Scheduling Parallel DAG Jobs Online to Minimize Average Flow Time

Kunal Agrawal Jing Li Kefu Lu Benjamin Moseley*

October 14, 2015

Abstract

In this work, we study the problem of scheduling parallelizable jobs online with an objective of minimizing average flow time. Each parallel job is modeled as a DAG where each node is a sequential task and each edge represents dependence between tasks. Previous work has focused on a model of parallelizability known as the arbitrary speed-up curves setting where a scalable algorithm is known. However, the DAG model is more widely used by practitioners, since many jobs generated from parallel programming languages and libraries can be represented in this model. However, little is known for this model in the online setting with multiple jobs. The DAG model and the speed-up curve models are incomparable and algorithmic results from one do not immediately imply results for the other. Previous work has left open the question of whether an online algorithm can be $O(1)$-competitive with $O(1)$-speed for average flow time in the DAG setting. In this work, we answer this question positively by giving a scalable algorithm which is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for any $\epsilon > 0$. We further introduce the *first* greedy algorithm for scheduling parallelizable jobs — our algorithm is a generalization of the shortest jobs first algorithm. Greedy algorithms are among the most useful in practice due to their simplicity. We show that this algorithm is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive for any $\epsilon > 0$.

*Department of Computer Science and Engineering, Washington University in St. Louis, 1 Brookings Drive, St. Louis, MO 63130. {kunal, li.jing, kefulu, bmoseley}@wustl.edu. B. Moseley and K. Lu work was supported in part by a Google Research Award and a Yahoo Research Award. K. Agrawal and J. Li were supported in part by NSF grants CCF-1150036 and CCF-1340571.

1 Introduction

Recently, most hardware vendors have moved to manufacturing multicore machines and there is increasing interest in enabling parallelism. Many languages and libraries, such as Cilk, Cilk Plus [26], Intel's Threading Building Blocks [32], OpenMP [31], X10 [35], 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 [8, 7, 6].

In most of this research, the parallel job is modeled as a directed acyclic graph (DAG) where each node of the DAG is a sequential sequence of instructions and each edge is a dependence between nodes. A node is *ready* to be executed when all its predecessors have been executed. For the case of a single job, schedulers such as a list scheduler [20] and a work-stealing scheduler [8] are known to be asymptotically optimal with respect to the makespan of the job.

In this paper, we are interested in multiprogrammed environments where multiple DAG jobs (say n jobs) share a single parallel machine with m processors, jobs arrive and leave online, and the scheduling objective is to provide a quality of service guarantee. Surprisingly, there is little work in this domain (see [34, 29, 1, 2, 23, 30] for exceptions). On the other hand, this problem has been extensively studied for *sequential* (non-parallizable) jobs and several quality of service metrics have been considered. The *flow time* of a job i is the amount of time job i waits after it arrives until it is completed under some schedule. The most widely considered objectives are minimizing the average flow time (or equivalently, the total flow time), the maximum flow time and more generally, the ℓ_k-norms of flow time. In this work, we focus on the average flow time objective, which optimizes the average quality of service; this is the most popular objective considered in online scheduling theory.

As stated above, this problem has been widely considered for sequential jobs where each job can be scheduled on only one processor at a time. In this case, when all m processors are identical it is known that any algo-

rithm is $\Omega(\min\{\log P, \log n/m\})$-competitive where P is the ratio of the largest to smallest processing time of the jobs [28]. In the face of these strong lower bounds, previous work has considered a *resource augmentation* analysis where the algorithm is given extra speed over the adversary [27]. With resource augmentation, several algorithms are known to be $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for average flow time where $\epsilon > 0$ and some function f which depends only on ϵ [14]. Such an algorithm is known as *scalable* and is the best positive result one can show for problems that have strong lower bounds on the competitive ratio. In particular, several greedy algorithms are known to be scalable including Shortest-Remaining-Processing-Time (SRPT) and Shortest-Job-First (SJF) [36, 19, 5, 9]. Similar results are also known in more general machine environments [10, 24, 3].

Parallel jobs have also been considered in this online multiprogrammed setting; however, the parallelism model most widely considered is the *arbitrary speed-up curve model*. In the speed-up curve model, each job i is associated with a sequence of phases. Phase j for job i is denoted by a tuple $(W_{i,j}, \Gamma_{i,j}(m'))$. The value $W_{i,j}$ denotes the total work of the jth phase of job i. The work for each phase must be processed in sequential order. $\Gamma_{i,j}(m')$ is a function that specifies the processing rate $W_{i,j}$ when given $1 \leq m' \leq m$ processors. It is generally assumed that $\Gamma_{i,j}(m')$ is a nondecreasing sublinear function. The speed-up curve model was introduced by [15] and a scalable algorithm, denoted Latest-Arrival-Processor-Sharing (LAPS) is known for the model [17]. This algorithm and its analysis have been very influential in scheduling theory [11, 13, 4, 22, 16, 21, 12, 18].

While the speed-up curve model is a theoretically elegant model, most languages and libraries generate parallel programs that are more accurately modeled using DAGs. Despite this, the DAG model has only been considered for online multiprogrammed environments in a limited way: for instance, in real-time environments where jobs must finish by their deadlines[34, 29]. The work of [33] consider a hybrid of the DAG model and the speed-up curve setting where each node in the DAG has a speed-up curve. They show a $(2+\epsilon)$-speed $O(\frac{\kappa}{\epsilon})$-competitive algorithm for any $\epsilon > 0$ where κ is the maximum number of independent tasks in a job's DAG. Previous work leaves many open questions. In particular, does there exist online scalable algorithms for average flow time as in the arbitrary speed-up curve setting? Further, is there an algorithm whose competitive ratio does not depend on κ?

Challenges with the DAG model:

- Interestingly, the speed-up curve and the DAG models appear to be incomparable. In particular, for the speed-up curve model, the instantaneous parallelism (the number of processors a job can use effectively at a particular instant) depends only on the phase the job is in, which in turn depends only on how much work of the job has been completed. In contrast, for the DAG model, the instantaneous parallelism depends also on which particular nodes have been processed so far. Since there are many possible ways to do the same amount of work, the instantaneous parallelism at a particular instant depends on the previous schedule. Since the DAG is unknown in advance, it is impossible to compute the best possible schedule that leads to best possible future parallelizability. [1]
- One of the goals of this paper is to design a greedy algorithm for DAG jobs. Interestingly, this presents unique challenges. In Section 4.2.1, we show a counterintuitive result for the DAG model. We construct an example showing that a greedy scheduling algorithm may actually fall behind in the total aggregate amount of work processed when compared to the same algorithm with less resource augmentation. Note that this can never happen for sequential jobs. This occurs for DAG jobs due to the dependences — by processing jobs faster, the scheduler later may not be able to efficiently pack the tasks of different jobs on the processors as it did in the slower schedule, due to the DAG structures of jobs. The example shows that standard scheduling techniques are not directly applicable to the DAG model, as typically the faster schedule never falls behind the slower schedule.
- A widely used analysis technique for bounding the total flow time, is the *fractional* flow time technique. Fractional flow time is an alternative objective function for which competitiveness is typically easier to prove. In addition, one can usually easily convert an algorithm that is competitive for fractional flow time to one that is competitive for average flow time by speeding up the algorithm by a small factor. Unfortunately, there are several hurdles for this technique in the DAG setting. In particular, it is not immediately clear how to define the fractional objective and, further, since an algorithm may still fall behind by using extra speed in the DAG setting, it is not obvious how to convert an algorithm that is competitive for fractional flow to one that is competitive for average flow time.

Results: We consider minimizing average flow time in the DAG scheduling model. The most natural algorithm

[1] The speed-up curve model also cannot be simulated using the DAG model. In the speed-up curve model one could have a speed-up curve of the form $\Gamma(m') = \sqrt{m'}$. In this case, a job is processed at a rate of $\sqrt{m'}$ when given $1 \leq m' \leq m$ processors. In the DAG setting, a job's parallelizability is linear up to the number of nodes ready to be scheduled and thus it is unclear how to simulate this speed-up curve.

to consider for average flow time in the DAG model is LAPS, since this algorithm is known to work well in the speed-up curve model. However, LAPS is a generalization of Round Robin and [33] showed that in the hybrid model, where jobs consist of a DAG and every node has it own speed-up curve, Robin Robin like algorithms must have a competitive ratio that depends on $\log \kappa$ even if they are given any $O(1)$ speed augmentation. We are able to show that this hybrid model is strictly harder than the DAG model and that LAPS is a scalable algorithm in the DAG model.

THEOREM 1.1. *LAPS is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for minimizing the average flow time in the DAG model.*

The result of LAPS also implies the following bound for Round Robin.

COROLLARY 1.1. *Round Robin is $(2+\epsilon)$-speed $O(1)$-competitive for any fixed $\epsilon > 0$ for minimizing the average flow time in the DAG model.*

LAPS is a nonclairvoyant algorithm in the sense that it schedules jobs without knowing the processing time of jobs or nodes until they have been completed. Theoretically, LAPS is a natural algorithm to consider. On the other hand, LAPS is a challenging algorithm to implement. In particular, LAPS requires a set of jobs to receive equal processing time, which is hard to achieve in practice with low overheads. More importantly, LAPS has another disadvantage that it is parameterized. The algorithm effectively splits the processors evenly amongst the ϵ fraction of the latest arriving jobs. This ϵ is the same constant used in the resource augmentation. In practice, it is unclear how to set ϵ. Theoretically, this type of algorithm is known as *existentially* scalable. That is, for each possible speed $(1+\epsilon)$ there exists a constant to input to the algorithm which makes it $O(1)$-competitive for any fixed $\epsilon > 0$. Note that in the speed-up curve model it is an intriguing open question whether an algorithm exists which is *universally* scalable. That is, the algorithm is $O(1)$-competitive given any speed $(1+\epsilon)$ without knowledge of ϵ.

In practice, the most widely used algorithms are simple greedy algorithms. They are easy to implement and features can be added to them to ensure low overhead from preemptions. Unfortunately, it is not clear how to adapt known greedy algorithms to the parallel scheduling environments. None are known to perform well for the speed-up curve settings. In this work, we consider a natural adaptation of Shortest-Job-First (SJF) to the DAG model and show the following theorem.

THEOREM 1.2. *SJF is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive for average flow time in the DAG model for any $\epsilon > 0$.*

To prove the theorem, we extend the definition of fractional flow time to the DAG model. As mentioned, it is not obvious how to convert an algorithm that is competitive for fractional flow to one that is competitive for total flow time. We give an analysis of such a conversion to the DAG model, but this is perhaps the most challenging part of the analysis and it is where we lose the factor of 2 speed.

This is the first greedy algorithm shown to perform well for parallelizable jobs in the online setting. The algorithm is simple and natural and could be used in practice. Unfortunately, we were unable to show it is a scalable algorithm. However, we hope our analysis techniques can be useful to resolving whether there exists universally scalable algorithms for scheduling parallelizable jobs.

2 Preliminaries

In the problem considered, there are n jobs that arrive over time that are to be scheduled on m identical processors. Each job i has an arrival time r_i and is represented as a Directed-Acyclic-Graph (DAG). A node in the DAG is *ready* to execute, if all its predecessors have completed. We assume the scheduler knows the ready nodes for a job at a point in time, but does not know the DAG structure a priori. Any set of ready nodes can be processed at once, but each processor can only execute one node at a time. A DAG job can be represented with two important parameters. The total *work* W_i is the sum of the processing time of the nodes in job i's DAG. The *critical-path length* C_i is the length of the longest path in job i's DAG, where the length of the path is the sum of the processing time of nodes on the path. We now state two straightforward observations regarding work and critical-path length.

OBSERVATION 1. *If a job i has all of its n ready nodes being executed by a schedule with speed s on m cores, where $n \leq m$, then the remaining critical-path length of i decreases at a rate of s. In other words, at each time step where not all m processors are executing jobs, all ready nodes of all unfinished jobs are being executed; hence, the remaining critical-path length of each unfinished job reduces by s.*

OBSERVATION 2. *Any job i takes at least $\max\{\frac{W_i}{m}, C_i\}$ time to complete in any schedule with unite speed, including OPT.*

Throughout the paper we will use A to specify the algorithm being considered unless otherwise noted. We let $W_i^A(t)$ denote the remaining processing time of all the nodes in job i's DAG at time t in A's schedule. Let $C_i^A(t)$ be the remaining length of the longest path in i's DAG where each node contributes its remaining processing time in job A's schedule at time t. Let $A(t)$ denote the set of jobs which are released and unsatisfied

in A's schedule at time t. In the above, we replace A with O to denote the same quantity in some fixed optimal solution. Note that $\int_{t=0}^{\infty} |A(t)|$ is exactly the total flow time, the objective we consider. Finally, let $\overline{W}_i(t) = \min\{W_i - W_i^O(t), W_i^A(t)\}$. We overload notation and let OPT refer to both the optimal solution's schedule and its final objective.

Potential Function Analysis: Throughout this paper we will utilize the potential function framework, also known as amortized analysis. See [25] for a survey on the technique. For this technique, one defines a potential function $\Phi(t)$ which depends on the state of the algorithm being considered and the optimal solution at time t. Let $G_a(t)$ denote the current cost of the algorithm at time t. This is the total waiting time of all the arrived jobs up to time t if the objective is total flow time. Similarly let $G_o(t)$ denote the current cost of the optimal solution up to time t. We note that $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}}$ is the change in the algorithm's objective at time t and this is equal to the number of unsatisfied jobs in the algorithm's schedule at time t, i.e. $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} = |A(t)|$. To bound the competitiveness of an algorithm, one shows the following conditions about the potential function.

Boundary condition: Φ is zero before any job is released and Φ is non-negative after all jobs are finished.

Completion condition: Summing over all job completions by the optimal solution and the algorithm, Φ does not increase by more than $\beta \cdot$ OPT for some $\beta \geq 0$.

Arrival condition: Summing over all job arrivals, Φ does not increase by more than $\alpha \cdot$ OPT for some $\alpha \geq 0$.

Running condition: At any time t when no job arrives or is completed,

$$\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} + \frac{\mathtt{d}\Phi(t)}{\mathtt{dt}} \leq c \cdot \frac{\mathtt{d}G_o(t)}{\mathtt{dt}} \tag{2.1}$$

Integrating these conditions over time one gets that $G_a - \Phi(0) + \Phi(\infty) \leq (\alpha+\beta+c) \cdot$ OPT by the boundary, arrival and completion conditions. This shows the algorithm is $(\alpha + \beta + c)$-competitive

3 Algorithm: LAPS

In this section, we analyze the LAPS scheduling algorithm for the DAG model. LAPS is a generalization of round robin. Round robin essentially splits the processing power evenly among all jobs. In contrast, at each step, LAPS splits the processing power evenly among the ϵ fraction of the jobs which arrived the latest. Note that LAPS is parametrized by the constant ϵ, the same constant used for the resource augmentation.

Specifically, let $A(t)$ denote the set of unsatisfied jobs in LAPS's queue at time t. Let $0 < \epsilon < \frac{1}{10}$ be some fixed constant. Let $A'(t)$ contain the $\epsilon|A(t)|$ jobs from $A(t)$ which arrived the latest. Each job in $A'(t)$ receives $\frac{m}{|A'(t)|}$ processors. Each DAG job in $A'(t)$ then assigns an arbitrary set of $\frac{m}{|A'(t)|}$ ready tasks on the processors it receives. If the job does not have $\frac{m}{|A'(t)|}$ ready tasks, it schedules as many tasks as possible and idles the remaining alloted processors.

We assume that the LAPS is given $1 + 10\epsilon$ resource augmentation. As mentioned in Section 2, $W_i^A(t)$ and $C_i^A(t)$ denote the aggregate remaining work and critical-path length, respectively, of job i at time t in the LAPS's schedule. $W_i^O(t)$ is the aggregate remaining work of job i in the optimal schedule at time t. Now we compare LAPS to the optimal schedule. To do this, we define a variable $Z_i(t) := \max\{W^A(t) - W^O(t), 0\}$ for each job i. The variable $Z_i(t)$ is the total amount of work job i has fallen behind in the LAPS's schedule as compared to the optimal schedule at time t. Finally, we define $\mathtt{rank}_i(t) = \sum_{j \in A(t), r_j \leq r_i} 1$ of job i to be the number of jobs in $A(t)$ that arrived before job i, including itself. Without loss of generality, we assume each job arrives at a distinct time.

Now we are ready to define our potential function.

$$\Phi(t) = \frac{10}{\epsilon} \sum_{i \in A(t)} \left(\frac{1}{m} \mathtt{rank}_i(t) Z_i(t) + \frac{100}{\epsilon^2} C_i^A(t) \right)$$

The following proposition follows directly from the definition of the potential function.

PROPOSITION 3.1. $\Phi(0) = \Phi(\infty) = 0$.

We begin by showing the increase in the potential function is bounded by OPT over the arrival and completion of all jobs.

LEMMA 3.1. *The potential function never increases due to job completion by the LAPS or optimal schedule.*

Proof. When the optimal schedule completes a job, it has no effect on the potential. When the LAPS completes a job i at time t, a term is removed from the summation. Notice that $Z_i(t) = 0$ and $C_i^A(t) = 0$, since the algorithm has completely processed the job. Thus the removal of this term has no effect on the potential. The only other change is that $\mathtt{rank}_j(t)$ decreases by 1 for all jobs $j \in A(t)$ where $r_j > r_i$. However, $Z_j(t)$ is always positive by definition, so this can only decrease the potential.

LEMMA 3.2. *The potential function increases by at most* $O(\frac{1}{\epsilon^3})$OPT *over the arrival of the jobs.*

Proof. When job i arrives at time t, it does not effect the rank of any other job since its arrival is after them. Further, by definition $Z_i(t)$ is 0 when job i arrives, since both LAPS and OPT cannot have worked on job i yet at the time it arrives. Finally, the value of $C_i^A(t) = C_i$. The increase in the potential will be $\frac{1000}{\epsilon^3}C_i$. By summing over the arrival of all jobs, the total increase is $\frac{1000}{\epsilon^3}\sum_{i\in[n]} C_i$. We know that each job i must wait at least C_i time units to be satisfied in OPT by Observation 2, so this is at most $O(\frac{1}{\epsilon^3})$OPT.

The remaining lemmas bound the change in the potential due to the processing of jobs by OPT and LAPS. We first consider the change in the potential due to the OPT and LAPS separately. Then we combine both changes and bound the aggregate change to be at most $-10|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$.

LEMMA 3.3. *At any time t, the potential function increases by at most $\frac{10}{\epsilon}|A(t)|$ due to the processing of jobs by OPT.*

Proof. Notice that the variables $C_i^A(t)$ do not change due to OPT. The only change occurs due to the optimal schedule decreasing $Z_i(t)$ for some jobs i. Let job i' be the job in $A(t)$ which arrived the latest. In the worst case, the optimal schedule uses all m processors to process job i' to decrease $Z_i(t)$ at a rate of m. This is the worst case because the rank of job i' is the largest. The total increase in the change of the potential is then $\frac{10}{\epsilon}\frac{1}{m}\texttt{rank}_{i'}(t)m$. Knowing that $\texttt{rank}_{i'}(t) = |A(t)|$, hence $\frac{10}{\epsilon}\frac{1}{m}\texttt{rank}_{i'}(t)m = \frac{10}{\epsilon}|A(t)|$.

LEMMA 3.4. *At any time t, the potential function increases by at most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$ due the processing of jobs by LAPS.*

Proof. Consider the set $A'(t)$ of jobs LAPS processes at time t. We break the analysis into two cases. In either case we show that the total change in the potential is a most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$.

Case 1: At least $\frac{\epsilon}{10}|A'(t)|$ jobs in $A'(t)$ have less than $\frac{m}{|A'(t)|}$ ready nodes at time t. Let $A_c(t)$ be this set of jobs.

Since each of these jobs has less than $\frac{m}{|A'(t)|}$ ready tasks at time t, then LAPS schedules all available tasks for these jobs. Hence, LAPS decreases $C_i^A(t)$ at a rate of $1 + 10\epsilon$ for each job $i \in A_c(t)$ since LAPS has $1 + 10\epsilon$ resource augmentation. We denote the change in the potential as C_1. Therefore,

$$C_1 = -\frac{1000}{\epsilon^3}(1+10\epsilon)|A_c(t)|$$

Note that $|A_c(t)| \geq \frac{\epsilon}{10}|A'(t)|$ and , we have

$$C_1 \leq -\frac{100}{\epsilon^2}(1+10\epsilon)|A'(t)| \leq -\frac{100}{\epsilon}(1+10\epsilon)|A(t)|$$

Finally, because $|A'(t)| = \epsilon|A(t)|$, we get

$$C_1 \leq -\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$$

Case 2: At least $(1-\frac{\epsilon}{10})|A'(t)|$ jobs in $A'(t)$ have at least $\frac{m}{|A'(t)|}$ nodes ready at time t. Let $A_w(t)$ be this set of jobs, so $|A_w(t)| \geq (1-\frac{\epsilon}{10})|A'(t)|$.

In this case, we ignore the decrease in the C variables and focus on the decrease in the Z variables due to the algorithms processing. We further ignore the decrease in the $Z_i(t)$ for jobs in $A_w(t) \cap O(t)$.

Notice that for every job i in $A_w(t)\backslash O(t)$ it is the case that $Z_i(t)$ decreases at a rate of $(1+10\epsilon)\frac{m}{|A'(t)|}$. This is because: (1) each of these jobs is given $\frac{m}{|A'(t)|}$ processors; (2) LAPS has $(1+10\epsilon)$ resource augmentation; (3) OPT completed job i by time t, if job i is in $A_w(t) \setminus O(t)$. Knowing this, we can bound the total change in the potential due to LAPS.

We will replace $1 + 10\epsilon$ with k in some intermediate steps for ease of notation and we denote the total change in the potential due to LAPS as C_2.

$$\begin{aligned} C_2 &= -\frac{10}{\epsilon}\sum_{i\in A_w(t)\setminus O(t)} \frac{1}{m}\texttt{rank}_i(t)\frac{(1+10\epsilon)m}{|A'(t)|} \\ &= -\frac{10k}{\epsilon}\sum_{i\in A_w(t)\setminus O(t)} \texttt{rank}_i(t)\frac{1}{|A'(t)|} \\ &\leq -\frac{10k}{\epsilon}\sum_{i\in A_w(t)\setminus O(t)} (1-\epsilon)|A(t)|\frac{1}{|A'(t)|} \end{aligned}$$

Note that $\texttt{rank}_i(t) \geq (1-\epsilon)|A(t)|$ for $i \in A'(t)$ and $|A'(t)| = \epsilon|A(t)|$, we have

$$\begin{aligned} C_2 &\leq -\frac{10k}{\epsilon^2}\sum_{i\in A_w(t)\setminus O(t)} (1-\epsilon) \\ &\leq -\frac{10k}{\epsilon^2}\left(\sum_{i\in A_w(t)} (1-\epsilon) - \sum_{i\in O(t)} 1\right) \end{aligned}$$

We can also derive $|A_w(t)| \geq (1-\frac{\epsilon}{10})|A'(t)|$. Again by replacing $|A'(t)|$ with $\epsilon|A(t)|$, we get

$$\begin{aligned} C_2 &\leq -\frac{10k}{\epsilon^2}\left((1-\frac{\epsilon}{10})\sum_{i\in A'(t)} (1-\epsilon) - \sum_{i\in O(t)} 1\right) \\ &\leq -\frac{10k}{\epsilon}\left((1-\frac{\epsilon}{10})\sum_{i\in A(t)} (1-\epsilon) - \frac{1}{\epsilon}\sum_{i\in O(t)} 1\right) \end{aligned}$$

Finally, because $\epsilon < 1/10$, we can derive

$$\begin{aligned} C_2 &\leq -\frac{10}{\epsilon}(1+10\epsilon)(1-\frac{\epsilon}{10})\sum_{i\in A(t)}(1-\epsilon) \\ &\quad + O(\frac{1}{\epsilon^2})|O(t)| \\ &\leq -\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)| \end{aligned}$$

Thus, in either case the total change in the potential is at most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$.

LEMMA 3.5. *Fix any time t. The total change in the potential is at most $-10|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$ due the processing of jobs by both algorithms.*

Proof. Now we know from Lemma 3.3 the change due to OPT processing jobs is at most $\frac{10}{\epsilon}|A(t)|$. Combining the change due to both algorithms in Lemma 3.3 and 3.4, we see that the aggregate change in the potential is at most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)| + \frac{10}{\epsilon}|A(t)| \leq -10|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$.

Thus, by the potential function framework and combining Lemma 3.1, 3.2 and 3.5 and Proposition 3.1 we have Theorem 1.1.

4 Algorithm: SJF

In this section we analyze a generalization of SJF to parallel DAG jobs. In this algorithm, the jobs are sorted according to their *original* work and the smallest have the highest priority. The algorithm takes the highest priority job and assigns all of its ready nodes to machines and then recursively considers the next highest priority job. This continues until all machines have a node to execute or there are no more ready nodes. In the event that a job being considered has more ready nodes than machines available, the algorithm choses an arbitrary set of nodes to schedule on the remaining machines. At first glance, this might be counterintuitive, since it doesn't take the critical-path length into consideration at all; one might think that we should give higher priority to jobs with longer critical-path length. However, as the analysis shows, it turns out that prioritizing based on just work provides good bounds.

4.1 Analysis of SJF for Fractional Flow Time We use fractional flow time to do this analysis. In this section, to avoid confusion, we refer to total flow time as *integral flow time* — recall that a job contributes 1 to the objective each time unit the job is alive and unsatisfied. In contrast, in fractional flow time, it contributes the fraction of the *work* which remains for the job; that is, the goal is to minimize $\sum_{t=0}^{\infty}\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i}$. Our analysis is structured as follows: We first compare the fractional flow time of SJF (with resource augmentation) to the integral flow time of the optimal algorithm. We then compare the integral flow time of SJF (with further resource augmentation) to its fractional flow time.

We will utilize a potential function analysis and define the potential functions as follows. Throughout the analysis we will assume without loss of generality that each job arrives at a distinct time and has a unique amount of work.

$$\Phi(t) = \frac{1}{\epsilon}\sum_{j\in A(t)} C_j^A(t) \; + \; \frac{1}{\epsilon m}\sum_{j\in A(t)}\left(\frac{\overline{W}_j(t)}{W_j}\sum_{i \mid \substack{i\in A(t)\cup O(t)\\ W_i\leq W_j}} W_i^A(t) - W_i^O(t)\right)$$

Using this potential function, our goal is to show the following theorem.

THEOREM 4.1. *SJF is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive when SJF's fractional flow time is compared against the optimal schedule's integral flow time.*

Note that $\Phi(0) = \Phi(\infty) = 0$, thus the boundary condition is true. We will now show the arrival and completion conditions.

LEMMA 4.1. *The potential function increases by at most $O(\frac{1}{\epsilon}\text{OPT})$ due to the arrival and completion of jobs.*

Proof. First consider the arrival condition. Suppose job j' arrives at a time t', then in the first term a new term is created, $\frac{1}{\epsilon}C_{j'}$. This is less than $\frac{1}{\epsilon}$ multiplied by the amount of time this job must wait to be completed in an optimal schedule because C_i is a lower bound on a job's integral flow time, according to Observation 2. The change of $\Phi(t')$ over all job arrivals in the first term is at most $\frac{1}{\epsilon}$OPT. Now consider the second term of $\Phi(t')$ when j' just arrives. The quantity $\overline{W}_{j'}(t') = 0$, because OPT has not worked on job j' yet. Though j' is a new term in the outer summation of the second term, this term is 0. Finally, j' may appear as a new term in the inner summation for all jobs $i \in A(t')$ with $W_i > W_{j'}$. However then $W_{j'}^A(t') - W_{j'}^O(t') = 0$ because both algorithm and optimal schedule have yet to work on j'. These are all the possible changes due to the arrival of job j', therefore the arrival condition holds.

Now consider when the optimal schedule completes some job j' at time t'. The only effect on the potential, is that a term may be removed from the inner summation of the second term if j' is no longer in $A(t') \cup O(t')$. This only happens if the job is also not in $A(t')$. If the job is not in $A(t')$ then $W_{j'}^A(t') - W_{j'}^O(t') = 0$ and there is no change to the potential due to the removal of the term.

Now consider when the algorithm completes some job j' at time t'. Because the job has completed, so $C_{j'}^A(t') = 0$ and $\overline{W}_{j'}(t') = 0$. Thus, removing terms from

the either the first summation or the outer summation of the second term has no effect on the potential. However we may remove a job from the inner summation of the second term. Again, this only occurs if $j' \notin O(t')$, which means that inner summation is 0. Therefore this does not cause a change in the potential. Overall, there is no change in the potential due to jobs being completed by either the algorithm or the optimal schedule.

Thus, we have shown the boundary conditions as well as the bounded the non-continuous changes in Φ. It remains to show how the potential changes due to the algorithm and optimal schedule processing jobs. These are the only remaining ways the potential may change. Fix some time t. Our goal is to bound $\frac{d\Phi(t)}{dt}$.

LEMMA 4.2. *The total change in Φ at time t due to the optimal schedule processing jobs is $O(|O(t)|) + \frac{1}{\epsilon}\sum_{i\in A(t)} \frac{W_i^A(t)}{W_i}$.*

Proof. Notice that the only changes that can occur due to the optimal schedule processing some job j is due to the changes in $W_j^O(t)$ and $\overline{W}_j(t)$, both of which are in the second term of $\Phi(t)$. Fix some job j that OPT processes at time t and suppose that OPT uses m'_j processors to process job j. Consider the change in $\Phi(t)$ due to $W_j^O(t)$ decreasing. The change only increases $W_j^A(t) - W_j^O(t)$ in the inner summation only if job i in the outer summation has $W_i \geq W_j$. Each machine in OPT has 1 speed and all work values are distinct, so the change is the following.

$$\frac{1}{\epsilon}\frac{m'_j}{m}\frac{\overline{W}_j(t)}{W_j} + \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i > W_j}} \frac{\overline{W}_i(t)}{W_i}$$

The first term is the job j itself and the second is the other jobs effected. Since $\frac{\overline{W}_i(t)}{W_i} \leq \frac{W_i^A(t)}{W_i}$ by definition of $\overline{W}_i(t)$, we have

$$\frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i \geq W_j}} \frac{\overline{W}_i(t)}{W_i} \leq \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i \geq W_j}} \frac{W_i^A(t)}{W_i}$$

Now consider the change induced in $\overline{W}_j(t)$ by OPT's processing. This variable could, in the worst case, increase at a rate of m'_j. This changes all of the inner summation terms where $W_i \leq W_j$. We omit the $-W_i^O(t)$ part of the inner summation, as this part only decreases the potential. The change is then the following.

$$\frac{1}{\epsilon}\frac{m'_j}{m}\frac{W_j^A(t)}{W_j} + \frac{1}{\epsilon}\frac{m'_j}{mW_j}\sum_{\substack{i\in A(t)\\ W_i < W_j}} W_i^A(t)$$

By definition, $\frac{W_j^A(t)}{W_j} \leq 1$. Additionally, in the summation we have $W_i < W_j$. Therefore the overall change from processing job j is:

$$\frac{1}{\epsilon}\frac{m'_j}{m} + \frac{1}{\epsilon}\frac{m'_j}{mW_j}\sum_{\substack{i\in A(t)\\ W_i < W_j}} W_i^A(t)$$
$$\leq \frac{1}{\epsilon}\frac{m'_j}{m} + \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i < W_j}} \frac{W_i^A(t)}{W_i}$$

Let $P^O(t)$ be the set of jobs the optimal schedule processes at time t. Clearly, the optimal schedule can use at most m processors at time t, i.e. $\sum_{j\in P^O(t)} m'_j \leq m$. Knowing this, we have the overall change is

$$\sum_{j\in P^O(t)}\left(\frac{1}{\epsilon}\frac{m'_j}{m} + \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i}\right)$$
$$\leq \left(\frac{1}{\epsilon} + \frac{1}{\epsilon}\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i}\right)$$

Finally we know that OPT must have at least one alive job if it processes some job. Thus we charge the $\frac{2}{\epsilon}$ to OPT's increase in its objective. This gives the lemma.

Now we consider the change in $\Phi(t)$ due to the algorithm processing jobs.

LEMMA 4.3. *The total change in Φ at time t due to the algorithm processing jobs is $O(|O(t)|) - (1+\epsilon)\frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)} \frac{W_i^A(t)}{W_i}$.*

Proof. For any job j, we know that either the algorithm is processing jobs $i \in A(t)$ where $W_i \leq W_j$ using all m processors or the algorithm is decreasing the critical-path, $C_j^A(t)$, at a rate of $(1+\epsilon)$. This is because, the algorithm by definition has either has assigned all processors to higher priority jobs or it is scheduling all available ready nodes for job j by Observation 1. Suppose that the algorithm decreases the critical-path of j. If this is the case then, this decreases $C_j^A(t)$ at a rate of $-(1+\epsilon)$. Alternatively, say the algorithm assigned all processors to jobs with higher priority than j. Then it is the case that $\sum_{i \mid i\in A(t)\cup O(t), W_i \leq W_j} W_i^A(t) - W_i^O(t)$ decreases at a rate of $-(1+\epsilon)m$ due to the algorithms processing.

Consider all jobs i in the potential. The decreases in the potential function due to the change in $W_i^A(t)$ and $C_i^A(t)$ over all jobs i the algorithm processes is at least the following,

$$-\frac{(1+\epsilon)}{\epsilon}\sum_{i\in A(t)}\frac{\overline{W}_i(t)}{W_i}$$

Consider the jobs in this summation, if $i \notin O(t)$ it is the case that $\overline{W}_i(t) = W_i^A(t)$. If $i \in O(t)$ then in the worst case $\overline{W}_i(t) = 0$. Nevertheless dropping all $i \in O(t)$ the decrease in the potential still

$$-\frac{(1+\epsilon)}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i}$$

The only other change that can occur is that the algorithm can cause $\overline{W}_j(t)$ to decrease for jobs j that the algorithm processes. When multiplied by $-W_i^O(t)$ this causes an increase in the potential function. Say that the algorithm processes job j using m'_j processors at time t. Let $P^A(t)$ be the set of jobs the algorithm processes. In the worst case, $\overline{W}_j(t)$ decreases at a rate of $(1+\epsilon)m'_j$ for each job $j \in P^A(t)$. The change is at most,

$$\begin{aligned}
&\frac{(1+\epsilon)}{m\epsilon} \sum_{j \in P^A(t)} \frac{m'_j}{W_j} \sum_{\substack{i \in O(t) \\ W_i \le W_j}} W_i^O(t) \\
&\le \frac{1+\epsilon}{m\epsilon} \sum_{j \in P^A(t)} m'_j \sum_{\substack{i \in O(t) \\ W_i \le W_j}} 1 \quad [W_i^O(t) \le W_i \le W_j] \\
&\le \frac{(1+\epsilon)}{m\epsilon} \sum_{j \in P^A(t)} m'_j \sum_{i \in O(t)} 1 \\
&\le \frac{(1+\epsilon)}{\epsilon} \sum_{i \in O(t)} 1 \quad [\textstyle\sum_{j \in P^A(t)} m'_j \le m] \\
&= \frac{(1+\epsilon)}{\epsilon} |O(t)|
\end{aligned}$$

Thus, the lemma follows assuming that $0 < \epsilon \le 1$ is a constant.

Now we are ready to show SJF's guarantees for fractional flow time.

Proof of [Theorem 4.1]

The total change in the potential due to the algorithm and optimal schedule processing jobs is the following from Lemmas 4.3 and 4.2. Note that we are summing over the terms, some of which are negative due to decreasing the potential.

$$\begin{aligned}
&O(|O(t)|) + \frac{1}{\epsilon} \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i} \; + \\
&\quad - \frac{(1+\epsilon)}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} \\
&\le O(|O(t)|) + \frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} + \\
&\quad \frac{1}{\epsilon} \sum_{i \in O(t)} \frac{W_i^A(t)}{W_i} + -(1+\epsilon)\frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} \\
&\le O(|O(t)|) + \frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} + \\
&\quad \frac{1}{\epsilon} \sum_{i \in O(t)} 1 + -(1+\epsilon)\frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} \\
&\le O(|O(t)|) + \frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} + \\
&\quad - (1+\epsilon)\frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} \\
&\le O(|O(t)|) - \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i}
\end{aligned}$$

Consider the second term. We know that

$$\begin{aligned}
&- \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i} \\
&= - \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i} + \sum_{i \in A(t) \cap O(t)} \frac{W_i^A(t)}{W_i} \\
&\le - \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i} + |O(t)|
\end{aligned}$$

Thus, we have proved that the total change in the potential plus the increase in the algorithm's objective, $\sum_{i \in A(t)} \frac{W_i^A(t)}{W_i}$, is bounded by $O(\frac{1}{\epsilon}\text{OPT})$. This completes the proof of the continues change in the potential. The theorem follows by this, Lemma 4.1 and the potential function framework. □

4.2 SJF Falls Behind with Resource Augmentation

Before we show how to convert the fractional flow time of SJF to its integral flow time and how to prove the competitiveness for SJF, we first present the challenges in the proof.

In particular, we show that SJF can fall behind with more resource augmentation. This is surprising because essentially the same scheduling algorithm is used, yet with speed augmentation it is actually possible for the

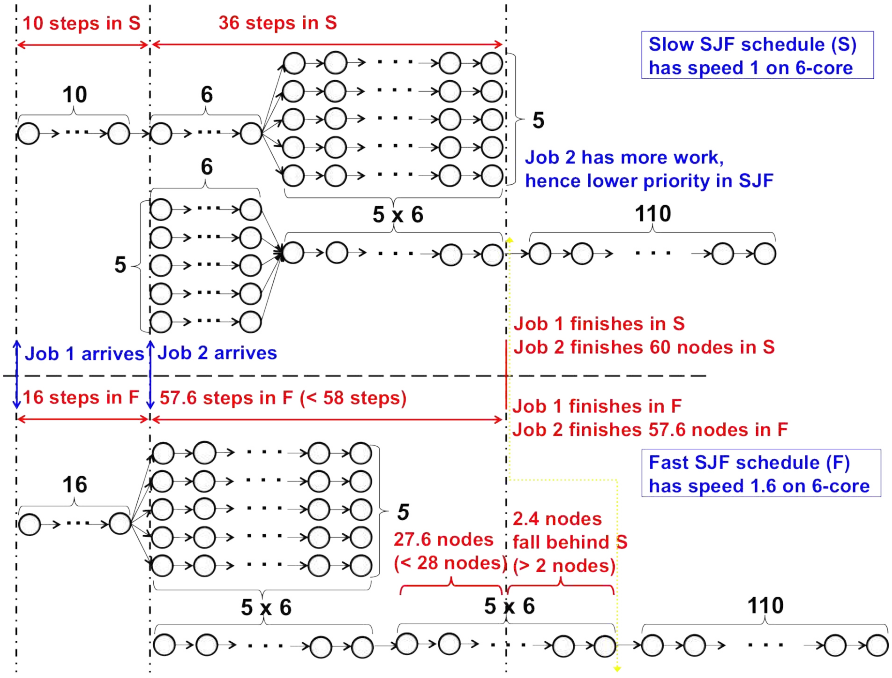

Figure 1: An example schedule of slow and fast SJF on 6 processors

fast schedule to have performed less aggregate work than the slow schedule at some time t. This difficult arises specifically due to the intricacies of the DAG model.

We consider two schedules: one slow schedule S with unit speed and one fast schedule F with speed s for some fixed constant S. We will show that for a given speed augmentation s and m processors, where $1 < s < 2-\frac{2}{m}$, we can always construct a counterexample showing that the fast schedule F falls behind the slow schedule S using two jobs J_1 and J_2.

First we shall give a concrete example where with 1.6 speed, F does less aggregate work than S does at some time t. Then, the general example for any speed $s < 2-\frac{2}{m}$ will be given. Intuitively, we show that the structure of J_1 on the fast schedule forces J_2 to be executed entirely sequentially, this severely limits the amount of work that can be done on J_2 by the fast schedule. As both schedules complete J_1, this directly shows that the fast schedule completes less aggregate work.

4.2.1 Example for Speed 1.6 on 6 processors In the concrete example, the fast schedule have 1.6 speed. Consider two jobs J_1 and J_2 as given in the figure. J_1 consists of a sequential chain of nodes of total length 16, followed by 5 chains of nodes all having total length 30 (i.e. a block of width 5 and length 30). Note the construction of the DAG means that at time 10 the fast schedule will have finished the entire chain, while the slow one will still have 6 nodes to do. J_2 arrives at the absolute time of 10 and consists of a block of width 5 with length 6, followed by a long sequential chain of nodes. In this example, the length of this chain is 140. Note that the total work of J_2 is 170, which is more than J_1's total work 166. Thus, J_2 has lower priority under both slow and fast SJF.

The time we consider to contradict the lemma is $t = 46$. By this point, both F and S have finished J_1, therefore it is sufficient to compare the amount of work done on J_2. In the slow schedule for the first 6 steps once J_2 arrives, due to the fact that J_1 can only utilize 1 processor, 30 nodes of J_2 is finished. A further 30 nodes of J_2 finishes for a total of 60 at time t.

The fast schedule is of more interest. With 1.6 speed augmentation, effectively 16 nodes can be finished in the time that the slow schedule requires to finish 10 nodes. Therefore, when J_2 arrives, the fast schedule has already finished the first chain and reached the highly parallel portion of J_1. As J_1 has higher priority than J_2, this forces J_2 to be executed on the only remaining processor sequentially. Hence, due to the length of the block in J_1, the first block (30 nodes) of J_2 is executed completely sequentially. The rest of J_2 is a chain and has to run sequentially due to the structure of the DAG. Therefore, J_2 is performed entirely sequentially.

Now we compare the amount of work of J_2 done by S and F during the time interval $[10, 46]$, which has length 36. Slow schedule with unit speed finishes 60 nodes of J_2. Taking the speed augmentation of 1.6 into account, F can sequentially execute $36 * 1.6 = 57.6$ nodes of J_2. Hence, less than 60 nodes of J_2 finishes executing by F. This means that F has fallen behind in comparison to S in terms of aggregate work at time $t = 46$.

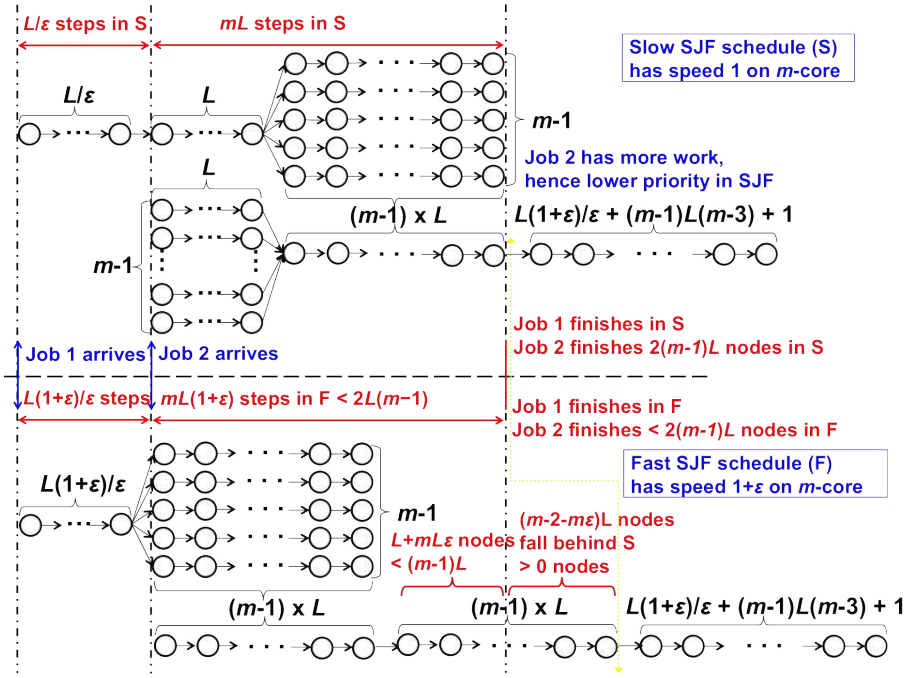

Figure 2: An example schedule of slow and fast SJF for m processors.

4.2.2 General Case for Speed s on m processors We now show the general case where a speed of $2 - \frac{2}{m}$ is necessary. We assume that the fast schedule is given some speed $s = 1 + \epsilon$ with the restriction that $0 < \epsilon < 1 - \frac{2}{m}$. Similar to the concrete example, we construct the two jobs with J_1 being a chain followed by a block and J_2 being almost the opposite but having larger work and lower priority. The key idea is that for J_1, the fast schedule must reach the highly parallel portion earlier, more precisely, at the release time of J_2. Note that for every node processed by the slow schedule in the initial chain of J_1, the fast schedule processes $1 + \epsilon$ nodes, gaining ϵ nodes over the slow schedule.

Consider Figure 2, for similarity to the previous example we introduce a constant L. In the previous example, we had $L = 6$. Let J_1 begin with a chain of length $\frac{L}{\epsilon} + L$, followed by a block of length $(m-1)L$ and parallelism (width) $(m-1)$. J_2 will consist of a block of length L with parallelism $(m-1)$ followed by a long chain of sufficient length such that J_2 has more work and lower priority than J_1. J_2 arrives at exactly time $\frac{L}{\epsilon}$.

The time that will be examined is time $t = \left(\frac{L}{\epsilon} + L\right) + (m-1)L$. Note that at this point both the schedules have finished J_1 and therefore it is sufficient to compare the amount of work done on J_2. In the slow schedule, J_2 arrives when only 1 processors is used to execute J_1, as the highly parallel block has not been reached. Therefore, for the next L time steps a total of $(m-1)L$ nodes of J_2 are finished with parallelism $m-1$. On the following $(m-1)L$ steps, J_1 occupies $m-1$ processors, while J_2 reaches its chain and is processed sequentially. A total of $2L(m-1)$ nodes of J_2 are finished at time t. We also note that a total of mL time steps have passed in the slow schedule between the arrival of J_2 and time t.

From the construction of the initial chain of J_1, the fast schedule completes all $\frac{L}{\epsilon} + L = \frac{L}{\epsilon}(1+\epsilon)$ nodes of the strand by the time $\frac{L}{\epsilon}$ that J_2 arrives. Due to the higher priority of J_1, the parallel block of J_1 take precedence over that of J_2. Note that the parallel block of J_1 has a width of $m-1$, which occupies all but one processor for as long as $(m-1)L$ steps. This forces J_2 to only execute sequentially on the remaining single processor for all its $(m-1)L$ nodes of the parallel block in J_2. When J_1 finally completes and all m processors are free, J_2 reaches its sequential chain. Therefore, J_2 is processed entirely sequentially in the fast schedule.

The amount of time which passes between the arrival of J_2 and t is just mL. Consider the speed augmentation of the fast schedule and recall that $\epsilon < 1 - \frac{2}{m}$. The total number of nodes of J_2, that the fast processor can sequential execute between the arrival of J_2 and t, is $mL(1+\epsilon) < mL(2 - \frac{2}{m}) = 2L(m-1)$. Recall that the slow schedule performed exactly $2L(m-1)$ nodes of J_2 during the same time interval. Therefore, the fast schedule with $1 + \epsilon$ speed performs less total aggregate work at time t in comparison to the slow schedule.

Note that this example does not hold when $\epsilon \geq 1$ as the final calculation would result in the fast processor finishing more nodes of J_2.

4.3 From Fractional to Integral We now compare the fractional flow time of SJF to its integral flow time and prove the following lemma. Note that this lemma, combined with Lemma 4.1 proves Theorem 1.2.

LEMMA 4.4. *If SJF is s-speed c-competitive for fractional flow time then SJF is $(2+\epsilon)s$-speed $O(\frac{c}{\epsilon^3})$-competitive for the integral flow time for any $0 < \epsilon \leq 1/2$.*

To show the Lemma 4.4, for the remaining portion of the section we will consider two schedules created by SJF. One schedule has s speed and the other $(2+\epsilon)s$ for some fixed $0 < \epsilon \leq 1/2$ and some constant s. To avoid confusion, we use F to denote the fast schedule and S to denote the slow schedule. Since both schedules are SJF, we assume that the tasks for a job are given the same priority in both algorithms — this priority can be arbitrary.

To begin the proof, we first show that F has always processed as much work as S at any time given a $(2+\epsilon)$ factor more speed. It may seem obvious that a faster schedule should do more work than the slower schedule. However, showing this is not straightforward in the DAG model. In fact, in Section 4.2, we have already showed that if the faster schedule has less than a $(2-\frac{2}{m})$ factor speed it will actually fall behind in total aggregate work compared to the slow schedule in some instances. In other words, F does not always process as much of each individual job as S at each point in time. This could cause F to later not achieve as much parallelism as S. Here we will show that F does not fall behind S given a $(2+\epsilon)$ factor more speed.

First, we give some more notations. Let $\mathcal{S}(t)$ ($\mathcal{F}(t)$) denote the queued jobs in S's (F's) schedule at time t, which have been released but not finished. Let $W_i^S(t)$ ($W_i^F(t)$) and $C_i^S(t)$ ($C_i^F(t)$) denote the remaining work and remaining critical-path length, respectively, for job i in S's (F's) schedule at time t. The following lemma states that if we only focus on jobs whose original processing time is less than some value ρ, it must be the case that F did more total work on these jobs than S. This lemma is where we require the 2 speed in the conversion from fractional to integral flow time.

LEMMA 4.5. *At all times t and for all $\rho \geq 0$, it is the case that $\sum_{i\in\mathcal{F}(t),W_i\leq\rho} W_i^F(t) \leq \sum_{i\in\mathcal{S}(t),W_i\leq\rho} W_i^S(t)$.*

Proof. For the sake of contradiction, say the lemma is not true and let t be the first time it is false for some ρ. Then at this time t, there must be some job i where $W_i^S(t) < W_i^F(t)$ and $W_i \leq \rho$.

At release time r_i the lemma still holds, i.e. $\sum_{i\in\mathcal{F}(r_i),W_i\leq\rho} W_i^F(r_i) \leq \sum_{i\in\mathcal{S}(r_i),W_i\leq\rho} W_i^S(r_i)$. Let V be the total volume of original work for jobs of size at most ρ which arrives during $[r_i,t]$. Note that S can do at most $ms(t-r_i)$ work during $[r_i,t]$ with speed s on m processors, we know that at time t

$$\sum_{i\in\mathcal{S}(t),W_i\leq\rho} W_i^S(t) \geq \sum_{i\in\mathcal{S}(r_i),W_i\leq\rho} (W_i^S(r_i) + V - ms(t-r_i))$$

Consider the time interval $[r_i,t]$. Notice that it must be the case that $t - r_i \geq (C_i - C_i^S(t))/s$, since the schedule S has decreased the critical-path of job i by $C_i - C_i^S(t)$ with a speed of s. Further, knowing that both of the schedules execute the nodes of a particular job in the same priority order for either schedule, then $C_i^S(t) \leq C_i^F(t)$. Therefore, we have

$$t - r_i \geq (C_i - C_i^S(t))/s \geq (C_i - C_i^F(t))/s \tag{4.2}$$

Now consider the amount of work done by F during $[r_i,t]$. Note that for at most a $\frac{C_i - C_i^F(t)}{s(2+\epsilon)}$ amount of time during $[r_i,t]$ the schedule F have some processors idling and not executing nodes of jobs with $W_i \leq \rho$. Otherwise, by Observation 1 F would have decreased the critical-path of job i during these non-busy time steps by strictly more than $\frac{C_i - C_i^F(t)}{s(2+\epsilon)} \cdot s(2+\epsilon) = C_i - C_i^F(t)$. Then the remaining critical-path of job i at time t in F would then be less than $C_i^F(t)$, contradicting the definition of $C_i^F(t)$. Thus, F processes a total volume of at least $(2+\epsilon)ms(t - r_i - \frac{C_i - C_i^F(t)}{s(2+\epsilon)})$ on jobs with original size at most ρ during $[r_i,t]$. Hence the following. (Here have omitted the repeated indices on some sums for brevity, and invoked equation 4.2 for one of the steps).

$$\begin{aligned}
&\sum_{i\in\mathcal{F}(t),W_i\leq\rho} W_i^F(t) \\
&\leq \sum W_i^F(r_i) + V - (2+\epsilon)s(t - r_i - \frac{C_i - C_i^F(t)}{s(2+\epsilon)}) \\
&\leq \sum W_i^F(r_i) + V - (2+\epsilon)s(t - r_i - \frac{t-r_i}{(2+\epsilon)}) \\
&= \sum W_i^F(r_i) + V - (1+\epsilon)s(t-r_i) \\
&\leq \sum_{i\in\mathcal{S}(r_i),W_i\leq\rho} W_i^S(r_i) + V - s(t-r_i) \\
&= \sum_{i\in\mathcal{S}(t),W_i\leq\rho} W_i^S(t)
\end{aligned}$$

This contradicts the definition of t.

Let $t_{i,\epsilon}^S$ denote the latest time t in S's schedule where $\frac{W_i^S(t)}{W_i} \geq \epsilon$. For the fractional flow time objective, job i always pays a cost of at least ϵ at each time during $[r_i, t_{i,\epsilon}^S]$ in S's schedule. Let $f_{i,\epsilon}^S = t_{i,\epsilon}^S - r_i$. It must be the case that job i's fractional flow time is greater than $\epsilon f_{i,\epsilon}^S$ in S.

For integral flow time we know that a job pays a cost of 1 each time unit it is unsatisfied. Thus, if the integral flow time of job i in F is bounded by $f^S_{i,\epsilon}$ we can charge this job's integral cost in F to the job's fractional cost in S. Also, according to Observation 2, for integral flow time the optimal schedule of speed 1 must make job i wait C_i time steps. Thus, if job i's flow time is bounded by C_i in F then we can charge job i's integral flow time in F directly to the optimal schedule. These two ideas are formalized in the following lemma.

For any schedule A, we let $\mathrm{IntCost}(A)$ denote the integral cost of A and $\mathrm{FracCost}(A)$ denote the fractional flow time of A. Finally, we let OPT^I denote the optimal schedule for integral flow time.

LEMMA 4.6. *Let $E^F(t)$ be the set of jobs $i \in \mathcal{F}(t)$ such that $t \leq r_i + \frac{10}{\epsilon^2}(\max\{f^S_{i,\epsilon}, C_i\})$. Consider the quantity $\sum_{t=0}^{\infty} |E^F(t)|$, which is the contribution to the total integral flow at time t from jobs in $E^F(t)$. It is the case that $\sum_{t=0}^{\infty} |E^F(t)| \leq O(\frac{1}{\epsilon^3})(\mathrm{FracCost}(S) + \mathrm{IntCost}(\mathrm{OPT}^I))$.*

Proof. **Case 1:** Consider a job i with $f^S_{i,\epsilon} = \max\{f^S_{i,\epsilon}, C_i\}$. In this case, job i can only be in $E^F(t)$ during $[r_i, r_i + \frac{10}{\epsilon^2} f^S_{i,\epsilon}]$. The total integral flow time that job i in F can accumulate during this interval is at most $\frac{10}{\epsilon^2} f^S_{i,\epsilon}$. By definition of $f^S_{i,\epsilon}$, job i's fractional flow in S is at least $\epsilon f^S_{i,\epsilon}$. Hence, the total integral flow time of all jobs in F where $f^S_{i,\epsilon} = \max\{f^S_{i,\epsilon}, C_i\}$ during times where they are in $E^F(t)$ is at most $O(\frac{1}{\epsilon^3})\mathrm{FracCost}(S)$.

Case 2: Consider a job i, with $C_i = \max\{f^S_{i,\epsilon}, C_i\}$. The integral flow time in OPT^I for job i is at least C_i by definition of the critical-path. Thus, we bound the integral flow time of all such jobs in F while they are in $E^F(t)$ by $O(\frac{1}{\epsilon^2})\mathrm{IntCost}(\mathrm{OPT}^I)$.

Intuitively, we think of the jobs in $E^F(t)$ as jobs which are *early* at time t. Let $L^F(t) = \mathcal{F}(t) \setminus E^F(t)$ be the set of *late* jobs at time t. The remaining portion of the proof focuses on bounding the integral flow time of jobs in F's schedule at times when they are in $L^F(t)$. We will prove that $O(\frac{1}{\epsilon})\sum_{i\in\mathcal{S}(t)} \frac{W^S_i(t)}{W_i} \geq |L^F(t)|$ at all times t. That is, the total fractional weight of jobs in S is greater than the number of late jobs in L at all times t. Thus, we can charge the integral flow time of jobs in $L^F(t)$ to the fractional flow time of S's schedule. This will complete the proof.

To prove this, we will show the following structural lemma about S and F. Let $\mathcal{S}_{=h}(t)$ ($\mathcal{F}_{=h}(t)$) denote the remaining jobs i in S's (F's) schedule at time t whose original work satisfies $2^{h-1} \leq W_i < 2^h$ for some integer $h \geq 1$. Let $W^S_{=h}(t) = \sum_{i\in\mathcal{S}(t), 2^{h-1}\leq W_i<2^h} W^S_i(t)$ ($W^F_{=h}(t) = \sum_{i\in\mathcal{F}(t), 2^{h-1}\leq W_i<2^h} W^F_i(t)$) denote the remaining work in S's (F's) schedule at time t for jobs i whose original work satisfies $2^{h-1} \leq W_i < 2^h$ for some $h \geq 1$. We will say job i is in *class* h, if $2^{h-1} \leq W_i < 2^h$.

LEMMA 4.7. *At all times t and for all $h \geq 1$, $|\mathcal{F}_{=h}(t) \cap L^F(t)| \leq \frac{10}{\epsilon}\frac{1}{2^h}\sum_{h'=1}^{h} W^S_{=h'}(t)$.*

Before we prove this lemma, we show how it can be used to bound the number of jobs in $L^F(t)$ in terms of the fractional weight of jobs in $\mathcal{S}(t)$.

LEMMA 4.8. *At all times t,*

$$O(\frac{1}{\epsilon})\sum_{i\in\mathcal{S}(t)} \frac{W^S_i(t)}{W_i} \geq |L^F(t)|$$

Proof. Notice that $|L^F(t)| = \sum_{h=1}^{\infty} |\mathcal{F}_{=h}(t) \cap L^F(t)|$. Using Lemma 4.7 we have the following.

$$\begin{aligned}
|L^F(t)| &= \sum_{h=1}^{\infty} |\mathcal{F}_{=h}(t) \cap L^F(t)| \\
&\leq \sum_{h=1}^{\infty} \frac{10}{\epsilon} \sum_{h'=1}^{h} \frac{1}{2^h} W^S_{=h'}(t) \quad \text{[By Lemma 4.7]} \\
&= \sum_{h=1}^{\infty} \frac{10}{\epsilon} \sum_{h'=1}^{h} (\frac{1}{2^{h'}} W^S_{=h'}(t)) \frac{1}{2^{h-h'}} \\
&= \frac{10}{\epsilon} \sum_{h'=1}^{\infty} (\frac{1}{2^{h'}} W^S_{=h'}(t)) \sum_{h=h'}^{\infty} \frac{1}{2^{h-h'}} \\
&\leq \frac{20}{\epsilon} \sum_{h'=1}^{\infty} \frac{1}{2^{h'}} W^S_{=h'}(t) \\
&\leq \frac{20}{\epsilon} \sum_{i\in\mathcal{S}(t)} \frac{W^S_i(t)}{W_i} \quad [2^{h'-1} \leq W_i < 2^{h'} \text{ if } i \text{ in class } h']
\end{aligned}$$

The previous lemma with Lemma 4.6 implies Lemma 4.4. All that remains is to prove Lemma 4.7.

Proof of [Lemma 4.7]

Assume for the sake of contradiction the lemma is not true. Let t be the earliest time the lemma is false for some class h, i.e. $|\mathcal{F}_{=h}(t) \cap L^F(t)| > \frac{10}{\epsilon}\sum_{i\in\mathcal{S}(t), W_i\leq 2^h} \frac{1}{2^h} W^S_i(t)$.

Let j^* denote the job in $L^F(t)$ which arrived the earliest and j^* is of some class $h' \leq h$. By definition of $L^F(t)$, this implies that S processed at least $(1-\epsilon)W_i$ for each job $i \in L^F(t)$ where $W_i \leq 2^h$ by time t. Since S has m processors of speed s, this means $t - r_{j^*} \geq \frac{1}{sm}\sum_{i\in L^F(t), W_i\leq 2^h}(1-\epsilon)W_i$.

Consider the interval $[r_{j^*}, t]$. We first make several observations about the length of this time interval. We know that $t - r_{j^*} \geq \frac{10}{\epsilon^2} C_{j^*}$ since $j^* \in L^F(t)$. We further know that during $[r_{j^*}, t]$ there can be at most C_{j^*} time steps where F is not using all m processors to execute nodes for jobs which are in a class at most h.

Otherwise job J^* would have finished all its C_{j^*} critical-path length by time t using Observation 1 and thus have been completed by t, a contradiction.

Now our goal is to bound the total work S and F can process for jobs in classes h or less during $[r_{j^*}, t]$. The schedule S can process at most $sm(t - r_{j^*})$ work on jobs of class at most h during $[r_{j^*}, t]$ since it has m machines of speed s. The schedule F processes at least $(2+\epsilon)sm(t - r_{j^*} - C_{j^*})$ work on jobs of class at most h by the observations above. Knowing that $t - r_{j^*} \geq \frac{10}{\epsilon^2} C_{j^*}$, we see that $(2+\epsilon)sm(t - r_{j^*} - C_{j^*}) \geq (2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*})$.

We will use these arguments to bound the total volume of work in S at time t to draw a contradiction. Let V denote the total original processing time of jobs which are of class at most h that arrive during $[r_{j^*}, t]$. By Lemma 4.5, we have $\sum_{i \in \mathcal{F}(r_{j^*}), W_i \leq 2^h} W_i^F(r_{j^*}) \leq \sum_{i \in \mathcal{S}(r_{j^*}), W_i \leq 2^h} W_i^S(r_{j^*})$. Thus,

$$\begin{aligned}
&\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) - \sum_{i \in \mathcal{F}(t), W_i \leq 2^h} W_i^F(t) \\
&\geq \left(\sum_{\substack{i \in \mathcal{S}(r_{j^*}) \\ W_i \leq 2^h}} W_i^S(r_{j^*}) + V - sm(t - r_{j^*}) \right) - \\
&\left(\sum_{\substack{i \in \mathcal{F}(r_{j^*}) \\ W_i \leq 2^h}} W_i^F(r_{j^*}) + V - (2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*}) \right) \\
&\geq (-sm(t - r_{j^*})) - \left(-(2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*}) \right) \\
&\qquad \text{[Lemma 4.5]} \\
&\geq \frac{1+\epsilon}{2} sm(t - r_{j^*}) \quad [\epsilon \leq 1/2]
\end{aligned}$$

This implies that

$$\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) \geq \frac{1+\epsilon}{2} sm(t - r_{j^*})$$

. We also know that

$$t - r_{j^*} \geq \frac{1}{sm} \sum_{i \in L^F(t), W_i \leq 2^h} (1-\epsilon) W_i$$

. With $\epsilon \leq 1/2$ this means that

$$\begin{aligned}
\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) &\geq \frac{1+\epsilon}{2} \sum_{i \in L^F(t), W_i \leq 2^h} (1-\epsilon) W_i \\
&\geq \frac{\epsilon}{4} \sum_{i \in L^F(t), W_i \leq 2^h} W_i
\end{aligned}$$

Knowing that jobs of class h have size at most 2^h and $\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) \geq \frac{\epsilon}{4} \sum_{i \in L^F(t), W_i \leq 2^h} W_i$, we complete the proof:

$$\begin{aligned}
&|\mathcal{F}_{=h}(t) \cap L^F(t)| \\
&= \sum_{\substack{i \in L^F(t) \\ 2^{h-1} \leq W_i < 2^h}} 1 \leq 2 \sum_{\substack{i \in L^F(t) \\ 2^{h-1} \leq W_i < 2^h}} \frac{W_i}{2^h} \\
&\leq \frac{10}{\epsilon} \sum_{i \in \mathcal{S}(t), W_i \leq 2^h} \frac{1}{2^h} W_i^S(t)
\end{aligned}$$

This contradicts the definition of time t and thus we have proven the lemma. □

References

[1] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive task scheduling with parallelism feedback. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[2] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive work stealing with parallelism feedback. In *Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2007.

[3] S. Anand, Naveen Garg, and Amit Kumar. Resource augmentation for weighted flow-time explained by dual fitting. In *SODA*, pages 1228–1241, 2012.

[4] Nikhil Bansal, Ravishankar Krishnaswamy, and Viswanath Nagarajan. Better scalable algorithms for broadcast scheduling. *ACM Transactions on Algorithms*, 11(1):3:1–3:24, 2014.

[5] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. *Journal of Discrete Algorithms*, 4(3):339–352, 2006.

[6] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, March 1999.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 207–216, July 1995.

[8] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[9] Carl Bussema and Eric Torng. Greedy multiprocessor server scheduling. *Operations research letters*, 34(4):451–458, 2006.

[10] J. S. Chadha, N. Garg, A. Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow

time on unrelated machines with speed augmentation. In *STOC*, 2009.

[11] Ho-Leung Chan, Jeff Edmonds, Tak Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. In *STACS*, pages 255–264, 2009.

[12] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Theory Comput. Syst.*, 49(4):817–833, 2011.

[13] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Nonclairvoyant speed scaling for weighted flow time. In *ESA (1)*, pages 23–35, 2010.

[14] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with epsilon resource augmentation. In *STOC*, pages 363–372, 2004.

[15] Jeff Edmonds. Scheduling in the dark. *Theor. Comput. Sci.*, 235(1):109–141, 2000. Preliminary version in *STOC* 1999.

[16] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the ℓ_k-norms of flow time without conservation of work. In *ACM-SIAM Symposium on Discrete Algorithms*, 2011.

[17] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Transactions on Algorithms*, 8(3):28, 2012.

[18] Kyle Fox, Sungjin Im, and Benjamin Moseley. Energy efficient scheduling of parallelizable jobs. In *SODA*, pages 948–957, 2013.

[19] Kyle Fox and Benjamin Moseley. Online scheduling on identical machines using srpt. In *SODA*, pages 120–128, 2011.

[20] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[21] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *Symposium on Parallel Algorithms and Architectures*, pages 11–20, 2010.

[22] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *SODA*, pages 1242–1253, 2012.

[23] Yuxiong He, Wen-Jing Hsu, and Charles E. Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. In *IPDPS*, 2007.

[24] Sungjin Im and Benjamin Moseley. Online scalable algorithm for minimizing lk-norms of weighted flow time on unrelated machines. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, 2011*, pages 95–108, 2011.

[25] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. A tutorial on amortized local competitiveness in online scheduling. *ACM SIGACT News*, 42(2):83–97, 2011.

[26] Intel. CilkPlus. `http://software.intel.com/en-us/articles/intel-cilk-plus`.

[27] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000. Preliminary version in *FOCS* 1995.

[28] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *Journal of Computer and Systems Sciences*, 73(6):875–891, 2007.

[29] Jing Li, Jian-Jia Chen, Kunal Agrawal, Chenyang Lu, Christopher D. Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *ECRTS 2014*, pages 85–96, 2014.

[30] Lin Ma, R.D. Chamberlain, and K. Agrawal. Performance modeling for highly-threaded many-core GPUs. In *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors (ASAP)*, pages 84–91, June 2014.

[31] OpenMP. OpenMP Application Program Interface v3.1, July 2011. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[32] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2010.

[33] Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 491–500, 2008.

[34] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D. Gill. Parallel real-time scheduling of dags. *IEEE Trans. Parallel Distrib. Syst.*, 25(12):3242–3252, 2014.

[35] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. In *PPoPP*, 2012.

[36] Eric Torng and Jason McCullough. Srpt optimally utilizes faster machines to minimize flow time. *ACM Transactions on Algorithms*, 5(1), 2008.