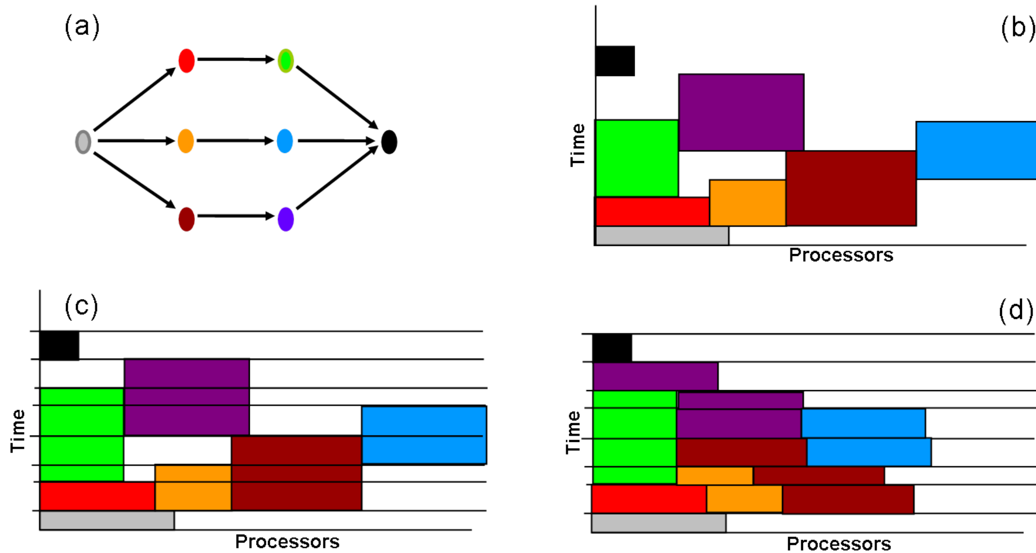

Figure 3: FlowFlex Stage 1.

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 $S(\tau_i^j) \leftarrow C(\tau_{i-1}^j)$ and initialize $\tau_i^j : [0, \infty) \to \{0, \ldots, P\}$ 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: **end for**
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*.
12: **end if**
13: **end for**

Figure 4: Algorithm for Scheduling Chains with Deadline Metric

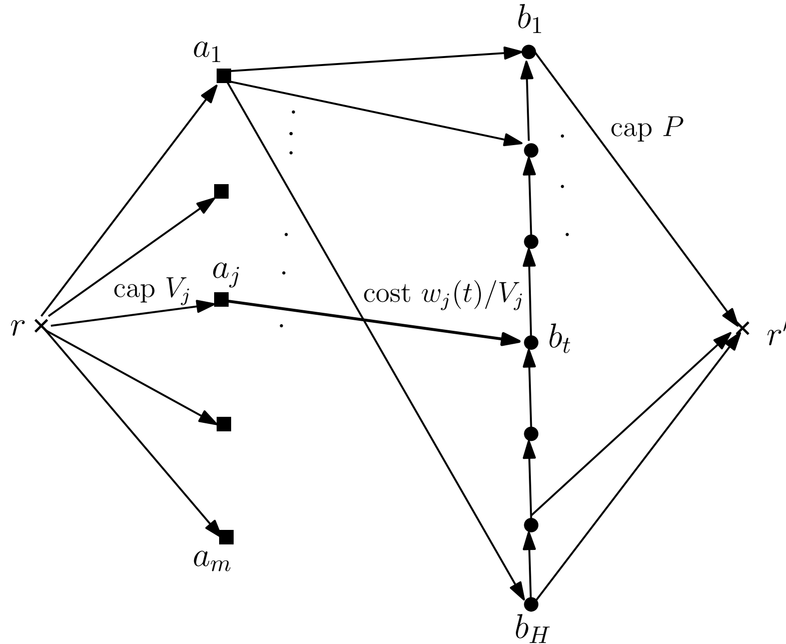

Figure 5: The Minimum Cost Flow Network.

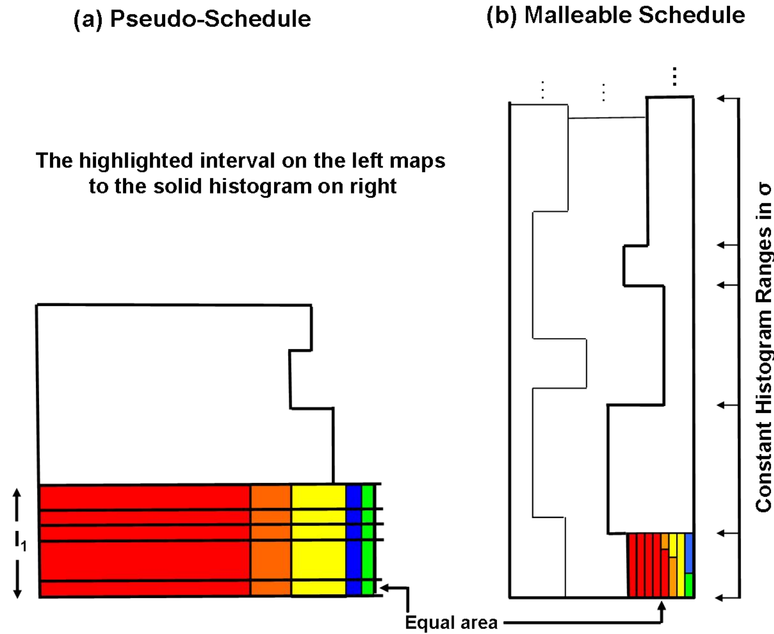

Figure 6: FlowFlex Stage 3.

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)

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)