Scalably Scheduling Processes with Arbitrary Speedup Curves

Jeff Edmonds* Kirk Pruhs†

Abstract

We give a scalable ($(1+\epsilon)$-speed $O(1)$-competitive) nonclairvoyant algorithm for scheduling jobs with sublinear nondecreasing speed-up curves on multiple processors with the objective of average response time.

1 Introduction

Computer chip designers are agreed upon the fact that chips with hundreds to thousands of processors chips will dominate the market in the next decade. The founder of chip maker Tilera asserts that a corollary to Moore's law will be that the number of cores/processors will double every 18 months [15]. Intel's director of microprocessor technology asserts that while processors will get increasingly simple, software will need to evolve more quickly than in the past to catch up [15]. In fact, it is generally agreed that developing software to harness the power of multiple processors is going to be a much more difficult technical challenge than the development of the hardware. In this paper, we consider one such software technical challenge: developing operating system algorithms/policies for scheduling processes with varying degrees of parallelism on a multiprocessor.

We will consider the setting where n processes/jobs arrive to the system of m processors over time. Job J_i arrives at time r_i, and has a work requirement w_i. At each point of time, a scheduling algorithm specifies which job is run on each processor at that time. An operating system scheduling algorithm generally needs to be *nonclairvoyant*, that is, the algorithm does not require internal knowledge about jobs, say for example the jobs' work requirement, since such information is generally not available to the operating systems. Job J_i completes after its w_i units of work has been processed. If a job J_i completes at time C_i, then its response time is $C_i - r_i$. In this paper we will consider the schedule quality of service metric *total response time*, which for a schedule S is defined to be $F(S) = \sum_{i=1}^{n}(C_i - r_i)$. For a fixed number of jobs, total response time is essentially equivalent to average response time. Average response time is by far the mostly commonly used schedule quality of service metric. Before starting our discussion of multiprocessor scheduling, let us first review resource augmentation analysis and single processor scheduling.

For our purposes here, resource augmentation analysis compares an online scheduling algorithm against an offline optimal scheduler with slower processors. Online scheduling algorithm A is s-speed c-competitive if

$$\max_I \frac{F(A_s(I))}{F(\mathrm{Opt}_1(I))} \leq c$$

where $A_s(I)$ is the schedule produced by algorithm A with speed s processors on input I, and $\mathrm{Opt}_1(I)$ is the optimal schedule for unit speed processors on input I, and $F(S)$ is the total response

*York University, Canada. jeff@cs.yorku.ca. Supported in part by NSERC Canada.

†Computer Science Department. University of Pittsburgh. kirk@cs.pitt.edu. Supported in part by an IBM faculty award, and by NSF grants CNS-0325353, CCF-0514058, IIS-0534531, and CCF-0830558.

Dagstuhl Seminar Proceedings 10071
Scheduling
http://drops.dagstuhl.de/opus/volltexte/2010/2546

time for schedule S [13, 17]. An algorithm A is said to be *scalable* if for every $\epsilon > 0$, there is a constant c_ϵ such A is $(1+\epsilon)$-speed c_ϵ-competitive [18, 19]. A scalable algorithm is $O(1)$-competitive on inputs I where $\text{Opt}_1(I)$ is approximately $\text{Opt}_{1+\epsilon}(I)$, which intuitively are inputs that do not fully load the server. So as the load increases, the performance of a scalable algorithm should be reasonably close to the performance of the optimal algorithm up until the server is almost fully loaded. For a more detailed explanation see [18, 19].

The nonclairvoyant algorithm Shortest Elapsed Time First (SETF) is scalable [13] for scheduling jobs on a single processor for the objective of total response time. SETF shares the processor equally among all processes that have been processed the least to date. Intuitively, SETF gives priority to more recently arriving jobs, until they have been processed as much as older jobs, at which point all jobs are given equal priority. The process scheduling algorithm used by most standard operating systems, e.g. Unix, essentially schedules jobs in way that is consistent with this intuition. No nonclairvoyant scheduling algorithm can be $O(1)$-competitive for total response time if compared against the optimal schedule with the same speed [16]. The intuition is that one can construct adversarial instances where the load is essentially the capacity of the system, and there is no time for the nonclairvoyant algorithm to recover from any scheduling mistakes.

One important issue that arises when scheduling jobs on a multiprocessor is that jobs can have widely varying degrees of parallelism. That is, some jobs may be be considerably sped up when simultaneously run on to multiple processors, while some jobs may not be sped up at all (this could be because the underlying algorithm is inherently sequential in nature, or because the process was not coded in a way to make it easily parallelizable). To investigate this issue, we adopt the following general model used in [8]. Each job consists of a sequence of phases. Each phase consists of a positive real number that denotes the amount of work in that phase, and a speedup function that specifies the rate at which work is processed in this phase as a function of the number of processors executing the job. The speedup functions may be arbitrary, other than we assume that they are nondecreasing (a job doesn't run slower if it is given more processors), and sublinear (a job satisfies Brent's theorem, that is increasing the number of processors doesn't increase the efficiency of computation).

The most obvious scheduling algorithm in the multiprocessor setting is Equi-partition (Equi), which splits the processors evenly among all processes. Equi is analogous to the Round Robin or Processor Sharing algorithm in the single processor setting. In what is generally regarded as a quite complicated analysis, it is shown in [8] that Equi is a $(2+\epsilon)$-speed $(\frac{2s}{\epsilon})$-competitive for total response time. It is also known that, even in the case of a single processor, speed at least $2+\epsilon$ is required in order for Equi to be $O(1)$-competitive for total response time [13].

1.1 Our Results

In this paper we introduce a nonclairvoyant algorithm, which we call $\text{LAPS}_{\langle\beta,s\rangle}$, and show that it is scalable for scheduling jobs with sublinear nondecreasing speedup curves with the objective of total response time.

$\text{LAPS}_{\langle\beta,s\rangle}$ (Latest Arrival Processor Sharing) Definition: This algorithm is parameterized by a real $\beta \in (0, 1]$. Let n_t be the number of jobs alive at time t. The processors are equally partitioned among the $\lceil \beta n_t \rceil$ jobs with the latest arrival times (breaking ties arbitrarily but consistently). Here s is the speed of the processor, which will be useful in our analysis.

Note that $\text{LAPS}_{\langle\beta,s\rangle}$ is a generalization of Equi since $\text{LAPS}_{\langle 1,s\rangle}$ identical to Equi_s. But as β decreases, $\text{LAPS}_{\langle\beta,s\rangle}$, in a manner reminiscent of SETF, favors more recently released jobs. The main result of this paper, which we prove in section 3, is then:

Theorem 1. $\text{LAPS}_{\langle\beta,s\rangle}$, *with speed* $s = (1+\beta+\epsilon)$ *processors, is* $\left(\frac{4s}{\beta\epsilon}\right)$*-competitive algorithm for scheduling processes with sublinear nondecreasing speedup curves for the objective of average response time. The same result holds if* $\text{LAPS}_{\langle\beta,s\rangle}$ *is given* s *times as many speed one processors as the adversary.*

Essentially this shows that, perhaps somewhat surprisingly, that a nonclairvoyant scheduling algorithm can perform roughly as well in the setting of scheduling jobs with arbitrary speedup curves on a multiprocessor, as it can when scheduling jobs on a single processor. Our proof of Theorem 1 uses a simple amortized local competitiveness argument with a simple potential function. When $\beta = 1$, that is when $\text{LAPS}_{\langle\beta,s\rangle} = \text{Equi}_s$, we get as a corollary of Theorem 1 that Equi is $(2+\epsilon)$-speed $(\frac{2s}{\epsilon})$-competitive, matching the bound given in [8], but with a much easier proof.

There is a unique feature of $\text{LAPS}_{\langle\beta,s\rangle}$ that is worth mentioning. $\text{LAPS}_{\langle\beta,s\rangle}$ is only $O(1)$-competitive when s is sufficiently larger (depending on β) than 1. Previously analyses showing $(1+\epsilon)$-speed $O(1)$-competitive were for algorithms that were not parameterized by ϵ. For example, in was shown in [13] that on one processor SETF is simultaneously $(1+\epsilon)$-speed $(1+\frac{1}{\epsilon})$-competitive for all $\epsilon > 0$ simultaneously. Thus we need to introduce a new notion of scalability. We say that an algorithm A_s, that is parameterized by a speed s, is *scalable* if for every $\epsilon > 0$, there is a constant c_ϵ such the algorithm $A_{1+\epsilon}$ is $(1+\epsilon)$-speed c_ϵ-competitive. So note that here the algorithm depends on the choice of ϵ.

Theorem 1 also improves the best known competitiveness result for broadcast/multicast pull scheduling. It is easiest to explain broadcast scheduling in context of a web server serving static content. In this setting, it is assumed that the web server is serving content on a broadcast channel. So if the web server has multiple unsatisfied requests for the same file, it need only broadcast that file once, simultaneously satisfying all the users who issued these requests. [11] showed how to convert any s-speed c-competitive nonclairvoyant algorithm for scheduling jobs with arbitrary speedup curves into a $2s$-speed c-competitive algorithm for broadcast scheduling. Using this result, and the analysis of Equi from [8], [11] showed that a version of Equi $(4+\epsilon)$-speed $O(1)$-competitive for broadcast scheduling with the objective of average response time. Using Theorem 1 we can then deduce that a broadcast version of $\text{LAPS}_{\langle\beta,s\rangle}$ is $(2+\epsilon)$-speed $O(1)$-competitive for broadcast scheduling with the objective of average response time.

1.2 Related Results

For the objective of total response time on a single processor, the competitive ratio of every deterministic nonclairvoyant algorithm is $\Omega(n^{1/3})$, and the competitive ratio of every randomized nonclairvoyant algorithm against an oblivious adversary is $\Omega(\log n)$ [16]. There is a randomized algorithm, Randomized Multi-Level Feedback Queues, that is $O(\log n)$-competitive against an oblivious adversary [2, 14]. The online clairvoyant algorithm Shortest Remaining Processing time is optimal for total response time. The competitive analysis of SETF_s for single processor scheduling was improved for cases when $s \gg 1$ in [3].

Variations of Equipartition are built into many technologies. For example, the congestion control protocol in the TCP Internet protocol essentially uses Equipartition to balance bandwidth to TCP connections through a bottleneck router. Extensions of the analysis of Equi in [8] to analyzing TCP can be found in [9, 10]. Other extensions to the analysis of Equi in [8] for related scheduling problems can found in [20–22]. In our results here, we essentially ignore the extra advantage that the online algorithm gains from having faster processors instead of more processors. [8] gives a better competitive ratio for Equi in the model with faster processors.

There are many related scheduling problems with other objectives, and/or other assumptions about the machine and job instance. Surveys can be found in [18, 19].

2 Preliminaries

In this section, we review the formal definitions introduced in [8]. An instance consists of a collection $J = \{J_1, \ldots, J_n\}$ where *job* J_i has a *release/arrival time* r_i and a sequence of phases $\langle J_i^1, J_i^2, \ldots, J_i^{q_i} \rangle$. Each phase is an ordered pair $\langle w_i^q, \Gamma_i^q \rangle$, where w_i^q is a positive real number that denotes the amount of *work* in the phase and Γ_i^q is a function, called the *speedup function*, that maps a nonnegative real number to a nonnegative real number. $\Gamma_i^q(p)$ represents the rate at which work is processed for phase q of job J_i when run on p processors running at speed 1. If these processors are running at speed s, then work is processed at a rate of $s\Gamma_i^q(p)$.

A schedule specifies for each time, and for each job, (1) a nonnegative real number specifying the number of processors assigned to that job, and (2) a nonnegative real speed. The number of processors assigned at any time can be at most m, the number of processors. Note that, formally, a schedule does not specify an assignment of copies of jobs to processors.

A nonclairvoyant algorithm only knows when processes have been released and finished in the past, and which processes have been run on each processor each time in the past. In particular, a nonclairvoyant algorithm does not know w_i^q, nor the current phase q, nor the speedup function Γ_i^q.

The *completion time* of a job J_i, denoted C_i, is the first point of time when all the work of the job J_i has been processed. Note that in the language of scheduling, we are assuming that preemption is allowed, that is, a job maybe be suspended and later restarted from the point of suspension. A job is said to be *alive* at time t, if it has been released, but has not completed, i.e., $r_i \leq t \leq C_i$. The *response/flow time* of job J_i is $C_i - r_i$, which is the length of the time interval during which the job is active. Let n_t be the number of active jobs at time t. Another formulation of total flow time is $\int_0^\infty n_t dt$.

A phase of a job is *parallelizable* if its speedup function is $\Gamma(p) = p$. Increasing the number of processors allocated to a parallelizable phase by a factor of s increases the rate of processing by a factor of s. A phase is *sequential* if its speedup function is $\Gamma(p) = 1$, for all $p \geq 0$. The rate that work is processed in a sequential phase is independent of the number of processors, even if it is zero. A speedup function Γ is *nondecreasing* if and only if $\Gamma(p_1) \leq \Gamma(p_2)$ whenever $p_1 \leq p_2$. A speedup function Γ is *sublinear* if and only if $\Gamma(p_1)/p_1 \geq \Gamma(p_2)/p_2$ whenever $p_1 \leq p_2$. We assume all speedup functions Γ in the input instance are nondecreasing and sublinear.

Let A be an algorithm and J an instance. We denote the schedule output by A with speed s processors on J as $A_s(J)$. Let $\mathrm{Opt}(J)$ be the optimal schedule with unit speed processors on input J. We let $F(S)$ denote the total response time incurred in schedule S,

3 Analysis of LAPS$_{\langle \beta, s \rangle}$

This section will be devoted or proving Theorem 1, that LAPS$_{\langle \beta, s \rangle}$ is scalable. We will assume that the online algorithm has sm unit speed processors while the adversary has m unit speed processors. Since in the context of preemptive scheduling, a speed s processor is always at least as useful as s unit speed processors, the analysis for speed augmentation will follow as a direct consequence of our analysis for machine augmentation.

Following the lead of [8] and [22], the first step in our proof is to prove that there is a worst case instance that contains only sequential and parallelizable phases.

Lemma 2. *Let A be a nonclairvoyant scheduler. Let J be an instance of jobs with sublinear-nondecreasing speedup functions. Then there is a job set J' that with only sequential and parallelizable phases such that $F(A(J')) = F(A(J))$ and $F(\text{Opt}(J')) \leq F(\text{Opt}(J))$.*

Proof. We explain how to modify J to obtain J'. We perform the following modification for each time t and each job J_i that A runs during the infinitesimal time $[t, t + dt]$. Let w be the infinitesimal amount of work processed by A during this time, and Γ the speedup function for the phase containing w. Let p_a denote the number of processors allocated by A to w at time t. So the amount of work in w is $\Gamma(p_a)dt$. Let p_o denote the number of processors allocated by Opt to w. It is important to note that Opt may not process w at time t. If $p_0 \leq p_a$, we then modify J by replacing this w amount of work with a sequential phase with work $w' = dt$. If $p_o > p_a$, we then modify J by replacing this w amount of work with parallelizable phase with work $w' = p_a dt$. Note that by construction, A will not be able to distinguish between the instances J and J' during the time period $[t, t + dt]$. Hence, since A is nonclairvoyant $A(J') = A(J)$. We are now left to argue that $F(\text{Opt}(J')) \leq F(\text{Opt}(J))$. We will accomplish this by giving a schedule X for J' that has total response time at most $F(\text{Opt}(J))$.

First consider the case that $p_o \leq p_a$. Because the speedup function Γ of the phase containing the work w is non-decreasing, it took $\text{Opt}(J)$ more than time dt to finish the work w. The schedule X will start working on the work w' with p_o processors when $\text{Opt}(J)$ started working on the work w, and then after X completes w', X can let these p_0 processors idle until $\text{Opt}(J)$ completes w.

Now consider that case that $p_o \geq p_a$. Again the schedule X will start working on w' when $\text{Opt}(J)$ started working on w. We now want to argue that X can complete w' with p_o processors in less time than it took $\text{Opt}(J)$ to complete w with p_o processors. It took time $\frac{p_a dt}{p_o}$ time for X to complete w' since the $p_a dt$ work in w' is parallelizable. It took $\text{Opt}(J)$ time $\frac{\Gamma(p_a)dt}{\Gamma(p_o)}$ to complete the $\Gamma(p_a)dt$ work in w. The fact X completes w' before $\text{Opt}(J)$ completes w follows since $\frac{p_a}{p_o} \leq \frac{\Gamma(p_a)}{\Gamma(p_o)}$ since $p_o \geq p_a$ and Γ is sublinear. □

By Lemma 2, it is sufficient to consider instances that contain only sequential and parallelizable phases. So for the rest of the proof we fix such an instance. Our goal is to bound the number N_t of jobs alive under Opt at time t in terms of what is happening under $\text{LAPS}_{\langle\beta,s\rangle}$ at this same time. This requires the introduction of a fair amount of notation. Let n_t denote number of jobs alive under $\text{LAPS}_{\langle\beta,s\rangle}$ at time t. Let m_t denote the number of these that are within a parallelizable phase at this time and let ℓ_t denote the same except for sequential phases. Let N_t, M_t, and L_t denote the same numbers except under Opt. Let $\widehat{N}_t$ denote the number jobs at time t that $\text{LAPS}_{\langle\beta,s\rangle}$ has not completed, but for which $\text{LAPS}_{\langle\beta,s\rangle}$ is ahead of Opt. Let $\widehat{\ell}_t$ denote the number jobs that $\text{LAPS}_{\langle\beta,s\rangle}$ has not completed at time t, and either $\text{LAPS}_{\langle\beta,s\rangle}$ is ahead of Opt on this job at this time, or $\text{LAPS}_{\langle\beta,s\rangle}$ is executing a sequential phase on this job at this time.

We note some relationships between these job counts Clearly $\widehat{N}_t \leq N_t$ since Opt has not completed these $\widehat{N}_t$ jobs. $\int_0^\infty L_t dt = \int_0^\infty \ell_t dt$ since each integral is simply the sum of the work of all sequential phases of all jobs. Finally note that $\widehat{\ell}_t \leq \widehat{N}_t + \ell_t$ since each of the $\widehat{\ell}_t$ jobs is either in a sequential phase, or is included in the count $\widehat{N}_t$. Thus we can conclude that the total cost to Opt is bounded as follows:

$$F\left(\text{Opt}(J)\right) = \int_0^\infty N_t dt = \frac{1}{2}\int_0^\infty \left(N_t + (M_t + L_t)\right) dt \geq \frac{1}{2}\int_0^\infty \left(\widehat{N}_t + 0 + \ell_t\right) dt \geq \int_0^\infty \frac{\widehat{\ell}_t}{2} dt$$

To prove c-competitiveness using an amortized local competitiveness argument [8, 18, 19], we need to define a potential function Φ_t such that the following conditions hold:

Boundary: Φ is initially and finally 0, that is, $\Phi_0 = \Phi_\infty = 0$.

Arrival: Φ_t does not increase when a new job arrives.

Completion: Φ_t does not increase when either the online algorithm or the adversary complete a job.

Running: For all times t when no job arrives or is completed,

$$n_t + \frac{d\Phi_t}{dt} \leq \frac{c\widehat{\ell_t}}{2} \tag{1}$$

By integrating the running condition over time, and using the boundary, arrival, and completion conditions, one can conclude that

$$F(\text{LAPS}_{\langle\beta,s\rangle}) = \int_0^\infty n_t dt \leq \int_0^\infty n_t dt + [\Phi_\infty - \Phi_0] = \int_0^\infty \left(n_t + \frac{d\Phi_t}{dt}\right) dt \leq \int_0^\infty \left(\frac{c\widehat{\ell_t}}{2}\right) dt \leq c \cdot F(\text{Opt})$$

We define the potential function Φ_t as follows. Let J_i denote the i^{th} of the n_t jobs currently alive under $\text{LAPS}_{\langle\beta,s\rangle}$ at time t, sorted by their arrival times r_i (breaking ties arbitrarily but consistently). So J_1 is the earliest arriving job, and J_{n_t} is the latest arriving job, among the jobs alive for $\text{LAPS}_{\langle\beta,s\rangle}$ at time t. Let x_i denote the amount of parallelizable work of J_i has been completed by Opt before time t, but that was not completed by $\text{LAPS}_{\langle\beta,s\rangle}$ before time t. Let $\gamma = \frac{2}{\epsilon m}$. The potential function is then:

$$\Phi_t = \gamma \sum_{i=1}^{n_t} i \cdot \max(x_i, 0) \tag{2}$$

The boundary conditions for Φ_t are trivially satisfied. If a new job J_j arrives, then the value of the potential function does not increase because $\text{LAPS}_{\langle\beta,s\rangle}$ will not be behind on that job (i.e. $x_j = 0$). If $\text{LAPS}_{\langle\beta,s\rangle}$ completes job J_j, then $j \max(x_j, 0) = 0$ since $x_j = 0$, and removing job J_j from the summation will not increase the coefficient i of any other job. Opt completing a job J_j has no effect on the potential function at all.

To establish inequality (1), consider an infinitesimal period of time $[t, t+dt]$ during which no jobs arrive or are completed by either Equi or Opt. Consider how much Φ_t can increase due Opt's processing during this period. Without loss of generality, Opt processes only parallelizable work. Opt processes this parallelizable work at rate at most m. This increases the sum of the x_i's for these jobs by a total of at most $m\ dt$. Opt can increase Φ_t the most by working only on the most recently arrived job because its coefficient is maximal. Since the most recently arrived job has coefficient n_t in Φ_t, the rate of increase in Φ_t due to Opt's processing is at most $\gamma m n_t$.

We now need to bound how much Φ_t must decrease due to $\text{LAPS}_{\langle\beta,s\rangle}$'s processing during the same infinitesimal period of time $[t, t+dt]$. The algorithm $\text{LAPS}_{\langle\beta,s\rangle}$ works on the $f_t = \lceil \beta n_t \rceil$ jobs with the latest arrival times. Ideally, for these jobs, the term $\max(x_i, 0)$ in the potential function decreases at a rate of $\frac{sm}{f_t}$. However, there are two possible reasons that this desired decrease will not occur. The first possible reason is that $\text{LAPS}_{\langle\beta,s\rangle}$ has processed one of these jobs more than Opt has at this time. For such jobs, $x_i \leq 0$ and hence $\max(x_i, 0)$ is already 0. The second possible reason is that the job is in a sequential phase under $\text{LAPS}_{\langle\beta,s\rangle}$ at this time. Because x_i measures only the work in parallelizable phases, any processing that $\text{LAPS}_{\langle\beta,s\rangle}$ does on a sequential phase does not decrease $\max(x_i, 0)$. Recall that we defined $\widehat{\ell_t}$ to be the number jobs that have at least

one of these properties. In the worst case, these $\widehat{\ell}_t$ jobs are those that arrive the most recently. Let us for the moment assume that $\widehat{\ell}_t \leq f_t$. In this case, $\text{LAPS}_{\langle\beta,s\rangle}$ effectively decreases the term $\max(x_i, 0)$ only for the jobs with coefficients in the range $[n_t - f_t + 1, n_t - \widehat{\ell}_t]$. The value of $\max(x_i, 0)$ decreases for these jobs at a rate of $\frac{sm}{f_t}$. Hence, the decrease in Φ_t due to $\text{LAPS}_{\langle\beta,s\rangle}$'s processing is at least

$$\begin{aligned}
& \gamma \sum_{i=n_t-f_t+1}^{n_t-\widehat{\ell}_t} i \cdot \frac{dx_i}{dt} \\
= \; & \gamma \sum_{i=n_t-f_t+1}^{n_t-\widehat{\ell}_t} i \cdot \left(-\frac{sm}{f_t}\right) \\
= \; & \frac{-sm\gamma}{2f_t}\left((n_t-\widehat{\ell}_t)(n_t-\widehat{\ell}_t+1) - (n_t-f_t)(n_t-f_t+1)\right) \\
= \; & \frac{sm\gamma}{2f_t}\left(2n_t\widehat{\ell}_t - \widehat{\ell}_t^2 + \widehat{\ell}_t - 2n_tf_t + f_t^2 - f_t\right) \\
\leq \; & \frac{sm\gamma}{2f_t}\left(2n_t\widehat{\ell}_t - 2n_tf_t + f_t^2 - f_t\right) \\
\leq \; & \frac{sm\gamma n_t\widehat{\ell}_t}{f_t} - sm\gamma n_t + \frac{sm\gamma f_t}{2} - \frac{sm\gamma}{2} \\
= \; & \frac{sm\gamma n_t\widehat{\ell}_t}{\lceil\beta n_t\rceil} - sm\gamma n_t + \frac{sm\gamma\lceil\beta n_t\rceil}{2} - \frac{sm\gamma}{2} \\
\leq \; & \frac{sm\gamma n_t\widehat{\ell}_t}{\beta n_t} - sm\gamma n_t + \frac{sm\gamma(\beta n_t+1)}{2} - \frac{sm\gamma}{2} \\
= \; & \frac{sm\gamma\widehat{\ell}_t}{\beta} - sm\gamma n_t + \frac{sm\gamma\beta n_t}{2}
\end{aligned}$$

Substituting back our bounds on the decrease in Φ_t due to $\text{LAPS}_{\langle\beta,s\rangle}$'s processing, and the increase in Φ_t due to Opt's processing, back into (1), we get:

$$\begin{aligned}
n_t + \frac{d\Phi_t}{dt} \; & \leq \; n_t + \left((\gamma m n_t) + \left(\frac{sm\gamma\widehat{\ell}_t}{\beta} - sm\gamma n_t + \frac{sm\gamma\beta n_t}{2}\right)\right) \\
& = \; \left(1 + \gamma m - sm\gamma + \frac{sm\gamma\beta}{2}\right) n_t + \frac{sm\gamma\widehat{\ell}_t}{\beta} \\
& \leq \; \frac{sm\gamma\widehat{\ell}_t}{\beta} \\
& = \; \frac{2s\widehat{\ell}_t}{\beta\epsilon} \\
& = \; \frac{c\cdot\widehat{\ell}_t}{2}
\end{aligned}$$

The last inequality follows since by substituting in $\gamma = \frac{2}{\epsilon m}$ and $s = 1+\beta+\epsilon$

$$1 + \gamma - s\gamma + \frac{s\gamma\beta}{2} = 1 + \frac{2}{\epsilon} - 2\frac{1+\beta+\epsilon}{\epsilon} + \frac{(1+\beta+\epsilon)\beta}{\epsilon}$$

which one can verify is not positive by multiplying through by ϵ, and collecting like terms.

Now consider that case in which $\widehat{\ell}_t \geq f_t$. In this case all of the $f_t = \lceil \beta n_t \rceil$ jobs being processed LAPS$_{\langle \beta, s \rangle}$ might be in sequential phases or have $\max(x_i, 0) = 0$ and hence LAPS$_{\langle \beta, s \rangle}$'s processing might not decrease Φ_t. Evaluating inequality (1), we find that

$$\begin{aligned} n_t + \frac{d\Phi_t}{dt} &\leq n_t + \gamma m n_t \\ &= \left(1 + \frac{2}{\epsilon}\right) n_t \\ &\leq \frac{2(1+\beta+\epsilon)}{\epsilon\beta} \lceil \beta n_t \rceil \\ &= \frac{2s}{\beta\epsilon} \cdot f_t \\ &\leq \frac{c \cdot \widehat{\ell}_t}{2} \end{aligned}$$

4 Conclusion

The algorithm LAPS algorithm, that we introduced in this paper, has found application in several subsequent papers. In was used in [4] as the job selection algorithm in a $O(1)$-competitive speed scaling algorithm on a single processor with the objective of minimizing a linear combination of response time and energy. LAPS was used instead of the more obvious choice of SETF because the analysis of speed scaling algorithms generally require amortized local competitiveness arguments, and it is not clear what potential function one should use with SETF. The potential function used in [4] is a modification of the potential function that we used here. A modification of LAPS was used in [5] as the job selection algorithm in a $O(\log m)$-competitive speed scaling algorithm on a multiprocessor processor with the objective of minimizing a linear combination of response time and energy. Finally [1] showed that the broadcast version of LAPS is scalable for broadcast scheduling, answering a decade old open question of whether such an algorithm exists. A scalable algorithm for broadcasting scheduling of unit work pages was given in [12].

Contemporaneously and subsequent to this research, other scalable algorithms, where the algorithm knows the speed $1 + \epsilon$ a priori, were discovered for broadcast scheduling [1, 6, 7, 12]. So it seems like allowing algorithms to depend on the speed is useful in finding scalable algorithms. It would be interesting to either find a scalable algorithm for these problems that didn't need to know the speed $1 + \epsilon$ a priori, or to prove that no such algorithm exists.

Acknowledgments: We thank Nicolas Schabanel and Julien Robert for helpful discussions.

References

[1] Nikhil Bansal, RaviShankar Krishnaswamy, and Vishwanath Nagarajan. Better scalable algorithms for broadcast scheduling. 2009.

[2] Luca Becchetti and Stefano Leonardi. Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. *J. ACM*, 51(4):517–539, 2004.

[3] Piotr Berman and Chris Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, 6(2):181–193, 1999.

[4] Ho-Leung Chan, Jeff Edmonds, Tak Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. In *Symposium on Theoretical Aspects of Computer Science*, pages 255–264, 2009.

[5] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *Symposium on Parallel Algorithms and Architectures*, pages 1–10, 2009.

[6] Chandra Chekuri, Sungjin Im, and Benjamin Moseley. Minimizing maximum response time and delay factor in broadcast scheduling. In *European Symposium on Algorithms*, 2009.

[7] Chandra Chekuri and Benjamin Moseley. Online scheduling to minimize the maximum delay factor. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1125, 2009.

[8] Jeff Edmonds. Scheduling in the dark. *Theoretial Computer Science*, 235:109–141, 2000.

[9] Jeff Edmonds. On the competitiveness of aimd-tcp within a general network. In *LATIN*, pages 567–576, 2004.

[10] Jeff Edmonds, Suprakash Datta, and Patrick Dymond. Tcp is competitive against a limited adversary. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 174–183, 2003.

[11] Jeff Edmonds and Kirk Pruhs. Multicast pull scheduling: When fairness is fine. *Algorithmica*, 36(3):315–330, 2003.

[12] Sungjin Im and Benjamin Moseley. An online scalable algorithm for average flowtime in broadcast scheduling. 2010. to appear in ACM-SIAM Symposium on Discrete Algorithms.

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

[14] Bala Kalyanasundaram and Kirk Pruhs. Minimizing flow time nonclairvoyantly. *J. ACM*, 50(4):551–567, 2003.

[15] Rick Merritt. Cpu designers debate multi-core future. *EE Times*, June 2008.

[16] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130:17–47, 1994.

[17] Cynthia Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, pages 163–200, 2002.

[18] Kirk Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Performance Evaluation Review*, 34(4):52–58, 2007.

[19] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling. In *Handbook on Scheduling*. CRC Press, 2004.

[20] Julien Robert and Nicolas Schabanel. Non-clairvoyant batch sets scheduling: Fairness is fair enough. In *European Symposium on Algorithms*, pages 741–753, 2007.

[21] Julien Robert and Nicolas Schabanel. Pull-based data broadcast with dependencies: be fair to users, not to items. In *ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[22] Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *Symposium on Discrete Algorithms*, pages 491–500, 2008.