Speed Scaling of Processes with Arbitrary Speedup Curves on a Multiprocessor

Ho-Leung Chan[*] Jeff Edmonds[†] Kirk Pruhs[‡]

Abstract

We consider the setting of a multiprocessor where the speeds of the m processors can be individually scaled. Jobs arrive over time and have varying degrees of parallelizability. A nonclairvoyant scheduler must assign the processes to processors, and scale the speeds of the processors. We consider the objective of energy plus flow time. We assume that a processor running at speed s uses power s^α for some constant $\alpha > 1$. For processes that may have side effects or that are not checkpointable, we show an $\Omega(m^{(\alpha-1)/\alpha^2})$ bound on the competitive ratio of any randomized algorithm. For checkpointable processes without side effects, we give an $O(\log m)$-competitive algorithm. Thus for processes that may have side effects or that are not checkpointable, the achievable competitive ratio grows quickly with the number of processors, but for checkpointable processes without side effects, the achievable competitive ratio grows slowly with the number of processors. We then show a lower bound of $\Omega(\log^{1/\alpha} m)$ on the competitive ratio of any randomized algorithm for checkpointable processes without side effects.

1 Introduction

Due to the power related issues of energy and temperature, major chip manufacturers, such as Intel, AMD and IBM, now produce chips with multiple cores/processors, and with dynamically scalable speeds, and produce associated software, such as Intel's SpeedStep and AMD's PowerNow, that enables an operating system to manage power by scaling processor speed. Currently most multiprocessor chips have only a handful of processors, but chip designers are agreed upon the fact that chips with hundreds to thousands of processors will dominate the market in the next decade. The founder of chip maker Tilera asserted that a corollary to Moore's law will be that the number of cores/processors will double every 18 months [13].

According to the well known cube-root rule, a CMOS-based processor running at speed s will have a dynamic power P of approximately s^3. In the algorithmic literature, this is

[*]The University of Hong Kong. hlchan@cs.hku.hk.

[†]York University. jeff@cs.yorku.ca. Supported in part by NSERC Canada.

[‡]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.

usually generalized to $P = s^\alpha$. Thus in principle, p processors running at speed s/p could do the work of one processor running at speed s but at $1/p^{\alpha-1}$ of the power. But in spite of this, chip makers waited until the power costs became prohibitive before switching to multiprocessor chips because of the technical difficulties in getting p speed s/p processors to come close to doing the work of one speed s processor. This is particularly true when one has many processors, and few processes, where these processes have widely varying degrees of parallelizability. That is, some processes may be considerably sped up when simultaneously run on multiple processors, while some processes 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 of parallelizability used in [8, 9, 18, 19]. Each process 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 process. The speedup functions may be arbitrary, other than we assume that they are nondecreasing (a process doesn't run slower if it is given more processors), and sublinear (a process satisfies Brent's theorem, that is, increasing the number of processors doesn't increase the efficiency of computation).

The operating system needs a *process assignment* policy for determining at each time, which processors (if any) a particular process is assigned to. We assume that a process may be assigned to multiple processors. In tandem with this, the operating system will also need a *speed scaling* policy for setting the speed of each processor. In order to be implementable in a real system, the speed scaling and process assignment policies must be online since the system will not in general know about processes arriving in the future. Further, to be implementable in a generic operating system, these policies must be nonclairvoyant, since in general the operating system does not know the size/work of each process when the process is released to the operating system, nor the degree to which that process is parallelizable. So a nonclairvoyant algorithm only knows when processes have been released and finished in the past, and which processes have been run on each processor at each time in the past.

The operating system has competing dual objectives, as it both wants to optimize some schedule quality of service objective, as well as some power related objective. In this paper, we will consider the formal objective of minimizing a linear combination of total response/flow time (the schedule objective) and total energy used (the power objective). (In the conclusion, we will discuss the relationship between this energy objective and a temperature objective). This objective of flow plus energy has a natural interpretation. Suppose that the user specifies how much improvement in flow, call this amount ρ, is necessary to justify spending one unit of energy. For example, the user might specify that he is willing to spend 1 erg of energy from the battery for a decrease of 6 micro-seconds in flow. Then the optimal schedule, from this user's perspective, is the schedule that optimizes $\rho = 6$ times the energy used plus the total flow. By changing the units of either energy or time, one may assume without loss of generality that $\rho = 1$.

So the problem we want to address here is how to design a nonclairvoyant process assignment policy and a speed scaling policy that will be competitive for the objective of flow plus energy.

The case of a single processor was considered in [7]. In the single processor case, the parallelizability of the processes is not an issue. If all the processes arrive at time 0, then in the optimal schedule, the power at time t is $\Theta(n_t)$, where n_t is the number of active processes at time t. The algorithm considered in [7] runs at a speed of $(1+\delta)n_t^{1/\alpha}$ for some constant $\delta \geq 0$. The process assignment algorithm considered in [7] is Latest Arrival Processor Sharing (LAPS). LAPS was proposed in [9] in the context of running processes with arbitrary speedup functions on fixed speed processors, and it was shown to be scalable, i.e., $(1+\epsilon)$-speed $O(1)$-competitive, for the objective of total flow time in this setting. LAPS is parameterized by a constant $\beta \in (0,1]$, and shares the processing power evenly among the $\lceil \beta n_t \rceil$ most recently arriving processes. Note that the speed scaling policy and LAPS are both nonclairvoyant. [7] showed that, by picking δ and β appropriately, the resulting algorithm is $4\alpha^3(1+(1+\frac{3}{\alpha})^\alpha)$-competitive for the objective of flow plus energy on a single speed scalable processor.

1.1 Our Results

Here we consider extending the results in [7] to the setting of a multiprocessor with m processors. It is straight-forward to note that if all of the work is parallelizable, then the multiprocessor setting is essentially equivalent to the uniprocessor setting. To gain some intuition of the difficulty that varying speedup functions pose, let us first consider an instance of one process that may either be sequential or parallelizable. If an algorithm runs this process on few of the processors, then the algorithm's competitive ratio will be bad if the process is parallelizable, and the optimal schedule runs the process on all of the processors. Note that if the algorithm wanted to be competitive on flow time, it would have to run too fast to be competitive on energy. If an algorithm runs this process on many of the processors, then the algorithm's competitive ratio will be bad if the process is sequential, and the optimal schedule runs the process on few processors. If the algorithm wanted to be competitive on energy, it would have to run too slow to be competitive on flow time. Formalizing this argument, we show in Section 2 a lower bound of $\Omega(m^{(\alpha-1)/\alpha^2})$ on the competitive ratio of any randomized nonclairvoyant algorithm against an oblivious adversary (with an additional assumption that we will now discuss).

At first glance, such a strong lower bound for such an easy instance might lead one to conclude that there is no way that the scheduler can be expected to guarantee reasonably competitive schedules. But on further reflection, one realizes that an underlying assumption in this lower bound is that only one copy of a process can be run. If a process does not have side effects, that is if the process doesn't change/effect anything external to itself, then this assumption is not generally valid. One could run multiple copies of a process simultaneously, with each copy being run on a different number of processors, and halt computation when the first copy finishes. For example, in the instance in the previous paragraph, one could be $O(1)$-competitive if the process didn't have side effects by running one copy on a single processor, and running one copy on the rest of the processors. Generalizing this approach, one can obtain a $O(\log m)$-competitive algorithm for instances consisting of processes that have no side effects, and where the speed-up function doesn't change. Unfortunately, we show in Section 2 that such a result can not be obtained if processes can have multiple phases with different speed-up functions. We accomplish this by showing that the competitive ratio of any randomized nonclairvoyant algorithm, that runs multiple independent copies of a process, against an oblivious adversary, is $\Omega(m^{\Omega(1/\alpha)})$.

Contemplating this second lower bound, it suggests that to be reasonably competitive, the algorithm must be able to process work on all copies of a job at the maximum rate of work processing on any copy. If a processes had small state, so that the overhead of checkpointing isn't prohibitive, one might reasonably approximate this by checkpointing (saving the state of) each copy periodically, and then restarting each copy from the point of execution of the copy that made the most progress. In Section 3, we formalize this intuition. We give a process assignment algorithm MultiLAPS, which is a modification of LAPS. We show that, by combining MultiLAPS with the natural speed scaling algorithm, one obtains an $O(\log m)$ competitive algorithm if all copies process work at the rate of the fastest copy. There are two steps in the analysis of MultiLAPS. The first step is to show that there is a worst-case instance where every speedup function is parallel up to some number of processors, and then is constant. This shows that the worst-case speedup functions for speed scalable processors are more varied than for fixed speed processors, where it is sufficient to restrict attention to only parallelizable and sequential speedup functions [8, 9, 18, 19]. The second step in the analysis of MultiLAPS is a reduction to essentially the analysis of LAPS in a uniprocessor setting. Technically, we need to analyze LAPS when some work is sequential, that is, it has the special property that it is processed at unit rate independent of the speed of the processor. We then discuss how to generalize the analysis of LAPS in [7] to allow sequential work, using techniques from [8, 9].

In Section 4 we then show a lower bound of $\Omega(\log^{1/\alpha} m)$ on the competitive ratio of any nonclairvoyant randomized algorithm against an oblivious adversary, for checkpointable processes without side effects.

Thus in summary, for processes that may have side effects, or that are not checkpointable, the achievable competitive ratio grows quickly with the number of processors. But for checkpointable processes without side effects, the achievable competitive ratio grows slowly with the number of processors. This shows the importance of being able to efficiently checkpoint multiple copies of a process, in a setting of processes with varying degrees of parallelizability and individually speed scalable multiprocessors.

1.2 Related results

We start with some results in the literature about scheduling with the objective of total flow time on a single fixed speed processor. It is well known that the online clairvoyant algorithm Shortest Remaining Processing Time (SRPT) is optimal. The competitive ratio of any deterministic nonclairvoyant algorithm is $\Omega(n^{1/3})$, and the competitive ratio of every randomized algorithm against an oblivious adversary is $\Omega(\log n)$ [14]. A randomized version of the Multi-Level Feedback Queue algorithm is $O(\log n)$-competitive [5, 11]. The nonclairvoyant algorithm Shortest Elapsed Time First (SETF) is scalable, that is, it is $(1+\epsilon)$-speed $O(1)$-competitive for any arbitrarily small but fixed $\epsilon > 0$. [10] SETF shares the processor equally among all processes that have been run the least.

We now consider scheduling processes with arbitrary speedup functions on fixed speed processors for the objective of total flow time. The algorithm Round Robin RR (also called Equipartition and Processor Sharing) that shares the processors equally among all processes is $(2+\epsilon)$-speed $O(1)$-competitive [8]. As mentioned before, LAPS is scalable [9].

We now consider speed scaling algorithms on a single processor for the objective of flow plus energy. [1, 17] give efficient offline algorithms. We now describe the results for online clairvoyant algorithms. This setting was studied in a sequence of papers [1–4, 12], which culminated in the following result [3]. The scheduling algorithm, that uses Shortest Remaining Processing Time (SRPT) for process assignment and power equal to one more than the number of active processes for speed scaling, is $(3+\epsilon)$-competitive for the objective of total flow plus energy on arbitrary-work unit-weight processes, even if the power function is arbitrary. So clairvoyant algorithms can be $O(1)$-competitive independent of the power function. [7] showed that nonclairvoyant algorithms can not be $O(1)$-competitive if the power function is growing too quickly. The case of weighted flow time has also been studied. The scheduling algorithm, that uses Highest Density First (HDF) for process assignment and power equal to the fractional weight of the active processes for speed scaling, is $(2+\epsilon)$-competitive for the objective of fractional weighted flow plus energy on arbitrary-work arbitrary-weight processes. An $O(1)$-competitive algorithm for weighted flow plus energy can then be obtained using the known resource augmentation analysis of HDF [6]. [2, 12] extend some of the results for the unbounded speed model to a model where there is an upper bound on the speed of a processor.

There are many related scheduling problems with other objectives, and/or other assumptions about the processors and instance. Surveys can be found in [15, 16].

1.3 Formal Problem Definition and Notations

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. $\Gamma_i^q(p)$ represents the rate at which work is processed for phase q of job i when one copy of the job is 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 copy of a job, (1) a nonnegative real number specifying the number of processors assigned to the copy of the job, and (2) a nonnegative real speed. We thus assume that if several processors are working on the same instance/copy of a job, then they must all run at the same speed. But different copies can run at different speeds. 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.

In this paper we consider several different models depending on how the processing on different copies interact. Assume multiple copies of job i are run, with the speed and number of processors assigned to the k-th copy being s_k and p_k. In the *independent processing model* if copy k is running in a phase with speedup function Γ, then work is processed on this copy at rate $s_k\Gamma(p_k)$ (independent of the rate of processing on the other copies). In the *maximum* *processing model*, if each copy of job J_i is in a phase with speedup function Γ, then each copy processes work at a rate of $\max_k s_k \Gamma(p_k)$. Thus in the maximum processing model each copy of each job is always at the same point in its execution.

The *completion time* of a job J_i, denoted C_i, is the first point of time when all the work on some copy of the job 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 *active* 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 $\int_0^\infty n_t dt$.

When running at speed s, a processor consumes $P(s) = s^\alpha$ units of energy per unit time, where $\alpha > 1$ is some fixed constant. We call $P(s)$ the *power function*.

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 of a job is *parallel up to q processors* if $\Gamma(p) = p$ for $p \leq q$, and $\Gamma(p) = q$ for $p > q$. 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. We further assume that all speedup functions satisfy $\Gamma(p) = p$ for $p \leq 1$. This natural assumption means that when a job J_i is assigned to a single processor, and shares this processor with other jobs, the rate that J_i is processed is the fraction of the processor that J_i receives times the speed of the processor.

Let A be an algorithm and J an instance. We denote the schedule output by A on J as $A(J)$. We let $F_A(J)$ and $E_A(J)$ denote the total flow time and energy incurred in $A(J)$, let $cost_A(J) = F_A(J) + E_A(J)$ denote the cost. We will use M as a short-hand for MultiLAPS. Let Opt be the optimal algorithm that always minimizes total flow time plus energy. A randomized algorithm A is c-competitive, or has competitive ratio c, if for all instances J, $E[cost_A(J)] \leq c \cdot cost_{\text{Opt}}(J)$.

2 Lower Bounds for Single Copy and Non-Checkpointing Algorithms

In this section we show that the competitive ratio must grow quickly with the number of processors if only one copy of each job can be running (Lemma 3), or if multiple copies are allowed, but no checkpointing is allowed (Lemma 4). We first start with a couple basic lemmas about optimal schedules that will be useful throughout the paper.

Lemma 1. *Consider a job with work w and with a single phase with a speedup function that is parallelizable up to q processors. Assume that the job is run on $p \leq q$ processors, then the optimal speed is $\frac{1}{((\alpha-1)p)^{1/\alpha}}$, for a cost of $\Theta(\frac{w}{p^{1-1/\alpha}})$. Assume that the job is run on $p \geq q$ processors, then the optimal speed is $\frac{1}{((\alpha-1)p)^{1/\alpha}}$, for a cost of $\Theta(\frac{wp^{1/\alpha}}{q})$.*

Proof. First consider that case that $p \leq q$. Let s be the speed of the processors. The flow plus energy is then $\frac{w}{ps} + ps^\alpha \frac{w}{ps} = w(\frac{1}{ps} + s^{\alpha-1})$. This is minimized by setting $s = \frac{1}{((\alpha-1)p)^{1/\alpha}}$, for a cost of $\Theta\left(\frac{w}{p^{1-1/\alpha}}\right)$.

Now consider the case that $p \geq q$. Let s be the speed of the processors. The flow plus energy is then $\frac{w}{qs} + ps^{\alpha}\frac{w}{qs}$. This is minimized by setting $s = \frac{1}{((\alpha-1)p)^{1/\alpha}}$, giving a cost of $\Theta(wp^{1/\alpha}/q)$. □

Lemma 2. *Consider a job with work w with a single phase with a speedup function that is parallelizable up to q processors. The optimal schedule uses $p = q$ processors, run at speed $\frac{1}{((\alpha-1)q)^{1/\alpha}}$, for a cost of $\Theta(\frac{w}{q^{1-1/\alpha}})$.*

Proof. From the proof of Lemma 1 we know that if the algorithm allocates $p \leq q$ speed s processors to this job, the cost is minimized by when $s = \frac{1}{((\alpha-1)p)^{1/\alpha}}$, for a cost of $\Theta\left(\frac{w}{p^{1-1/\alpha}}\right)$. This is minimized by making p as big as possible, namely $p = q$. From the proof of Lemma 1 we know that if the algorithm allocates $p \geq q$ speed s processors to this job, the cost is minimized when $s = \frac{1}{((\alpha-1)p)^{1/\alpha}}$, giving a cost of $\Theta(wp^{1/\alpha}/q)$. This is minimized by making p as small as possible, namely $p = q$. Thus in either case, the optimal scheduling policy uses $p = q$ processors, run at speed $\frac{1}{((\alpha-1)q)^{1/\alpha}}$, for a cost of $\Theta(\frac{w}{q^{1-1/\alpha}})$. □

Lemma 3. *Any randomized nonclairvoyant algorithm, that only runs one copy of each job, must be $\Omega(m^{(\alpha-1)/\alpha^2})$-competitive against an oblivious adversary that must specify the input a priori.*

Proof. Applying Yao's technique, we give a probability distribution over the input, and show that every deterministic algorithm A will have an expected competitive ratio of $\Omega(m^{(\alpha-1)/\alpha^2})$. The instance will be selected uniformly at random from two possibilities. The first possible instance consists of one job with $m^{1-1/\alpha}$ units of parallelizable work. The second possible instance will consist of job with one unit of work that is parallel up to one processor. By plugging these parameters into Lemma 2, one can see that the optimal cost is $\Theta(1)$ for both instances.

Let p denote the number of processors used by the algorithm A. By Lemma 1, the cost for the algorithm A is either $\Omega((\frac{m}{p})^{1-1/\alpha})$ or $\Omega(p^{1/\alpha})$ depending on the instance. Both the maximum and the average of these two costs is minimized by balancing these two costs, which is accomplished by setting $p = m^{1-1/\alpha}$. This shows that the competitive ratio is $\Omega(m^{(\alpha-1)/\alpha^2})$. □

Lemma 4. *In the independent processing model, any randomized nonclairvoyant algorithm must be $\Omega(m^{(\alpha-1)/\alpha^2})$-competitive against an oblivious adversary that must specify the input a priori.*

Proof. Applying Yao's technique, we give a probability distribution over the inputs with the property that every deterministic algorithm A will have expected competitive ratio $\Omega(m^{(\alpha-1)/\alpha^2})$. The random instance consists of a single job with many phases. Each phase will be randomly chosen to be one of the two job instances given in Lemma 3, that is, each phase will either be parallelizable or parallel up to one processor, and the optimal cost for each phase will be $\Theta(1)$.

Consider a particular copy of the job run by A. Because each phase is so small, we can assume that the algorithm A allocates a fixed number of processors p running at a fixed speed

s for the duration of the phase. By the proof of Lemma 3, no matter what the algorithm does, the probability is at least a half that it incurs a cost of $\Omega(m^{(\alpha-1)/\alpha^2})$ during this phase. One can think of the phases as Bernoulli trials with outcomes being cost $\Omega(m^{(\alpha-1)/\alpha^2})$ with probability at least a half, and smaller cost with probability at most a half. Applying a Chernoff bound, with high probability the algorithm has cost $\Omega(m^{(\alpha-1)/\alpha^2})$ on nearly half of the stages. While the optimal cost on each stage is 1.

By a union bound, the probability that any copy has average cost per phase much less than $\Omega(m^{(\alpha-1)/\alpha^2})$ is small. □

3 Analysis of MultiLAPS

In this section, we assume that multiple copies of a job may be run simultaneously. Each copy of a job may be assigned a different number of processors, but each processor running this copy must be run at the same speed. We assume that at each moment in time, the rate that work is processed on each copy of a job is the maximum of the rates of the different copies (so all copies of a job are always at the same point of execution). We give a nonclairvoyant algorithm MultiLAPS for this setting, and show that it is $O(\log m)$-competitive for flow time plus energy.

We now describe the algorithm LAPS from [9], and the algorithm MultiLAPS that we introduce here. We then give some underlying motivation for the design of MultiLAPS.

Algorithm LAPS: Let $\delta \geq 0$ and $\beta > 0$ be real constants. At any time t, the processor speed is $(1+\delta)(n_a)^{1/\alpha}$, where n_a is the number of active jobs at time t. The processor processes the $\lceil \beta n_a \rceil$ active jobs with the latest release times by splitting the processing equally among these jobs. For our purposes in this paper, we will take $\delta = 0$. □ The

intuition behind the definition of LAPS builds on the fact that the algorithm SETF, which prioritizes jobs that have been processed the least (which are generally the most recently arrived jobs) is scalable if all the work is fully parallelizable [10]. LAPS favors most recently arrived jobs, but spreads the processing out over a linear number out of caution that the most recently arriving jobs might be in sequential phases.

Algorithm MultiLAPS: Let $0 < \beta < 1$ be a real number that parametrizes MultiLAPS. Let $\mu = 1/3$. Consider any time t. Let n_a be the number of active jobs at t. Each of the $\lceil \beta n_a \rceil$ active jobs with the latest release times will be run at this point in time. Call these jobs the *late jobs*. For each late job J_i, a *primary copy* of J_i is run on a group of $p_a = \mu \frac{m}{\lceil \beta n_a \rceil}$ processors, where each processor in this group is run at speed $s_a = \frac{1}{\mu}(\frac{n_a}{m})^{1/\alpha}$. Note that the primary copies of the late jobs are equally sharing a μ fraction of the processors. Furthermore for each late job J_i, there are $\lfloor \log p_a \rfloor$ *secondary copies* of J_i run. The j^{th}, $j \in [1, \lfloor \log p_a \rfloor]$, secondary copy of J_i is run on a group of 2^i processors, where each processor in this group is run at speed $2(\frac{1}{2^i})^{1/\alpha}$. □

Intuition behind the design of MultiLAPS: Let us give a bit of intuition behind the design of MultiLAPS. If μ was 1, and no secondary copies were run, then MultiLAPS would essentially be adopting the strategy of LAPS of sharing the processing power evenly among the latest arriving β fraction of the jobs. LAPS is $O(1)$-competitive when all work is parallelizable up to the number of available processors [7]. However, if a primary copy of a job is run on many processors, the online algorithm may be wasting a lot of energy if this work is not highly parallelizable. To account for this possibility, MultiLAPS runs the primary copy a little faster, freeing up some processors to run secondary copies of the job on fewer processors and at a faster speed. The number of processors running the secondary copies are geometrically decreasing by a factor of 2, while the speeds are increasing by a factor of $2^{1/\alpha}$. Thus each copy of a job is using approximately the same power. Intuitively, one of the copies is running the late job on the "right" number of processors. Thus MultiLAPS uses a factor of $O(\log m)$ more energy than optimal because of the $\log m$ different equi-power copies of the job. Setting $\mu \leq \frac{1}{3}$ guarantees that that MultiLAPS doesn't use more than m processors. $\square$

The rest of this section is devoted to proving the following theorem.

Theorem 5. *In the maximum processing model,* MultiLAPS *is $O(\log m)$-competitive for total flow time plus energy.*

Overview of the proof of Theorem 5: We now give an overview of the structure of our proof of Theorem 5. In Lemma 6 we show how to reduce the analysis of MultiLAPS on arbitrary instances to the analysis of MultiLAPS on canonical instances. We define an instance to be *canonical* if the speedup function for each job phase is parallel up to the number p_o of processors that Opt uses on that phase (and is constant there after). The value of p_o may be different for each phase. More specifically, we show how to construct a canonical instance J from an arbitrary instance K such that the cost of MultiLAPS on K is identical to the cost of MultiLAPS on J, and the optimal cost for J is at most the optimal cost for K.

We then define a variation of the uniprocessor setting, that we call the Sequential Setting. In the Sequential Setting, a job can have sequential phases, which are phases that are processed at a unit rate independent of the computational resources assigned to the job.

We then show how to reduce the analysis of MultiLAPS on canonical instances to the analysis of LAPS in the Sequential Setting. More precisely, from an arbitrary canonical instance J, we show how to create an instance J' for the Sequential Setting. We show in Lemma 7 that the flow time for MultiLAPS on J is identical to the flow time of LAPS on J', and the energy used by MultiLAPS on J is at most $O(\log m)$ times the energy used by LAPS on J'.

We then need to relate the optimal schedule for J to the optimal schedule for J'. To accomplish this we classify each phase of a job in J as either saturated or unsaturated depending on the relationship between the speedup function and how many processors MultiLAPS uses for this phase. We consider two instances derived from J, an instance J_{sat} consisting of only the saturated phases in J, and an instance J_{uns} consisting of only the unsaturated phases in J. We then consider two instances derived from the instance J', an instance J'_{par} consisting of parallel phases in J', and an instance J'_{seq} consisting of sequential phases in J'. The transformation of J to J' transforms phases in J_{sat} to phases in J'_{par} and transforms phases in J_{uns} to phases in J'_{seq}. It will be clear that the optimal cost for J is at least the optimal cost for J_{sat} plus the optimal cost for J_{uns}. We then show in Lemma 8 that the optimal cost for J_{sat} is at least the optimal cost for J'_{par}, and in Lemma 9 that the optimal cost for J_{uns} is at least the optimal cost for J'_{seq}. We then discuss how to generalize the analysis of LAPS in [7], using techniques from [8, 9], to show that the cost of LAPS is at most a constant factor larger than the optimal cost for J'_{par} plus the optimal cost for J'_{seq}.

This line of reasoning allows us to prove our Theorem 5 as follows:

$$\begin{aligned}
cost_M(K) &= cost_M(J) \\
&= O(\log m) \cdot cost_{\text{LAPS}}(J') \\
&= O(\log m) \cdot (cost_{\text{Opt}}(J'_{par}) + cost_{\text{Opt}}(J'_{seq})) \\
&= O(\log m) \cdot (cost_{\text{Opt}}(J_{sat}) + cost_{\text{Opt}}(J_{uns})) \\
&= O(\log m) \cdot cost_{\text{Opt}}(J) \\
&= O(\log m) \cdot cost_{\text{Opt}}(K)
\end{aligned}$$

The first and final equalities follow from Lemma 6. The second equality follows from Lemma 7. The third equality follows from the analysis of LAPS in the sequential setting. The fourth equality follows from Lemma 8 and Lemma 9. The fifth equality will be an obvious consequence of the definitions of J_{sat} and J_{uns}. □

We now execute the proof strategy that we have just outlined. We first show that there is a worst-case instance for MultiLAPS that is canonical.

Lemma 6. *Let K be any input instance. There is a canonical instance J such that $F_M(J) = F_M(K)$, $E_M(J) = E_M(K)$, and $cost_{\text{Opt}}(J) \leq cost_{\text{Opt}}(K)$.*

Proof. We construct J by modifying each job in K as follows. Consider an infinitesimally small phase of a job in K with work w and speedup function Γ. Let p_o be the number of processors that Opt allocates to this phase when scheduling K.

We modify this phase so that the new speedup function is $\Gamma'(p) = \frac{p}{p_o}\Gamma(p_o)$ for $p \leq p_o$ and $\Gamma'(p) = \Gamma(p_o)$ for $p \geq p_o$.

Note that MultiLAPS may process this phase in several copies of this job. Assume that the i-th copy is processed by p_i processors of speed s_i. Due to the modification of speedup function, the rate of processing for the i-th copy changes from $\Gamma(p_i)s_i$ to $\Gamma'(p_i)s_i$. If $p_i \geq p_o$, then the rate of processing on the i-th copy does not increase since Γ is nondecreasing. Now consider a copy where $p_i < p_o$. By the definition of Γ', the rate of processing $\Gamma'(p_i)s_i = \frac{p_i\Gamma(p_o)}{p_o}s_i$. Since $p_i < p_o$ and since Γ is sublinear, $\frac{\Gamma(p_o)}{p_o} \leq \frac{\Gamma(p_i)}{p_i}$. Plugging this back in, we get that the rate of processing for copy is at most $s_i\Gamma(p_i)$. So MultiLAPS doesn't finish this phase in the modified instance before it can finish the phase in K. We then decrease the work of this phase so that the time when this phase is first completed (among all of the copies) is identical to when it completes in MultiLAPS(K). Note that by construction the schedule of MultiLAPS on this modified instance is identical to MultiLAPS(K), while Opt may get better performance due to the reduction of work.

Finally, we create J by multiplying both the work of this phase and the speedup function by the same factor of $\frac{p_o}{\Gamma'(p_o)}$ to make the final speed up function for this phase parallel up to p_o processors. This change does not effect the schedules of either MultiLAPS and Opt. □

Definition of the Sequential Setting: Everything is defined identically as in Subsection 1.3, with the following two exceptions. Firstly, there is only a single processor. Secondly, job phases can be *sequential*, which in the context of this paper means that work in this phase is processed at a rate of 1 independent of the fraction of the processor assigned to the job, and the speed of the processor. So sequential work is processed at rate 1, even if it is run at a speed much greater than 1, or is not even run at all. Sequential work doesn't correspond to any realistic situation, but is merely mathematical construct required for the proof. $\square$

Definition of the Transformation of a Canonical Instance J into the instance J' in the Sequential Setting: We transform each job in J into a job in J' by modifying each phase of the original job. At each point in time, consider a phase and the copy in MultiLAPS with the highest processing rate on this phase. Let Γ_{p_o} be the speedup function of the phase, which is parallel up to p_o processors. We say the phase is currently "saturated" if $\mu\frac{m}{\lceil\beta n_a\rceil} \le p_o$, and "unsaturated" otherwise. Note that $\mu\frac{m}{\lceil\beta n_a\rceil}$ is the number of processors assigned to the primary copy in MultiLAPS. Thus, a phase is saturated if all copies in MultiLAPS are processing in the parallel range of Γ_{p_o}, and unsaturated otherwise.

Consider the case that the phase is saturated. The copy with the highest processing rate in MultiLAPS is the one with $p_a = \mu\frac{m}{\lceil\beta n_a\rceil}$ processors of speed $s_a = \frac{1}{\mu}(\frac{n_a}{m})^{1/\alpha}$, giving a rate of $\frac{m}{\lceil\beta n_a\rceil}(\frac{n_a}{m})^{1/\alpha} = m^{1-1/\alpha}\cdot\frac{(n_a)^{1/\alpha}}{\lceil\beta n_a\rceil}$. We modify this phase to be fully parallelizable, and scale down the work by a factor of $m^{1-1/\alpha}$. Note that the processing rate of LAPS is $\frac{(n_a)^{1/\alpha}}{\lceil\beta n_a\rceil}$, so it will complete the phase using the same time.

Consider the case that the phase is unsaturated. Let r be the fastest rate that any copy in MultiLAPS is processing work in this phase. We modify this phase to be sequential, and scale down the work by a factor of r. By the definition of sequential, the processing rate of LAPS on this phase is 1, so it will complete the phase using the same time as MultiLAPS. $\square$

We now show that the cost of MultiLAPS on J is at most a log factor more than the cost of LAPS on J' in the Sequential Setting.

Lemma 7. *$cost_M(J) = F_{\text{LAPS}}(J') + O(\log m)E_{\text{LAPS}}(J')$. From this we can conclude that $cost_M(J) = O(\log m)cost_{\text{LAPS}}(J')$.*

Proof. By construction, the flow time for MultiLAPS(J) is identical to the flow time for LAPS(J'). We now show that the energy used by MultiLAPS(J) is at most a log factor more than the energy used by LAPS(J').

The power in MultiLAPS(J) is the sum of the powers in the m processors. Note that for each of the $\lceil\beta n_a\rceil$ late jobs, MultiLAPS allocates p_a processors of speed s_a to a primary copy of this job. Recall that $p_a = \mu\frac{m}{\lceil\beta n_a\rceil}$ and $s_a = \frac{1}{\mu}(\frac{n_a}{m})^{1/\alpha}$. MultiLAPS also runs $\log p_a$ secondary copies, where the i-th copy is run on 2^i processors of speed $2(\frac{1}{2^i})^{1/\alpha}$. Hence, the total power for MultiLAPS(J) is:

$$\begin{aligned}
&\lceil\beta n_a\rceil\left(\mu\frac{m}{\lceil\beta n_a\rceil}\cdot(\frac{1}{\mu}(\frac{n_a}{m})^{1/\alpha})^\alpha + \sum_{i=1}^{\lfloor\log p_a\rfloor} 2^i(2(\frac{1}{2^i})^{1/\alpha}))^\alpha\right)\\
\le\ &(\frac{1}{\mu})^{\alpha-1}n_a + 2^\alpha\lceil\beta n_a\rceil\log p_a
\end{aligned}$$

LAPS(J') runs at speed $n_a^{1/\alpha}$, and hence power n_a. Since $p_a \le m$, we conclude that $E_M(J) = O(\log m)E_{\text{LAPS}}(J')$. $\square$

We now want to show a lower bound for Opt(J). To state this lower bound, we need to introduce some notation. Define J_{sat} to be the instance obtained from J by removing all unsaturated phases in each job and directly concatenating the saturated phases. Define J_{uns} to be the instance obtained from J by removing all saturated phases, and directly concatenating the unsaturated phases. Define J'_{par} to be the instance obtained from J' by removing all sequential phases in each job and directly concatenating the parallel phases. Define J'_{seq} to be the instance obtained from J' by removing all parallel phases in each job and directly concatenating the sequential phases. Note that the transformation from J to J' transforms a phase in J_{sat} to a phase in J'_{par}, and transforms a phase in J_{uns} to a phase in J'_{seq}. Obviously, Opt can only gain by scheduling J_{sat} and J_{uns} separately. That is,

$$cost_{\mathrm{Opt}}(J) \geq cost_{\mathrm{Opt}}(J_{sat}) + cost_{\mathrm{Opt}}(J_{uns})$$

We now want to show that $cost_{\mathrm{Opt}}(J'_{par}) = O(cost_{\mathrm{Opt}}(J_{sat}))$ and $cost_{\mathrm{Opt}}(J'_{seq}) = O(cost_{\mathrm{Opt}}(J_{uns}))$.

Lemma 8. $cost_{\mathrm{Opt}}(J'_{par}) = O(cost_{\mathrm{Opt}}(J_{sat}))$.

Proof. We construct a schedule Opt(J'_{par}) from the schedule Opt(J_{sat}) phase by phase. Each phase in Opt(J'_{par}) will end no later than the corresponding phase in Opt(J_{sat}), and the schedule for Opt(J'_{par}) will use less energy than the schedule Opt(J_{sat}).

Consider a infinitesimal saturated phase in J_{sat}. Let p_o and s_o be the number of processors and speed allocated by Opt(J_{sat}) to this phase. By the definition of canonical, the phase is parallelizable up to p_o processors. Thus Opt(J_{sat}) is processing at a rate of $p_o s_o$. Define Opt(J'_{par}) so that it runs at speed $s'_o = (\frac{1}{m})^{1-1/\alpha} p_o s_o$ on this phase. Since the transformation scales down the work by a factor of $m^{1-1/\alpha}$, Opt(J'_{par}) will complete the phase at the same time as Opt(J_{sat}).

Now we need to argue that at any point of time, the power for Opt(J_{sat}) will be at least the power for Opt(J'_{par}). The power at this time in Opt(J_{sat}) is $P = \sum_j p_{o,j}(s_{o,j})^\alpha$, where the sum is over all jobs j it is processing and $p_{o,j}$ and $s_{o,j}$ are the number and speed of the processors allocated to j. Keeping $R = \sum_j p_{o,j} \ s_{o,j}$ fixed, P is minimized by having all the $s_{o,j}$ to be the same fixed value s_o. This gives $R = \sum_j p_{o,j} s_o = s_o m$ and

$$P \geq \sum_j p_{o,j} \ {s_o}^\alpha = {s_o}^\alpha m = \left(\frac{R}{m}\right)^\alpha m = \left(\frac{1}{m}\right)^{\alpha-1} R^\alpha$$

By our definition of Opt(J'_{par}), the power P' in Opt(J'_{par}) can be bounded as follows:

$$\begin{aligned} P' &= \left(\sum_j \left(\frac{1}{m}\right)^{1-1/\alpha} p_{o,j} \ s_{o,j}\right)^\alpha \\ &= \left(\left(\frac{1}{m}\right)^{1-1/\alpha} R\right)^\alpha \\ &= \left(\frac{1}{m}\right)^{\alpha-1} R^\alpha \\ &\leq P \end{aligned}$$

□

Lemma 9. $cost_{\mathrm{Opt}}(J'_{seq}) = O(cost_{\mathrm{Opt}}(J_{uns}))$.

Proof. Consider a unsaturated phase in J_{uns} that is parallel up to p_o processors. We graciously allow $\mathrm{Opt}(J_{uns})$ to schedule each phase in J_{uns} in isolation of the other phases. This only improves $\mathrm{Opt}(J_{uns})$. Consider a particular phase in J_{uns} with a speedup function that is parallel up to p_o processors, and that has work w. By Lemma 1, the total flow time plus energy incurred for $\mathrm{Opt}(J_{uns})$ is $\Theta\left(\frac{w}{(p_o)^{1-1/\alpha}}\right)$. $\mathrm{Opt}(J'_{seq})$ will allocate zero processors to the corresponding phase, and process the phase at rate 1 since the work in J'_{seq} is sequential. Hence $\mathrm{Opt}(J'_{seq})$ incurs no energy cost for this phase.

So to finish the proof we will show that the flow time for this phase in $\mathrm{Opt}(J'_{seq})$ is at most the cost of this phase in $\mathrm{Opt}(J_{uns})$, namely $\Theta\left(\frac{w}{(p_o)^{1-1/\alpha}}\right)$. Recall that in the transformation from J to J', this work is scaled down by the fastest rate that this phase is processed by any copy in MultiLAPS. Consider the copy in MultiLAPS that is processing in the parallel range with the most number of processors, i.e., the copy with 2^i processors such that 2^i is maximized and at most p_o. Since the phase is unsaturated $2 \cdot 2^i > p_o$. By the definition of MultiLAPS, the processing rate of this copy is at least $2^i \cdot 2(\frac{1}{2^i})^{1/\alpha} = 2(2^i)^{1-1/\alpha} \geq 2(\frac{p_o}{2})^{1-1/\alpha} \geq p_o^{1-1/\alpha}$. Thus the work in this phase in J'_{seq}, and the flow time for this phase in $\mathrm{Opt}(J'_{seq})$, is at most $\frac{w}{p_o^{1-1/\alpha}}$. □

One can extend the analysis for LAPS in the uniprocessor setting, to the Sequential Setting, using the techniques used in [8, 9]. We refer the reader to [9] for full details, and just give the underlying intuition here. The analysis uses amortized local competitiveness. That is, it is shown that at every time,

$$P_{\mathrm{LAPS}} + n_{\mathrm{LAPS}} + d\Phi/dt \leq c \cdot P_{\mathrm{Opt}} + n_{\mathrm{Opt}}$$

where P denotes power, n denotes the number of active jobs, Φ is the potential function, and c is the desired competitive ratio. So when $P_{\mathrm{LAPS}} + n_{\mathrm{LAPS}}$ is large, the processing of LAPS lowers the potential function Φ enough to make the equation true. Now consider the sequential setting. The difficulty that arises is that the processing that LAPS does on sequential jobs may not lower the potential function. However, if the number of sequential phases that LAPS is processing is very small, then raising the speed of LAPS by a small amount will be enough so that the potential function decreases sufficiently quickly due to the processing on the non-sequential jobs. If the number of sequential phases is large at a particular time, then the increase in flow time that LAPS is experiencing on these jobs is also experienced by the adversary at some point in time. This increase in flow time experienced by the adversary pays for the increase in flow time for LAPS at this point of time. Note that by definition of LAPS, the power used by LAPS is comparable to the increase in flow time experienced by LAPS. We can thus derive the following theorem.

Theorem 10. $cost_{\mathrm{LAPS}}(J) = O(cost_{\mathrm{Opt}}(J'_{par}) + cost_{\mathrm{Opt}}(J'_{seq}))$.

4 Lower Bound for Checkpointable Multiple Copies

We show here that in the maximum processing model, every randomized algorithm has a competitive ratio at least poly-log.

Theorem 11. *In the maximum processing model, the competitive ratio for every randomized nonclairvoyant algorithm, is* $\Omega\left(\log^{1/\alpha} m\right)$ *against an oblivious adversary that must specify the input a priori.*

Proof. Applying Yao's technique, we give a probability distribution over the input, and show that every deterministic algorithm A will have an expected competitive ratio of $\Omega\left(\log^{1/\alpha} m\right)$. There are $\Theta(\log m)$ possible instances, each selected with equal probability. For each $j \in [0, \log(m)]$, instance J_j will consist of one job with work $w_j = 2^{(1-1/\alpha)j}$ and speedup function $\Gamma_{2^j}(p)$, where $\Gamma_q(p)$ is the speed up function that is parallelizable up to q processors. By applying Lemma 2, $\mathrm{Opt}(J_j)$ allocates $p_j = 2^j$ processors, each of speed $s_j = (2^j)^{-1/\alpha}$, resulting in a cost of $\Theta(1)$.

Now consider any deterministic nonclairvoyant algorithm A. Rounding the number of processors a copy is run on to a factor of two doesn't change the objective by more than a constant factor, and there is no significant benefit from running two copies on an equal number of processors. Since the algorithm is nonclairvoyant, it will gain no information about the identity of J_j until some copy finishes. Since the power function is convex, it is best for the algorithm to run each copy at constant speed. Thus we can assume that the algorithm runs $\log m$ copies of the job, with copy i run on 2^i processors at at some constant speed s_i. Note that the algorithm can set $s_i = 0$ if it doesn't want to run a copy on that many processors. The power of copy i is $P'_i = p_i {s_i}^\alpha = 2^i {s_i}^\alpha$ and the total power for A is $P' = \sum_i P'_i$.

Let $R_{\langle i,j\rangle} = \Gamma_{2^j}(2^i)s_i$ denote the rate that the copy i is processing work on job J_j. Because we are assuming that the work completed on a job is the maximum of that completed by the groups working on it, we have that $R_j = \max_i R_{\langle i,j\rangle}$ is the rate that A completes work on job J_j and we have that $T_j = \frac{w_j}{R_j}$ is the time until the job is completed. The adversary chooses j to maximize this time. We bound this maximum, denoted by T, as follows

$$\begin{aligned}
\frac{1}{T} &= \min_j \frac{1}{T_j} \\
&= \frac{O(1)}{\log m} \sum_j \frac{1}{T_j} \\
&\leq \frac{O(1)}{\log m} \sum_j \frac{\sum_i R_{\langle i,j\rangle}}{w_j} \\
&= \frac{O(1)}{\log m} \sum_i \sum_j \frac{\Gamma_{2^j}(2^i)s_i}{w_j} \\
&= \frac{O(1)}{\log m} \sum_i s_i \left[\sum_{j\in[0,i]} \frac{\Gamma_{2^j}(2^i)}{w_j} + \sum_{j\in[i+1,\log m]} \frac{\Gamma_{2^j}(2^i)}{w_j} \right] \\
&= \frac{O(1)}{\log m} \sum_i s_i \left[\sum_{j\in[0,i]} \frac{2^j}{2^{(1-\frac{1}{\alpha})j}} + \sum_{j\in[i+1,\log m]} \frac{2^i}{2^{(1-\frac{1}{\alpha})j}} \right]
\end{aligned}$$

$$\begin{aligned}
&= \frac{O(1)}{\log m}\sum_i s_i \left[\sum_{j\in[0,i]} 2^{\frac{1}{\alpha}j} + 2^i \cdot \sum_{j\in[i+1,\log m]} \frac{1}{2^{(1-\frac{1}{\alpha})j}}\right] \\
&= \frac{O(1)}{\log m}\sum_i s_i \left[2^{\frac{1}{\alpha}i} + 2^i \cdot \frac{1}{2^{(1-\frac{1}{\alpha})i}}\right] \\
&= \frac{O(1)}{\log m}\sum_i P_i'^{1/\alpha}
\end{aligned}$$

Subject to $P' = \sum_i P_i'$, the sum $\sum_i P_i'^{1/\alpha}$ is maximized by setting each P_i' to $\frac{P'}{\log m}$ giving

$$\frac{1}{T} = \frac{O(1)}{\log m}\sum_i \left(\frac{P'}{\log m}\right)^{1/\alpha} = O\left(\frac{P'}{\log m}\right)^{1/\alpha}$$

The total cost for A is

$$\begin{aligned}
F_A + E_A &= T + P'T \\
&= (1+P')T \\
&= \Omega\left((1+P')\left(\frac{\log m}{P'}\right)^{1/\alpha}\right) \\
&= \Omega\left(\log^{1/\alpha} m\right)
\end{aligned}$$

Recalling that the optimal cost is $O(1)$, the result follows. $\square$

5 Conclusion

In summary, we have shown that for jobs that may have side effects or that are not checkpointable, the achievable competitive ratio grows quickly with the number of processors. And for checkpointable jobs without side effects, the achievable competitive ratio grows slowly with the number of processors.

It is relatively straight-forward to observe that the techniques in this paper give an $O(1)$-competitive algorithm in the case that all speed up functions are parallel up to some number of processors and then sequential, and the online algorithm knows this degree of parallelizability.

Acknowledgments: We thank Nikhil Bansal, Tak Wah Lam, Lap Kei Lee, and Alberto Marchetti-Spaccamela for helpful discussions.

References

[1] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3(4), 2007.

[2] Nikhil Bansal, Ho-Leung Chan, Tak-Wah Lam, and Lap-Kei Lee. Scheduling for bounded speed processors. In *International Colloquium on Automata, Languages and Programming*, pages 409 – 420, 2008.

[3] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 693–701, 2009.

[4] Nikhil Bansal, Kirk Pruhs, and Cliff Stein. Speed scaling for weighted flow time. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 805–813, 2007.

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

[6] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. *Journal of Discrete Algorithms*, 4(3):339–352, 2006.

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

[8] Jeff Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

[9] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 685–692, 2009.

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

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

[12] Tak-Wah Lam, Lap-Kei Lee, Isaac To, and Prudence Wong. Speed scaling functions for flow time scheduling based on active job count. In *European Symposium on Algorithms*, pages 647–659, 2008.

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

[14] Rajeev Motwani, Steven Phillips, and Eric Torng. Nonclairvoyant scheduling. *Theorertical Computer Science*, 130(1):17–47, 1994.

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

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

[17] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard J. Woeginger. Getting the best response for your erg. *ACM Transactions on Algorithms*, 4(3), 2008.

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

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