2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

Practically Efficient Scheduler for Minimizing Average Flow Time of Parallel Jobs

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

I-Ting Angelina Lee
Washington University in St. Louis
angelee@wustl.edu

Jing Li
New Jersey Institute of Technology
jingli@njit.edu

Kefu Lu
Washington University in St. Louis
kefulu@wustl.edu

Benjamin Moseley
Carnegie Mellon University
moseleyb@andrew.cmu.edu

***Abstract*—Many algorithms have been proposed to efficiently schedule parallel jobs on a multicore and/or multiprocessor machine to minimize average flow time, and the complexity of the problem is well understood. In practice, the problem is far from being understood. A reason for the gap between theory and practice is that all theoretical algorithms have prohibitive overheads in actual implementation including using many preemptions.**

One of the flagship successes of scheduling theory is the work-stealing scheduler. Work-stealing is used for optimizing the flow time of a single parallel job executing on a single machine with multiple cores and has a strong performance in theory and in practice. Consequently, it is implemented in almost all parallel runtime systems.

This paper seeks to bridge theory and practice for scheduling parallel jobs that arrive online, by introducing an adaptation of the work-stealing scheduler for average flow time. The new algorithm Distributed Random Equi-Partition (DREP) has strong practical and theoretical performance. Practically, the algorithm has the following advantages: (1) it is non-clairvoyant; (2) all processors make scheduling decisions in a decentralized manner requiring minimal synchronization and communications; and (3) it requires a small and bounded number of preemptions. Theoretically, we prove that DREP is $(4+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for average flow time.

We have empirically evaluated DREP using both simulations and actual implementation by modifying the Cilk Plus work-stealing runtime system. The evaluation results show that DREP performs well compared to other scheduling strategies, including those that are theoretically good but cannot be faithfully implemented in practice.

***Index Terms*—parallel scheduling, online scheduling, work stealing, average flow time**

I. Introduction

In many application environments such as clouds, grids, and shared servers, clients send jobs to be processed on a server over time. The multicore and/or multiprocessor server schedules the jobs with the goal of both using the server resources efficiently and providing a good quality of service to the client jobs. One of the most popular quality metrics is the ***average flow time***, where the ***flow time*** of a job is the amount of time between the job's arrival at the server and its completion. Formally, if the completion (finish) time of job J_i is f_i and its release (arrival) time is r_i, then the flow time of job J_i is $f_i - r_i$. The average (total) flow time objective focuses on minimizing $\sum_i (f_i - r_i)$.

For sequential programs that can only utilize one core at a time, minimizing average flow time has been studied extensively [1–5]. However, most machines now consist of multiple cores and the parallelism of machines is expected to increase. In addition, there is a growing interest in parallelizing applications, so that individual jobs may themselves have internal parallelism and can execute on multiple cores at the same time to shorten their processing time. Programming languages, such as variants of Cilk [6–9], Intel's TBB [10], OpenMP [11], X10 [12], and Habanero [13, 14], are designed to allow the programmer to write parallel programs (jobs). Note that this work considers the parallel online scheduling problem of a single server machine, instead of an HPC cluster. Hence, a parallel job considered in this work can run in parallel on the multiple cores of a single server machine.

In this paper, we focus on minimizing average flow time in a setting where n parallel jobs (written using one of these parallel languages) arrive over time (online) and share a single machine with m processors.[1] We are particularly interested in designing a *theoretically good and practically efficient* algorithm for this problem which can be implemented in real systems. We first provide some context for our work and then explain our theoretical and experimental contributions.

Scheduling a Single Parallel Job: A parallel program can be represented as a ***directed acyclic graph*** (DAG) where each ***node*** of a DAG is a sequence of instructions that must execute sequentially and each edge is a dependence between nodes. A node is ***ready*** to be executed when all its predecessors have been executed. The scheduler decides when to execute which node on which processor. The problem of scheduling a *single* parallel job on m processors of a single machine to minimize ***makespan*** — the flow time of a single job — has been studied extensively.

The earliest studied scheduler is a ***greedy scheduler*** or ***list scheduling***, in which, on every time step, each processor arbitrarily takes a ready node and executes it. Such a scheduler

[1] In this paper, we use the words processor and core interchangeably.

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00024

is ***work conserving***, i.e., a processor is idle only if there are no ready nodes that are not currently being executed by other processors. It is known that the greedy scheduler is $2 - \frac{2}{m}$ competitive for makespan. This theoretical result, however, ignores the high scheduling overheads: the property of work conserving is expensive to maintain precisely. In particular, the scheduler often must keep a centralized queue of ready nodes and processors must access this queue to find ready nodes to work on and to put extra ready nodes that they generate. The synchronization overhead on this queue is large since it is accessed by all processors frequently.

Most practical systems use a ***work-stealing*** [15] scheduler instead, which tries to mimic the greedy property but has lower overheads. In work stealing, each processor, or a ***worker***, maintains its own ***deque*** (a double-ended queue) of ready nodes. A worker, for the most part, makes scheduling decision *locally*, pushing and popping ready nodes from the bottom of its own deque. Only when it runs out of ready nodes, it turns into a ***thief*** and steals from other *randomly chosen* workers. Work-stealing has low synchronization and communication overheads since workers only need to communicate with one another when they are stealing and even then, the contention is distributed due to the randomness in stealing.

The work-stealing scheduler is known to be asymptotically optimal with respect to makespan while also being practically efficient. It has many nice properties such as low overhead [16], bounded number of cache misses [17], low space usage, and low communication overhead [15]. Work-stealing is currently one of the most popular schedulers in practice, implemented by many parallel languages and libraries, such as the ones mentioned earlier. Most works on work stealing have focused on scheduling a single parallel job, however — while it has been extended in a limited manner for multi-programmed environments with multiple jobs for some objectives [18–21], no natural adaptation exists to handle multiple parallel jobs to minimize average flow time.

Scheduling Multiple Jobs to Minimize Average Flow Time: Average flow time is difficult to optimize even for sequential jobs. Any online algorithm is $\Omega(\min\{\log k, \log n/m\})$-competitive where k is the ratio of the largest to the smallest processing time of the jobs [22]. Due to this strong lower bound, previous work has focused on using a ***resource augmentation*** analysis to differentiate between algorithms, in which the algorithm is given faster processors over adversary [23]. This is regarded as the best positive theoretical result that can be shown for problems with strong lower bounds on their competitive ratio. The first such result for sequential jobs was shown in [1] and several other algorithms have been shown to have similar guarantees for sequential jobs [2–5].

For parallel DAG jobs, Agrawal et. al [24] proved the first theoretical results on average flow time for scheduling multiple DAG jobs online showing that latest-arrival-processor-sharing (LAPS) [25] — an algorithm that generalizes round robin — is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive in this model. This work also proposed a greedy algorithm called smallest-work-first (SWF) and showed that it is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive. Similar results have been shown in other parallel models [25–27].

Unlike the work-stealing scheduler result, these theoretical discoveries do not lead to good practical schedulers. There are several reasons why, the most important of which is ***preemption***. A preemption occurs when a processor switches between jobs it is working on without finishing the job. Every known online algorithm that has strong theoretical guarantees for minimizing the average flow time of parallel jobs requires an unacceptably large number of preemptions. For example, the LAPS algorithm requires splitting the processing power evenly among a set of jobs, so it preempts jobs every infinitesimal time step. Therefore, it has an unbounded number of preemptions. The SWF algorithm is a greedy scheduler and requires redistributing processors to jobs every time the parallelism (number of ready nodes) of any job changes. In the worst case, the number of preemptions by SWF depends on the total number of nodes in all the DAGs combined, which can number in the millions per program.

Practically, when a preemption occurs the state of a job needs to be stored and then later restored; this leads to a large overhead. In addition, once a preemption occurs for a job in the schedule, a different processor may be the one to resume it later — a process called ***migration*** — which has even higher overhead. Therefore, from a practical perspective, schedulers with a large number of preemptions have high overhead and this leads to a large gap between theory and practice.

Requirements for a Practical Scheduler: For scheduling parallel jobs online to minimize average flow time, we would like to replicate the success of work stealing and build on current theoretical discoveries to find an algorithm that has both strong theoretical guarantees and good practical performance. Such an algorithm should ideally have the following properties.

- It should provide good theoretical guarantees.
- It should be ***non-clairvoyant***, i.e., it requires no information about the properties of a job to make scheduling decision; that is, the scheduler is oblivious to the processing time, parallelism, DAG structure, etc., when making scheduling decisions. (SWF does not satisfy since it must know the processing time of the job when it arrives.)
- It should be ***decentralized***, i.e., require no or little global information or coordination between processors to make scheduling decisions.
- It should perform few preemptions or migrations.

Challenges for Designing a Practical Scheduler: To allow for low scheduling overhead, we want to design a decentralized work-stealing based scheduler. This can lead to both fewer preemptions and smaller synchronization overhead. Thus, the first question is whether we can optimize average flow in multi-programmed environments by only allowing processors to work on jobs in their own deque until their deque is empty and only make scheduling decisions on steal attempts — similar to normal work stealing.

For a related problem of minimizing ***maximum flow time***, how to design such a scheduler is known [18]. Unfortunately,

for average flow time, using a scheduler that never preempts until its deque is empty will not lead to good theoretical guarantees. Consider the following example. A large parallel job arrives first and occupies all processors. After this, a huge number of small jobs arrive. The optimal scheduler will complete the small jobs before the large job, but any greedy scheduler that does not preempt will continue to give all processors to the big job. This causes a huge number of small jobs to have a large flow time. One can extend this example to show both that preemptions are necessary and that natural adaptations of work stealing fail to yield good performance.

Therefore, the question remains: Can the gap between theory and practice be closed for scheduling multiple parallel jobs? Developing practical algorithmic techniques for this problem has the potential to influence the area similar as the work-stealing scheduler did.

Contributions: We have developed a practically efficient scheduling algorithm with strong theoretical guarantees for minimizing average flow time, called ***Distributed Random Equi-Partition (DREP)***, which operates as follows. When a new job arrives at time t, each processor decides to assign itself to the new job with probability $1/n_t$, where n_t is the number of incomplete jobs at time t. Processors assigned to a particular job work on the ready nodes of this job using a work-stealing scheduler. When a job completes, each processor assigned to that job randomly picks an unfinished job and assigns itself to this unfinished job. Preemptions only occur when jobs arrive. The DREP algorithm uses a decentralized protocol, has a small number of preemptions, and is non-clairvoyant. We will prove the following theorem about the DREP.

Theorem 1.1: When processors assigned to a particular job execute ready nodes of the job using a work-stealing scheduler, DREP is $(4+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for minimizing average flow time in expectation for parallel DAG jobs on m identical processors for all fixed $0 \leq \epsilon \leq \frac{1}{4}$

DREP improves upon the prior results for average flow time in two aspects. First, DREP uses a decentralized scheduling protocol. Second, DREP uses very few preemptions. Previous algorithms required a global coordination and a number of preemptions unbounded in terms of m and n. We show that using DREP, the number of preemptions is bounded: critically, DREP only preempts a job when a new job arrives.

Theorem 1.2: DREP requires processors to switch between unfinished jobs at most $O(mn)$ times over the entire schedule. Moreover, if jobs are sequential, the total expected number of preemptions is $O(n)$.

For sequential jobs, DREP matches the best-known results for *clairvoyant* algorithms which require complete knowledge of a job [1]. Our result is the first non-clairvoyant algorithm having guarantees on the number of preemptions and on average flow time simultaneously, even for sequential jobs. The closest result is that for Shortest-Elapsed-Time-First for sequential jobs, which is $(1+\epsilon)$-speed $O(1)$-competitive for average flow time on identical processors [23, 28].

The practical improvements of the algorithm are slightly offset by having a worse speed augmentation than what is known for LAPS in theory, but we believe that DREP is the first theoretical result which could realistically be implemented and used in systems. To verify this, we have evaluated this algorithm via both simulations and real implementation.

For simulation evaluations, we compared DREP against schedulers that are theoretically good but cannot be implemented faithfully in practice due to frequent preemptions, including shortest-remaining-processing-time (SRPT) [3], shortest-job-first (SJF) [29] and round-robin (RR) [26]. The simulation is designed to approximate a lower-bound on the average flow time, since it does not account for any scheduling or preemption overheads. Our evaluation showed that DREP approaches the performance of these (close to optimal) schedulers as the number of processors increases.

For evaluations based on actual implementation, we extended Cilk Plus [7], a production quality work-stealing runtime system originally designed to process a single parallel job. We implemented DREP as well as other schedulers that are implementable but do not provide bounds on average flow, including an approximated version of smallest-work-first (SWF) [24], which can be thought of a natural extension to SJF for parallel jobs and is clairvoyant. The empirical evaluation based on the actual implementation demonstrates that DREP has comparable performance with SWF.

Other Related Work: Most prior work on scheduling parallel jobs has considered a different model known as the ***arbitrary speed-up curves*** model [30]. In this model, each job i is processed in phases sequentially. During the jth phase for job i the job is associated with a speed-up function $\Gamma_{i,j}(m')$ specifying the rate at which the job is processed when given m' processors. Typically it is assumed that $\Gamma_{i,j}$ is a non-decreasing concave function, although some exceptions exist [31]. Great strides have been made in understanding this model and $(1+\epsilon)$-speed $O(1)$-competitive algorithms are known for average flow time [25], the ℓ_k-norms of flow time [32, 33], and flow time plus energy [34] and results are known for maximum flow time [35]. The work of [36] considers a hybrid of the DAG and the speed-up curves models. While the speed-up curve model has been extensively studied, the model is an idealized theoretical model. As argued in [18, 24], the results between the arbitrary speed-up curves model and the DAG model cannot be directly translated, and no one model subsumes the other directly. This work focuses on the DAG model because it most closely corresponds to jobs generated by parallel programs written using modern parallel languages.

II. Preliminaries

We consider the problem of scheduling n total jobs that arrive online and must be scheduled on m identical processors. Each job is in the form of a directed acyclic graph (DAG). For a given job J_i, there are two important parameters: its ***work***, W_i, which is the sum of the processing times of all the nodes in the DAG, and its ***critical-path length***, C_i, which is the length of the longest path through its DAG, where the length is the sum of the processing times of the nodes along that path. Below are two observations involving these parameters.

Observation 1: Any job J_i takes at least $\max\{\frac{W_i}{m}, C_i\}$ time to complete in any schedule with unit speed.

Observation 2: If a job J_i has all of its r ready nodes being executed by a schedule with speed s, where $r \leq m$, then the remaining critical-path length of i decreases at a rate of s.

When analyzing a scheduler A (DREP in our case), let $W_i^A(t)$ be the remaining work of job J_i in A's schedule at time t. Let $C_i^A(t)$ be the remaining critical-path length for job J_i in A's schedule at time t: the longest remaining path. Let $A(t)$ be the set of ***active*** jobs in A's schedule which have arrived but unfinished at time t. In all these notations, we replace the index A with O when referring to the same quantity in the optimal schedule. We overload notation and let OPT refer to both the final objective of the optimal schedule and the schedule itself.

Potential Function Analysis: We will utilize the potential function framework, also known as amortized local competitiveness. For this technique, one defines a potential function $\Phi(t)$, which depends on the state of the considered scheduler A and the optimal solution at time t. Let $G_a(t)$ (respectively, $G_o(t)$) denote the current cost of A at time t. If the objective is total flow time, then this is the total waiting time of all the arrived jobs up to time t. The change in A's objective at time t is denoted by $\frac{\mathrm{d}G_a(t)}{\mathrm{d}t}$; for the sum of completion times, this is equal to the number of active jobs in A's schedule at time t, i.e. $\frac{\mathrm{d}G_a(t)}{\mathrm{d}t} = |A(t)|$. To bound the competitiveness of a scheduler A, one shows the following conditions.

- **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 $\beta \cdot \mathrm{OPT}$ for some $\beta \geq 0$.
- **Arrival condition:** Summing over all job arrivals, Φ does not increase by more than $\alpha \cdot \mathrm{OPT}$ for some $\alpha \geq 0$.
- **Running condition:** At any time t when no job arrives or completes, $\frac{\mathrm{d}G_a(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} \leq c \cdot \frac{\mathrm{d}G_o(t)}{\mathrm{d}t}$

Integrating these conditions over time, one gets that $G_a - \Phi(0) + \Phi(\infty) \leq (\alpha+\beta+c) \cdot \mathrm{OPT}$, showing that A is $(\alpha+\beta+c)$-competitive.

The design of the potential functions follows that in [37]. The potential function is parameterized by time. At each instant of time, the potential is designed to approximate the future cost of the scheduler assuming no more jobs arrive. The idea is for the scheduler to decrease the potential by doing work to pay for the active jobs currently contributing to the schedule's cost. Since the potential is roughly the future cost of the scheduler, the active jobs can be charged against this decrease. One crucial detail is that each job's size in the potential is changed to the job's *lag* – how far the algorithm is behind the optimal solution on processing the job. This lag is not straightforward for parallel jobs since jobs have both critical-path length and total work. We choose the lag to be the total remaining work of the job in the algorithm's schedule minus the total remaining work of the job in the optimal schedule. The lag is used in the potential instead of the total remaining work because intuitively, on a job one should only pay for being behind the optimal solution.

III. DREP for Sequential Jobs: A New Non-Clairvoyant Algorithm

We first introduce our algorithm Distributed Random Equi-Partition (DREP) for the case where jobs are sequential. The idea of DREP is that it picks a random set of m jobs to work on and re-assigns processors to jobs only when a job arrives or completes. Specifically, when a new job arrives, if there are one or more free processors then one such processor takes the new job. If all processors are busy, each processor switches to the new job with probability $\frac{1}{|A(t)|}$ (breaking ties arbitrarily to give the job at most one processor), where $|A(t)|$ is the number of active jobs at the moment. Jobs that are not taken by any processor are stored in a queue. A job J_j may be in this queue for two reasons: (1) J_j was not assigned to a processor on arrival (no processor happened to switch to it); or (2) J_j was executing on some processor and that processor preempted J_j to switch to another job that arrived later. When a job completes, the processor assigned to the job chooses a job to work on uniformly at random from the queue of jobs.

DREP's theoretical guarantee on average flow time for sequential jobs is subsumed by the analysis for parallel jobs (Section IV). An important feature of DREP is the small number of preemptions, which only occur when jobs arrive, and the total number of preemptions is $O(n)$ in expectation, implying the second part of Theorem 1.2. This is because either there is a free processor which takes the new job (no preemption) or there are at least m active jobs, in which case the probability that a processor preempts is $\frac{1}{|A(t|} \leq \frac{1}{m}$. Therefore, on a job arrival, the expected number of preemptions is 1. We note that this is the first non-clairvoyant algorithm in the sequential setting, even on a single processor, to use $O(n)$ preemptions and be competitive for average flow time.

In the next section, we show how to adapt this algorithm when jobs are parallel. In particular, it shows how to combine the algorithm with work stealing.

IV. DREP with Work-Stealing: A Practical Parallel Scheduling Algorithm

This section presents a practical scheduler, based on combining work-stealing and DREP from the prior section, for scheduling parallel jobs to minimize average flow time. We show that the performance bound of this scheduler is $O(1)$-competitive using $O(1)$-speed augmentation.

A. Combining DREP with Work-Stealing

We first describe work-stealing and then explain the modifications needed to combine it with DREP.

Work Stealing: Work-stealing is a decentralized randomized scheduling strategy to execute a single parallel job. It does not use a centralized data structure to keep track of ready nodes. Instead, each processor p maintains a double-ended queue, or ***deque***, of ready nodes. When a processor p executes a node u, u may enable one, two, or zero ready nodes. Note that like prior works, we assume that a node has out-degree

at most two. This is because the out-degree of nodes in a parallel program is constant in practice, since the system can only spawn a constant number of nodes in constant time. In addition, any constant out-degree can be converted to two out-degree with no asymptotic change in work and span, so it is typical to assume out-degree of two in the theoretical analysis. If one ready node is enabled, p simply executes it. If two ready nodes are enabled, p pushes one to the bottom of its deque and executes the other. If zero ready nodes are enabled, then p pops the node at the bottom of its deque and executes it. If p's deque is empty, p becomes a ***thief***, randomly picks a ***victim*** processor and ***steals*** the top of the victim's deque. If the victim's deque is empty and the steal is ***unsuccessful***, the thief continues to steal at random until it finds work. At all times, every processor is either working or stealing; like most prior work, we assume that each steal attempt requires constant work.

DREP with Work Stealing: At time t, each processor is assigned to some job, and we maintain a queue of all jobs in the system. The processors assigned to the same job use work stealing to execute the job. When a new job arrives, each processor may preempt itself with probability $\frac{1}{|A(t)|}$, upon which it is de-assigned from its current job and assigned to the new job. When a job completes, each processor assigned to the job independently picks a job J uniformly at random from the job queue and is assigned to J. Since preemptions only occur when jobs arrive, there are at most $O(mn)$ preemptions — fewer in most cases, since generally not all processors will preempt themselves on job arrival.

The main modifications to the standard work stealing are (1) handling the deques to support multiple jobs instead of a single job and (2) implementing the preemption when a new job arrives. In standard work-stealing, each processor has exactly one deque permanently associated with it; the total number of deques is equal to the number of processors. This property no longer holds in this new scheduler as there are multiple jobs with preemptions. Therefore, instead of associating deques with processors, we associate deques with jobs. At time step t, let $p_i(t)$ be the number of processors working on a job J_i that has started executing but yet not finished. J_i maintains a set of $d_i(t)$ deques, where $d_i(t) \geq p_i(t)$. Each processor p working on J_i will be assigned one of these deques to work on. Once assigned a deque, a processor works as usual, pushing and popping nodes from its assigned deque. When p's deque is empty, it picks a random number between 1 and $d_i(t)$ and only steals from the $d_i(t)$ deques that are associated with J_i.

Now we describe how to handle job arrivals. Say a processor p was working on job J_i and therefore working on an assigned deque d. Suppose a new job J_j arrives and processor p is unassigned from J_i and assigned to J_j. The deque d remains associated with J_i; p will mark the deque d "muggable." A new deque d' associated with J_j will be assigned to p to work on. Therefore, at any time, each job J_i has a set of $d_i^a(t) = p_i(t)$ ***active deques***, deques currently assigned to processors working J_i, and $d_i^m(t)$ ***muggable*** deques, deques not currently assigned to any processor working on J_i. The total number of deques $d_i(t) = d_i^m(t) + p_i(t)$.

When a processor p assigned to J_i steals, it randomly steals from the deques associated with J_i. If the victim deque d is active (a processor is working on it), the steal proceeds as usual: p takes the top node of d. If the victim deque d is muggable, p performs ***mugging***, taking over the entire deque.

When a job completes, each of the processors assigned to this job chooses an available job to work on uniformly at random from the queue of jobs.

A few things to note. (1) Muggable deques are only created when jobs arrive. (2) Muggable deques are never empty, since the processor can simply deallocate its empty assigned deque instead of marking it as muggable. (3) Muggings are always successful, since the thief can take over the deque. (4) Once a thief mugs a deque, it can always do at least one unit of work since muggable deques are never empty.

B. Analysis of DREP with Work-Stealing

This section analyzes the performance of DREP with work-stealing for minimizing average flow time. The goal is to show Theorem 1.1. Throughout this section, we assume that the algorithm is given $4 + 4\epsilon$ resource augmentation for $\epsilon \leq \frac{1}{4}$.

We will define a potential function and argue that the arrival, completion and running conditions are satisfied. However, we break from the standard potential function analysis of parallel jobs (from [24]) because the work-stealing algorithm is not strictly work-conserving. Typically, the potential functions used previously use Observation 2 to ensure a job's critical path decreases whenever the job has fewer ready nodes than the number of cores it receives. However, this observation does not apply to work stealing. Therefore, our potential function will have another potential function embedded within it, adapted from prior work on work stealing.

Probability of Working on a Job: We first give a lemma on the probability that a processor is working on a specific job.

Lemma 4.1: For any job $J_j \in A(t)$ and a processor i, the probability that i is working on J_j at t is $\frac{1}{|A(t)|}$.

Proof: We prove the lemma inductively on the arrival and completion of jobs. Fix any time t and let $n' = |A(t)|$ be the number of alive jobs in the algorithm just before time t.

First consider the arrivals of jobs. Initially, when there are no jobs, the lemma statement is vacuously true. At time step t, say there are n' jobs alive, and a new job $J_{n'+1}$ arrives. The probability of any processor i switching to this job $J_{n'+1}$ is $\frac{1}{n'+1}$, since there are now $n'+1$ jobs alive. Now consider any job J_j that was alive before the new job arrived. By the inductive hypothesis processor i is working on J_j with probability $\frac{1}{n'}$ just before job $J_{n'+1}$'s arrival. A processor that was working on J_j has a probability of $(1 - \frac{1}{n'+1})$ of not switching to the newly arrived job. Therefore, the probability that the processor continues working on J_j is then $\frac{1}{n'}(1 - \frac{1}{n'+1}) = \frac{1}{n'+1}$.

As for completion, say that a job $J_{j'}$ is completed at time t. Suppose a processor i becomes free after a job finishes. In the algorithm, the processor chooses a new job to work

on at random. This precisely gives a probability of $\frac{1}{n'-1}$ to process any specific job — the desired probability. The lemma holds for any alive job and any processor i that became free. Alternatively, consider a processor i not working on the job completed. Let $i \to j$ be the event that processor i is working on job J_j just before time t and $i \nrightarrow j$ be the event it is not. This processor is working on any alive J_j with probability $\Pr[i \to j \mid i \nrightarrow j'] = \Pr[i \to j \text{ and } i \nrightarrow j']/\Pr[i \nrightarrow j']$.

Inductively, we have $\Pr[i \nrightarrow j'] = 1 - \frac{1}{n'}$ and $\Pr[i \to j \text{ and } i \nrightarrow j'] = \Pr[i \to j] = \frac{1}{n'}$. Therefore, $\Pr[i \to j \mid i \nrightarrow j'] = \frac{1}{n'-1}$. ∎

Potential Function: We now define the potential function for the algorithm. Recall that potential functions are designed to approximate the algorithm's future cost at any time t assuming no more jobs arrive. This approximation is relative to the optimal remaining cost. To define the potential, we introduce some notations. Let $Z_i(t) := \max\{W^A(t) - W^O(t), 0\}$ for each job J_i. The variable $Z_i(t)$ is the total amount of work job J_i has fallen behind in algorithm A at time t as compared to the optimal solution (the lag of i). Further, let $C_i^A(t)$ be the remaining critical path length for job J_i in the algorithm's schedule. Define $\texttt{rank}_i(t) = \sum_{j \in A(t), r_j \leq r_i} 1$ of job J_i to be the number of jobs in $A(t)$ that arrived before job J_i.

The overall potential function has an embedded potential function adapted from prior work on work stealing. To avoid confusion, we call the overall potential as the ***flow potential***. The first term $\frac{1}{m}\texttt{rank}_i(t)Z_i(t)$, which we call the ***work term***, captures the remaining cost from the total remaining work of the jobs. The second term $d_i^m(t)$, which we call the ***mug term***, is used to handle the number of muggings. The last term (described next), which we call the ***critical-path term***, captures the remaining cost due to the critical path of the current jobs.

For defining the critical-path term, we embed a different potential function, which we call the ***steal potential***, similar to the potential function used by prior analysis on work stealing [38]. Given a job J_i with critical-path length C_i executed using work stealing, we define the ***depth*** $\mathbf{d}(u)$ of node u as the length of the longest path that ends with this node in the DAG. The ***weight*** of a node is $w(u) = C_i - \mathbf{d}(u)$. The steal potential of a node is defined as follows: a ready node that is on the deque has potential $\psi(u) = 3^{2w(u)}$ and an ***assigned*** node, a node that is executing, has potential $\psi(u) = 3^{2w(u)-1}$. The total steal potential of a job J_i at time t, represented by $\psi_i(t)$, is the sum of the steal potentials of all its ready and assigned nodes at time t.

The overall flow potential of a job J_i is the following

$$\Phi_i(t) = \frac{10}{\epsilon}\left(\frac{\texttt{rank}_i(t)}{m}(Z_i(t) + d_i^m(t)) + \frac{320}{\epsilon^2}\log_3 \psi_i(t)\right)$$

The total potential of the schedule is $\Phi(t) = \sum_{i \in A(t)} \Phi_i(t)$.

Intuition behind the Analysis: We want to show a few results: (1) the potential does not increase when jobs complete; (2) the potential increase is bounded due to job arrivals; and (3) the running condition holds in expectation. Showing the arrival and completion conditions are not difficult. The challenge is in proving the running condition.

There are two cases for the running condition depending on the algorithm's status. One is when most processors are executing nodes of some job. The other is when there are many processors with no work to execute. The major challenges are in the second case. Typically, under a work-conserving scheduler, we can argue that if many processors have no work to do, then there must be few ready nodes in the system; this would allow us to use Observation 2 to argue that the critical-path length of all jobs are decreasing and thus, we are making progress towards completing the jobs. However, in a work-stealing scheduler, it is challenging to quantify that the algorithm is making progress even if many processors are idle. As in [38], the steal potential function allows us to argue the following: if a job has $d_i(t)$ deques, then $d_i(t)$ steal attempts reduce the critical-path length by a constant in expectation.

This brings us to another complication. In a normal work-stealing scheduler, $d_i(t) = p_i(t) = m$ where $p_i(t)$ is the number of processors given to job i at time t and $d_i(t)$ is the number of deques at time t. At a high-level, this means the total number of steal attempts in expectation is bounded by mC_i. But in our case, $p_i(t)$ changes over time. Worse still, $d_i(t)$ can be much larger than $p_i(t)$ when J_i has a lot of muggable deques. In particular, while steal attempts are "effective" at reducing the critical-path length when $d_i(t) \approx p_i(t)$, they are ineffective when too many steals are muggings caused by the presence of a large number of muggable deques. We must account for these steal attempts using the additional d_i^m term.

To handle these complications, the analysis uses resource augmentation $4 + 4\epsilon$. This means that each time step of OPT will be $4 + 4\epsilon$ time steps for A. We index time according to OPT's time steps. During these 4 time steps, no new jobs can arrive; jobs can only complete. In particular, say job J_i has $p_i(t)$ processors before time step t. Then during this time step t, at least $(4+4\epsilon)p_i(t)$ processor steps were spent on this job (if the job did not complete during this time step).[2] We will argue that during this step, if a job has $2p_i(t)$ steals (but not too many muggings), then the steal potential of the job reduces by a constant factor; therefore, the flow potential of the job reduces sufficiently since the flow potential's critical-path term is the log of the steal potential. If instead at least $(2+2\epsilon)p_i(t)$ of these time steps were spent on executing nodes of the job or mugging, then we will argue that the potential reduces due to the work and mug terms.

Analysis: In order to prove Theorem 1.1, we first show the completion and arrival conditions in Lemma 4.2, similar to prior work on potential functions [37]. Then we show the running condition in Proposition 4.3, which is proven using Lemmas 4.4 to 4.9.

Lemma 4.2: The completion of jobs by either A or OPT do not increase the potential. The arrival of all jobs increases the potential function by $O(\frac{1}{\epsilon^2})$OPT in expectation.

[2]A job cannot lose processors during a time step since no new jobs can arrive in the middle of a time step. A job may gain processors since work-stealing scheduler A may complete jobs during the time step, but that will only increase the number of processor steps available to the active jobs.

Proof: When A completes a job, removing the work and critical-path terms from the potential has no effect on either this job or other jobs. The rank of other jobs could decrease, but this can only decrease the potential. Completion in OPT also has no effect for the same reason. In addition, when a job completes, other jobs only gain processors; therefore, the number of muggable deques d_i^m cannot increase for any job.

When J_i arrives, $Z_i = d_i^m = 0$. Its steal potential is $\psi_i(t) = 3^{2C_i}$; therefore, the critical-path term in $\Phi_i(t)$ is $\frac{320}{\epsilon^2}\log_3 \psi_i(t) = O(1/\epsilon^2)C_i$. Over all jobs, the total change in critical-path term of Φ is bounded by $O(1/\epsilon^2)\sum_i C_i$. Since C_i is a lower bound on a job's execution time, this quantity is bounded by OPT's objective function.

When a job J_i arrives, the work term and the critical-path term of other jobs don't change because the rank of other jobs remains the same. We now consider the change in the mug term d_j^m of other jobs. When a job arrives, each other job loses $\frac{m}{|A(t)|} - \frac{m}{|A(t)|+1}$ processors in expectation and therefore creates that many more muggable deques in expectation. Therefore, the expected increase in potential from the mug term is

$$\begin{aligned}\mathbb{E}\left[\frac{\mathrm{d}\Phi(t)}{\mathrm{dt}}\right] &\le \frac{10}{\epsilon}\sum_{i\in A(t)}\left(\frac{\mathtt{rank}_i(t)}{m}\left(\frac{m}{|A(t)|} - \frac{m}{|A(t)|+1}\right)\right)\\ &\le \frac{10}{\epsilon}\frac{1}{|A(t)|(|A(t)|+1)}\sum_{i\in A(t)}(\mathtt{rank}_i(t))\\ &\le \frac{10}{\epsilon}\frac{|A(t)|^2}{|A(t)|(|A(t)|+1)} \le \frac{10}{\epsilon}\end{aligned}$$

Therefore, each job arrival changes the mug term by a constant. Since each job takes at least constant time to complete in OPT, we get the bound. ∎

Proposition 4.3: In expectation, the running condition holds at any time t. That is, at any time t it is the case that $\frac{\mathrm{d}G_a(t)}{\mathrm{dt}} + \frac{\mathrm{d}\Phi(t)}{\mathrm{dt}} \le O(\frac{1}{\epsilon^2}) \cdot \frac{\mathrm{d}G_o(t)}{\mathrm{dt}}$.

The running condition involves the instantaneous change at any moment in time. We index time by OPT's time steps, and bound this for each fixed time step t. At time t, consider the set of active jobs in DREP $A(t)$. Though $A(t)$ is a random variable dependent on the processing of DREP, we will show that the running condition holds for any $A(t)$. If we do so, by the definition of expected value, we have shown that in expectation the running condition holds. First, we bound how much the optimal can increase the potential.

Lemma 4.4: The optimal schedule's processing of jobs at t increases the potential function by at most $\frac{10}{\epsilon}|A(t)|$.

Proof: The optimal schedule's processing only changes the first term $Z_i(t)$ for any job that it processed the critical path term depends on the algorithm as well as $d_i^m(t)$. The first term for any job is a product of the rank and work remaining of the job. Therefore, the increase in potential is maximized if OPT uses all m processors to work on the job with maximum rank in $A(t)$. Therefore, the increase in potential is at most $m\frac{10}{\epsilon}\frac{1}{m}|A(t)| = \frac{10}{\epsilon}|A(t)|$. ∎

The increase in the potential due to the optimal solution needs to be offset by either charging it to the optimal cost or by showing a decrease in the potential from the algorithm's processing of jobs. First we consider the case where we can charge to the optimal solutions cost.

Claim 4.5: At time t, if $|O(t)| \ge \frac{\epsilon}{10}|A(t)|$, then the running condition is satisfied.

Proof: Note that the potential never increases due to A's processing of jobs since A can only decrease the remaining work and critical-path lengths of jobs. If $|O(t)| \ge \frac{\epsilon}{10}|A(t)|$, we will ignore the algorithm's impact on the potential and combine with Lemma 4.4 to examine the running condition.

$$\begin{aligned}\frac{\mathrm{d}G_a(t)}{\mathrm{dt}} + \frac{\mathrm{d}\Phi(t)}{\mathrm{dt}} &\le |A(t)| + \frac{10}{\epsilon}|A(t)| \le (1+\frac{10}{\epsilon})\frac{10}{\epsilon}|O(t)|\\ &\le O(\frac{1}{\epsilon^2})|O(t)| = O(\frac{1}{\epsilon^2})\frac{\mathrm{d}G_o(t)}{\mathrm{dt}}\end{aligned}$$

∎

Recall that we are using a speed augmentation of $4+4\epsilon$. Therefore, each time step has $(4+4\epsilon)$ processor steps which are spent either working or stealing, where some steal attempts become muggings if they find a muggable deque. We first argue about work and mugging steps. Fix a job J_i. If any time step starts with a lot of muggable deques for job J_i, then at least half the processor steps in that time step are spent on either working or mugging. The reason is straightforward — if a time step has a lot of muggable deques, then many of the steal attempts will become muggings. Therefore for job J_i, either a lot of work is done or there were a lot of muggings.

Lemma 4.6: If a job has $d_i(t) \ge 2p_i(t)$ deques at the beginning of the time step, then it has $(2+2\epsilon)p_i(t)$ work plus mugging steps in expectation.

Proof: $1/2$ of the deques are muggable at the beginning of the time step. Say the job has s steal attempts and w work steps. The expected number of mugging steps is $s/2$. Say that the total number of processor steps in the time step were $x \ge (4+4\epsilon)p_i$. Therefore, the total expected number of work plus mugging steps is $s/2+w = s/2+x-s = x-s/2 \ge x-x/2 = x/2 \ge (2+2\epsilon)p_i$. ∎

We can now argue that if time step t has many work plus mugging steps for a job that is not in OPT's queue, then this time step reduces this job's flow potential.

Lemma 4.7: If a job $J_i \in A(t)$ and $J_i \notin O(t)$, and this job does at least $(2+2\epsilon)p_i(t)$ work or mugging steps during this time step, then the change in flow potential due to A in this step is $\mathbb{E}\left[\frac{\mathrm{d}\Phi_i^A(t)}{\mathrm{dt}}\right] \le -\frac{20+20\epsilon}{\epsilon|A(t)|}\mathtt{rank}_i(t)$.

Proof: We know that $\mathbb{E}[p_i] = m/|A(t)|$. Therefore, the expected number of work plus mugging steps is $(2+2\epsilon)m/|A(t)|$. Each mugging reduces the number of muggable deques d_i^m by 1 in expectation. In addition, since this job is not in OPT's queue, each work step reduces this job's $Z_i(t)$ term by 1. Therefore, we can plug in this change in potential into the potential function to get $\mathbb{E}\left[\frac{\mathrm{d}\Phi_i^A(t)}{\mathrm{dt}}\right] \le -\frac{10}{\epsilon}\frac{\mathtt{rank}_i(t)}{m}\mathbb{E}\left[\frac{\mathrm{d}Z_i(t)+d_i^m(t)}{\mathrm{dt}}\right] \le -\frac{20+20\epsilon}{\epsilon|A(t)|}\mathtt{rank}_i(t)$. ∎

We now must argue about time steps that have a lot of steal attempts but not too many muggings. Here, we can use the original work stealing analysis showing that steal attempts

reduce steal potential and thus the critical-path term in the flow potential. We will use a known lemma from the paper [38].

Lemma 4.8: The depth-potential $\psi_i(t)$ never increases. In addition, if a job has d deques and there are d steal attempts between time t_1 and t_2, then $\Pr\{\psi_i(t_1) - \psi_i(t_2) \geq \psi_i(t_1)/4\} > \frac{1}{4}$. Hence, $E[\log \psi_i(t_2)] \leq E[\log \psi_i(t_1)] - \frac{1}{16}$.

We can now argue that a time step with "enough steal attempts" and not too many muggable deques reduces the critical-path term of the flow potential $\Phi_i(t)$.

Lemma 4.9: If job J_i has $p_i(t)$ processors and $d_i(t) \leq 2p_i(t)$ deques, then if the job has $2p_i(t)$ steal attempts or completes, the change in flow potential of this job due to A is $\mathbb{E}\left[\frac{\mathrm{d}\Phi_i^A(t)}{\mathrm{d}t}\right] \leq -200/\epsilon^2$.

Proof: From Lemma 4.8, we know that $\mathbb{E}\left[\frac{\mathrm{d}\psi_i(t)}{\mathrm{d}t}\right] \leq -1/16$ if it has enough steal attempts; the same is trivially true if the job completes. Plugging it into the potential, we get $\mathbb{E}\left[\frac{\mathrm{d}\Phi_i^A(t)}{\mathrm{d}t}\right] \leq -\frac{10}{\epsilon}\frac{320}{\epsilon}\frac{1}{16} \leq -200/\epsilon^2$ ■

We can now complete the proof of the running condition.

Proof of [Lemma 4.3] **Case 1:** At least $\epsilon/10\,|A(t)|$ jobs have more than $2p_i(t)$ steal attempts and $d_i \leq 2p_i(t)$. In this case, due to Lemma 4.9, each of these jobs reduces the flow potential by $200/\epsilon^2$; therefore, the total flow potential reduction due to A is at least $20/\epsilon\,|A(t)|$.

Case 2: At least $(1-\epsilon/10)\,|A(t)|$ have fewer than $2p_i(t)$ steal attempts or lots of deques $d_i > 2p_i(t)$. In the first case, this job has more than $(2+2\epsilon)p_i$ work steps in a straightforward way since there are a total of $(4+4\epsilon)p_i$ steps in that time step. In the second case, from Lemma 4.6, the time step has more than $(2+2\epsilon)p_i$ work plus mugging steps. Therefore, in either case, the total number of work and mugging steps is at least $(2+2\epsilon)p_i$.

In addition, from Lemma 4.8, we know that the algorithm can never increase the potential during execution. Hence, Claim 4.5 is still true. Therefore, we only need worry about the case where OPT has few jobs — fewer than $\epsilon\,|A(t)|\,/10$. In this case, among the $(1-\epsilon/10)\,|A(t)|$ jobs that have many work and mugging steps, at least $(1-\epsilon/5)\,|A(t)|$ of these jobs are in $A(t)$, but not in $O(t)$. We apply Lemma 4.7 on these jobs to obtain $\mathbb{E}\left[\frac{\mathrm{d}\Phi^A(t)}{\mathrm{d}t}\right] \leq \sum_{i\in A(t)\setminus O(t)} -\frac{20+20\epsilon}{\epsilon|A(t)|}\mathtt{rank}_i(t)$. Assuming the worst case that these are the lowest rank jobs we get the following change to the potential.

$$\begin{aligned}\mathbb{E}\left[\frac{\mathrm{d}\Phi^A(t)}{\mathrm{d}t}\right] &\leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\sum_{i=1}^{(1-\frac{\epsilon}{5})|A(t)|} i \\ &\leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\frac{(1-\frac{\epsilon}{5})^2|A(t)|^2}{2} \\ &\leq -\frac{1}{\epsilon}|A(t)|(10+3\epsilon) \quad [\epsilon \leq \tfrac{1}{2}]\end{aligned}$$

Therefore, in both cases, the flow potential reduces by at least $\frac{1}{\epsilon}|A(t)|(10+3\epsilon)$ due to A. Since OPT increases the flow potential by at most $\frac{10}{\epsilon}|A(t)|$ from Lemma 4.4 and we have $\frac{\mathrm{d}G_a(t)}{\mathrm{d}t} = |A(t)|$, the running condition is satisfied. □

It has been shown that the arrival, completion and running conditions hold. Thus, we can conclude that the work-stealing scheduler is constant competitive with $(4+4\epsilon)$ speed augmentation completing the proof of the main theorem.

V. Experimental Evaluation

This section presents the evaluation of DREP through both simulation and empirical experiments based on actual implementations. Simulations allow us to compare DREP with a wide variety of scheduling policies, including ones that are clairvoyant and/or infeasible to implement due to the need to *preempt at infinitesimal time steps*. The actual implementation allows us to evaluate DREP against a set of practical scheduling policies that are implementable but do not provide any theoretical bounds, including an approximation of Smallest Work First (SWF) [24], i.e., the SJF counterpart for parallel jobs, which is clairvoyant and work conserving. We obtain the actual implementations by modifying Cilk Plus [7], a production quality parallel runtime system, to approximate SWF and DREP and compare their performance in practice.

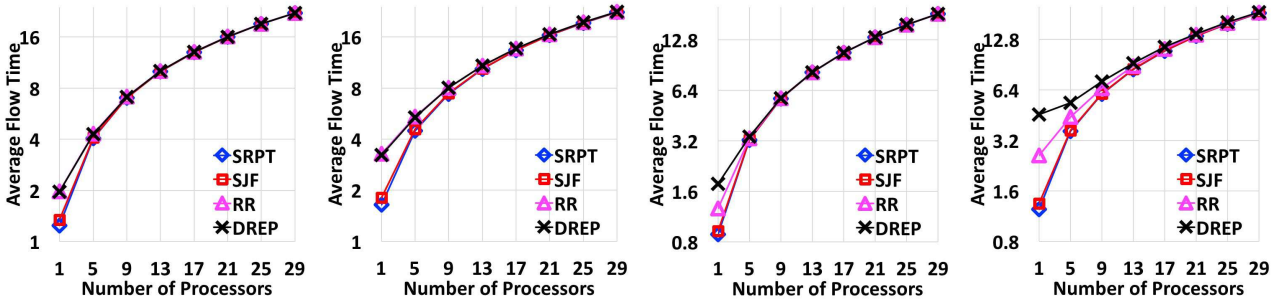
A. Evaluation Based on Simulations

Compared Algorithms: Via simulations, we compare DREP against a wide variety of schedulers: shortest-remaining-processing-time (SRPT) [3], shortest-job-first (SJF) [29] (which generalizes to smallest-work-first (SWF) [24] for parallel jobs), and round robin (RR) [26]. We compare against SRPT and SJF, because they are ***scalable***, i.e., $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for average flow for sequential jobs on multiprocessors. We also compare to RR, which is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive, because intuitively DREP simulates RR by uniformly and randomly partitioning cores across all active jobs.

It is important to note that all the existing algorithms, including the ones that we compared in the simulation, suffer from frequent preemptions, high overheads, and non-clairvoyance. LAPS [25], in particular, is very difficult to implement since it needs to know the parameter epsilon (speedup against the optimal) and preempts at infinitesimal time steps — it must process epsilon fraction of arriving jobs equally at any time. Because of this, LAPS is even difficult to implement in the simulation. Therefore, we do not compare against LAPS in the simulation experiments.

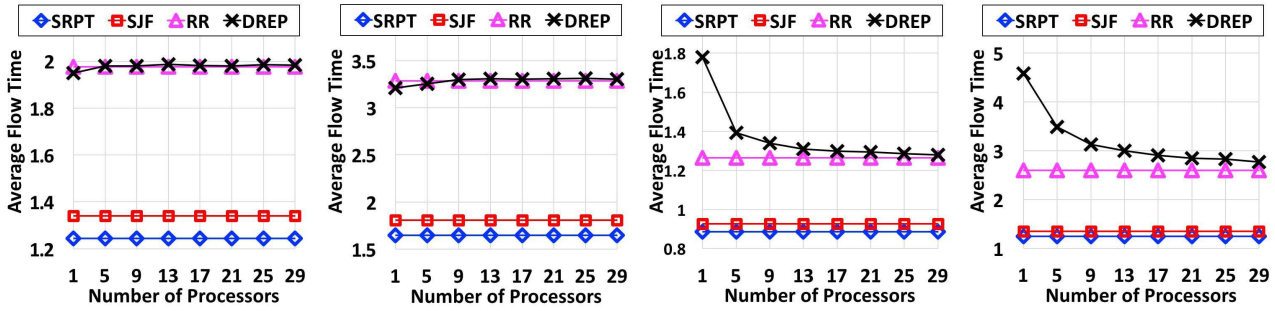
Moreover, the simulation results can be thought of as the lower bounds of what these scheduling algorithms can achieve, because they do not account for any scheduling or preemption overhead, which can significantly increase the average flow time in practice.

Setup: We use two different work distributions from real-world applications to generate the workloads: the ***Bing workload*** and the ***Finance workload*** [20]. We randomly generate a job by randomly sample its work from the experimented work distribution. For each work distribution, we vary the ***queries-per-second (QPS)*** to generate three levels of system loads: ***low*** ($\sim 50\%$), ***medium*** ($\sim 60\%$), and ***high*** ($\sim 70\%$) ***load*** (machine utilization), respectively. For a particular QPS, we randomly generate the inter-arrival time between jobs using a Poisson process with a mean equal to $1/QPS$. For each



(a) Finance workload, low load (b) Finance workload, high load (c) Bing workload, low load (d) Bing workload, high load

Fig. 1: Sequential jobs with multiprocessors setting with low and high machine utilizations



(a) Finance workload, low load (b) Finance workload, high load (c) Bing workload, low load (d) Bing workload, high load

Fig. 2: Fully parallel jobs setting with low and high machine utilizations

experiment setting, we generate $100,000$ jobs and report their average flow time under different schedulers.

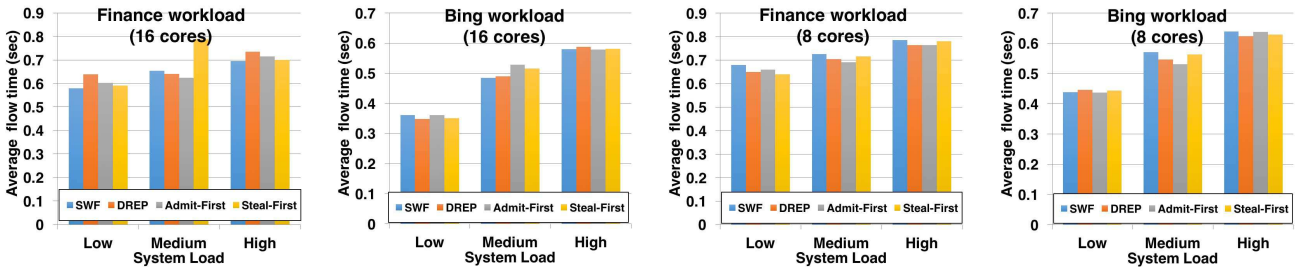
We also evaluate the impact on the average flow by increasing the number of processors. To ensure that the average machine utilization remains the same across experiments, we scale the amount of work of each job according to the number of processors.

We simulate two job settings: (1) the ***sequential jobs with multiprocessors setting***, where each job is sequential and can use only one processor at any time, and (2) the ***fully parallel jobs setting***, where each job obtains near-linear speedup with respect to the number of processors given. These two settings capture the two extreme cases of scheduling parallel jobs. Note that in our simulation experiments, we assume that all jobs are equally parallel since running accurate simulations with different and changing parallelisms is difficult. In our real experiments, we do not make this assumption.

Comparison: Figure 1 shows the results of simulating the sequential jobs on multiple processors setting, and Figure 2 shows the results for the fully parallel job setting. We only show the results with the low and high machine utilizations, since the trend is similar with medium utilization.

For sequential jobs on multiprocessors in Figure 1, SRPT and SJF have been proved to be scalable for average flow; but they are both clairvoyant, i.e., requiring the a priori knowledge of the amount of work for each job. In contrast, DREP and RR are non-clairvoyant and DREP's performance is very close to RR's performance in both workloads. When the number of processors is small, the gap between DREP/RR and SRPT/SJF is the widest while DREP gets close to optimal as the number of cores increases. This is because, intuitively, SRPT and SJF always work on the "right job", while DREP and RR give equal processing time to all jobs. In particular, with a small number of processors, DREP is more likely to encounter situations where smaller jobs that arrive later are stuck waiting for long jobs that occupy all the processors. Other schedulers either have the advantage of clairvoyant and thus know which jobs are smaller and should be processed first (e.g., SJF and SRPT), or they have the advantage of very frequent preemptions (e.g., RR), allowing them to preempt the long jobs in such scenario. DREP has comparable performance without such advantages and is thus more practical.

For the fully parallel job setting in Figure 2, we compare against SRPT and SWF. Since jobs are fully parallel, SRPT and SWF schedulers reduce to SRPT and SJF for sequential jobs on a single sequential machine (since the job with the least remaining work or the job with the smallest work will occupy the whole machine), so SRPT is optimal and SJF is scalable. Thus, in these experiments, these schedulers are operating in an "easier setting." In addition, SRPT and SJF can now devote all their processors to "the right job", while DREP may still get unlucky and not process small jobs that get stuck in the queue. Even so, the difference in performance is at most a factor of 3.25 compared to SRPT (which is optimal) and less than 3 compared to SJF (which is scalable). In this setting, DREP's performance is still close to RR and approaches RR as the number of cores increase. Note that on a small number of cores the gap between DREP and RR is larger on the Bing workload



(a) Finance workload on 16 core (b) Bing workload on 16 cores (c) Finance workload on 8 cores (d) Bing workload on 8 cores

Fig. 3: Parallel Cilk Plus jobs on multicore with varying system load and different work distributions

than the Finance workload. This is because Bing workload has some very large jobs. For other algorithms, this does not matter, as they can still finish short jobs fast by either being clairvoyant (SRPT, SJF) or doing many preemptions (RR). However, DREP will occasionally schedule a large job. With 1 core, this can have a large negative effect on the outcome. As the number of cores increase, this effect diminishes – therefore, DREP is worst on Bing with 1 core but converges to RR on many cores.

B. Evaluation Based on Real Implementation

To evaluate the empirical performance and practicality of DREP, we implemented a work-stealing based DREP in Cilk Plus [7], a widely-used parallel runtime system. For comparison, we implemented a few variants of work-stealing based scheduling strategies: steal-first [20], admit-first [20], and an approximation of smallest-work-first [24] explained below.

Setup: Similar to the simulations, we evaluate the schedulers using the Finance and Bing workloads. The data is collected on a 16-core machine with Linux 4.1.7 with RT_PREEMPT patch. Each data point presented is the average flow of an execution with $10,000$ jobs.

DREP Implementation: We implemented DREP in Cilk Plus by adding a global job queue. At the platform startup, a master thread inserts jobs into the job queue according to the workload specification. During the execution, a worker (a surrogate of a core) is assigned to an active job and only steals work from this job. By DREP, an active job is associated with $n/|A(t)|$ workers in expectation. This is achieved by letting the master thread determine upon a job arrival that whether a core should preempt with a probability of $1/|A(t)|$. If it determines that a core should preempt to work on the newly arrived job, it notifies the worker by setting a flag. Once the worker notices that the flag is set, it switches to work on the job specified by the master. In our current implementation, a worker checks whether this flag is set on steal attempts. In an improved implementation, a worker can check the flag at function calls, allowing the new job to be worked on faster while paying some small overheads of frequent checking. We left this implementation as our future work. Each active job keeps track of its associated deques. When a worker runs out of work, it randomly steals into the set of deques associated with the assigned job.

Other Scheduling Policies: We implemented several variants of work-stealing based schedulers and an approximation of SWF to compare with DREP. Both steal-first and admit-first extends the standard work-stealing algorithm by also incorporating a FIFO job queue. In ***steal-first***, a worker, upon running out of work, tries to steal work from other workers, favoring jobs that have started processing. Only when it cannot find any work to do among jobs that have started, it then admits a new job from the queue. ***Admit-first*** does the opposite — whenever a worker runs out of work, it always admits a new job from the queue, if there is one. Both admit-first and steal-first have been shown to work well for max flow time [18], especially steal-first which approximates FIFO. We also implemented an approximation of SWF, where every worker when running out of work, checks every active job in the system and works on the job with the smallest amount of work.

Comparison: Theoretically and from the simulations, SWF has performance advantages both by being clairvoyant and by requiring frequent preemptions. However, Figure 3 shows that DREP has comparable performance in practice with the work-stealing based SWF for all the different settings. In practice, preemption overhead is not negligible, so a scheduler cannot preempt very frequently. In particular, the approximation of SWF cannot immediately preempt the execution of a large job to work on the newly available work from a smaller job. In contrast, DREP tries to maintain an approximately equal number of workers (cores) to each active job, so that a large parallel job can hardly monopolize the entire system. The implemented steal-first in Figure 3 only bears $2n$ number of failed stealing attempts before admitting a new job. Its performance becomes worse when it allows more failed stealing attempts, which is thus not shown in the figure. Not surprisingly, DREP and admit-first have similar performance for average flow time. This is because admit-first keeps at least one worker per job when the number of active jobs is smaller than the number of cores. In addition, admit-first lets workers to randomly steal from each other, resulting in roughly equal resources between jobs, which is the same with DREP.

VI. Conclusion

In this paper, we introduced a practically efficient scheduler for optimizing the average flow time of parallel jobs. The

scheduler randomly distributes processors between the jobs, and each job uses work stealing to execute in parallel on its assigned processors. While this algorithm has a slightly worse theoretical guarantee than the best-known algorithm for the problem, it is the first provably efficient algorithm that has low enough overhead to use in practice for parallel jobs. The evaluations demonstrate its strong performance.

For future work, it is of interest to design schedulers for parallel jobs on processors of different speeds and prove for the best offline approximation ratio or online competitive ratio. This problem is much harder and is still not well-understood even offline. As far as the authors are aware, no prior work has addressed this problem theoretically in the online model.

ACKNOWLEDGMENT

This work is supported in part by a NJIT Seed Grants and NSF grants CCF-1845146, CCF-1830711, CCF-1824303, CCF-1733873, CCF-1527692, CCF-1150036 and CCF-1340571.

REFERENCES

[1] C. Chekuri, A. Goel, S. Khanna, and A. Kumar, "Multi-processor scheduling to minimize flow time with epsilon resource augmentation," in *STOC*, 2004, pp. 363–372.

[2] E. Torng and J. McCullough, "Srpt optimally utilizes faster machines to minimize flow time," *ACM Transactions on Algorithms*, vol. 5, no. 1, 2008.

[3] K. Fox and B. Moseley, "Online scheduling on identical machines using srpt," in *SODA*, 2011, pp. 120–128.

[4] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs, "Online weighted flow time and deadline scheduling," *Journal of Discrete Algorithms*, vol. 4, no. 3, pp. 339–352, 2006.

[5] C. Bussema and E. Torng, "Greedy multiprocessor server scheduling," *Operations research letters*, vol. 34, no. 4, pp. 451–458, 2006.

[6] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. Montreal, Quebec, Canada: ACM, 1998, pp. 212–223.

[7] Intel, "Intel Cilk™ Plus," https://www.cilkplus.org/.

[8] C. E. Leiserson, "The Cilk++ concurrency platform," *Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, March 2010.

[9] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson, "Programming with exceptions in JCilk," *Science of Computer Programming*, vol. 63, no. 2, pp. 147–171, Dec. 2008.

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

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

[12] O. Tardieu, H. Wang, and H. Lin, "A work-stealing scheduler for x10's task parallelism with suspension," in *PPoPP*, 2012.

[13] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, "The Habanero multicore software research project," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 2009, pp. 735–736.

[14] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: the new adventures of old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 2011, pp. 51–61.

[15] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *JACM*, vol. 46, no. 5, pp. 720–748, 1999.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, July 1995, pp. 207–216.

[17] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2000, pp. 1–12.

[18] K. Agrawal, J. Li, K. Lu, and B. Moseley, "Scheduling parallelizable jobs online to minimize the maximum flow time," in *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. Pacific Grove, California, USA: ACM, 2016, pp. 195–205.

[19] K. Agrawal, Y. He, and C. 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, pp. 112–120.

[20] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley, "Work stealing for interactive services to meet target latency," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 14:1–14:13. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851151

[21] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. D. Gill, and C. Lu, "Randomized work stealing for large scale soft real-time systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 203–214.

[22] S. Leonardi and D. Raz, "Approximating total flow time on parallel machines," *Journal of Computer and Systems Sciences*, vol. 73, no. 6, pp. 875–891, 2007.

[23] B. Kalyanasundaram and K. Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM*, vol. 47, no. 4, pp. 617–643, 2000. Preliminary version in *FOCS* 1995.

[24] K. Agrawal, J. Li, K. Lu, and B. Moseley, "Scheduling parallel DAG jobs online to minimize average flow time," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, 2016, pp. 176–189.

[25] J. Edmonds and K. Pruhs, "Scalably scheduling processes with arbitrary speedup curves," *ACM Transactions on Algorithms*, vol. 8, no. 3, p. 28, 2012.

[26] J. Edmonds, "Scheduling in the dark," *Theor. Comput. Sci.*, vol. 235, no. 1, pp. 109–141, 2000. Preliminary version in *STOC* 1999.

[27] K. Fox, S. Im, and B. Moseley, "Energy efficient scheduling of parallelizable jobs," *Theoretical Computer Science*, vol. 726, pp. 30–40, 2018.

[28] N. Barcelo, S. Im, B. Moseley, and K. Pruhs, "Shortest-elapsed-time-first on a multiprocessor," in *Design and Analysis of Algorithms - First Mediterranean Conference on Algorithms, MedAlg 2012, Kibbutz Ein Gedi, Israel, December 3-5, 2012. Proceedings*, 2012, pp. 82–92.

[29] C. Bussema and E. Torng, "Greedy multiprocessor server scheduling," *Operations Research Letters*, vol. 34, no. 4, pp. 451–458, 2006. [Online]. Available: https://doi.org/10.1016/j.orl.2005.07.005

[30] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng, "Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics," *J. Scheduling*, vol. 6, no. 3, pp. 231–250, 2003.

[31] R. Ebrahimi, S. McCauley, and B. Moseley, "Scheduling parallel jobs online with convex and concave parallelizability," in *Approximation and Online Algorithms - 13th International Workshop, WAOA 2015, Patras, Greece, September 17-18, 2015.*, 2015, pp. 183–195.

[32] J. Edmonds, S. Im, and B. Moseley, "Online scalable scheduling for the ℓ_k-norms of flow time without conservation of work," in *ACM-SIAM Symposium on Discrete Algorithms*, 2011.

[33] A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs, "Scheduling jobs with varying parallelizability to reduce variance," in *Symposium on Parallel Algorithms and Architectures*, 2010, pp. 11–20.

[34] H. Chan, J. Edmonds, and K. Pruhs, "Speed scaling of processes with arbitrary speedup curves on a multiprocessor," *Theory of Computing Systems*, vol. 49, no. 4, pp. 817–833, 2011.

[35] K. Pruhs, J. Robert, and N. Schabanel, "Minimizing maximum flowtime of jobs with arbitrary parallelizability," in *Approximation and Online Algorithms - 8th International Workshop, WAOA 2010, Liverpool, UK, September 9-10, 2010. Revised Papers*, 2010, pp. 237–248.

[36] J. Robert and N. Schabanel, "Non-clairvoyant scheduling with precedence constraints," in *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '08, 2008, pp. 491–500.

[37] S. Im, B. Moseley, and K. Pruhs, "A tutorial on amortized local competitiveness in online scheduling," *ACM SIGACT News*, vol. 42, no. 2, pp. 83–97, 2011.

[38] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *10th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998, pp. 119–129.