Server Scheduling in the Weighted ℓ_p Norm

Nikhil Bansal[1] and Kirk Pruhs[2]*

[1] Department of Computer Science, Carnegie Mellon University, USA
nikhil@cs.cmu.edu
WWW home page: http://www.cs.cmu.edu/~nikhil
[2] Department of Computer Science, University of Pittsburgh, USA
kirk@cs.pitt.edu
WWW home page: http://www.cs.pitt.edu/~kirk

Abstract. We explain how the apparent goals of the Unix CPU scheduling policy can be formalized using the weighted ℓ_p norm of flows. We then show that the online algorithm, Highest Density First (HDF), and the nonclairvoyant algorithm, Weighted Shortest Elapsed Time First (WSETF), are almost fully scalable. That is, they are $(1+\epsilon)$-speed $O(1)$-competitive. Even for unit weights, it was known that there is no $O(1)$-competitive algorithm. We also give a generic way to transform an algorithm A in an algorithm B in such a way that if A is $O(1)$-speed $O(1)$-competitive with respect to some ℓ_p norm of flow then B is $O(1)$-competitive with respect to the ℓ_p norm of completion times. Further, if A is online (nonclairvoyant) then B is online (nonclairvoyant). Combining these results gives an $O(1)$-competitive nonclairvoyant algorithm for ℓ_p norms of completion times.

1 Introduction

1.1 Motivation

Tanenbaum [15, page 704] describes the generic Unix CPU scheduling policy as follows. Each process initially has a *nice* value in the range -20 to 20. Lower *nice* values correspond to processes that are more important. Users can set the *nice* value of a process to be in the range from 0 to 20 with a `nice` system call. Only the system administrator can give a process a negative *nice* value. Once a second the *priority* of a process is recalculated using the formula:

$$priority = CPUusage + nice + base$$

Here the *CPUusage* parameter is an exponential weighted moving average of past CPU usage, the *nice* parameter is the *nice* value for the process, and the *base* parameter is used to give higher priority to jobs that have just returned from some sort of interruption (say for I/O). Confusingly enough, the high priority

* Supported in part by NSF grant CCR-0098752, NSF grant ANI-0123705, and NSF grant ANI-0325353.

jobs are those whose computed *priority* value is smallest. The jobs with highest priority are then scheduled using a Round Robin(RR) policy, typically with the quantum on order of 100 milliseconds. Round Robin shares the processor equally among all processes.

Round Robin represents an apparent effort to balance between optimizing the worst case Quality of Service (QoS) and optimizing the average case QoS. If the goal was to optimize worst case QoS then the best algorithm would be First Come First Served (FCFS). If the goal was to optimize average QoS then Shortest Elapsed Time First (*SETF*) is generally considered to be the best non-clairvoyant algorithm. Processes with lower *nice* values get more of the CPU, but the *CPUusage* parameter works to try to prevent starvation. That is, the *CPUusage* parameter will be high for processes that have been run a lot recently, and thus these processes will have a higher computed *priority*, and thus these processes will be given less CPU time in the near future. So it seems that the Unix system designers' goals for the process scheduling policy were:

Goal A: Amongst jobs of the same priority, there should be some balance between optimizing for average QoS and optimizing for worst case QoS.

Goal B: Higher priority jobs should get a greater share of the CPU resources, but lower priority jobs should not be starved.

In this paper we try to formalize these goals and then analyze algorithms with respect to this formalization.

In the literature, the most common QoS measure for a single process/job J_i is clearly flow/response/waiting time $f_i = c_i - r_i$, where c_i is the time that the job completes and r_i is the time that the job enters the system. The most common way to compromise between optimizing for the average and optimizing for the worst case is to optimize the ℓ_p norm, generally for something like $p = 2$ or $p = 3$. For example, the standard way to fit a line to collection of points is to pick the line with minimum least squares, equivalently ℓ_2, distance to the points, and Knuth's TEXtypesetting system uses the ℓ_3 metric to determine line breaks [12, page 97]. The ℓ_p, $1 < p < \infty$, metric still considers the average in the sense that it takes into account all values, but because x^p is strictly a convex function of x, the ℓ_p norm more severely penalizes outliers than the standard ℓ_1 norm. Analyses of algorithms for optimizing $(\sum F_i^p)^{1/p}$, the ℓ_p norms of flow, can be found in [3].

The most common way that priorities of jobs is formalized is to assume that each job J_i has a positive weight w_i and then to have the objective function be maximizing the weighted QoS. By far the most commonly studied QoS measure for a collection of equal priority jobs is average flow time, and logically enough, the most commonly studied QoS measure for jobs with variable priorities is weighted flow time $\sum w_i \cdot F_i$, e.g. [2, 5–7]. It is easy to see that even an optimal algorithm for optimizing weighted flow time does not in general accomplish **Goal B** as it can starve low weight jobs if there are always higher weight jobs to be run.

If one wishes wishes to achieve both **Goal A** and **Goal B**, then the appropriate objective function to optimize would be something like the weighted ℓ_p

norms of flow, that is, $(\sum w_i F_i^p)^{1/p}$, where $p > 1$ is some small constant. Note that in any competitive schedule for the weighted ℓ_p norm of flow, a low weight job J_i would eventually be scheduled even in the face of a constant stream of high weight jobs.

In [3] it was shown that there is no $O(1)$-competitive online scheduling algorithm for any unweighted ℓ_p norm of flow. This motivated the authors of [3], and us, to fall back to resource augmentation analysis [9]. In the context of a scheduling minimization problem with an objective function F, an algorithm A is s-speed c-competitive if

$$\max_{\mathcal{I}} \frac{F(A_s(\mathcal{I}))}{F(Opt_1(\mathcal{I}))} \leq c$$

where $A_s(\mathcal{I})$ denotes the the schedule that algorithm A with a speed s produces on input $\mathcal{I}$, and similarly $Opt_1(\mathcal{I})$ denotes the adversarial schedule for $\mathcal{I}$ with a unit speed processor. A $(1+\epsilon)$-speed $O(1)$-competitive algorithm is said to be *almost fully scalable* [13]. The intuition is that such an algorithm should perform well up to load close to the capacity of the system since increasing speed corresponds to lowering the load. This intuition is borne out in the lower bound instances, such as those in [3], that show no algorithm can be $O(1)$-competitive. In the lower bound instances, the system is fully loaded, so that there are no spare resources to recover from even small mistakes in scheduling decisions. For a more in depth discussion of this motivation see [9, 3, 13]. In [3] it is shown that several standard algorithms — *SETF*, Shortest Remaining Processing Time(*SRPT*), and Shortest Job First(*SJF*) — are almost fully scalable for any ℓ_p norm of flow. Surprisingly, *RR* is not almost fully scalable for any ℓ_p norm of flow. Note that this result would argue against the use of RR by Unix.

1.2 Our Results

We first show in section 3 that the results in [3] can be extended to the case where the objective function is the weighted ℓ_p norm of flow. In particular, we show that the algorithm Highest Density First(*HDF*) is almost fully scalable. *HDF* always runs the job that has the largest weight to work ratio. *HDF* is the natural generalization of *SJF*. Note however that *HDF* is clairvoyant, that is, it needs to know the work of a job at its release time. While this might be reasonable in a web server serving static documents, this is not reasonable in the context of an operating system.

We then show in section 4 that the obvious nonclairvoyant generalization of the nonclairvoyant algorithm *SETF*, Weighted Shortest Elapsed Time First (*WSETF*), is almost fully scalable. For a job J_i, let $x_i(t)$ denote the amount of work done on that job by time t. We define the measure of a job J_i as $||J_i||_t = \frac{x_i(t)}{w_i}$. Amongst the jobs with the smallest measure, *WSETF* splits the processor proportionally to weights of the jobs. So, if $J_1, \ldots, J_k$ are the jobs that have the smallest measure, then the job J_j will receive a $w_j/(\sum_{i=1}^{k} w_i)$ fraction of the processor. Thus this result suggests the adoption of the algorithm WSETF by Unix.

An interesting aspect of our analysis of *HDF* and *WSETF* is that we first transform the problem on the weighted instance to a related problem on the unweighted instance. This makes the problem simpler and also allows us to use previous results on unweighted scheduling.

There is a lot of literature on scheduling to minimize total/average completion time (a nice survey can be found in [11]), and average weighted completion time [8, 1]. While this does not appear to be an interesting objective function from a computer systems point of view, it seems to be of general academic interest. So one natural academic question to ask is whether there are good online algorithms when the objective is the ℓ_p norm of completion time, or the weighted ℓ_p norm of completion time. In section 5 we give a rather generic way to transform an algorithm for a flow time problem, which possibly uses resource augmentation, to obtain an algorithm for the corresponding completion time problem, which *does not* use resource augmentation. A nice property of our transformation is that online algorithms are transformed to online algorithms, and non-clairvoyant algorithms are transformed to non-clairvoyant algorithms. As a corollary of this result, we will obtain $O(1)$ competitive online and non-clairvoyant algorithms for minimizing the ℓ_p norms of weighted completion time.

1.3 Other Related Results

The following results are known about online algorithms when the objective function is average flow time. The competitive ratio of every deterministic nonclairvoyant algorithm is $\Omega(n^{1/3})$, the competitive ratio of every randomized nonclairvoyant algorithm against an oblivious adversary is $\Omega(\log n)$ [14]. The randomized nonclairvoyant algorithm *RMLF*, proposed in [10], is $O(\log n)$-competitive against an oblivious adversary [4]. The online clairvoyant algorithm *SRPT* is optimal. The online clairvoyant algorithm *SJF* is almost fully scalable [5]. The nonclairvoyant algorithm *SETF* is almost fully scalable [9].

For online weighted flow time, the best known competitive ratio is $O(\log W)$ [2]. It is an outstanding open question whether an $O(1)$-competitive algorithm exists.

2 Definitions

We assume a collection of jobs $\mathcal{J} = J_1, \ldots, J_n$. For J_i, the release time is denoted by r_i, the work/size by p_i, and weight by w_i. Without loss of generality we assume that all job sizes and job weights are integers. The completion time c_i^S of a job J_i in a schedule S is the first time after r_i where J_i has been processed for p_i time units. The flow time of J_i in S is $f_i = c_i^S - r_i$. A clairvoyant algorithm learns p_i at time r_i. A nonclairvoyant algorithm only knows a lower bound on p_i equal to the length of time that it has run J_i. For an algorithm A on an input instance $\mathcal{I}$ with an s speed processor, let $F^p(A, \mathcal{I}, s)$ denote the sum of the p^{th} powers of the flow time of all jobs. Similarly, $WF^p(A, \mathcal{I}, s)$ will denote the sum of weighted p^{th} powers of the flow time (i.e. $\sum_i w_i f_i^p$) of all jobs. Finally, for the measure F^p, let $Opt(F^p, \mathcal{I}, s)$ denote the value of the optimum schedule for the

F^p measure on $\mathcal{I}$ with a speed s processor. Similarly, let $Opt(WF^p, \mathcal{I}, s)$ denote the optimum value for the WF^p measure.

3 Analysis of *HDF*

In this section we show that *HDF*, a natural generalization of *SJF* is a $(1+\epsilon)$-speed $O(1/\epsilon^2)$-competitive online algorithm for minimizing the weighted ℓ_p norms of flow time.

The algorithm *HDF* at any time works on the job which has the largest weight to processing time ratio. The ties are broken in favor of the partially executed job. We will show that

Theorem 1. *HDF is* $(1+\epsilon)$*-speed,* $O(1/\epsilon^2)$*-competitive for minimizing the weighted* ℓ_p *norms of flow time.*

The main idea of the proof will be to reduce the weighted problem to an unweighted problem and then invoke the result for ℓ_p norms of unweighted flow time. We first define the relevant notation.

Given an instance $\mathcal{I}$, we define an instance $\mathcal{I}'$ obtained by applying the following transformation to each job in $\mathcal{I}$: Consider a job $J_i \in \mathcal{I}$. The instance $\mathcal{I}'$ is obtained by replacing J_i by w_i identical jobs each of size p_i/w_i and weight 1, and release time r_i. We denote these w_i jobs by $J'_{i1}, \ldots, J'_{iw_i}$. Let $X_i = \{J'_{i1}, \ldots, J'_{iw_i}\}$ denote this collection of jobs obtained from J_i. Note that all jobs in $\mathcal{I}'$ have the same weight.

Lemma 1. *For* $\mathcal{I}$ *and* $\mathcal{I}'$ *as defined above,*

$$Opt(F^p, \mathcal{I}', 1) \leq Opt(WF^p, \mathcal{I}, 1) \tag{1}$$

Proof. Let S be the schedule which minimizes the weighted ℓ_p norm of flow time for $\mathcal{I}$. Given S, we create a schedule for $\mathcal{I}'$ as follows. At any time t, work on a job in X_i if and only if J_i is executed at time t under S. Clearly, all jobs in X_i finish when J_i finishes execution, thus no job in X_i has a flow time higher than that of J_i. By definition, the contribution of J_i to WF^p is $w_i f_i^p$. Also, the contribution to the measure F^p of each of the w_i jobs in X_i will be at most f_i^p, and hence the total contribution of jobs in X_i to F^p is at most $w_i f_i^p$. Since the optimum schedule for $\mathcal{I}'$ can be no worse than the schedule constructed above, the result follows.

From Theorem 3 in [3] we know that *SJF* is $(1+\epsilon)$-speed, $O(1/\epsilon)$ competitive for the (unweighted) ℓ_p norms of flow time , or equivalently *SJF* is $(1+\epsilon)$-speed $O(1/\epsilon^p)$ competitive for the F^p measure. This implies that,

$$F^p(SJF, \mathcal{I}', 1+\epsilon) = O(\frac{1}{\epsilon^p}) Opt(F^p, \mathcal{I}', 1) \tag{2}$$

We now relate the performance of *HDF* on $\mathcal{I}$ with a $(1+\epsilon)$ times faster processor to that of *SJF* on $\mathcal{I}'$.

Lemma 2.

$$WF^p(HDF, \mathcal{I}, 1+\epsilon) \leq (1+\frac{1}{\epsilon})^p F^p(SJF, \mathcal{I}', 1) \tag{3}$$

Proof. We claim that for every job $J_i \in \mathcal{I}$ and every time t, if J_i is alive at time t under *HDF* with a $1+\epsilon$ speed processor, then at least $\frac{\epsilon}{1+\epsilon} w_i$ jobs in $X_i \in \mathcal{I}'$ are alive at time t under *SJF* with a 1 speed processor.

The claim above immediately implies the result for the following reason. Consider the time $t^- = (f_i + r_i)^-$ just before J_i finishes execution under *HDF*. Then J_i contributes exactly $w_i f_i^p$ to $WF^p(HDF, \mathcal{I}, 1+\epsilon)$, while the $\geq \epsilon w_i/(1+\epsilon)$ jobs in X_i that are unfinished by time t contribute at least $\epsilon w_i/(1+\epsilon) f_i^p$ to $F^p(SJF, \mathcal{I}', 1)$. Taking the contribution over each job, the result follows.

We now prove the claim. Suppose for the sake of contradiction that t is the earliest time when J_i is alive under *HDF* and there are fewer than $\epsilon/(1+\epsilon) w_i$ jobs from X_i left under *SJF*. Since J_i is alive under *HDF* and *HDF* has a $1+\epsilon$ faster processor, it has spent less than $p_i/(1+\epsilon)$ time on J_i, whereas *SJF* has spent strictly more than $p_i/(1+\epsilon)$ time on X_i. Thus there was a some time t', such that $r_i \leq t' < t$ during which *HDF* was running $J_j \neq J_i$ while *SJF* was working on some job from X_i. Since $t' \geq r_i$, it follows from the property of *HDF* that J_j has higher density than that of J_i. This implies that jobs in X_j have smaller size than X_i. Since *SJF* works on X_i at time t', it must have already finished all the jobs in X_j by t'. Since J_j is alive at time t', this contradicts our assumption of the minimality of t.

Proof. (of Theorem 1) By Equations 2 and 3 we have that

$$WF^p(HDF, \mathcal{I}, (1+\epsilon)^2) = O(1/\epsilon)^{2p} Opt(F^p, \mathcal{I}', 1)$$

Combining this with Equation 1 gives us the result.

4 Analysis of *WSETF*

4.1 Algorithm Description

For a job J_i with weight w_i , let $p_i(t)$ denote the amount of work done on J_i by time t. We define the norm of a job J_i as $||J_i||_t = \frac{p_i(t)}{w_i}$.

Algorithm *WSETF*: At all times, *WSETF* splits the processor, proportional to weights of the jobs, among the jobs J_i that have the smallest norm $||J_i||_t$. So, if $J_1, \ldots, J_k$ are the jobs that have the smallest norm. Then J_j, for $i = 1, \ldots, k$, will receive $w_j/(\sum_{i=1}^{k} w_i)$ fraction of the processor.

Note that for all jobs J_i that *WSETF* executes, the norm increases at the same rate and thus stays the same.

4.2 Analysis

As in the analysis of *HDF* the main step of our analysis will be to relate the behavior of *WSETF* on an instance $\mathcal{I}$ with weighted jobs to that of *SETF* on another instance $\mathcal{I}'$ which consists of unweighted jobs. We then use the results about (unweighted) ℓ_p norms of flow time under *SETF* to obtain results for *WSETF*.

Given an instance $\mathcal{I}$ consisting of weighted jobs, let $\mathcal{I}'$ denote the instance defined as in Section 3 which consists of unweighted jobs. Suppose we run *WSETF* on $\mathcal{I}$ and *SETF* on $\mathcal{I}'$ with the same speed processor. Then the schedules produced by *WSETF* and *SETF* are related by the following simple observation.

Lemma 3. *At any time t, a job $J_i \in \mathcal{I}$ is alive and has received $p_i(t)$ units of service if and only if each job in $X_i \in \mathcal{I}'$ is alive and has received exactly $p_i(t)/w_i$ amount of service. In particular, this implies that if J_i has flow time f_i then each $J'_{ik} \in X_i$ for $k = 1, \ldots, w_i$ has flow time f_i.*

Proof. We view the execution of *WSETF* on $\mathcal{I}$ as follows: If at any time *WSETF* allocates x units of processing to a job of weight w_i, then we think of it as allocating x/w_i units of processing to each of the w_i jobs in the collection X_i. Thus the norm of job J_i under *WSETF* is exactly equal to the amount of service received by a job in X_i. Since *WSETF* at any time shares the processor among jobs with the smallest norm in the ratio of their weights, this is identical to the behavior of *SETF* on $\mathcal{I}'$ which works equally on the jobs which have received the smallest amount of service.

Theorem 2. *WSETF is a $1+\epsilon$-speed, $O(1/\epsilon^{2+2/p})$-competitive non-clairvoyant algorithm for minimizing the weighted ℓ_p norms of flow time.*

Proof. By Lemma 3 we know that if $J_i \in \mathcal{I}$ has flow time f_i, then the w_i jobs in X_i have flow time f_i. Thus the ℓ_p norm of unweighted flow time for $\mathcal{I}'$ is $(\sum_i w_i f_i^p)^{1/p}$ which is identical to the weighted flow time for $\mathcal{I}$ under *WSETF*, which implies that

$$WF^p(WSETF, \mathcal{I}, 1) = F^p(SETF, \mathcal{I}', 1) \tag{4}$$

By Equation 1 we know that $Opt(F^p, \mathcal{I}', 1) \leq Opt(WF^p, \mathcal{I}, 1)$. By the main result of Section 7 in [3] about the competitiveness of *SETF* for unweighted ℓ_p norms of flow time we know that

$$F^p(SETF, \mathcal{I}', (1+\epsilon)) = O(1/\epsilon^{2p+2}) Opt(F^p, \mathcal{I}', 1) \tag{5}$$

Now, by Equations 4, 5 and 1 we get that

$$WF^p(WSETF, \mathcal{I}, 1+\epsilon) = O(1/\epsilon^{2p+2}) Opt(WF^p, \mathcal{I}, 1)$$

Thus the result follows.

5 Completion Time Scheduling

In this section, we give a rather generic way to transform an algorithm for a flow time problem that possibly uses resource augmentation to obtain an algorithm for the corresponding completion time problem that does not use resource augmentation. Our transformation carries online algorithms to online algorithms and also preserves non-clairvoyance. As a corollary of this result we will obtain $O(1)$-competitive online and non-clairvoyant algorithms for minimizing the weighted ℓ_p norms of completion time.

We first make precise the notion of a completion time measure corresponding to a flow time measure. Given a schedule S for n jobs, this determines the flow times $f_1, \ldots, f_n$ and the completion times $c_1, \ldots, c_n$. Let $\mathcal{G}$ be some function that takes as input n real numbers and outputs another real number. Given a schedule S, we define the functions $\mathcal{F}$ and $\mathcal{C}$ as follows:

$$\mathcal{F}(S) = \mathcal{G}(f_1, f_2, \ldots, f_n)$$

$$\mathcal{C}(S) = \mathcal{G}(c_1, c_2, \ldots, c_n)$$

For example, if $\mathcal{G}(x_1, \ldots, x_n) = (\sum_i w_i x_i^p)^{1/p}$, then $\mathcal{F}$ and $\mathcal{C}$ are simply the weighted ℓ_p norms of flow time and completion time respectively.

Our technique for converting a flow time result to a completion time result will require two properties from the function $\mathcal{G}$.

Scalability: For any positive real number $k, \mathcal{G}(kx_1, \ldots, kx_n) = k\mathcal{G}(x_1, \ldots, x_n)$. In particular, if we scale all the flow times in a schedule by k times then $\mathcal{F}(S)$ increases by k times.

We now motivate the next property that we require from the function $\mathcal{G}$. We first point out a somewhat surprising property of the ℓ_p norms of the completion time measure. While it is easy to see that minimizing the total weighted flow time (i.e. ℓ_p norm with $p = 1$) is equivalent to minimizing the total weighted completion time, this is not the case for $p > 1$. In particular, it could be the case that a schedule which is optimum for the $\sum_i f_i^2$ measure is suboptimal for $\sum_i c_i^2$ measure and vice versa.

Consider the following instance with just two jobs. The first job has size 10 and arrives at $t = 0$, the second job has size 1 and arrives at $t = 8$. A simple calculation shows that in order to minimize the total flow time squared, it is better to first finish the longer job and then the smaller job. This incurs a total flow time squared of $10^2 + 3^2 = 109$, where as the other possibility which is to finish the small job as soon as it arrives an then finish the big job incurs a total flow time squared of $11^2 + 1^2 = 122$. On the other hand, if we consider completion time squared, finishing the larger job first incurs a cost of $10^2 + 11^2$. If instead if finish the smaller job first, this incurs a cost of $9^2 + 11^2$. Thus the optimal schedule for ℓ_p norms of flow time need not be optimal for ℓ_p norms of completion time and vice versa.

We say that a function $\mathcal{G}$ is $\rho - good$ if is satisfies the following condition:

Given a problem instance $\mathcal{I}$ and any two arbitrary schedules S and S' for $\mathcal{I}$. If $\mathcal{F}(S) \leq c\mathcal{F}(S')$, then $\mathcal{C}(S) \leq \rho c\mathcal{C}(S')$.

Lemma 4. *$\mathcal{G}(x_1, \ldots, x_n) = (\sum_i w_i x_i^p)^{1/p}$ is $2 - good$ for all $p \geq 1$.*

Our main result is the following:

Theorem 3. *Let $\mathcal{G}$ be a $\rho - good$ function. If there is an s-speed, c-competitive online algorithm with respect to the measure $\mathcal{F}$ (derived from $\mathcal{G}$), then this algorithm can be transformed into another online algorithm which is 1-speed, ρcs-competitive with respect to the corresponding completion time measure $\mathcal{C}$. Moreover, non-clairvoyant algorithms are transformed into non-clairvoyant algorithms.*

We now describe the transformation:

Let A be a s-speed, c-competitive algorithm for a flow time problem. Let $\mathcal{I}$ be the original instance where job J_i has release date r_i and size p_i. The online algorithm (which we call B) is the defined as follows:

1. When a job arrives at time r_i, pretend that it has not arrived till time sr_i.
2. At any time t, run A on the jobs for which $t \geq sr_i$

Proof. (of Theorem 3) Let I' be the instance obtained from $\mathcal{I}$ by replacing job $J_i \in \mathcal{I}$ by a job J_i' that has release date sr_i and size sp_i. Also, let $\mathcal{I}''$ be the instance from $\mathcal{I}$ by replacing the job $J_i \in \mathcal{I}$ with a job J_i'' that has release date sr_i and size p_i.

Let $Opt(\mathcal{F}, \mathcal{I}, x)$ (resp $Opt(\mathcal{C}, \mathcal{I}, x)$) denote the flow time cost (resp completion time cost) of the optimum schedule on $\mathcal{I}$ run using an x speed processor. We first relate the values of the optimum schedules for $\mathcal{I}$ and $\mathcal{I}'$.

Fact 4 $Opt(\mathcal{C}, \mathcal{I}', 1) = sOpt(\mathcal{C}, \mathcal{I}, 1)$

By our resource augmentation guarantee for the algorithm A, we know that

$$\mathcal{F}(A, \mathcal{I}', s) \leq cOpt(\mathcal{F}, \mathcal{I}', 1)$$

By the $\rho - goodness$ of $\mathcal{G}$ the above guarantee on flow time implies that

$$\mathcal{C}(A, \mathcal{I}', s) \leq c\rho Opt(\mathcal{C}, \mathcal{I}', 1) \tag{6}$$

We now relate $\mathcal{I}'$ to $\mathcal{I}''$.

Fact 5 $\mathcal{C}(A, \mathcal{I}', s) = \mathcal{C}(A, \mathcal{I}'', 1)$

Now, by definition of the algorithm B, executing the algorithm A on $\mathcal{I}''$ with a speed 1 processor is exactly the schedule produced by B on $\mathcal{I}$ using a 1 speed processor. So the completion times are identical. This implies that

$$\mathcal{C}(B, \mathcal{I}, 1) = \mathcal{C}(A, \mathcal{I}'', 1) \tag{7}$$

Now using Facts 4 and 5 and Equations 6 and 7 it follows that

$$\mathcal{C}(B, \mathcal{I}, 1) \leq c\rho sOpt(\mathcal{C}, \mathcal{I}, 1)$$

Thus we are done.

For $\mathcal{G}(x_1, \ldots, x_n) = (\sum_i w_i x_i^p)^{1/p}$, it is easily seen that the scalability property is satisfied, and Lemma 4 implies that it is $2 - good$. Thus by Theorems 1, 2 and 3 we get that

Corollary 1. *There exist $O(1)$-competitive clairvoyant and non-clairvoyant algorithms for minimizing the weighted ℓ_p norms of completion time.*

References

1. F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, M. Sviridenko, "Approximation Schemes for Minimizing Average Weighted Completion Time with Release Dates", Foundations of Computer Science (FOCS), 32-44, 1999.
2. N. Bansal, K. Dhamdhere, "Minimizing Weighted Flow Time", ACM/SIAM Symposium on Discrete Algorithms (SODA), 508–516, 2003.
3. N. Bansal, K. Pruhs, "Server scheduling in the L_p norm: a rising tide lifts all boats", ACM Symposium on Theory of Computing (STOC), 242–250, 2003.
4. L. Becchetti, and S. Leonardi, "Non-Clairvoyant Scheduling to Minimize the Average Flow Time on Single and Parallel Machines" ACM Symposium on Theorey of Computing (STOC), 2001.
5. L. Becchetti, S. Leonardi, A. Marchetti–Spaccamela, K. Pruhs, "Online weighted flow time and deadline scheduling", Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX), 2001.
6. C. Chekuri and S. Khanna, "Approximation schemes for preemptive weighted flow time", ACM Symposium on Theory of Computing (STOC), 2002.
7. C. Chekuri, S. Khanna and A. Zhu, "Algorithms for weighted flow time", ACM Symposium on Theory of Computing (STOC), 2001.
8. L.A. Hall, A. Schulz, D.B. Shmoys and J. Wein, "Scheduling to minimize average completion time: off-line and on-line approximation algorithms", Mathematics of Operations Research 22, 513–549, 1997.
9. B. Kalyanasundaram, and K. Pruhs, "Speed is as powerful as clairvoyance", *Journal of the ACM*, **47**(4), 617 – 643, 2000.
10. B. Kalyanasundaram, and K. Pruhs, "Minimizing flow time nonclairvoyantly", *Journal of the ACM*, July 2003.
11. D. Karger, C. Stein and J. Wein, "Scheduling algorithms", CRC handbook of theoretical computer science, 1999.
12. D. Knuth, *The TeXbook*, Addison Wesley, 1986.
13. K. Pruhs, J. Sgall, E. Torng, "Online Scheduling", to appear in *Handbook on Scheduling: Algorithms, Models and Performance Analysis*, CRC press.
14. R. Motwani, S. Phillips, and E. Torng, "Non-clairvoyant scheduling", *Theoretical Computer Science*, **130**, 17–47, 1994.
15. A. Tanenbaum, "Operating systems: design and implementation", Prentice-Hall, 2001.