APPROXIMATING TOTAL FLOW TIME

BY

DAVID PATTISON BUNDE

B.S., Harvey Mudd College, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

©Copyright by David Pattison Bunde, 2002

Abstract

We address the scheduling problem of minimizing total flow. Our primary focus is on the online setting. We begin by showing a structural similarity between the uniprocessor Shortest Remaining Processing Time (SRPT) and Shortest Processing Time (SPT) schedules. This structural result leads to a proof that SPT is $(\Delta + 1)/2$-competitive for nonpreemptive uniprocessor total flow, where Δ is the ratio between the longest and shortest job durations. This competitive ratio improves on the ratio of $\Delta + 1$ previously shown by Epstein and van Stee as a special case of their work on a resource-augmented version of the problem. The same technique leads us to an $O(\Delta \log \min\{n, \Delta\})$-competitive algorithm for nonpreemptive multiprocessor total flow, the first competitive algorithm known for this problem. Then we show that $(\Delta + 1)/2$ and $(\Delta + 6)/8$ are lower bounds on the competitive ratio of deterministic and randomized busy algorithms, respectively. (A busy algorithm is one that is not unnecessarily idle.) Since SPT is a busy deterministic algorithm, it is the most competitive among deterministic algorithms and cannot be much less competitive than the best randomized busy algorithm.

We also briefly consider offline scheduling. In this setting, we show that total flow cannot be approximated to a factor of $n^{1-\epsilon}$ or $\Delta^{1-\epsilon}$ for any constant $\epsilon > 0$ unless P=NP. The proof of this result closely follows the uniprocessor approximability lower bound of $n^{1/2-\epsilon}$ by Kellerer et al. and the multiprocessor bound of $n^{1/3-\epsilon}$ by Leonardi and Raz, neither of which are restricted to busy algorithms.

To my fiancée Jennie

Acknowledgments

This thesis would not have been possible without support from many people. My advisor Jeff Erickson taught me how to perform and evaluate research by sharing his own process and opinions. He listened to early versions of this work, before it was well articulated or entirely correct, and helped me to bring out my ideas. Cindy Phillips and Vitus Leung, my technical mentors at Sandia National Laboratories, introduced me to scheduling and led me to become familiar with its literature and techniques. It was during work and discussions with them that the seeds of my first results were planted. I thank Michael Bender for refusing to believe my "obvious" claims and forcing me to present my results in a rigorous way. My work has also benefited from many discussions with colleagues in the CS Department, especially Shripad Thite, Mitch Harris, Sariel Har-Peled, and Steve LaValle.

Table of Contents

Chapter

List of Tables viii

List of Figures ix

1 Introduction 1
- 1.1 Scheduling 1
- 1.2 Formal Definition and Notation 2
- 1.3 Overview 5

2 Background and Related Work 6
- 2.1 Competitive Analysis and Approximation 6
- 2.2 Lower Bound Techniques 7
  - 2.2.1 Adversary Arguments 7
  - 2.2.2 Reductions 9
- 2.3 Known Results 14
  - 2.3.1 Total Flow 15
  - 2.3.2 Weighted Total Flow 19
  - 2.3.3 Max Stretch 19
  - 2.3.4 Average/Total Stretch 20
  - 2.3.5 Error 20

3 Online Setting 22
- 3.1 Block Structure 22
- 3.2 Schedule ECHO 24
- 3.3 Shortest Processing Time 25
- 3.4 Multiprocessor Approximation 27
- 3.5 Lower Bounds 28
  - 3.5.1 Deterministic Scheduling 29
  - 3.5.2 Randomized Scheduling 29
  - 3.5.3 Randomized Scheduling with Restarts 31
  - 3.5.4 Randomized Busy Scheduling 32

4 Lower Bounds in the Offline Busy Setting 34
- 4.1 Uniprocessor 34
- 4.2 Multiprocessor 37

5 Conclusions and Open Problems . . . 39
5.1 Conclusions . . . 39
5.2 Open Problems . . . 39
5.2.1 Randomized Approximation of Flow . . . 40
5.2.2 Multiprocessor Setting . . . 40
5.2.3 Other Parameters . . . 40
5.2.4 Smoothed Analysis . . . 41

Bibliography . . . 42

List of Tables

2.1 Approximibility of Total Flow 15
2.2 Approximibility of Uniprocessor Max Stretch 20
2.3 Approximibility of Average (or Total) Stretch 20

List of Figures

2.1 Small jobs in proof of Theorem 4 11
2.2 Small jobs in proof of Theorem 5 13
2.3 Instance where SRPT falls behind in number of jobs completed 16

3.1 Illustration for the proof of Theorem 9 23
3.2 Example of an ECHO schedule 24
3.3 Example of a Slide Operation 26
3.4 Instance where SPT is better than ECHO 27
3.5 Instance Where the SRPT Schedule Migrates a Job 28

Chapter 1

Introduction

1.1 Scheduling

Scheduling problems occur throughout computer science. From the earliest days of computing, when access to computers was available only during specific pre-scheduled time slots, great care has been taken to efficiently manage these expensive machines. Users of large, expensive multiprocessor machines still submit their programs to queues from which schedules are generated. The scheduling process is better hidden on desktops, but even these small systems use scheduling algorithms to share the Central Processing Unit (CPU) between multiple user and operating systems processes. Scheduling algorithms also control access to resources shared by multiple processors, such as network bandwidth and devices like printers.

Nor are scheduling problems restricted to computer systems. Groups working on large projects typically schedule task graphs, which divide the project into separate tasks and record the dependencies between them. Meetings between people are scheduled so that rooms are available. Traffic lights enforce a scheduling discipline to control access to intersections. On an individual level, we all are called upon to perform multiple tasks during the day and must schedule our own time between them. Scheduling problems appear whenever a limited resource is shared between different users or tasks.

While this thesis cannot address scheduling in all of its generality, we present a number of results that advance our understanding of scheduling.

1.2 Formal Definition and Notation

Before we discuss our results, we need to formally define the scheduling problems we address. Traditionally in computer science, the definition of a scheduling problem is given as three sets of parameters. The first set of parameters is the list of resources to be scheduled and the relationship between different resources. In keeping with the processor scheduling example, the resources are typically called processors and the things to be scheduled are called jobs. Sometimes the problem specifies a particular number of processors, such as 1 or 2, but general multiprocessor algorithms that schedule m processors are also studied. Throughout this thesis, we will assume the case of *uniform processors*, meaning that a job takes the same amount of time on any processor. The cases where some processors are faster than others (*related processors*) and where jobs take arbitrarily-different amounts of time on different processors (*unrelated processors*) have also been considered, as have the problems where jobs need to be run on several processors either in a particular order (*flow shop scheduling*) or in any order (*open shop scheduling*).

The second set of parameters given to specify a scheduling problem is the type of input included in a problem instance. A uniprocessor scheduling problem takes as input a set of n jobs $\{J_1, J_2, \dots, J_n\}$, each of which must exclusively occupy the processor for its processing time p_i. The jobs in a multiprocessor problem can be either *serial jobs*, each of which runs on a single processor, or *parallel jobs* that must run on several processors simultaneously. In this thesis, we restrict our attention to problems with serial jobs when considering multiprocessor problems.

Other information is associated with each job in different scheduling problems. We consider scheduling problems that assign a *release time* r_i to each job J_i, meaning that a processor cannot begin J_i before r_i. Some scheduling problems also assign a *weight* w_i to each job, indicating

its relative importance. Other inputs considered in the literature include a partial order $\prec$ of precedence constraints, where $J_i \prec J_j$ means that J_i must be completed before beginning J_j, and *deadlines*, where the deadline d_i is the time by which job J_i should (or must) be finished.

A different type of problem input is special abilities of the processors or restrictions on the schedule. By default, the processors are *nonpreemptive*, meaning that once a job is started, it must be run to completion. A more powerful processor allows *preemption*, meaning that jobs can be "paused" and resumed without penalty. Intermediate in power between these is a processor that allows *restarts*, where a job is stopped and then started from the beginning at a later time. One restriction that can be imposed on the schedule is that it be *busy*, i.e. the processors cannot be idle if there is a job that can be started.[1]

The last set of parameters used to specify a scheduling problem is the metric that is used to measure performance. Generally the problem's objective is to minimize this metric, though some exceptions exist (see [1]). The following are some metrics that have been studied:

- *Makespan*, $\max_i C_i^A$, where C_i^A is the time at which job J_i is finished in the schedule generated by algorithm A. C_i^A is called the *completion time* of J_i (in the schedule generated by A). The makespan is the time when the last job is finished. On a single processor any busy schedule minimizes makespan, but minimizing makespan on two processors is a generalization of PARTITION and so is NP-hard [14].

- *Sum of completion times*, $\sum_i C_i^A$. This is equivalent to average completion time.[2] The main criticism of this metric is that it assumes there is a "starting time" and different choices of starting time yield different values of the metric.

- *Sum of weighted completion times*, $\sum_i w_i C_i^A$. This is equivalent to average weighted completion time. It is the same as sum of completion times except that different jobs are given different relative importance. It also suffers from having a start time.

[1] Busy schedules are also called *work conserving*, *conservative*, and *active*.

[2] Both average completion time and average flow time have been called "response time" in the literature. To avoid confusion, we avoid this terminology.

- *Total flow time*, $\sum_i F_i^A$, where $F_i^A = C_i^A - r_i$ is the *flow* of job i. This is equivalent to average flow time.[3] By considering flow instead of completion time, the problem of picking a starting time mentioned above is avoided; a different starting time would affect both the completion and release times, resulting in no net effect on the flow. Note that an optimal solution for the total completion time metric is an optimal solution for the total flow metric, but an approximation for total completion time does not become an approximation for total flow time because of the effect of the starting time.

- *Weighted total flow time*, $\sum_i w_i F_i^A$. This is equivalent to *average weighted flow time.* It is the same as total flow time except different jobs are given different relative importance.

- *Maximum stretch*, $\max_i S_i^A$, where $S_i^A = F_i^A / p_i$ is called the *stretch* of job i.[4] Stretch is a special case of weighted flow, where each job's weight is the inverse of its processing time. This metric captures the intuition that it is worse to make short jobs wait because they are more likely to be interactive jobs with a user waiting for them to complete, while users submitting longer jobs do not expect a quick result and thus will probably find something else to do in the meantime.

- *Sum of stretch*, $\sum_i S_i^A$. This is equivalent to average stretch.

The three sets of problem parameters can be represented compactly using the $\alpha|\beta|\gamma$ notation proposed by Graham et al. [18], with extensions by Błażewicz et al. [7], Veltman et al. [29], and Drozdowski [11]. In this notation, α describes the processors, β describes the jobs, and γ gives the performance metric. For example, $1|r_i|\sum_i F_i$ represents uniprocessor nonpreemptive total flow with release times, while $m|r_i, pmtn|\sum_i w_i F_i$ represents preemptive weighted total flow with release times on m processors.

Scheduling problems are considered in both the online and offline settings. In an *online* problem, scheduling decisions at time t can only be based on knowledge of jobs that have

[3] Both average completion time and average flow time have been called "response time" in the literature. To avoid confusion, we avoid this terminology.

[4] Stretch is also called slowdown since it is the difference between the time algorithm A takes to run the job and the time it takes in an ideal setting.

been released by time t, i.e. jobs in $\{J_i : r_i \leq t\}$. *Offline* problems, on the other hand, give the algorithm full knowledge of the input instance before any scheduling decisions are made. Although both are studied, there is generally more interest in the harder online setting, because most scheduling applications are ongoing processes, with jobs being finished and new jobs arriving continuously.

In this thesis, we focus exclusively on minimizing total flow. We focus primarily on non-preemptive problems, though this requires some consideration of the preemptive setting as well.

1.3 Overview

The rest of this thesis is organized as follows. Chapter 2 describes the techniques we use and summarizes related work.

Chapter 3 considers online scheduling. Section 3.1 defines a schedule decomposition and shows that it is shared by two common scheduling algorithms, Shortest Remaining Processing Time (SRPT) and Shortest Processing Time (SPT). Sections 3.2 and 3.3 use this decomposition to show that SPT is competitive for uniprocessor nonpreemptive flow. Section 3.4 applies these techniques to the multiprocessor case and gives the first known competitive algorithm for this problem. Section 3.5 gives some lower bounds on the competitive ratio of uniprocessor flow.

Chapter 4 considers offline scheduling, giving some lower bounds on the competitive ratio of busy algorithms. Section 4.1 gives a uniprocessor lower bound and Section 4.2 gives one for the multiprocessor setting.

Finally, Chapter 5 summarizes our results and discusses some open problems.

Chapter 2

Background and Related Work

This chapter gives the background necessary to understand this thesis and then discusses previous work to put our work into context. Section 2.1 defines competitive analysis and approximability analysis, which we use throughout the remainder of the thesis. Section 2.2 describes the techniques used to prove lower bounds in these two types of analysis. Then Section 2.3 summarizes the previous work to minimize total flow and related scheduling problems.

2.1 Competitive Analysis and Approximation

For most scheduling problems, it is not possible to find an optimal solution in the online setting. In this case, *competitive analysis* is used to capture how well various algorithms perform. The idea is to compare an algorithm's performance to the optimal performance. An algorithm is ρ-competitive if, for all problem instances, the algorithm gives a solution with cost at most ρ times the cost of the optimal solution. If the algorithm is randomized, its expected cost on the input instance is used for this calculation, where the expectation is over the random bits used by the algorithm. Similarly, if the input instance is randomized, the algorithm's expected cost and the expected optimal value are used.

Although it is always possible to find the optimal solution to an offline scheduling problem, doing so may take an unacceptable amount of time because many scheduling problems are

NP-hard [14, 10]. To capture the relative performance of non-optimal algorithms, we use *approximation analysis*, the offline analog of competitive analysis. Instead of comparing an online algorithm to the optimal solution, we compare a fast (generally polynomial-time) algorithm to the optimal solution. An algorithm that produces a solution with cost at most ρ times the optimal cost is said to be a ρ-approximation algorithm or to ρ approximate the solution.

2.2 Lower Bound Techniques

Now we describe the techniques used to prove lower bounds on ρ, the competitive ratio or the approximability ratio.

2.2.1 Adversary Arguments

For online problems, lower bounds on the competitive ratio are proven using an *adversary argument*, which consists of one or more input instances that might be given to the algorithm by "the adversary", who wants to algorithm to perform poorly.

2.2.1.1 Deterministic Setting

Adversary arguments are simplest against deterministic algorithms. In this case, the input instance can be constructed with knowledge of the algorithm's behavior. An example of this technique appears in the proof of the following theorem:

Theorem 1 (Kellerer et al. [20]) *Any deterministic online algorithm for minimizing total flow time on a single nonpreemptive processor has an* $\Omega(n)$ *competitive ratio.*

Proof. We construct an adversarial input based on the algorithm's behavior on a prefix of the input. At time 0, a single job of length 1 is released. Any deterministic algorithm starts this job at some time t if no other jobs are released before then. Immediately after time t (i.e. $t+\epsilon$ for some small ϵ), the adversary releases $n-1$ jobs with processing time 0.[1] The algorithm

[1] The use of jobs with processing time 0 seems somewhat questionable, but they can be replaced by extremely short jobs arriving right after one another. This same argument with short jobs in the place of jobs with processing time 0 is given in the proof of Theorem 12 in Section 3.5.1.

has total flow arbitrarily close to $n-1$ while the adversary can schedule the jobs for flow at most 2; run the jobs of length 0 as soon as they arrive and run the first job either at time 0 or immediately after the jobs of length 0. □

This proof highlights the adversary's power when it knows exactly what the algorithm will do on a partial input instance, namely the first job. For a deterministic algorithm, this ability is reasonable because the adversary can do a "test run" of the algorithm, releasing only this job to determine when the algorithm starts it. A closely-related idea is an *adaptive adversary*, one who is allowed to generate the input instance as the algorithm is running, i.e. to adapt the instance to the algorithm's execution choices. There are two variations of adaptive adversaries, depending on what the algorithm's performance is compared to; an *adaptive online adversary* is forced to make scheduling decisions as it generates the input instance, while an *adaptive offline adversary* computes the offline optimum schedule.

2.2.1.2 Randomized Setting

To show lower bounds on the competitive ratio against randomized scheduling algorithms, it is customary to use an *oblivious adversary*. In this model, the adversary generates an input instance with knowledge of the algorithm, but not of the values of random bits used by the algorithm. Since an algorithm's behavior is affected by the values of its random bits, this limits the adversary's ability to exactly tailor the input to the algorithm.

At first glance, the algorithm's random bits also seem like an impediment to analyzing its competitive ratio by greatly increasing its range of behavior. This problem is resolved by Yao's Lemma:

Theorem 2 (Yao [31]) *The expected cost of an optimal deterministic algorithm for an arbitrarily chosen input distribution is a lower bound on the expected worst-case cost of an optimal randomized algorithm.*

By relating the performance of a randomized algorithm to the performance of an optimal deterministic algorithm, Yao's Lemma allows us to perform all of our lower bound analysis

on deterministic algorithms. To get a lower bound for a randomized algorithm, we generate a random input distribution and bound the competitive ratio of the optimal deterministic algorithm on that input. Then, by Yao's Lemma, the same bound applies to any randomized algorithm.

As an example of this technique, we give the following from Epstein and van Stee [12]:

Theorem 3 (Epstein and van Stee [12]) *Any nonpreemptive algorithm for minimizing uniprocessor total weighted flow time has competitive ratio at least $\Omega(\sqrt{W})$, where W is the ratio between the largest and smallest weight.*

Proof. Using Yao's Lemma, we consider a deterministic algorithm on a randomized input instance. At time 0, the adversary releases a "light" job with unit processing time and unit weight. The instance has only one other job, which we call the "heavy" job since it has size 0 and weight W. This job is released at a time chosen uniformly at random from the interval $(0, \sqrt{W})$. The optimal solution is at most 2; run the heavy job as soon as it arrives and run the light job either at time 0 or immediately after finishing the heavy job.

A deterministic algorithm for this problem will wait some time S before starting the light job if the heavy job does not arrive by that time. If $S \geq \sqrt{W}/2$, then it will wait at least $\sqrt{W}/2$ half the time, for expected weighted flow of $\sqrt{W}/4$. Otherwise, there is probability $1/(2\sqrt{W})$ that the heavy job arrives in the interval $(S, S + 1/2)$. If this occurs, the algorithm has cost at least $W/2$, giving the algorithm an expected weighted flow of at least $\sqrt{W}/4$. In either case, the expected competitive ratio is $\Omega(\sqrt{W})$. □

2.2.2 Reductions

For offline problems, lower bounds are shown by reductions of formally hard problems to approximate scheduling. The hard problems are generally NP-complete problems, a class of problems that are widely believed to require superpolynomial time. A reduction from an NP-complete problem P_1 to approximating the scheduling problem P_2 better than $f(n)$ is a procedure for converting an instance of P_1 into an instance of P_2 such that approximating this

instance within $f(n)$ gives a solution to the original instance of P_1. This technique was first used to prove a lower bound on the approximation ratio of a scheduling problem by Kellerer, Tautenhahn, and Woeginger [20]. They used the following NP-complete problem [14]:

> **Numerical 3-Dimensional Matching (N3DM)**: Given three sets of positive integers $A = \{a_1, a_2, \ldots, a_k\}$, $B = \{b_1, b_2, \ldots, b_k\}$, and $C = \{c_1, c_2, \ldots, c_k\}$, with $\sum_{i=1}^{k}(a_i + b_i + c_i) = kD$, do there exist permutations π and ψ such that $a_i + b_{\pi(i)} + c_{\psi(i)} = D$ holds for all i?

Essentially, this problem asks if members of A, B, and C can be grouped in triples, each of which sums to D. The argument by Kellerer et al. [20] is essentially that a sufficiently-good polynomial-time approximation for uniprocessor nonpreemptive total flow gives a polynomial-time solution to N3DM. Since N3DM is strongly NP-complete, the approximation algorithm can take time polynomial in kD rather than $\log kD$, the length of its binary encoding. Following Kellerer et al. [20], similar reductions of N3DM were used to prove inapproximability results for multiprocessor flow [22] and max-stretch [5]. All of these results give approximation lower bounds of the form $n^{\alpha-\epsilon}$ for some constant α and arbitrary $\epsilon > 0$ unless P=NP.

2.2.2.1 Uniprocessor Lower Bound

We now describe the results by Kellerer et al. [20] and Leonardi et al. [22] to illustrate the use of reductions to prove approximability lower bounds and to prepare for the proofs of similar results in Chapter 4.

Theorem 4 (Kellerer, Tautenhahn, Woeginger [20]) *No polynomial-time algorithm approximates offline uniprocessor total flow time to within a factor of $n^{1/2-\epsilon}$ for any constant $\epsilon > 0$ unless P=NP.*

The basic idea behind the reduction is to periodically release streams of small jobs that must be run immediately; each stream contains enough jobs that any delay greatly increases the flow. These streams of small jobs partition the time axis into "bins" that can be exactly

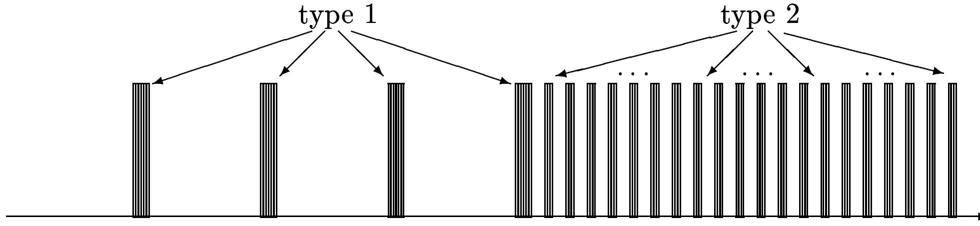


Figure 2.1: Small jobs in proof of Theorem 4

filled by triples of jobs when a matching occurs in the original instance of N3DM. When these bins cannot be filled exactly, either some bins must be overfilled, delaying the small jobs at the end of a bin, or large jobs must be postponed until after all the bins. To penalize large jobs that are postponed, closely-spaced streams of small jobs are released for a long time after the end of bins. The streams of small jobs are shown in Figure 2.1 diagram, where the bins are formed by jobs of type 1 and the later jobs are jobs of type 2.

The integers a_i, b_i, and c_i are changed into jobs of duration $a_i + 2r$, $b_i + 4r$, and $c_i + 8r$ respectively, where we make the parameter r greater than D so that each bin must contain exactly one element from each of A, B, and C if it is exactly filled. This means that each group of type 1 is separated by $D + 14r$. We separate the groups of type 2 by r so that large jobs cannot fit between two of them.

Next we denote the number of groups of type 2 with the parameter g. Thus, the "tail" of type 2 groups has length rg. Since this is the cost of delaying a long job until after the tail, we will make it the cost of delaying a group of small jobs. To prevent the bins from being overfilled, we need to prevent the groups of type 1 from being delayed by a single time unit. Therefore, each group of type 1 will consist of rg jobs. Similarly, preventing a big job from fitting between groups of type 2 requires that these groups not be delayed by r so each type 2 group will consist of g jobs. In order for each group to have total length 1, we give a processing time of $1/rg$ to each job of type 1 and a processing time of $1/g$ to each job of type 2.

The optimal flow of this instance is $k(D + 14r) + k + g$ for the actual processing time of the jobs and $3\binom{k}{2}(D + 14r + 1)$ for the waiting time of the long jobs. Since $r > D$, these sum

to $O(g + k^2 r)$. If we assume $g = \Theta(k^2 r)$, this becomes $O(k^2 r)$. By design, the flow is at least $rg = \Theta(k^2 r^2)$ when the instance of N3DM does not have a solution, i.e. when the bins cannot be filled properly.

In order for an $n^{\alpha-\epsilon}$ approximation to distinguish between an instance of N3DM that has a solution and one that does not, we need $k^2 r n^{\alpha-\epsilon} = \Theta(k^2 r^2)$. Thus, we want $n^{\alpha-\epsilon} = \Theta(r)$. Since the number of jobs is $3k + krg + g^2 = O(k^4 r^2)$, this gives $\alpha \leq 1/2$. We use the values

$$\begin{aligned} r &= \lceil 2Dn^{(1-\epsilon)/2} \rceil \\ g &= 100rk^2 \\ n &= \lceil (20k)^{4/\epsilon} D^{2/\epsilon} \rceil \end{aligned}$$

to achieve $\alpha = 1/2$ and prove Theorem 4.

2.2.2.2 Multiprocessor Lower Bound

The lower bound on multiprocessor approximability by Leonardi and Raz [22] has a very similar flavor. Like the proof of Theorem 4 presented above, the argument by Leonardi and Raz [22] uses an instance with small jobs of type 1 to create bins and small jobs of type 2 to delay large jobs that are not placed into a bin. Figure 2.2 shows these small jobs in a 4 processor instance, where the processors are labeled P_1 through P_4. So that each processor is assigned the same number of bins, assume that the number of triples k is an integral multiple of the number of processors m.

Unlike in the uniprocessor case, overfilling a bin does not delay a group of type 2 jobs by r. The jobs that are "pushed" by an overly-large job can be run on another processor once it finishes its group of small jobs. To compensate for the presence of other processors, each group of type 2 now consists of rg jobs, each with duration $1/rg$. It is this use of extra jobs in the tail that leads to a weaker bound in the multiprocessor case:

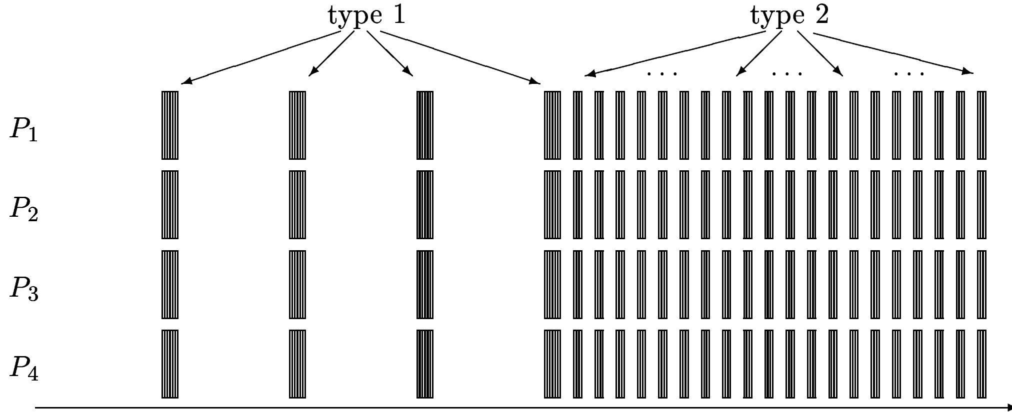


Figure 2.2: Small jobs in proof of Theorem 5

Theorem 5 (Leonardi and Raz [22]) *No polynomial-time algorithm approximates offline nonpreemptive total flow to within a factor of* $n^{1/3-\epsilon}$ *on* $m \leq n^{\epsilon/4}$ *processors for any constant* $\epsilon > 0$ *unless P=NP.*

The actual instance used to prove this theorem is the following:

- *Large jobs.* Each $a_i \in A$, $b_i \in B$, and $c_i \in C$ becomes a job with processing time $2mr + a_i$, $4mr + b_i$, and $8mr + c_i$ respectively. All these jobs are released at time 0.
- *Small jobs of type 1.* Each of these jobs has processing time $1/rg$. Released as m simultaneous streams beginning at $l(14mr + D + 1)$ for each $l = 1, \ldots, k/m$. Each stream consists of rg jobs, for rgk total jobs with combined processing time of k.
- *Small jobs of type 2.* Each of these jobs has processing time $1/rg$. Released as m simultaneous streams beginning at $(k/m)(14mr + D + 1) + lmr - 1$ for each $l = 1, \ldots, g/m$. Each stream consists of rg jobs, for rg^2 total jobs with combined processing time of g.

The extra factor m in the size of large jobs and in the distance between groups of small jobs compensates for spreading the small jobs of type 2 over m processors, so that the tail of small jobs stretches as far as in the previous theorem.

Delaying a large job until after the small jobs of type 2 results in a flow of at least gr since this is when the last group of type 2 jobs starts. If one of the large jobs is running while a

stream is released, the displaced jobs cause 1 extra job to be waiting during the first $1/rg$ time units of the stream, 2 extra jobs to be waiting during the second $1/rg$ time units, and so on. This extra waiting represents a flow of $(1/rg)\sum_{i=1}^{rg} i > rg/2$.

The proof uses values

$$\begin{aligned} r &= \lceil 3Dn^{(1-\epsilon)/3} \rceil \\ g &= 100rk^2 \\ n &= \lceil (30k)^{4/\epsilon} D^{3/\epsilon} \rceil \end{aligned} \tag{2.1}$$

so that the total number of jobs is

$$3k + rgk + rg^2 \leq 2rg^2 \leq 900^2 D^3 n^{1-\epsilon} k^4 \leq n$$

When the N3DM instance has a solution, each large job waits at most $(k/m)(14mr+D+1)$ and every small job runs as soon as it is released, for total flow of at most

$$(k/m)(14mr + D + 1)3k + k + g \leq 150rk^2$$

Thus, an $n^{(1-\epsilon)/3}$ approximation of the scheduling instance would have flow at most $50r^2k^2$ since $n^{(1-\epsilon)/3} \leq (r/3)$. As previously shown, a scheduling instance has flow greater than this when the N3DM instance from which is was generated does not have a solution. Therefore, an $n^{(1-\epsilon)/3}$ approximation would allow us to solve N3DM.

As a final note, the condition $n \geq m^{4/\epsilon}$ in the statement of Theorem 5 comes from Equation (2.1) because $k \geq m$.

2.3 Known Results

Now we summarize the previously-known algorithms and lower bounds.

		preemptive	non-preemptive
1	online	1 [Sc68,Sm78]	$\Omega(\sqrt{n})$ [30] $\Omega(\sqrt{\Delta})$ [12] $\leq \Delta + 1$ [13] deterministic alg.: $\Omega(n)$ [20] $\Omega(\Delta)$ [13]
	offline		$O(\sqrt{n})$ $\omega(n^{1/2-\epsilon})$ if $\mathrm{P} \neq \mathrm{NP}$ [20]
m	online	$\Theta(\log \min\{\Delta, n/m\})$ [22]	deterministic alg.: $\Omega(n/m^2)$ [13]
	offline	$O(\log \min\{\Delta, n/m\})$ [22]	$O(\sqrt{n/m}\log(n/m))$ $\omega(n^{1/3-\epsilon})$ if $\mathrm{P} \neq \mathrm{NP}$ $\omega(\Delta^{1/3-\epsilon})$ if $\mathrm{P} \neq \mathrm{NP}$ [22]

Table 2.1: Approximibility of Total Flow

2.3.1 Total Flow

The known approximability of total flow is summarized in Table 2.1.

Probably the best known algorithm for minimizing total flow is Shortest Remaining Processing Time (SRPT). This algorithm always runs the job with the least processing time remaining, using preemption to run newly-arrived jobs that are shorter than the currently-running job. The reason for its fame is that it gives an optimal solution to the problem of minimizing total flow on preemptive uniprocessors, as shown by Schrage [24] and Smith [27]. The first proof, due to Schrage [24], uses induction to prove the following:

Theorem 6 (Schrage [24]) *At any time on a uniprocessor system, SRPT has finished as many jobs as any other algorithm.*

The optimality of SRPT for uniprocessor total flow follows immediately because flow can also be calculated as $\int_t \delta(t)dt$, where $\delta(t)$ denotes the number of jobs that have been released and not finished at time t.

Unfortunately, Theorem 6 does not hold in the multiprocessor setting, as demonstrated by Figure 2.3. Leonardi and Raz [22] used the idea demonstrated by this figure to show an $\Omega(\log(n/m))$ lower bound on the competitive ratio of any online algorithm trying to minimize

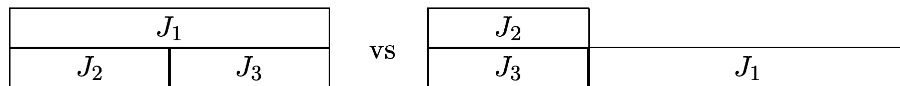


Figure 2.3: Instance where SRPT falls behind in number of jobs completed

total flow on m processors. They also showed that SRPT is $O(\log(n/m))$-competitive in this setting, meaning is it still the best algorithm in some sense.

Because SRPT is a specific optimal algorithm, it is often used as the point of comparison for computing the competitive ratio of other scheduling algorithms. Comparisons against SRPT occur even when considering nonpreemptive algorithms, since SRPT acts as a lower bound on the optimal nonpreemptive flow. Such applications have led to observations about the structure of SRPT schedules such as the following, which we use in our results:

Theorem 7 (Kellerer et al. [20]) *If SRPT begins a job J_i while J_j is preempted, J_i will be finished before J_j is resumed.*

When preemptions are not allowed, optimal solutions to total flow are no longer possible in the online setting. All deterministic algorithms are $\Omega(n)$-competitive for uniprocessor problems [20] (see Theorem 1) and $\Omega(n/m^2)$-competitive for multiprocessor problems [13]. Even with randomization, Vestjens [30] gives an $\Omega(\sqrt{n})$ lower bound for uniprocessor problems.[2] Epstein and van Stee [12] showed that this bound holds even when the processor is allowed to restart jobs.

Even offline, it is not easy to approximate total flow. Kellerer et al. [20] showed that, unless P=NP, uniprocessor total flow cannot be approximated within $n^{1/2-\epsilon}$ for any $\epsilon > 0$ (see Theorem 4). They also gave an $O(\sqrt{n})$-approximation algorithm. In the multiprocessor case, Leonardi and Raz [22] showed that, unless P=NP, multiprocessor total flow cannot be approximated within $n^{1/3-\epsilon}$ for any $\epsilon > 0$ (see Theorem 5 above) and gave an $O(\sqrt{n/m}\log(n/m))$-approximation algorithm.

[2]The proof presented by Vestjens [30] seems to be incorrect; refer to Section 2.3.5 for details. The result is correct, however, as shown by Epstein and van Stee [12] and in Section 3.5.2.

2.3.1.1 Approximations in terms of Δ

The strong inapproximability bounds in terms of n make it natural to consider approximations in terms of other parameters. One choice is Δ, the ratio between the largest and smallest processing times. Epstein and van Stee [13] showed that Shortest Processing Time (SPT), which begins the shortest available job whenever a processor becomes idle, is a $\Delta + 1$ approximation. They also showed an $\Omega(\min\{n, \Delta\})$ lower bound for deterministic algorithms.[3] A slight modification of their $\Omega(\sqrt{n})$ bound [12] gives an $\Omega(\sqrt{\Delta})$ lower bound in the case of randomized algorithms with restart.

For the online multiprocessor problem, Leonardi and Raz [22] showed that SRPT is $O(\log \Delta)$-competitive and gave a matching $\Omega(\log \Delta)$ lower bound. When preemptions are not allowed, Epstein and van Stee [13] give an $\Omega(\Delta/m)$ lower bound for deterministic algorithms.[4]

In the offline uniprocessor setting, Kellerer et al. [20] gives a lower bound of $\omega(\Delta^{1/3-\epsilon})$ for any constant $\epsilon > 0$ unless P=NP.[5] Leonardi and Raz [22] give the same bound in the multiprocessor setting when $m \leq n^{\epsilon/4}$.[6]

2.3.1.2 Problem variations

Several variations of flow minimization are also considered in the literature. One variation that applies in the multiprocessor setting, is to eliminate *migration*, where a job is started on one processor, preempted, and then resumed on a different processor. Since resuming a job on a different processor involves transfering the program's data and state information between the affected processors, a reasonable objective is to minimize migration. The natural way to

---

[3] The actual approximation result by Epstein and van Stee [13] was that an algorithm called Revised Levels running on lm processors could get a solution with at most $O(\min\{\Delta^{1/l}, n^{1/l}\})$ times the flow of the optimal algorithm running on m processors. The lower bound was $\Omega(\min\{n^{1/l}, \Delta^{1/l}\}/(12l)^l)$ in the same setting. The quoted results follow by plugging in $l = 1$ and noticing that Revised Levels on a single processor is SPT.

[4] This result is not specifically claimed by Epstein and van Stee [13]; it is acheived by noticing that $\Delta = 2n/m$ in their $\Omega(n/m^2)$ lower bound argument.

[5] This result is not specifically claimed by Kellerer et al. [20]; it is achieved by noticing that $\Delta = O(r^2 g) = O(r^3 k^2) = O(n^{3/2})$ in their $n^{1/2-\epsilon}$ lower bound argument (see the proof of Theorem 4).

[6] This result is not specifically claimed by Leonardi and Raz [22]; it is achieved by noticing that $\Delta = O(mr^2 g) = O(mr^3 k^2) = O(r^3 k^3) = O(n)$ in their $\omega(n^{1/3-\epsilon})$ lower bound argument (see the proof of Theorem 5).

avoid migration without always running jobs to completion is to preempt only in favor of a job that is "significantly" smaller than the current job. Awerbuch et al. [2] propose an algorithm that considers jobs in classes, where class k consists of jobs with remaining duration in the interval $[2^k, 2^{k+1})$. Their algorithm keeps a stack of jobs for each processor as well as a pool of unassigned jobs. A processor will take the shortest job in the pool if the new job is in a smaller class than the job it is currently working on or if the processor is currently idle. This algorithm has a competitive ratio of $O(\min\{\log n, \log \Delta\})$, asymptotically equal to the ratio of SRPT and the lower bounds for multiprocessor total flow.

Another desirable feature of a scheduling algorithm is that it avoids *starvation*, which occurs when a job is delayed arbitrarily by later-arriving jobs. Starvation can occur in SRPT, where a job can be delayed arbitrarily by the arrival of shorter jobs. A simple alternative algorithm that avoids starvation is Processor Sharing (PS), where each released job runs for some amount of time (called a *quantum*) before being preempted in favor of the next job. Bansal and Harchol-Balter [3] suggest that the fear of starvation is exaggerated by empirically showing that SRPT has lower max stretch and max flow than PS under what they claim are realistic job distributions. They also show that SRPT has better average flow and average stretch than PS under all input distributions.

The last variation we discuss is a variation in the scheduling model itself. In practice, the duration of a job is often not known until the job completes. Even when job durations are "known", the values given are generally estimates, either user-supplied or based on previous runs of the program. Scheduling problems modeling these circumstances, when job durations are not known, are called *nonclairvoyant*. For total flow in the nonclairvoyant setting, Motwani et al. [23] show that deterministic algorithms are $\Omega(\sqrt[3]{n})$-competitive and that randomized algorithms are $\Omega(\log n)$-competitive. Kalyanasundaram et al. [19] give an $O(\log n \log\log n)$-competitive randomized algorithm that resembles Multilevel Feedback (MLF) algorithms, which appear in modern operating systems [26]. These algorithms resemble PS using multiple queues of different priority levels, where the "next" job is the head of highest priority queue that is

nonempty and jobs move to lower priority queues as their elapsed runtime increases. See Sgall [25] for additional results on nonclairvoyant scheduling.

2.3.2 Weighted Total Flow

Unlike unweighted total flow, weighted total flow cannot be solved optimally in the online setting. There are lower bounds of $4/3$ for randomized algorithms [9, 12] and 2 for deterministic algorithms [12]. If Δ is known ahead of time, Chekuri et al. [9] give an $O(\log^2 \Delta)$-competitive preemptive algorithm.

When preemption is not allowed, even with randomization and restarts, there is an $\Omega(n)$ bound on the competitive ratio by Epstein and van Stee [12]. Epstein and van Stee [12] also show a lower bound of $\Omega(\sqrt{W})$ when restarts are not allowed, where W is the ratio between the largest and smallest weights (see Theorem 3). In the multiprocessor setting, Chekuri et al. [9] give an $\Omega(\min\{\sqrt{\Delta}, \sqrt{W}, (n/m)^{1/4}\})$ lower bound on the competitive ratio of randomized algorithms.

In the offline setting, the news is somewhat better. For uniprocessor problems, Chekuri et al. [9] give a $(2+\epsilon)$-approximation algorithm that runs in $n^{O(\log^2 n)}$ time assuming that the processing times are polynomial in n. Chekuri et al. [8] give a PTAS when $\Delta = O(1)$ and an $(1+\epsilon)$-approximation algorithm with running time $O(n^{O(\log W \log \Delta/\epsilon^2)})$.

2.3.3 Max Stretch

Minimizing maximum stretch would be desirable in a variety of settings, but the only work we were able to find on the subject is the paper by Bender et al. [5] that introduced the idea. They show that the best possible uniprocessor competitive ratio is between $\sqrt[3]{\Delta}/2$ and $\sqrt{\Delta}$. For the offline uniprocessor problem, they give a PTAS if the processor allows preemption, but give a nonpreemptive lower bound of $\omega(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless P=NP.

These results are summarized in Table 2.2.

	preemptive	non-preemptive
online	$\geq \sqrt[3]{\Delta}$ [5]	$\omega(n^{1-\epsilon})$ if P $\neq$ NP [5]
	$\leq \sqrt{\Delta}$ [5]	
offline	$(1+\epsilon)$ [5]	

Table 2.2: Approximibility of Uniprocessor Max Stretch

		preemptive	non-preemptive
1	online	≤ 2 [15]	$\geq \Delta$ [4]
	offline	$(1+\epsilon)$ [6]	???
m	online offline	≤ 14 [15]	???

Table 2.3: Approximibility of Average (or Total) Stretch

2.3.4 Average/Total Stretch

As may be expected, algorithms designed to minimize total flow also work well to minimize average or total stretch. SRPT is 2-competitive for average stretch of uniprocessor systems and 14-competitive for multiprocessor systems [15]. The algorithm by Awerbuch et al. [2] described in Section 2.3.1.2 is also $O(1)$-competitive [4]; Becchetti et al. [4] claim improvement of the constant from 63 to 37 in a tech report version. A variation of this algorithm, given by Chekuri et al. [9], is 9.82-competitive when migration is allowed and 17.32-competitive without it. An online PTAS is not possible for preemptive uniprocessor or multiprocessor average stretch [15], but Bender et al. [6] give one for the offline preemptive uniprocessor setting. When preemption is not allowed, Becchetti et al. [4] give a lower bound of Δ on the competitive ratio. These results are summarized in Table 2.3.

2.3.5 Error

Before moving on to our results, we would like to draw the reader's attention to what we believe is an error in the literature. The apparent error occurs in the proof of the following theorem by Vestjens [30]:

Theorem 8 *Any non-preemptive online algorithm is $\Omega(\sqrt{n})$-competitive.*

The proof is an adversary argument whose instance involves a job arriving according to the continuous probability density function $f(x) = (e/(e-1))e^{-x/\sqrt{n}}/\sqrt{n}$. In computing the expected optimal flow for a deterministic algorithm, Vestjens [30] evaluates the integrals

$$\int_0^S (x+1)f(x)dx + \int_S^{S+1} ((S+1)+(n-1)(S+1-x))f(x)dx \int_{S+1}^{\sqrt{n}} (S+1)f(x)dx$$

to get

$$S+1+\frac{e}{e-1}\left[\sqrt{n}(1-e^{-S/\sqrt{n}}) - S + (n-1)e^{-S/\sqrt{n}}(1-e^{-1/\sqrt{n}})\right] \tag{2.2}$$

and then claims that this expression is minimized by $S = 0$. Equation (2.2) actually decreases over the range $[0, \sqrt{n}]$. Presumably the probability density function was given incorrectly or the integral was evaluated incorrectly. In any case, the theorem is correct; we give an alternate proof in Section 3.5.2 that avoids the continuous probability density function. Epstein and van Stee [12] showed it to be true even when the processor is allowed to restart jobs.

Chapter 3

Online Setting

In this chapter we consider the competitiveness of online algorithms for total flow. In Section 3.1, we show that SRPT and SPT share a decomposition into blocks. In Section 3.2, we use this decomposition to define a schedule ECHO, that we show to be $(\Delta + 1)/2$-competitive. In Section 3.3, we show that SPT has flow no higher than ECHO, so it is also $(\Delta + 1)/2$-competitive. In Section 3.4, we apply these techniques to give the first known approximation for online multiprocessor total flow. Then, in Section 3.5, we give various lower bounds on the competitive ratio of online randomized algorithms.

3.1 Block Structure

In this section, we study a decomposition of the SRPT schedule. Define a *block* as either a *simple block* consisting of a single job running without preemption or a *preemption block*, beginning with a job that is preempted and ending when that job is finished. Although blocks have recursive structure, as alluded to by Theorem 7, we only consider blocks that begin while no preempted jobs are waiting. We now show that SPT also obeys this block structure:

Theorem 9 *Provided they start at the same time with the same set of available jobs and break ties between jobs of equal size in the same way, SPT completes the jobs in each block while SRPT is executing that block.*

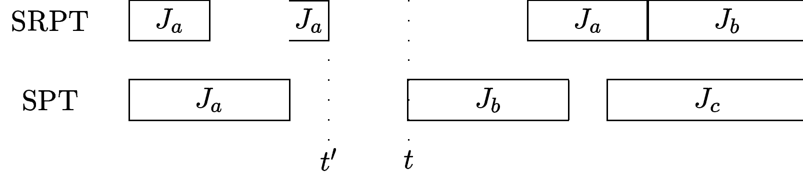


Figure 3.1: Illustration for the proof of Theorem 9

Proof. It suffices to prove that the algorithms complete the same jobs during the first block; the theorem then follows by induction.

For simple blocks, the schedules are identical since SPT and SRPT use the same criteria to choose a job with which to start a block.

For preemption blocks, we first argue that SPT never begins a job from a later block. Suppose this is not the case. Let t be the first time SPT begins a job that SRPT runs in a later block. Let J_a be the job beginning the current block, J_b be the job SPT starts at time t, and $p_a(t)$ be the remaining processing time of J_a at time t. Since SRPT placed J_b in a later block, $p_b > p_a(t)$. Without loss of generality, we assume at least one of the algorithms has started each job in the instance by time t. This implies that no jobs arrive after time t and that SRPT executes J_b immediately after finishing the block. SPT has the same final completion time, but finishes with some other job J_c. See Figure 3.1.

Let $t' \leq t$ be the latest time before t that SRPT worked on J_a. Since SRPT worked on J_a immediately prior to t', it had finished all jobs shorter than $p_a(t)$ available at that time. Because of this, the jobs SRPT worked on between t' and t arrived no earlier than t'. These jobs were shorter than $p_a(t)$ and thus shorter than p_b so SPT schedules them before J_b. Since SPT starts J_b at time t, only $t - t'$ such work arrives, so SRPT also finishes it at time t. Thus, SRPT resumes work on J_a at time t. It is not interrupted since no jobs arrive after t. Since finishing J_a ends the block for SRPT, it runs J_b next. When SPT finishes J_b, it runs 1 or more jobs, ending with J_c. Because $p_b > p_a(t)$, SRPT finishes J_a before SPT finishes J_b and thus starts J_b before SPT starts J_c. These jobs are the last to run so they finish simultaneously,

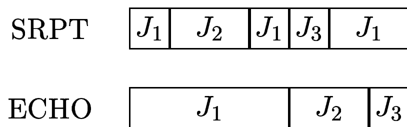


Figure 3.2: Example of an ECHO schedule

implying $p_b > p_c$. This relationship is not possible because SPT ran J_b instead of J_c and both must have been available because we have assume that no jobs are released after time t.

Thus, SPT works only on jobs from the current block. SRPT does not have jobs partially completed at the end of a block by Theorem 7. Thus, SPT has time to finish all the jobs in the block. The theorem follows because SPT generates busy schedules. □

3.2 Schedule ECHO

We define a schedule ECHO by giving a list of jobs that it executes one after another. For each block, the list begins with the first job in the SRPT schedule and then gives the other jobs in order of increasing completion time in the SRPT schedule, as shown in Figure 3.2. The overall list is a concatenation of the block lists. Because ECHO is formed by reordering jobs within blocks, it has the same block structure as SRPT and SPT.

To show that ECHO is a legal schedule, we need to show ECHO does not run a job before its release time. This is obvious for simple blocks and first jobs in preemption blocks. For other jobs in preemption blocks, we claim ECHO starts jobs SRPT has already finished. For SRPT at time t, let $l(t)$ be the work remaining on the block's first job, $p(t)$ be the work done on partially-complete jobs other than the first, and $w(t)$ be the work remaining on the currently-running job. We show that

$$w(t) \leq l(t) - p(t) \tag{3.1}$$

with strict inequality unless SRPT is working on the block's first job. This means that SRPT finishes a job before ECHO runs out of previously-scheduled work. The invariant (3.1) holds

before the first preemption because $w(t) = l(t)$ and $p(t) = 0$. Without preemptions, decreasing $w(t)$ compensates for changes in $l(t)$ or $p(t)$. When a job is preempted, the new job is shorter than $w(t)$ so the new $w(t)$ (its processing time) is less than $l(t) - p(t)$. By Theorem 7, preemptions behave like a stack, with the running job on top and preempted jobs below. When a job J_i is completed $l(t)$, $p(t)$, and $w(t)$ revert to the values they had when J_i began, again preserving the relationship.

Now consider the competitiveness of ECHO. Jobs in simple blocks finish at the same time in ECHO and SRPT. For a preemption block beginning with J_i, let x be the sum of sizes of jobs other than J_i. A lower bound on the SRPT flow is $p_i + 2x$. The worst case is that all jobs other than J_i are size 1, that gives ECHO a flow of $p_i + (1 + p_i)x$. (Without loss of generality, we assume processing times range from 1 to Δ.) This gives a competitive ratio of $(p_i + (p_i + 1)x)/(p_i + 2x)$. Since $p_i > 1$, this is less than $(p_i + 1)/2 \leq (\Delta + 1)/2$. Summing the result for each block gives the following:

Theorem 10 *ECHO is* $(\Delta + 1)/2$*-competitive for total flow.*

3.3 Shortest Processing Time

ECHO could be run online by simulating SRPT and remembering the order in which it finishes jobs, but this is unnecessary. We now show that SPT has the same competitive ratio:

Theorem 11 *SPT is* $(\Delta + 1)/2$*-competitive for total flow.*

Proof. We prove the theorem by showing that SPT generates a schedule no worse than ECHO for every problem instance. Theorem 10 then implies that SPT is $(\Delta + 1)/2$-competitive.

To compare SPT and ECHO, consider changing an SPT schedule into an ECHO schedule. By Theorem 9, it is sufficient to consider a single preemption block; the theorem then follows by induction. Without loss of generality, assume that SPT runs jobs in numerical order: J_1 followed by J_2 and so on, where J_1 is the job SRPT preempted to begin the block. The transformation from SPT to ECHO will take the form of repeatedly removing the first difference between them.

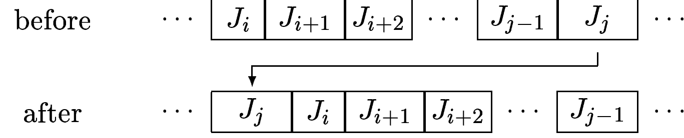


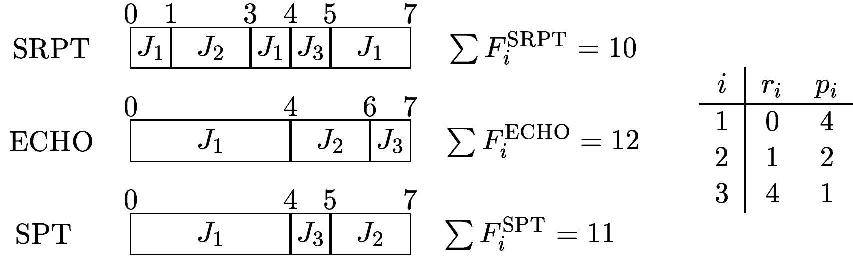
Figure 3.3: Example of a Slide Operation

Let the schedules first differ at time t, with SPT running J_i and ECHO running J_j for some $j > i$. This difference is removed by running J_j at time t and delaying $J_i, \ldots, J_{j-1}$ by p_j, "sliding" these jobs to make room for J_j, as illustrated in Figure 3.3. This step changes flow by $(j-i)p_j - \sum_{k=i}^{j-1} p_k = \sum_{k=i}^{j-1}(p_j - p_k)$. We will think of the change in the latter way, as a change for each inversion. Flow increases when a job moves ahead of a smaller job and flow decreases when it moves ahead of a larger job.

We now show that the transformation process results in a net increase of flow. For each slide that results in decreased flow, we find previous operations to "charge". Suppose sliding J_a affects only one larger job J_b. This means that SPT finished J_b before J_a while SRPT finished J_a before J_b. Thus, J_a had not arrived when SPT began J_b at some time t. Let X be the set of jobs SRPT finished by time t. Job J_1, the first job in the block, is not in X because finishing J_1 would have ended the block. Since SRPT spent time $p_1 - l(t)$ working on job J_1 and time $p(t)$ working on other unfinished jobs, it spent time $t - (p_1 - l(t)) - p(t)$ working on jobs in X. This quantity simplifies to

$$t - (p_1 - l(t)) - p(t) \geq t - p_1 + w(t) \geq t - p_1$$

by the invariant in Equation (3.1) and since $w(t) \geq 0$. If SRPT was working on J_1 at time t, the second inequality is strict since SRPT cannot finish J_1 by time t or the block would end. On the other hand, if SRPT was not working on J_1 at time t, then the first inequality is strict because Equation (3.1) gives strict inequality in this case. Thus, SRPT worked on jobs in X for more than $t - p_1$ time units prior to t.



i	r_i	p_i
1	0	4
2	1	2
3	4	1

Figure 3.4: Instance where SPT is better than ECHO

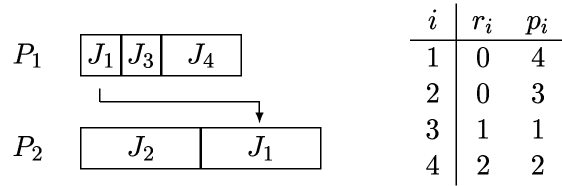
Since SPT spent p_1 time units working on J_1, it spent at most $t - p_1$ time units working on jobs in X. Thus, there is some job that SRPT finished by time t but SPT did not. All such jobs are longer than p_b because otherwise SPT would have run them before J_b (they were clearly available at time t because SRPT had finished them). Furthermore, all were finished before J_a in SRPT (and ECHO), but after J_b in SPT. Arbitrarily select one such job J_c and charge moving J_a before J_b to moving J_c before J_a. This leaves a net change of $(p_a - p_b) + (p_c - p_a) = p_c - p_b > 0$.

Now suppose a slide moves J_a ahead of several larger jobs $J'_1, J'_2, \ldots, J'_k$, listed in order of SPT execution. The argument above gives a job J_c to charge for moving J_a ahead of J'_1. Instead of charging all of J_c, however, we just charge it the length of J'_1. Then, while looking to charge for J'_2, disregard J'_1 from SPT and the charged portion of J_c from SRPT, allowing us to repeat the argument above. Continue inductively for the remaining J'_j. □

We have shown that SPT always generates a schedule at least as good as ECHO. The reverse is not true; Figure 3.4 gives an instance where SPT generates a strictly better schedule.

3.4 Multiprocessor Approximation

Now we consider the multiprocessor setting. In this setting, SRPT does not have the same block structure because preempted jobs can be restarted on different processors, as demonstrated in Figure 3.5. However, we can apply the technique described above to the schedule of Awerbuch et al. [2] described in Section 2.3.1.2. Blocks can be defined for each processor of this schedule in the same way we defined them for SRPT. Using these blocks, a version of ECHO can



i	r_i	p_i
1	0	4
2	0	3
3	1	1
4	2	2

Figure 3.5: Instance Where the SRPT Schedule Migrates a Job

be run; each processor runs the first job of a block, followed by the others in order of increasing completion time. By the argument in the proof of Theorem 10, the total flow of jobs run on each processor increases by a factor of $O(\Delta)$, for an overall competitive ratio of $O(\Delta \log \min\{n, \Delta\})$. No online approximation was previously known for the nonpreemptive multiprocessor case; the best known offline approximation is $O\left(\sqrt{n/m}\log(n/m)\right)$ on m processors, due to Leonardi and Raz [22].

3.5 Lower Bounds

Now we change modes from showing that algorithms achieve a given competitive ratio to giving lower bounds on the competitive ratio that can be achieved by any algorithm. In Section 3.5.1, we show an asymptotic lower bound of $(\Delta + 1)/2$ for deterministic algorithms. Since this matches the competitive ratio of ECHO and SPT, these algorithms are both optimally competitive. Section 3.5.2 gives an $\Omega(\sqrt{\Delta})$ lower bound for randomized algorithms. Section 3.5.3 shows that lower bounds of $\Omega(\sqrt[4]{n})$ and $\Omega(\sqrt[4]{\Delta})$ hold even when the algorithm is allowed to restart jobs. Section 3.5.4 considers busy randomized algorithms and gives a lower bound of $(\Delta + 6)/8$. Since ECHO and SPT are both busy algorithms, this shows that randomization does not allow more than a constant factor improvement for busy algorithms.

After discovering these results, we found an $\Omega(\sqrt{n})$ lower bound for randomized algorithms that use restarts due to Epstein and van Stee [12]. A slight modification of their proof to remove jobs of length 0 gives a lower bound of $\Omega(\sqrt{\Delta})$. This result is stronger than Theorem 13 in Section 3.5.2 because it allows restarts and gives a better bound than Theorem 14 in Section

3.5.3. Thus, the results of Sections 3.5.2 and 3.5.3 are included purely to illustrate alternate lower bound constructions.

3.5.1 Deterministic Scheduling

Now we give lower bound arguments, beginning with the deterministic uniprocessor setting.

Consider the following adversarial job instance: A job with processing time Δ is released at time 0. When the algorithm starts this job, the adversary releases a job with processing time 1 each of the next $n-1$ time units. The algorithm will have flow at least $\Delta + (\Delta+1)(n-1)$. Meanwhile, the optimum solution delays the long job for at most $\Delta + (n-1)$ time units and runs the short jobs as they arrive, for a total flow of $2(n-1)+2\Delta$. The ratio between these two quantities approaches $(\Delta+1)/2$ as $n \to \infty$, proving the following theorem:

Theorem 12 *For any fixed Δ, all online deterministic uniprocessor algorithms are at least $(\Delta+1)/2$-competitive for total flow.*

3.5.2 Randomized Scheduling

Now we bound the competitiveness of a randomized uniprocessor algorithm.

Theorem 13 *Any randomized online algorithm for minimizing total flow time on a single machine has an $\Omega(\sqrt{\Delta})$ competitive ratio.*

Before proving Theorem 13, we give an alternate proof of Theorem 8, the randomized $\Omega(\sqrt{n})$ lower bound, that uses jobs with processing time 0.

We use Yao's Lemma and bound the competitiveness of an optimal deterministic algorithm on a randomized input instance. To create the input instance, select $x \in \{1,2\}$ uniformly at random. Depending on this choice, the input consists of the following jobs:

- a job with processing time 1 released at time 0, and
- $n-1$ jobs of processing time 0, released in groups of $\sqrt{n-1}$ each, at time $i/2$ for each $i = 1, 2, \ldots, \sqrt{n-1}$ where $i \neq x$.

This instance partitions the first $\sqrt{n-1}/2$ units of time into intervals of size $1/2$ separated by groups of jobs with processing time 0.

The optimal solution is to run the job with processing time 1 starting either at time 0 or at time $1/2$ so that it runs in the pair of intervals whose dividing group was not included because of the choice of x. This solution costs at most $3/2 = O(1)$.

If the online algorithm starts running the long job before time $1/2$ and $x = 2$, the first group of jobs is delayed by at least $1/2$. This gives an expected total flow of at least $Pr[x = 2]\sqrt{n-1}/2 = \sqrt{n-1}/4 = \Omega(\sqrt{n})$.

If the online algorithm waits until time $1/2$ before starting the long job and $x = 1$, it is too late to fit the long job into any of the remaining intervals. In this case, the algorithm either delays the long job until at least time $\sqrt{n-1}$, for expected total flow of at least $Pr[x = 1]\sqrt{n-1} = \sqrt{n-1}/2 = \Omega(\sqrt{n})$, or delays running one of the groups of short jobs by at least $1/2$, for expected total flow at least $Pr[x = 1]\sqrt{n-1}/2 = \sqrt{n-1}/4 = \Omega(\sqrt{n})$. Thus, the online algorithm's expected total flow is $\Omega(\sqrt{n})$ and there is an $\Omega(\sqrt{n})$ lower bound on approximating total flow.

To prove a lower bound in terms of Δ rather than n, we use very short jobs to replace the jobs with processing time 0. More specifically, a group of $\sqrt{n-1}$ jobs with processing time 0 released at time t is replaced by $\sqrt{n-1}$ jobs with processing time $1/(n-1)$ each, released at $t + (i/(n-1))$ for each $i = 0, 1, \ldots, \sqrt{n-1} - 1$.

The optimal solution increases by $(n-1)/(n-1) = 1$ because the short jobs now have non-zero flow and an additional $\sqrt{n-1}(\sqrt{n-1}/(n-1)) = 1$ because the long job starts $\sqrt{n-1}/(n-1)$ later in order to finish the stream that arrives at its old starting time, delaying the stream that arrives at its old ending time by this much. Even with these additions, the optimal cost remains constant. The online algorithm's expected cost remains $\Omega(\sqrt{n})$ and so we still have an $\Omega(\sqrt{n})$ lower bound. Since $\Delta = n - 1$ in the input instance given above, this translates into an $\Omega(\sqrt{\Delta + 1}) = \Omega(\sqrt{\Delta})$ lower bound on the competitive ratio and the theorem is proven.

3.5.3 Randomized Scheduling with Restarts

When the machine allows restarts, we can still prove the following bound:

Theorem 14 *Any randomized online algorithm for minimizing total flow time on a single machine has an* $\Omega(\sqrt[4]{n})$ *and* $\Omega(\sqrt[4]{\Delta})$ *competitive ratio even if it is allowed to restart jobs.*

Proof. Using Yao's Lemma, we consider the performance of a deterministic algorithm on a randomized adversarial input. We first give a version that uses jobs with processing time 0. Let $x \in \{1, 2, \ldots, \sqrt[4]{n-1}\}$ be chosen uniformly at random. Then the input instance consists of the following jobs:

- One job with processing time 1 released at time 0. We call this the *long job*.
- $(n-1)^{3/4}$ *divider jobs* with processing time 0 released in groups of $\sqrt{n-1}$ at time i for each $i = 1, 2, \ldots, \sqrt[4]{n-1}$.
- $\sqrt{n-1}$ *tripwire jobs* with processing time 0 released in groups of $\sqrt[4]{n-1}$ at time $i + 1/3$ for each $i = 0, 1, 2, \ldots, \sqrt[4]{n-1} - 1$.
- $(n-1)^{3/4} - \sqrt{n-1}$ *trap jobs* with processing time 0 released in groups of $\sqrt{n-1}$ at time $i + 2/3$ if $i \neq x$ for each $i = 0, 1, 2, \ldots, \sqrt[4]{n-1} - 1$
- $n - 1 - 2(n-1)^{3/4}$ *tail jobs* with processing time 0 released in groups of $\sqrt{n-1}$ at time $\sqrt[4]{n-1} + i/2$ for each $i = 0, 1, \ldots, \sqrt{n-1} - 2\sqrt[4]{n-1}$.

The total number of jobs is $1 + (n-1)^{3/4} + \sqrt{n-1} + (n-1)^{3/4} - \sqrt{n-1} + n - 1 - 2(n-1)^{3/4} = n$.

If all the short jobs execute as soon as they arrive, the dividers separate each of the first $\sqrt[4]{n-1}$ units of time into a separate interval. Each interval has a tripwire group and all but one has a trap group. Following the intervals, a long stretch of the time line is divided into shorter intervals by the tail jobs. The divider, trap, and tail groups contain too many jobs to be delayed. This setup means that the long job must be run in the interval without a trap group, but an online algorithm cannot determine if an interval contains a trap group until after it has delayed that interval's tripwire group.

The optimal solution is to run the long job in the interval without a trap group, i.e. the one indicated by x. This causes the long job to wait at most $\sqrt[4]{n-1}-1 = O(\sqrt[4]{n})$ and the tripwire jobs in that interval to wait 2/3 each, for $O(\sqrt[4]{n})$ total flow time.

An online algorithm wishing to be better than $\Omega(\sqrt[4]{n})$-competitive cannot displace any of the divider, trap, or tail groups in order to place the long job. Nor can it delay the long job until after the tail. It is forced to try running the long job in some intervals, delaying the tripwire jobs, and then aborting the long job if the trap jobs appear in that interval. We show that whichever intervals are selected for this technique, the expected flow time is $\Omega(\sqrt{n})$. If the interval indicated by x is not selected, then clearly the algorithm has $\Omega(\sqrt{n})$ flow. Since each interval is indicated by x with probability $1/\sqrt[4]{n-1}$, each interval not selected by the algorithm contributes $\Omega(\sqrt[4]{n})$ expected flow. On the other hand, each selected interval that is before the interval indicated by x causes algorithm to delay the tripwire jobs by a constant amount, which increases flow by $\Omega(\sqrt[4]{n})$. With probability 1/2, the interval indicated by x is in the second half of the intervals, meaning that the first $\sqrt[4]{n-1}/2$ intervals contribute $\Omega(\sqrt[4]{n})$ each to the expected flow. Thus, we have a constant probability of $\Omega(\sqrt{n})$ expected flow so the expected flow is at least this large and we have an $\Omega(\sqrt[4]{n})$ lower bound on the approximation ratio.

To turn this argument into a lower bound in terms of Δ, we replace the jobs having processing time 0 with streams of jobs having processing time $1/(n-1)$. Since these streams take non-zero time to complete, the divider, tripwire, trap, and tail jobs should now be released at times of the form $i\left(1+1/\sqrt{n-1}\right)$, $i+(i-1)/\sqrt{n-1}+1/3$, $i+(i-1)/\sqrt{n-1}+2/3$, and $\sqrt[4]{n-1}+1/\sqrt[4]{n-1}+i/2$. The same reasoning as above gives an $\Omega(\sqrt[4]{n})$ bound. Since now $\Delta = n-1$, the $\Omega(\sqrt[4]{\Delta})$ bound follows immediately. □

3.5.4 Randomized Busy Scheduling

Although it seems possible that randomness will allow a significantly-improved competitive ratio in the general case, we now show a linear lower bound for the special case of busy schedules:

Theorem 15 *Any busy uniprocessor online algorithm is asymptotically at least $(\Delta + 6)/8$-competitive for total flow.*

Proof. Using Yao's Lemma, we consider the performance of a deterministic algorithm on a randomized adversarial input. Let $x \in \{0, \Delta/2\}$ be chosen uniformly at random. Two jobs of the instance with processing time Δ and $\Delta/2$ are released at time 0. The remaining $n-2$ jobs all have processing time 1 and are released at $\Delta/2 + x + i$ for each $i = 0, \ldots, n-2$.

If $x = 0$, the optimal schedule is to run the job with processing time $\Delta/2$, then the short jobs, and then the job with processing time Δ. This gives a flow of $2\Delta + 2(n-2)$. Incorrectly beginning with the job having processing time Δ gives a flow of at least $(n+3)\Delta/2 + 2(n-2)$.

If $x = \Delta/2$, the optimal schedule is to run the job with processing time Δ, then the short jobs, and then the job with processing time $\Delta/2$. This gives a flow of $(5/2)\Delta + 2(n-2)$. Starting with the job having processing time $\Delta/2$ instead gives a flow of at least $(n+2)\Delta/2 + n - 2$. Thus, the expected optimal flow is

$$\frac{1}{2}(2\Delta + 2(n-2)) + \frac{1}{2}((5/2)\Delta + 2(n-2)) = \frac{9}{4}\Delta + 2(n-2)$$

If an algorithm chooses to run the job with processing time $\Delta/2$ first, its expected flow is

$$\frac{1}{2}(2\Delta + 2(n-2)) + \frac{1}{2}\left(\frac{(n+2)\Delta}{2} + n - 2\right) = n\left(\frac{\Delta}{4} + \frac{3}{2}\right) + \frac{3}{2}\Delta - 3$$

If it chooses to run the job with processing time Δ first, its expected flow is

$$\frac{1}{2}\left(\frac{(n+3)\Delta}{2} + 2(n-2)\right) + \frac{1}{2}\left(\frac{5}{2}\Delta + 2(n-2)\right) = n\left(\frac{\Delta}{4} + 2\right) + 2\Delta - 4$$

The first choice is better and it yields a competitive ratio that approaches $(\Delta+6)/8$ as $n \to \infty$.

□

Chapter 4

Lower Bounds in the Offline Busy Setting

Now we turn to offline scheduling problems. This chapter gives approximability lower bounds for busy algorithms. Not only do busy algorithms have intuitive appeal when trying to minimize total flow time, sometimes being busy is a requirement, as in the case at Sandia National Laboratories, where there is political pressure to ensure that expensive computing hardware has high utilization rates [16, 17]. Unfortunately, the theoretical news on such algorithms is quite bad. This section gives uniprocessor and multiprocessor approximability lower bounds for busy algorithms that are worse than the approximation ratios of known nonbusy algorithms to solve these problems.

4.1 Uniprocessor

The best approximability lower bound for offline nonpreemptive uniprocessor total flow is due to Kellerer et al. [20], which shows $n^{1/2-\epsilon}$-inapproximability unless P=NP. This result, presented as Theorem 4 in Section 2.2.2.1, is based on a construction that uses small jobs to partition the time axis into "bins" and additional small jobs to penalize algorithms that delay

jobs past these bins. When considering only busy algorithms, the second type of small jobs is not necessary, allowing the following improvement:

Theorem 16 *No polynomial-time algorithm that always generates busy schedules approximates offline uniprocessor nonpreemptive minimum total flow time to within a factor of $n^{1-\epsilon}$ or $\Delta^{1-\epsilon}$ for any constant $\epsilon > 0$ unless P=NP.*

Proof. To show the bound in terms of n, we prove the equivalent statement that nonpreemptive total flow cannot be approximated to within a factor of $n^{1-1/\alpha}$ for any positive integer α unless P=NP.

As in the proof of Theorem 4, we use a reduction from N3DM. Turn the a_i, b_i, and c_i from the instance of N3DM into "large" jobs of processing times $2r + ai$, $4r + b_i$, and $8r + c_i$ respectively. Release all these jobs at time 0. In addition, "bins" are formed by streams of "small" jobs that begin at $(14r + D + 1)i - 1$ for each $1 \leq i \leq k$. Each stream consists of rg jobs, each with processing time $1/rg$ and with release times $1/rg$ apart.[1]

The resulting scheduling instance has $3k$ large jobs and krg small ones. For a given value of α, we use the values

$$\begin{aligned} n &= 53^{\alpha^2} k^{3\alpha^2} D^{\alpha^3+\alpha^2}, && (4.1) \\ r &= D n^{(1/\alpha)(1-1/\alpha)}, && (4.2) \\ g &= 50k^2 r^{\alpha} && (4.3) \end{aligned}$$

so the total number of jobs is

$$3k + krg = 3k + 50k^3 r^{\alpha+1} < 53k^3 r^{\alpha+1} = 53k^3 D^{\alpha+1} n^{1-1/\alpha^2} = n$$

Suppose that the instance of N3DM does not have a solution. By construction, this means that at least 2 of the bins between streams of small jobs cannot be exactly filled. Consider the

[1] If a scheduling instance with integer processing times and release times is desired, scale up the entire instance by a factor of rg.

first such bin. This bin cannot be underfilled without producing a non-busy schedule. If it is overfilled, then the stream of small jobs denoting the end of the bin must be delayed at least 1 time unit. Therefore, these rg jobs have combined flow at least rg.

Now suppose that the instance of N3DM has a solution. Construct a schedule where each small job begins execution as soon as it arrives. Schedule a_i, $b_{\pi(i)}$, and $c_{\psi(i)}$ one after another at $(14r+D+1)(i-1)$, $(14r+D+1)(i-1)+2r+a_i$, and $(14r+D+1)(i-1)+6r+a_i+b_{\pi(i)}$. These three jobs then fit into bin i, between small job streams $i-1$ and i. This schedule is busy because each triple of jobs in a bin has combined processing time $(2+4+8)r+D = 14r+D$, exactly filling the gap between streams of small jobs. This schedule has total flow time

$$k + \left(3\sum_{i=1}^{k}(14r+D+1)(i-1)\right) + \left(3\sum_{i=1}^{k}(2r+a_i) + 2\sum_{i=1}^{k}(4r+b_{\pi(i)}) + \sum_{i=1}^{k}(8r+c_{\psi(i)})\right)$$

where the three terms are the total flow of the small jobs, the flow accumulated by large jobs waiting for their bins, and the flow of large jobs within their assigned bins. Since $a_i + b_{\pi(i)} + c_{\psi(i)} = D$ and $r > D$, this is at most $k + 24rk(k-1) + 25rk < 50rk^2$.

If we run an $n^{1-1/\alpha}$ approximation algorithm on a scheduling instance when the original N3DM instance has a solution, it generates a solution with total flow at most $n^{1-1/\alpha}50rk^2$. Since $r^\alpha = D^\alpha n^{1-1/\alpha}$, this is $50r^{\alpha+1}k^2/D^\alpha = rg/D^\alpha$, less than the flow that must occur if the N3DM instance did not have a solution. Thus, an $n^{1-1/\alpha}$ approximation algorithm allows us to decide N3DM, meaning that it is NP-hard to approximate the scheduling problem to this accuracy.

To get the bound in terms of Δ, observe that

$$\begin{aligned} \Delta &= O(r^2 g) = O(r^2k^2D^\alpha n^{1-1/\alpha}) = O(k^2D^{\alpha+2}n^{1+1/\alpha-2/\alpha^2}) \\ &= O(k^2D^{\alpha+2}n^{1+1/\alpha}) = O(n^{1+1/\alpha}) \end{aligned}$$

Plugging this into the bound $n^{1-1/\alpha}$ shows that total flow cannot be approximated to a factor of $\Delta^{(1-1/\alpha)/(1+1/\alpha)} = \Delta^{1-2/(\alpha+1)}$, which approaches Δ closer than any constant power. □

4.2 Multiprocessor

The best approximability lower bound for offline nonpreemptive multiprocessor total flow is due to Leonardi and Raz [22], who show $n^{1/3-\epsilon}$-inapproximability unless P=NP. This result is presented as Theorem 5 in Section 2.2.2.2. As in the previous section, restricting consideration to busy algorithms allows some jobs to be removed from the lower bound construction, yielding the following improved bound:

Theorem 17 *No polynomial-time algorithm that always generates busy schedules approximates offline multiprocessor nonpreemptive minimum total flow time to within a factor of $n^{1-\epsilon}$ or $\Delta^{1-\epsilon}$ for any $\epsilon > 0$ unless P=NP.*

Proof. To show the bound in terms of n, we prove the equivalent statement that nonpreemptive total flow cannot be approximated to $n^{1-1/\alpha}$ for any positive integer α unless P=NP.

To create a scheduling instance, turn the a_i, b_i, and c_i from the instance of N3DM into "large" jobs of processing times $2r + a_i$, $4r + b_i$, and $8r + c_i$ respectively.[2] These jobs are all released at time 0. In addition, m streams of small jobs are released at $l(14r + D + 1)$ for each $l = 1, \ldots, k/m$. Each stream consists of rg jobs with processing time $1/rg$, one released each $1/rg$ time units.

The proof now follows from essentially the same argument as Theorem 16. The instance created has $3k + krg$ jobs. When the N3DM instance has a solution, the total flow is

$$k + \left(3\sum_{i=1}^{k/m}(14r + D + 1)(i-1)\right) + \left(3\sum_{i=1}^{k}(2r + a_i) + 2\sum_{i=1}^{k}(4r + b_{\pi(i)}) + \sum_{i=1}^{k}(8r + c_{\psi(i)})\right)$$

which is at most $k + 24r(k/m)(k/m-1) + 25rk$, still less than $50rk^2$. When the N3DM instance does not have a solution, one of the streams of small jobs is displaced. This is somewhat less costly than in the uniprocessor case because these short jobs can be run on different processors, but displacing a stream still cause one extra job to be waiting for each $1/rg$ time units of the

[2]The factor of m in the processing times used in Theorem 5 is no longer necessary because there is no "tail" of jobs in this proof.

stream that have passed. Thus, i extra jobs wait in the interval $\left(\frac{i-1}{rg}, \frac{i}{rg}\right]$, for each $i = 1, \ldots, rg$. This extra waiting contributes $(1/rg)\sum_{i=1}^{rg} i > rg/2$ total additional flow.

We use the values for n, r, and g given in Equations (4.1), (4.2), and (4.3). Therefore, when the N3DM instance has a solution, an $n^{1-1/\alpha}$ approximation has flow at most rg/D^{α}. Because $\Delta > 2$ and $\alpha \geq 1$, this is still less than the optimal flow of an instance generated when N3DM does not have a solution. We conclude that an $n^{1-1/\alpha}$ approximation allows us to solve the N3DM decision problem, giving the stated bound in terms of n. The argument at the end of the proof of Theorem 16 above suffices to convert this into the bound in terms of Δ. □

Chapter 5

Conclusions and Open Problems

5.1 Conclusions

We have increased understanding of scheduling to minimize total flow by considering approximability in terms of the parameter Δ. In the online setting, our most important results are showing that SPT is $(\Delta+1)/2$-competitive and that SRPT and SPT share a decomposition into blocks. We also show that $(\Delta+1)/2$ is the best possible competitive ratio for a deterministic algorithm. It is also nearly the best possible for a randomized busy algorithm; we show a lower bound of $(\Delta+6)/8$ for this case. For general randomized algorithms and general randomized algorithms that allow restarts, we give alternate proofs for the previously-known lower bounds $\Omega(\sqrt{\Delta})$ and $\Omega(\sqrt[4]{\Delta})$.

In the offline setting, we modified inapproximability proofs for the general problem to strengthen them for the special case of busy algorithms. On both uniprocessors and multiprocessors, we showed that it is NP-hard to approximate total flow to within $\Delta^{1-\epsilon}$ or $n^{1-\epsilon}$ for any constant $\epsilon > 0$.

5.2 Open Problems

Despite the progress we have made, a number of open problems remain.

5.2.1 Randomized Approximation of Flow

The most obvious open problem is to close the gap between SPT, the most competitive deterministic algorithm with a ratio of $\Theta(\Delta)$, and the randomized lower bound of $\Omega(\sqrt{\Delta})$ we show in Section 3.5.2. We conjecture that the lower bound can be improved to something linear in Δ even for a randomized algorithm.

5.2.2 Multiprocessor Setting

An obvious question arising from Chapter 3 is how well online nonpreemptive multiprocessor total flow can be approximated. SPT and the $O(\Delta \log \min\{n, \Delta\})$-competitive algorithm given in Section 3.4 are candidates for further study. The later is the only algorithm known to be competitive for this problem, but we were not able to come up with lower bounds worse than the best known $\Omega(\Delta/m)$. We suspect that a competitive ratio linear in Δ can be achieved.

A potential subproblem is to identify structural properties of optimal schedules, SRPT, or the optimally-competitive algorithm given by Awerbuch et al. [2]. As mentioned previously, multiprocessor SRPT does not have the block structure we exploited in this thesis, but perhaps some generalization of it can be shown to hold.

5.2.3 Other Parameters

In this work, we showed that SPT performs well for total flow as long as the range of job processing times is small, even if many jobs are submitted. By using the parameter Δ, we were able to characterize a hard instance. It would be interesting to see if bounded competitive ratios exist in terms of other parameters, both for total flow and other scheduling problems.

Another possible parameter for flow is one that somehow captures variation in the density of job arrivals. The use of such a parameter is motivated by the typical lower bound example, which involves some relatively large jobs and then a stream of short jobs arriving very close together. Bounding a parameter like this may essentially give the algorithm assurances that its

input will not misbehave too horribly and could be similar to *nearly online* problems, in which some knowledge of the next job to arrive is given (see Labetoulle et al. [21]).

5.2.4 Smoothed Analysis

We would also be very interested in the smoothed analysis of a scheduling problem. *Smoothed analysis* was introduced by Spielman and Teng [28] as a way of explaining the empirically good performance of the simplex algorithm for linear programming even though its worst-case performance is superpolynomial. Smoothed analysis is worst-case analysis of an arbitrary input instance after it has been subjected to a slight random perturbation. Its use can be justified for scheduling problems because the process of submitting jobs is somewhat noisy and given job durations are generally estimates. Scheduling problems may have relatively low smoothed complexity because lower bound arguments generally involve jobs being released at precise intervals to form streams.

One way smoothed analysis might be applied to scheduling is to vary job durations. When no estimate of job duration is given, the problem is nonclairvoyant scheduling, discussed in Section 2.3.1.2, but the case where the given durations are only somewhat incorrect, either adversarially or randomly, has not been considered to our knowledge.

Bibliography

[1] E. Arkin, M. Bender, J. Mitchell, and S. Skiena. The lazy bureaucrat scheduling problem. *Information and Computation*, 2002.

[2] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing flow time without migration. In *Proc. 31st Symp. on Theory of Computation*, pages 198–205, 1999.

[3] N. Bansal and M. Harchol-Balter. Analysis of SRPT scheduling: Investigating unfairness. In *SIGMETRICS 2001 Conference on Measurement and Modeling of Computer Systems*, pages 279–290, 2001.

[4] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Scheduling to minimize average stretch without migration. In *Proc. 11th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 548–557, 2000.

[5] M. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 270–279, 1999.

[6] M. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proc. 13th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 762–771, 2002.

[7] J. Blażewicz, J. Lenstra, and A. R. Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[8] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *Proc. 34rd Symp. on Theory of Computation*, pages 297–305, 2002.

[9] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *Proc. 33rd Symp. on Theory of Computation*, pages 84–93, 2001.

[10] P. Crescenzi and V. Kann. A compendium of NP optimization problems. `http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html`, 2002.

[11] M. Drozdowski. Scheduling multiprocessor tasks — an overview. *European Journal of Operational Research*, 94:215–230, 1996.

[12] L. Epstein and R. van Stee. Lower bounds for on-line single-machine scheduling. In *Proc. 26th Symp. Math. Found. Comput. Sci.*, number 2136 in LNCS, pages 338–350. Springer-Verlag, 2001.

[13] L. Epstein and R. van Stee. Optimal on-line flow time with resource augmentation. In *Proc. 13th Symp. Fund. Comp. Theory*, number 2138 in LNCS, pages 472–482. Springer-Verlag, 2001.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[15] J. Gehrke, S. Muthukrishnan, R. Rajaraman, and A. Shaheen. Online scheduling to minimize average stretch. In *Proc. 40th Symp. Found. Computer Science*, pages 433–442, 1999.

[16] General Accounting Office (GAO) Reports. Information technology: Department of Energy does not effectively manage its supercomputers. GAO/RCED-98-208, 1998.

[17] General Accounting Office (GAO) Reports. Supporting congressional oversight: Framework for considering budgetary implications of selected GAO work. GAO/RCED-01-447, 2001.

[18] R. Graham, E. Lawler, J. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Math*, 5:287–326, 1979.

[19] B. Kalyanasundaram and K. Pruh. Minimizing flow time nonclairvoyantly. In *Proc. 38th Symp. Found. Computer Science*, pages 345–352, 1997.

[20] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *Proc. 28th Symp. on Theory of Computing*, pages 418–426, 1996.

[21] J. Labetoulle, E. Lawler, J. Lenstra, and A. R. Kan. Preemptive scheduling of uniform machines subject to release dates. In *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.

[22] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. 29th Symp. on Theory of Computation*, pages 110–119, 1997.

[23] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130:17–47, 1994.

[24] L. Schrage. A proof of the optimality of the shortest processing remaining time discipline. *Operations Research*, 16:687–690, 1968.

[25] J. Sgall. *On-line scheduling — A Survey*, pages 196–231. Number 1442 in LNCS. Springer-Verlag, 1998.

[26] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley, 1998.

[27] D. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 26(1):197–199, 1976.

[28] D. A. Spielman and S.-H. Teng. Smoothed analysis: Why the simplex algorithm usually takes polynomial time. In *Proc. 33rd Symp. on Theory of Computation*, pages 296–305, 2001.

[29] B. Veltman, B. Lageweg, and J. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[30] A. Vestjens. *On-line scheduling.* PhD thesis, Eindhoven University of Technology, Nov 1997.

[31] A. Yao. Towards a unified measure of complexity. In *Proc. 18th Symp. Found. Computer Science*, pages 222–227, 1977.