Summer 8-15-2019

Scheduling Multiple Parallel Jobs Online

Kefu Lu
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
I-Ting Angelina Lee, Chair
Kunal Agrawal
Michael Bender
Jeremy Buhler
Roman Garnett
Benjamin Moseley

Scheduling Multiple Parallel Programs Online

by

Kefu Lu

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2019
St. Louis, Missouri

Contents

List of Figures

Acknowledgments

I am very grateful for all the support given to me by everyone during my graduate studies. Numerous are those who have supported me throughout the years; now, near the end of my graduate studies, it is finally time for me to acknowledge and offer my sincere gratitude to all of these people.

I would like to thank Benjamin Moseley for his guidance during my graduate career. He gave me the inspiration to continue my studies in algorithmic theory at the beginning of my graduate career. Without him, this thesis would not have been possible. I am very grateful for all his efforts in guiding and teaching me. I have learned a great deal from him, both in the realm of research and in other areas. He has always been there for me and I am thankful for the great advice he offered me when I needed it the most. In my future career, I can only aspire to offer such sagacious advice onto others.

I have had many amazing collaborators whom I am very grateful to have worked with. First I would like to thank Kunal Agrawal. In addition to being an amazing researcher, she had an enormous impact on my graduate career. She was also my teacher when I first learned about algorithms many years ago. She has provided many brilliant insights and had an influence in all of the results in this thesis. I would also like to thank Angelina Lee, Jeremy Buhler, and Roman Garnett for their inspirations in research. I am very thankful for the opportunities I had to work together with all these people. Their insight and guidance were invaluable to me and they were instrumental in many of my results during my graduate career.

I would also like to thank my fellow students and collaborators: Jing Li, Gustavo Malkomes, Thomas Lavastida, Yuyan Wang, Shaurya Ahuja, Robert Utterback and Shamoli Gupta. I have enjoyed working on all manners of research with them, whether in theory or in implementation. I was fortunate to have worked with all of them. I also give special thanks to Jing Li. I was my advisor's first student; she supported me as a senior student and gave me plenty of helpful advice on my thesis, my career, and my presentations, in addition to research. I also thank many other fellow students for making my graduate studies enjoyable, though I did not have a chance to work together with them on research: Evan Balzuweit, Adam Drescher and Shali Jiang. I would also like to thank my friends during my graduated studies whose support I greatly benefited from, sadly I cannot name all of you here.

I must also thank my parents for all the support they have given me over the years. Though they knew nothing about computer science, they were still nevertheless always tried to encourage me onwards, thought not always very efficiently, as my results agglomerated over the years. I dedicate this thesis to them.

Kefu Lu

Washington University in Saint Louis

August 2019

ABSTRACT OF THE DISSERTATION

Scheduling Multiple Parallel Programs Online

by

Kefu Lu

Doctor of Philosophy in Computer Science

Washington University in St. Louis, August 2019

Research Advisor: I-Ting Angelina Lee

The prevalence of parallel processing has only increased in recent years. Today, most computing machines available on the market shifted from using single processors to possessing a multicore architecture. Naturally, there has been considerable work in developing parallel programming languages and frameworks which programmers can use to leverage the computing power of these machines. These languages allow users to create programs with internal parallelism. The next, and crucial, step is to ensure that the computing system can efficiently execute these parallel jobs.

Executing a single parallel job efficiently is a very well-studied problem in parallel computing. In the area of job scheduling, there is extensive work on scheduling multiple *sequential* jobs to minimize important objectives. However, there is little work on scheduling multiple jobs that have internal parallelism.

This dissertation focuses on designing theoretically efficient and practically good scheduling algorithms for parallelizable jobs in the identical machines setting. Specifically, this research consider jobs in the Directed-Acyclic-Graph (DAG) model of parallelism and studies the problem of scheduling multiple DAG jobs to optimize objectives such as average flow time, maximum flow time, and throughput. The overarching goal of the research is to deeply examine the problem of scheduling multiple parallel jobs and to take the first steps towards creating a body of knowledge comparable to the extensive amount of existing work on scheduling sequential jobs.

Chapter 1

Introduction

One of the main goals in computer science is to efficiently perform computation. In recent years, computing systems have become more and more parallel in nature due to physical limitations and the need to reduce power consumption. Most computing devices, ranging from cellphones to desktop computers to servers, now have multiple processors. This trend will only continue into the future. Therefore, exploiting the parallelism of computing systems will only grow in importance. To achieve this goal, many techniques have been developed to allow programmers create internal parallelism in the tasks they seek to achieve. Libraries and languages such as Cilk[13], Intel Thread-Building Blocks[28], and OpenMP[37] are a few examples of these technologies. Programmers can use these technologies to create programs, or *jobs*, that multiple processors can work on at the same time to complete at a faster rate. The efficient execution of these programs is an important area of study in parallel computing.

In another setting, interactive services such as web search and online gaming are hosted on clouds and servers. In this sort of situation, the service provider must handle requests, or jobs, that arrive from clients over time. Latency is a metric that many clients pay close attention to, therefore, service providers seek to minimize latency in order to satisfy their customers. In order to reduce the amount of processing time necessary to complete client requests, these services often run on massive parallel machines with many processors. Therefore it is important for the server to utilize all the resources available to it in order to complete jobs efficiently. In this case, the server must know how to handle and execute multiple parallelizable jobs which clients may submit. Knowing provably efficient algorithms for this sort of *job scheduling* problem is of great benefit to the service providers.

The focus of this thesis is scheduling multiple parallelizable jobs in the Directed-Acyclic-Graph (DAG) model online for many different objectives including maximum flow time, average flow time, and throughput. These objectives and the model will be defined more precisely in the following sections. Nevertheless, we cannot begin this work without acknowledging the fact that scheduling jobs is a massive area in theoretical computer science and that there is a large body of work on scheduling sequential jobs for all of these objectives.

The goal of the thesis is to achieve an understanding of scheduling parallel DAG jobs that complement the immense amount of knowledge on scheduling sequential jobs (of which [39] is a good survey, albeit slightly old).

1.1 Client-Server Scheduling Model

Scheduling is a problem which arises in many areas of computer science ranging from operating systems to distributed computing. Therefore it is unsurprising that it is a rich area of study in theoretical computer science. The particular type of scheduling this thesis considers is client-server scheduling. In this model, there are many *clients* which, over time, send jobs to a *server*. The server must make decisions over the order in which the jobs are processed. Nowadays, the server itself may be a very powerful system composed of many processors, designed with the intention to process jobs quickly. Client-server scheduling has received much attention because it captures many common applications such as when a search engine receives web search requests from users, or when a cloud computing platform receives tasks to perform for clients. On a smaller scale, it is also similar to the type of problem an operating system would face when the user tries to run many different programs.

In client-server scheduling, there is a set J of n jobs (from clients) which arrive over time at a server. The server can complete a job by processing it, but a job may only be processed after it arrives. In the *offline* version of this problem, the server knows all information about all the jobs ahead of time. For instance, it knows the entire arrival sequence of the n jobs and also knows how long each job must be processed in order to complete the job. This version of the problem, though somewhat restrictive, is still suitable for many applications such as allocating computing time on a large computing cluster when all the jobs submitted are can be known. Alternatively, there is the *online* problem. Here, the server does not know the arrival sequence of the jobs. It will only learn of a job's existence when it arrives at the server. This is more natural for applications such as web searches, where a server does not have any idea about when users will submit a search. In this case, the server must schedule jobs without knowing about possible future job arrivals. This thesis focuses on the online scheduling problem. Therefore, most of the results and problems mentioned from this point forward will be online scheduling problems.

1.1.1 Objectives in Online Scheduling

Usually, the goal of scheduling to efficiently perform jobs. There are many different metrics that have been considered in online scheduling literature. In this section we will describe the ones that this thesis examines[1]:

- **Flow Time:** Response time or latency is one of the most widely used metric in online scheduling. In online scheduling literature, this quantity is usually referred to as the *flow time* whereas the other two terms are the more commonly understood terms used in other areas. The flow time of a specific job is defined as the difference between its completion time and arrival time. This is the exactly amount of time that the job spends at the server before it is completed. Formally, for job J_i with arrival time r_i and completion time c_i in the schedule, the flow time is defined as $F_i = r_i - c_i$. Depending on the scheduling algorithm, the job may be completed at different times, resulting in a different flow time for the job. Flow time is a quantity defined for each specific job, leading to several objectives which can be defined for the set of all jobs overall.

  – Average Flow Time: This is the average flow time over all the jobs. Minimizing this quantity minimizes the average amount of time a job spends in the system. This objective corresponds to the average quality of service that the server provides to a job. Average flow time is the most well-explored objective in online scheduling. Note that while minimizing this quantity ensures a good average quality of service, there are disadvantages to this objective such as fairness; the server can provide good service to most jobs but terrible service to a few jobs while still achieving a small average flow time. Also, minimizing the total flow time over all

[1]Note that the flow time objectives are minimization problems and the throughput objective is maximization

the jobs is the same as minimizing the average flow time since they only differ by a fixed constant n, the number of jobs. In literature, total flow time is often the actual objective being minimized. Hence, formally, the objective function often appears like this:

$$\textit{total flow time} := \sum_{J_i \in J} F_i = \sum_{J_i \in J} c_i - r_i$$

An alternative way to think about this objective is that each unfinished job contributes 1 to the total flow time objective per instant of time. Thus, if the set of unfinished jobs at any point in time in the schedule is denoted by the function $J(t)$, the total flow time objective can also be written this way:

$$\textit{total flow time} = \int_0^\infty |J(t)| dt$$

- Maximum Flow Time: This is the largest flow time across all the jobs. In contrast to average flow time, minimizing this objective minimizes the worst case service any job receives. Under this objective, a server cannot get away with providing terrible service to even a single job. This objective is useful for ensuring fairness between every job, an important consideration in many types of scheduling applications. Formally:

$$\textit{maximum flow time} := \max_{J_i \in J} F_i = \max_{J_i \in J} c_i - r_i$$

- Weighted (Average or Maximum) Flow Time: In some cases, not all jobs have equal priority to a server. In these instances, each job has a positive weight w_i associated with it where a higher weight represents higher priority. Each job's flow time is then multiplied by this weight to obtain a weighted flow time for each job. The weighted maximum flow time and weighted average flow time can be defined naturally using the weight flow time for each job. Alternatively, one can think of these as the weighted generalizations of the previous objectives.

- **Throughput:** Maximizing the throughput of a schedule is an objective which often arises from the need to process a large number of jobs. In the throughput maximization problem, each job which arrives to the server has an associated relative deadline D_i. This means that if the job is to be completed, the job must be completed within D_i units of time after it has arrived. From the server's perspective, one useful goal is to maximize the number of jobs that are completed by their deadlines because doing so means that the server is efficiently completing jobs. Jobs in the throughput maximization problem actually each possess a profit p_i that is awarded if it is completed by its deadline. The objective is to create a schedule that maximizes the amount of profit. If all jobs had the same profit, this is the same as maximizing the number of jobs completed by their deadlines. However, the problem is more general when profits are allowed to differ.

1.2 Background on Online Scheduling

In online scheduling, the scheduling algorithm does not know of job arrivals ahead of time. It must make decisions without knowledge of the future. There are inherent difficulties for such algorithms due to this requirement. Therefore, most of the time the algorithm cannot create the optimal schedule. The principal way to analyze these online algorithms is to use *competitive analysis*, detailed in the following paragraph.

For any input sequence I to an online algorithm there exists an optimal solution which achieves a objective of $\mathrm{OPT}(I)$. Let an online algorithm A achieves an objective of $A(I)$ on the same instance. For a minimization problem, the algorithm A is said to be a c-competitive algorithm if for any input sequence I, the objective it achieves is at least $A(I) \leq c \cdot OPT(I)$. Likewise, for maximization the algorithm A must achieve $A(I) \geq \frac{1}{c} OPT(I)$ for any input sequence I. The constant c is known as the *competitive ratio*.

Competitive analysis and the competitive ratio is worst-case analysis since there could be input instances which result in a much worse ratio compared to others, but the definition requires c to hold for all instances. The goal in designing online algorithms is to create algorithms that have achieve good competitiveness. For many problems in online scheduling, we would like to have $O(1)$-competitive algorithms - in this case c is a constant value.

1.2.1 Resource Augmentation

However, many online scheduling problems are very difficult and do not admit any $O(1)$-competitive algorithms. For instance, there are some objectives for which there exist strong, super constant lower bounds. Theoreticians have put forth the technique of *resource augmentation* in order to better understand these objectives. Resource augmentation is a (now) standard form of analysis where the algorithm is allowed more resources than the optimal solution that it is compared to. For example, in online scheduling the algorithm is analyzed to be run on either extra processors or faster processors compared to the optimal schedule. The latter, called *speed augmentation*, is the most common form of resource augmentation analysis for online scheduling. An algorithm with s speed will process jobs s times faster than the optimal schedule is allowed to process jobs[2]. In general, speed augmentation is more powerful than allowing the equivalent amount of extra processors[3].

In resource augmentation analysis, an algorithm A is said to be s-speed c-competitive if it achieves a competitive ratio of c while using s times the speed that is given to the optimal schedule in the analysis. The strongest possible theoretical result for some online scheduling problems is a *scalable* algorithm, which is an algorithm that is $(1+\epsilon)$-speed $O(1)$-competitive for some constant $\epsilon \in (0, 1]$. Of course, the $O(1)$ in the competitive ratio actually hides some function of ϵ, but in literature it is often written this way as $O(1)$ instead of $O(f(\epsilon))$ with some function f, since ϵ is a constant.

Resource augmentation is useful for a few reasons:

- It allows system designers to feel secure in knowing the performance achieved by the scheduler. For example, when a system is designed with a specific performance target in mind. The system designer knows that the performance of the system is better than the optimal performance that any schedule can achieve on a system with slightly less speed.

- It allows theoreticians and practitioners to differentiate between algorithms. Without resource augmentation, most algorithms are terrible at some online scheduling problems since there are strong lower bounds; this does not allow us to predict which algorithms would work well in practice. With resource augmentation, it is still the case that not all scheduling algorithms will be scalable or achieve $O(1)$-competitiveness, but some

[2]Usually, OPT is given a speed of 1; the algorithm is given a speed of s.

[3]To see this, consider a job which can only be run on a single processor at a time. Here, having a single processor that is twice as fast is beneficial, while having two processors with normal speed does not allow the job to be completed any faster.

will. Resource augmentation allows us to identify which algorithms would likely work well in practice. Such predictive power is one of the main goals for algorithm analysis!

1.2.2 Scheduling Setting

J is the set of n input jobs which arrive over time to the server. Preemptions are allowed in our problems, this means that the scheduler can pause a job and restart it at a later time without any cost. Our problems are also online, thus, the algorithm does not know of future job arrivals. Furthermore, an algorithm is *non-clairvoyant* if it also learns no other information about a job once the job arrives, most importantly, it does not know the amount of processing that a job requires. In contrast, a *clairvoyant* scheduler knows all information about a job once it arrives. Non-clairvoyant scheduling is significantly more difficult for many online problems. We will usually state whether an algorithm is non-clairvoyant when it is described.

The server will be composed of m processors, or machines. These two terms are used interchangeably. This thesis considers the *identical machines* setting where all m processors are the same. In the resource augmentation analysis, the algorithm is allowed to have s speed for all the processors.

As mentioned previously, the main difference in this thesis compared to previous work is that we focus on scheduling parallelizable jobs. In the next section we will define the model for these parallel jobs which will be used throughout this thesis.

1.2.3 Parallel Jobs

In this thesis we focus on programs created through dynamic multithreading. This sort of parallelism is commonly found in many parallel libraries and languages such as Cilk[13], Intel Thread-Building Blocks[28], and OpenMP[37]. Dynamic multithreading is popular as the programmer only needs to express algorithmic parallelism without the need to deal with specifically binding computations to processors. The library or language itself handles the actual execution of the program and it is important for it to schedule the program efficiently.

A dynamic multithreaded job J_i can be represented as a *Directed-Acyclic-Graph (DAG)* G_i. Nodes of the DAG will represent tasks and edges will correspond to dependencies. The job will be complete once all of its tasks have been completed. The time when the all nodes of the DAG are completed is the completion time of the job (c_i).

A node (task) is a series of instructions for the processor. Each node v in G_i has an associated processing time p_v; the instructions in node v must be processed sequentially on a processor for p_v time to be completed. The edges in the DAG represent dependencies; a node in G_i cannot be executed until all of its predecessors in G_i have been executed. We say that a node is *ready* if all of its predecessors have been processed. Multiple ready nodes for the same job can be processed at the same time, hence, parallel processing is possible. Each processor can only work on one node at a time.

Figure 1.1 depicts a DAG job with 7 nodes with some nodes that have different processing times.

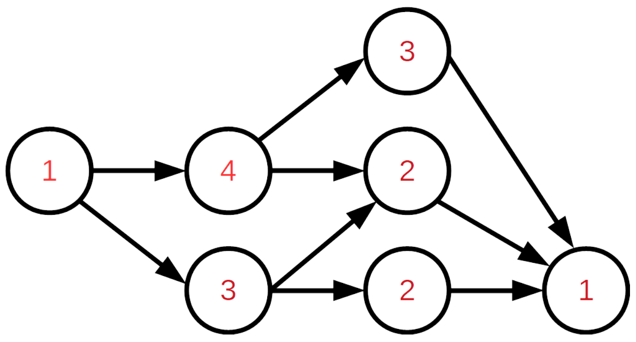


Figure 1.1: A parallel job modeled as a directed-acyclic-graph. The processing time of each node is located at the center of each node. The critical path length, the longest path through this DAG, is 9. The total work, sum of all the processing times, is 16

It is assumed that the scheduler does not know the DAG of each job in advance; the DAG structure unfolds dynamically as the job executes. Realistically, the scheduler only knows of the current ready nodes of the job.

Usually it is unnecessary to involve the exact DAG structures of the jobs in the analysis of the scheduling algorithms. Instead, for each job J_i there are two important parameters frequently used throughout the analysis:

- *total work* W_i. (T_1 in literature). This is the sum of the processing times of all the nodes. On a single processor, completing the job will take this amount of time.
- *critical path length* P_i. (T_∞ in literature). This defined as the total processing time along the longest path through the DAG. Note that even given infinite processors, this is the amount of time it takes to complete this job. In literature this is occasionally referred to as the *span* of a job.

In practice, measuring the exact DAG structure of a parallel program is very difficult; measuring the total work or the critical path is possible through profiling.

A DAG, with all of its nodes and edges, represents a single program. Efficiently scheduling a single DAG job is a well studied problem both theoretically and in practice. In this thesis, our task is to schedule multiple DAG jobs in order to optimize the objective functions described in the section 1.1.1. We seek to leverage both classic scheduling theory and the DAG model of parallelism in order to develop provably good and practical algorithms for these tasks.

In this thesis we will describe schedulers which may use several different levels of knowledge about the parallel program itself. The most complete level of information one can have about a DAG job is to know the entire DAG structure of the job. A scheduler which has this level of knowledge will be known as *clairvoyant*. A lesser level of knowledge would be if the scheduler is allowed to know the quantities W_i and P_i, which are the total work and critical path, in addition to the current ready nodes it may process. Though these extra quantities are not as powerful as the entire DAG structure, they still provide some crucial information about the job. We refer to schedulers which access this level of information as *semi-non-clairvoyant*. If the scheduler is only allowed to know which current ready nodes may be processed, we will refer to it as *non-clairvoyant* since it only has a very basic level of knowledge about the job. One might ponder whether it is technically possible for a scheduler to also not know which current ready nodes may be processed. However, this would similar to not allowing a machine to know which instructions it can process next in order to continue running a program. For most practical purposes, such a low level of information is too restrictive. We might refer to this level as *complete-non-clairvoyant* but we do not consider schedulers with this level of information in the thesis.

1.3 Overview

Chapter 1.3 contains a detailed table of notation which will be used throughout the thesis. Of course, each individual result may have additional notation described in their own chapter. It is followed by a summary of related work and a summary of my contributions on scheduling parallel jobs in chapter 2. After this summary, each chapter which follows will contain details of each result mentioned in chapter 2. With chapter 3 focusing on minimizing the average flow time, chapter 4 focusing on the maximum flow time, chapter 6 on the throughput, and chapter 5 on more practical algorithms for average flow time. Then, I offer some concluding remarks in the final chapter of the thesis.

Table of Notation

General

OPT OPT	The optimal solution or optimal objective
m	The number of processors

Jobs in General

n	Number of jobs
J	Set of jobs
J_i	i-th job
r_i	Arrival time of job J_i
c_i	Completion time of job J_i
w_i	Weight of job J_i
D_i	Relative deadline of job J_i
d_i	Deadline of job J_i
p_i	Profit of job J_i

Directed-Acyclic-Graph (DAG) Jobs

G_i	Directed-Acyclic-Graph of Job J_i
v	A node in a DAG
p_v	Processing time of node v
W_i	Total work of job J_i
P_i	Critical path length of job J_i

Scheduling Algorithm Names

SRPT	Shortest-Remaining-Processing-Time
FIFO	First-in-first-out (Chapter 4)
LAPS	Latest-Arrival-Processor-sharing (Chapter 3)
BWF	Biggest Weight First (Chapter 4)
SWF (SJF)	Smallest Work First (Shortest Job First) (Chapter 3
RR	Round-Robin
DREP	Distributed Random Equi-Partition (Chapter 5)

Chapter 2

Overview and Related Work

This chapter will provide an overview of the various results in this dissertation. First we provide some context by describing similar work on related areas. The thesis focuses on scheduling multiple parallelizable jobs to optimize several different objectives; there are two natural problems that align closely against this focus. Firstly, the problem of scheduling a single parallelizable job on multiple processors. This is well-studied for the DAG model of parallelism and here are various known algorithms such as greedy scheduling and work stealing [24, 13, 14, 1]. Secondly, scheduling sequential jobs to optimize objectives such as flow time is also an extensively studied problem both for a single processor and for multiple processors [4, 8, 7]. It is important to have some understanding of both of these problems before describing the principal results of this thesis.

We begin by giving a quick summary of scheduling techniques for a single DAG job, then an overview of online scheduling results. Finally we describe our contributions to the problem of scheduling multiple parallelizable jobs.

2.1 Work Stealing

In most parallel programming libraries, the programmer only specifies the algorithmic parallelism in the program. This determines the DAG structure of the program. The parallel library usually provides a runtime system to execute these DAG jobs efficiently. When running a single DAG job, at any moment in time there are ready nodes which should be processed. At a high level, there are two main strategies for the runtime system: centralized scheduling and work-stealing scheduling [13, 14].

In centralized scheduling, the runtime system keeps a centralized datastructure with all the current ready nodes. The datastructure is shared by all processors and contains exactly which ready nodes must be processed. The benefit of a centralized scheduler is that it is a *greedy scheduler* where no processor will idle if there exists ready nodes to be processed. In practice, however, this scheduler often has high overhead because access to the centralized datastructure itself must be strictly controlled. These synchronization overheads can lead to poor performance.

In randomized work-stealing [14], there is no centralized datastructure used to keep all the current ready nodes. Instead, each of the m processors has an associated local double-ended queue (deque). When a processor enables new ready nodes, it pushes the node to the bottom of its deque. When the processor completes its current node, it takes the first ready node from the bottom of its deque. If there are no nodes in its deque to process, the processor will attempt to steal a node from a random processor's deque instead. When performing a steal, the *thief* will always pop the ready node at the top of the *victim*'s deque. If the victim does not have any work in its deque, the steal attempt fails. The benefit of this scheduler is that it is often efficient in practice. There is no centralized queue which must be strictly controlled. The only source of contention is when a thief must steal from the deque of a victim. Since the thief peeks at the top of the deque while the victim usually only peeks at the bottom, there is little synchronization necessary most of the time. However, due to being a randomized scheduler, work-stealing does not strictly have the greedy property. Even when there are many ready nodes available there may still be processors making steal attempts which fail due to the random choice of the victim. Though, it is possible to prove probabilistic bounds on the performance of the randomized work-stealing scheduler.

The key idea to understand here is that we know of algorithms to schedule a single DAG job and furthermore, we know that randomized work-stealing is a practical scheduler for a single DAG job that have been implemented in programming languages[13].

2.2 Online Scheduling

There exists an extensive amount of work on scheduling sequential jobs. [39] is a useful survey of results in this area. The thesis will focus on the objectives of average flow time, maximum flow time, and throughput, as described in section 1.1.1. Therefore, we will highlight results concerning those objectives. The most important distinction to note is that almost all of the work described in this section will deal with sequential jobs - these jobs do not experience any benefit from being processed by multiple processors at the same time.

Minimizing the average flow time is the most popular objective in online scheduling. In the case of sequential jobs on m identical processors, it is known that any algorithm is $\Omega(\min\{\log P, \log n/m\})$-competitive where P is the ratio of the largest to smallest processing time of the jobs [34]. This competitive ratio is in fact achieved by the algorithm Shortest-Remaining-Processing-Time (SRPT). This is a case where there is a strong lower bound on the possible performance of any scheduling algorithm. Due to these strong lower bounds, previous work has considered an analysis using the resource augmentation technique described in section 1.2.1 which was first introduced by Kalyanasundaram in [30]. With resource augmentation, several algorithms are known to be scalable, meaning that they use $(1 + \epsilon)$-speed to achieve $O(f(\epsilon))$-competitiveness for average flow time [18]. Here $\epsilon > 0$ and f is a function which depends only on ϵ. Several algorithms are scalable for this problem including SRPT and Shortest-Job-First (SJF) [45, 40, 11, 15]. In summary, we know which algorithms work well for average flow time in the sequential jobs setting.

Minimizing the maximum flow time is another important objective. For sequential jobs on m identical processors, the algorithm First-In-First-Out (FIFO) achieves a competitiveness ratio of $(3/2 - \frac{1}{m})$ [4, 12]. This is a very strong result which does not require resource augmentation. Weighted maximum flow is a very similar objective where different jobs are given different weights. Minimizing the weighted maximum flow time is much more difficult as it can be shown that any algorithm is $\Omega(W^{.4})$-competitive where W is the ratio of the maximum weight to minimum weight. This is true even when jobs are both sequential and unit sized [19].

The throughput maximization problem is another difficult online scheduling problem. Even for a single processor, there exists a deterministic algorithm which is $O(\delta)$-competitive, where δ is the ratio of the maximum to minimum density of a job [9, 10, 33, 46]. Here the *density* of job J_i is $\frac{p_i}{W_i}$ (the ratio of its profit to its work). Also, this is the best possible result for any deterministic online algorithm even if all jobs have unit profit. In the case where the algorithm can be randomized, $\Theta(\min\{\log \delta, \log \Delta\})$ is the optimal competitive ratio [29, 32]. Here Δ is the ratio of the maximum to minimum job processing time. With resource augmentation, there is an $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive algorithm for any fixed $\epsilon > 0$ [30].

Scheduling parallelizable jobs is not an entirely new problem. However, much of thee previous work focuses on the *arbitrary speed-up curves* model of parallelism. This is a model of parallelism very different from the DAG model. In the speed-up curves model, each job J_i is associated with a sequence of phases. Note that this is a linear path of phases, unlike a DAG. The j-th of job J_i is denoted by a tuple $(W_{i,j}, \Gamma_{i,j}(m'))$. The value $W_{i,j}$ is the total work of the j-th phase of job J_i. Each phase may only be worked on when all the work on the previous phase has been completed. The value $\Gamma_{i,j}(m')$ is a speed-up function that specifies the processing rate of the phase when given $1 \leq m' \leq m$ processors. It is usually assumed that $\Gamma_{i,j}(m')$ is a nondecreasing sublinear function. The speed-up function for each phase serves to model the parallelism in the jobs. The linear sequence of phases models how the parallelism of a job may change as it is being processed. The arbitrary speed-up curve model was first introduced by [20].

There are many differences between the speed-up curves model and the DAG model which we will focus on. It is still an interesting question whether the two models are equivalent, but one model cannot be trivially used to simulate the other. For example, in the arbitrary speed-up curves model, the current speed-up function only depends on the phase of the job, which in turn only depends on the total amount of work already done on the job. In the DAG model the current parallelism not only depends on the work done on the job but also the exact nodes which have been done since there are precedence constraints between individual strands of work. This presents more difficulty for the scheduler since the DAG structure is unknown to the algorithm. It is unclear how to simulate these sort of DAG jobs in the arbitrary speed-up curves model. Similarly, jobs in the speed-up curves model also do not translate to the DAG model easily. Consider that the DAG model, the parallelism of a job scales linearly up to the number of current ready nodes. However, in the speed-up curves model, the speed-up functions are allowed to be any concave sublinear function. These arbitrary concave functions are not easily simulated by DAGs. Due to these differences, it is not clear whether algorithms in one model would work well for the same problem in the other model.

In the speed-up curves model, there are several known results for the flow time objectives in particular. For minimizing the average flow time, an algorithm called Latest-Arrival-Processor-Sharing (LAPS) is known [22]. This algorithm is scalable for average flow time. The analysis of LAPS is also notable for its introduction of a technique known as amortized local competitiveness, which has since become a very important technique in online scheduling. LAPS and this analysis technique have been very influential in scheduling theory [17, 6, 26, 21, 25, 16, 23].

For the problem of minimizing the maximum flow time in the arbitrary speed-up curves model, the only positive result is a $(1+\epsilon)$-speed $O(\log n)$-competitive algorithm for the unweighted case [38]. This result is complemented by a lower bound showing that no algorithm can be s-speed $o(\log n)$-competitive for any constant resource augmentation $s > 0$. Note that this result is quite surprising. Firstly, in the sequential jobs case, the simple FIFO algorithm achieves constant competitiveness without any resource augmentation. Secondly, maximum flow time is typically viewed as an easier objective to optimize compared to average flow time. Yet in the speed-up curves setting, average flow time admits a scalable algorithm, LAPS, yet maximum flow time does not admit any scalable algorithms.

Note that all of these results for parallelizable jobs on flow time are in the speed-up curves model. The speed-up curves model is theoretically elegant to analyze. However, the DAG model is well suited for programs written using parallel programming languages. It is well connected to practice and it is important to understand scheduling in this model as well.

2.3 Results

In chapter 3 we study the problem of minimizing the average flow time of a set of DAG jobs. Our work is the first theoretical work to provide an algorithm in this model of parallelism for the average flow time objective. There are two principal results described within the chapter.

1. LAPS is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive algorithm for minimizing the average flow time. This is a scalable algorithm that is non-clairvoyant. There is some similarity between this result and the corresponding result in the speed-up curves model, which is the other well known model of parallelism.

2. Smallest-Work-First (SWF) is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive algorithm for minimizing the average flow time. Unlike LAPS, this can be seen as a simple greedy algorithm since it always simply executes the job with the smallest original work. Interestingly, no simple greedy algorithm works for average flow in the speed-up curves model.

3. Round-Robin is a $(2+\epsilon)$-speed $O(1)$-competitive algorithm for minimizing the average flow time. This is not a principal result of our work on average flow time since it can be seen as a corollary of the first result about LAPS. Round-Robin is a very similar algorithm for which a similar style of analysis will yield the result.

These first results open the way for further work on other important objectives in the DAG model. Note, however, that the algorithms described in this chapter are chiefly theoretical algorithms. There is substantial difficulty in implementing algorithms such as LAPS in practice. Overcoming these practical difficulties is a theme which will permeate throughout the rest of the work described in the thesis.

In chapter 4 we study the problem of minimizing the maximum flow time of a set of DAG jobs. This is another objective which has not been studied in the DAG model of parallelism and we give an algorithm with a good theoretical guarantee for the problem. We also study the problem of weighted maximum flow time and give a strong theoretical result.

1. FIFO is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive algorithm for minimizing the maximum flow time of a set of DAG jobs.
2. Biggest-Weight-Fist (BWF) is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive algorithm for minimizing the weighted maximum flow time of a set of DAG jobs.

Note that both of these results are quite interesting theoretically when compared to results in the speed-up curves model of parallelism, where there is a lower bond stating that there is no algorithm with constant speed augmentation which can be $o(\log n)$-competitive. Here we have a scalable algorithm in the DAG model.

In the same chapter 4 we also examine more practical schedulers for maximum flow time. Though FIFO is a good algorithm in theory, there are aspects of FIFO which makes it inefficient to implement. We will elaborate on these difficulties within the chapter. To develop practical algorithms for maximum flow time we will draw inspiration from the work-stealing scheduler which was described in section 2.1. This allows use to arrive at several useful results.

- *Admit-first* work-stealing is a scalable scheduling algorithm for *reasonable jobs*. Specifically, admit-first with $(1+\epsilon)$-speed has maximum flow time $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$ over n jobs for any fixed $\epsilon > 0$ with high probability. Note that if any job has span $\Omega(\lg n)$ or work $\Omega(m \lg n)$, then $\mathrm{OPT} \geq \ln n$. Therefore admit-first is scalable with $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive with high probability.

- There is a lower bound on the competitive ratio of work-stealing which is $\Omega(\lg n)$. Specifically, if all jobs are tiny with work $o(\lg n)$, then work stealing cannot be scalable due to the randomization involved. This effectively means that our result for admit-first work-stealing is tight.

These results are notable because they involve scheduling algorithms designed to be similar to those used in practice. In particular, it is possible to test these algorithms in practice and compared the performance to the theoretically best algorithms (such as FIFO) which can only be simulated. We perform and discuss these experiments in the chapter.

In the spirit of discovering algorithms which can work in practice, we note that the first algorithms given for average flow time in the DAG model were LAPS and SJF. Neither of these algorithms could be easily implemented. In chapter 5 we specifically try to design an algorithm for average flow time that once again incorporates practical ideas from randomized work-stealing. Specifically, we describe a randomized scheduler that tries to limit the preemption overheads which LAPS would induce.

- Distributed Random Equi-Partition (DREP) using work-stealing is a $(4+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive algorithm for average flow time.

This algorithm has worse theoretical performance than the best algorithms for the problem; LAPS only needs $(1+\epsilon)$-speed to be $(\frac{1}{\epsilon^3})$-competitive. However, because DREP is designed with practice in mind, we are able to implement the algorithm for empirical testing. We discuss the strong experimental results that DREP yields when compared to the better theoretical algorithms. It has performance similar to simulations of LAPS and SJF which do not include the scheduling overhead of those algorithms.

Up to this point we will have examined the main flow time objectives in the DAG model. We have given essentially the best theoretical results possible for these flow time objectives and also gave several practical algorithms for the same problems. Though admittedly there are definitely a few other objectives which remain open such as the L_k-norms of flow time.

In chapter 6, we shift our focus and examine the throughput problem. This problem is very different from the flow time objective problems. Every DAG job has an associated deadline d_i. Jobs also have a profit p_i, which is the profit the scheduler receives when the job is completed by their deadline. The goal of the scheduler is to complete jobs and maximize the profit of the schedule. For this problem with DAG jobs we are able to show both a lower bound and a matching upper bound.

- Any semi-nonclairvoyant scheduler requires at least $(2-1/m)$ speed augmentation to be competitive for maximizing throughput.
- There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.

We also give results for a generalization of this problem known as the general profit problem. Unfortunately that problem is difficult to define here without introducing an excessive amount of notation so it will be explained in detail in chapter 6. The results in the chapter essentially give the complete picture on the throughput and general profit problems.

Chapter 3

Average Flow Time

3.1 Introduction

In this chapter we considering minimizing the average flow time in the DAG scheduling model. The most natural algorithm 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 [43] showed that in the hybrid model of parallelism, where jobs consist of a DAG and every node has it own speed-up curve, Robin Robin style algorithms must have a competitive ratio that depends on $\log \kappa$ even if they are given any $O(1)$ speed augmentation. We show that LAPS is a scalable algorithm in the DAG model of parallelism. Hence, the hybrid model in [43] is strictly harder than the DAG model.

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

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

Corollary 3.1.1. *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 non-clairvoyant 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, the LAPS algorithm requires a set of jobs to receive equal processing time, which is hard to achieve in practice with low overheads. 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 $\epsilon > 0$. In the speed-up curve model it is an intriguing open question whether an algorithm exists which is *universally* scalable. That is, an algorithm which is $O(1)$-competitive given any speed $(1+\epsilon)$ where the algorithm does not use knowledge of ϵ.

In practice, the most widely used algorithms are simple greedy algorithms. They are easy to implement. However, no greedy algorithms are known to perform well in the speed-up curves model; simple adaptations of SJF and SRPT perform poorly.

We will also consider a natural adaptation of SJF to the DAG model and show the following theorem.

Theorem 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 will extend the technique of fractional flow time to the DAG model. It is not obvious how to convert an algorithm that is competitive for fractional flow to one that is competitive for total flow time. This is the most challenging part of the analysis; this is where we must use the 2 speed. Note that SJF is the first greedy algorithm shown to perform well for parallelizable jobs in the online setting.

We will begin by first provide some additional notation and observations necessary for this chapter. We will also briefly summarize the technique of potential function analysis which will be used in the analysis these algorithms. Then we will describe the result with the LAPS algorithms followed by that of the SJF algorithm.

3.2 Preliminaries

In this problem, 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 of the jobs. Any set of ready nodes can be processed at once, but each processor can only execute a single 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 J_i's DAG. The critical path length C_i is the length of the longest path in job J_i's DAG, where the length of the path is the sum of the processing time of nodes on the path. We first 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 processors, where $n \leq m$, then the remaining critical-path length of i decreases at a* *rate of s. In other words, during 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 decreases by s.*

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

3.2.1 Additional notation

We will use A to specify the algorithm being considered unless otherwise noted and use $W_i^A(t)$ to denote the remaining processing time of all the nodes in job J_i's DAG at time t in A's schedule. Similarly let $C_i^A(t)$ be the remaining length of the longest path in J_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 not yet completed in A's schedule at time t.

For the quantities that were just describted, we will replace A with O to denote the same quantity in some optimal solution. Also note that $\int_{t=0}^{\infty} |A(t)|$ is exactly the total flow time, which is 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.

3.2.2 Potential Function Analysis:

In this chapter we will utilize the potential function framework, also known as amortized analysis. In 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. The quantity $\frac{\mathrm{d}G_a(t)}{\mathtt{dt}}$ 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, that is, $\frac{\mathrm{d}G_a(t)}{\mathtt{dt}} = |A(t)|$. Showing the following conditions about the potential function is sufficient for proving that the algorithm is competitive.

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 is completed,

$$\frac{\mathrm{d}G_a(t)}{\mathtt{dt}} + \frac{\mathrm{d}\Phi(t)}{\mathtt{dt}} \leq c \cdot \frac{\mathrm{d}G_o(t)}{\mathtt{dt}} \tag{3.1}$$

Notice that integrating these conditions over time one results in $G_a - \Phi(0) + \Phi(\infty) \leq (\alpha + \beta + c) \cdot \mathrm{OPT}$ by the boundary, arrival and completion conditions. This shows the algorithm is $(\alpha + \beta + c)$-competitive.

3.3 LAPS in the DAG Model

In this section, we analyze the LAPS (Latest-Arrival-Processor-Sharing) scheduling algorithm for the DAG model. LAPS is a generalization of round robin. Round robin splits the processing power evenly among all jobs. In contrast, 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; this is the same constant used for the resource augmentation.

We will use $A(t)$ to denote the set of unsatisfied jobs in LAPS's queue at time t. Let ϵ be a fixed constant which will be used in the algorithm and let $0 < \epsilon < \frac{1}{10}$. We then define $A'(t)$ to be the set of $\epsilon|A(t)|$ jobs from $A(t)$ which arrived the latest. Specifically, the last arriving jobs by count.

The algorithm of LAPS is the following: Each DAG 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 nodes on the processors it receives. If the job does not have $\frac{m}{|A'(t)|}$ ready nodes, it schedules as many tasks as possible and idles the remaining alloted processors. This algorithm is summarized in algorithm 1.

Algorithm 1 The LAPS algorithm

```
1: Examine the current alive jobs $A(t)$
2: Find the set $A'(t)$ of latest $\epsilon|A(t)|$ jobs
3: Schedule jobs in $A'(t)$ equally
```

For the analysis we assume that the LAPS is given $1 + 10\epsilon$ resource augmentation. As mentioned in Section 3.2, $W_i^A(t)$ and $C_i^A(t)$ denote the aggregate remaining work and critical path length, respectively, of job J_i at time t in the LAPS's schedule. $W_i^O(t)$ is the aggregate remaining work of job J_i in the optimal schedule at time t. Now we can 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 J_i. The variable $Z_i(t)$ is the total amount of work job J_i has fallen behind in the LAPS's schedule as compared to the optimal schedule at time t. At a high level, this is how far the algorithm is lagging behind the optimal schedule for J_i. Finally, we define $\text{rank}_i(t) = \sum_{j \in A(t), r_j \le r_i} 1$ of job i to be the number of jobs in $A(t)$ that arrived before job J_i, including itself. Without loss of generality, we assume each job arrives at a distinct time.

Now we will define our potential function.

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

The following proposition follows directly from the definition of the potential function since there are no jobs in $A(t)$ at time 0 and at time ∞.

Proposition 3.3.1. $\Phi(0) = \Phi(\infty) = 0$.

We will first show that the increase in the potential function is bounded by OPT over the arrival and completion of all jobs.

Lemma 3.3.2. *The potential function never increases due to job completion by the LAPS or the optimal schedule.*

Proof. When the optimal schedule completes a job J_i, there is no change in the potential the amount the algorithm is lagging behind on job J_i does not change, nor are there terms removed from the summation as the summation only concerns LAPS's unfinished jobs.

When LAPS completes a job J_i at time t, a term is removed from the summation. Notice at this point both $Z_i(t) = 0$ and $C_i^A(t) = 0$, since the algorithm has completely processed the job. Therefore, the removal of this term has no effect on the potential.

The other change to the potential caused by the completion of a job is that $\text{rank}_j(t)$ decreases by 1 for all jobs $J_j \in A(t)$ where $r_j > r_i$. However, $Z_j(t)$ is always positive by definition, decreasing the rank can only decrease potential. $\square$

Lemma 3.3.3. *The potential function increases by at most $O(\frac{1}{\epsilon^3})\text{OPT}$ over the arrival of the jobs.*

Proof. When job J_i arrives at time t, it does not affect the rank of any other job since it is the latest arriving job. Furthermore, by definition $Z_i(t)$ is 0 when job i arrives, since both neither LAPS nor OPT have worked on J_i yet. 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 over all jobs is $\frac{1000}{\epsilon^3}\sum_{i\in[n]} C_i$. However, we know that each job i must wait at least C_i time units to be satisfied in OPT by Observation 2, so this quantity is at most $O(\frac{1}{\epsilon^3})\text{OPT}$. $\square$

The remaining lemmas bound the change in the potential due to the processing of jobs by OPT and LAPS. We will 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 3.3.4. *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. The variables $C_i^A(t)$ do not change due to OPT. The change in the potential due to the optimal schedule will be due to the change in $Z_i(t)$ for some jobs J_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 increase $Z_i(t)$ at a rate of m. This is the worst case because the rank of job i' is the largest. Thus, processing this job changes the potential the most. The total increase in potential is then $\frac{10}{\epsilon}\frac{1}{m}\operatorname{rank}_{i'}(t)m = \frac{10}{\epsilon}\operatorname{rank}_{i'}(t)$. Knowing that $\operatorname{rank}_{i'}(t) = |A(t)|$, we can say that the change in the potential can be at most $\frac{10}{\epsilon}\operatorname{rank}_{i'}(t) = \frac{10}{\epsilon}|A(t)|$. □

Now we calculate the effect of LAPS processing the jobs.

Lemma 3.3.5. *At any time t, the potential function changes by at most (more negative than) $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$ due the processing of jobs by LAPS.*

Proof. First note that we are trying to show that LAPS causes a sufficiently negative change in the potential to offset the processing of OPT. Therefore, we want to show that the change is *at most* some negative amount. Though it can be more negative than this amount.

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 at 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, 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 for this case as C_1. Looking at the change in the $C_i^A(t)$ we have that:

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

Note that $|A_c(t)| \geq \frac{\epsilon}{10}|A'(t)|$ because of we are in this case. So we have

$$C_1 = -\frac{1000}{\epsilon^3}(1 + 10\epsilon)|A_c(t)| \leq -\frac{100}{\epsilon^2}(1 + 10\epsilon)|A'(t)|$$

Finally, because $|A'(t)| = \epsilon|A(t)|$ by definition, we have

$$C_1 \leq -\frac{100}{\epsilon}(1 + 10\epsilon)|A(t)| \leq -\frac{10}{\epsilon}(1 + \epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$$

This completes the first case of the proof.

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. Note that $|A_w(t)| \geq (1 - \frac{\epsilon}{10})|A'(t)|$.

In this case, we ignore the decrease in the C variables (which we used for the previous case) and instead focus on the change in the Z variables due to LAPS's processing. We further ignore the decrease in the $Z_i(t)$ for jobs in $A_w(t) \cap O(t)$. This is due to it being difficult to accurately account for OPT's processing of those jobs. Hence we will only consider the jobs that the optimal schedule cannot process since it has already completed them.

For every job i in $A_w(t) \setminus O(t)$, $Z_i(t)$ 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 due to LAPS
2. LAPS has $(1+10\epsilon)$ resource augmentation
3. OPT already completed job i by time t if job i is in $A_w(t) \setminus O(t)$, therefore it cannot affect $Z_i(t)$.

We denote the total change in the potential due to LAPS in this case as C_2.

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

The last step above is due to $\operatorname{rank}_i(t) \geq (1-\epsilon)|A(t)|$ for $i \in A'(t)$ because of the scheduling policy of LAPS. Also, by definition $|A'(t)| = \epsilon|A(t)|$, thus we can further simplify the above expression to the following ones below.

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

Note that these sums only depend on the number of jobs in $A_w(t)$. We also know that $|A_w(t)| \geq (1 - \frac{\epsilon}{10})|A'(t)|$ by the case's definition. And by replacing $|A'(t)|$ with $\epsilon|A(t)|$, we can further simplify to the following ones below.

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

Finally, because $\epsilon < 1/10$, we can derive the following expressions.

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

This completes the second case of the lemma.

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 3.3.6. *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 the algorithm and the optimal schedule.*

Proof. Now we know from Lemma 3.3.4 the change due to OPT processing jobs is at most $\frac{10}{\epsilon}|A(t)|$. We will aggregate this with the change in potential due to LAPS. Combining lemma 3.3.4 and 3.3.5, we see that the aggregate change in the potential is at most the following expression.

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

□

Thus, by the potential function framework and combining Lemma 3.3.2, 3.3.3 and 3.3.6 and Proposition 3.3.1 we have Theorem 1.

3.4 SJF in the DAG Model

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 job with the smallest work have the highest priority. The algorithm takes the highest priority job and assigns all of its ready nodes to machines. Then it 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. This algorithm is summarized in

Algorithm 2 The SJF algorithm

```
1: Identify the current alive jobs $A(t)$
2: Sort the current alive jobs $A(t)$ by their smallest original work in non-increasing order
3: Let $J_i$ be the first job in the ordering
4: Assign free processors to $J_i$ until all $J_i$'s ready nodes are being processed
5: Repeat the process with the next smallest job, until either no more jobs or free processors
```

Note that interestingly, this scheduling strategies does not take the critical path length into consideration at all. One might intuitive also consider scheduling jobs which have high critical path length as they will take a long time to complete. However, as the analysis shows, prioritizing based on work without taking the critical path length into account is the way that achieves good theoretical performance.

3.4.1 Analysis of SJF for Fractional Flow Time

We use fractional flow time for this analysis. To avoid confusion, we will refer to total flow time as *integral flow time* — recall that a job contributes 1 to the objective during each time unit the job is alive and unfinished. In contrast, in fractional flow time, jobs contributes the fraction of the *work* which remains for the job. Then, the goal is to minimize $\sum_{t=0}^{\infty} \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i}$. Note that since our algorithm prioritizes based on just work, our fractional flow time is also defined based just on work.

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.

Through this we will prove the competitiveness of SJF's integral flow time to that of the optimal algorithm's integral flow time.

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. We will utilize a potential function analysis. Recall again that, $W_i^A(t)$ and $C_i^A(t)$ denote the aggregate remaining work and critical path length, respectively, of job J_i at time t in the algorithm's schedule. $W_i^O(t)$ is the aggregate remaining work of job J_i in the optimal schedule at time t. The variable $\overline{W}_j(t)$ in the potential is defined as $\overline{W}_j(t) := \max\{W_j^A(t) - W_j^O(t), 0\}$. This variable is the total amount of work job J_j has fallen behind in SJF's schedule. The potential function itself is the following expression.

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

Note that there are two main terms in this potential. One term is just a summation of remaining critical path lengths. Another is a summation of fractional flow times. These terms will often be referred to as the first term and the second term in the potential, respectively. We will show the following theorem with using this potential function.

Theorem 3. *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 by definition, $\Phi(0) = \Phi(\infty) = 0$ since there are no jobs at those times; the boundary condition holds. We will now show the arrival and completion conditions.

Lemma 3.4.1. *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 arrivals of jobs. Suppose job J'_j arrives at a time t', then in the first term of the potential, a new term is created in the summation with the value $\frac{1}{\epsilon}C_{j'}$. This is less than $\frac{1}{\epsilon}$ times the flow time of this job in an optimal schedule because C_i is a lower bound on a job's integral flow time (Observation 2). So, the increase in potential over all job arrivals, only accounting for the first term, is at most $\frac{1}{\epsilon}\text{OPT}$.

Now consider the second term of $\Phi(t')$ when job J'_j arrives. The quantity $\overline{W}_{j'}(t') = 0$, because OPT has not worked on job j' yet so there is no difference in the amount of work remaining for SJF or OPT; though J'_j will cause a new quantity to appear in the outer summation of the second term of the potential, this quantity is 0.

Finally, J'_j may appear as a new quantity in the inner summation for all jobs $i \in A(t')$ with $W_i > W_{j'}$. However then $W^A_{j'}(t') - W^O_{j'}(t') = 0$ because once again neither SJF or OPT has worked on J'_j yet. Therefore, when job J'_j arrives there is no change in the second term of the potential.

Hence, the arrival condition holds. Next we will consider when job are completed.

When the optimal schedule completes some job J'_j at time t'. The only effect on the potential is that some quantity may be removed from the inner summation of the second term if J'_j is no longer in $A(t') \cup O(t')$. This is the only place where the potential considers the set of jobs OPT is still working on. However, note that the only way for such a quantity to be removed from the summation is if the job is also not in $A(t')$. If the job is not in at this time and OPT just completed it, then clearly $W^A_{j'}(t') - W^O_{j'}(t') = 0$; there is no work remaining for either SJF or OPT on this job. Therefore, there is no change to the potential due to the removal of this quantity.

Now consider when SJF completes some job J'_j at time t'. Because the job has been completed, $C^A_{j'}(t') = 0$ and $\overline{W}_{j'}(t') = 0$. Thus, removing quantities from the either the first term 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 both SJF and OPT must have completed this job. Thus, similar to the early case, $W^A_{j'}(t') - W^O_{j'}(t') = 0$. Therefore there is no change in the potential.

Overall, we can see that there is no change in the potential due to jobs being completed by either the algorithm or the optimal schedule. □

We have shown the boundary conditions as well as the bounded the changes in Φ due to the arrival and completion of jobs. 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. Let us fix some specific time t. Our goal now is to bound $\frac{d\Phi(t)}{dt}$. We will begin by considering the optimal schedule's processing of jobs, then we will examine the processing of the algorithm, finally we will combine the two.

Lemma 3.4.2. *The total change in $\Phi(t)$ at time t due to the optimal schedule processing jobs is $O(|O(t)|) + \frac{1}{\epsilon}\sum_{i \in A(t)} \frac{W^A_i(t)}{W_i}$.*

Proof. Observe that the only changes in the potential which may occur due to the optimal schedule processing a job is due to changes in the quantities $W^O_i(t)$ and $\overline{W}_j(t)$, both of which are in the second term of $\Phi(t)$.

Let us consider a particular job J_i which OPT processes at time t and suppose that OPT uses m'_i processors to process this job. We will later take a sum over all possible jobs that OPT processes.

First, consider the change in $\Phi(t)$ due to $W_i^O(t)$ decreasing. This quantity is within the inner summation of the second term of the potential and only exists if the other job, job $J_j \in A(t)$, has the property that $W_i \leq W_j$. In such a case, the processing of the optimal schedule increases the quantity $W_i^A(t) - W_i^O(t)$ since it reduces the quantity of $W_i^O(t)$. Each machine in OPT has 1 speed and all work values for jobs are distinct, so the change in potential is the following.

$$\frac{1}{\epsilon}\frac{m'_i}{m}\frac{\overline{W}_i(t)}{W_i} + \frac{1}{\epsilon}\frac{m'_i}{m}\sum_{\substack{j\in A(t)\\ W_i<W_j}}\frac{\overline{W}_j(t)}{W_j} = \frac{1}{\epsilon}\frac{m'_i}{m}\sum_{\substack{j\in A(t)\\ W_i\leq W_j}}\frac{\overline{W}_j(t)}{W_j}$$

There are two parts to this change in potential. The first is for job j itself, which is weighted by the factor $\frac{\overline{W}_i(t)}{W_i}$. The second part accounts for all the other affected jobs which cause various different changes depending on their own $\frac{\overline{W}_j(t)}{W_j}$. Since we assumed that all jobs had distinct work, we can absorb the expression into one sum.

Also, because $\frac{\overline{W}_j(t)}{W_j} \leq \frac{W_j^A(t)}{W_j}$ by definition of $\overline{W}_j(t)$, we can simplify to the following expression, which gives the amount of change caused by the change in value of $W_i^O(t)$ when OPT uses m'_i processors on job J_i.

$$\frac{1}{\epsilon}\frac{m_i'}{m}\sum_{\substack{j\in A(t)\\ W_i\le W_j}}\frac{\overline{W}_j(t)}{W_j}\le\frac{1}{\epsilon}\frac{m_i'}{m}\sum_{\substack{j\in A(t)\\ W_i\le W_j}}\frac{W_j^A(t)}{W_j}$$

Now we will consider the change in potential caused by the change in the value of $\overline{W}_i(t)$ by OPT's processing of job J_i. This variable could, in the worst case, increase at a rate of m_i'. This will be multiplied by all the inner summation terms which include all other jobs J_j where $W_j \le W_i$. Note that the job whose processing we are considering is job J_i and all the other jobs are jobs $J_j \in A(t)$. In multiplying by the inner summation we will omit the $-W_j^O(t)$ part of the inner summation since that part only decreases the potential. We are interested in how much OPT can increase the potential in the worst case so we can omit these decreasing quantities which are difficult to bound. We can then compute the change in potential due to the change in the value of $\overline{W}_i(t)$ as the following.

$$\frac{1}{\epsilon}\frac{m_i'}{m}\frac{W_i^A(t)}{W_i}+\frac{1}{\epsilon}\frac{m_i'}{mW_i}\sum_{\substack{j\in A(t)\\ W_j<W_i}}W_j^A(t)$$

By definition, $\frac{W_i^A(t)}{W_i}\le 1$. Additionally, in the summation for all the W_js we have $W_j < W_i$, therefore we can switch out all the W_i for W_j and place it inside the summation. Then the change in potential due to the change in the value of $\overline{W}_i(t)$ can be simplified to the following.

$$\frac{1}{\epsilon}\frac{m_i'}{m} + \frac{1}{\epsilon}\frac{m_i'}{mW_i}\sum_{\substack{j\in A(t)\\ W_j<W_i}} W_j^A(t) \leq \frac{1}{\epsilon}\frac{m_i'}{m} + \frac{1}{\epsilon}\frac{m_i'}{m}\sum_{\substack{j\in A(t)\\ W_j<W_i}} \frac{W_j^A(t)}{W_j}$$

Now, we combine the two changes due to changing the values of $W_i^O(t)$ and $\overline{W}_i(t)$ when OPT processes J_i into the following expression.

$$\left(\frac{1}{\epsilon}\frac{m_i'}{m}\sum_{\substack{j\in A(t)\\ W_i\leq W_j}} \frac{W_j^A(t)}{W_j}\right) + \left(\frac{1}{\epsilon}\frac{m_i'}{m} + \frac{1}{\epsilon}\frac{m_i'}{m}\sum_{\substack{j\in A(t)\\ W_j<W_i}} \frac{W_j^A(t)}{W_j}\right) = \frac{1}{\epsilon}\frac{m_i'}{m} + \frac{1}{\epsilon}\frac{m_i'}{m}\sum_{j\in A(t)} \frac{W_j^A(t)}{W_j}$$

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, thus $\sum_{i\in P^O(t)} m_i' \leq m$. Knowing this, we sum over all the jobs to arrive at the total change in potential due to OPT processing jobs.

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

Finally we know that OPT must have at least one alive job for it to process any jobs. Thus, the first part of this expression, $\frac{1}{\epsilon}$, is $O(|O(t)|)$. With this, we have proven the lemma that the change in potential by OPT processing jobs is no more than $O(|O(t)|) + \frac{1}{\epsilon}\sum_{i\in A(t)} \frac{W_i^A(t)}{W_i}$. $\square$

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

Lemma 3.4.3. *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_j, we know that either the algorithm is processing jobs $J_i \in A(t)$ 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 assigned all processors to higher priority jobs or it is scheduling all available ready nodes for job J_j, where in the second case by observation 1 we know that the remaining critical path of the job is decreasing.

Suppose that the algorithm decreases the critical path of J_j. Then, this decreases the value of $C_j^A(t)$ at a rate of $-(1+\epsilon)$. Alternatively, say the algorithm assigned all processors to jobs with higher priority than J_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 those jobs and decreasing their remaining work. In the second case here, the change in potential is $-\frac{1+\epsilon}{\epsilon}\frac{\overline{W}_j(t)}{w_j}$.

Now we consider all jobs i in the potential. The change in the potential function due to the change in $W_i^A(t)$ and $C_i^A(t)$ over all jobs the algorithm processes can be bounded by the $-\frac{(1+\epsilon)}{\epsilon}\sum_{i \in A(t)} \frac{\overline{W}_i(t)}{W_i}$. Note that since $\frac{\overline{W}_i(t)}{W_i} \leq 1$ by definition, decreasing the critical path will cause a more negative value than decreasing the work of all higher priority jobs. We lose some of the decrease in potential with this expression, but it is still sufficient.

Consider the jobs in the summation, if $J_i \notin O(t)$ then it is the case that $\overline{W}_i(t) = W_i^A(t)$, but if $J_i \in O(t)$ then in the worst case $\overline{W}_i(t) = 0$ and we do not allow this quantity to become negative, so we cannot say that the algorithm is decreasing the potential due to decreasing this term. Nevertheless dropping all $J_i \in O(t)$ the decrease in the potential can still be bounded by the following expression.

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

The only other change that can occur is that when the algorithm causes $\overline{W}_j(t)$ to decrease for jobs J_j that the algorithm processes. It could then be multiplied by a $-W_i^O(t)$ because the second summation in the second term is allowed to be negative. This will actually cause an increase in the potential function. Suppose the algorithm processes job j using m'_j processors at time t. We will let $P^A(t)$ denote 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 the following:

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

Note that assuming that $0<\epsilon\le 1$ is a constant. And therefore this is $O(|O(t)|)$ Taking the combination of these two parts results in the following expression.

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

This is exactly the statement of the lemma. □

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

Proof of [Theorem 3]

The total change in the potential due to the algorithm and optimal schedule processing jobs can be computed from combining Lemmas 3.4.3 and 3.4.2. Note that we are summing over the terms, some of which are negative due to decreasing the potential.

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

Where, we arrived at the final result by noticing that the second term in the penultimate step can be simplified like this:

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

So, 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 $O(\frac{1}{\epsilon}\mathrm{OPT})$. This completes the proof of the continuous change in the potential. The overall competitiveness of the algorithm then follows due to this, Lemma 3.4.1, and the boundary conditions. □

3.4.2 From Fractional Flow to Integral Flow

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 3 proves Theorem 2.

Lemma 3.4.4. *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 $0 < \epsilon \le 1/2$.*

To show Lemma 3.4.4, we will consider two schedules created by SJF. One schedule uses s speed and the other $(2+\epsilon)s$ for some fixed $0 < \epsilon \le 1/2$ 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 nodes 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 actually not straightforward in the DAG model. In fact, in Section 3.4.3, we will actually show that if the faster schedule has less than a $(2-\frac{2}{m})$ factor more speed than the slower schedule 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. So, we must show that F does not fall behind S given a factor of $(2+\epsilon)$ more speed.

First, we will give some additioal notation specific to this section. Let $\mathcal{S}(t)$ ($\mathcal{F}(t)$) denote the queued jobs in S's (or 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. We require the 2 speed in the conversion from fractional to integral flow time in this particular lemma.

Lemma 3.4.5. *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, assume the lemma is not true and let t be the first time that it is false for some ρ. Then at this time t, there must be some job J_i where $W_i^S(t) < W_i^F(t)$ and $W_i \leq \rho$.

When J_i is released at time r_i the lemma 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 the total volume of jobs in S's schedule with original size at most ρ is at least the following:

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

Consider the time interval $[r_i, t]$. Note 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 $C_i - C_i^S(t)$ using 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

$$t - r_i \geq (C_i - C_i^S(t))/s \geq (C_i - C_i^F(t))/s \tag{3.2}$$

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 while not executing nodes of jobs with $W_i \leq \rho$. Otherwise, by Observation 1 F would have decreased the critical-path of job J_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 J_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, where we invoked equation 3.2 for one of the steps.

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

Note that this contradicts the definition of t as a time where F did less work than S. $\square$

Now let $t_{i,\epsilon}^S$ denote the latest time t in S's schedule where $\frac{W_i^S(t)}{W_i}\ge\epsilon$. For the fractional flow time objective, job J_i always incurs a cost of at least ϵ at each moment in time during $[r_i,t_{i,\epsilon}^S]$ in S's schedule. Let $f_{i,\epsilon}^S=t_{i,\epsilon}^S-r_i$. This means that job J_i's fractional flow time is greater than $\epsilon f_{i,\epsilon}^S$ in S. In the case of integral flow time we know that a job pays a cost of 1 each time step that it is incomplete. Thus, if the integral flow time of job i in F is bounded by $f_{i,\epsilon}^S$ we can charge this job's integral cost in F to the job's fractional cost in S. Alternatively, according to Observation 2, for integral flow time the optimal schedule of speed 1 must make job J_i wait C_i time steps. Thus, if job J_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 instead. 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 3.4.6. *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 $\max\{f^S_{i,\epsilon}, C_i\} = f^S_{i,\epsilon}$. 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 $\max\{f^S_{i,\epsilon}, C_i\} = f^S_{i,\epsilon}$ 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 $\max\{f^S_{i,\epsilon}, C_i\} = C_i$. 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 $O(\frac{1}{\epsilon^2})\mathrm{IntCost}(\mathrm{OPT}^I)$. $\square$

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 3.4.7. *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 3.4.8. *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 3.4.7 we have the following.

$$\begin{aligned}
|L^F(t)| &= \sum_{h=1}^{\infty} |\mathcal{F}_{=h}(t) \cap L^F(t)| \\
&\leq \sum_{h=1}^{\infty} \frac{10}{\epsilon} \sum_{h'=1}^{h} \frac{1}{2^h} W^S_{=h'}(t) \quad \text{[By Lemma 3.4.7]} \\
&= \sum_{h=1}^{\infty} \frac{10}{\epsilon} \sum_{h'=1}^{h} (\frac{1}{2^{h'}} W^S_{=h'}(t)) \frac{1}{2^{h-h'}} \\
&= \frac{10}{\epsilon} \sum_{h'=1}^{\infty} (\frac{1}{2^{h'}} W^S_{=h'}(t)) \sum_{h=h'}^{\infty} \frac{1}{2^{h-h'}} \\
&\leq \frac{20}{\epsilon} \sum_{h'=1}^{\infty} \frac{1}{2^{h'}} W^S_{=h'}(t) \\
&\leq \frac{20}{\epsilon} \sum_{i \in \mathcal{S}(t)} \frac{W_i^S(t)}{W_i} \quad [2^{h'-1} \leq W_i < 2^{h'} \text{ if } i \text{ in class } h']
\end{aligned}$$

□

The previous lemma with Lemma 3.4.6 implies Lemma 3.4.4. All that remains is to prove Lemma 3.4.7.

Proof of [Lemma 3.4.7]

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. $|\mathcal{F}_{=h}(t) \cap L^F(t)| > \frac{10}{\epsilon} \sum_{i \in \mathcal{S}(t), W_i \leq 2^h} \frac{1}{2^h} W_i^S(t)$.

Let j^* denote the job in $L^F(t)$ which arrived the earliest; j^* is of some class $h' \leq h$. By definition of $L^F(t)$, this implies that S processed at least $(1-\epsilon)W_i$ work 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 because of observation 1 and thus have been completed by t, which is 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 3.4.5, we have $\sum_{i \in \mathcal{F}(r_{j^*}), W_i \leq 2^h} W_i^F(r_{j^*}) \leq \sum_{i \in \mathcal{S}(r_{j^*}), W_i \leq 2^h} W_i^S(r_{j^*})$. And therefore,

$$
\begin{aligned}
&\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) - \sum_{i \in \mathcal{F}(t), W_i \leq 2^h} W_i^F(t) \\
&\geq \left(\sum_{\substack{i \in \mathcal{S}(r_{j^*}) \\ W_i \leq 2^h}} W_i^S(r_{j^*}) + V - sm(t - r_{j^*}) \right) - \left(\sum_{\substack{i \in \mathcal{F}(r_{j^*}) \\ W_i \leq 2^h}} W_i^F(r_{j^*}) + V - (2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*}) \right) \\
&\geq (-sm(t - r_{j^*})) - \left(-(2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*}) \right) \qquad \text{[Lemma 3.4.5]} \\
&\geq \frac{1+\epsilon}{2} sm(t - r_{j^*}) \qquad [\epsilon \leq 1/2]
\end{aligned}
$$

This implies that

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

We also know that

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

Using the fact that $\epsilon \le 1/2$, we can simplify the expressions.

$$\sum_{i \in \mathcal{S}(t), W_i \le 2^h} W_i^S(t) \ge \frac{1+\epsilon}{2} \sum_{i \in L^F(t), W_i \le 2^h} (1-\epsilon) W_i \ge \frac{\epsilon}{4} \sum_{i \in L^F(t), W_i \le 2^h} W_i$$

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

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

This contradicts the definition of time t, which means we have proven the lemma. □

This completes the proof of theorem 2.

3.4.3 SJF Falls Behind with Resource Augmentation

In this section we present an interesting side-result for the SJF scheduler in the DAG model.

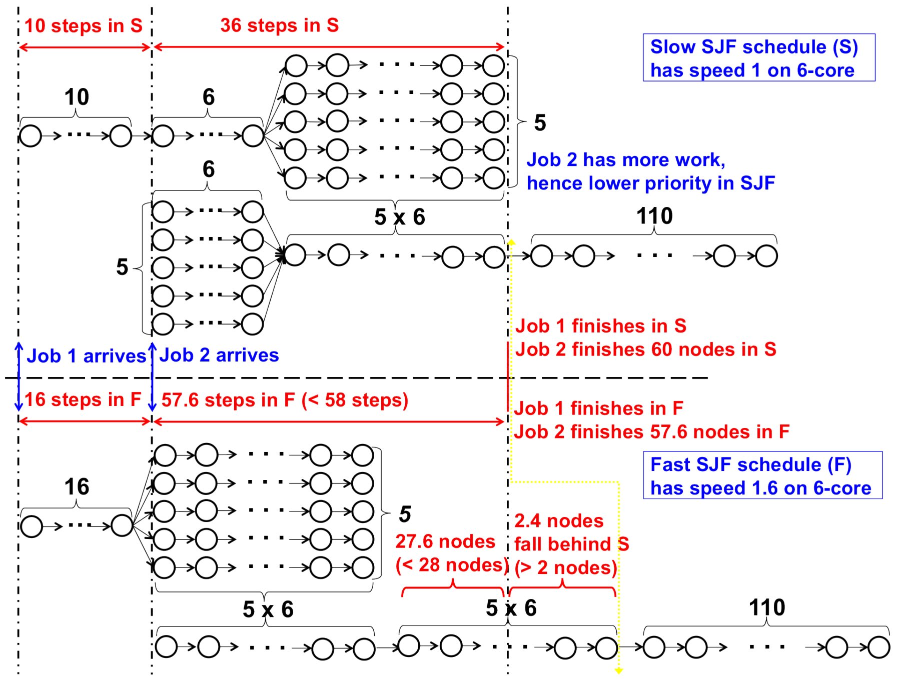

10 steps in S
36 steps in S
Slow SJF schedule (S) has speed 1 on 6-core
10
6
5
Job 2 has more work, hence lower priority in SJF
6
5 x 6
110
5
Job 1 finishes in S
Job 2 finishes 60 nodes in S
Job 1 arrives
Job 2 arrives
16 steps in F
57.6 steps in F (< 58 steps)
Job 1 finishes in F
Job 2 finishes 57.6 nodes in F
Fast SJF schedule (F) has speed 1.6 on 6-core
16
5
2.4 nodes fall behind S (> 2 nodes)
27.6 nodes (< 28 nodes)
5 x 6
5 x 6
110

Figure 3.1: An example schedule of slow and fast SJF on 6 processors

In particular, we show that SJF can fall behind itself in total aggregate work when given more resource augmentation. This is surprising because basically the same scheduling algorithm is used, yet with speed augmentation it is actually possible for the 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.

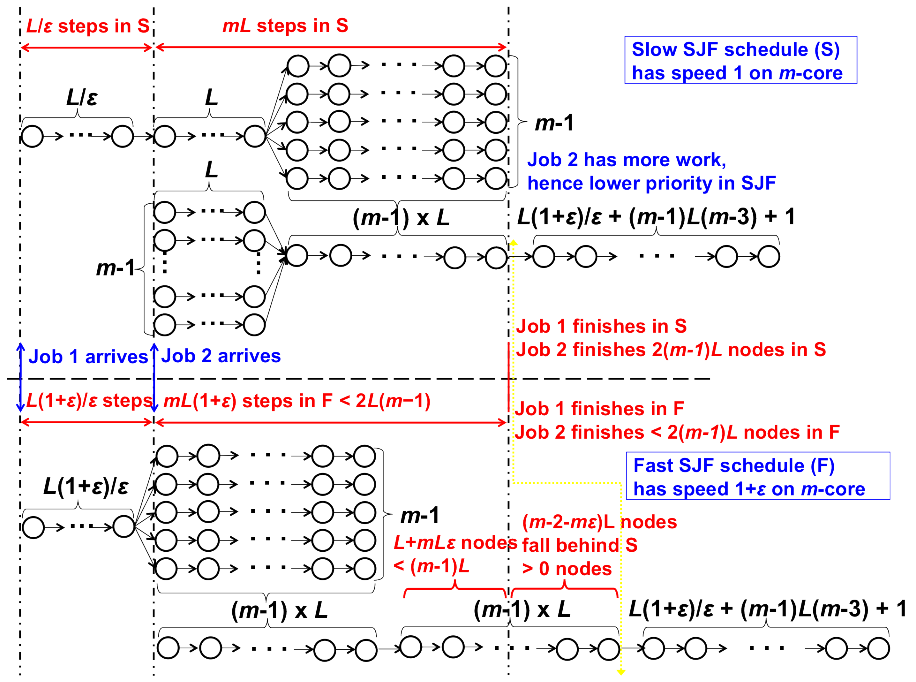
First we shall give a concrete example with 1.6 speed where F does less aggregate work than S does at some time t. Then, we present the general example for any speed $s < 2 - \frac{2}{m}$. Intuitively, we show that the structure of J_1 on the fast schedule forces J_2 to be executed entirely sequentially,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

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 the slow and fast SJF.

The time we consider for the contradiction is at $t = 46$. 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



L/ε steps in S
mL steps in S
Slow SJF schedule (S) has speed 1 on m-core
L/ε
L
L
m-1
Job 2 has more work, hence lower priority in SJF
(m-1) x L
L(1+ε)/ε + (m-1)L(m-3) + 1
m-1
Job 1 finishes in S
Job 2 finishes 2(m-1)L nodes in S
Job 1 arrives
Job 2 arrives
L(1+ε)/ε steps
mL(1+ε) steps in F < 2L(m−1)
Job 1 finishes in F
Job 2 finishes < 2(m-1)L nodes in F
Fast SJF schedule (F) has speed 1+ε on m-core
L(1+ε)/ε
m-1
L+mLε nodes < (m-1)L
(m-2-mε)L nodes fall behind S > 0 nodes
(m-1) x L
(m-1) x L
L(1+ε)/ε + (m-1)L(m-3) + 1

Figure 3.2: An example schedule of slow and fast SJF for m processors.

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. The 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 $t = 46$.

General Case for Speed s on m processors

We now show the general case where a speed of $2 - \frac{2}{m}$ is necessary for the fast schedule to catch up in total aggregate work compared to the slow schedule. 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 3.2, for similarity to the previous example we introduce a constant L. In the previous concrete 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.

3.5 Conclusion

In this chapter we have proved the first results on a flow time objective in the DAG model. Though it may be not too unexpected that algorithms such as LAPS perform well in theory, it is interesting that SJF also works for this problem since there were no greedy algorithms known for average flow time for parallelizable jobs. Certainly, this is an interesting step towards more practical average flow time algorithms in the DAG model. Additionally, the

DAG model can occasionally exhibit very interesting behaviour as we have shown in section 3.4.3 where the same scheduler with more speed can somehow fall behind in aggregate work compared to one with less speed. Nevertheless, with this first result on average flow time in the DAG model, the way is open for optimizing other flow time objective - a task we will actively pursue in the upcoming chapters.

Chapter 4

Maximum Flow Time

In this chapter we consider the problem of minimizing the maximum flow time of a set of jobs in the DAG model. We are scheduling these jobs on m identical machines or processors. These are the first known non-trivial results for maximum flow time in the DAG model. All of the algorithms considered in this chapter are *non-clairvoyant*, meaning that they have no prior knowledge of the size or structure of the jobs or when they arrive.

We will describe several contributions to this area. First, we start with an idealized First-in-First-Out (FIFO) scheduler. At each time step, FIFO looks at jobs in the order of arrival and allocates each job as many processors it can use until it runs out of jobs or processors. For this scheduler, we prove the following theorem in section 4.2.

Theorem 4. *FIFO is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive scheduler for the maximum flow time in the DAG model.*

We then generalize this result to the work stealing scheduler in 4.3. The reason we use work stealing is because it is a practical and efficient scheduler that is used in many parallel languages and libraries. In comparison, an implementation of the ideal FIFO scheduler is likely to have high overhead since it is centralized and potentially preempts jobs to re-allocate processors at every time step.

For work stealing, we prove that a version of it called *admit-first* is scalable for "reasonable jobs". More specifically, we prove the following theorem about admit-first work-stealing.

Theorem 5. *Admit-first work-stealing with* $(1+\epsilon)$*-speed has a maximum flow time of* $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$ *over* n *jobs for any fixed* $\epsilon > 0$ *with high probability.*

Note that if any job has span $\Omega(\lg n)$ or work $\Omega(m \lg n)$, then $\mathrm{OPT} \geq \ln n$ and admit-first is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive with high probability, therefore it would be scalable.

We will also introduce a generalization of the admit-first scheduler called steal-k-first. Our goal in this generalization is to design a work-stealing scheduler that is closest to FIFO since FIFO has strong theoretical performance though it suffers in implementation. Steal-k-first is parameterized by k. Intuitively as k becomes larger, this algorithm becomes closer to the FIFO scheduler. Theoretically, this scheduler is $(k+1+\epsilon))$-speed $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$-competitive for any $\epsilon > 0$ and $k \geq 0$. It reduces to admit-first when $k = 0$.

To complement our algorithmic results on work-stealing, we also provide a lower bound for work-stealing schedulers in 4.4.

Theorem 6. *The competitive ratio of any randomized work-stealing scheduler is* $\Omega(\lg n)$*.*

Note this means if all jobs in the input are tiny jobs with work $o(\lg n)$, then work stealing cannot be scalable due to the randomization involved. This shows that our bounds for work-stealing are essentially tight.

To test these more practical scheduling algorithms, we evaluated them experimentally in section 4.5. We implemented admit-first and steal-k-first in Thread Building Blocks (TBB) and compared their performance with a simulated optimal scheduler on both realistic and synthetic workloads. Experimental results shows that a work stealing scheduler (especially steal-k-first) will have comparable performance to the optimal scheduler.

Finally, in section 4.6 we consider the case where jobs have weights and show that the non-clairvoyant algorithm Biggest-Weight-First (BWF) works well for this problem.

Theorem 7. *BWF is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive algorithm for the weighted maximum flow time in the DAG model of parallelism.*

Due to lower bounds present for weighted maximum flow time in the online setting, this is the best positive result that can be shown.

4.1 Preliminaries

In this online scheduling problem, there are n jobs which arrive over time. These jobs are scheduled on m identical processors. Each job J_i has an arrival (release) time r_i, which is the first time an online scheduler is aware of the job. At any point in time, we will use $A(t)$ to refer to the set of alive jobs in algorithm A's schedule. Alive jobs are jobs which have arrived but have not yet been completed.

In the more general weighted maximum flow problem, each job could have a weight w_i — this weight is known to the scheduler when the job arrives and may not be correlated to the work of the job. For unweighted maximum flow, $w_i = 1$ for all jobs. For the majority of this chapter we will consider the unweighted case. We will only consider the weighted case in section 4.6.

Recall that c_i is the completion time of job J_i in the algorithm's schedule. We use $F_i = c_i - r_i$ as the flow time of job J_i in the algorithm's schedule. The goal of the scheduler is to minimize $\max_{i \in [n]} w_i F_i$.

As in the previous chapter, we are working with Directed-Acyclic-Graph jobs here as well. Recall that each of these jobs J_i corresponds to a DAG G_i. Each node (task) v in G_i has an associated processing time p_v and the node must be processed sequentially on a processor for p_v time to be completed. A node in G_i cannot be executed until all of its predecessors in G_i have been executed. We say that a node is *ready* if all of its predecessors have been processed. Multiple ready nodes for the same job can be scheduled simultaneously. A job is completed once all nodes in its DAG are completely processed. We do not assume that the scheduler knows the DAG in advance.

Two parameters important to the analysis of DAG jobs are the critical path length C_i and the total work W_i. We have previously defined these quantities very early in section 1.2.3. But they will be used frequently in this chapter so we shall briefly mention them again:

- The critical path length of J_i is the length of the longest path in G_i where each node v in the longest path contributes p_v to the length of the path. Note that C_i is a lower bound on the execution time of J_i for any scheduler.

- The work W_i of job J_i is the sum of the processing times of all the nodes in the DAG. $\frac{W_i}{m}$ is a lower bound on the execution time of the job on m processors.

The following proposition will be used throughout this chapter, it states that any time a scheduler is working on all the ready nodes for some job J_j, then the scheduler must be decreasing the remaining critical path of J_j. This proposition has been shown in many previous works including [36].

Proposition 4.1.1. *If during each time step during a time interval $[t', t]$, a scheduler of speed s is always scheduling all available nodes for a job J_j, then the scheduler reduces the critical path length of J_i by $s(t - t')$.*

We are now ready to give and analyze the first algorithms for minimizing the maximum flow time for DAG jobs.

4.2 FIFO for Maximum Flow Time

In this section our goal is to prove theorem 4 which states that the algorithm First-In-First-Out (FIFO) is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for minimizing the maximum unweighted flow time for any $0 < \epsilon < 1$.

The FIFO algorithm is defined as follows. At any time t, FIFO orders the jobs in increasing order by their arrival time, breaking ties arbitrarily. The algorithm then assigns all of the ready nodes for the first job to unique processors, then recursively does the same for the next job in the list. This continues until all processors have been assigned some node or there are no more ready nodes available. The algorithm may have a choice on which ready nodes of a job to schedule if the job has more ready nodes than the number of processors that have not been assigned to a node when the job is considered. In this case, we assume the scheduler chooses an arbitrary set of ready nodes from the job. Algorithm 3 provides an overview of this procedure.

Algorithm 3 The FIFO algorithm

```
1: Examine the current alive jobs, $A(t)$
2: Sort jobs by arrival time in increasing order
3: Execute jobs in order by assigning processors to ready nodes
```

The rest of this section is devoted to proving Theorem 4. We assume for the remainder of this section that FIFO is given $(1+\epsilon)$-speed for some constant $0 < \epsilon < 1$ and we will show that FIFO is $\frac{3}{\epsilon}$ competitive. We will now use proof by contradiction.

Assume for the sake of contradiction that FIFO has a competitive ratio larger than $\frac{3}{\epsilon}$. We consider the instance for which FIFO does not achieve a competitive ratio of $\frac{3}{\epsilon}$. Let job J_i be the job with the maximum flow F_i in this instance in FIFO's schedule. Therefore, OPT $< \frac{\epsilon}{3}F_i$ by assumption. Since no jobs that arrive later than J_i has any effect on how or when J_i is scheduled due to FIFO's scheduling policy, we may assume that J_i is the last job to arrive.

We begin by showing that during the time interval job J_i is alive in FIFO's schedule, the processors must be busy for most of the interval. We define one *time step* as the time period for a s-speed processor to execute one unit of work. In other words, in one time step m processors with speed s can finish m work of jobs. On processors with different speeds, the length of a time step will be different; the number of time steps on a s-speed processor in T units of time is sT, while it is T on a processor with 1 speed. Intuitively, we want each time step to correspond to a unit amount of work being performed by a processor.

Lemma 4.2.1. *During the interval $[r_i, c_i]$ in FIFO's schedule, there can be no more than $\frac{\epsilon}{3}F_i$ time steps where not all m processors are busy working on jobs.*

Proof. For the sake of contradiction, suppose there are at least $\frac{\epsilon}{3}F_i$ time steps during $[r_i, c_i]$ where not all processors are busy.

Consider FIFO's scheduling policy. Anytime during $[r_i, c_i]$ when FIFO is not processing nodes on every processor, FIFO must be scheduling all of the ready nodes of J_i. Due to this, at these times FIFO is working on the critical path length of J_i by Proposition 4.1.1. Let the critical path length of J_i be C_i, then we have $C_i \geq \frac{\epsilon}{3}F_i$.

OPT cannot finish a job in less time than its critical path length, this leads to $\text{OPT} \geq C_i \geq \frac{\epsilon}{3}F_i$, so the competitive ratio is $\frac{F_i}{\text{OPT}} \leq \frac{3}{\epsilon}$, which is a contradiction. □

Lemma 4.2.1 showed that for most of the time steps in $[r_i, c_i]$ FIFO has m processors busy working. In the next lemma, we show that the work done by FIFO during $[r_i, c_i]$ is concentrated on jobs which did not arrive before $r_i - F_i$.

We define *processor idling steps* to be the aggregate number of time steps per processor where the processor is not working on any job. Hence, during a single time step where not all m processors are busy, there can be at most m processor idling steps (as there are m processors).

Lemma 4.2.2. *During $[r_i, c_i]$, FIFO does more than $m(1 + \frac{\epsilon}{3})F_i$ work on jobs which arrived after $r_i - F_i$.*

Proof. Since J_i is the job with the maximum flow time F_i in the schedule, all previous jobs must have had less flow time than F_i. Therefore, all jobs which received any processing during $[r_i, c_i]$ must have arrived no earlier than $r_i - F_i$.

To complete the lemma we calculate the total work done during $[r_i, c_i]$. From Lemma 4.2.1 the number of processor idling steps is at most $m\frac{\epsilon}{3}F_i$ during $[r_i, c_i]$. Since the processors have speed $1+\epsilon$, the total work that is done during $[r_i, c_i]$ is at least

$$m(1+\epsilon)F_i - m\frac{\epsilon}{3}F_i > m(1+\frac{\epsilon}{3})F_i$$

which completes the lemma. □

Using the previous lemmas we can complete the proof.

Proof of Theorem 4 We consider the work of the optimal schedule. OPT achieves a flow time of $\mathrm{OPT} < \frac{\epsilon F_i}{3}$ from the assumption that FIFO does not achieve a competitive ratio of $\frac{3}{\epsilon}$.

Consider all the jobs which arrived during $[r_i - F_i, r_i]$, OPT must finish every such job before $r_i + \frac{\epsilon}{3}F_i$. During the interval $[r_i - F_i, r_i + \frac{\epsilon}{3}F_i]$ the optimal schedule can do at most $m(1+\frac{\epsilon}{3})F_i$ work with 1 speed.

However from Lemma 4.2.2, we know that FIFO did a significant amount of work. The jobs which arrive after $r_i - F_i$ must have more than $m(1+\frac{\epsilon}{3})F_i$ work. Hence the optimal schedule cannot possibly finish all jobs by time $r_i + \frac{\epsilon}{3}F_i$. This is a contradiction. □

This concludes the proof of Theorem 4. We will examine a variant of the work-stealing scheduler in the next section.

4.3 Work-Stealing for Unweighted Maximum Flow time

In this section, we consider a variation of work stealing called steal-k-first work stealing. Randomized work stealing is a very effective scheduler for a single DAG job in practice since the amount of scheduling and synchronization overhead is small. We would like to leverage and extend this to the case with multiple DAG jobs.

The formal definition of this algorithm will be discussed later. Our goal in this section is to show the following theorem.

Theorem 8. *The maximum unweighted flow time of the steal-k-first work stealing scheduler with* $(k+1+(k+2)\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln n\})$ *for any* $k \geq 0$ *and any* $0 < \epsilon < \frac{1}{k+2}$ *with high probability.*

By scaling the constant ϵ using the constant k in Theorem 8, we can trivially obtain the Corollary below.

Corollary 4.3.1. *The maximum unweighted flow time of the steal-k-first work stealing scheduler with* $(k+1+\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln n\})$ *for any* $k \geq 0$ *and any* $0 < \epsilon < 1$ *with high probability.*

The admit-first work stealing scheduler is the same as the steal-k-first with the constant $k = 0$. In such a case we have the following result.

Corollary 4.3.2. *The maximum unweighted flow time of the admit-first work stealing scheduler with* $(1+\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln n\})$ *for any* $0 < \epsilon < 1$ *with high probability. In particular, if* $\mathrm{OPT} \geq \ln n$*, then the scheduler is* $(1+\epsilon)$*-speed* $O(\frac{1}{\epsilon^2})$*-competitive with high probability.*

Work Stealing for a Single Job

The work stealing scheduler [14] is a distributed scheduler for scheduling a single parallel program. We have described the work stealing scheduler very early in Section 2.1. Work stealing is a As this section focuses extensively on work stealing, we will briefly describe the scheduler once again.

In work stealing, the runtime system creates a worker thread for every available core. Each worker maintains a local double-ended queue (deque). When a worker generates new ready nodes it pushes the new work onto the bottom of its deque. When a worker finishes its current node, it pops a ready node from the bottom of its deque. If there are no nodes in the local deque, the worker becomes a *thief* and *randomly* picks a *victim* worker and tries to steal work from the top of the victim's deque. We assume that it takes a unit time step to steal work between workers.

However, importantly, work stealing is not strictly a greedy scheduler though it provides strong probabilistic guarantees of linear speedup for a single job [14].

Work Stealing for Multiple Jobs

Though the work stealing scheduler is designed for scheduling a single job, we can extend it to scheduling multiple jobs in a straightforward way. In addition to the deque of each worker, a global FIFO queue is dedicated for the arrival and admission of new jobs. When a new job is released, it is inserted into the tail of the global queue. A worker will *admit* a job by popping if from the head of the global queue in FIFO order.

Under different admission strategies, workers could choose to steal work or admit a job in different orders. In this paper, we consider a strategy, namely *steal-k-first work stealing*, in which each worker always tries to randomly steal first and only tries to admit a new job if there are k consecutive unsuccessful steal attempts for some constant $k \geq 0$.

Now we analyze the theoretical performance of steal-k-first and we present its empirical performance in Section 4.5.

Proof Structure

To prove steal-k-first is competitive for maximum flow time, we need to show that it does not fall far behind the optimal schedule. We assume for the sake of contradiction that it does at some time t. Then we go back in time to a point t' where the algorithm was not far behind the optimal solution. This time is carefully defined by recursively going back in time ensuring (1) that the algorithm is always doing a significant amount of work during $[t', t]$ and (2) that we can actually find t' while ensuring (1) is true.

After finding such a time t', we are able to show that while the algorithm may fall far behind the optimal schedule during $[t', t]$ due to not taking advantage of the parallelizability of jobs, it eventually is able to do a large amount of work. With faster speed, it catches up and this allows us to bound its performance.

Before formally proving the theorem, we first show that steal-k-first does not idle much when there are jobs to execute.

Idling Steps in Steal-k-First

We define *processor idling steps* to be the aggregate number of time steps per processor where the processor is not working on a job (and is stealing instead). Recall we have assumed that each steal attempt takes 1 time step. To bound the idling time in steal-k-first's schedule, we first state a theorem from [14], which provides the bound on the time that a work stealing scheduler spends on stealing during the execution of a *single* job.

Lemma 4.3.3. *During the time interval $[e_i, c_i]$ where e_i and c_i are the execution start time and completion time of a job J_i respectively, the expected number of steal attempts is bounded by $32mP_i$ where P_i is the critical-path length and m is the number of processors. Moreover, for any $\delta > 0$, the number of steal attempts is bounded by $64mP_i + 16\ln(1/\delta)$ with probability at least $1 - \delta$.*

The Lemma above only applies to the case of a single job. By extending it we can obtain a useful lemma for the case with n jobs. In the following lemma, let e_i denote the time that job J_i is admitted from the global queue by a processor. This is the first time the job is started.

Lemma 4.3.4. *For a time interval that lies between the start time e_i and completion time c_i of a job J_i, with probability at least $1 - \frac{1}{n}$, the number of processor idling steps is bounded by $64mP_i + 32\ln(n) \leq 64m\text{OPT} + 32\ln(n)$.*

Proof. Consider Lemma 4.3.3 and choose $\delta = \frac{1}{n^2}$. The probability of any job J_i exceeding the idling time bound $64mP_i + 16\ln(n^2) = 64mP_i + 32\ln(n)$ during $[e_i, c_i]$ is $\frac{1}{n^2}$. This idling time bound holds for any time interval that is between $[e_i, c_i]$. Union bounding over all n jobs and subtracting from 1 yields the probability in the lemma. $\square$

We will also use the following lemma to later bound the idling time due to steal attempts between the arrival time r_i and the start time e_i of a job J_i.

Lemma 4.3.5. *Under steal-k-first with a speed of $s = k+1+(k+2)\epsilon$, the number of idling steps during a time interval $[t', t]$ that is contained in $[r_i, e_i]$, the time between when a job arrives and is removed from the global queue, is at most $\frac{k}{k+1}(k+1+(k+2)\epsilon)m(t-t')+km$.*

Proof. Every time a processor has more than k steal attempts, the processor will do one unit of work. Thus for any time interval of length $(t-t')$ there can be at most a $s\frac{k}{k+1}(t-t')+k$ steal attempts per processor.

Otherwise, $\frac{s}{k+1}(t-t')+1$ work will be done by the processor etaoinshrdlu The lemma follows by aggregating over all processors. □

Now we will bound the amount of work steal-k-first does. We say that a job J_i *spans* a time interval $[t_a, t_{a-1}]$, if its release time $r_i \leq t_a$ and its completion time $c_i \geq t_{a-1}$.

Lemma 4.3.6. *If a job spans a time interval $[t_a, t_{a-1}]$, then steal-k-first work stealing with speed $k+1+(k+2)\epsilon$ does at least $\frac{k+1+(k+2)\epsilon}{k+1}m(t_b-t_a)-(km+64m\text{OPT}+32\ln(n))$ work with probability at least $1-\frac{1}{n}$.*

Proof. By definition, $[t_a, t_{a-1}]$ lies between $[r_i, c_i]$. From Lemma 4.3.5, the number of idling steps during $[t_a, e_i]$ is at most $\frac{k}{k+1}(k+1+(k+2)\epsilon)m(e_i-t_a)+km \leq \frac{k}{k+1}(k+1+(k+2)\epsilon)m(t_{a-1}-t_a)+km$.

From Lemma 4.3.4, the number of idling steps during $[e_i, t_b]$ is at most $64m\text{OPT}+32\ln(n)$ with probability at least $1-\frac{1}{n}$.

Thus, during $[t_a, t_{a-1}]$ the amount that work steal-k-first with speed $k+1+(k+2)\epsilon$ does is at least the following:

$$\begin{aligned}
&(k+1+(k+2)\epsilon)\, m(t_{a-1}-t_a) - (64m\mathrm{OPT} + 32\ln(n)) \\
&- \left(\frac{k}{k+1}(k+1+(k+2)\epsilon)\, m(t_{a-1}-t_a) + km\right) \\
&= \frac{k+1+(k+2)\epsilon}{k+1} m(t_b - t_a) - (km + 64m\mathrm{OPT} + 32\ln(n))
\end{aligned}$$

This occurs with probability at least $1 - \frac{1}{n}$. $\square$

We have now shown that steal-k-first work stealing does a reasonable amount of work. We will eventually use this knowledge to prove Theorem 8.

Time Intervals in Steal-k-First

The main challenge in analyzing steal-k-first is that it is difficult to show that the remaining processing time of jobs in its queue is comparable to that of OPT's queue. Rather than directly bounding the differences between the two queues as done previously for FIFO, we will construct a set of time intervals where steal-k-first must be busy most of the time. Using the assumption that steal-k-first has resource augmentation, we will draw a contradiction by showing that steal-k-first has completed a large amount of work which is more than the total amount of work available during a time interval.

From here on, our goal is to show that the steal-k-first with $(k+1+(k+2)\epsilon)$-speed achieves a maximum flow time of $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$ with high probability. To simplify the proof,

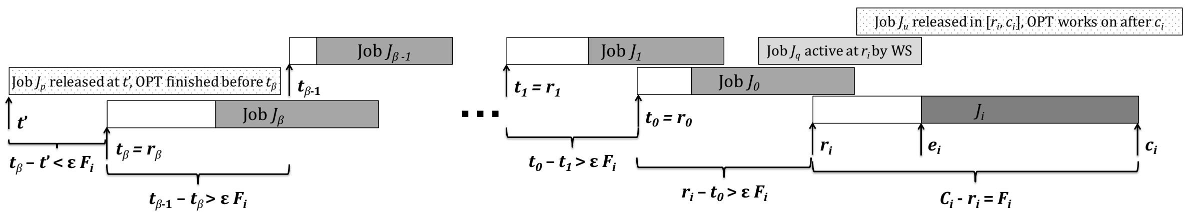

Job J_u released in $[r_i, c_i]$, OPT works on after c_i
Job $J_{\beta-1}$
Job J_1
Job J_q active at r_i by WS
Job J_p released at t', OPT finished before t_β
$t_{\beta-1}$
$t_1 = r_1$
Job J_0
t'
Job J_β
$t_0 = r_0$
J_i
r_i
e_i
c_i
$t_\beta - t' < \varepsilon F_i$
$t_\beta = r_\beta$
$t_0 - t_1 > \varepsilon F_i$
$t_{\beta-1} - t_\beta > \varepsilon F_i$
$r_i - t_0 > \varepsilon F_i$
$C_i - r_i = F_i$

Figure 4.1: An example execution trace of work-stealing identifying jobs' release and completion times.

we rewrite the objective to eliminate the max and show instead that steal-k-first achieves a maximum flow of $\frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$, $k \geq 0$ is a constant and $0 < \epsilon < \frac{1}{k+2}$.

Let J_i be the job in steal-k-first's schedule with the maximum flow time F_i. Recall that r_i and c_i are the arrival and completion time of J_i, respectively. To show contradiction, we assume that $F_i \geq \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$.

We will recursively define a set of time intervals

$$T = \{[t', t_\beta], [t_\beta, t_{\beta-1}], [t_{\beta-1}, t_{\beta-2}] \ldots [t_1, t_0], [t_0, r_i], [r_i, c_i]\}$$

where $t' \leq t_\beta \leq t_{\beta-1} \leq \ldots \leq t_1 \leq t_0 \leq r_i \leq c_i$. To illustrate the time intervals, Figure 4.1 shows an example execution trace of steal-k-first.

Let t_0 be the arrival time of the earliest arriving job among the jobs that are not finished by steal-k-first right before time r_i. For instance, in Figure 4.1 there are two jobs (job J_0 and job J_q) that are active right before r_i. Among then, job J_0 has the earliest arrival time, so t_0 is defined using it. If there are no jobs right before r_i, let $t_0 = r_i$.

Now we will define further intervals recursively. Given the time t_{a-1}, we want to define t_a. If $t_{a-1} - t_a \leq \epsilon F_i$, then we are finished defining intervals; otherwise, we define t_a as the arrival time of the earliest arriving job among those that are not finished by steal-k-first right before time t_{a-1}. We say that a certain job J_a *defines* an interval $[t_a, t_{a-1}]$, if it is the earliest arriving job unsatisfied by steal-k-first right before t_{a-1} and t_a is its arrival time.

Note that this process of defining intervals will always terminate. Specifically, the procedure terminates when $t_{a-1} - t_a \leq \epsilon F_i$, which must happen if one goes back to the first time a job arrives. We let β denote the maximum value that a takes during this inductive definition. Hence, $[t_\beta, t_{\beta-1}]$ is the earliest time interval defined in this scheme. Moreover, the arrival time t' of the earliest arriving job among those that are unfinished right before time t_β satisfies $t' - t_\beta \leq \epsilon F_i$. As in Figure 4.1, interval $[t', t_\beta]$ is the first such interval that has length less than ϵF_i.

Work Done by Steal-k-First

We intend to show that steal-k-first does a lot of work during the interval $[t_\beta, c_i]$. In fact, we will show that if the assumption of $F_i \geq \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$ is true, then steal-k-first would have done more work than the total work of all jobs that are active during $[t_\beta, c_i]$, which is not possible and leads to a contradiction, thus proving the theorem.

To do so, we partition $[t_\beta, c_i]$ into two sets of time intervals, specifically $S_1 = \{[t_a, t_{a-1}], \forall\, 0 < a \leq \beta\} \cup \{[t_0, r_i]\}$ during $[t_\beta, r_i]$, and $S_2 = \{[r_i, c_i]\}$. We first show that for intervals in S_1, steal-k-first does more work than OPT.

Lemma 4.3.7. *For any time interval $[t_a, t_{a-1}] \in S_1$ during $[t_\beta, r_i]$, with probability at least $1 - \frac{1}{n}$ the work that steal-k-first does is more than $m(t_{a-1} - t_a)$, which is as much as* OPT *does.*

Proof. By definition, there is a job J_a which defines this time interval. Specifically, this job spans the time interval. According to Lemma 4.3.6, we know that with probability $1 - \frac{1}{n}$ the amount of work that steal-k-first does is at least $\frac{k+1+(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - (km + 64m\text{OPT} + 32\ln(n))$.

Recall that by assumption that $F_i > \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$ and by definition $(t_{a-1} - t_a) > \epsilon F_i$, we have

$$\begin{aligned}(t_{a-1} - t_a) > \epsilon F_i > \frac{65}{\epsilon}(\text{OPT} + \ln(n) + k) =& \frac{1}{\epsilon}\frac{1}{m}(65km + 65m\text{OPT} + 65m\ln(n)) \\ >& \frac{1}{\epsilon}\frac{1}{m}(km + 64m\text{OPT} + 32\ln(n))\end{aligned}$$

Hence, $(km + 64m\text{OPT} + 32\ln(n)) < \epsilon m(t_{a-1} - t_a)$.

Thus during any time interval $[t_a, t_{a-1}]$ in S_1, the work done by steal-k-first (with speed $k + 1 + (k+2)\epsilon$) on jobs is at least:

$$\begin{aligned}&\frac{k+1+(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - (km + 64m\text{OPT} + 32\ln(n)) \\ &> m(t_{a-1} - t_a) + \frac{(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - \epsilon m(t_{a-1} - t_a) \\ &= m(t_{a-1} - t_a) + \frac{\epsilon}{k+1} m(t_{a-1} - t_a)\end{aligned}$$

Clearly OPT with only 1 speed can only do at most $m(t_{a-1} - t_a)$ work during this time interval. Therefore, steal-k-first does more work. □

We now show that for S_2, steal-k-first does a lot of work too.

Lemma 4.3.8. *During $[r_i, c_i] \in S_2$, the amount of work that steal-k-first does on jobs is more than $mF_i + \epsilon mF_i + m\text{OPT}$ with probability $1 - \frac{1}{n}$.*

Proof. Consider the work that steal-k-first does during $[r_i, c_i]$. By definition this interval has a length of F_i and we know that J_i spans this interval.

Directly applying Lemma 4.3.6, we derive that with probability $1 - \frac{1}{n}$ the amount of work done by steal-k-first during $[r_i, c_i]$ is at least the following.

$$\begin{aligned}&\frac{k+1+(k+2)\epsilon}{k+1} mF_i - (km + 64m\text{OPT} + 32\ln(n))\\ =&mF_i + \epsilon mF_i + \frac{\epsilon}{k+1} mF_i - (km + 64m\text{OPT} + 32\ln(n))\end{aligned}$$

By definition, $0 < \epsilon < \frac{1}{k+2}$. Therefore, $\frac{1}{k+1}\frac{1}{\epsilon} > 1$. Also recall by assumption we have that $F_i > \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$. Then, we can obtain the following.

$$\begin{aligned}\frac{\epsilon}{k+1} mF_i >&\frac{m}{k+1}\frac{65}{\epsilon}(\text{OPT} + \ln(n) + k)\\ >&65m(\text{OPT} + \ln(n) + k)\\ >&(km + 64m\text{OPT} + 32\ln(n)) + m\text{OPT}\end{aligned}$$

From the last line it should be clear that $\frac{\epsilon}{k+1} mF_i - (km + 64m\text{OPT} + 32\ln(n)) > m\text{OPT}$. Therefore, the amount of work done by steal-k-first during $[r_i, c_i]$ is more than $mF_i + \epsilon mF_i + m\text{OPT}$ with probability $1 - \frac{1}{n}$. □

We need one more critical argument to complete the analysis. The reason we defined these time intervals inductively is to identify the jobs that are active under steal-k-first during $[t_\beta, c_i]$. The total volume of these jobs is bounded by the work that OPT can finish. However, just showing that steal-k-first does more work than OPT during $[t_\beta, c_i]$ will not suffice, as OPT could have done part of this work either before t_β or after c_i. As shown in Figure 4.1, the two jobs (job J_p and job J_u) in dotted shade are executed by steal-k-first during $[t_\beta, c_i]$, while OPT finished job J_p before t_β and started working on job J_u after c_i. The next lemma bounds the maximum amount of work that are available for steal-k-first to work on during $[t_\beta, c_i]$.

Lemma 4.3.9. *For jobs that are active under steal-k-first during $[t_\beta, c_i]$, their total amount of work is at most $m(r_i - t_\beta) + m(\epsilon F_i + \mathrm{OPT} + F_i)$.*

Proof. By definition, $[t_\beta, c_i]$ consists of time intervals of S_1 during $[t_\beta, r_i]$ and time interval of $S_2 = \{[r_i, c_i]\}$. Also recall that the length of interval $[r_i, c_i]$ is F_i. Hence, the total length of $[t_\beta, c_i]$ is $(r_i - t_\beta) + F_i$.

Moreover, by definition of t_β, the earliest arriving job that is unsatisfied by steal-k-first just before time t_β must have arrived no earlier than time $t_\beta - \epsilon F_i$. Thus, the jobs that are active under steal-k-first during $[t_\beta, c_i]$ all arrived during $[t_\beta - \epsilon F_i, c_i]$.

Further, all these jobs have an optimal maximum flow time no more than OPT under the optimal scheduler. Therefore, OPT must be able to complete all of them by time $c_i + \mathrm{OPT}$. Knowing that OPT can only work on these jobs during $[t_\beta - \epsilon F_i, c_i + \mathrm{OPT}]$, the total amount of work of those jobs can have volume at most $m(r_i - t_\beta) + m(\epsilon F_i + \mathrm{OPT} + F_i)$. □

Finally, we are now able to complete the proof.

Proof of Theorem 8 To prove the theorem, we consider the jobs that are active under steal-k-first during $[t_\beta, c_i]$. By Lemma 4.3.9, we know that the total amount of work of these jobs, denoted as X, is bounded: $X \leq m(r_i - t_\beta) + m(\epsilon F_i + \mathrm{OPT} + F_i)$. Note that these jobs are the only ones available for steal-k-first to work on during $[t_\beta, c_i]$. Therefore, during $[t_\beta, c_i]$ steal-k-first cannot do more than X work even with speedup.

Also consider the minimum amount of work that steal-k-first must have done during $[t_\beta, c_i]$, denoted as Y, if we assume $F_i > \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$ is true. We will see that $Y > X$, which leads to a contradiction.

From Lemma 4.3.7, we know that during $[t_\beta, r_i]$ the amount of work steal-k-first does is more than the following.

$$m\left(\sum_{0<a\leq\beta}(t_{a-1} - t_a) + (r_i - t_0)\right) = m(r_i - t_\beta)$$

From Lemma 4.3.8, we know that during $[r_i, c_i]$, steal-k-first does more than $mF_i + \epsilon m F_i + m\mathrm{OPT}$ work. Thus, for interval $[t_\beta, c_i]$, we have $Y > m(r_i - t_\beta) + mF_i + \epsilon m F_i + m\mathrm{OPT}$.

Now we compare X and Y:

$$\begin{aligned} Y - X >& m(r_i - t_\beta) + mF_i + \epsilon m F_i + m\mathrm{OPT} \\ &- m(r_i - t_\beta) - m(\epsilon F_i + \mathrm{OPT} + F_i) > 0 \end{aligned}$$

Hence, $Y > X$. In other words, if the assumption of F_i is true, during $[r_i, c_i]$ steal-k-first must have done more work than the total available work, which gives a contradiction.

Thus, we obtain the theorem. $\square$

Remarks on Steal-k-First

Note that for steal-k-first work stealing with $k = 0$, instead of steal first, this scheduler will in fact admit all jobs from the global queue first. We denote this special case as *admit-first*. From Theorem 8, we know that the theoretical performance of steal-k-first is better with smaller constant k.

Therefore, admit-first has the best theoretical performance and is $O(\frac{1}{\epsilon^2})$-competitive with high probability with $1 + \epsilon$ speed, as it guarantees that a job's execution is not delayed by unnecessary random stealing. This is Theorem 5.

However, we will show in Section 4.5 that steal-k-first for a relatively large k performs better than admit-first empirically. Intuitively, if there is any job available for stealing, then in expectation m consecutive random steal attempts would be able to find the stealable work. Thus, for $k \geq m$, steal-k-first better approximates FIFO, which we know works well.

In contrast, in admit-first jobs could run sequentially when there are more than m unfinished jobs. During these times, jobs at the end of the global queue take long time to be admitted and they further take longer time to finish sequential execution in the worst case. Hence, this could increase the maximum flow time of the system.

Moreover, steal-k-first requires a speed of more than $(k + 1)$ theoretically to be competitive mainly due to the worst case scenario where each job has a unit time of work but takes k stealing steps to admit. However, in practice, jobs have much larger work and the constant k steal attempts for admitting a job is negligible in practice.

4.4 Work Stealing Lower Bound

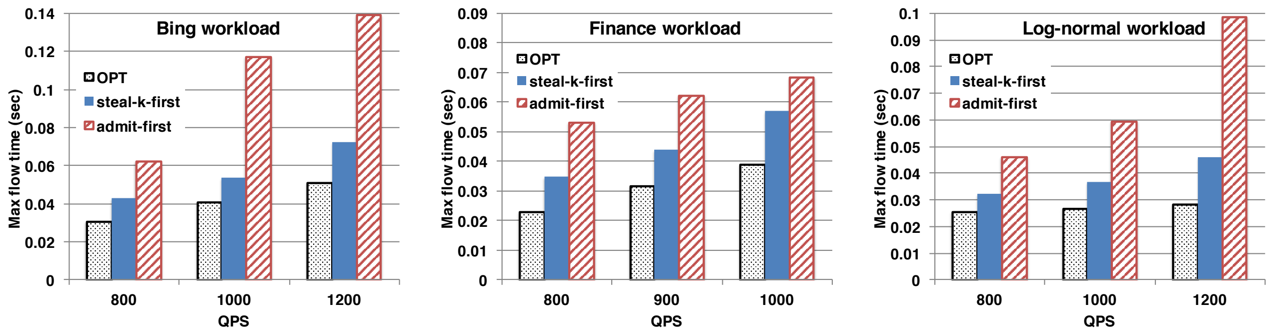
In this section we give a lower bound for the work stealing algorithm. We show that in the online setting, when given any constant speed, the scheduler is $\Omega(\log n)$ competitive. This shows that our upper bound analysis of the algorithm is effectively tight.

Lemma 4.4.1. *Work stealing is $\Omega(\log n)$-competitive for maximum flow time in the online setting when given any constant resource augmentation.*

Proof. Let n be an input parameter and let the number of machines be $m = \log n$. Let s be a constant specifying the resource augmentation given to work stealing. The schedule consists of n jobs, which are identical. A job consists of one task which is the predecessor of $m/10$ independent tasks. Note that the total work of the job is $m/10 + 1$ and can be competed by a 1 speed scheduler scheduler in 2 time steps. A single job is released at multiples of $2m$ starting at time 0. Note that even if a job is executed sequentially, it will complete in only $m/10 + 1$ time steps. Thus, these jobs do not have overlapping times where multiple jobs are alive in any non-idling schedule.

Now fix any job and consider the probability that the job executes completely sequentially by a work stealing scheduler. This occurs if every steal attempt fails to find the processor holding the tasks for the job. In a single time step, the probability that $m-1$ processors do not successfully steal is $(1 - \frac{1}{m-1})^{m-1} \geq \frac{1}{2e}$ for sufficiently large m. The probability that all processors fail to steal for $m/10$ time steps is greater than $(\frac{1}{2e})^{m/10}$.

Now consider the expected number of jobs which execute sequentially by work stealing. There are $n = 2^m$ jobs released. The expected number of jobs to execute sequentially is $2^m(\frac{1}{2e})^{m/10} \geq 1$. Thus, the expected maximum flow time of work stealing with s speed is



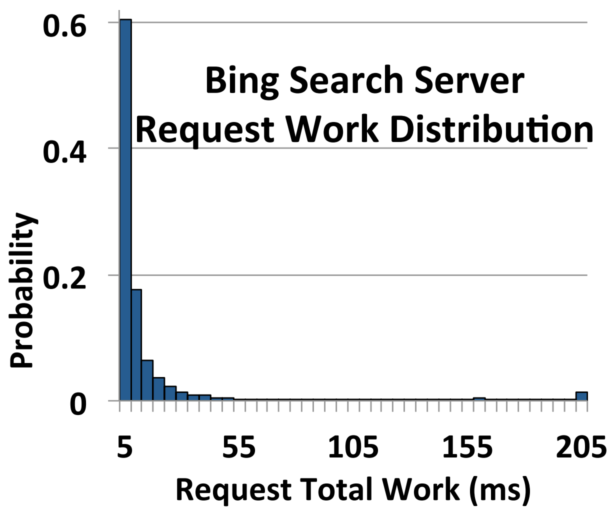
(a) Bing workload (b) Finance workload (c) Log-normal workload

Figure 4.2: Experimental results comparing the maximum flow time running on three work distributions with three different load settings and scheduled using simulated OPT, steal-k-first, and admit-first (from left to right). Note that the scale of the y-axis for the figures differ. From all different settings, OPT has the smallest max flow time, while admit-first has the largest max flow time.

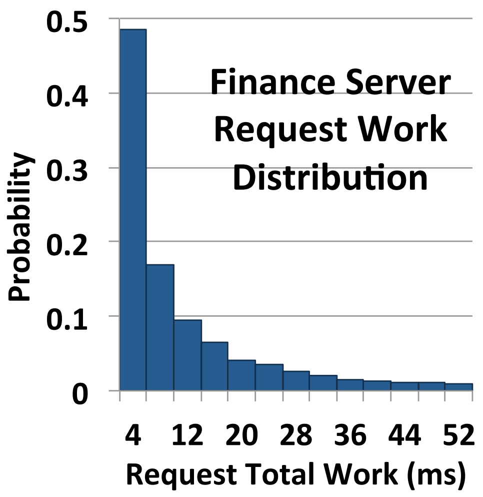
$\frac{m/10+1}{s} = \frac{\log n}{s}$. Knowing that the optimal solution has maximum flow time 2 and $s = O(1)$, the lemma follows. $\square$

4.5 Experimental Results for Unweighted Maximum Flow Time

In this section we present the experimental results using realistic and synthetic workloads to compare the performance of OPT and two work stealing strategies: (1) *Admit-first* where workers preferentially admit jobs from the global queue and only steal if the queue is empty, and (2) *Steal-k-first* where workers preferentially steal and only admit a new job if k steal attempts fail (we use $k = 16$). Our experiments indicate that steal-k-first performs better and is almost comparable to an optimal scheduler.



(a) Bing search



(b) Finance server

Figure 4.3: Work distribution of two workload: Bing web search [31] and an option pricing finance server [42].

Setup: We conduct experiments on a server with dual eight-core Intel Xeon 2.4Ghz processors with 64GB main memory. The server runs Linux version 3.13.0, with processor throttling, sleeping, and hyper-threading disabled. The work-stealing algorithms are implemented in Intel Thread Building Block (TBB) [41] version 4.3, a well-engineered popular work-stealing runtime library. We extended TBB to schedule multiple jobs arriving online by adding a global FIFO queue for admitting jobs and we implement both admit-first and steal-k-first.

Since we do not know the optimal scheduler, we must approximate it using a simulated scheduler by reducing a parallel scheduling problem to a sequential scheduling problem on a single processor. In particular, for this lower bound, we assume that there is no preemption overhead and that each job can get linear speedup (fully parallelizable). Therefore, we can execute each job one at a time assuming it is a sequential job with execution time equal to its W/m where W is its total work. We then run all jobs using FIFO which is optimal in this setting. When jobs are fully parallelizable, this reduces the problem to the case where there is only one machine. In this setting, it is well known that FIFO is optimal for maximum flow time [19]. Thus, this scheduler has the performance for maximum flow time that is at least as good as any feasible scheduler, including the optimal schedule.

Workloads: We evaluate different strategies on work distributions from two real-world applications shown in Figure 4.3 and additional synthetic workloads with log-normal distribution. Henceforth we shall refer to workload generated from the three distributions as the *Bing workload*, the *finance workload* and the *log-normal workload*, respectively. For each distribution, we select a set of queries-per-second, QPS, to generate workloads with low ($\sim$ 50%), medium ($\sim$ 60%), and high ($\sim$ 70%) machine utilization respectively, and the inter-arrival time between jobs is generated by a Poisson process with a mean equal to $1/QPS$. Each job contains CPU-intensive computation and is parallelized using parallel for loops. $100,000$ jobs are used to obtain a single point in the experiments.

Figure 4.2 shows the experimental results comparing simulated OPT, steal-k-first and admit-first under three different work distributions and three different load settings (i.e., query-per-second). The experiments indicate that, even though our results on OPT are lower bounds on maximum flow time, steal-k-first performs comparably to OPT — matching our intuition that it is a closer approximation for maximum flow time, as discussed at the end of Section 4.3.

Recall that steal-k-first has worse theoretical performance than admit-first. However, in practice, admit-first generally performs worse in terms of maximum flow time and the performance difference increases as load increases (for instance, for Bing and log-normal workloads with high utilization, admit first has twice the maximum flow). This matches our intuition — at higher loads, admit-first executes jobs more or less sequentially, while steal-first provides parallelism to already admitted jobs before admitting new jobs. Therefore steal-first is closer to FIFO in that it tries to execute jobs that arrived earlier with more parallelism. Therefore, in practice, steal-first is likely to be a good implementation for schedulers that want to minimize maximum flow time without incurring the large overheads of FIFO.

4.6 Weighted Maximum Flow Time

In this section our goal is to prove that the algorithm Biggest-Weight-First (BWF) is a scalable algorithm for minimizing the maximum weighted flow time, which is Theorem 7.

BWF is defined as follows similarly to FIFO except that priority is given to the jobs with the biggest weight. At any time t, BWF orders the jobs in decreasing order by their weight, breaking ties arbitrarily. The algorithm then assigns all of the ready nodes for the first job to some processor. The algorithm recursively does the same for the next job in the list. This continues until all processors have been assigned some node or there are no more ready nodes available. Like FIFO, BWF may have a choice on which ready nodes of a job to schedule if the job has more ready nodes than the number of processors which have not been assigned to a node when the job is considered. In this case, we assume the scheduler chooses an arbitrary set of ready nodes.

The reminder of this section is devoted to proving Theorem 7. For the rest of this section, we assume that BWF is given $(1 + 3\epsilon)$-speed for some constant $0 < \epsilon < \frac{1}{3}$ and we will show that BWF is $\frac{3}{\epsilon^2}$ competitive. Fix any sequence of jobs and let OPT denote the optimal schedule on this instance as well as the optimal maximum weighted flow time. Let F_a^* be the flow time of a job J_a in OPT.

Let J_i be the job in BWF's schedule with the maximum weighted flow time $w_i F_i$. For the sake of contradiction, we assume that $w_i F_i > \frac{3}{\epsilon^2}\mathrm{OPT}$. Since $\mathrm{OPT} = w_i F_i^*$, $F_i > \frac{3}{\epsilon^2} F_i^*$, where F_i^* is the flow time of J_i in OPT. By comparing to the weight w_i of job J_i, any jobs with weight at least w_i are referred as *heavy* jobs, and any jobs with less weight than w_i are referred as *light* jobs.

Time Intervals in BWF

Similar to the time intervals specified in Section 4.3, we will inductively define a set of time intervals

$$T = \{[t', t_\beta], [t_\beta, t_{\beta-1}], [t_{\beta-1}, t_{\beta-2}] \ldots [t_1, t_0], [t_0, r_i], [r_i, c_i]\}$$

where $t' \leq t_\beta \leq t_{\beta-1} \leq \ldots \leq t_1 \leq t_0 \leq r_i \leq c_i$.

Recall that r_i and c_i are the arrival and completion time of J_i, respectively. Consider the *heavy* jobs that BWF is scheduling right before r_i. Let t_0 be the arrival time of the *earliest* arriving one of those jobs. If there are no heavy jobs right before r_i, then let $t_0 = r_i$.

Now we define further intervals recursively. Given the times t_{a-1}, we want to define t_a. If $t_{a-1} - t_a \leq \epsilon F_i$, then we are done defining time intervals; otherwise, we define t_a to be the arrival time of the earliest arriving heavy job J_a that are unsatisfied under BWF right before time t_{a-1}. Again if there are no heavy jobs unsatisfied by BWF just before time t_{a-1} then let $t_a = t_{a-1}$. We let β denote the maximum value that a takes during this inductive definition. Hence, $[t_\beta, t_{\beta-1}]$ is the earliest time interval defined in this scheme.

Note that this process of defining intervals is almost the same as in Section 4.3. The only difference is that the job J_a, which defines the interval $[t_a, t_{a-1}]$, is the earliest unfinished *heavy* job under BWF. We only consider heavy jobs, because under BWF only heavy jobs can preempt job J_i and other heavy jobs; any light jobs can only execute when all the available nodes of all the active heavy jobs are already executing by some processors.

Thus, when analyzing the flow time of J_i and other heavy jobs, we can ignore the remaining light jobs, since they cannot interfere with the execution of heavy ones. Hence, the processor idling steps in the remaining of this section refers to the time steps where a processor is not working on nodes corresponding to heavy jobs.

We begin the proof by showing that during all time intervals between $[t_\beta, r_i]$, BWF is using most time steps to process ready nodes for heavy jobs.

Lemma 4.6.1. *During any interval $[t_a, t_{a-1}]$ where $a \leq k$, the number of processor idling steps is at most $m\frac{\epsilon^2}{3}F_i$.*

Proof. For the sake of contradiction, assume that this is not true. Then consider the job which defines $[t_a, t_{a-1}]$ and let this job be J_a. By definition this heavy job arrived at t_a and is still being processed at time t_{a-1}. From BWF's scheduling policy, every time step during $[t_a, t_{a-1}]$, where some processors find no nodes from heavy jobs to work on, all ready nodes of J_a are being scheduled. Hence the processors are decreasing the remaining critical path of J_a at these times by Proposition 4.1.1. Since the job is not finished until at t_{a-1}, this job must have a critical-path length P_a longer than $P_a > t_{a-1} - t_a > \frac{\epsilon^2}{3}F_i$. Also since J_a is a heavy job and $w_a \geq w_i$ and by assumption $w_iF_i > \frac{3}{\epsilon^2}\mathrm{OPT}$, its weighted flow time is at least

$$w_a(t_{a-1} - t_a) > w_a\frac{\epsilon^2}{3}F_i \geq w_i\frac{\epsilon^2}{3}F_i > \frac{\epsilon^2}{3}\frac{3}{\epsilon^2}\mathrm{OPT} \geq \mathrm{OPT}$$

However, OPT cannot complete a job faster than its critical-path length, so $F_a^* \geq P_a$. Further, J_a's weighted flow time under OPTis at most the maximum weighted flow time OPT. We have

$$\text{OPT} \geq w_a F_a^* \geq w_a P_a > w_a(t_{a-1} - t_a) > \text{OPT}$$

This gives a contradiction. □

Using the previous lemma, we bound the aggregate amount of work done by BWF on heavy jobs during $[t_\beta, r_i]$.

Lemma 4.6.2. *During $[t_\beta, r_i]$, the amount of work that BWF does on heavy jobs is more than $m(1 + 2\epsilon)(r_i - t_\beta)$.*

Proof. From Lemma 4.6.1, we know that there are only $m\frac{\epsilon^2}{3}F_i$ processor idling steps where a processor is not working on nodes corresponding to heavy jobs during any time interval $[t_a, t_{a-1}]$. In addition, we know $t_{a-1} - t_a > \epsilon F_i$, since $a \leq \beta$. Hence, the work done by BWF (with $1 + 3\epsilon$ speed) on heavy jobs during $[t_a, t_{a-1}]$ is at least:

$$\begin{aligned}
&m(1 + 3\epsilon)(t_{a-1} - t_a) - m\frac{\epsilon^2}{3}F_i \\
>&m(1 + 3\epsilon)(t_{a-1} - t_a) - m\frac{\epsilon}{3}(t_{a-1} - t_a) \\
>&m(1 + 2\epsilon)(t_{a-1} - t_a)
\end{aligned}$$

Summing over all the intervals results in the lemma. □

Similarly, we can bound the amount of work done by BWF on heavy jobs during $[r_i, c_i]$.

Lemma 4.6.3. *During $[r_i, c_i]$, the amount of work that BWF does on heavy jobs is more than $m(1+2\epsilon)F_i$.*

Proof. By assumption, $F^* < \frac{\epsilon^2}{3}F_i$. Since OPT cannot finish a job in less time than its critical-path length, job J_i has $P_i \leq F_i^* < \frac{\epsilon^2}{3}F_i$. From Proposition 4.1.1, we can derive that the number of processor idling steps where a processor is not working on heavy jobs is at most mP_i. Hence, the amount of work done by BWF during $[r_i, c_i]$ is at least $m(1+3\epsilon)F_i - mP_i > m(1+3\epsilon)F_i - m\frac{\epsilon^2}{3}F_i > m(1+2\epsilon)F_i$, since $\epsilon < \frac{1}{3}$. □

Now we bound the maximum amount of work that are available for BWF to work on during $[t_\beta, c_i]$.

Lemma 4.6.4. *For jobs that are active under BWF during $[t_\beta, c_i]$, their total amount of work is at most $m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$.*

Proof. By definition, the total length of $[t_\beta, c_i]$ is $(r_i - t_\beta) + F_i$. Moreover, by definition of t_β, the earliest arriving heavy job that is unsatisfied by BWF just before time t_β must have arrived no earlier than time $t_\beta - \epsilon F_i$. Thus, the heavy jobs that are active under BWF during $[t_\beta, c_i]$ all arrived during $[t_\beta - \epsilon F_i, c_i]$.

Furthermore, all these heavy jobs have an optimal maximum weighted flow time no more than OPT under the optimal scheduler, i.e., OPT $\geq F_a^* w_a$. By definition of a heavy job $w_a \geq w_i$ and by assumption $w_i F_i > \frac{3}{\epsilon^2}$OPT, we have $w_a F_i \geq w_i F_i > \frac{3}{\epsilon^2}\text{OPT} > \frac{3}{\epsilon^2}F_a^* w_a$. Thus, the flow time F_a^* of these heavy jobs under the optimal schedule is $F_a^* < \frac{\epsilon^2}{3}F_i$.

Therefore, OPT must be able to complete all of them by time $c_i + \frac{\epsilon^2}{3}F_i$. Knowing that OPT can only work on these jobs during $[t_\beta - \epsilon F_i, c_i + \frac{\epsilon^2}{3}F_i]$, the total amount of work of those jobs can have volume at most $m(r_i - t_\beta + F_i) + m(\epsilon F_i + \frac{\epsilon^2}{3}F_i) = m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$. □

Finally, we are ready to complete the proof.

Proof of Theorem 7 To prove the theorem, we consider the heavy jobs that are active under BWF during $[t_\beta, c_i]$. By Lemma 4.6.4, we know that the total amount of work of these jobs, denoted as X, is bounded: $X \leq m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$. Note that these jobs are the only ones available for BWF to work on, so during $[t_\beta, c_i]$ BWF cannot do more than X work even with speedup.

On the other hand, consider the minimum amount of work that BWF must have done during $[t_\beta, c_i]$, denoted as Y, assuming that $w_i F_i > \frac{3}{\epsilon^2}\mathrm{OPT}$ is true. We will see that $Y > X$, which leads to a contradiction.

From Lemma 4.6.2, we know that during $[t_\beta, r_i]$ the amount of work BWF does is more than $m(1 + 2\epsilon)(r_i - t_\beta)$ From Lemma 4.6.3, we know that during $[r_i, c_i]$, BWF does more than $m(1 + 2\epsilon)F_i$ work. Thus, for interval $[t_\beta, c_i]$, we get $Y > m(r_i - t_\beta) + m(1 + 2\epsilon)F_i$.

Now we compare X and Y and note that $\epsilon < \frac{1}{3}$:

$$\begin{aligned} Y - X >& m(r_i - t_\beta) + m(1 + 2\epsilon)F_i \\ & - m(r_i - t_\beta) - m(1 + \epsilon + \frac{\epsilon^2}{3})F_i > 0 \end{aligned}$$

Hence, if the assumption of $w_i F_i$ is true, then during $[r_i, c_i]$ BWF must have done more work than the total available work, which gives a contradiction. By scaling ϵ, we obtain the theorem. □

Remarks

This result of weighted flow time can be applied to maximum stretch. In the sequential setting, weighted flow time captures maximum stretch by setting the weight to be the inverse of the processing time. In other words, the flow of a job is scaled by the inverse of its processing time in the stretch objective for sequential jobs.

However, stretch is not well-defined for DAG jobs. In particular, it is unclear whether the flow time should scaled by the inverse of the total work or the critical path length. Although there are two natural interpretations of the stretch in the DAG setting, both of them can be still captured by weighted flow time if the algorithm has knowledge of the total work or the critical path length of the jobs.

It remains an open question whether there are any scalable algorithms for maximum stretch for DAGs in the case that the algorithm is completely non-clairvoyant.

4.7 Conclusion

The DAG model has been influential in design of theoretically good and practically efficient schedulers for executing single parallel program. We have now given the first results in this model for maximum flow time, an important scheduling metric, for multiprogrammed environment where jobs arrive online. In combination with the result described in chapter 3, we have covered two of the most important objectives in online scheduling.

Our results in this chapter also offer hints that the online scheduling of parallel programs in the DAG model might be different than in the arbitrary speed-up curves model. It would be of interest to further explore connections and differences between these two models.

Chapter 5

Practical Average Flow Time

In this chapter, we will return to the problem of minimizing average flow time in the DAG model of parallelism. For this problem, n parallel DAG jobs arrive over time (online) and share a single machine with m identical processors. However, in this chapter we are particularly interested in designing a *theoretically good and practically efficient* algorithm for this problem which can be implemented in real systems.

In chapter 3, we discussed the first theoretical results on average flow time for scheduling multiple DAG jobs online, show by Agrawal et. al [2].

There we showed that Latest-Arrival-Processor-Sharing (LAPS) [22] — an algorithm which generalizes Round-Robin (RR) — is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive in this model. We also described a variant of a greedy algorithm called shortest-job-first (SJF), more precisely defined in our case as smallest-work-first (SWF), and showed that it is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive.

However, 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. Both of these algorithms require an unacceptable number of preemptions. The LAPS algorithm requires splitting the processing power evenly among a set of jobs, so it preempts jobs every infinitesimal time step; 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 in practice.

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

For scheduling DAG jobs online to minimize average flow time, we would like to replicate the success of work stealing (a scheduler for single DAGs that works well in practice) and build on current theoretical discoveries to find an algorithm that has both strong theoretical guarantees and good practical performance. We would like to provide an algorithm which achieve the following:

- It provides good theoretical guarantees.
- It is 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.

- 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.

To incur low scheduling overhead, we want to design a decentralized work stealing based scheduler. Work stealing has been described extensively in both the previous chapter and section 2.1. Work stealing can lead to both fewer preemptions and smaller synchronization overhead.

The natural approach to extend work stealing to this problem is to only allow 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.

Recall that in the previous chapter, we used this sort of approach to design a practical scheduler for minimizing *maximum flow time* [3]. Unfortunately, for average flow time, using a scheduler that never preempts until its deque is empty does not lead to good theoretical guarantees.

Consider the following input. 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 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.

In this chapter we will define an algorithm 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 show the following theorems about this algorithm.

Theorem 9. *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 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 sinc DREP only preempts a job when a new job arrives.

Theorem 10. *DREP requires processors to switch between unfinished jobs at most $O(mn)$ times over the entire schedule. If jobs are sequential, the total expected number of preemptions is $O(n)$.*

In theory, DREP has a worse speed augmentation than what is known for LAPS. But unlike LAPS, 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 simulations, 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) [40], shortest-job-first (SJF) [15] and round-robin (RR) [20]. 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 [27], 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) [2]. The empirical evaluation based on the actual implementation demonstrates that DREP has comparable performance with SWF.

In the following section we introduce some preliminaries and notation necessary for the chapter. Then, we describe and analyze the DREP algorithm in sections 5.2 and 5.3. We will describe the experimental results in the following section.

5.1 Preliminaries

In this average flow time problem we are again 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). There are two important paramemters for a 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.

Two important observations about these quantities are here below.

Observation 3. *Any job J_i takes at least $\max\{\frac{W_i}{m}, C_i\}$ time to complete in any schedule with unit speed.*

Observation 4. *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 critical 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 will let OPT refer to both the final objective of the optimal schedule and the schedule itself.

Potential Function Analysis:

Similar to chapter 3, we will use potential function analysis to analyze our algorithm. We will briefly restate this framework.

Recall that in this technique, we define 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. The change in A's objective at time t is denoted by $\frac{\mathrm{d}G_a(t)}{\mathtt{dt}}$; for the sum of completion times, this is equal to the number of active jobs in A's schedule at time t, i.e. $\frac{\mathrm{d}G_a(t)}{\mathtt{dt}} = |A(t)|$. 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, $\frac{\mathrm{d}G_a(t)}{\mathtt{dt}} + \frac{\mathrm{d}\Phi(t)}{\mathtt{dt}} \leq c \cdot \frac{\mathrm{d}G_o(t)}{\mathtt{dt}}$

Integrating these conditions over time shows that A is $(\alpha + \beta + c)$-competitive.

In the next section we define DREP and give an theoretical analysis of its performance.

5.2 DREP for Sequential Jobs

We will 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 tries to take 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 reason:

1. J_j was not assigned to a processor on arrival (no processor happened to switch to it)
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 5.3). 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 10. 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 will adapt this algorithm to the case where jobs are parallelizable DAGs. We will show how to combine the algorithm with work stealing.

5.3 DREP with Work-Stealing: A Practical Parallel Scheduling Algorithm

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

5.3.1 Combining DREP with Work-Stealing

We will 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. 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. Like in 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. 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 are two main modifications necessary to the standard work stealing. First, we must handle the deques to support multiple jobs instead of a single job. Second, we must 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 must handle when jobs arrive. 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 makes a steal attempt, 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 instead, p performs a *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.

We will also note the following facts about muggable deques.

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 everything.

4. Once a thief mugs a deque, it can always do at least one unit of work since muggable deques are never empty.

5.3.2 Analysis of DREP with Work-Stealing

Now we will analyze the DREP algorithm for minimizing average flow time. The goal is to show Theorem 9. 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 [2]) because the work-stealing algorithm is not strictly work-conserving. Typically, the potential functions used previously use Observation 4 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. To deal with this, our potential function will have another, different potential function embedded within it, adapted from prior work on work stealing.

Probability of Working on a Job:

We will first give a lemma on the probability that a processor is working on a specific job.

Lemma 5.3.1. *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 trivially 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 the completion of jobs, 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 $i \not\to j$ be the event it is not. This processor is working on any alive J_j with probability $\Pr[i \to j \mid i \not\to j'] = \Pr[i \to j \text{ and } i \not\to j']/\Pr[i \not\to j']$.

Inductively, we have $\Pr[i \not\to j'] = 1 - \frac{1}{n'}$ and $\Pr[i \to j \text{ and } i \not\to j'] = \Pr[i \to j] = \frac{1}{n'}$. Therefore, $\Pr[i \to j \mid i \not\to j'] = \frac{1}{n'-1}$. $\square$

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 new 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 $\text{rank}_i(t) = \sum_{j \in A(t), r_j \leq r_i} 1$ 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 function the *flow potential*. The first term $\frac{1}{m}\text{rank}_i(t) Z_i(t)$, 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 [5]. 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 can now be defined.

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

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

Analysis:

In order to prove Theorem 9, we first show the completion and arrival conditions in Lemma 5.3.2.

Then we will show the running condition in Proposition 5.3.3, which is proven using Lemmas 5.3.4 to 5.3.9.

Now we we show the completion and arrival conditions.

Lemma 5.3.2. *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.*

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

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

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. □

Proving the running condition is will be far more difficult than proving the completion and arrival conditions. 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 lie 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 4 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 [5], 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, unfortunately, 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).[4] 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,

[4]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.

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. Now we will begin our proof of the running condition.

Proposition 5.3.3. *In expectation, the running condition holds at any time t. That is, at any time t it is the case that $\frac{\mathrm{d}G_a(t)}{dt} + \frac{\mathrm{d}\Phi(t)}{dt} \leq O(\frac{1}{\epsilon^2}) \cdot \frac{\mathrm{d}G_o(t)}{dt}$.*

The running condition involves the instantaneous change of the potential 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 5.3.4. *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 5.3.5. *At time t, if $|O(t)| \geq \frac{\epsilon}{10}|A(t)|$, 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 $|O(t)| \geq \frac{\epsilon}{10}|A(t)|$, we will ignore the algorithm's impact on the potential. We will just use 5.3.4 to examine the running condition.

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

□

The other case, where we consider the decrease in potential from the algorithm processing jobs, is significantly more difficult. Recall 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 5.3.6. *If a job has $d_i(t) \geq 2p_i(t)$ 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 $x \geq (4+4\epsilon)p_i$. Therefore, the total expected number of work plus mugging steps is $s/2 + w = s/2 + x - s = x - s/2 \geq x - x/2 = x/2 \geq (2+2\epsilon)p_i$. □

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 5.3.7. *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 $\mathbb{E}\left[\frac{\mathrm{d}\Phi_i^A(t)}{dt}\right] \leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\,\mathrm{rank}_i(t)$.*

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 $\mathbb{E}\left[\frac{\mathrm{d}\Phi_i^A(t)}{\mathtt{dt}}\right] \leq -\frac{10}{\epsilon}\frac{\mathrm{rank}_i(t)}{m}\mathbb{E}\left[\frac{\mathrm{d}Z_i(t)+d_i^m(t)}{\mathtt{dt}}\right] \leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\,\mathrm{rank}_i(t)$. □

Now we need to consider 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 [5].

Lemma 5.3.8. *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 5.3.9. *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* $\mathbb{E}\left[\frac{d\Phi_i^A(t)}{dt}\right] \leq -200/\epsilon^2$.

Proof. From Lemma 5.3.8, we know that $\mathbb{E}\left[\frac{d\psi_i(t)}{dt}\right] \leq -1/16$ if it has enough steal attempts; the same is trivially true if the job completes. Plugging it into the potential, we get $\mathbb{E}\left[\frac{d\Phi_i^A(t)}{dt}\right] \leq -\frac{10}{\epsilon}\frac{320}{\epsilon}\frac{1}{16} \leq -200/\epsilon^2$ □

We can now complete the proof of the running condition.

Proof of [Lemma 5.3.3] **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 5.3.9, 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 straight-forward way since there are a total of $(4+4\epsilon)p_i$ steps in that time step. In the second case, from Lemma 5.3.6, 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 5.3.8, we know that the algorithm can never increase the potential during execution. Hence, Claim 5.3.5 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 5.3.7 on these jobs to obtain $\mathbb{E}\left[\frac{d\Phi^A(t)}{dt}\right] \leq \sum_{i\in A(t)\setminus O(t)} -\frac{20+20\epsilon}{\epsilon|A(t)|}\text{rank}_i(t)$. Assuming the worst case that these are the lowest rank jobs we get the following change to the potential.

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

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 5.3.4 and we have $\frac{\mathrm{d}G_a(t)}{\mathtt{dt}} = |A(t)|$. Therefore, the running condition is satisfied. □

With this last lemma, we have shown that the arrival, completion and running conditions hold. Note that we used $(4+4\epsilon)$ speed in the analysis of the running condition. Therefore, we can conclude that the work-stealing scheduler is $O(1)$ competitive with $(4+4\epsilon)$ speed augmentation completing the proof of the main theorem.

5.4 Experimental Evaluation

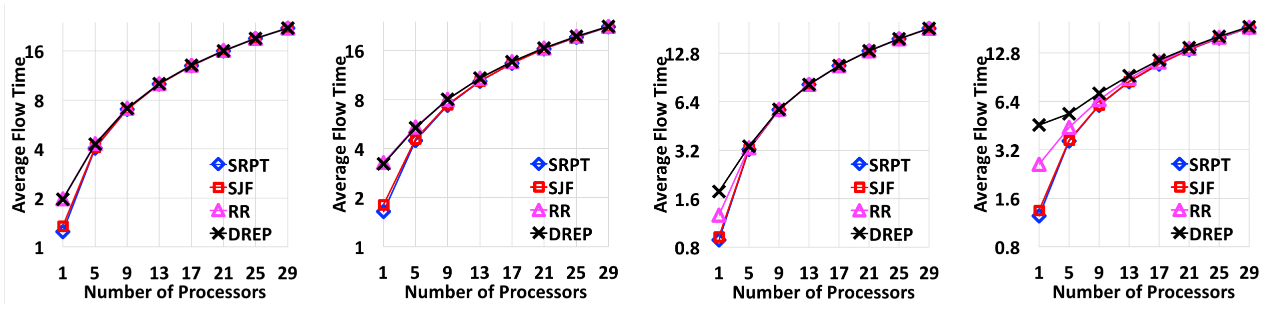
This section presents the evaluation of DREP through both simulation and empirical experiments based on actual implementations. Simulations allow us to compare DREP with a wide variety of scheduling policies, including ones that are clairvoyant and/or infeasible to implement due to the need to *preempt at infinitesimal time steps.* 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) [2], i.e., the SJF counterpart for parallel jobs, which is clairvoyant and work conserving. We obtain the actual implementations by modifying Cilk Plus [27], a production quality parallel runtime system, to approximate SWF and DREP and compare their performance in practice.

5.4.1 Evaluation Based on Simulations

Compared Algorithms: Via simulations, we compare DREP against a wide variety of schedulers: shortest-remaining-processing-time (SRPT) [40], shortest-job-first (SJF) [15] (which generalizes to smallest-work-first (SWF) [2] for parallel jobs), and round robin (RR) [20]. 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.

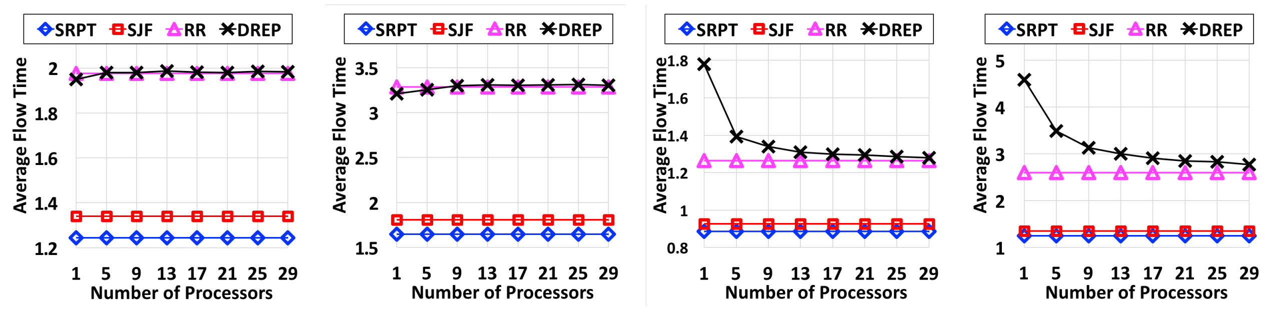
It is important to note that all the existing algorithms, including the ones that we compared in the simulation, suffer from frequent preemptions, high overheads, and non-clairvoyance. LAPS [22], in particular, is very difficult to implement since it needs to know the parameter epsilon (speedup against 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.



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

Figure 5.1: 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

Figure 5.2: Fully parallel jobs setting with low and high machine utilizations

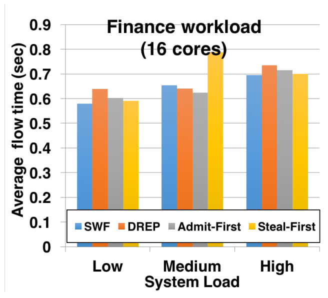
Setup: We use two different work distributions from real-world applications to generate the workloads: the *Bing workload* and the *Finance workload* [35]. We randomly generate a job by randomly sample its work from the experimented work distribution. For each work distribution, we vary the *queries-per-second (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 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.

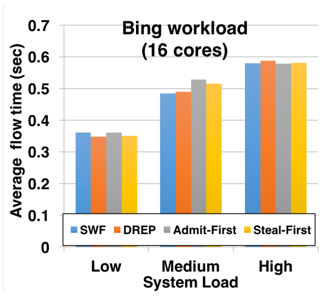
We simulate two job cases: (1) the *sequential jobs with multiprocessors setting*, where each job is sequential and can use only one processor at any time, and (2) the *fully parallel jobs setting*, where each job obtains near-linear speedup with respect to the number of processors given. These two settings 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 5.1 shows the results of simulating the sequential jobs on multiple processors setting, and Figure 5.2 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.

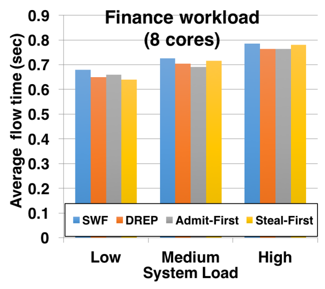
For sequential jobs on multiprocessors in Figure 5.1, SRPT and SJF have been proved to be scalable for average flow; but they are both clairvoyant, i.e., requiring the a priori knowledge of the amount of work for each job. In contrast, DREP and RR are non-clairvoyant and DREP's performance is very close 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



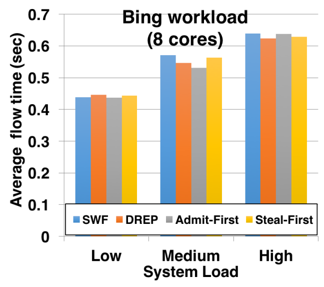
(a) Finance workload on 16 core



(b) Bing workload on 16 core



(c) Finance workload on 8 core



(d) Bing workload on 8 core

Figure 5.3: Parallel Cilk Plus jobs on multicore with varying system load and different work distributions

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 5.2, 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 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.

5.4.2 Evaluation Based on Real Implementation

To evaluate the empirical performance and practicality of DREP, we implemented a work-stealing based DREP in Cilk Plus [27], a widely-used parallel runtime system. For comparison, we implemented a few variants of work-stealing based scheduling strategies: steal-first [35], admit-first [35], and an approximation of smallest-work-first [2] 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 [3], 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 5.3 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 5.3 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.

5.5 Conclusion

This chapter focused on 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.

Chapter 6

Throughput

In this chapter, we examine a problem in the directed acyclic graph model that is very different from the previous problems which focused on flow time objectives. Our problem is instead on scheduling preemptive jobs online to meet deadlines. In this problem we again have n jobs to schedule on m machines. However, each job J_i arrives at time r_i and has a deadline d_i. The relative deadline of the job is then $D_i = d_i - r_i$. Each job also has a profit or weight p_i; this amount of profit is gained by the scheduler when the job is completed by its deadline. The *throughput* of a schedule is defined as the total profit of the jobs completed by their deadlines and the goal of the scheduler is to maximize the profit it obtains.

After the throughput result, we will also study a generalization of throughput called the *general profit* scheduling problem. In this problem, each job J_i is associated with a function $p_i(t)$ which specifies the profit obtained for finishing job J_i at $r_i + t$. It is assumed that p_i can be different for each job J_i but the functions are arbitrary non-increasing functions; a scheduler should not obtain more profit for delaying a job.

As expected, we are working in the directed acylic graph (DAG) model of parallelism. We will give the *first* non-trivial results for scheduling DAG jobs online to maximize throughput and then generalize these results to the general profit problem. To formalize these results, recall that there are two important parameters in the DAG model: the critical-path length C_i of job J_i (its execution time on an infinite number of processors) and its total work W_i (its uninterrupted execution time on a single processor). The value of $\max\{C_i, W_i/m\}$ is a lower bound on the amount of time any 1-speed scheduler takes to complete job J_i on m cores.

We will focus on schedulers that are aware of the values of C_i and W_i when the job arrives, but are unaware of the internal structure of the job's DAG. That is, besides C_i and W_i, the only other information a scheduler has on a job's DAG is which nodes are currently available to execute. For DAG tasks, this is a reasonable model for the real world programs written in languages mentioned above since the DAG generally unfolds dynamically as the program executes. We call such an algorithm *semi-non-clairvoyant* [5].

Even with a semi-non-clairvoyant scheduler, we can prove the following lower bound.

Theorem 11. *In the DAG model, there exists inputs where any semi-non-clairvoyant scheduler requires speed augmentation of* $2-1/m$ *to be* $O(1)$*-competitive for maximizing throughput.*

Roughly speaking, scheduling even a single DAG job in time smaller than $\frac{W_i - C_i}{m} + C_i$ is a hard problem even offline when the entire job structure is known in advance. This is captured by the classic problem of scheduling a precedence constrained jobs to minimize the makespan. For this problem, there is no $2 - \epsilon$ approximation assuming a variant of the unique games conjecture [44]. In Section 6.3, we will give an example DAG where any semi-non-clairvoyant scheduler will take roughly $\frac{W_i - C_i}{m} + C_i$ time to complete while a fully clairvoyant scheduler can finish in time W_i/m. By setting the relative deadline to be $D_i = W_i/m = C_i$, every

[5]It is also very difficult to optimize throughput if the scheduler does not know these quantities

semi-non-clairvoyant scheduler will require a speed augmentation of $2-1/m$ to have bounded competitiveness because otherwise it will complete no jobs.

With the previous lower bound in place, we cannot hope for a $(1+\epsilon)$-speed $O(1)$-competitive algorithm. To circumvent this hurdle, one could hope to show $O(1)$-competitiveness by either using more resource augmentation or by making an assumption on the input. Intuitively, the hardness of the problem comes from having a relative deadline D_i that is very close to $\max\{C_i, W_i/m\}$ because then no scheduler can finish the job without resource augmentation. In practice, jobs rarely have such tight deadlines. We show that so long as $D_i \geq (1+\epsilon)(\frac{W_i-C_i}{m}+C_i)$ then there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm.

Theorem 12. *If for every job J_i it is the case that $(1+\epsilon)(\frac{W_i-C_i}{m}+C_i) \leq D_i$, then there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.*

We note that this immediately implies the following corollary, which uses $(2+\epsilon)$ speed augmentation but has no assumptions on the input.

Corollary 6.0.1. *There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.*

Proof. No schedule can finish a job J_i if its relative deadline is smaller than $\max\{C_i, \frac{W_i}{m}\}$ and we may assume that no such job exists. Using this, we have that $(\frac{W_i}{m}+C_i) \leq 2D_i$. Consider transforming the problem instance giving the algorithm *and* the optimal solution together $2+\epsilon$ speed. In this case, the condition of Theorem 12 is met since we can view this as scaling the work in each node of the jobs by $2+\epsilon$. This scales both the work and critical-path length by $2+\epsilon$. The corollary follows by observing that in this case we are comparing to an optimal solution with $2+\epsilon$ speed which is only stronger an optimal solution with 1 speed. □

We note that the theorem also immediately implies the following corollary for "reasonable jobs."

Corollary 6.0.2. *There is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive for maximizing throughput if $(W_i - C_i)/m + C_i \leq D_i$ for all jobs J_i.*

Note that this theorem uses speed augmentation but has a tighter assumption on the deadlines than in theorem 12. This assumption on the deadlines is reasonable since, as we show in Section 6.3, there exists inputs for which even the best semi-non-clairvoyant scheduler has unbounded performance if the deadline is tighter.

Later on, we will consider the general profit scheduling problem. We will first make the following assumption, similar to the assumption in the throughput problem: all jobs J_i has a general profit function which satisfies $p_i(d) = p_i(x_i^*)$, where $0 < d \leq x_i^*$ for some $x_i^* \geq (1+\epsilon)(\frac{W_i - C_i}{m} + C_i)$. This assumption states that there is no additional benefit for completing a job J_i before time x_i^*. This is the natural generalization of our assumption for throughput case since it basically means that the algorithm can take up to time x_i^* to finish the job. The function is arbitrarily decreasing otherwise. Using this, we show the following.

Theorem 13. *If for every job J_i it is the case that $p_i(d) = p_i(x_i^*)$, where $0 < d \leq x_i^*$ for some value of $x_i^* \geq (1+\epsilon)(\frac{W_i - C_i}{m} + C_i)$ then there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm for the general profit objective.*

This gives the following corollary immediately, just as for throughput, which removes the assumption but uses speed augmentation.

Corollary 6.0.3. *There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing general profit.*

We will begin by describing the constraints on the jobs and the algorithm in section 6.1 and 6.2.1. Then we will analyze the algorithm in the deadlines case, after which we will give the example which requires $(2+\epsilon)$ speed for any semi-non-clairvoyant scheduler in section 6.3. The general cost function problem will follow in section 6.4.

6.1 Preliminaries

In the problem considered, there is a set $\mathcal{J}$ of n jobs $\{J_1, J_2, ..., J_n\}$ which arrive online. The jobs are scheduled on m identical processors. Job J_i arrives at time r_i. Let $p_i(t)$ be an arbitrary non-negative non-increasing function for job J_i. The value of $p_i(t)$ is the profit obtained by completing job i at time $r_i + t$. Under some schedule, let t_i be the time it takes to complete J_i after its arrival. The goal is for the scheduler to maximize $\sum_{i\in[n]} p_i(t_i)$.

Scheduling jobs with deadlines is a special case of this problem. In the deadlines problem, each job J_i has a deadline d_i and the scheduler obains a profit of p_i if it is completed by this time. Here, we will let $D_i = d_i - r_i$ be the relative deadline of the job. To make the underlying ideas of our approach clear, we will first focus on proving this case and the more general problem will be later, in section 6.4.

Each job is represented by a Directed-Acyclic-Graph (DAG) as in the previous chapters. A node in the DAG is *ready* to execute if all its predecessors have completed. A job is *completed* only when *all* nodes in the job's DAG have been processed. The scheduler knows the ready nodes for a job at any point in time, but does not know the entire DAG structure.

A DAG job has two important parameters. The total *work* W_i is the sum of the processing time of the nodes in job i's DAG. The *span* or *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. Because there are costs and profits in this problem, we will use the notation L_i instead of C_i to refer to the critical path length of a job in order to avoid confusion. Using this notatation, to show Theorem 12 we assume that $(1+\epsilon)(\frac{W_i - L_i}{m} + L_i) \leq D_i$ for all jobs J_i. This assumption shall be maintained throughout this chapter.

6.2 Jobs with Deadlines

First, we give an algorithm and analysis proving Theorem 12, which is the throughput problem where jobs have deadlines and profits. There are a lot of notation necessary for this algorithm and they can be found in Tables 6.1, 6.2 and 6.3. Throughout the proof, we let C^O denote the jobs that the optimal solution completes by their deadline and let $\left\|C^O\right\|$ denote the total profit obtained by the optimal solution. Our goal is to design a scheduler that achieves profit close to $\left\|C^O\right\|$. Throughout the proof, it will be useful to discuss the aggregate number of processors assigned to a job over all time. We define a *processor step* to be a unit of time on a single processor.

6.2.1 Algorithm

In this section, we introduce our algorithm S. On every time step, S must decide which jobs to schedule and which ready nodes of each job to schedule. When a job J_i arrives, S calculates n_i — the number of processors "allocated" to J_i. On any time step when S

decides to run J_i, it will always allocate n_i processors to J_i. In addition, since S is semi-non-clairvoyant, it is unable to distinguish between ready nodes of J_i; when it decides to allocate n_i nodes to J_i, it arbitrarily picks n_i ready nodes to execute if more than n_i nodes are ready.

We first state some observations regarding work and critical-path length.

Observation 5. *If a job J_i has all of its r ready nodes being executed by a schedule with speed s on m processors, where $r \leq m$, then the remaining critical-path length of J_i decreases at a rate of s.*

We have made the above observation for previous results in the DAG model.

As mentioned earlier, we also assume that the deadline for each job follows the condition that $(1+\epsilon)(\frac{W_i - L_i}{m} + L_i) \leq D_i$ for some positive constant ϵ.

We define the following **constants**. Let $\delta < \epsilon/2$, $c \geq 1 + \frac{1}{\delta\epsilon}$ and $b = (\frac{1+2\delta}{1+\epsilon})^{1/2} < 1$ be fixed constants. For each job J_i, the algorithm calculates n_i as $\frac{(W_i - L_i)}{\frac{D_i}{1+2\delta} - L_i}$. The value of n_i is the number of processors our algorithm will give to job J_i if we decide to execute J_i on some time step.

Let $x_i := \frac{W_i - L_i}{n_i} + L_i$. By Observation 5 it is the case that if n_i processors are given to job i for x_i units of time then the job will be completed regardless of the order the nodes are executed in. This will be Observation 6.

Observation 6. *Job J_i can meet its deadline if it is given n_i dedicated processors for x_i time steps in the interval $[r_i, d_i]$.*

We define the *density* of a job as $v_i = \frac{p_i}{x_i n_i}$. Note that this is a non-standard definition of density. We define the density as $\frac{p_i}{x_i n_i}$ instead of $\frac{p_i}{W_i}$, because we will think of job i requiring

$x_i n_i$ processor steps to complete by Scheduler S. Thus, this definition of density indicates the potential profit per processor step that S can obtain by executing J_i.

The scheduler S maintains jobs that have arrived but are unfinished in two priority queues. A priority queue Q stores all the jobs that have been *started* by S. In this priority queue, the jobs are sorted according to the density from high to low. Another priority queue P stores all the jobs that have arrived but have not yet been started by S. Jobs in P are also sorted according to their densities from high to low.

Job Execution

At each time step t, S picks a set of jobs in Q to execute, in order from highest to lowest density. If a job J_i has been completed or if its absolute deadline d_i has passed ($d_i > t$), S removes the job from Q. When considering job J_i, if the number of unallocated processors is at least n_i the scheduler assigns n_i processors to J_i for execution. Otherwise, it continues on to the next job. S stops this procedure when either all jobs have been considered or when there are no remaining processors to allocate.

We introduce some notations to describe how jobs are moved from queue P to Q. A job J_i is δ-**good** if $D_i \geq (1+2\delta)x_i$. A job is δ-**fresh** at time t if $d_i - t \geq (1+\delta)x_i$. For any set T of jobs, let the set $A(T, v_1, v_2)$ contains all jobs in T with density within the range $[v_1, v_2)$. We define $N(T, v_1, v_2) = \sum_{J_i \in A(T, v_1, v_2)} n_i$. This is the total number of processors that S allocates to jobs in $A(T, v_1, v_2)$. We will say that the set of job $A(T, v_1, v_2)$ *requires* $N(T, v_1, v_2)$ processors.

Adding Jobs to Q

There are two types of events that may cause S to add a job to Q. These events can occur when either a job arrives or S completes a job. When a job J_i arrives, S adds it to queue Q if it satisfies the following conditions:

(1) J_i is δ-good;

(2) For all job $J_j \in Q \cup \{J_i\}$ it is the case that $N(Q \cup \{J_i\}, v_j, cv_j) \leq bm$. In words, the total number of processors required by jobs in $Q \cup \{J_i\}$ with density in the range $[v_j, cv_j)$ is no more than bm.

If these conditions are met, then J_i is inserted into queue Q; otherwise, job J_i is inserted into queue P (and remain un-started). When a job is added to Q, we say that the job is *started* by S.

At the completion of a job, S considers the jobs in P from highest to lowest density. S first removes all jobs with absolute deadlines that have already passed. Then S checks if a job J_i in P can be moved to queue Q by checking whether job J_i is δ-fresh and condition (2) from above. If both the conditions are met, then J_i is moved from queue P to queue Q.

Remark

Note that the Scheduler S pre-computes a fixed number of processors n_i assigned to each job, which may seem strange at first glance. However, this makes sense because n_i is approximately the minimum number of dedicated cores job J_i requires to complete by $\frac{D_i}{1+2\delta} \to D_i$, without knowing J_i's DAG structure. In addition, as long as J_i can complete by its deadline, it obtains the same profit p_i. Therefore, there is no need to complete J_i earlier by executing J_i on more dedicated cores. Moreover, by carefully assigning n_i, we are able to bound the number of processor steps spent on job J_i as shown in Lemma 6.2.3, which is critical for bounding the profit obtained by the optimal solution.

Analysis Outline

Our goal is to bound the total profit that S obtains. We first discuss some basic properties of S in Section 6.2.2. In Section 6.2.3 be bound the total profit of all the jobs S starts by the total profit of jobs that S completes. Then in Section 6.2.4 we bound the total profit of the jobs the optimal solution completes by the total profit of jobs that S starts. Putting these two together, we are able to bound the performance of S.

6.2.2 Properties of the Scheduler

We begin by showing some structural properties for S that we will leverage in the proof. We first bound the number of processors n_i that S will allocate to job J_i.

Lemma 6.2.1. *For every job J_i, the following holds: $n_i \leq b^2 m$.*

Proof. By assumption we know that $D_i \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$

The definition of n_i gives the following.

$$n_i = \frac{W_i - L_i}{\frac{D_i}{1+2\delta} - L_i} \leq \frac{W_i - L_i}{\frac{1+\epsilon}{1+2\delta}(\frac{W_i - L_i}{m} + L_i) - L_i} \leq \frac{1+2\delta}{1+\epsilon} m = b^2 m$$

□

Lemma 6.2.2. *Every job J_i is δ-good, i.e. $x_i(1+2\delta) \le D_i$.*

Proof. Note that $L_i \le \frac{1}{1+\epsilon}D_i$ by assumption. Since $n_i = \frac{W_i - L_i}{\frac{D}{1+2\delta} - L_i}$, we have $x_i(1+2\delta) = (\frac{W_i - L_i}{n_i} + L_i)(1+2\delta) = (\frac{D_i}{1+2\delta} - L_i + L_i)(1+2\delta) \le D_i$. □

This next lemmas bounds the total number of processor steps occupied by a job.

Lemma 6.2.3. *$x_i n_i \le aW_i$, where a is $1 + \frac{1+2\delta}{\epsilon - 2\delta}$.*

Proof. By definition we have

$$\begin{aligned} x_i n_i = W_i - L_i + n_i L_i \le W_i + \frac{W_i - L_i}{\frac{D_i}{1+2\delta} - L_i} L_i \le W_i + \frac{W_i - L_i}{\frac{D_i}{1+2\delta} - \frac{D_i}{1+\epsilon}} \Big(\frac{D_i}{1+\epsilon}\Big) \\ \le W_i + \frac{(W_i - L_i)D_i(1+2\delta)}{D_i(\epsilon - 2\delta)} \le W_i + \frac{W_i(1+2\delta)}{\epsilon - 2\delta} \le W_i\Big(1 + \frac{1+2\delta}{\epsilon - 2\delta}\Big) \end{aligned}$$

□

Observation 7. *At any time and for any $v > 0$, the total number of processors required by all the jobs J_i that are in queue Q and have density $v \le v_i < cv$ is no more than bm, i.e. $N(Q, v_i, cv_i) \le bm$.*

Proof. Jobs are only added to queue Q when a new job arrives or a job completes. According to algorithm S, at both times, a job is only added to Q when this condition is satisfied. □

Now we are ready to begin the first part of the proof.

6.2.3 Bounding the Profit of Jobs S Completes

In this section, we bound the profit of jobs completed by S compared to the profit of all jobs it ever starts (adds to Q). Let R denote the set of jobs S starts (that is, the set of jobs added to queue Q). Among the jobs in R, let C be the set of jobs it completes and U be the set of jobs that it does not complete. We say job J_i (and its assigned processors) is **v-dense** for some given density v, if job J_i has density $v_i \geq v$. For any set A of jobs, define $\|A\|$ as $\sum_{i \in A} p_i$, the sum of the profits of jobs in the set.

Lemma 6.2.4. *For a job $J_i \in U = R \setminus C$ that was added to queue Q but does not complete by its deadline, S must have run cv_i-dense jobs for at least δx_i time steps where J_i is in Q using at least $(1-b)m$ processors at each such time.*

Proof. Since J_i is at least δ-fresh when added to Q and it does not complete by its deadline, there are at least δx_i time steps where S is not executing J_i because of Observation 6. In each of these the time steps, all the m processors must be executing v_i-dense jobs.

By Observation 7, jobs in Q with density in range $[v_i, cv_i)$ require at most $N(Q, v_i, cv_i) \leq bm$ processors to execute. Therefore, for each of the δx_i time steps, there are at least $(1-b)m$ processors executing jobs which are cv_i-dense. So the total number of processor steps where cv_i-dense jobs are executing is at least $\delta x_i(1-b)m$. □

We now bound the profit of the jobs completed by their deadline under S by those jobs which are started.

Lemma 6.2.5. $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|R\|$.

Proof. To prove this lemma, we will use a charging scheme with credit transfers between the jobs. We give each job $J_i \in R$ a bank account B_i. Initially, all completed jobs (in C) are given p_i credits and other jobs (in U) have 0 credit. We will transfer credits between the jobs in C and the jobs in U. We want to show that after the credit transfer, every job J_i in R will have $B_i \geq (\epsilon - \frac{1}{(c-1)\delta})p_i$. This implies $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta})\,\|R\|$.

Now we explain how credits are transferred. For each time step, a processor executing J_i will transfer $\frac{v_j n_j}{\delta bm}$ credits from B_i to every job J_j in queue Q that has density $v_j \leq \frac{v_i}{c}$.

For every job $J_j \in U$, Lemma 6.2.4 implies that there are at least δx_j time steps where at least $(1-b)m$ processors are executing cv_j-dense jobs. By our credit transfer strategy J_j will receive at least $\frac{v_j n_j}{\delta bm}$ credits from each processor in a time step. Therefore, the total credits J_j receives is at least

$$\delta x_j(1-b)m(\frac{v_j n_j}{\delta bm}) = v_j x_j n_j(\frac{1-b}{b}) = p_i(\frac{1-b}{b}).$$

This bounds the total amount of credit each job receives. We now show that not too much credit is transferred out of each job's account. We bound this on a job by job basis. Fix a job $J_i \in R$ and consider how many credits it transfers to other jobs during its execution. By Observation 6, we know that J_i can execute for at most x_i time steps on n_i dedicated processors before its completion.

The job J_i will also transfer credit away to all jobs in Q with density less than $\frac{v_i}{c}$ at any point in time where J_i is being processed. These are the jobs in $A(Q, 0, \frac{v_i}{c})$. Let us fix an integer $l \geq 1$ and consider the set of jobs $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ in Q that have density within the range $[\frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$.

Note that the total number of processors required by them is $N(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l}) \leq bm$ by Observation 7. Knowing that a job J_j in $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ has density $v_j \leq \frac{v_i}{c^l}$ by definition, we can understand that the total credits job J_i gives to jobs in $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ per processor assigned to J_i during any time step is at most the following:

$$\sum_{J_j \in A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} \frac{v_j n_j}{\delta bm} \leq \sum_{J_j \in A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} \frac{\frac{v_i}{c^l} n_j}{\delta bm} = \frac{v_i}{\delta bmc^l} \sum_{J_j \in A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} n_j$$
$$= \frac{v_i}{\delta bmc^l} N(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l}) \leq \frac{v_i}{\delta bmc^l} bm = \frac{v_i}{\delta c^l}.$$

This bounds the total credit transferred to jobs in $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ during a time step for each processor assigned to J_i. We sum this quantity over all $l \geq 1$ and all n_i processors assigned to i to bound the total credit transferred away from job J_i during a time step. In this calculation, recall that $c > 1$ by definition.

$$\frac{n_i v_i}{\delta} \sum_{l=1}^{\infty} \frac{1}{c^l} = \left(\frac{n_i v_i}{\delta}\right) \frac{\frac{1}{c}}{1 - \frac{1}{c}} = \left(\frac{n_i v_i}{\delta}\right) \frac{1}{c-1}$$

Therefore, the total credits J_i transfers to all the jobs in $A(Q, 0, \frac{v_i}{c})$ over all times wheere it is executed is at most $(\frac{x_i n_i v_i}{\delta})\frac{1}{c-1} = \frac{p_i}{(c-1)\delta}$ due to the fact that a job will be executed for at most x_i time steps in S's schedule.

Now we put will put these two observations together. Each job J_i receives at least $p_i \frac{1-b}{b}$ credit and pays at most $\frac{p_i}{(c-1)\delta}$. After the credit transfer, the credits that a job J_i has is at least the following:

$$p_i \frac{1-b}{b} - \frac{p_i}{(c-1)\delta} = p_i(\epsilon - \frac{1}{(c-1)\delta})$$

By our choice of c, this quantity is always positive. Therefore, we conclude that $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|R\|$. This completes the proof of the lemma. $\square$

6.2.4 Bounding the Profit of Jobs OPT Completes

In this section, we bound the profit of the jobs OPT completes by all of the jobs that S starts. Our high level goal is to first bound the total amount of time OPT spends processing jobs that S does not complete by the time that S spends processing jobs. Then using this and properties of S we will be able to bound the total profit of jobs OPT completes. At a high level, this works since S focuses on processing high density jobs and OPT and S both spend a similar amount of time processing jobs. We will begin by showing that if not too many processors are executing $\frac{v_i}{c}$-dense jobs then all such jobs must be currently executing.

Lemma 6.2.6. *For any density v_i and time, if there are less than $b(1-b)m$ processors executing $\frac{v_i}{c}$-dense jobs, then all $\frac{v_i}{c}$-dense jobs in queue Q are executing and $N(Q, \frac{v_i}{c}, \infty) < b(1-b)m$.*

Proof. By definition, there are at least $m - b(1-b)m > bm - b(1-b)m = b^2m$ processors executing jobs with density less than $\frac{v_i}{c}$. For the sake of contradiction, suppose there is a $\frac{v_i}{c}$-dense job J_j that is not being executed in S. By Lemma 6.2.1 we know that $n_j \leq b^2m$. Therefore, J_j would have been executed by S on the b^2m processors that are executing lower density jobs, this a contradiction.

Now we know all $\frac{v_i}{c}$-dense jobs in queue Q are executing. By assumption they are using less than $b(1-b)m$ processors. Therefore the lemma follows. $\square$

In the next lemma, we show that if not too many processors are running $\frac{v_i}{c}$-dense jobs then when a job arrives or completes, the schedule S will start processing a v_i-dense job that is δ-fresh, for any density v_i (if such a job exists). In particular, the job J_j will pass condition (2) of for adding jobs to Q in the definition of S.

Lemma 6.2.7. *Consider a fixed density v_i. At a time where a new job arrives or a job completes if there are less than $b(1-b)m$ processors executing $\frac{v_i}{c}$-dense jobs, then a δ-fresh v_i-dense job J_j (arriving or in queue P) will be added to Q by S, assuming such a job J_j exists.*

Proof. By Lemma 6.2.6, we know that all $\frac{v_i}{c}$-dense jobs in queue Q are executing on less than $b(1-b)m$ processors. By Lemma 6.2.1, we know that $n_j \leq b^2m$. Therefore,

$$N(Q \cup \{J_j\}, \frac{v_i}{c}, \infty) < b(1-b)m + b^2m = bm$$

Consider any δ-fresh job J_j that is also v_i-dense. Consider any job J_k where $J_j \in A(Q \cup \{J_i\}, v_k, cv_k)$. By definition of J_j being v_i-dense it must be the case that $A(Q \cup \{J_i\}, v_k, cv_k) \subseteq A(Q \cup \{J_j\}, \frac{v_i}{c}, \infty)$. The above implies that $N(Q \cup \{J_i\}, v_k, cv_k) \leq N(Q \cup \{J_j\}, \frac{v_i}{c}, \infty) \leq bm$. Thus, the condition (2) in our algorithm is satisfied. □

For an arbitrary set of jobs $\mathcal{E}$ and any $v \geq 0$, we let $T_O(v, \mathcal{E})$ denote the total work processed by the optimal schedule for the jobs in $\mathcal{E}$ that are v-dense. We similarly let $T_S(v, \mathcal{E})$ be the total number of processors steps S used for executing jobs in $\mathcal{E}$ that are v-dense over all time. Now we are ready to bound the time that OPT spends on jobs that S never adds to Q.

Lemma 6.2.8. *Consider the jobs in $\mathcal{J} \setminus R$, the jobs that are never added to Q (never started). For all $v > 0$, $T_O(v, \mathcal{J} \setminus R) \leq \frac{1+2\delta}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$.*

Proof. Let $\{I_k = [s_k, e_k]\}$ be the set of maximal time intervals where at least $b(1-b)m$ processors are running $\frac{v}{c}$-dense jobs in S's schedule. Notice that by definition $\sum_{k=1}^{\infty}(e_k - s_k)b(1-b)m \leq T_S(\frac{v}{c}, \mathcal{J})$.

Consider a job in $J_i \in \mathcal{J} \setminus R$ that is both δ-good and v-dense and additionally arrives during $[s_k, s_{k+1})$. Note that during the intervals $[e_k, s_{k+1}]$, less than $b(1-b)m$ processors are executing $\frac{v}{c}$-dense jobs. Lemma 6.2.7 implies that if J_i arrives during $[e_k, s_{k+1}]$ it will be added to Q. This contradicts the assumption that $J_i \in \mathcal{J} \setminus R$. Therefore, J_i must arrive during $[s_k, e_k)$ and is in queue P at time e_k.

Note that at time e_k, the number of processors executing $\frac{v}{c}$-dense jobs decreases, so there must be a job that completes at time e_k. Again, by Lemma 6.2.7 if J_i is δ-fresh at time e_k then it will be added to Q at this time. Again, this contradicts $J_i \in \mathcal{J} \setminus R$. Thus, the only reason that S does not add J_i to Q is because J_i is not δ-fresh at time e_k. Knowing that J_i is δ-good at r_i and is not δ-fresh at e_k, we have $e_k - s_k \geq e_k - r_i \geq \delta x_i$.

So at time e_k, J_i is not δ-fresh. So $d_i - e_k < (1+\delta)x_i < \frac{1+\delta}{\delta}(e_k - s_k)$.

Let K_k be the set of v-dense jobs that arrive during $[s_k, s_{k+1})$ but are not completed by S. Because OPT can only execute all jobs in K_k during $[s_k, d_i]$ with at most m processors, we can show the following:

$$T_O(v, K_k) \leq (d_i - s_k)m = ((d_i - e_k) + (e_k - s_k))m \leq \frac{1+2\delta}{\delta}(e_k - s_k)m$$

This completes the proof after the following calculations.

$$T_O(v, U) = \sum_{k=1}^{\infty} T_O(v, K_k) \le \sum_{k=1}^{\infty} (\frac{1+2\delta}{\delta}) m(e_k - s_k) \le \frac{1+2\delta}{\delta} \frac{1}{b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$$

$\square$

Using the previous lemma, we can bound the profit of jobs completed by OPT by the profit of jobs started by S.

Lemma 6.2.9.

$$\left\|C^O\right\| \le \left(1 + (1 + \frac{1+2\delta}{\epsilon - 2\delta})(1 + \frac{1}{\epsilon\delta}) \frac{1+2\delta}{\delta b(1-b)}\right) \|R\|$$

.

Proof. We may assume WLOG that OPT completes all jobs it starts. First we partition C^O, the jobs that OPT completes, into C_R^O and C_S^O where $C_S^O = C^O \cap R$, that is, the set of jobs that our algorithm started at some point. The remaining jobs are placed in C_R^O. Clearly $\left\|C_S^O\right\| \le \|R\|$. Now it remains to bound $\left\|C_R^O\right\|$.

Consider every job in C_R^O and let the set of densities of these jobs be $\{\mu_1, \mu_2, \ldots, \mu_m\}$ from high to low. For notational simplicity let $\mu_0 = \infty$ and $\mu_{m+1} = 0$. Recall that OPT completed all jobs it started. Thus for each job with density μ_i, it ran the job for a corresponding W_i processor steps. Let β_i denote the number of processor steps our algorithm takes to run jobs with densities within $(\frac{\mu_{i-1}}{c}, \frac{\mu_i}{c}]$.

We have $T_O(v, \mathcal{J} \setminus R) \leq \frac{1+2\delta}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$ from Lemma 6.2.8 for all densities v. Equivalently, for any given density v, we have the following.

$$T_O(v, \mathcal{J} \setminus R) = \sum_{i=1}^{v} W_i \leq \frac{1+2\delta}{\delta b(1-b)} \sum_{i=1}^{v} \beta_i = \frac{1+2\delta}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$$

We then sum over all densities. The subtraction of densities is necessary to insure that each density is only counted a single time.

$$\sum_{v=1}^{m} \left((\mu_v - \mu_{v+1}) \sum_{i=1}^{v} W_i \right) \leq \sum_{v=1}^{m} \left((\mu_v - \mu_{v+1}) \frac{1+2\delta}{\delta b(1-b)} \sum_{i=1}^{v} \beta_i \right)$$

The left hand side can be simplified:

$$\sum_{v=1}^{m} \left((\mu_v - \mu_{v+1}) \sum_{i=1}^{v} W_i \right) = \sum_{i=1}^{m} W_i \sum_{v=i}^{m} (\mu_v - \mu_{v+1}) = \sum_{i=1}^{m} W_i (\mu_i - \mu_{m+1}) = \sum_{i=1}^{m} W_i \mu_i$$

The right hand side similarly simplifies to $\frac{1+2\delta}{\delta b(1-b)} \sum_{i=1}^{m} \beta_i \mu_i$, leading to the inequality that $\sum_{i=1}^{m} W_i \mu_i \leq \frac{1+2\delta}{\delta b(1-b)} \sum_{i=1}^{m} \beta_i \mu_i$. Recall that densities such as μ_i are defined by $\mu_i = \frac{p_i}{x_i n_i}$ and $x_i n_i \leq a W_i$. Therefore:

$$\sum_{i=1}^{m} W_i \mu_i = \sum_{i=1}^{m} \frac{W_i p_i}{x_i n_i} \geq \sum_{i=1}^{m} \frac{W_i p_i}{a W_i} \geq \sum_{i=1}^{m} \frac{p_i}{(1 + \frac{1+2\delta}{\epsilon - 2\delta})} = \frac{1}{(1 + \frac{1+2\delta}{\epsilon - 2\delta})} \|C_R^O\|$$

And also, by the definition of β_i, we know that $\sum_{i=1}^{m} \beta_i \frac{\mu_i}{c} \leq \|R\|$.

We combine these expressions to obtain the following.

$$\frac{1}{(1+\frac{1+2\delta}{\epsilon-2\delta})}\left\|C_R^O\right\| \leq \sum_{i=1}^{m} W_i\mu_i \leq \frac{1+2\delta}{\delta b(1-b)}\sum_{i=1}^{m}\beta_i\mu_i \leq \frac{1+2\delta}{\delta b(1-b)}c\left\|R\right\|$$
$$\Rightarrow \left\|C_R^O\right\| \leq \left(1+\frac{1+2\delta}{\epsilon-2\delta}\right)\left(\frac{1+2\delta}{\delta b(1-b)}\right)c\left\|R\right\|$$
$$\Rightarrow \left\|C^O\right\| = \left\|C_R^O\right\| + \left\|C_S^O\right\| \leq \left(1+(1+\frac{1+2\delta}{\epsilon-2\delta})(1+\frac{1}{\epsilon\delta})\frac{1+2\delta}{\delta b(1-b)}\right)\left\|R\right\|$$

□

Finally we are ready to complete the proof of bounding the profit OPT obtains by the total profit the algorithm obtains for jobs it completed.

Lemma 6.2.10.

$$\left\|C^O\right\| \leq \frac{\left(1+(1+\frac{1+2\delta}{\epsilon-2\delta})(1+\frac{1}{\epsilon\delta})\frac{1+2\delta}{\delta b(1-b)}\right)}{\epsilon-\frac{1}{(c-1)\delta}}\left\|C\right\|$$

Proof. The proof of this lemma is simply through the combination of Lemma 6.2.5 and Lemma 6.2.9. □

Therefore, we prove Theorem 12 by showing that scheduler S is $O(\frac{1}{\epsilon^6})$-competitive for jobs with deadlines and profits, when $(1+\epsilon)(\frac{W_i-L_i}{m}+L_i) \leq D_i$.

In the next section we will show that any semi-non-clairvoyant scheduler must have roughly 2 speed if it is competitive for throughput unless it makes some assumptions on the deadlines of jobs.

Figure 6.1: Additional Notation

OPT	optimal schedule and also optimal objective
m	the number of processors
W_i	the total work of job J_i
L_i	the span of job J_i
D_i	relative deadline of job J_i
r_i	the arrival time of J_i
d_i	the absolute deadline of J_i (that is, $r_i + D_i$)
$A(T, v_1, v_2)$	all jobs in T with density within the range $[v_1, v_2)$
$N(T, v_1, v_2)$	$= \sum_{J_i \in A(T,v_1,v_2)} n_i$, the total number of processors required by $A(T, v_1, v_2)$
v-dense	if Job J_i has density $v_i \geq v$
δ	$< \epsilon/2$
c	$\geq 1 + \frac{1}{\epsilon\delta}$
b	$= (\frac{1+2\delta}{1+\epsilon})^{1/2} < 1$
a	$= 1 + \frac{1+2\delta}{\epsilon-2\delta}$

Figure 6.2: Additional Notation for Throughput

p_i	the profit of job J_i
n_i	$= \frac{(W_i - L_i)}{\frac{D_i}{1+2\delta} - L_i}$, the number of processors allocated to J_i
x_i	$= \frac{W_i - L_i}{n_i} + L_i$, the maximum execution time of J_i
v_i	$= \frac{p_i}{x_i n_i}$ the density of J_i
δ-good	job J_i has $D_i \geq (1 + 2\delta)x_i$
δ-fresh	at time t, job J_i has $d_i - t \geq (1 + \delta)x_i$
R	the set of jobs started by S
C	the set of jobs completed by S
U	unfinished jobs by S (that is, $R \setminus C$)
C^O	the set of jobs completed by OPT
$\mathcal{J}$	the set of all jobs
$T_O(v, \mathcal{E})$	the total work processed by the optimal schedule for the jobs in $\mathcal{E}$ that are v-dense
$T_S(v, \mathcal{E})$	the total number of processors steps S used for executing jobs in $\mathcal{E}$ that are v-dense

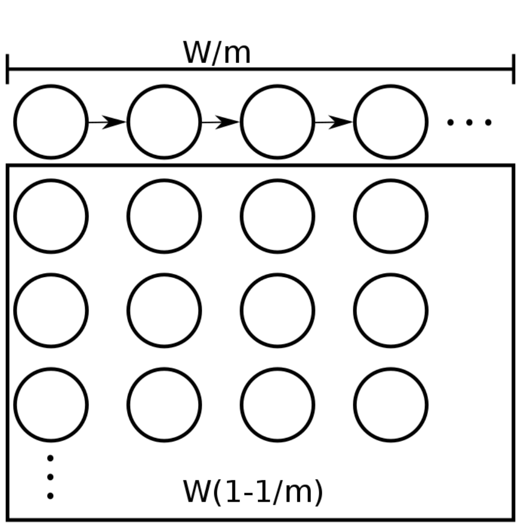
Figure 6.3: Additional Notation for General profit

$p_i(t)$	the profit of job J_i if the job with arrival time r_i completes by $r_i + t$
n_i	$= \frac{(W_i - L_i)}{\frac{x_i^*}{1+2\delta} - L_i}$, the number of processors allocated to J_i
x_i	$= \frac{W_i - L_i}{n_i} + L_i$, the maximum execution time of J_i
v_i	$= \frac{p_i(D_i)}{x_i n_i}$ the density of J_i

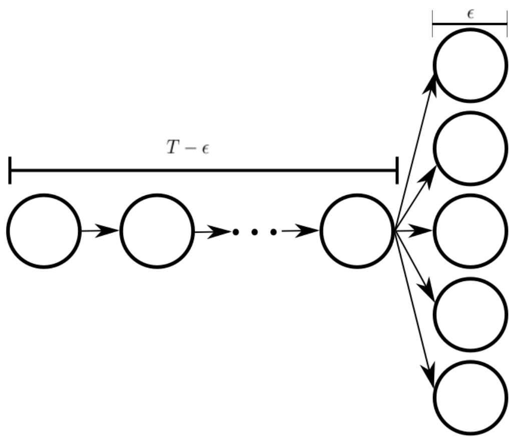
6.3 Examples

In this section, we will give some example DAGs to show why Theorem 12 is close to the best theorem we can hope for using two examples. The first example, shown in Figure 6.4a, shows the limitations of semi-non-clairvoyance. In particular, a semi-non-clairvoyant scheduler does not know the structure of the DAG in advance since the DAG unfolds dynamically. At any time step, the scheduler only knows the ready nodes available for execution. Given this limitation, consider the DAG shown in Figure 6.4a. This job has one sequential chain with length $L = \frac{W}{m}$, where W is the total work of the job and m is the number of processors. The remaining $W - W/m$ work are fully parallelizable in a block and can also be done in parallel with the chain. Therefore, L is the span of the jobs.

Since a semi-non-clairvoyant scheduler cannot distinguish between ready nodes, it may make unlucky choices and execute the entire block of $W - W/m = W - L$ ready nodes first in $(W - L)/m$ time steps and then execute the chain of L nodes sequentially — leading to a total time of $(W - L)/m + L$. On the other hand, a fully clairvoyant scheduler can execute the entire DAG in W/m time. Therefore, a semi-non-clairvoyant scheduler needs at least $2 - 1/m$ speed augmentation to ensure that it can complete the DAG at the same time as OPT.



(a) Non-clairvoyance limitation



(b) Difficult DAG

Figure 6.4: Two Examples

We now show another example DAG indicating that it would be reasonable to always set deadlines as $D \geq (W - L)/m + L$ if we do not know the structure of the DAG a priori. Figure 6.4b shows an example DAG, which consists of a chain of $L - \epsilon$ nodes followed by $W - L + \epsilon$ nodes that can run in parallel. Each node in the DAG takes ϵ time to run, so the total work of the DAG is W and the span is L. For such a DAG, even a fully clairvoyant scheduler needs $L - \epsilon + \frac{W-L+\epsilon}{m} = \frac{W-L}{m} + L - \epsilon(1 - \frac{1}{m})$, which approaches to $\frac{W-L}{m} + L$ when $\epsilon \to 0$.

6.4 Jobs with General Profit Functions

In this section, we will focus on a more general problem. In particular, each job J_i has a non-negative non-increasing profit function $p_i(t)$ indicating its profit if the job with arrival time r_i completes by $r_i + t$. Our goal is to design a scheduler that maximizes the profit to make it close to what the optimal solution can obtain. That optimal profit will be denoted by $\|O\|$.

First, we present our scheduler S parameterized using a fixed constant $0 < \epsilon < 1$. Similar to Section 6.2.1, let $\delta < \epsilon/2$, $c \geq 1 + \frac{1}{\delta\epsilon}$ and $b = (\frac{1+2\delta}{1+\epsilon})^{1/2} < 1$ be fixed constants.

Upon the arrival of a job J_i, the scheduler S assigns a number of allocated cores n_i, a relative deadline D_i and a set of time steps I_i to J_i (according to the assignment procedure described below). For each time step t in I_i, we can say that J_i is assigned to t. Scheduler S always executes the highest density jobs that is assigned to t. If S decides to execute J_i in a time step, it will give n_i processors to J_i. Let $x_i := \frac{W_i - L_i}{n_i} + L_i$. We again define the *density* of a job as $v_i = \frac{p_i(D_i)}{x_i n_i} = \frac{p_i(D_i)}{W_i + (n_i - 1)L_i}$. We will now formally specify the algorithm of scheduler S for job assignment and execution.

subsubsection*Assigning cores, deadlines and slots When a job J_i arrives, the scheduler will assign a relative deadline D_i and a set of time steps I_i with n_i processors. These time steps are the only time steps in which J_i is allowed to run.

Recall (from Theorem 13) that we assume that the profit function stays the same until some value $x_i^* \geq (\frac{W_i - L_i}{m} + L_i)(1 + \epsilon)$. The number of assigned processors n_i is calculated as $n_i = \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - L_i}$.

The assignment for the deadline D_i is determined by searching all the potential deadlines D to find the minimum valid deadline. The set of time steps I_i is determined using the chosen deadline D_i.

For each *potential relative deadline* $D > (1+\epsilon)L_i$, the scheduler S checks whether it is a valid deadline through the following steps.

First, it selects a set of time steps I. It does so using the following process. Assuming D is assigned to J_i, then the density of J_i is $v = \frac{p_i(D)}{W_i+(n_i-1)L_i}$. For each time step t from r_i to $r_i + D$, let $\|I(t)\|$ be the number of time steps that have already been added to I before considering time step t. Let $J(t)$ denote the set of jobs that are currently has time t among its assignments. We only add t to the set I if it satisfies the following condition about the density of jobs which have t assigned: For every job $J_j \in J(t)$, $N(J(t) \cup \{J_i\}, v_j, cv_j) \le bm$. In words, the total number of processors required by jobs in $J(t) \cup \{J_i\}$ with density in the range $[v_j, cv_j)$ is no more than bm.

I contains all the time steps during $[r_i, r_i + D_i)$ that can be assigned to J_i. If $\|I\| \ge (1+\delta)\left(\frac{W_i-L_i}{n_i} + L_i\right)$, which is at least δ times longer than the time J_i required to run on n_i processors, then the deadline D is said to be *valid*. A valid assignment always exists by setting the deadline large enough.

Among all the valid deadlines, S chooses the smallest valid deadline for J_i. This selection results in the highest profit. Given this deadline D_i, J_i will be assigned with the corresponding set I_i. Because D_i is the minimum valid deadline, the corresponding set I_i must satisfy $\|I_i\| = (1+\delta)\left(\frac{W_i-L_i}{n_i} + L_i\right)$; otherwise, there must exist a shorter deadline D that is also valid. Intuitively, with this assignment, J_i can complete by its deadline if no other jobs interfere. Note that J_i may not be completed by its deadline as we will allow higher density jobs that arrive after J_i to be scheduled during I_i. These higher density jobs might interrupt J_i.

Executing Jobs

At each time step t, S picks a set of jobs in $J(t)$ to execute in order from highest to lowest density, where $J(t)$ are the set of jobs that have been assigned to time step t. That is, jobs J_i where $t \in I_i$. When considering job J_i, if the number of unallocated processors is at least n_i, then the scheduler allocates n_i processors to J_i. Otherwise, it continues on to the next job in $J(t)$. S stops this procedure when either all jobs have been considered or when there are no remaining processors to allocate.

Remark

Unlike the scheduler for jobs with deadlines, here we try to complete a job J_i by a calculated deadline D_i that is as close to x_i^* as possible. This is because the obtained profit decreases as the completion time increases but there is no additional benefit for completing a job J_i before time x_i^*. With a carefully designed deadline D_i, we are able to prove the performance bound of the scheduler. Similar to Section 6.2, we start by stating the basic properties of the scheduler S, followed by bounding the total profit obtained by S.

However, the proofs that bound the profit of jobs that are completed by OPT differ greatly from that for jobs with deadlines. This is because in addition to losing the profit of jobs that do not complete by their assigned deadlines, scheduler S can also have less profit compared to OPT if the completion time of a job under S is later than under OPT. By taking into account all these jobs, we are still able to bound the performance of S for jobs with general profit functions.

6.4.1 Properties of the Scheduler

We begin by showing some structural properties for S that we will leverage in the proof and can be obtained directly from the algorithm of scheduler S. These lemmas are similar to the lemmas shown in Section 6.2.2 if we replace x_i* with D_i. We state them here again for completeness.

Lemma 6.4.1. *For every job J_i we have that $n_i \le b^2 m$, where $b = (\frac{1+2\delta}{1+\epsilon})^{1/2}$.*

Proof. By definition, we know that $x_i^* \ge (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$. Therefore, we have

$$n_i = \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - L_i} \le \frac{W_i - L_i}{\frac{1+\epsilon}{1+2\delta}(\frac{W_i - L_i}{m} + L_i) - L_i} \le \frac{1+2\delta}{1+\epsilon} m = b^2 m$$

□

Lemma 6.4.2. *Under scheduler S, we have $x_i n_i \le a W_i$ and $v_i \ge \frac{p_i(D_i)}{aW_i}$, where $a = 1 + \frac{1+2\delta}{\epsilon - 2\delta}$.*

Proof. By definition, $x_i^* > L_i(1+\epsilon)$. Therefore, we have

$$\begin{aligned} x_i n_i = W_i - L_i + n_i L_i = W_i + \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - L_i} L_i &\le W_i + \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - \frac{x_i^*}{1+\epsilon}} \Big(\frac{x_i^*}{1+\epsilon}\Big) \\ \le W_i + \frac{(W_i - L_i) x_i^* (1+2\delta)}{x_i^* (\epsilon - 2\delta)} \le W_i \Big(1 + \frac{1+2\delta}{\epsilon - 2\delta}\Big) \end{aligned}$$

Therefore, we have $v_i = \frac{p_i(D_i)}{x_i n_i} \ge \frac{p_i(D_i)}{aW_i}$. □

Lemma 6.4.3. *For every job J_i with the assignment n_i, D_i and I_i, Job J_i can meet its deadline D_i, if it is executed by S for at least x_i time steps in I_i (on n_i dedicated processors).*

Lemma 6.4.4. *For every job J_i, $x_i(1+2\delta) \leq x_i^*$.*

Proof. Note that $L_i \leq \frac{1}{1+\epsilon} D_i$ by requirement of potential assignment. Since $n_i = \frac{W_i - L_i}{\frac{x_i^*}{1+\epsilon} - L_i}$, we have $x_i(1+2\delta) = (\frac{W_i - L_i}{n_i} + L_i)(1+2\delta) \leq (\frac{x_i^*}{1+\epsilon} - L_i + L_i)(1+2\delta) = \frac{x_i^*}{1+\epsilon}(1+2\delta) \leq x_i^*$. □

Lemma 6.4.5. *At any time step t during the execution and for any density range $[v, cv)$, the total number of cores required by all the jobs $J_i \in J(t)$ (that have been assigned to t) with density $v \leq v_i < cv$ is no more than bm, i.e. $N(J(t), v_i, cv_i) \leq bm$.*

6.4.2 Bounding the Profit of Jobs S Completes

Similar to Section 6.2.3, we bound the profit of jobs completed by scheduler S compared to the profit of all jobs. Let $\mathcal{J}$ denote the set of jobs arrived during the execution, C denote the set of jobs that actually complete before their deadlines assigned by S, and $U = \mathcal{J} \setminus C$ be the set of jobs that didn't finish by their deadlines assigned by S. We say job J_i (and its assigned processors during execution) is v**-dense**, if its density $v_i \geq v$. For any set A of jobs, define $\|A\|$ as $\sum_{J_i \in A} p_i(D_i)$, the sum of the profits of jobs in the set under S.

Lemma 6.4.6. *For a job $J_i \in \mathcal{J} \setminus C$ that does not complete by its deadline, the number of time steps in I_i where S runs cv_i-dense jobs using at least $(1-b)m$ processors is at least δx_i.*

Proof. From Lemma 6.4.3, we know that job J_i can complete if it can execute for x_i time steps by S. Also note that according to the assignment process $(1+\delta)x_i = \|I_i\|$, where $\|I_i\|$ is the number of time steps assigned to J_i during $[r_i, r_i + D_i]$. Since it does not complete by its deadline, there are at least δx_i time steps in I_i where S does not execute J_i. Consider each of these time steps t. According to Lemma 6.4.5, jobs in $J(t)$ with density in range $[v_i, cv_i)$ require at most $N(J(t), v_i, cv_i) \leq bm$ processors to execute. Therefore, there must be at least $(1-b)m$ processors executing cv_i-dense jobs. Otherwise, S would execute all jobs in $A(J(t), v_i, cv_i)$, which includes job J_i. □

Lemma 6.4.7. $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|\mathcal{J}\|$.

The proof of this lemma uses a charging scheme and proceeds exactly like the proof of Lemma 6.2.5.

6.4.3 Bounding the Profit of Jobs OPT Completes

Similar to Section 6.2.4, we will now bound the profit of the jobs OPT completes. We are first going to consider the number of processor steps OPT spends on jobs that S finishes later than OPT. For these jobs, we can assume that S makes no profit at all since in the worst case, the profit function may become 0 as soon as OPT finishes it. Our high level goal is to first bound the total number of processor steps OPT spends on these jobs, which will allow us to bound OPT's profit. This section of the will differ greatly from the throughput proof.

We begin by showing that if not too many processors are executing $\frac{v_i}{c}$-dense jobs then all such jobs must be currently processed under S.

Lemma 6.4.8. *Consider a job J_i and a time $t^* < D_i$. For any time step $t \in [r_i, r_i + t^*] \setminus I_i$ (that is not added to I_i by S), the total number of processors required by $\frac{v_i}{c}$-dense jobs in $J(t)$ must be more than $b(1-b)m$, i.e., $N(J(t), \frac{v_i}{c}, \infty) > b(1-b)m$.*

Proof. Because $t \in [r_i, r_i + t^*] \setminus I_i$ and $t^* < D_i$, we know that time step t is before D_i.

Since t is not added to I_i, it must be the case that for some density $v_j \in (\frac{v_i}{c}, v_i]$, the required condition is not true, i.e., $N\left(J(t) \cup \{J_i\}, v_j, cv_j\right) > bm$. Note that v_j must be in the range $(\frac{v_i}{c}, v_i]$. This is because without assigning J_i to time step t it is true that $N\left(J(t), v_j, cv_j\right) \leq bm$ according to S, therefore J_i must have a density within the range of $[v_j, cv_j)$ in order to make impact.

By Lemma 6.4.1, we know that $n_i \leq b^2 m$. Therefore, we will have the following.

$$N\left(J(t), v_j, cv_j\right) = N\left(J(t) \cup \{J_i\}, v_j, cv_j\right) - n_i > bm - b^2 m = b(1-b)m$$

Therefore, we obtain $N(J(t), \frac{v_i}{c}, \infty) \geq N\left(J(t), v_j, cv_j\right) > b(1-b)m$. □

Let O be the set of jobs completed by OPT. For each job $J_i \in O$, let d be the difference between J_i's completion time and arrival time under OPT; the profit of J_i under OPT is $p_i(d)$. According to the assumption in Theorem 13, we know that if $d \leq x_i^*$, then $p_i(d) = p_i(x_i^*)$ for some $x_i^* \geq (\frac{W_i - L_i}{m} + L_i)(1+\epsilon)$. Therefore, we can assume that OPT assigns a relative deadline D_i^* to J_i, where $D_i^* = \max\{d, x_i^*\}$. Thus, OPT obtains a profit of $p_i(d) = p_i(D_i^*)$.

Lemma 6.4.9. *Consider a job J_i such that D_i assigned by scheduler S is larger than the deadline D_i^* assigned by* OPT*, i.e., $D_i > D_i^*$, the number of time steps during $[r_i, r_i + D_i^*)$ where scheduler S is actively executing $\frac{v_i}{c}$-dense jobs on at least $b(1-b)m$ cores is at least $\frac{\delta}{1+2\delta} D_i^*$.*

Proof. By definition of D_i^* and Lemma 6.4.4, we know that $D_i^* \geq x_i^*$.

Consider the number of time steps in time interval $[r_i, r_i + D_i^*]$ that are added to I_i, it must be less than $(1+\delta)\left(\frac{W_i - L_i}{n_i} + L_i\right) = (1+\delta)x_i$; otherwise, D_i^* would be a valid deadline under scheduler S with higher profit. Therefore, the number of time steps in $[r_i, r_i + D_i^*] \setminus I_i$ is more than $D_i^* - (1+\delta)x_i \geq D_i^* - \frac{1+\delta}{1+2\delta}x_i^* \geq D_i^* - \frac{1+\delta}{1+2\delta}D_i^* = \frac{\delta}{1+2\delta}D_i^*$.

By Lemma 6.4.8, we know that for each time step $t \in [r_i, r_i + D_i^*] \setminus I_i$, the total number of processors required by $\frac{v_i}{c}$-dense jobs in $J(t)$ must be more than $b(1-b)m$. Therefore, there must be at least $b(1-b)m$ cores executing $\frac{v_i}{c}$-dense jobs under scheduler S at time step t and the number of such steps is at least $\frac{\delta}{1+2\delta}D_i^*$. □

Among the jobs in O, let O_1 be the set of jobs that the deadline D_i assigned by scheduler S is no larger than the deadline set by OPT, i.e., $D_i \leq D_i^* < \infty$. In other words, the obtained profit of these jobs under scheduler S is no less than that under OPT, i.e., $p_i(D_i) \geq p_i(D_i^*)$, since the profit function $p_i(t)$ is non-increasing.

Let O_2 be the remaining jobs $O_2 = O \setminus O_1$. Let $\|X\|^*$ be the total profit that OPT obtains from jobs in X and $\|X\|$ be the total profit that S obtains from jobs in X. For jobs in O_1, we have $\|O_1\|^* \leq \|O_1\|$.

For an arbitrary set of jobs $\mathcal{E}$ and any $v \geq 0$ let $T_O(v, \mathcal{E})$ denote the total work processed by the optimal schedule for the jobs in $\mathcal{E}$ that are v-dense. Let β_i denote the total number of time steps where S is actively processing job J_i. By definition, we have $\beta_i \leq \frac{x_i}{1+\epsilon}$. We similarly let $T_S(v, \mathcal{E})$ be the summation of $\beta_i n_i$ over all jobs i in $\mathcal{E}$ that are v-dense. Note that this counts the total number of processor steps S executes jobs in $\mathcal{E}$ that are v-dense over all time.

Now we are ready to bound the time that OPT spends on jobs O_2 that scheduler S obtains less profit than OPT.

Lemma 6.4.10. *Consider a job J_i in O_2, the deadline D_i assigned by scheduler S is longer than deadline D_i^* assigned by* OPT*. For all $v > 0$, $T_O(v, O_2) \leq \frac{2(1+2\delta)}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$.*

Proof. For any job $J_i \in O_2$, we denote the lifetime of J_i under OPT as the time interval $[r_i, r_i + D_i^*)$, where D_i^* is the deadline assigned by OPT. For any density $v > 0$, let l be the number of time steps which make up the union of the lifetimes of all jobs in $A(O_2, v, \infty)$. By definition, $T_O(v, O_2) \leq lm$, since OPT can execute them on at most m processors.

Let $M \subseteq O_2$ be the minimum subset of O_2 that the union of the lifetimes of jobs in M covers the same time intervals of jobs in O_2. By the minimality of M, we know that at any time t, there are at most two jobs in M that cover time t. Therefore, we can further partition M into two sets M_1 and M_2, where for any two jobs in M_1 or any two jobs in M_2, their lifetimes do not overlap. By definition, either M_1 or M_2 has a union lifetime that is at least $l/2$. WLOG, we assume that it is M_1.

Consider $J_i \in M_1$ and let k_i be the number of time steps during its lifetime $[r_i, r_i + D_i^*)$ where scheduler S is actively executing $\frac{v_i}{c}$-dense jobs on at least $b(1-b)m$ cores. By Lemma 6.4.9, we know $k \geq \frac{\delta}{1+2\delta} D_i^*$. Therefore, during $[r_i, r_i + D_i^*)$ the number of processor steps where S is processing $\frac{v_i}{c}$-dense jobs is at least $b(1-b)m \frac{\delta}{1+2\delta} D_i^*$.

Let $K = \sum_{M_1} k_i$, be the total number of processor steps where S is processing $\frac{v}{c}$-dense jobs (since $v_i \geq v$) during the intervals in M_1. Thus, by definition,

$$K \geq \frac{\delta b(1-b)}{1+2\delta} m \sum_{J_i \in M_1} D_i^* > \frac{\delta b(1-b)}{1+2\delta} m \times \frac{l}{2} \geq \frac{\delta b(1-b)}{2(1+2\delta)} T_O(v, O_2)$$

Clearly, by adding additional intervals that are not in M_1, we have $T_S(\frac{v}{c}, \mathcal{J}) \geq K > \frac{\delta b(1-b)}{2(1+2\delta)} T_O(v, O_2)$, which gives us the bound. $\square$

Lemma 6.4.11.

$$\|O\|^* = \|O_1\|^* + \|O_2\|^* \leq \left(1 + (1 + \frac{1+2\delta}{\epsilon - 2\delta})(1 + \frac{1}{\epsilon\delta})\frac{2(1+2\delta)}{\delta b(1-b)}\right) \|\mathcal{J}\|$$

Proof. First, by the definition of O_1 and O_2, we have $\|O\|^* = \|O_1\|^* + \|O_2\|^*$ and $\|O_1\|^* \leq \|O_1\| \leq \|\mathcal{J}\|$. Now it remains to bound $\|O_2\|$.

We have $T_O(v, O_2) \leq \frac{2(1+2\delta)}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$ from Lemma 6.4.10 for all densities v. The remaining proof for the lemma is similar to that in Lemma 6.2.9, except for a different that will be involved. Therefore, $\|O_2\|^* \leq (1+\frac{1+2\delta}{\epsilon-2\delta})c\frac{2(1+2\delta)}{\delta b(1-b)} \|\mathcal{J}\|$. Taking the summation of $\|O_1\|^* + \|O_2\|^*$ completes the proof. $\square$

We are now ready to complete the proof and bound the profit OPT obtains by the total profit the algorithm obtains for jobs it completed.

Lemma 6.4.12. $\left\|C^O\right\| \leq \frac{1+ac\frac{2(1+2\delta)}{\delta b(1-b)}}{\epsilon - \frac{1}{(c-1)\delta}} \|C\|$.

Proof. This is just by combination of Lemma 6.4.7 and Lemma 6.4.11. $\square$

6.5 Conclusion

In this chapter we gave the first non-trivial result showing a scheduling algorithm which is provably good for maximizing throughput for DAGs. In addition, we extend the result and give an algorithm for the general profit scheduling problem with DAG jobs.

Chapter 7

Conclusion

Over the recent years, computing systems have grown more and more parallel. From mobile phones to web servers, most computers now have multiple processors. This trend is predicted to continue into the future. Therefore, exploiting the parallelism of computing systems will only grow in importance. This thesis focused on developing techniques which improve the efficiency of parallel systems - specifically, how to schedule multiple programs in a multicore system.

We study jobs in the DAG model, which naturally model parallel programs generated by many common languages and libraries. We work in the client-server scheduling model and give algorithms that are theoretically sound and algorithms that perform well in practice. In chapter 3 we gave the first theoretically good algorithm for minimizing the average flow time of a set of DAG jobs. We analyze both the algorithms LAPS and SJF and prove that LAPS is scalable while SJF is $(2 + \epsilon)$-speed, constant competitive. This result opened the way for other objectives for scheduling DAG jobs online. In chapter 4 we examine the problem of minimizing the maximum flow time. First we showed that FIFO is a scalable algorithm. However, FIFO is not an algorithm that is easy to implement for DAG jobs in practice. We also incorporated the randomized work-stealing scheduler to design a practical algorithm for maximum flow time. Using the idea of work-stealing, we revisit the problem of average flow time in chapter 5 and show a practical algorithm, DREP, that has strong theoretical guarantees and low scheduling overhead. Finally, we examine different online scheduling objectives, throughput and general cost, in chapter 6, and give strong theoretical results for the problem.

The goal of this thesis was to develop a theory of scheduling multiple parallel programs. To that end, we looked at the commonly studied online scheduling objectives and provided strong results for each of them. Of course, there are still many open problems remaining in scheduling multiple DAGs. I mention a few of the most interesting ones here.

There is an open problem remaining within the details of our maximum flow time result. For sequential jobs in the online non-clairvoyant setting, resource augmentation is not necessary to obtain a constant competitive algorithm. However, this is not the case for parallel jobs in the arbitrary speed-up curves model where there is a $O(\log n)$ lower bound even for schedulers which uses $O(1)$ speed augmentation. In the DAG model, we have show that FIFO is $(1+\epsilon)$-speed $O(1)$-competitive, but there may exist an algorithm that is constant competitive which does not require speed augmentation[6]. Intuitively, FIFO seems like the correct algorithm for this problem, however, it is difficult to prove its competitiveness without developing new proof techniques involving the structure of DAG jobs. It would be very interesting to resolve this open problem.

There are also some other flow time objectives which remain open such as the L_k norms of flow time and stretch. Though less popular objectives than the average and the maximum flow time, there are well known results in the case of sequential jobs. It is natural to try

[6]Note that for the weighted maximum flow problem, there does exist a lower bound for schedulers without resource augmentation. This means our result in chapter 4 for weighted max flow is tight up to constant factors

to extend these result to the DAG model. Stretch, in particular, is a interesting problem in this regard.

Finally, the work in this thesis were all on the identical machines setting where all the processors we schedule on are exactly the same. There are many other settings which would be interesting to study. There is not much known about scheduling DAG jobs in the related machines setting where different processors have different speeds. There is also not much known when other resources are taken into account, such as memory. Both of these are worthwhile problem settings that correspond to real-world computing systems. It is well worth trying to understate the way to efficiently run parallel jobs since these system will only grow more and more sophisticated in the future.

One day, we shall understand scheduling parallel programs just as well as for sequential programs. This thesis represents a major step in developing the theory of scheduling jobs in the DAG model. By designing theoretically good and practically efficient scheduling algorithms for many of the most popular online scheduling objectives, we have made much progress towards the ultimate goal.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 219–228, 2013.

[2] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin 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*, pages 176–189, 2016.

[3] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallelizable jobs online to minimize the maximum flow time. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 195–205, 2016.

[4] Christoph Ambühl and Monaldo Mastrolilli. On-line scheduling to minimize max flow time: an optimal preemptive algorithm. *Oper. Res. Lett.*, 33(6):597–602, 2005.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. volume 34, pages 115–144, 2001.

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

[7] Nikhil Bansal and Kirk Pruhs. The geometry of scheduling. *CoRR*, abs/1008.4889, 2010.

[8] Neal Barcelo, Sungjin Im, Benjamin Moseley, and Kirk 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*, pages 82–92, 2012.

[9] Sanjoy K. Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, Dennis E. Shasha, and Fuxing Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[10] Sanjoy K. Baruah, Gilad Koren, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, and Dennis E. Shasha. On-line scheduling in the presence of overload. In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 100–110, 1991.

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

[12] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1998, San Francisco, California, USA.*, pages 270–279, 1998.

[13] 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 *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of*

Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995, pages 207–216, 1995.

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

[15] Carl Bussema and Eric Torng. Greedy multiprocessor server scheduling. *Oper. Res. Lett.*, 34(4):451–458, 2006.

[16] 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.

[17] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Non-clairvoyant speed scaling for weighted flow time. In *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, pages 23–35, 2010.

[18] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with ϵ resource augmentation. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 363–372, New York, NY, USA, 2004. ACM.

[19] Chandra Chekuri, Sungjin Im, and Benjamin Moseley. Online scheduling to minimize maximum response time and maximum delay factor. *Theory of Computing*, 8(1):165–195, 2012.

[20] Jeff Edmonds. Scheduling in the dark. *Theor. Comput. Sci.*, 235(1):109–141, 2000.

[21] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the lk-norms of flow time without conservation of work. In *Proceedings of the Twenty-Second*

Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011, pages 109–119, 2011.

[22] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Trans. Algorithms*, 8(3):28:1–28:10, 2012.

[23] Kyle Fox, Sungjin Im, and Benjamin Moseley. Energy efficient scheduling of parallelizable jobs. volume 726, pages 30–40, 2018.

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

[25] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 11–20, 2010.

[26] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1242–1253, 2012.

[27] Intel. Intel Cilk^{TM} Plus. https://www.cilkplus.org/.

[28] Intel. Intel thread building blocks, 2017.

[29] Bala Kalyanasundaram and Kirk Pruhs. Fault-tolerant real-time scheduling. *Algorithmica*, 28(1):125–144, 2000.

[30] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[31] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM 2015, Shanghai, China, February 2-6, 2015*, pages 7–16, 2015.

[32] Gilad Koren and Dennis E. Shasha. MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theor. Comput. Sci.*, 128(1&2):75–97, 1994.

[33] Gilad Koren and Dennis E. Shasha. Dˆover: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. *SIAM J. Comput.*, 24(2):318–339, 1995.

[34] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *J. Comput. Syst. Sci.*, 73(6):875–891, 2007.

[35] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn 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*, PPoPP '16, pages 14:1–14:13, New York, NY, USA, 2016. ACM.

[36] 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 *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 85–96, 2014.

[37] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, May 2013.

[38] Kirk Pruhs, Julien Robert, and Nicolas 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, pages 237–248, 2010.

[39] Kirk Pruhs, Jirí Sgall, and Eric Torng. Online scheduling. In *Handbook of Scheduling - Algorithms, Models, and Performance Analysis.* 2004.

[40] Dana Randall, editor. *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011*. SIAM, 2011.

[41] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[42] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S. McKinley. Exploiting processor heterogeneity in interactive services. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pages 45–58, 2013.

[43] Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 491–500, 2008.

[44] Ola Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 745–754, 2010.

[45] Eric Torng and Jason McCullough. SRPT optimally utilizes faster machines to minimize flow time. *ACM Trans. Algorithms*, 5(1):1:1–1:25, 2008.

[46] Gerhard J. Woeginger. On-line scheduling of jobs with fixed start and end times. *Theor. Comput. Sci.*, 130(1):5–16, 1994.