Scheduling Out-Trees Online to Optimize Maximum Flow

Kunal Agrawal
Washington University in St. Louis
St. Louis, United States
kunal@wustl.edu

Benjamin Moseley
Carnegie Mellon University
Pittsburgh, United States
moseleyb@andrew.cmu.edu

Heather Newman
Carnegie Mellon University
Pittsburgh, United States
hanewman@andrew.cmu.edu

Kirk Pruhs
University of Pittsburgh
Pittsburgh, United States
kirk@cs.pitt.edu

ABSTRACT

We consider online scheduling. on m identical processors. Jobs are parallel programs constructed using dynamic multithreading (also called fork-join parallelism). Jobs arrive over time online and the goal is to optimize maximum flow. Essentially all prior work on this problem has used a relaxed form of analysis where the algorithm has faster speed processors than the optimum and this paper seeks to understand the problem without this strong assumption. We show that the most natural algorithm, First-In-First-Out (FIFO), is $\Omega(\log m)$-competitive for jobs that are out-trees. For this challenging class where jobs are out-trees, we give new clairvoyant algorithm that is $O(1)$-competitive. We then give some circumstantial evidence that FIFO is $O(\log m)$-competitive, even on arbitrary jobs.

CCS CONCEPTS

• **Theory of computation** → *Distributed algorithms*; **Online algorithms**; **Parallel algorithms**.

KEYWORDS

Maximum flow, Dynamic multithreaded jobs

ACM Reference Format:
Kunal Agrawal, Benjamin Moseley, Heather Newman, and Kirk Pruhs. 2024. Scheduling Out-Trees Online to Optimize Maximum Flow. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626183.3659955

1 INTRODUCTION

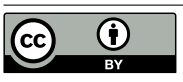
We consider scheduling on identical processors parallel programs that are constructed using dynamic multithreading (also called fork-join parallelism), and that arrive over time, to optimize maximum flow. Dynamic multithreading is common in many parallel languages and libraries, such as Cilk dialects [14, 19], Intel TBB [25], Microsoft Parallel Programming Library [12] and OpenMP [23]. In these parallel languages, programmers express algorithmic parallelism through linguistic constructs such as "spawn" and "sync," "fork" and "join," or "parallel for" loops. In these parallel languages, the resulting programs are naturally modeled by series-parallel directed acyclic graphs (DAGs), where the nodes are atomic computational steps, and a directed edge (u, v) indicates that the predecessor node u needs to be executed before successor node v in order to ensure logical correctness. However, programmers using these parallel languages generally do not explicitly specify how the computation should be parallelized. The task of efficiently parallelizing the computation falls to the run-time scheduler. At any time, a node u is said to be *ready* if all of its predecessor nodes have been executed. A runtime scheduler must select which ready nodes to execute at any time step.

We consider a setting where dynamic multithreaded programs, which we will call jobs, arrive over time. Thus at each time, the run-time scheduler's task is to select nodes, which we will call **subjobs**, from the various digraphs (derived from the programs) to run at that time. See Figure 1 for an example job and possible schedules. The most natural quality-of-service metric for a program/job is its flow (time), which is the duration of time between when the job arrives to be executed and when the job finishes execution. The most natural quality-of-service metrics for a schedule are some norm of the quality-of-service of the individual jobs, with the ℓ_1-norm and the ℓ_∞-norm being the mostly commonly considered norms.

This paper considers the ℓ_∞-norm, which is the maximum flow of any job. Minimizing the maximum flow is usually the standard quality-of-service metric that is simplest for a scheduler to optimize, is the most commonly considered objective when the overriding concern is fairness (as the metric optimizes for the worst-case quality of service over all the jobs). This paper is targeted at addressing the natural open problem, stated for example in the SPAA 2016 paper [4], of whether there is a run-time scheduling algorithm for dynamic multithreaded jobs, that is $O(1)$-competitive with respect to (minimizing) maximum flow.

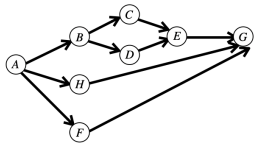
The scheduling problem of scheduling multiple jobs on a set of processors can be thought of as a two-level scheduling problem: the scheduler must decide how many processors to allocate to each job and then decide which subjobs of each job to process on the allocated processors. Intutively, when we want to minimize maximum flow, we should process older jobs first; this intuition is also stated by the authors of [4]:

> " . . . intuitively FIFO is the "right" scheduling policy for maximum flow time."



SPAA '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3659955



m_1	A	B	C	D	E	G
m_2			F	H		
m_3						

m_1	A	B	C	E	G	
m_2		H	D			
m_3		F				

Figure 1: Two possible packings for one job (DAG) on three processors, respecting the DAG structure.

Roughly speaking, prior work has established that FIFO is the "right" algorithm for scheduling sequential programs that arrive over time, and scheduling fully parallelizable programs that arrive over time when we are optimizing for maximum flow time. For fully parallelizable jobs that arrive over time FIFO is optimal with respect to maximum flow. For sequential jobs (i.e., DAGs that are chains) that arrive over time, FIFO is $(3 - \frac{2}{m})$-competitive with respect to maximum flow [6, 8].

For dynamic multithreaded jobs, a FIFO scheduling algorithm can be stated as follows: for every time step, allocate as many processors to the oldest job as it can use (the number of its ready subjobs) and then move on to the second oldest job until either all processors have been allocated or all ready subjobs have been scheduled. However, the last job that is scheduled by this method may get fewer processors than the number of its ready subjobs; therefore, the scheduler must decide which of its ready subjobs to schedule. Our intuition was that the simplest variant, one which arbirarily selects the subjobs, was likely to be "good enough."

This intuition was guided by the fact that for one dynamic multithreaded program, a classic result states that any work-conserving scheduler that doesn't unnecessarily idle a processor is 2-competitive with respect to makespan [16]. This analysis is based on the observation that if the algorithm must idle a processor at some time then the algorithm is reducing the span (length of the longest path) of that job. When we are scheduling multiple dynamic multithreaded jobs, we can generalize this property. For any work-conserving scheduler (a scheduler that doesn't unnecessarily idle processors if there are ready subjobs available), if the scheduler must idle a processor at some time, then the span of every unsatisfied job is reduced at that time. We can call this property the *span reduction property*. Since FIFO is work-conserving, it has this span reduction property. Initially, it seemed to us to be improbable, bordering on implausible, that arbitrary FIFO was not $O(1)$-competitive with respect to maximum flow for dynamic multithreaded jobs that arrive over time.

Lower Bound for FIFO:. Our first contribution, which we summarize in Section 4, is that our significant efforts to show that FIFO is $O(1)$-competitive were in vain, as in fact the competitive ratio of FIFO with respect to maximum flow is $\Omega(\log m)$, where m is the number of processors. This result holds even when the computation structure is an out-tree rather than a general directed acyclic graph. An immediate corollary of this lower bound is that an algorithm prioritizing older jobs and having the span-reduction property are not sufficient to imply $O(1)$-competitiveness even for trees.

New Algorithm for Out-Trees: In light of this lower bound for FIFO, we naturally turned our attention to scheduling dynamic multithreaded jobs that are out-trees. We show in Section 5 that there is an $O(1)$-competitive run-time scheduling algorithm $\mathcal{A}$ for maximum flow, under the assumption that the run-time scheduler learns the shape (subjobs and their dependencies) of the out-tree when it arrives. In some situations, e.g., real-time scheduling applications, this is a reasonable assumption. Despite needing to know the shape of the jobs, we believe the algorithm design and analysis provide insight that will be useful in making further progress on algorithms for scheduling dynamic multithreaded programs.

Out-tree scheduling is interesting, since the structure of many common algorithms, when naturally implemented as dynamic multithreaded programs, are out-trees. In particular, any tail-recursive algorithm, like Quicksort, naturally results in a dynamic multithreaded program that is an out-tree. In addition, many algorithms, such as those that contain a sequence of parallel for-loops, can be thought of as a series of out-trees. One may be able to potentially generalize the out-tree algorithm to such programs as well.

Upper Bound for FIFO for Batched Instances: Finally, given that FIFO is most likely to be used in practice, it is is natural to understand its theoretical performance. Our final contribution, which we cover in Section 6, is to show that FIFO is $O(\log \max\{m, \textsc{Opt}\})$-competitive on what we call batched instances. Here Opt is the optimal maximum flow time, and a batched instance is one where jobs only arrive at times that are integer multiples of Opt.

We now discuss some important insights gained from these theoretical results.

Challenges for designing algorithms for dynamic multithreaded programs: As mentioned above, a scheduler for multiple dynamic multithreaded jobs has two jobs: It must decide how many processors to allocate to each job and must also decide which subjobs to schedule on the allocated processors. Therefore, it faces the problem of both *inter-* and *intra-* job scheduling. It will be useful to view the jobs as geometric forms and a schedule as packing of these forms into a two-dimensional space (formed by the Cartesian product of time with the processors). See Figure 1.

The task of the runtime scheduler is akin to the task faced by a Tetris player in that geometric forms have to be packed as they arrive online without knowledge of the forms that will arrive in the future. However, the task faced by the runtime scheduler is even more daunting. The rules for the ways that a geometric form can be feasibly packed in a schedule are much more complicated than the rules of packing in Tetris. The parallelizability of the programs at a particular time is the number of ready subjobs: therefore, this parallelizability depends significantly on which subjobs the scheduler processed in the past. Therefore, the scheduler's past decisions change the "future" shape of the piece.

Intuitively, the hardest instances for a runtime scheduler are those where it is possible to pack/schedule all the jobs relatively soon after they arrive in such a way that the space/schedule is fully packed. That is, there are never any idle processors. To be competitive on such instances, the runtime scheduler must, after

some finite time, also be able to fully pack jobs relatively soon after they arrive. In other words, the online scheduler can never ever allow a processor to be idle; it has to be able to schedule the jobs so that full parallelism can be achieved on every step. This is because, otherwise, the online scheduler would keep falling behind optimal in terms of the total work scheduled. In particular, if the maximum flow time is to be constant competitive, then after $O(\textsc{Opt})$ time (where $\textsc{Opt}$ is the maximum flow time of the optimal algorithm), the scheduler must fully pack the schedule for these instances.

Our lower bound example for FIFO indicates that this is a fatal flaw. In particular, arbitrary FIFO can make mistakes in *intra-job* scheduling: when it has a smaller number of processors available than the number of ready subjobs for a job, it can choose a subset of subjobs to schedule that reduces future parallelizability. Therefore, FIFO is not able to fully pack the schedule until the amount of outstanding work (the amount of work by which FIFO is lagging behind the optimal scheduler) is quite large, leading to a large competitive ratio.

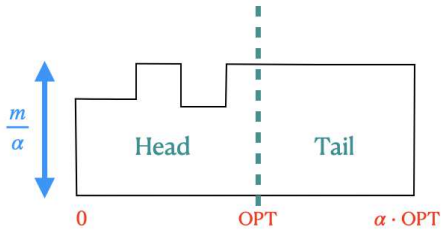


Figure 2: Generic LPF Schedule.

Out-tree Scheduling Algorithm: As we mentioned above, we provide an $O(1)$-competitive algorithm for out-trees. This scheduling algorithm is based on the idea of controlling the shape of our tetris pieces in order to create pieces that can be fit together well. The first step in the design of our algorithm is to show that a simple greedy algorithm we call Longest Path First (LPF), which always prioritizes the subjobs/nodes with the highest height in the DAG, is optimal for a single job. The next step is to observe that the LPF schedule LPF$[m/\alpha]$ on m/α processors (where α is a constant that we will eventually pick in the analysis), has a nice shape. We do not have control over the shape of LPF$[m/\alpha]$ in the first $\textsc{Opt}$ time units, which we call the head of the schedule, but the shape of the portion of LPF$[m/\alpha]$ after time $\textsc{Opt}$, which we call the tail of the schedule, is essentially a rectangle with width m/α processors and length at most $(\alpha - 1)\textsc{Opt}$ units of time. See Figure 2. Our algorithm $\mathcal{A}$ schedules the head of each job using m/α processors when the job arrives, and uses FIFO to prioritize jobs when scheduling the tails of the jobs, and uses the shape of the LPF schedule of each job for intra-job scheduling. Leveraging that the tails have a rectangular shape, we are able to adapt the type of analysis technique that is used for sequential jobs to show that $\mathcal{A}$ is $O(1)$-competitive.

A key takeaway is that, one potential way to design a good algorithm for dynamic multithreaded jobs is to design an the intra-job scheduling policy to nicely "shape" a job. However, for general DAGs (not trees), this shaping is significantly more complicated. In particular, while longest path first is an optimal heuristic for trees for intra-job scheduling, there is no such optimal heuristic for DAGs. Therefore, shaping a DAG is significantly more challenging.

Insights from FIFO upper bound for batched instances: As we mentioned above, for batched instances, where jobs arrive at integer multiples of $\textsc{Opt}$, FIFO is $O(\log \max\{m, \textsc{Opt}\})$-competitive. The starting point for our analysis is the span-reduction property of FIFO, but leveraging this property to obtain an analysis is not straightforward. Our analysis is a rather intricate inductive argument, with a somewhat complicated inductive hypothesis, that captures the ways in which FIFO can be behind the optimal schedule sufficiently accurately to allow the induction to go through. Note that our analysis does not use any assumptions about the structure of the dynamic multithreaded programs, so it holds for jobs that can be modeled by DAGs.

Given this result, we conjecture that out-trees are worst-case instances for FIFO, and FIFO is $\Theta(\log m)$-competitive. It seems improbable to us that instances with arbitrary arrivals are significantly harder for FIFO than batched instances. It further seems improbable that instances in which the optimal maximum flow is super-polynomial in m are significantly harder than instances where the optimal maximum flow is polynomially bounded in m. But we believe the most important takeaway from this result is that our inability, despite significant effort, to show FIFO is $O(\log m)$-competitive on general instances provides further evidence of the importance of designing the runtime scheduler to appropriately "shape" the jobs if one is to have some hope of being able to analyze the runtime scheduling algorithm.

2 RELATED WORK

We now review some closely related work.

Scheduling Dynamic Multithreaded Jobs: Scheduling of dynamic multithreaded programs has been studied extensively, both theoretically and empirically. Many languages and libraries such as Cilk, Cilk Plus [19], Intel's Threading Building Blocks [25], OpenMP [23], and X10 [26], have been designed to allow programmers to write parallel programs. In addition, there has been extensive research on provably good and practically efficient schedulers for these programs in the case where a single job (program) is executing on the parallel machine [9–11]. For the case of a single job, schedulers such as a list scheduler [16] and a work-stealing scheduler [11] are known to be asymptotically optimal with respect to the makespan of the job.

There has also been some theoretical work on scheduling multiple parallel jobs which share a machine [2, 7, 17], but none of this work considers flow time objectives.

Resource Augmentation Results: Scheduling multiple dynamic multithreaded programs that are modeled using directed acyclic graphs has been studied in the context of resource augmentation analysis, and in particular speed augmentation analysis [20]. An s-speed c-competitive algorithm achieves a competitive ratio of c when given processors s times the speed of the optimal schedule. A scalable algorithm is $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for any $\epsilon > 0$ where $f(\epsilon)$ is some function that only depends on ϵ. Intuitively speed augmentation analysis assumes away the existence of the hard instances where the optimal schedule is tightly packed. [4]

showed that FIFO is a scalable algorithm for the objective of maximum flow, in some sense at least arguably establishing that FIFO is the "right" algorithm when the load is not too high. [1, 3] give a scalable algorithm for average flow and [5] provides a scalable algorithm for throughput.

Tree/Forest Scheduling: Algorithmic research on scheduling trees on identical processors goes back to at least the 1960's [18, 22]. A reasonable summary of the research on this problem up to 1980 can be found in [15]. Most relevant here is that [18] shows that the greedy Longest Path First algorithm is optimal for an in-forest. [15] mentions a paper [13] that may contain the result that Longest Path First is optimal for an out-forest, but neither Google nor our university libraries know how to locate this paper.

Speedup Curves: The other standard model of parallelizability of computational tasks is the speed-up curves model (also called the malleable task model). In the arbitrary speed-up curves setting, each job J_j consists of μ_j phases and the ith phase is associated with a tuple $(p_{i,j}, \Gamma_{i,j}(m'))$. The value of $p_{i,j}$ is the work of the ith phase for job j and $\Gamma_{i,j}(m')$ is a speed-up function that specifies the rate $p_{i,j}$ is processed at when job J_j is given m' processors when in the ith phase. The phases of the job must be processed sequentially and $\Gamma_{i,j}$ specifies the parallelizability of J_i during phase i. It is generally assumed that $\Gamma_{i,j}$ is a non-decreasing sublinear function. Note that the DAG model that we consider and the speedup model are fundamentally different from an algorithmic perspective, and there does not appear to be a straightforward way of translating results from one model to the other (for a more in-depth discussion of this issue, see [4, 21]). The result in the speedup curves model that is most relevant here is an $O(1)$-competitive algorithm shown in [21] when the algorithm learns the speed-up curves when a job arrives. Prior to this, a $(1+\epsilon)$-speed $O(\log n)$-competitive algorithm for maximum flow, and a matching general lower bound for any online algorithm, were given in [24].

3 PRELIMINARIES

DAG Model. We represent a dynamic multithreaded job J_i, $i \in [n]$, as a directed acyclic graph (DAG) $G_i = (V_i, E_i)$, where the vertices, which we call *subjobs*, represent some sequential computation and edges represent dependencies between vertices. We assume without any significant loss of generality that each subjob is an atomic computation step that takes unit time to compute. Further, each job J_i has an associated nonnegative integer release/arrival time r_i. (We may occasionally refer to the release time of a subjob of V_i, which is also r_i; this is distinct from the time a subjob becomes "ready," as discussed below.) Since the jobs are independent, the vertices of the G_i's are disjoint, that is $V_i \cap V_j = \emptyset$ if $i \neq j$.

A (feasible) schedule S for a collection $\mathcal{J} = \{J_1, \ldots, J_n\}$ of jobs on m processors over a period of time is an injective function that maps a time in $\mathbb{N}$ to the subcollection of subjobs run at that time, with the following properties

- A most m subjobs are run at any time. That is, for all times t, $|S(t)| \leq m$.
- Every subjob is scheduled exactly once. That is, for every job J_i and every subjob j of J_i, there exists a unique time t such that $j \in S(t)$.
- S respects the precedence constraints. That is, for every job J_i and for every edge $(j,k) \in E_i$, if $j \in S(t)$ and $k \in S(u)$ then $t < u$.
- Every subjob is scheduled after its release time. That is, for every job J_i and for every subjob j of J_i, if $j \in S(t)$ then $t > r_i$.

Due to the nature of the problems we consider, the processor that runs a subjob at a particular unit of time is not relevant. It will be convenient to use J_i and G_i interchangeably. Conceptually, we think of subjobs as being scheduled in unit intervals. So if $j \in S(t)$ then we think of the execution of subjob j as starting at time $t-1$ and finishing/completing at time t. So in the top schedule in Figure 1, $C \in S(3)$. We will sometimes refer to the unit of time between $t-1$ and t in the schedule as *time step* t. Further, if we reference the state of the schedule at time t, we are referring to once time step t has completed.

We will call a subjob j of job J_i *ready* at time t if $r_i \leq t$, all of j's predecessors have been completed by time t, and j has not been completed by time t. At each time t, the online scheduler can select up to m ready subjobs to schedule between time t and $t+1$, i.e., during time step $t+1$.

Maximum Flow Objective. The completion time C_i^S of a job J_i in a schedule S is the maximum completion time of any subjob in J_i in S, that is, C_i^S is the maximum value of t for which $j \in S(t)$ for any subjob j of J_i. The flow (time) F_i^S of a job J_i in a schedule S is $C_i^S - r_i$, that is, the duration of time from when the job is released until when the job is completed. The maximum flow objective for a schedule S is:

$$F_{\max}^S = \max_{i \in [n]} F_i^S.$$

The problem we consider is that of minimizing the maximum flow.

The span P_i of job J_i is the number of vertices in the longest path in G_i. Note that the span is a lower bound on the flow F_i^S of J_i in any schedule S, regardless of the number of processors m. The work W_i of job J_i is the aggregate number of subjobs in J_i, i.e., $W_i = |V_i|$. Note that $\lceil W_i/m \rceil$ is a lower bound on the flow F_i^S of J_i in any schedule S on m processors.

Online Setting. The online/run-time scheduler becomes aware of J_i at time r_i. There are myriad reasonable assumptions about what the online scheduler learns about J_i at time r_i. The one that will be of most concern to us is what we will call a *clairvoyant* scheduler, which means that the scheduler learns the DAG G_i at time r_i (though we will also consider non-clairvoyant schedulers in the batched setting).

We use $\mathcal{A}[I, m]$ to denote the schedule output by algorithm $\mathcal{A}$ on input I using m processors. We use $\textsc{Opt}[I, m]$ to denote the optimal schedule on input I using m processors. It will also be convenient to use $\textsc{Opt}[I, m]$ to denote the optimal maximum flow time for instance I on m processors. If I or m can be readily deduced from context, we may drop them from the notation.

An algorithm $\mathcal{A}$ is *c-competitive* for the objective of minimizing maximum flow if for every input I, $F_{\max}^{\mathcal{A}[I,m]} \leq c \cdot F_{\max}^{\textsc{Opt}[I,m]}$.

FIFO in DAGs. At each time t, the algorithm FIFO schedules an arbitrary collection of subjobs subject to two constraints: (1) if there are fewer than m ready subjobs, then FIFO schedules all the ready

subjobs, and (2) if a ready subjob j is not scheduled at t, then all of the subjobs scheduled at t arrived no later than when j arrived.

4 FIFO LOWER BOUND

In this section, we give a class of instances that shows that FIFO is $\Omega(\log m)$ on jobs that are out-trees.

Lower Bound Instance: A single new job J_i is released at time of the form $i(m+1)$, for $i \in \mathbb{N}$. Each job consists of m layers of at most $m+1$ subjobs. In each of the layers there is a single important subjob, which we call the key subjob, that is a predecessor of each subjob on the next layer. There are other subjobs in the layer and the number of them is to be defined. There are no other precedence constraints. Consider the first time t that FIFO schedules some subjob j on a layer ℓ. Assume that right before FIFO schedules subjob j that there were k remaining available processors. Then the number of subjobs on layer ℓ is defined to be $k+1$, and the key subjob for layer ℓ will be the one that FIFO did not schedule at time t.

Note that the optimal maximum flow for each job J_i, and for the instance as a whole, is at most $m+1$ as one could schedule the key subjob on layer ℓ of each job J_i at time $r_i + \ell$, and schedule arbitrary ready non-key subjobs on the remaining processors.

Consider an arbitrary job J_i, and the times $t_1, t_2 \ldots$ that FIFO runs some subjob from job J_i. Consider time t_h. By construction, if h is even and $h \leq 2m$ then FIFO scheduled only the key subjob from layer $h/2$ at time t_h. Similarly, if h is odd and $h \leq 2m$ then FIFO is running all the non-key jobs from layer $(h+1)/2$ at time t_h. Thus job J_i completes at time $t_{2m} + 1$. We now partition each layer into two sublayers, with one part/sublayer consisting solely of the key subjobs, which we call a sequential sublayer, and the other part/sublayer consisting of all the non-key subjobs, which we call a parallel sublayer. So by construction, we can think of FIFO as processing sublayers, alternating between parallel and sequential. We establish bounds on FIFO in Lemma 4.1 by showing that if the number of unfinished jobs is less than $\lg m - \lg\lg m$, then the number of unfinished jobs will continue to increase.

Lemma 4.1. *Let $U(t)$ be the number of unfinished sublayers of jobs released strictly before time t that are unfinished by FIFO at time t. If $U(t(m+1)) < \lg m - \lg\lg m$ then $U(t(m+1)) < U((t+1)(m+1))$.*

Proof. First, for notational convenience let us renumber the jobs so that the i^{th} oldest job alive at time $t(m+1)$ is numbered i. So the oldest alive job is J_1, and the job that arrived at time $t(m+1)$ is J_ℓ. Let K_i be the number of sublayers that FIFO processed from job i between time $t(m+1)$ and time $(t+1)(m+1)$, and let $S_i = \sum_{j=1}^{i} K_i$ be the number of sublayers that FIFO processed from jobs J_1 through J_i between time $t(m+1)$ and time $(t+1)(m+1)$. Then note that $S_1 = K_1$ and for $h > 1$:

$$S_h - S_{h-1} = K_h \tag{1}$$

$$\leq (m+1) - \sum_{i=1}^{h-1} \left\lfloor \frac{K_i}{2} \right\rfloor \tag{2}$$

$$\leq (m+1) + (h-1) - \sum_{i=1}^{h-1} \frac{K_i}{2} \tag{3}$$

$$= m + h - S_{h-1}/2 \tag{4}$$

The first inequality follows because during the $m+1$ time units between time $t(m+1)$ and $(t+1)(m+1)$ it is the case that job h can not be processed at a time when the parallel sublayer of any earlier arriving job is being processed, and half the sublayers are parallel. And thus

$$S_h - S_{h-1} \leq m + h - S_{h-1}/2 \tag{5}$$

or equivalently by adding S_{h-1} to each side:

$$S_h \leq S_{h-1}/2 + m + h \tag{6}$$

Setting $h = \ell$ and expanding this recurrence yields

$$S_\ell \leq \frac{K_1}{2^{\ell-1}} + \sum_{i=0}^{\ell-2} \frac{m+\ell}{2^i} \tag{7}$$

$$\leq \sum_{i=0}^{\ell-1} \frac{m+\ell}{2^i} \tag{8}$$

$$\leq (m+\ell)(2 - 1/2^{\ell-1}) \tag{9}$$

The first inequality follows because $K_1 \leq m+1$. As each job has $2m$ sublayers, the number of unfinished sublayers for FIFO at time $(t+1)(m+1)$ will be strictly greater than the unfinished sublayers for FIFO at time $t(m+1)$ if

$$(m+\ell)(2 - 1/2^{\ell-1}) \leq 2m \tag{10}$$

Or equivalently

$$\ell(2^\ell - 1) \leq m \tag{11}$$

This inequality holds if $\ell \leq \lg m - \lg\lg m$, □

Theorem 4.2. *The competitive ratio of FIFO is at least $\lg m - \lg\lg m$.*

Proof. After $2m \lg m$ jobs are released, either at some time there were $\lg m - \lg\lg m + 1$ unfinished jobs at some point, or the number of unprocessed sublayers increased by a least 1 each time a job was released. In the former case, the flow time of the oldest job would then be at least $(m+1)(\lg m - \lg\lg m)$. In the latter case, the number of unprocessed sublayers would have to be at least $2m \lg m$. Then as each job contains at most $2m$ sublayers, there must be $\lg m$ unfinished jobs, and thus the flow time of the oldest job would then be at least $(m+1)(\lg m - 1)$. □

5 CLAIRVOYANT ALGORITHM FOR SCHEDULING OUT-TREES

In this section we will be concerned with instances where each job G_i is an out-forest, which is a collection of out-trees. An out-tree is a tree whose edges are directed away from the root. We will adopt standard tree terminology. We use predecessors and ancestors interchangeably, and successors and descendants interchangeably. The height $H(j)$ of a subjob $j \in J_i$ is the number of nodes in the longest path from j to a leaf in J_i; therefore, a leaf has height 1. The depth $D(j)$ of a subjob $j \in J_i$ is the number of nodes in the path from a root of the tree containing j to j. Thus, a root of a tree has depth 1. We use $W_i(d)$ to denote the number of subjobs in J_i with depth strictly greater than d. Throughout this section α will be a positive integer that evenly divides m, and β is also be a positive integer constant. To obtain our competitive algorithm, we will eventually set $\alpha = 4$ and $\beta = 258$.

In subsection 5.1 we show that the clairvoyant algorithm, Longest Path First (LPF), is optimal for maximum flow on one job. We will use LPF as a component in the design of our algorithm $\mathcal{A}$. We also show some properties of the LPF schedule on m/α processors that we will use in the analysis of algorithm $\mathcal{A}$. In subsection 5.2, we give another algorithm Most Children (MC) that we will use as a component in the design of algorithm $\mathcal{A}$, and give some properties of MC that we will use in the analysis of algorithm $\mathcal{A}$. In subsection 5.3, we give an initial version of our clairvoyant algorithm $\mathcal{A}$, that additionally requires a priori knowledge of the optimal objective value Opt and assumes that jobs only arrive at integer multiples of Opt/2, and show that $\mathcal{A}$ is $O(1)$-competitive under these assumptions. Finally, in subsection 5.4 we show how, at the cost of increasing the competitive ratio by an $O(1)$-factor, we can remove the requirement that $\mathcal{A}$ knows Opt a priori, and allow arbitrary release times.

5.1 Longest Path First Algorithm

Algorithm LPF Description: At any time t, assign ready subjobs to processors in order of decreasing height (largest height subjobs are scheduled first) until either all processors have been assigned jobs or there are no ready jobs.

For the rest of this subsection, we consider the following setting. There is a single job J released at time zero. Let Opt be the optimal schedule, and optimal maximum flow time, for J on m processors. Let $S = \text{LPF}(J, m/\alpha)$ be the schedule produced by LPF using m/α processors on input J. In Lemma 5.1 we give a natural lower bound on Opt. In Lemma 5.2 we give a structural property of S (the LPF schedule) that will later be useful in our analysis of algorithm $\mathcal{A}$. In Lemma 5.3 we prove that LPF is optimal on m processors, and α-competitive on m/α processors. In Corollary 5.4 we note that our proof of Lemma 5.3 establishes that the lower bound in Lemma 5.1 is in fact tight for an out-forest where all the jobs arrive at the same time.

Lemma 5.1. *Let d be a nonnegative integer parameter such that there exists a node of depth d in J. Then $\text{Opt} \geq d + \left\lceil \frac{W(d)}{m} \right\rceil$.*

Proof. No subjob with depth greater than d can be run in the first d time steps, and the duration of time to finish the subjobs with depth greater than d is at least $\lceil W(d)/m \rceil$ because only m subjobs can be run at each time. □

Lemma 5.2. *Let t be any time such that $1 \leq |S(t)| \leq m-1$. That is, this is the last time that the LPF schedule on m/α processors had an idle processor. Then either*

- *each subjob in $S(t)$ is a leaf (and thus $F^S_{\max} = t$, meaning the job completes on this time step), or*
- *for each time $s < t$ and for each subjob $j \in S(t)$ that is not a leaf, there is subjob $k \in S(s)$ such that k is the ancestor $t-s$ hops from j in J (so $t-s$ hops towards the root of J from j).*

Proof. For $i \geq 1$, let j^i denote the ancestor of j that is i hops from j. To reach a contradiction, let s be the latest time, strictly before time t, when there is a subjob $j \in S(t)$ such that j is not a leaf in J, and the ancestor j^{t-s} is not in $S(s)$. Because s is chosen maximally, ancestors j^{t-s-1} through j^1 execute at time steps $s+1$ through $t-1$, respectively. Moreover, j^{t-s} is scheduled at some time step $s' < s$, because it is not scheduled at s but j^{t-s-1} is scheduled, and thus ready, at $s+1$. Since j^{t-s} is scheduled at s' and $s' < s$, j^{t-s-1} must have been ready at s; however, it was not scheduled at s. Due to the longest path property, this can only happen if for every subjob $k \in S(s)$ it is the case that

$$H(k) \geq H(j^{t-s-1}) = H(j) + (t-s-1)$$

as otherwise $\mathcal{A}$ would have run j^{t-s-1} at time s.

We now break the proof into two cases. In the first case assume that for every subjob $k \in S(s)$ it is the case that some descendant of k is ready at time t. But that is a contradiction to S having an idle processor at time t since then m/α jobs would be executing at time t. In the second case assume there is a subjob $k \in S(s)$ that does not have a descendant subjob ready at time t. But then it must be the case that $H(k) \leq (t-s)$ since the leaf descendant of k was scheduled at time $t-1$ (or earlier). Combining the two bounds on $H(k)$ gives that $H(j) \leq 1$, contradicting that j is not a leaf. □

Lemma 5.3. *LPF on m/α processors is α-competitive with respect to the optimal algorithm on m processors. That is, $F^S_{\max} \leq \alpha \text{Opt}$.*

Proof. Let t be the last time, strictly before time $F^S_{\max}$, such that there is an idle processor at time t in S. If no such t exists then $F^S_{\max} = \lceil \alpha W(0)/m \rceil$. As α evenly divides m, it just be the case that $\lceil \alpha W(0)/m \rceil \leq \alpha \lceil W(0)/m \rceil$. Thus the claim follows by Lemma 5.1.

Otherwise, by Lemma 5.2 it must be the case that all subjobs $j \in S(t)$ have depth t in J. Thus, as all jobs with depth at most t are finished by time t in S, and there are no idle processors in S between time t and time $F^S_{\max}$, it must be the case that

$$F^S_{\max} - t \leq \lceil \alpha W(t)/m \rceil \leq \alpha \lceil W(t)/m \rceil.$$

Thus $F^S_{\max} \leq t + \alpha \lceil W(t)/m \rceil$. And thus again the claim follows by Lemma 5.2. □

Corollary 5.4. *Let D be the maximum depth of a subjob in J. Then*

$$\text{Opt} = \max_{d \in [0,D]} \left(d + \left\lceil \frac{W(d)}{m} \right\rceil \right)$$

Proof. The fact that Opt is lower bounded by the righthand side follows from Lemma 5.1, and the fact that $F^{\text{Opt}}_{\max}$ is upper bounded by the righthand side is implicit in the proof of Lemma 5.3. □

5.2 The Maximum Children Algorithm

The Maximum Children (MC) algorithm is an online algorithm that schedules jobs over time. However the initial input to the Maximum Children (MC) algorithm is a feasible schedule S of an out-forest job J on m/α processors, with the property that the only time that S has an idle processor is at time $F^S_{\max}$. The MC algorithm's task it to schedule all the subjobs in S over time. At each time t, the MC algorithm learns the number of processors $m_t \leq m/\alpha$ that are available to schedule jobs at time t.

Algorithm MC Description: At each time time t the MC algorithm iterates the following process over every processor available at time t (in arbitrary order). To determine the subjob that MC will run on that processor, let ℓ be minimum such that there are unprocessed jobs in $S(\ell)$, and let j be an unprocessed subjob in $S(\ell)$ with a maximum number of children in $S(\ell+1)$. Then subjob j is scheduled on that processor at time t. So intuitively, MC first prioritizes subjobs scheduled earlier in S, and then prioritizes subjobs with more subjobs scheduled at the next time step in S.

In Lemma 5.5 we establish that until it finishes all the subjobs, MC always keeps the allocated m_t processors busy. Intuitively, MC achieves this by ensuring that as many children as possible are enabled for the next time step.

Lemma 5.5. *For each time t, either MC finishes processing all the subjobs in S by time t, or MC schedules m_t subjobs at time t.*

Proof. To reach a contradiction, let t be the first time when this statement is not true. Let ℓ be smallest step in S such that $S(\ell)$ has a subjob that MC did not process before time t. Let U be the collection of subjobs in $S(\ell)$ not processed by MC before time t. If $m_t \le |U|$, then this clearly is a contradiction as all ancestors of jobs in $S(\ell)$ must be scheduled strictly before time ℓ in S. So now let us assume $m_t > |U|$. If $S(\ell+1)$ is empty, then MC finishes processing the subjobs in S at time t, which is a contradiction. So let us now assume that $S(\ell+1)$ is not empty.

Note, by the definition of the MC algorithm, that every subjob j in $S(\ell) - U$ has a child in $S(\ell+1)$, or no subjob in U has a child in $S(\ell+1)$. To see this, note that if such a j did not have a child in $S(\ell+1)$, then neither can any of the other subjobs in $S(\ell)$ that MC chooses later. Thus the MC algorithm will schedule the remaining subjobs from U, and $\min(m_t - |U|, |S(\ell+1)|)$ subjobs from $S(\ell+1)$, at time t. If $m_t - |U| > |S(\ell+1)|$ then $\ell+1$ is the last time subjobs are scheduled in S, and then MC must finish scheduling the subjobs in S at time t, which is a contradiction. Otherwise, if $m_t - |U| \le |S(\ell+1)|$ then, MC schedules m_t subjobs from S at time t, which is again a contradiction. □

5.3 Super-clairvoyant Algorithm for Semi-batched Out-forest Instances

So far, in this section, we have considered scheduling single jobs. We will now describe our algorithm $\mathcal{A}$ for scheduling multiple jobs on m processors, but for now we make two simplifying assumptions:

- That the online algorithm is super-clairvoyant, which in this setting means that the online algorithm knows a priori the value of Opt.
- That the instance I is semi-batched, which in this setting means that all release times are integer multiples of $\textsc{Opt}/2$.

In the next section, we will show how to remove the necessity of these assumptions. For convenience, we will (without loss of generality) view all the jobs arriving at the same time $i\textsc{Opt}/2$ as being one job J_i. Let $S_i = \text{LPF}(J_i, m/\alpha)$ be the LPF schedule on job J_i on m/α processors. Define the subjobs in S_i scheduled by time Opt to be the head of J_i (S_i), and the rest of the subjobs to be the tail of J_i (S_i).

Algorithm $\mathcal{A}$ Description: At times of the form $i\textsc{Opt}/2$, the algorithm commits to a schedule between this time and time $(i+1)\textsc{Opt}/2$. Note that this description assumes $\alpha > 2$. The scheduling between time $i\textsc{Opt}/2$ and time $(i+1)\textsc{Opt}/2$ is constructed in three sequential phases:

(1) First, on the first m/α processors, the job J_i (the latest arriving job) is scheduled according to S_i. That is, the schedule is identical to the first $\textsc{Opt}/2$ time units of S_i — for $1 \le t \le \textsc{Opt}/2$, the subjobs of J_i scheduled at time t in S_i are exactly the subjobs of J_i run at time $r_i + t$ in $\mathcal{A}(I)$.
(2) On the next m/α processors, we schedule the job J_{i-1} (the second latest job). Again, the schedule for job J_{i-1} is identical to the second $\textsc{Opt}/2$ time units of S_{i-1}. That is, for $1 \le t \le \textsc{Opt}/2$, the subjobs of J_{i-1} run at time $t + \textsc{Opt}/2$ in S_{i-1} are exactly the subjobs of job J_{i-1} run at time $r_i + t$ in $\mathcal{A}(I)$.
(3) Finally, on the remaining processors, we schedule the earlier jobs which are still unfinished. Let $u(1) < \ldots < u(k)$ be integers such that the jobs, other than J_i and J_{i-1}, that are unfinished at time $i\textsc{Opt}/2$, are exactly $J_{u(1)}, \ldots J_{u(k)}$. Then these jobs are scheduled in FIFO order, so $J_{u(j)}$ is scheduled before $J_{u(j+1)}$. The schedule for each $J_{u(j)}$ is constructed using the MC algorithm on the unprocessed portion of $S_{u(j)}$ where m_t is set to the minimum of the remaining number of available processors at time t and m/α.

So jobs J_i and J_{i-1} are given the highest priority and scheduled with the Longest Path First (LPF) algorithm, but on only a limited number (namely m/α) of processors. Then the rest of the unfinished jobs are prioritized in FIFO order, and scheduled using the Most-Children Algorithm on their LPF schedule on m/α processors.

Here, we make an important observation. All the jobs which are executed using the Most-Children algorithm have already been executed using the LPF algorithm for Opt time steps in the past. Therefore, due to Lemma 5.2, the remaining LPF schedule for these jobs does not contain any idle processors (except perhaps on the last step). Therefore, the LPF schedule satisfies the structural property needed for these jobs to be scheduled using MC algorithm.

Theorem 5.6. *Algorithm $\mathcal{A}$ is 129-competitive for maximum flow on out-forests on semi-batched instances.*

Proof. Let I be an arbitrary instance. Assume to reach a contradiction that some job J_i is unfinished at time $r_i + \beta\textsc{Opt}/2$. We now reason exclusively about the schedule $\mathcal{A}(I)$.

Define s to be the earliest time such that there is no idle processor from time s until time r_i. Since the input is semi-batched, there are at most 2 jobs released between times $s - \textsc{Opt}$ and $s - 1$. Because there was an idle processor at time $s - 1$, by Lemma 5.5, there are at most α jobs released before time $s - \textsc{Opt}$ that are unfinished at time s. Thus the unfinished work at time s is at most $(\alpha + 2)m\textsc{Opt}$.

For convenience, we now define jobs to be early if they arrive at time r_i or before, and late otherwise. Note that after time r_i, from the algorithm $\mathcal{A}$'s point of view, the tails of all late jobs have lower priority than the tails of any early job. Further, after time r_i, the heads of the late jobs never run on more than $2m/\alpha$ processors. Thus, after time r_i, the scheduling of the tails of early jobs is unaffected by the tails of the late jobs. So conceptually our analysis ignores the tails of the late jobs.

Now consider $\mathcal{A}(I)$ between time $r_i + \text{Opt}$ and time $r_j + \beta\text{Opt}/2$. Note that all the heads of early jobs have been fully processed by time $r_i + \text{Opt}$. There can be at most $(\alpha - 1)\text{Opt}$ times where there are m/α or more processors that are either idle or processing a subjob from the tail of a late job. Otherwise, by Lemma 5.5 and the fact that S_i uses m/α processors after time Opt, J_i would have been completed by time $r_j + \beta\text{Opt}/2$. Thus there must be

$$(\beta/2 - 1 - (\alpha - 1))\text{Opt}$$

times where at least $(m - 3m/\alpha)$ processors are processing a subjob from the tail of an early job. Thus the work processed between time $r_j + \text{Opt}$ and time $r_j + \beta\text{Opt}/2$ on the tails of the early jobs is at least

$$(\beta/2 - \alpha)(m - 3m/\alpha)\text{Opt}.$$

As $\mathcal{A}(I)$ did not idle any processor between time s and time r_i, the amount of early work is at least $(r_i - s)m\text{Opt}$ plus the work in the tails of the early jobs processed in $\mathcal{A}(I)$ after time r_i, so at least

$$(r_i - s)m\text{Opt} + (\beta/2 - \alpha)(m - 3m/\alpha)\text{Opt}.$$

Since at most $(\alpha + 2)m\text{Opt}$ of the early work was unprocessed by time s, the amount of work arriving between time s and time r_i was at least

$$(r_i - s)m\text{Opt} + (\beta/2 - \alpha)(m - 3m/\alpha)\text{Opt} - (\alpha + 2)m\text{Opt}.$$

So the "excess" work arriving between time s and time r_i is at least

$$(\beta/2 - \alpha)\text{Opt}(m - 3m/\alpha) - (\alpha + 2)m\text{Opt}$$

If this excess work is greater than Opt then this contradicts that there is a feasible schedule with maximum flow time Opt. Note that when $\alpha = 4$ and $\beta > 256$ we do obtain this contradiction. □

5.4 Clairvoyant Algorithm for General Out-tree Instances

We can now show that $\mathcal{A}$ can be used for general instances and not just semi-batched instances. First, we argue that at the cost of a factor of 2 in the competitive ratio, we can assume that the input is semi-batched if we know Opt. Consider an arbitrary instance I. We modify I to create new instance I' in the following way. The job that arrives at time $i\text{Opt}$ in I' is the union over all jobs that arrived between $(i-1)\text{Opt} + 1$ and $i\text{Opt}$ in I. Note that an online algorithm can implement this delay by just ignoring jobs that arrived between $(i-1)\text{Opt} + 1$ and $i\text{Opt} - 1$ until time $i\text{Opt}$. Note that the optimal maximum flow for I' is at most 2Opt, as one feasible schedule for I' is to delay the optimal schedule for I by Opt units of time. Thus the new release times are integer multiples of the new optimal objective value divided by two.

Since this technique requires that the online algorithm knows Opt, now we show how the online algorithm can use the standard guess-and-double technique to remove the assumption that it knows Opt a priori. The online algorithm can maintain a lower bound AOpt to Opt, which is initialized to 1, and use AOpt in place of Opt in the algorithm $\mathcal{A}$. Once that algorithm observes that a job has flow more than $\beta\text{AOpt}/2$, it knows that $\text{AOpt} < \text{Opt}$. The algorithm then doubles AOpt, and restarts itself with the release time of all unfinished jobs, and all future arriving jobs delayed by $\beta\text{Opt}/2$. Now let 2^k be the final value of AOpt. Then we know that $\text{Opt} \geq 2^{k-1}$. Until AOpt reaches 2^k, jobs will be delayed at most $\sum_{h=0}^{k} 2^h\beta/2 \leq 2^k\beta$. And after AOpt reaches 2^k none will be delayed by more than $\beta 2^k/2$. Thus no job will be delayed more than $(3/2)\beta 2^k \leq 3\beta\text{Opt}$, which is 6 times the bound.

So removing the assumptions from the previous subsection costs at most a factor of 12 in the competitive ratio. Thus we can conclude:

Theorem 5.7. *Algorithm $\mathcal{A}$ is 1548-competitive for maximum flow on out-forests.*

6 FIFO UPPER BOUND

This section presents the proof that FIFO is $O(\log \max\{\text{Opt}, m\})$-competititve for *batched* instances in the *non-clairvoyant* setting. Recall that in this setting, jobs arrive at integer multiples of Opt only. For $i \in [n]$, let job i be the job that has release time $r_i = i\text{Opt}$. We may assume only one job arrives at $i\text{Opt}$, by taking a union of DAGs if necessary.

We now sketch what the inductive argument must capture before giving the full formal argument. We wish to show that the total unfinished work of FIFO never falls too far behind the optimal, say by at most a $c \cdot \text{Opt}$ additive amount. If this were true, then one can easily bound the competitiveness of FIFO by at most $O(c)\text{Opt}$. Unfortunately, inducting on just the total work in FIFO and Opt is insufficient as it is important to know *which* subjobs of each job have been processed.

The inductive argument needs to keep track of more structure on the jobs. To this end, the argument will also keep track of how much sequential work (roughly speaking, the amount of completed span) of each job has been completed. The intuition for why this is useful is that Opt must also spend the same amount of time on sequential work, so finished sequential work implies a bound on remaining work (see Lemma 6.4 below). Thus, we will use a strong inductive hypothesis that keeps track of both the work and completed span of unfinished jobs.

Theorem 6.1. *For batched instances, FIFO is $O(\log \max\{\text{Opt}, m\})$-competitive for maximum flow in non-clairvoyant schedulers.*

Notation and Terminology:

- S is the schedule output by FIFO, and S^{Opt} is an optimal schedule. When clear from context, we will simply write Opt instead of S^{Opt}.
- Opt: The maximum flow for OPT. (We are abusing notation here for convenience.)
- Define τ to be the largest number such that $\tau \geq 2m\text{Opt}$ and $\log \tau$ is integral. Note this implies that $\tau < 4m\text{Opt}$.

The remaining terms refer to the schedule S from FIFO.

- $w_i(t)$ denotes remaining work of job i at time t. More formally,

  $$w_i(t) := W_i - |\{j : j \in V_i \text{ and } j \in S(u) \text{ for some } u \leq t\}|$$

  so in particular, $w_i(r_i) = W_i$ and $w_i(C_i^S) = 0$.
- For job i, S_i is the schedule S restricted to jobs that arrived before or at r_i. So for $t > r_i$,

  $$S_i(t) := S(t) \setminus \{j : j \in V_k, r_k > r_i\}.$$

  We say t is *idle in S_i* if $|S_i(t)| < m$ and *complete in S_i* if $|S_i(t)| = m$.

- $z_i(t)$ is the number of idle time steps between times r_i and t in the schedule S_i. More formally, for $r_i \le t \le C_i^S$, we define
$$z_i(t) := |\{u : r_i < u \le t, |S_i(u)| < m\}|.$$
To easily handle corner cases that arise in the proof, we set $z_i(t) = \infty$ when $t > C_i^S$.

The following propositions are straightforward to verify. The first relates idle time steps to sequential work, and we will invoke it repeatedly. The second bounds the remaining work of the optimal schedule as a function of time, and is important for comparing the progress that FIFO makes to the progress that the optimal schedule makes over time.

Proposition 6.2. *Let $i \in [n]$ and t be a time step idle in S_i, with $r_i < t \le C_i^S$. Then $V_i \cap S(t) \neq \emptyset$. Moreover, for each $v \in V_i \cap S(t)$, there exists a directed path in G_i ending in v that contains at least $z_i(t)$ vertices. As a consequence, $z_i(t) \le$ Opt.*

Proposition 6.3. *For any $i \in [n]$ and time $t \ge r_i$, the schedule Opt has at most $m\text{Opt} - m(t - r_i)$ work remaining on J_i at time t.*

The following key lemma says that if FIFO has completed a lot of sequential work in a DAG, it cannot have a huge amount of work left, since OPT had to spend the same amount of time on that sequential work as well. (We note that a weakening of this lemma holds for any algorithm that does not unnecessarily idle processors, by not restricting to the schedule S_i in the definition of $z_i(t)$. However, in analyzing FIFO, we will use crucially that in the definition of $z_i(t)$, we consider the *restricted* schedule S_i.)

Lemma 6.4. *At each time $t \ge r_i$, it must be the case that $w_i(t) \le (\text{Opt} - z_i(t))m$.*

As a sanity check, note that the lemma implies $w_i(t) \le m \cdot \text{Opt}$ always, which is clearly true. Further, the lemma states that if $z_i(t) = \text{Opt}$, then $w_i(t) = 0$. To see why this is true, consider the first time t, call it t_{Opt}, that $z_i(t) = \text{Opt}$. Since t_{Opt} is chosen minimially, t_{Opt} is idle in S_i. This implies that either all nodes in G_i have been executed (i.e., $w_i(t_{\text{Opt}}) = 0$, as desired), or there is a child c_v of a node $v \in V_i \cap S(t_{\text{Opt}})$ that has yet to be scheduled. But by Proposition 6.2, this means c_v is on a directed path with at least Opt + 1 nodes, which is impossible by definition of Opt.

Proof of Lemma 6.4. Fix i. For $k \in \mathbb{N}_{\ge 0}$, let t_k be the smallest t with $r_i \le t \le C_i^S$ such that $z_i(t) = k$. The case $k = 0$ is trivial so assume $k > 0$. It suffices to show that $w_i(t_k) \le (\text{Opt} - z_i(t_k))m$; for, $w_i(t_k) \le (\text{Opt} - z_i(t_k))m$ implies the lemma for all t, since $w_i(t) \le w_i(t_k)$ and $z_i(t_k) = z_i(t)$ for all $t_k \le t < t_{k+1}$.

Define $U_k := V_i \cap S(t_k)$, i.e., U_k is the set of subjobs in job J_i that S works on during time step t_k. Define R_k to be the set of remaining subjobs that S has not scheduled by time t_k, i.e., $v \in R_k$ if $v \in V_i \cap S(u)$ for any $u > t_k$. Also, since t_k is chosen minimally, t_k is idle in S_i. So $U_k \neq \emptyset$, and every $v \in R_k$ is a descendant of some subjob in U_k.

Intuitively, we will show that at time t_k, S cannot be behind (in terms of amount of work remaining) Opt at the time that Opt first started working on U_k. Formally, define t_k^{Opt} to be the first time that the optimal schedule works on some subjob from U_k, that is,
$$t_k^{\text{Opt}} := \min\left\{u > r_i : v \in S^{\text{Opt}}(u) \text{ for some } v \in U_k\right\}.$$
This implies that the set of remaining subjobs that the optimal schedule has not scheduled by t_k^{Opt} includes all descendants of subjobs in U_k, so in particular, includes R_k.

Let $v_k^{\text{Opt}} \in U_k \cap S^{\text{Opt}}(t_k^{\text{Opt}})$ (such v_k^{Opt} exists by definition of t_k^{Opt}). Also, since $v_k^{\text{Opt}} \in U_k$, we have by Proposition 6.2 that $t_k^{\text{Opt}} - r_i \ge z_i(t_k)$. So by Proposition 6.3, there are at most $(\text{Opt} - z_i(t_k))m$ remaining subjobs for the optimal schedule to schedule. So by the last line of the preceding paragraph, $w_i(t_k) = |R_k| \le (\text{Opt} - z_i(t_k))m$, as desired. □

The following lemma is the main lemma, which gives Theorem 6.1 as an immediate corollary.

Lemma 6.5. *Consider any time $t = i\text{Opt}$, i.e., the arrival time of job i. Define $j := i - \log\tau$. Then*

(1) Jobs 0 through $j-1$ are done by time t, i.e., $C_k^S \le t$ for $k < j$.

(2) For $0 \le \ell \le \log\tau - 1$, we have
$$\frac{1}{m} \cdot \sum_{k=j}^{j+\ell} w_k(t) \le \ell\,\text{Opt} + \min_{k=j}^{j+\ell} z_k(t). \tag{12}$$

(3) For $0 \le \ell \le \log\tau - 1$, we have
$$\frac{1}{m} \cdot \sum_{k=j}^{j+\ell} w_k(t) \le \sum_{k=1}^{\ell+1} (1 - 1/2^k)\text{Opt}. \tag{13}$$

Let us remark on some implications of the lemma. First, it states that job $j = i - \log\tau$ is the oldest job still alive at time $t = i\text{Opt}$ (if it is still alive), and the only jobs that may still be alive upon the arrival of job i at time t are the $\log\tau$ jobs $j = i - \log\tau$ through $i - 1$. Second, each inequality indexed by ℓ bounds the total amount of work remaining on the $\ell + 1$ oldest jobs among these $\log\tau$ jobs. Roughly speaking, the inequalities (12) lower bound the amount of sequential work completed in terms of the remaining work; in turn, combining this lower bound with Lemma 6.4 will allow us to deduce the inequalities (13), which give an absolute bound on the remaining work. Both sets of inequalities are inspired by the lower bound in Section 4.

Taking $\ell = 0$, the inequalities upper bound the remaining work on the oldest job j at time t:
$$w_j(t) \le m \cdot \min\{z_j(t), \text{Opt}/2\}.$$
The naive bound on $w_j(t)$ is $m \cdot \text{Opt}$, so we can view the inequalities above as refinements of this bound (as $z_j(t) \le \text{Opt}$ by Proposition 6.2). Moreover, by taking $t' = (i+1)\text{Opt}$, the lemma states that this remaining work $w_j(t)$ on job j is finished in the Opt time steps between t and t'. So the lemma implies Theorem 6.1, because the flow time of any job j is at most $(\log\tau + 1) \cdot \text{Opt}$.

On the other hand, taking $\ell = \log\tau - 1$, the inequalities state that the total remaining work on the $\log\tau$ jobs that may still be alive at time t is at most
$$m \cdot (\log\tau - 1)\text{Opt} + m \cdot \min_{k=j}^{j+\log\tau-1} z_k(t)$$
and also at most
$$m \cdot \sum_{k=1}^{\log\tau} (1 - 1/2^k)\text{Opt}.$$

As before, both bounds can be viewed as refinements of the naive bound, which in this case is $\log \tau \cdot m \cdot \textsc{Opt}$.

Now we prove the main lemma.

Proof of Lemma 6.5. The proof is by induction on i. The base case of $i = 0$ is trivial.

We inductively assume (1) — (3) in the statement of the lemma hold for $t = i\textsc{Opt}$. Now consider time $t' = (i+1)\textsc{Opt}$. As in the statement of the lemma, $j = i - \log \tau$. We must prove that:

(1′) Job j is done by time t'.

(2′) For all $0 \le \ell \le \log \tau - 1$, we have

$$\frac{1}{m} \cdot \sum_{k=j+1}^{j+1+\ell} w_k(t') \le \ell\textsc{Opt} + \min_{k=j+1}^{j+1+\ell} z_k(t') \tag{14}$$

(3′) For all $0 \le \ell \le \log \tau - 1$, we have

$$\frac{1}{m} \cdot \sum_{k=j+1}^{j+1+\ell} w_k(t') \le \sum_{k=1}^{\ell+1}(1 - 1/2^k)\textsc{Opt} \tag{15}$$

Note that we only need to prove in (1′) that job j is done by t', as we know by the inductive hypothesis that jobs 0 through $j-1$ completed by time t, thus also by time t'.

Proof of (2′) for $0 \le \ell < \log \tau - 1$. First, let us consider the inequalities (14) for $0 \le \ell < \log \tau - 1$ (We will consider the last value $\ell = \log \tau - 1$ later.) Note that for this range of ℓ, job i is not counted in any of these inequalities.

This is relatively straightforward. Since $1 \le \ell + 1 \le \log \tau - 1$ for this range of ℓ, we may replace ℓ with $\ell + 1$ in (12) in the inductive hypothesis for t. So

$$\frac{1}{m} \cdot \sum_{k=j}^{j+1+\ell} w_k(t) - \min_{k=j}^{j+1+\ell} z_k(t) \le (\ell+1)\textsc{Opt}. \tag{16}$$

There are Opt time steps between t and t'. Consider the schedule $S_{j+1+\ell}$ (the schedule S restricted to jobs that arrived at or before $r_{j+1+\ell}$). Let $t < u \le t'$, and we say time step u is either complete or idle in $S_{j+1+\ell}$ as defined earlier. If the time step u is complete in $S_{j+1+\ell}$, then $(1/m)(\sum_{k=j}^{j+1+\ell} w_k(u-1))$ reduces by 1 to $(1/m)(\sum_{k=j}^{j+1+\ell} w_k(u))$ in time step u. (Note we are using that by (1) in the inductive hypothesis, S_{j+1+l} is only working on jobs j through $j+1+\ell$.) Otherwise, the time step u is idle in $S_{j+1+\ell}$, so it is also idle in S_k for $j \le k \le j+1+\ell$. This means that $z_k(u) = z_k(u-1) + 1$ (if job k is alive during time step u), so $-\min_{k=j}^{j+1+\ell} z_k(u-1)$ decreases by at least 1 to $-\min_{k=j}^{j+1+\ell} z_k(u)$. (Note that setting $z_k(u) = \infty$ for $C_k^S \le u - 1$ handles the case that job k is not alive during time step u). Therefore, in the Opt time steps between t and t', the left-hand side of (16) reduces by at least Opt, giving (14) for $0 \le \ell < \log \tau - 1$.

Proof of (1′). By similar arguments, we can show that the oldest job, job j, completes in the Opt time steps between t and t'. Consider the schedule S_j. Since inductively jobs 1 through $j-1$ are complete by time t, job j can be the only job scheduled during S_j. So all time steps between t and C_j^S are either idle time steps in S_j, or complete time steps during which m subjobs—all of job j— are scheduled. There are most $\textsc{Opt} - z_j(t)$ idle time steps after time t, by Proposition 6.2. Moreover, by the inductive hypothesis, $w_j(t) \le m \cdot z_j(t)$, so there are at most $z_j(t)$ complete time steps in S_j after time t. Thus the remaining idle and complete time steps in S_j after time t can complete in the Opt time steps between t and t', so job j completes by by time t'.

Proof of (3′) for $0 \le \ell < \log \tau - 1$. Now consider the inequalities from (15). To prove them, we will do an inner induction on ℓ. First we do the base case of $\ell = 0$. In particular, we want to show that $w_{j+1}(t') \le m\textsc{Opt}/2$. Since we showed above that (14) holds for $\ell = 0$, we know that

$$w_{j+1}(t')/m \le z_{j+1}(t').$$

Also, by Lemma 6.4,

$$w_{j+1}(t') \le (\textsc{Opt} - z_{j+1}(t'))m.$$

So combining the two inequalities above gives

$$w_{j+1}(t') \le (\textsc{Opt} - w_{j+1}(t')/m)m$$

and rearranging terms gives the desired bound

$$w_{j+1}(t') \le m\textsc{Opt}/2.$$

Now let us prove this set of inequalities (15) for all $1 \le \ell < \log \tau - 1$. Namely, we want to prove that

$$\frac{1}{m} \cdot \sum_{k=j+1}^{j+1+\ell} w_k(t') \le \sum_{k=1}^{\ell+1}(1 - 1/2^k)\textsc{Opt}.$$

Assume inductively that the inequalities (15) hold up to $\ell - 1$. We will prove (15) holds for ℓ. Observe that

$$\frac{1}{m} \cdot \sum_{k=j+1}^{j+\ell} w_k(t') \le \sum_{k=1}^{\ell}(1 - 1/2^k)\textsc{Opt} \tag{17}$$

$$\frac{1}{m} \cdot \left(\sum_{k=j+1}^{j+1+\ell} w_k(t')\right) - z_{j+1+\ell}(t') \le \ell\textsc{Opt} \tag{18}$$

where (17) holds due to the inductive hypothesis on $\ell - 1$, and (18) follows as we already proved that the inequalities (14) hold for $0 \le \ell < \log \tau - 1$.

Assume for a contradiction that

$$\frac{1}{m} \cdot \sum_{k=j+1}^{j+1+\ell} w_k(t') > \sum_{k=1}^{\ell+1}(1 - 1/2^k)\textsc{Opt}.$$

Then combining the above inequality with (18) gives

$$z_{j+1+\ell}(t') \ge \left(-\ell + \sum_{k=1}^{\ell+1}(1 - 1/2^k)\right) \cdot \textsc{Opt}$$

$$= \left(\ell + 1 - \ell - \sum_{k=1}^{\ell+1}(1/2^k)\right) \cdot \textsc{Opt} = \frac{1}{2^{\ell+1}} \cdot \textsc{Opt}$$

Now plugging this bound into Lemma 6.4, we have

$$\frac{1}{m} \cdot w_{j+\ell+1}(t') \le (1 - 1/2^{\ell+1})\textsc{Opt}.$$

Adding the above inequality to inequality (17) gives

$$\frac{1}{m} \cdot \sum_{k=j+1}^{j+1+\ell} w_k(t') \le \sum_{k=1}^{\ell+1}(1 - 1/2^k)\textsc{Opt}$$

which contradicts our assumption.

Proof of (2′) for $\ell = \log\tau - 1$. Now we prove (14) for the last value of $\ell = \log\tau - 1$; this inequality counts job i, the job that arrives at time t. We will first show that

$$\frac{1}{m} \cdot \sum_{k=i-\log\tau}^{i} w_k(t) \le \log\tau \cdot \textsc{Opt}.$$

To see this, note from (13) we have that

$$\sum_{k=i-\log\tau}^{i-1} w_k(t) \le m \cdot \sum_{k=1}^{\log\tau} (1 - 1/2^k)\textsc{Opt} = m \cdot (\log\tau - 1 + 1/\tau)\textsc{Opt}$$

and since the left-hand side is integral,

$$\sum_{k=i-\log\tau}^{i-1} w_k(t) \le \lfloor m \cdot (\log\tau - 1 + 1/\tau)\textsc{Opt} \rfloor.$$

Since $w_i(t) = W_i \le m\textsc{Opt}$, and $m\textsc{Opt}$ is integral,

$$\sum_{k=i-\log\tau}^{i} w_k(t) \le \lfloor m \cdot (\log\tau - 1 + 1/\tau)\textsc{Opt} + m\textsc{Opt} \rfloor \tag{19}$$

so

$$\sum_{k=i-\log\tau}^{i} w_k(t) \le \lfloor m \cdot (\log\tau + 1/\tau)\textsc{Opt} \rfloor$$

and now using that $m \cdot \log\tau \cdot \textsc{Opt}$ is an integer,

$$\sum_{k=i-\log\tau}^{i} w_k(t) \le m \cdot \log\tau \cdot \textsc{Opt} + \lfloor m\textsc{Opt}/\tau \rfloor$$

and since $\tau \ge 2m\textsc{Opt}$, $\lfloor m\textsc{Opt}/\tau \rfloor = 0$, so we have

$$\frac{1}{m} \cdot \sum_{k=i-\log\tau}^{i} w_k(t) \le \log\tau \cdot \textsc{Opt}$$

and thus also

$$\frac{1}{m} \cdot \sum_{k=i-\log\tau}^{i} w_k(t) - \min_{k=i-\log\tau}^{i} z_k(t) \le \log\tau \cdot \textsc{Opt}.$$

Now we can use the same argument as in the proof of (2′) for $0 \le \ell < \log\tau - 1$, and conclude that

$$\frac{1}{m} \cdot \sum_{k=i-\log\tau+1}^{i} w_k(t') - \min_{k=i-\log\tau+1}^{i} z_k(t') \le (\log\tau - 1) \cdot \textsc{Opt}$$

which is precisely the inequality (14) for $\ell = \log\tau - 1$.

Proof of (3′) for $\ell = \log\tau - 1$. Recall from the proof of (3′) for $0 \le \ell < \log\tau - 1$ that the only reason we could not extend the induction to $\ell = \log\tau - 1$ was becuase we had not yet established inequality (14) or $\ell = \log\tau - 1$. Now that we have established this in the previous paragraph, the induction extends.

This concludes the proof.

□

Remark. The batched arrival assumption is used crucially in the proof, as it implies that at most $m\textsc{Opt}$ new work can arrive in a period of Opt time steps. Even relaxing this assumption slightly (e.g., new jobs can arrive only every $\textsc{Opt}/2$ time steps, which means that at most $m \cdot 3\textsc{Opt}/2$ arrives in Opt time steps) causes the current proof to break down; see line (19).

7 CONCLUSION

This paper has addressed the open problem from the SPAA 2016 paper [4] on what the best competitive ratio is with respect to (minimizing) maximum flow for scheduling multithreaded jobs online. This paper showed perhaps surprisingly that the most natural algorithm, FIFO, is not constant competitive. We develop a new clairvoyant algorithm for the out-tree class of instances and show this algorithm is constant competitive.

This paper has helped to develop the algorithmic understanding run-time scheduling to optimize maximum flow. However, there are still many natural open questions worthy of being addressed:

- Is FIFO $O(\log m)$-competitive on general instances? We believe that obtaining such a result would likely require significant additional insight.
- Is FIFO asymptotically optimally competitive among nonclairvoyant algorithms? A nonclairvoyant run-time scheduler only learns of a subjob j in a DAG G_i when all of the predecessors of j are completed. If so, this would salvage the intuition of the authors of [4], and our intuition, about FIFO being the "right" algorithm. It does not seem that one can extend the $\Omega(\log m)$ lower bound for FIFO in a straight-forward manner to a lower bound for a general nonclairvoyant algorithm.
- Is there an $O(1)$-competitive nonclairvoyant algorithm for out-trees? It is not clear if or how a nonclairvoyant algorithm can "shape" an out-tree as our clairvoyant algorithm did. Its also not clear whether the design of the intra-job scheduling policy, or the design of the inter-job scheduling policy, or both, is the bottleneck to obtaining such a result.
- Is there an $O(1)$-competitive clairvoyant algorithm for series-parallel DAGs? We conjecture that designing an intra-job scheduling policy that in some way "shapes" the job appropriately would be the key step in obtaining such an algorithm.
- Is there an $O(1)$-competitive nonclairvoyant algorithm for series-parallel DAGs?
- Is there an $O(1)$-competitive clairvoyant algorithm for general DAGS?
- Is there an $O(1)$-competitive nonclairvoyant algorithm for general DAGS?

ACKNOWLEDGMENTS

Kunal Agrawal was supported in part by NSF grants CCF-2106699, CCF-2107280, PPoSS-2216971. Benjamin Moseley and Heather Newman were supported in part by a Google Research Award, Inform Research Award, Carnegie Bosch Junior Faculty Chair, NSF grants CCF-2121744, CCF-1845146, and ONR Award N000142212702. Kirk Pruhs was supported in part by NSF grants CCF-1907673, CCF-2036077, CCF-2209654, and an IBM Faculty Award

REFERENCES

[1] Kunal Agrawal, I-Ting Angelina Lee, Jing Li, Kefu Lu, and Benjamin Moseley. 2019. Practically Efficient Scheduler for Minimizing Average Flow Time of Parallel Jobs. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*. IEEE, 134–144. https://doi.org/10.1109/IPDPS.2019.00024

[2] Kunal Agrawal, Charles E Leiserson, Yuxiong He, and Wen Jing Hsu. 2008. Adaptive work-stealing with parallelism feedback. *ACM Trans. Computer Syst.* 26, 3 (2008), 7.

[3] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2016. Scheduling Parallel DAG Jobs Online to Minimize Average Flow Time. In *SODA '16*. 176–189.
[4] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2016. Scheduling Parallelizable Jobs Online to Minimize the Maximum Flow Time. In *ACM Symposium on Parallelism in Algorithms and Architectures*, Christian Scheideler and Seth Gilbert (Eds.). ACM, 195–205.
[5] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2018. Scheduling Parallelizable Jobs Online to Maximize Throughput. In *LATIN 2018: Theoretical Informatics - 13th Latin American Symposium, Buenos Aires, Argentina, April 16-19, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10807)*, Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro (Eds.). Springer, 755–776. https://doi.org/10.1007/978-3-319-77404-6_55
[6] Christoph Ambühl and Monaldo Mastrolilli. 2005. On-line scheduling to minimize max flow time: an optimal preemptive algorithm. *Oper. Res. Lett.* 33, 6 (2005), 597–602.
[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. Association for Computing Machinery, New York, NY, USA, 119–129. https://doi.org/10.1145/277651.277678
[8] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. 1998. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *SODA '98*. 270–279.
[9] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably Efficient Scheduling for Languages with Fine-grained Parallelism. *J. ACM* 46, 2 (March 1999), 281–321. https://doi.org/10.1145/301970.301974
[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*. 207–216.
[11] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *JACM* 46, 5 (1999), 720–748.
[12] Colin Campbell and Ade Miller. 2011. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press.
[13] G.I. Davida and D.J. Linton. 1976. A new algorithm for the schedule of tree structured tasks. In *Proc. Conf. Inform. Syst.* 543–548.
[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.
[15] Teofilo F. Gonzalez and Donald B. Johnson. 1980. A New Algorithm for Preemptive Scheduling of Trees. *J. ACM* 27, 2 (apr 1980), 287–312.
[16] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 2 (1969), 416–429.
[17] Yuxiong He, Wen-Jing Hsu, and Charles E Leiserson. 2008. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Trans. Parallel Distrib. Syst.* 19, 9 (2008), 1263–1279.
[18] T. C. Hu. 1961. Parallel Sequencing and Assembly Line Problems. *Operations Research* 9, 6 (1961), 841–848.
[19] Intel. 2013. Intel CilkPlus. https://www.cilkplus.org/.
[20] Bala Kalyanasundaram and Kirk Pruhs. 2000. Speed is as powerful as clairvoyance. *J. ACM* 47, 4 (2000), 617–643.
[21] Benjamin Moseley, Ruilong Zhang, and Shanjiawen Zhao. 2022. Online scheduling of parallelizable jobs in the directed acyclic graphs and speed-up curves models. *Theor. Comput. Sci.* 938 (2022), 24–38. https://doi.org/10.1016/J.TCS.2022.10.005
[22] R. R. Muntz and E. G. Coffman. 1970. Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. *J. ACM* 17, 2 (apr 1970), 324–338.
[23] OpenMP. 2013. OpenMP Application Program Interface v4.0. http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
[24] Kirk Pruhs, Julien Robert, and Nicolas Schabanel. 2010. Minimizing Maximum Flowtime of Jobs with Arbitrary Parallelizability. In *WAOA '10*. 237–248.
[25] James Reinders. 2010. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media.
[26] Olivier Tardieu, Haichuan Wang, and Haibo Lin. 2012. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. In *PPoPP '12*.