Online Weighted Flow Time and Deadline Scheduling

Luca Becchetti * *Stefano Leonardi* * *Alberto Marchetti–Spaccamela* *

Kirk R. Pruhs †

Abstract

In this paper we study some aspects of Weighted Flow Time on parallel machines. We first show that the online algorithm Highest Density First is an $O(1)$-speed $O(1)$-approximation algorithm for $P|r_i, pmtn|\sum w_i F_i$. We then consider a related Deadline Scheduling Problem that involves minimizing the weight of the jobs unfinished by some unknown deadline D on a uniprocessor. We show that any c-competitive online algorithm for weighted flow time must also be c-competitive for Deadline Scheduling. We finally give an $O(1)$-competitive algorithm for Deadline Scheduling.

1 Introduction

We consider several special cases and variations of the online scheduling problem $P|r_i, pmtn|\sum w_i \cdot F_i$. In this problem a sequence of jobs has to be processed on a set of m identical machines. The ith job has a release time r_i, a processing time or length x_i and a non-negative weight w_i. At any time, only one processor may be running any particular job. Processors may preempt one job to run another. The ith job is completed after it has been run for x_i time units. The flow time F_i of the ith job is the difference between its completion time and its release date. The objective function to minimize is the weighted flow time $\sum w_i \cdot F_i$.

The main motivation for studying Weighted Flow Time is that by far the most commonly used measure of system performance is average total flow/response/waiting time, or equivalently average user perceived latency. This motivates the study of the weigthed case.

Another motivation for considering weighted flow time is that it is a special case of broadcast scheduling. The setting we refer to is a client-server system where the server uses broadcast communication to download files to the clients. One notable commercial example of such a system is Hughes' DirecPC system; In this system the clients request web pages over phone lines, and the web pages are broadcasted via satellite to all clients (so it may be possible to satisfy several clients request with a single broadcast). The weighted flow time problem $1|r_i, pmtn|\sum w_i \cdot F_i$ is a special case of the broadcast problem where all requests for file i arrive at a single time r_i (however, there may be many requests for i at time r_i).

There have been some recent results on minimizing average flow time [1, 8, 10, 4]; however, minimizing weighted flow time is not yet well understood. In fact, the uniprocessor offline problem

*Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza". {becchetti,leon,alberto}@dis.uniroma1.it. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), IST-2000-14084 (APPOL), IST-1999-10440 (Brahms) and by the MURST Projects "Algorithms for Large Data Sets: Science and Engineering" and "Resource Allocation in Computer Networks".

†Computer Science Department, University of Pittsburgh. kirk@cs.pitt.edu. Supported in part by NSF grant CCR-9734927 and by a grant from the US Airforce.

$1|r_i, pmtn|\sum w_i \cdot F_i$ is known to be NP-hard [5] but there is no offline polynomial-time constant approximation algorithm we are aware of.

Resource augmentation analysis was proposed [9] as a method for analyzing scheduling algorithms that are hard to approximate. In this paper we adopt the notation and terminology of [12] and we say that an *s-speed c-approximation* algorithm A has the property that the value of the objective function of the schedule that A produces with speed s ($s > 1$) processors is at most c times the optimal value of the objective function for speed 1 processors. Standard competitive analysis for online algorithms assumes that both the online and the optimal offline algorithm use unit speed processors. So, with respect to the terminology we use when referring to resource augmentation analysis, A is *c-competitive* if it is a 1-speed c-approximation algorithm. One positive result for weighted flow time on one machine is an LP-based online algorithm, Preemptively-Schedule-Halves-by-$\bar{M}_j$, that is known to be an $O(1)$-speed $O(1)$-approximation algorithm [12] for the problem.

The contribution of this paper is twofold. In section 2, we extend and simplify the resource augmentation analysis of $1|r_i, pmtn|\sum w_i F_i$ given in [12] and we extend it to the multiprocessor case. Namely, we show that the polynomial time, online algorithm Highest Density First (HDF), which always runs the job that maximizes its weight divided by its length, is an $O(1)$-speed $O(1)$-approximation algorithm. While this analysis of HDF is not stated in [12], upon reflection one can see that all the insights necessary to assert that HDF is an $O(1)$-speed $O(1)$-approximation algorithm on a single processor are inherent in their analysis of their proposed algorithm Preemptively-Schedule-Halves-by-$\bar{M}_j$. Besides making this result explicit, our proof has the advantages that (1) our analysis is from first principles and does not require understanding of a rather complicated LP lower bound, and (2) our analysis also easily extends to the multiprocessor problem $P|r_i, pmtn|\sum w_i F_i$, while the result in [12] apparently does not.

In section 3 we consider standard (1-speed) competitive analysis of $1|r_i, pmtn|\sum w_i F_i$. We first show that every c-competitive online algorithm A has to be *locally c-competitive*, that is, at every time t, the overall weight of the jobs that A has not finished by time t can be at most c times the minimum possible weight of the unfinished jobs at time t. The requirement of local competitiveness suggests that we consider what we call the Deadline Scheduling Problem (DSP).

The input to DSP consists of n jobs, with each job having a length and a non-negative weight. Both the length of a job and its weight are revealed to the algorithm when the job is released. The goal is to construct a schedule (linear order) where the overall weight of the jobs unfinished by some initially unknown deadline D is minimized. In the standard three field scheduling notation, one might denote this problem as $1|d_i = D|\sum w_i \cdot U_i$. The requirement of local competitiveness means that any c-competitive online algorithm for $1|r_i, pmtn|\sum w_i F_i$ must be c-competitive for DSP. The DSP problem can be seen as a dual to the deadline scheduling problem considered in [7]. The setting considered in [7] was the same as DSP, except that the goal was to maximize the jobs completed before the deadline D on multiple machines. In [7] it is shown that a greedy algorithm CSM is essentially strongly (non-constant) competitive for the maximization problem. The second main contribution of this paper is an online $O(1)$-competitive algorithm for DSP on one machine.

2 Resource Augmentation Analysis of $P|r_i, pmtn|\sum w_i F_i$

In this section, we adopt the following notation. We use $A(s)$ to denote the schedule produced by algorithm A with a speed s processors, and abuse notation slightly by using A to denote $A(1)$. Let $U^A(t)$ be the jobs that have been released before time t, but not finished by algorithm A by time t. We use $y_j^A(t)$ to denote the remaining unprocessed length of job j at time t according to the schedule produced by algorithm A. Similarly, we use $p_j^A(t) = p_j - y_j^A(t)$ to denote the processed length of job j before time t. So from time t, algorithm A, with a speed s processor, could finish j in $y_j^A(t)/s$ time units. We let $w_j^A(t) = y_j^A(t)\frac{w_j}{p_j}$ be the fractional remaining weight of job j at time t in A's schedule. We let $W^A(t) = \sum_{j \in U^A(t)} w_j$ denote the overall weight of jobs that A has not completed by time t. We use $W^{\textsc{Opt}}(t)$ to denote the minimum over all algorithms A of $W^A(t)$.

It is well known that the weighted flow time of a schedule S is equal to $\int W^S(t)dt$. Hence to prove that an algorithm A is a c-approximation algorithm for weighted flow time it is sufficient to show that A is locally c-competitive, this meaning that, at any time t, $W^A(t) \leq cW^{\textsc{Opt}}(t)$. Observe that, in principle, A might be a c-approximation algorithm without being locally c-competitive. We show further in this section that this in fact is not the case for the problem we are considering.

Recall that, given m processors, the algorithm Highest Density First (HDF) always runs the up to m densest available jobs. We now prove, via a simple exchange argument, that giving HDF a faster processor doesn't decrease the time that HDF runs any unfinished job by any time.

Lemma 1 *For any number of processors, for any job instance, for any time t, and for any $j \in U^{\mathrm{HDF}(1+\epsilon)}(t)$, it is the case that $p_j^{\mathrm{HDF}(1+\epsilon)}(t) \geq (1+\epsilon)p_j^{\mathrm{HDF}(1)}(t)$, or equivalalently, before time t it is the case that* HDF*$(1+\epsilon)$ has run job j for more time than* HDF*(1).*

Proof. Assume to reach a contradiction that there is a time t and a job j where this does not hold. Further, assume that t is the first such time where this fails to hold. Note that obviously $t > 0$. HDF(1) must be running j at time t by the definition of t. Also by our assumptions, HDF$(1+\epsilon)$ can not have finished j before time t, and does not run j at time t. Hence, HDF$(1+\epsilon)$ must be running some job h at time t, that HDF(1) is not running at time t. HDF$(1+\epsilon)$ could not be idling any processor at time t since j was available to be run, but wasn't selected. Since HDF$(1+\epsilon)$ is running job h instead of job j, job h must be denser than job j. Then the only reason that HDF(1) wouldn't be running h is if it finished h before time t. But then HDF(1) would have had to spend more time processing h before time t than HDF$(1+\epsilon)$ did. However, this contradicts our assumption of the minimality of t. □

We now construct a lower bound $W^{\mathrm{FHDF}}(t)$ to $W^{\textsc{Opt}}(t)$ in the uniprocessor setting. Define $W^{\mathrm{FHDF}}(t) = \sum_{j \in U^{\mathrm{HDF}(1)}(t)} w_j^{\mathrm{HDF}(1)}(t)$ to be the remaining fractional weight at time t for HDF(1). So if HDF(1) has run 1/3 of a job j by time t then j only contributes $\frac{2}{3}w_j$ to $W^{\mathrm{FHDF}}(t)$.

Lemma 2 *For any instance of the problem $1|r_i, pmtn|\sum w_i F_i$, and any time t, it must be the case that $W^{\mathrm{FHDF}}(t) \leq W^{\textsc{Opt}}(t)$.*

Proof. We use a simple exchange argument. To reach a contradiction, let Opt be an optimal schedule that agrees with the schedule HDF(1) the longest. That is, assume that Opt and HDF(1) schedule the same job at every time strictly before time t, and that no other optimal schedule agrees with HDF(1) for a longer initial period. Let i be the job that HDF(1) is running at time t and let j be the job that Opt is running at time t. The portion of job i that HDF(1) is running at time t must be run later by Opt at some time, say s. Now create a new schedule Opt$'$ from Opt by swapping the jobs run at time s and time t, that is Opt$'$ runs job i at time t, and job j at time s. Note that job i has been released by time t since it is run at that time in the schedule HDF(1). By the definition of HDF, it must be the case that $\mu_i \geq \mu_j$. Hence, for all times u, $W^{\textsc{Opt}'}(u) \leq W^{\textsc{Opt}}(u)$, and Opt$'$ is also an optimal solution. This is then a contradiction to the assumption that Opt was the optimal solution that agreed with HDF(1) the longest. □

We now establish that HDF$(1+\epsilon)$ is locally $(1+\frac{1}{\epsilon})$-competitive against FHDFin the uniprocessor setting.

Lemma 3 *For a uniprocessor system, for any $\epsilon > 0$, for any job instance, for any time t, and for any job j, it is the case that $W^{\mathrm{HDF}(1+\epsilon)}(t) \leq (1+\frac{1}{\epsilon})W^{\mathrm{FHDF}}(t)$*

Proof. It follows from lemma 1 that the only way that $W^{\mathrm{HDF}(1+\epsilon)}(t)$ can be larger than $W^{\mathrm{FHDF}}(t)$ is due to the contributions of jobs that were run but not completed by both HDF$(1+\epsilon)$ and HDF(1). Let j be such a job. Since, HDF(1) can not have processed j for more than $p_j^{\mathrm{HDF}(1+\epsilon)}(t)/(1+\epsilon)$ time units before time t,

$$w_j^{\mathrm{HDF}(1)}(t) \geq \frac{1}{p_j}\left(p_j - \frac{p_j^{\mathrm{HDF}(1+\epsilon)}(t)}{1+\epsilon}\right) w_j$$

The ratio $w_j/w_j^{\mathrm{HDF}(1)}(t)$ is then at most

$$\frac{p_j}{\left(p_j - \frac{p_j^{\mathrm{HDF}(1+\epsilon)}(t)}{1+\epsilon}\right)}$$

When $p_j^{\mathrm{HDF}(1+\epsilon)}(t) = p_j$, which is where this ratio is maximized, this ratio is $1+1/\epsilon$. Thus the result follows. □

We are then able to conclude the following.

Theorem 4 *Highest Density First (*HDF*) is a $(1+\epsilon)$-speed $(1+\frac{1}{\epsilon})$-approximation algorithm for the problem $1|r_i, pmtn|\sum w_i F_i$.*

Proof. If follows from lemma 2 that $W^{\mathrm{FHDF}}(t)$ is a lower bound to $W^{\mathrm{OPT}}(t)$. It is established in lemma 3 that HDF$(1+\epsilon)$ is $(1+\frac{1}{\epsilon})$-competitive against FHDF. □

Theorem 5 *Highest Density First (*HDF*) is a $(2+\epsilon)$-speed $O(1)$-approximation algorithm for $P|r_i, pmtn|\sum w_i F_i$.*

Proof. The proof follows the same lines as the proof of Theorem 4, and uses some techniques from the analysis of SRPT in [12]. We first define $W^{\mathrm{FHDF}(2)}(t) = \sum_{j \in U^{\mathrm{HDF}(2)}(t)} w_j^{\mathrm{HDF}(2)}(t)$, and show that each $W^{\mathrm{FHDF}(2)}(t)$ is a lower bound to $W^{\mathrm{OPT}(1)}(t)$. The following property of busy algorithms A (algorithms that don't unnecessary idle a processor) is shown in [12]:

$$\forall t \quad \sum_j p_j^{A(2)}(t) \geq \sum_j p_j^{\mathrm{OPT}(1)}(t).$$

That is, at all times it is the case that A, with a 2 speed processor, has done at least as much work as any unit speed schedule. Let I_t be the set of jobs finished by OPT by time t. Hence, if the input was only I_t, HDF(2) would have finished all of I_t by time t since HDFis a busy algorithm. One can show, via a simple exchange argument, that HDFhas the property that adding more jobs to the instance never decreases the weight that it has processed by time t, that is $\sum_j w_j \frac{p_j^{\mathrm{HDF}}(t)}{p_j}$ is monotone non-decreasing as more jobs are added to the job set. Therefore,

$$\forall t \quad \sum_{j \in U^{\mathrm{HDF}(2)}(t)} w^{\mathrm{HDF}(2)}(t) \leq W^{\mathrm{OPT}(1)}(t)$$

We then conclude by noting that HDF$(2+2\epsilon)$ is locally $(1+\frac{1}{\epsilon})$-competitive against FHDF(2) by lemma 2 □

3 Deadline Scheduling Problem

In this secton we first show that a c-competitive algorithm for minimizing weighted flow time must be also locally *c-competitive.* We know that local c-competitiveness is sufficient to establish c-competitiveness. We now show that for online algorithms, with speed 1 processors, it is the case that local c-competitiveness is also necessary.

Theorem 6 *Every c-competitive deterministic online algorithm A for* $1|r_i, pmtn|\sum w_i F_i$ *must be locally c-competitive.*

Proof. Let A be a c-competitive algorithm. Assume to reach a contradiction that there is an instance I, and a rational time t such that $W^A(t) > c \cdot W^{\text{OPT}}(t)$. Intuitively, we construct a new instance I' on which A is not c-competitive by bringing in a sequence of arbitrarily short and dense jobs starting from time t. First remove any jobs from I that are released after time t. Let ℓ be least common multiple of $\{y_j^A(t) \mid j \in U^A(t)\} \cup \{y_j^{\text{OPT}}(t) \mid j \in U^{\text{OPT}}(t)\}$, that is, the unexecuted portions of the jobs unfinished by A at time t union the unexecuted portions of the jobs in OPT at time t. Let w be sufficiently small when compared with $W^{\text{OPT}}(t)$. Let ϵ_w be an arbitrarily small length such that ℓ/ϵ_w is an integer, and for all $j \in U^A(t)$ it is the case that $\frac{w_j}{y_j(t)} \leq \frac{w}{\epsilon_w}$, that is, $\frac{w}{\epsilon_w}$ is greater than the density of any jobs portions in $U^A(t)$. Between time t, and some large time T, we bring in one job of length ϵ_w, and weight w, every ϵ_w time units.

It is easy to see the optimal strategy for A is to always run these new jobs after time t. By making T sufficiently large, the relative contribution to the weighted flow time of jobs waiting before time t, becomes negligible. The contribution to the weighted flow time for the jobs released after time t is $w \cdot \epsilon_w \frac{(T-t)}{\epsilon_w} = w \cdot (T-t)$ since each of these $\frac{T-t}{\epsilon_w}$ jobs have flow time ϵ_w. Thus the competitive ratio can be approximated arbitrarily well by $\frac{W^A(t) \cdot T + w \cdot T}{W^{\text{OPT}}(t) \cdot T + w \cdot T}$. Since w is small when compared with $W^{\text{OPT}}(t)$, The competitive ratio is well approximated by $\frac{W^A(t)}{W^{\text{OPT}}(t)}$, which is bigger than c by assumption. This contradicts the assumption that A was a c-competitive algorithm. $\square$

The negative result shown above motivates the DSP problem: the goal is to design an algorithm that, at each time instant t, completes a set of jobs whose overall weight is a constant factor away from the set of jobs completed by an optimal algorithm *that knows* t. The simplification with respect to minimizing weighted flow time is that in DSP all jobs are released at time 0 and that we consider a simplified objective function.

We now give an example to show that DSP is more subtle than one might at first think. Consider the following set of jobs, released at time 0:

Number of Jobs	weight	length	Density = weight/length
1	k	k^2	$1/k$
k^3	1	1	1

Figure 1: A non-trivial input for DSP. $k \gg 1$.

Two natural schedules are: (1) First run the low density job followed by the k^3 high density jobs, and (2) First run the k^3 high density jobs followed by the low density job. It is easy to see that the first algorithm is not constant competitive at time $D = k^3$, and that the second algorithm is not constant competitive at time $D = k^3 + k^2 - 1$. In fact, what the online algorithm should do in this instance is to first run $k^3 - k$ of the high density jobs, then run the low density job, and then finish with the remaining k high density jobs. A little reflection reveals that this schedule is $O(1)$-competitive for this instance of DSP. This instance demonstrates that the scheduler has to balance between delaying low density jobs, and delaying low weight jobs. Further in this section we propose an $O(1)$-competitive algorithm for DSP. This algorithm has to recursively deal with difficulties similar to those that arise in the instance in table 3. It is possible to construct instances where one can show that essentially the full complexity of our algorithm description is necessary to achieve $O(1)$-competitiveness.

An alternative way to look at DSP is by reversing the time axis. This makes DSP equivalent to the problem minimizing the overall weight of jobs that have been started by some unknown deadline. This problem is more formally stated below.

NDSP Problem Statement: The input consists of n jobs released at time 0, with known processing times and weights, plus an a priori unknown deadline t. The goal is to find a schedule that minimizes the total weight of jobs that have been *started* before time t including the job running at time t. Intuitively, time t in this schedule corresponds to time $\sum p_i - t$ in the original schedule.

In this section, it is convenient to use the notation $A(t)$ to refer to both the schedule produced by algorithm A before time t, and the weight of the jobs started by algorithm A before time t. Similarly, $\text{OPT}(t)$ refers to the minimum weight collection of jobs with length at least t, and the schedule that obtains this minimum. Note that in general there is not a single schedule that optimizes all t's simultaneously, that is, the schedules $\text{OPT}(t)$ and $\text{OPT}(t')$ may be very different. For a fixed instance of NDSP, the competitive ratio of an algorithm A is the maximum over all $t \in [0, \sum p_j]$ of $A(t)/\text{OPT}(t)$.

Description of Algorithm OFF: Algorithm OFF calls a recursive algorithm $\text{OFF}'(t)$ that computes schedules $\text{OFF}_1(t), \ldots, \text{OFF}_u(t)$. The schedule produced by OFF is then the schedule among $\text{OFF}_1(t), \ldots, \text{OFF}_u(t)$ with minimum weight. When $\text{OFF}'(t)$ is initially called, $\mathcal{J}$ is the collection of all jobs, and in general, $\mathcal{J}$ contains the jobs that $\text{OFF}'(t)$ may conceivably schedule in the future. When $\text{OFF}'(t)$ is initially called, $s = 0$, and in general s is the time that the last job in the current schedule ends. When $\text{OFF}'(t)$ is initially called, $h = 1$, and in general h is the index for the current schedule $\text{OFF}_h(t)$ that $\text{OFF}'(t)$ is constructing.

1. Let j be the least dense job in $\mathcal{J}$
2. Remove j from $\mathcal{J}$
3. If $s + p_j < t$ then
   (a) Append j to the end of the schedule $\text{OFF}_h(t)$
   (b) Let $s = s + p_j$
   (c) Go to step 1
4. Else if $s + p_j \geq t$ then
   (a) Let $\text{OFF}_{h+1}(t) = \text{OFF}_h(t)$. **Comment:** Initialize $\text{OFF}_{h+1}(t)$.
   (b) Append j to the end of the schedule $\text{OFF}_h(t)$. **Comment:** Schedule $\text{OFF}_h(t)$ is now completed.
   (c) Let $h = h + 1$
   (d) Remove from $\mathcal{J}$ any jobs with weight greater than $w_j/2$
   (e) If the total length of the jobs in $\mathcal{J}$ is at least $t - s$ then recursively call $\text{OFF}'(t)$

We first describe a polynomial time offline algorithm OFF for NDSP, and then show that OFF is 3-approximate with respect to OPT(t). We then propose a polynomial time online algorithm called R for NDSP, and show that R is 8-competitive with respect to OFF. Hence, we can conclude that R is a 24-competitive algorithm. Both R and OFF at various points in their execution must select the lowest density job from some collection; Ties may be broken arbitrarily, but in order to simplify our analysis, we assume that OFF and R break ties similarly (say by the original numbering of the jobs).

Intuitively, OFF tries to schedule the least dense jobs until time t. The algorithm OFF must then be concerned with the possibility that the job that it has scheduled at time t, say j, is of too high a weight. So OFF recursively tries to construct a lower aggregate weight schedule to replace j, from those unscheduled jobs with weight at most $w_j/2$. Observe that algorithm OFF runs in polynomial time.

If a job j is scheduled in step 3a then we say that j was *scheduled non-recursively*. It a job j is scheduled in step 4b then we say that j was *scheduled recursively*, and that afterwards OFF$'(t)$ *recursed on* j, or sometimes, that OFF$'(t)$ *recursed on the jobs in* $\mathcal{J}$ *after* j.

Theorem 7 OFF$(t) \leq 3$OPT(t).

Proof. The jobs in the schedule OFF are ordered in increasing order of density. Assume that the jobs in OPT(t) are ordered by increasing density as well. Let us introduce some notation. Let $f_1, \ldots, f_u$ be the sequence of jobs on which OFF$'(t)$ recursed. Let $b(f_h)$ be the time at which f_h is started in OFF$_h(t)$. This notation is illustrated in figure 2.

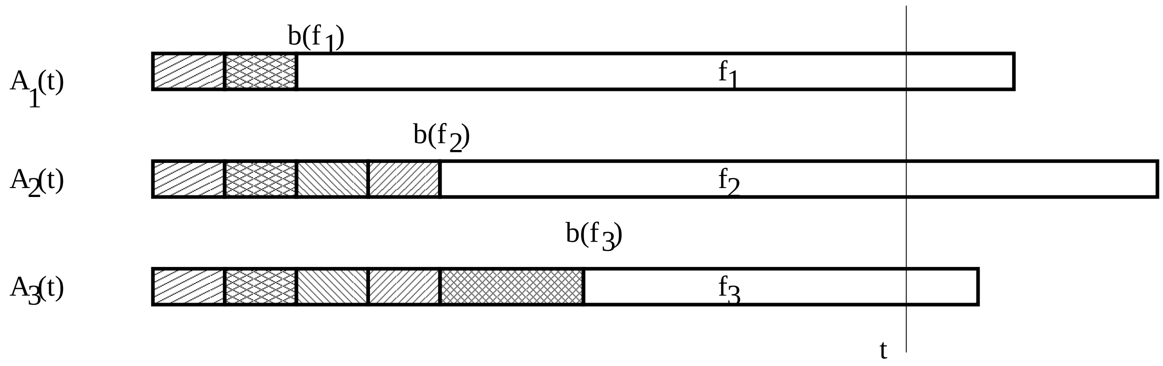

Figure 2: A depiction of our notation with $k = 3$.

We break the proof into two cases. In the first case we assume there is some schedule OFF$_h(t)$, such that OFF$_h(t)$ is running a non-recursively scheduled job j at some time s, OPT(t) is running some job i this time s, and $\mu_j > \mu_i$. (As a technicality, we can assume that if such a time s exists, then we can pick a time s where neither OFF$_h(t)$ nor OPT(t) started nor finished a job at time s. So let us make this assumption about s.) It is easy to see that OFF$_1(t)$ is always running a job with density at most the density of the job that OPT(t) is running at that time. Hence, it must be the case that $h > 1$. Since $i \neq j$, there must be some job g run by OPT(t) before i, that OFF$_h(t)$ does not run before j. Note that it may be the case that $g = i$. Then the only possible reason that OFF$'(t)$ didn't select g instead of j, is that g was eliminated in step 4d, before OFF$'(t)$ recursed on some f_d, $d \leq h-1$. Hence, $w_g \geq w_{f_{h-1}}/2$, and $w_{f_{h-1}} \leq 2$OPT(t) since w_g is scheduled in OPT(t). By the minimality of h, it must be the case that, at all times before time $b(f_{h-1})$, the density of the job that OFF$_{h-1}(t)$ is running is at most the density of the job that OPT(t) is running. Hence, the aggregate weight of the jobs in

$\text{OFF}_{h-1}(t)$, exclusive of f_{h-1}, is at most $\text{OPT}(t)$. We can then conclude that $\text{OFF}_{h-1}(t) \le 3\text{OPT}(t)$, and hence $\text{OFF}(t) \le 3\text{OPT}(t)$ by the definition of OFF.

In the second case assume that, at all times in all the schedules $\text{OFF}_h(t)$, it is the case that the job that $\text{OFF}_h(t)$ is running at this time is at most the density of the jobs that $\text{OPT}(t)$ is running at this time. Hence, the weight of the jobs in the last schedule $\text{OFF}_u(t)$, exclusive of f_u, is at most $\text{OPT}(t)$. Consider the time that $\text{OFF}'(t)$ recursively schedules f_u. The fact that $\text{OFF}_u(t)$ is the last schedule produced by $\text{OFF}'(t)$ means that the total lengths of jobs in $\text{OFF}_u(t)$, union the jobs in $\mathcal{J}$ after the subsequent step 4d is strictly less than t. The jobs in $\mathcal{J}$ at this time, are those jobs with weight at most $w_{f_u}/2$ that were not previously scheduled in $\text{OFF}_u(t)$. Hence, the total length of the jobs in the original input with weight at most $w_{f_u}/2$ is strictly less than t. Therefore at least one job in $\text{OPT}(t)$ has weight at least $w_{f_u}/2$. Hence, $w_{f_u} \le 2\text{OPT}(t)$. Therefore, once again we can conclude that $\text{OFF}_u(t) \le 3\text{OPT}(t)$, and that $\text{OFF}(t) \le 3\text{OPT}(t)$. □

We now turn to describing the algorithm R. Intuitively, the algorithm R first considers the lowest density job, say j. However, R can not immediately schedule j because it may have high weight, which might mean that R wouldn't be competitive if the deadline occurred soon after R began executing j. To protect against this possibility, R tries to recursively schedule jobs with weight at most $w_j/2$, before it schedules j, until the aggregate weight of those jobs scheduled before j exceeds w_j. We now give a more formal description of the algorithm R.

Description of Algorithm R: The algorithm R repeatedly calls a recursive algorithm R' until all the jobs have been scheduled. The algorithm R' takes as input a collection $\mathcal{J}$ of jobs. Initially, $\mathcal{J}$ is the collection of all jobs, and in general, $\mathcal{J}$ will be those jobs not yet scheduled by previous calls to R'. The algorithm R' is described below:

1. Select a $j \in \mathcal{J}$ of minimum density. Remove j from $\mathcal{J}$.
2. (a) Initialize a collection $\mathcal{I}_j$ to the empty set. In general, $\mathcal{I}_j$ will be the jobs scheduled during step 2.
   (b) Let $\mathcal{J}_j$ be those jobs in $\mathcal{J}$ with weight at most $w_j/2$.
   (c) If $\mathcal{J}_j$ is empty then go to step 3.
   (d) Recursively apply R' to $\mathcal{J}_j$. Add the jobs scheduled in this recursive call to $\mathcal{I}_j$. Remove the jobs scheduled in this recursive call from $\mathcal{J}$.
   (e) If the aggregate weight of the jobs in $\mathcal{I}_j$ exceeds w_j then go to step 3, else go to step 2c.
3. Schedule j after all jobs of $\mathcal{I}_j$.

Alike OFF, R also runs in polynomial time.

We say a job j was *picked* if is was selected in step 1. If, after j is picked, a recursive call to R' is made with input $\mathcal{J}_j$, we say that R *recurses* on j.

We now explain how to think of the schedule R as a forest, in the graph theoretic sense. The forest has a vertex for every job in R. For this reason we will henceforth use the terms job and vertex interchangeably. The jobs in the subtree $T(j)$ rooted at job j are j, and those jobs scheduled in the loop in step 2 after j was selected in step 1. The ordering of the children is by non-decreasing density, and in the order in which R' selects and schedules these jobs. The roots of the forest are those vertices selected in step 1 every time algorithm R' is invoked by algorithm R, i.e. at the highest level of recursion. Similarly, the ordering of the roots of the forest is also by non-decreasing density, and in

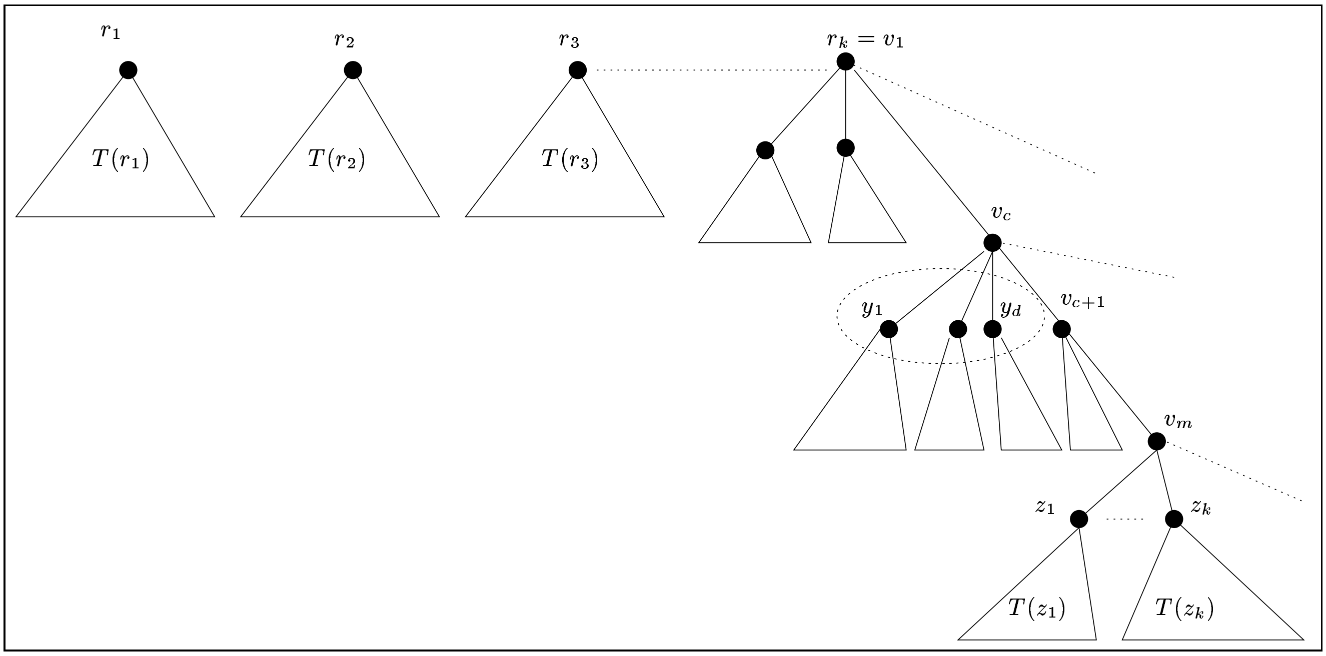

Figure 3: Forest structure of R's schedule.

the order in which R selects and schedules these jobs. We adopt the convention that the orderings of trees in the forest, and children of a vertex, are from left to right.

Before proceeding we need to introduce some notation. Given a descendent y of a vertex x in some tree, we denote by $P(x, y)$ the path from x to y in T, x and y inclusive. If y is not a descendent of x, then $P(x, y)$ is the empty path. We denote by $r_1, \ldots, r_k$ the roots of the trees in R that contain at least one job that R started before time t. Let $v_1, \ldots, v_m$ be the vertices in the path $P(r_k, v_m)$, where $v_1 = r_k$, and where v_m in the job that R is running at time t. Finally, denote by $w(S)$ the total weight of all the jobs in set S. See Figure 3 for an illustration of the forest structure of R and some of our definitions.

The following facts can easily be seen to follow from the definition of algorithm R. The post-order of the jobs within each tree is also the order that these jobs are scheduled by R. All jobs in a tree $T(r_h)$, $h < k$, are finished by R before time t. All jobs in a tree $T(r_h)$, $h > k$, are not started by R before time t. The jobs of $T(r_k)$ that are started by R before time t are: (i) The jobs in $T(v_m)$, and (ii) the jobs in $T(\ell)$, for each left sibling ℓ of a job in $P(v_2, v_m)$. All the jobs in the path $P(r_k, v_{m-1})$ are not started by R before time t. All the jobs in a $T(\ell)$, ℓ a right sibling of a job on the path $P(v_2, v_m)$, are not started by R before time t.

The following lemma shows that the aggregate weight of the vertices in a subtree is not too large.

Lemma 8 *For any job x in R, $w(T(x)) \leq 4w_x$.*

Proof. The proof is by induction on the depth of x. The base case (i.e. x is a leaf) is straightforward. Let y be x's rightmost child. The aggregate weight of the subtrees rooted at the left siblings of y is at most w_x, or R would not have recursed from x to y. Since R removes jobs of weight greater than $w_x/2$ in step 2b, we know that $w_y \leq w_x/2$. By induction $w(T(y)) \leq 4w_y$, and therefore $w(T(y)) \leq 2w_x$. Hence, by adding the weight of the subtrees rooted at x's children to the weight of x, we get that $w(T(x)) \leq w_x + w_x + 2w_x = 4w_x$. □

Theorem 9 *$R(t) \leq 8\textsc{Off}(t)$.*

Proof. We will charge all the weight of the jobs started by R before time t to the jobs scheduled by OFF in such way that no job in OFF is charged for more than 8 times its weight.

First notice that OFF completes all the jobs $r_1, \ldots, r_{k-1}$ by time t. This is because all of the jobs with density less than μ_{v_1} are in trees $T(r_1), \ldots, T(r_{k-1})$, and the total lengths of the jobs in the trees $T(r_1), \ldots, T(r_{k-1})$ is strictly less than t by the assumption that t occurs in tree $T(r_k)$. We charge $w(T(r_i))$ to each r_i, $1 \leq i \leq k-1$. By lemma 8 each r_i is charged at most four times its weight.

We are left to charge the weight of the jobs in $T(r_k)$ started by R before time t. We consider three cases:

(i) OFF schedules job $r_k = v_1$;
(ii) OFF schedules a job on path $P(v_1, v_m)$;
(iii) OFF schedules no job on path $P(v_1, v_m)$.

In case (i), we charge $w(T(r_k))$ to r_k. This way r_k is charged at most four times its weight. So now assume that r_k is picked but not scheduled by OFF. To finish our analysis of case (ii) and case (iii) we need the following claim, which illuminates the similarities between the algorithms R and OFF.

Claim 10 *Assume* OFF *schedules no job on $P(v_1, v_b)$, $1 \leq b \leq m$. Then* OFF *non-recursively schedules all of children of v_b that are scheduled by R before time t, and* OFF *recurses on v_b.*

Proof. First notice that OFF picks $r_k = v_1$ in step in step 1 before OFF ever recurses. This is because all of the jobs with density less than μ_{v_1} are in trees $T(r_1), \ldots, T(r_k)$, and and total lengths of the jobs in the trees $T(r_1), \ldots, T(r_k)$ is strictly less than t. Since by assumption OFF does not schedule r_k, it must be the case that OFF recurses on r_k.

Now let v_c be a vertex in $P(v_1, v_b)$ on which OFF recurses, but does not schedule. The jobs that OFF then on recurses after v_c are the jobs with density greater than μ_{v_c} and weight at most $w_{v_c}/2$. The jobs that R recurses on after v_c are those jobs with density greater than μ_{v_c}, and weight at most $w_{v_c}/2$, that R had not previously scheduled. Let $y_1, \ldots, y_d$ be the children of v_c in R that are scheduled, and finished, by R before time t. Let us consider the actions of OFF$'$ after recursing on v_c but before recursing again. We claim that at this time it must be the case that OFF schedules non-recursively all of the jobs $y_1, \ldots, y_d$ before it recurses again. This is because it can be seen from the schedule R that the aggregate length of all of the jobs with density at most μ_{y_d}, and weight at most $w_{v_c}/2$ is strictly less than t. Similarly, in the case that $c < m$, OFF$'$ must eventually pick job v_{c+1} in step 1 since the aggregate lengths of the jobs that OFF might pick before v_{c+1} is less than t. The claim then follows by inductively applying this argument until $c = b$ □

Now consider case (ii). Let v_b be the job on $P(v_1, v_m)$ closest to v_1 scheduled by OFF. We charge to vertex v_b the weight of all the jobs in $T(v_b)$ started by R before t. By lemma 8, v_b is charged no more than four times its own weight. By Claim 10, OFF also scheduled all of the left siblings ℓ of any vertex $v_c \in P(v_1, v_b)$; We charge $w(T(\ell))$ to each such ℓ. By lemma 8, each such ℓ is charged no more than four times its own weight.

Now consider case (iii). By claim 10, OFF schedules all of the left siblings ℓ of the jobs on $P(v_1, v_m)$ in R. We charge $w(T(\ell))$ to each such ℓ. By lemma 8, each such ℓ is charged no more than four times its own weight. By claim 10, OFF schedules all of the children of v_m in R.

We now argue that it must be the case that $w(T(v_m)) - w_{v_m} \geq w_{v_m}$. Assume to reach a contradiction that $w(T(v_m)) - w_m < w_{v_m}$; That is, assume that the recursion of R' on v_m was completed because $w(\mathcal{J}_t)$ was less than w_{v_m} at step 2e. By Claim 10, algorithm OFF recursed on v_m. The jobs that OFF recursed on at this time were the jobs with density greater than μ_{v_m}, and weight at most $w_{v_m}/2$. The jobs that R recursed after v_m were those jobs with density greater than μ_{v_m} and weight at most $w_{v_m}/2$, that R had not scheduled earlier. Hence, the fact that R had not yet reached time t after scheduling the rightmost child of v_m, implies that OFF would not have recursed on job v_m because the aggregate length of the jobs in $\mathcal{J}$ at that time would be insufficient to reach time t. Hence, OFF would have to schedule some job on the path $P(v_1, v_m)$, which is a contradiction.

So we may now assume that the recursion of R on v_m completed because $w(T(v_m)) - w_{v_m} \geq w_{v_m}$. Denote by $z_1, \ldots, z_k$ all the children of v_m. By claim 10 the jobs $z_1, \ldots, z_k$ are all scheduled non-recursively by OFF. We charge $w(T(z_i))$, $1 \leq i \leq k$, to each such z_i for the weight of the jobs in z_i's subtree. By lemma 8, each such z_i is charged no more than four times its own weight for this. We charge $w(T(z_i))$, $1 \leq i \leq k$, to each such z_i for the weight of v_m. By the assumption of this subcase, this is sufficient to fully charge away the weight of v_m. Hence, each such z_i is charged at most eight times its weight. □

The following corollary immediately follows from Theorems 7 and 9 by reversing the schedule given by R.

Corollary 11 *There exists a 24-competitive algorithm for DSP.*

4 Conclusions

In this paper we considered several aspects of the problem of minimizing the weighted flow time of a set of jobs on identical machines. We first tackled the problem by using the tools of resource augmentation analysis, proving that with $(1+\epsilon)$-fast processors the very simple HDF heuristic can provide solutions whose value is arbitrarily close to optimum in the case of one machine, while it is at most a constant factor away from optimum in the case of multiple machines.

Turning to the classical case of unit speed processors, we proved that the property of local competitiveness is necessary and sufficient to be globally competitive. This brought us to further identify and recognize the importance of a subproblem of $1|r_i, pmtn| \sum w_i \cdot F_i$, i.e. minimizing the overall weight of the jobs that are unfinished by some unknown deadline. For this latter problem, which we named DSP, we provided both offline and online polynomial time, constant approximation algorithms. While the study of DSP in our opinion provides deeper insight into $1|r_i, pmtn| \sum w_i \cdot F_i$, we found the problem worth investigation for its own. It is in some sense dual to the one considered in [7], but interestingly a constant competitive algorithm for DSP exists, whereas this does not hold for the problem considered in [7].

We finally interpret the results in this paper as suggesting that a reasonably simple 1-speed $O(1)$-competitive algorithm for the problem of minimizing the average broadcast time of a set of requests is unlikely, since minimizing the weighted flow time is a particular case of broadcast scheduling to minimize the average flow time.

References

[1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the Flow Time without Migration. *In Proc. ACM Symposium on the Theory of Computing (STOC '99)*, 1999.

[2] S. Aacharya, and S. Muthukrishnan, "Scheduling on-demand broadcasts: new metrics and algorithms", 4th ACM/IEEE International Conference on Mobile Computing and Networking, 43 – 54, 1998.

[3] Y. Bartal, and S. Muthukrishnan, "Minimizing Maximum Response Time in Scheduling Broadcasts", 558-559, Proceedings of the Eleventh Annual ACM/SIAM Symposium on Discrete Algorithms, pages 558-559, 2000.

[4] L. Becchetti, and S. Leonardi. Non-Clairvoyant Scheduling to Minimize the Average Flow Time on Single and Parallel Machines. *To appear in Proc. of the 33rd ACM Symposium on the Theory of Computing.*

[5] J. Labetoulle, E. Lawler, J.K. Lenstra, A. Rinnooy Kan, "Preemptive scheduling of uniform machines subject to release dates", in W.R. Pulleyblank (eds.), *Progress in Combinatorial Optimization*, 245–261, Academic Press, 1984.

[6] E. Lawler, J.K. Lenstra, A. Rinnooy Kan, and D. Shmoys, "Sequencing and Scheduling: Algorithms and Complexity", *Logistics of Production and Inventory*, Handbooks in OR & MS 4, Elsevier Science, Chapter 9, 445–522, 1993.

[7] Vincenzo Liberatore, "Scheduling jobs before shut-down", SWAT, 2000.

[8] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 110–119, El Paso, Texas, 1997.

[9] B. Kalyanasundaram, and K. Pruhs, "Speed is as powerful as clairvoyance", *IEEE Symposium on Foundations of Computation*, 214 – 221, 1995.

[10] B. Kalyanasundaram, and K. Pruhs. Minimizing Flow Time Nonclairvoyantly. *In Proc. of IEEE Symposium on Foundations of Computer Science (FOCS '97)*, 1997, pages 345-352.

[11] B. Kalyanasundaram, K. Pruhs, and M. Velauthapillai, "Scheduling broadcasts in wireless networks", *European Symposium on Algorithms* (ESA), 2000.

[12] C. Phillips, C. Stein, E. Torng, and J. Wein "Optimal time-critical scheduling via resource augmentation", *ACM Symposium on Theory of Computing*, 140 – 149, 1997.

[13] J. Shanmugasundaram, A. Nithrakashyap, R. Sivasankaran, and K. Ramamritham, "Efficient concurrency control for broadcast environments", Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD 99), pages 85–96, 1999.