

Scalably Scheduling Processes with Arbitrary Speedup Curves

Jeff Edmonds*

Kirk Pruhs†

“With multi-core it’s like we are throwing this Hail Mary pass down the field and now we have to run down there as fast as we can to see if we can catch it.”

— David Patterson, UC Berkeley computer science professor

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 Tiler asserts that a corollary to Moore’s law will be that the number of cores/processors will double every 18 months [9]. 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 [9]. 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 over time. Job J_i arrives at time r_i , and has a work requirement w_i . 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. So at each point of time, a *nonclairvoyant* scheduling algorithm specifies which job is run on each processor at that time knowing only when jobs arrived in the past, what the job assignment was in the past, and when jobs completed in the past. 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

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

where $A_s(I)$ is the schedule produced by algorithm A with speed s processors on input I , and $\text{Opt}_1(I)$ is the optimal total response time schedule for unit speed processors on input I [7, 11]. A $(1+\epsilon)$ -speed $O(1)$ -competitive algorithm is said to be *scalable* [12, 13]. (The constant in the competitive ratio will generally depend upon ϵ .) To understand the motivation for the definition of *scalability* consider the sort of quality of service curve, such as the one in figure 1, that is ubiquitous in server systems. That is, there is a relatively modest degradation in quality of service as the load increases until one nears some threshold — this threshold is essentially the capacity of the system — after which any increase in the load precipitously degrades the quality of service provided by the server. The concept of load is not so easy to formally define, but generally reflects the number of users of the system. Note that increasing the speed of a server by a factor of s is essentially equivalent to lowering the load on the server by a factor of s . A scalable algorithm is $O(1)$ -

*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.

competitive on inputs I where $\text{Opt}_1(I)$ is approximately $\text{Opt}_{1+\epsilon}(I)$. Thus the performance curve of a scalable scheduling algorithm should be at no worse than shown in figure 1; That is, the scheduling algorithm should scale reasonably well up to quite near the capacity of the system.

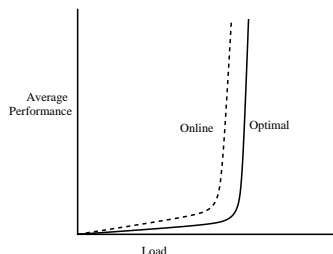


Figure 1: The worst possible performance curve of an $(1+\epsilon)$ -speed $O(1)$ -competitive online algorithm.

The nonclairvoyant algorithm Shortest Elapsed Time First (SETF) is scalable [7] 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 [10]. 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 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 [3]. 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 [3] 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 [7].

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.

LAPS_{⟨β,s⟩} (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 1.1. *LAPS_{⟨β,s⟩}, with speed $s = (1 + \beta + \epsilon)$ processors, is $(\frac{4s}{\beta\epsilon})$ -competitive algorithm for scheduling processes with sublinear nondecreasing speedup curves for the objective of average response time. The same result holds if LAPS_{⟨β,s⟩} 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 1.1 essentially 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 1.1 that Equi is $(2+\epsilon)$ -speed $(\frac{2s}{\epsilon})$ -competitive, matching the bound given in [3], but with a much easier proof.

There is one unique feature of $\text{LAPS}_{\langle\beta,s\rangle}$ that is worth mentioning. As we show in section 4, $\text{LAPS}_{\langle\beta,s\rangle}$ is only $O(1)$ -competitive when s is sufficiently larger (depending on β) than 1. All the other scalability analyzes of scheduling algorithms give $O(1)$ -competitiveness for any speed greater than one. For example, one one processor SETF is simultaneously $(1+\epsilon)$ -speed $(1+\frac{1}{\epsilon})$ -competitive for all $\epsilon > 0$ simultaneously.

Theorem 1.1 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. [6] 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 [3], [6] showed that a version of Equi $(4+\epsilon)$ -speed $O(1)$ -competitive for broadcast scheduling with the objective of average response time. Using Theorem 1.1 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 $\Omega(\log n)$ [10]. There is a randomized algorithm, Randomized Multi-Level Feedback Queues, that is $O(\log n)$ -competitive against an oblivious adversary [1, 8]. 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 [2].

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 [3] to analyzing TCP can be found in [4, 5]. Other extensions to the analysis of Equi in [3] for related scheduling problems can be found in [14–16]. In our results here, we essentially ignore the extra advantage that the online algorithm gains from having faster processors instead of more processors. [3] 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 [12, 13].

2 Preliminaries

In this section, we review the formal definitions introduced in [3]. An instance consists of a collection $J = \{J_1, \dots, J_n\}$ where job J_i has a *release/arrival time* r_i and a sequence of phases $\langle J_i^1, J_i^2, \dots, J_i^{q_i} \rangle$. 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. $\Gamma_i^q(\rho)$ represents the rate at which work is executed for phase q of job i when given ρ processors.

A phase of a job is *parallelizable* if its speedup function is $\Gamma(\rho) = \rho$. Increasing the number of processors allocated to a parallelizable job by a factor of s increases the rate of processing by a factor of s . A phase is *sequential* if its speedup function is $\Gamma(\rho) = 1$, for all $\rho \geq 0$. The rate that work is processed in a sequential phase is independent of the number of processors, even if it is zero. Formally, a speedup function Γ is *nondecreasing* if and only if $\Gamma(\rho_1) \leq \Gamma(\rho_2)$ whenever $\rho_1 \leq \rho_2$. Formally, a speedup function Γ is *sublinear* if and only if $\Gamma(\rho_1)/\rho_1 \geq \Gamma(\rho_2)/\rho_2$ whenever $\rho_1 \leq \rho_2$.

A *schedule* \mathcal{S}_s for a given job set J with n jobs on sp processors is a function from $\{1, \dots, n\} \times [0, \infty)$ to $[0, sp]$, where $\mathcal{S}_s(i, t)$ is the number of processors allocated to job J_i at time t . We allow a job to be allocated a non-integral number of processors (this is allowable because we are considering preemptive scheduling). In order for \mathcal{S}_s to be feasible it must be that case that for all t , $\sum_{i=1}^n \mathcal{S}_s(i, t) \leq sp$, that is at most sp processors are allocated at any given time. Also in order to be feasible, we require that for all i , there exist $r_i = c_i^0 < c_i^1 < \dots < c_i^{q_i} = C_i$ such that for all $1 \leq q \leq q_i$, $\int_{c_i^{q-1}}^{c_i^q} \Gamma_i^q(\mathcal{S}_s(i, t)) dt = w_i^q$, which ensures that before a phase of a job begins, the job must have been released and all of the previous phases of the job must have been completed. The *completion* time of a job J_i , denoted C_i , is the completion time of the last phase of the job. 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 time* of job J_i , $C_i - r_i$, is the length of the time interval during which the job is *alive*. Let n_t be the number of jobs alive at time t . Then another formulation of total response time is $\int_0^\infty n_t dt$.

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

We will assume that the online algorithm has more processors than the adversary. 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. For simplicity of analysis we will scale the number of processors so that the adversary has one unit speed processor, and $\text{LAPS}_{\langle\beta,s\rangle}$ has $s = 1 + \epsilon + \beta$ unit speed processors. So when we say that ρ processors are devoted to a job, this really means that ρp processors are devoted to the job, where p is the number of processors for the adversarial/optimal schedule that we are comparing $\text{LAPS}_{\langle\beta,s\rangle}$ to. Following the lead of [3] and [16], the first step in our proof is to prove that there is a worst case instance that contains only sequential and parallelizable phases.

LEMMA 3.1. *Let S_s be a nonclairvoyant scheduler with s unit speed processors. 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 $F(S_s(J')) = F(S_s(J))$ and $F(\text{Opt}(J')) \leq F(\text{Opt}(J))$, where Opt means optimal on one unit speed processor.*

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 S_s runs during the infinitesimal time $[t, t + dt]$. Let w be the infinitesimal amount of work processed by S_s during this time, and Γ the speedup function for the phase containing w . Let p_s denote the number of processors allocated by S_s to w at time t . So the amount of work in w is $\Gamma(p_s)dt$. 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 $p_o \leq p_s$, we then modify J by replacing this w amount of work with a sequential phase with work $w' = dt$. If $p_o > p_s$, we then modify J by replacing this w amount of work with parallelizable phase with work $w' = p_s dt$. Note that by construction, S_s will not be able to distinguish between the instances J and J' during the time period $[t, t + dt]$. Hence, since S_s is nonclairvoyant $S_s(J') = S_s(J)$. We are now left to argue that $F(\text{Opt}(J')) \leq F(\text{Opt}(J))$. We will give a schedule X for J' that has total response time at most $F(\text{Opt}(J))$.

First consider the case that $p_o \leq p_s$. 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_o processors idle until $\text{Opt}(J)$ completes w .

Now consider that case that $p_o \geq p_s$. 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 $\frac{p_s dt}{p_o}$ time for X to complete w' since the $p_s dt$ work in w' is parallelizable. It took $\text{Opt}(J)$ time $\frac{\Gamma(p_s)dt}{\Gamma(p_o)}$ to complete the $\Gamma(p_s)dt$ work in w . The fact X completes w' before $\text{Opt}(J)$ completes w follows since $\frac{p_s}{p_o} \leq \frac{\Gamma(p_s)}{\Gamma(p_o)}$ since $p_o \geq p_s$ and Γ is sublinear. ■

By Lemma 3.1, 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:

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

To establish Theorem 1.1 using an amortized local competitiveness argument [3, 12], we need to define a potential function Φ_t such that the following conditions hold:

Boundary: Φ_t is initially 0, and finally nonnegative.

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,

$$(3.1) \quad n_t + \frac{d\Phi_t}{dt} \leq \frac{1}{2}c\widehat{\ell}_t$$

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

$$\begin{aligned} F(\text{LAPS}_{\langle\beta,s\rangle}) &= \int_0^\infty n_t dt \\ &\leq \int_0^\infty n_t dt + [\Phi_\infty - \Phi_0] \\ &= \int_0^\infty \left(n_t + \frac{d\Phi_t}{dt} \right) dt \\ &\leq \int_0^\infty \left(\frac{1}{2}c\widehat{\ell}_t \right) dt \\ &\leq c \cdot F(\text{Opt}) \end{aligned}$$

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 . So J_1 is the earliest arriving job. 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 $\gamma = \frac{2}{c}$. The potential function is then:

$$(3.2) \quad \Phi_t = \gamma \sum_{i=1}^{n_t} i \cdot \max(x_i, 0)$$

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 $x_j = 0$, 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.

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 to Opt's processing during this period. Without loss of generality, Opt processes only parallelizable work. Opt processes this parallelizable work at at most unit rate. This increases the sum of the x_i 's for these jobs by a

total of at most 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 γn_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 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 $\frac{s}{f_t}$. However, there are two possible reasons that this desired decrease will not occur. The first possible reason is that $\text{LAPS}_{\langle\beta,s\rangle}$, though not done the job, is ahead of Opt at this time. For such jobs, $x_i \leq 0$ and hence $\max(x_i, 0)$ is already 0. The second possible reason is 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, $\text{LAPS}_{\langle\beta,s\rangle}$ 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 $\frac{s}{f_t}$. Hence, the decrease in Φ_t due to $\text{LAPS}_{\langle\beta,s\rangle}$'s processing is at least

$$\begin{aligned} \frac{d\Phi_t}{dt} &= \gamma \sum_{i=n_t-f_t+1}^{n_t-\widehat{\ell}_t} i \cdot \frac{dx_i}{dt} \\ &= \gamma \sum_{i=n_t-f_t+1}^{n_t-\widehat{\ell}_t} i \cdot \left(-\frac{s}{f_t} \right) \\ &= \frac{-s\gamma}{2f_t} \left[(n_t - \widehat{\ell}_t)(n_t - \widehat{\ell}_t + 1) \right. \\ &\quad \left. - (n_t - f_t)(n_t - f_t + 1) \right] \\ &= \frac{s\gamma}{2f_t} \left[2n_t\widehat{\ell}_t - \widehat{\ell}_t^2 + \widehat{\ell}_t - 2n_t f_t + f_t^2 - f_t \right] \\ &\leq \frac{s\gamma}{2f_t} \left[2n_t\widehat{\ell}_t - 2n_t f_t + f_t^2 - f_t \right] \\ &\leq \frac{s\gamma n_t \widehat{\ell}_t}{f_t} - s\gamma n_t + \frac{s\gamma f_t}{2} - \frac{s\gamma}{2} \\ &= \frac{s\gamma n_t \widehat{\ell}_t}{\lceil \beta n_t \rceil} - s\gamma n_t + \frac{s\gamma \lceil \beta n_t \rceil}{2} - \frac{s\gamma}{2} \\ &\leq \frac{s\gamma n_t \widehat{\ell}_t}{\beta n_t} - s\gamma n_t + \frac{s\gamma(\beta n_t + 1)}{2} - \frac{s\gamma}{2} \\ &= \frac{s\gamma \widehat{\ell}_t}{\beta} - s\gamma n_t + \frac{s\gamma \beta n_t}{2} \end{aligned}$$

Now evaluating running condition (line 3.1), we find that

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

The last inequality follows since by substitution in $\gamma = \frac{2}{\epsilon}$ 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 $\text{LAPS}_{\langle\beta,s\rangle}$ might be in sequential phases or have $\max(x_i, 0) = 0$ and hence $\text{LAPS}_{\langle\beta,s\rangle}$'s processing might not decrease Φ_t . Evaluating running condition, we find that

$$\begin{aligned} n_t + \frac{d\Phi_t}{dt} &\leq n_t + [(\gamma n_t) + (0)] \\ &= \left(1 + \frac{2}{\epsilon} \right) n_t \\ &\leq \frac{2(1 + \beta + \epsilon)}{\epsilon} \frac{1}{\beta} \cdot \lceil \beta n_t \rceil \\ &= \frac{2s}{\beta\epsilon} \cdot f_t \\ &\leq \frac{1}{2}c \cdot \widehat{\ell}_t \end{aligned}$$

4 Two Worst Case Instances for $\text{LAPS}_{\langle\beta,s\rangle}$

Theorem 1.1 proves that this algorithm with $s = (1 + \beta + \epsilon)$ processors is $\left(\frac{4s}{\beta\epsilon}\right)$ -competitive. This result breaks if the number of the jobs allocated processors is either decreased from $\lceil \beta n_t \rceil$ to $\lceil o(n_t) \rceil$ or increased to $\lceil (\beta + \epsilon)n_t \rceil$. This section provides two worst case instances for $\text{LAPS}_{\langle\beta,s\rangle}$, the *steady state stream* and the *MPT* [10] examples. These instances show that Theorem 1.1 is tight in each of these two extremes. In fact, the formulation of the algorithm $\text{LAPS}_{\langle\beta,s\rangle}$ was first motivated by trying to find a trade off between these two instances.

THEOREM 4.1. *There is a set of jobs on which $\text{LAPS}_{\langle\beta,s\rangle}$ has a competitive ratio of $\frac{s}{(s-1)\beta} = \frac{s}{(\beta+\epsilon)\beta}$.*

Proof. The set of jobs referred to as *steady state stream* was introduced in [3]. Not only does it provide a lower

bound, it also provides intuition as to why $\text{LAPS}_{\langle\beta,s\rangle}$ manages to work at all.

The job set J consists of two streams, one of parallelizable work and the other of sequential work. The parallelizable stream is a sequence of unit work jobs such that if Opt allocates its p processors to it, it can finish each of these jobs just as the next arrives so that there is always one job alive at any given moment. Similarly, the sequential stream has ℓ very small sequential jobs arrive continuously so that the new one arrives just as the previously arrived ones complete so that there are always ℓ alive. Recall, that sequential work completes at a constant rate even with zero processors allocated to it. These sequential jobs arrive often enough that they are effectively always the most recently arrived jobs.

It is perhaps hard to believe that a nonclairvoyant scheduler, even with sp processors, can perform well here. The scheduler does not know which of the jobs is parallelizable. Hence it wastes most of the processors of the sequential jobs, and falls further and further behind on the parallelizable jobs. $\text{LAPS}_{\langle\beta,s\rangle}$, however, is able to automatically “self adjust” the number of processors wasted on the sequential jobs so that it performs competitively. It may take a while for the system under $\text{LAPS}_{\langle\beta,s\rangle}$ to reach a “steady state”, but when it does, let \widehat{n} denote the number of jobs alive.

We will show that a key resource for getting and keeping n_t high is the parallelizable work, X_t , completed by Opt but not by $\text{LAPS}_{\langle\beta,s\rangle}$ by time t . The parallelizable work is released and completed by Opt at a rate of $\Gamma(p) = 1$. $\text{LAPS}_{\langle\beta,s\rangle}$ allocates $\frac{sp}{\beta\widehat{n}}$ of its sp processors to the $\beta\widehat{n}$ jobs with the latest arrival times, $\beta\widehat{n} - \ell$ of them being parallelizable. Hence it completes parallelizable work at a rate of $s\frac{\beta\widehat{n} - \ell}{\beta\widehat{n}}$. We say that the system reaches a *steady state* when the amount of work in X_t remains constant, giving $s\frac{\beta\widehat{n} - \ell}{\beta\widehat{n}} = 1$ or $\widehat{n} = \frac{s}{(s-1)\beta}\ell$. In the steady state, Opt's cost per unit time is $\ell + 1$, and $\text{LAPS}_{\langle\beta,s\rangle}$'s cost per unit time is \widehat{n} . This gives a competitive ratio of $\frac{s}{(s-1)\beta} = \frac{1 + \beta + \epsilon}{(\beta + \epsilon)\beta}$.

For this lower bound, it is sufficient to observe that if fewer than \widehat{n} jobs are alive under $\text{LAPS}_{\langle\beta,s\rangle}$, then its the amount X_t that it is behind in the parallelizable work will continue to increase. Because each parallelizable job has unit work, the number \widehat{n} of uncompleted jobs continues to increase. Once in the steady state, it can stay there for a long enough time so that these costs dominate the cost of reaching the steady state. ■

This is within a factor of four with our upper bound when $\beta \ll \epsilon$. In particular, it shows that the ratio is $\omega(1)$ when $\beta = o(1)$. This is why the algorithm $\text{SETF}_s \approx \text{LAPS}_{\langle 0,s \rangle}$, which allocates processors to only

one job, does not perform well when there are sequential jobs.

THEOREM 4.2. *There is a set of only parallelizable jobs on which $\text{LAPS}_{\langle\beta,s\rangle}$ has a competitive ratio of $\Omega(n^\epsilon)$ with $s = \frac{2}{2-\beta} - \beta\epsilon$ or $\Omega(n^{1-\epsilon})$ with $s = 1 + \beta\epsilon$.*

Proof. Here we define the *MPT* instance, which was introduced in [3, 7, 10] as a lower bound instance for *Equi* and for *Equi*_{2+ ϵ} . We modify this instance to lower bound the performance of $\text{LAPS}_{\langle\beta,s\rangle}$. In this instance the number of alive jobs n_t briefly peeks way above the steady state number \hat{n} . This is possible even though the total parallelizable work X_t in the system is decreasing rapidly, because the work remaining per job decreases even faster. The *MPT* instance contains a stream of $n-\ell$ parallelizable jobs. As done there, there could also be many newly arriving sequential jobs in order to distract $\text{LAPS}_{\langle\beta,s\rangle}$ away from the parallelizable work, however, the result is just as good, when we assume that there are ℓ extra parallelizable jobs that arrive at time zero. The i^{th} stream job J_i has release time $r_i = \sum_{j=1}^{i-1} w_j$ and work w_i , where the values w_i will be carefully defined later. The first $(1-\beta)\ell$ of the ℓ extra jobs will never be executed by $\text{LAPS}_{\langle\beta,s\rangle}$ and hence can have zero work. The remaining $\beta\ell$ extra jobs will be identical to the first stream job J_1 , with $r_1 = 0$ and work w_1 . On this job set, *Opt* completes the zero work in the first $(1-\beta)\ell$ extra jobs in zero time, ignores the remaining $\beta\ell$ extra jobs, and uses all p processors to complete the parallelizable stream in place, giving a flow time of $\sum_{i=1}^{n-\ell} (1+\beta\ell)w_i$. In contrast $\text{LAPS}_{\langle\beta,s\rangle}$ manages to complete none of the jobs, giving a flow time of $\sum_{i=1}^{n-\ell} (i+\ell)w_i$.

[3, 7, 10] sets $w_i = (\frac{\ell}{i+\ell})^q$ for some q . Then $n_i = i+\ell$ is simply the number of jobs alive after job J_i arrives and the ℓ^q simply scales all the w_i so that the first job has $w_1 = 1$. Hence, it is equivalent, yet simpler, to set $w_i = (\frac{1}{n_i})^q$. We will now show that $\text{LAPS}_{\langle\beta,s\rangle}$ is not competitive when $q = 2 - \epsilon$.

$$\begin{aligned} \frac{F(\text{LAPS}_{\langle\beta,s\rangle}(J))}{F(\text{Opt}(J))} &= \frac{\sum_{i=1}^{n-\ell} (i+\ell)w_i}{\sum_{i=1}^{n-\ell} (1+\beta\ell)w_i} \\ &\approx \frac{\sum_{n_i=\ell}^n n_i \left(\frac{1}{n_i}\right)^q}{\sum_{n_i=\ell}^n \beta\ell \left(\frac{1}{n_i}\right)^q} \\ &= \frac{\frac{1}{-q+2} [n_i^{-q+2}]_{n_i=\ell}^n}{\frac{\beta\ell}{-q+1} [n_i^{-q+1}]_{n_i=\ell}^n} \end{aligned}$$

The magic of setting $q = 2 - \epsilon$ is that $n_i^{-q+2} = n_i^\epsilon$ increases with n_i , while $n_i^{-q+1} = n_i^{-1+\epsilon}$ decreases. Hence, the nominator is dominated by the $n_i = n$ term while the denominator is dominated by the $n_i = \ell$ term.

This gives

$$\begin{aligned} \frac{F(\text{LAPS}_{\langle\beta,s\rangle}(J))}{F(\text{Opt}(J))} &\approx \frac{\frac{1}{-q+2} n^{-q+2}}{\frac{\beta\ell}{q-1} \ell^{-q+1}} \\ &= \frac{\frac{1}{\epsilon} n^\epsilon}{\frac{\beta\ell}{1-\epsilon} \ell^{-1+\epsilon}} \\ &= \Omega(n^\epsilon) \end{aligned}$$

To make sure that none of the jobs complete under $\text{LAPS}_{\langle\beta,s\rangle}$, we will see that it suffices that the parameter q is set so that $s = \frac{\beta q}{1-(1-\beta)^q}$. Consider the job J_i . We show that the total work computed on J_i is at most w_i . The first step is to show that J_i is executed by $\text{LAPS}_{\langle\beta,s\rangle}$ during the time period $[r_k, r_{k+1}]$ of length w_k , for every k such that $k \geq i$ and $n_k < \frac{n_i}{1-\beta}$. Unless it is one of the first $(1-\beta)\ell$ of the ℓ extra jobs, J_i will get executed when it first arrives. The number n_k of jobs alive at any given moment in the future continues to increase as more stream jobs arrive. There are $n_i = i+\ell$ jobs that did not arrive after J_i . Hence, $n_k - n_i$ is the number of jobs to whom $\text{LAPS}_{\langle\beta,s\rangle}$ gives preference over J_i . Finally, βn_k is the number of jobs that $\text{LAPS}_{\langle\beta,s\rangle}$ runs. It follows that J_i will continue to be executed until $n_k - n_i \geq \beta n_k$ or equivalently $n_k \geq \frac{n_i}{1-\beta}$. During such a time period $[r_k, r_{k+1}]$ of length w_k , there are $n_k = k+\ell$ jobs alive. Hence, J_i is allocated $\frac{sp}{\beta n_k}$ processors. We can then compute the total work completed on J_i to be

$$\begin{aligned} \sum_{k=i}^{\frac{n_i}{1-\beta}-\ell} \frac{s}{\beta n_k} w_k &= \sum_{n_k=n_i}^{\frac{n_i}{1-\beta}} \frac{s}{\beta n_k} \left(\frac{1}{n_k}\right)^q \\ &= \frac{s}{\beta(-q)} \left[\left(\frac{1}{n_k}\right)^q \right]_{n_k=n_i}^{\frac{n_i}{1-\beta}} \\ &= \frac{s}{\beta q} \left[\left(\frac{1}{n_i}\right)^q - \left(\frac{1-\beta}{n_i}\right)^q \right] \\ &= \frac{s}{\beta q} [1 - (1-\beta)^q] \cdot w_i \end{aligned}$$

This is why it suffices that the parameter q is set so that $s = \frac{\beta q}{1-(1-\beta)^q}$ to insure that the total work computed on J_i is at most w_i .

What remains is to understand this odd relation $s(q) = \frac{\beta q}{1-(1-\beta)^q}$. We saw that the pivotal value of q at which $\text{LAPS}_{\langle\beta,s\rangle}$ is no longer competitive occurs when $q = 2$. This gives that the pivotal speed is $s(2) = \frac{2\beta}{1-(1-\beta)^2} = \frac{2\beta}{2\beta-\beta^2} = \frac{2}{2-\beta}$. In order to decrease q from 2 to $2 - \epsilon$, s needs to decrease from $s(2)$ to $s(2 - \epsilon) \approx s(2) - \frac{ds(2)}{dq}\epsilon$. Maple was kind enough to give that $\frac{ds(2)}{dq} \leq \beta$ for $\beta \in [0, 1]$. Hence, with extra resources $s = \frac{2}{2-\beta} - \beta\epsilon$, $\text{LAPS}_{\langle\beta,s\rangle}$ is not competitive.

By changing ϵ to $1 - \epsilon$, one gets that the competitive ratio of $\Omega(n^{1-\epsilon})$ by setting $q = 1 + \epsilon$. $s(1) = 1$ and $\frac{ds(1)}{dq} \approx \beta$ for $\beta \in [0, 1]$. Hence, with extra resources $s = 1 + \beta\epsilon$, the competitive ratio is $\Omega(n^{1-\epsilon})$. ■

For example, with $\beta = 1$, $\text{LAPS}_{\langle\beta,s\rangle} = \text{Equi}_s$ and the result corresponds to that in [3] that competitive ratio is $\Omega(n^\epsilon)$ with $s = 2 - \epsilon$ and $\Omega(n^{1-\epsilon})$ with $s = 1 + \epsilon$. When β is an infinitesimal, the pivotal speed is $s = 1 + \frac{1}{2}\beta$, which is fairly close to the upper bound that requires speed of more than $s = 1 + \beta$ in order to be competitive.

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

References

- [1] 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.
- [2] Piotr Berman and Chris Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, 6(2):181–193, 1999.
- [3] Jeff Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235:109–141, 2000.
- [4] Jeff Edmonds. On the competitiveness of aimd-tcp within a general network. In *LATIN*, pages 567–576, 2004.
- [5] Jeff Edmonds, Suprakash Datta, and Patrick Dymond. Tcp is competitive against a limited adversary. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 174–183, 2003.
- [6] Jeff Edmonds and Kirk Pruhs. Multicast pull scheduling: When fairness is fine. *Algorithmica*, 36(3):315–330, 2003.
- [7] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.
- [8] Bala Kalyanasundaram and Kirk Pruhs. Minimizing flow time nonclairvoyantly. *J. ACM*, 50(4):551–567, 2003.
- [9] Rick Merritt. Cpu designers debate multi-core future. *EE Times*, June 2008.
- [10] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130:17–47, 1994.
- [11] Cynthia Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, pages 163–200, 2002.
- [12] Kirk Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Performance Evaluation Review*, 34(4):52–58, 2007.
- [13] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling. In *Handbook on Scheduling*. CRC Press, 2004.
- [14] Julien Robert and Nicolas Schabanel. Non-clairvoyant batch sets scheduling: Fairness is fair enough. In *European Symposium on Algorithms*, pages 741–753, 2007.
- [15] 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.
- [16] Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *Symposium on Discrete Algorithms*, pages 491–500, 2008.