Nonclairvoyantly Scheduling Power-Heterogeneous Processors

Anupam Gupta Ravishankar Krishnaswamy
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA

Kirk Pruhs
Computer Science Department
University of Pittsburgh
Pittsburgh USA

***Abstract*—We show that a natural nonclairvoyant online algorithm for scheduling jobs on a power-heterogeneous multiprocessor is bounded-speed bounded-competitive for the objective of flow plus energy.**

***Keywords*-Speed scaling, power management**

I. Introduction

Many computer architects believe that architectures consisting of heterogeneous processors/cores will be the dominant architectural design in the future [BSC08], [KTR+04], [KTJ06], [Mer08], [MWK+06]. The main advantage of a heterogeneous architecture, relative to an architecture of identical processors, is that it allows for the inclusion of processors whose design is specialized for particular types of jobs, and for jobs to be assigned to a processor best suited for that job. Most notably, it is envisioned that these heterogeneous architectures will consist of a small number of high-power high-performance processors for critical jobs, and a larger number of lower-power lower-performance processors for less critical jobs (see figure 1 for a visual representation of such an architecture). Naturally, the lower-power processors would be more energy efficient in terms of the computation performed per unit of energy expended, and would generate less heat per unit of computation. For a given area and power budget, heterogeneous designs can give significantly better performance for standard workloads [BSC08], [Mer08]; Evaluations in [KTJ06] suggest a figure of 40% better performance, and evaluations in [MWK+06] suggest a figure of 70% better performance. Moreover, even processors that were designed to be homogeneous, are increasingly likely to be heterogeneous at run time [BSC08]: the dominant underlying cause is the increasing variability in the fabrication process as the feature size is scaled down (although run time faults will also play a role). Since manufacturing yields would be unacceptably low if every processor/core was required to be perfect, and since there would be significant performance loss from derating the entire chip to the functioning of the least functional processor (which is what would be required in order to attain processor homogeneity), some processor heterogeneity seems inevitable in chips with many processors/cores.

Anupam Gupta and Ravishankar Krishnaswamy were supported in part by NSF awards CCF-0448095 and CCF-0729022, and an Alfred P. Sloan Fellowship.

Kirk Pruhs was supported in part by NSF grants CNS-0325353, IIS-0534531, and CCF-0830558, and an IBM Faculty Award.

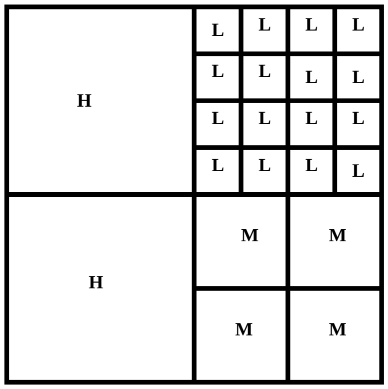

Figure 1. An example layout of an architecture consisting of two high power processors (labeled H), four medium power processors (labeled M), and sixteen lower power processors (labeled L).

The position paper [BSC08] argues for the fundamental importance of research into scheduling policies for heterogeneous processors, and identified three fundamental challenges in scheduling heterogeneous multiprocessors: (1) the OS must discover the status of each processor, (2) the OS must discover the resource demand of each job, and (3) given this information about processors and jobs, the OS must match jobs to processors as well as possible. The contribution of this paper is probably best summarized as an initial step in a theoretical worst-case investigation of challenge (2) and (3) in tandem. To explain this contribution however, it is necessary to first review the results in [GKP], which is some sense completely solved challenge (3) from a worst-case theoretical perspective.

A. The Problem Formulation, and the Clairvoyant Algorithm and Analysis

[GKP] introduced the following model, building on an earlier model in [BCP09]. Each processor i has a known collection of allowable speeds $s_{i,1}, \ldots, s_{i,f(i)}$, and associated powers $P_{i,1}, \ldots, P_{i,f(i)}$. A collection of jobs arrive in an online fashion over time. Job j arrives in the system at its release time r_i. Job j has an associated *size* $p_j \in \mathbb{R}_{>0}$, as well as a *importance/weight* $w_j \in \mathbb{R}_{>0}$.

An online scheduler has two component policies:

Job Selection: Determines which job to run on each processor at any time.

Speed Scaling: Determines the speed of each processor at each time.

The objective considered in [GKP] is that of *weighted flow plus energy*. A job of size p takes $\frac{p}{s}$ units of time to complete if run at speed s. The flow F_j of a job j is its completion time C_i minus its release time r_j. The weighted flow for a job j is $w_j F_j$, and the weighted flow for a schedule is $\sum_j w_j F_j$. The intuitive rationale for the objective of weighted flow plus energy can be understood as follows: Assume that the possibility exists to invest E units of energy to decrease the flow of jobs $j_1, \ldots, j_k$ by $x_1, \ldots, x_k$ respectively; then an optimal scheduler for this objective would make such an investment if and only if $\sum_{i=1}^{k} w_i x_i \geq E$. So the importance w_j of job j can be viewed as specifying an upper bound on the amount of energy that the system is allowed to invest to reduce j's flow time by one unit of time (assuming that this energy investment in running j faster doesn't change the flow time of other jobs)— hence jobs with higher weight are more important, since higher investments of energy are permissible to justify a fixed reduction in flow.

[GKP] considered the following natural online algorithm, consisting of three policies, which we will call GKP:

Job Selection: On each processor, always run the highest density job assigned to that processor. The density of a job is its weight divided by its size.

Speed Scaling: The speed of each processor is set so that the resulting power is the (fractional) weight of the unfinished jobs on that processor. This guarantees that the energy used will be identical to the weighted (fractional) flow time.

Assignment: When a new job arrives, it is greedily assigned to the processor that results in the least increase in the projected future (fractional) weighted flow, assuming the adopted speed scaling and job selection policies, and ignoring the possibility of jobs arriving in the future.

[GKP] evaluated this algorithm using resource augmentation analysis [KP00], which is a type of worst-case comparative analysis, and which we now explain within the context of the type of problem that we consider here. An algorithm A is said to be c-competitive relative to a benchmark algorithm B if for all inputs I it is the case that

$$A(I) \leq c \cdot B(I)$$

where $A(I)$ is the value of the objective of the schedule output by algorithm A on input I, and $B(I)$ is the value of the objective on the benchmark schedule for input I. In other words, the competitiveness of c represents a worst-case error relative to the benchmark. The most obvious choice for the benchmark is probably the optimal schedule for each instance I. But since scheduling on identical processors with the objective of total flow (and even scheduling on a single processor with the objective of weighted flow), is a special case of the problem we consider, bounded competitiveness relative to the optimal schedule is not possible [LR07], [BC09].

This is a common phenomenon in online scheduling problem, and the standard remedy, called resource augmentation, is to use as a benchmark the optimal schedule that uses slightly slower processors. In this context, an algorithm A is σ-speed c-competitive if A, equipped with processors with speeds $\sigma \cdot s_{i,1}, \ldots, \sigma \cdot s_{i,f(i)}$, and associated powers $P_{i,1}, \ldots, P_{i,f(i)}$ is c-competitive relative to the benchmark of the optimal schedule for processors with speeds $s_{i,1}, \ldots, s_{i,f(i)}$, and associated powers $P_{i,1}, \ldots, P_{i,f(i)}$.

To understand the motivation for resource augmentation analysis, note that it is common for systems to posses the following (informally defined) *threshold property*: The input or input distributions can be parameterized by a load λ, and the system is parameterized by a capacity μ. The system then has the property that its QoS would be very good when the load λ is at most 90% of the system capacity μ, and it is horrible if λ exceeds 110% of μ. Figure 2 gives such an example of the QoS curve for a system that has this kind of threshold property. Figure 2 also shows the performance of an online algorithm A which compares reasonably well with the performance of an optimal algorithm. Notice however that the competitive ratio of A relative to the optimal is very large when the load is near capacity μ since there is a large vertical gap between the two curves at load values slightly below μ. In order to completely explain why the curves for A and optimal in Figure 2 are "close", we need to also measure the horizontal gap between curves. This intuitively measures the ratio of the maximum load for which A has good performance with respect to the (typically larger) maximum load for which the optimal can guarantee good performance. To this end, we would like to say something like A performs at most c times worse than optimal on inputs with σ times higher load. Notice that multiplying the load by a factor of σ is equivalent slowing the system down by a factor of σ. This can be captured by a statement which says that A with an σ times faster processor is at most c times as bad as optimal.

The informal notion of an online scheduling algorithm A being "reasonable" is then generally formalized as A having bounded competitiveness for some small constant speed augmentation σ. Such a scheduling algorithm (when c is modest) would guarantee a system capacity of at least μ/σ, which is at least a constant fraction of the optimal capacity. The informal notion of an online scheduling algorithm being "good", is then generally formalized as A has has bounded competitiveness even if $\sigma = 1 + \epsilon$ is arbitrarily close to one.

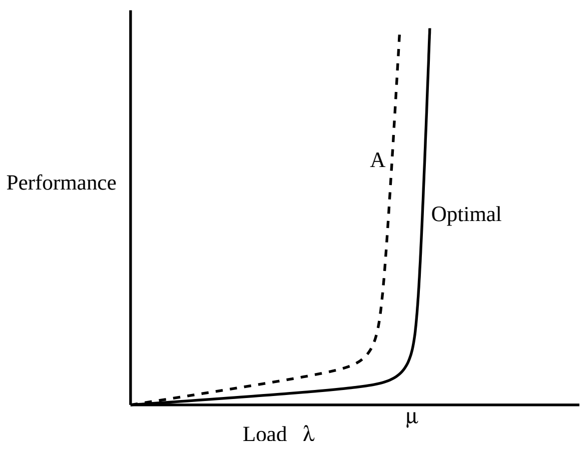

Figure 2. QoS curves of an almost fully scalable online algorithm A and the optimal algorithm for a system with the threshold property.

Such an algorithm is called *scalable* since it would guarantee a system capacity arbitrarily close to the optimal capacity, while also ensuring that the QoS remains comparable (to within a constant factor). For a more detailed elaboration see [PST04], [Pru07].

The main result in [GKP] was that the online algorithm GKP was scalable for weighted flow plus energy. The analysis in [GKP] extended theorems showing similar results for weighted flow plus energy on a uniprocessor [BCP09], [AWT09], and for weighted flow on a multiprocessor without power considerations [CGKM09].

B. The Contribution of this Paper

So in some sense [GKP] shows that the natural greedy algorithm GKP has the best possible worst-case performance for challenge (3) from [BSC08]. However note that the GKP algorithm is clairvoyant, that is, it needs to know the job sizes when jobs are released. In the GKP algorithm, the job selection policy needs to know the size of a job to compute its density, and the assignment policy must know the size and density to compute the future costs. Thus the GKP algorithm is not directly implementable as in general one can not expect the system to know job sizes when they are released.

Thus the natural question left open in [GKP] is what is the best possible nonclairvoyant scheduling algorithm. Nonclairvoyant algorithms do not require knowledge of a size of a job. This can be viewed as addressing challenges (2) and (3) from [BSC08] in tandem (described in Section I). We note that it is both practically natural, and mathematically necessary, to assume that the system does know the importance of each job.

In this paper, we make a first step toward addressing the open question of finding the best nonclairvoyant scheduling policy. In particular, we consider the simplification that each job has the same importance, or equivalently, we consider the objective of (unweighted) flow time plus energy. Our main result is that a natural nonclairvoyant algorithm is bounded-speed bounded-competitive for the objective of flow plus energy. More precisely, we show that this natural nonclairvoyant scheduling algorithm is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for the objective of flow plus energy. So intuitively, if this scheduling algorithm is adopted then the system should have capacity at least approximately half of the optimal system capacity. So using the standard interpretation, this natural nonclairvoyant algorithm should be viewed as "reasonable". Or at least the algorithm seems as reasonable as Equipartition is for the objective of minimizing average flow time (without energy considerations) as it is known that Equipartition is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive in this context [EP09].

We now describe the component policies of our nonclairvoyant algorithm:

Speed Scaling: A collection of processors and associated speed settings are selected so as to maximize the aggregate speed, subject to the constraints that the of cardinality of the selected processors at most the number of unfinished jobs, and the aggregate power is at most the number of unfinished jobs.
Job Selection: The jobs share this processing power equally.

In other words, the job selection policy is Equipartition (or equivalently Round Robin or Processor Sharing). As for our speed scaling policy, the intuition is that it tries to maximize the effective aggregate speed of the assigned processors subject to the same maximum power constraint as in GKP and earlier algorithms. We show that this speed scaling policy can be implemented using a simple and efficient greedy algorithm.

However, note that, in contrast to the GKP algorithm, this algorithm produces migratory schedules, that is, the same job may be run on different processors over time (however, no job is run on different machines at the same time). But it is easy to see that job migration is an unavoidable consequence of nonclairvoyance, that is, any bounded-speed bounded-competitive nonclairvoyant algorithm must migrate jobs.

C. Related Results

Let us now consider related scheduling problems where there are no power considerations. First let us assume a single processor. The online clairvoyant algorithm Shortest Remaining Time (SRPT) is optimal for unweighted flow. The nonclairvoyant algorithm Shortest Elapsed Time First is scalable for unweighted flow [KP00]. The nonclairvoyant algorithm Round Robin is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive [EP09] for unweighted flow. The algorithm Highest Density First (HDF) is scalable for weighted flow [BL04], and there is no online algorithm that has bounded competitiveness against the optimal schedule [BC09]. Now let us consider multiple identical processors. SRPT is $O(\log n)$-competitive against the optimal schedule, and no better competitiveness is achievable [LR07]. The clairvoyant algorithm HDF is scalable for weighted flow [BT06]. The nonclairvoyant algorithm Weighted LAPS, which is based on the algorithm LAPS in [EP09], is scalable for weighted flow, and this can be inferred from the problem being a special case of the problem of broadcast scheduling [BKN].

We now turn our attention on prior work on scheduling involving power management. For the case of a single processor with unbounded speed and a polynomially bounded power function $P(s) = s^\alpha$, [PUW08] gave an efficient *offline* algorithm to find the schedule that minimizes average flow subject to a constraint on the amount of energy used, in the case that jobs have unit work. However, no such result involving an energy constraint is possible when we transition to online algorithms. Therefore, [AF07] introduced the objective of *flow plus energy* and gave a constant competitive algorithm algorithm for this objective in the case of unit work jobs. Subsequently, [BPS09] gave a constant competitive algorithm for the objective of weighted flow plus energy. The competitive ratio was improved by [LLTW08b] for the unweighted case using a potential function specifically tailored to integer flow. [BCLL08] extended the results of [BPS09] to the bounded speed model, and [CEL$^+$09] gave a *nonclairvoyant* algorithm that is $O(1)$-competitive.

Remaining on a single processor, [BCP09] dropped the assumptions of unbounded speed and polynomially-bounded power functions, and gave a 3-competitive algorithm for the objective of unweighted flow plus energy, and a 2-competitive algorithm for fractional weighted flow plus energy, when the power function could be arbitrary. The former analysis was subsequently improved to show 2-competitiveness, along with a matching lower bound on the competitive ratio [AWT09].

Moving on to the setting of multiple machines, [LLTW08a] considers the problem of minimizing flow plus energy on multiple homogeneous processors, where the allowable speeds range between zero and some upper bound, and the power function is polynomial. [LLTW08a] show that an algorithm that uses a variation of round robin for the assignment policy, and uses the job selection and speed scaling policies from [BPS09], is scalable for this problem. [CEP09] show that bounded-competitiveness for the objective of flow plus energy is not achievable on multiprocessors if jobs can be run simultaneously on multiprocessors, and have varying speed-ups (i.e jobs have different degrees of parallelism). [CEP09] give an optimally log competitive algorithm building on the results in [CEL$^+$09].

D. Preliminaries

1) Scheduling Basics.: A schedule specifies for each time and each processor, a speed for that processor and a job that each processor runs. We assume that no job may be run on more than one processor simultaneously. The speed is the rate at which work is completed; a job j with size p_j run at a constant speed s completes in $\frac{p_j}{s}$ seconds. A job is completed when all of its work has been processed. The flow time of a job is the completion time of the job minus the release time of the job. The weighted flow of a job is the weight of the job times the flow time of the job (for our results, all jobs have the same weight, and we could think of the weights as being unit).

2) Power Functions.: As noted in [BCP09] we can interpolate the discrete speeds and powers of a processor to a piecewise linear function in the obvious way. See figure 3 for an illustration. To elaborate, let s_1 and s_2 be two allowable speeds for a processor, with associated powers P_1 and P_2. By time multiplexing the speeds s_1 and s_2 with proportion λ and $1-\lambda$ respectively (here $\lambda \in [0,1]$), one can effectively have a processor that runs at speed $\lambda s_1 + (1-\lambda)s_2$ with power $\lambda P_1 + (1-\lambda)P_2$. Note that then this is just the linear interpolation of the two points (s_1, P_1) and (s_2, P_2). As noted in [BCP09] we may then assume without loss of generality that the power function has the following properties: $P(0) = 0$, is strictly increasing, and is strictly convex. We will use P_i to denote the resulting power function for processor i, and use Q_i to denote P_i^{-1}; i.e., $Q_i(y)$ gives us the speed that we can run processor i at, if we specify a limit of y. We note that since P_i is convex, Q_i is concave and we exploit this fact in our proofs.

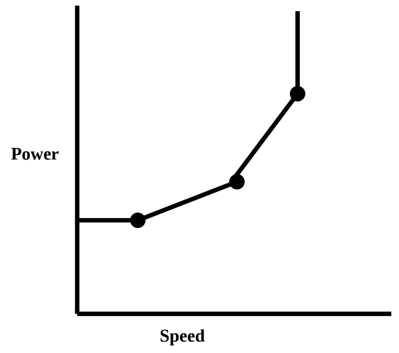

Figure 3. An illustration of the natural extension of a power function for three discrete speeds to a piecewise linear power function.

3) Local Competitiveness and Potential Functions.: Finally, let us quickly review the technique of amortized competitiveness analysis on a single processor, which we use in our proofs. Consider an objective G (in our setting, it is unweighted flow plus energy). Let $G_A(t)$ be the increase in the objective in the schedule for algorithm A at time

t. So when G is unweighted flow plus energy, $G_A(t)$ is $P_a(t) + n_a(t)$, where $P_a(t)$ is the total power used by A at time t and $n_a(t)$ is the number of unfinished jobs for A at time t. Let OPT be the optimal benchmark schedule we could like to compare against. Then, the algorithm A is said to be locally c-competitive if for all times t, if $G_A(t) \leq c \cdot G_{OPT}(t)$. A weaker notion of that of amortized competitiveness: To prove A is $(c+d)$-competitive using an amortized local competitiveness argument, it suffices to give a potential function $\Phi(t)$ such that the following conditions hold (see for example [Pru07]).

Boundary condition: Φ is zero before any job is released and Φ is non-negative after all jobs are finished.

Completion condition: Φ does not increase due to completions by either A or OPT.

Arrival condition: Φ does not increase more than $d \cdot OPT$ due to job arrivals.

Running condition: At any time t when no job arrives or is completed,

$$G_A(t) + \frac{d\Phi(t)}{dt} \leq c \cdot G_{OPT}(t) \tag{1}$$

The sufficiency of these conditions for proving $(c+d)$-competitiveness follows from integrating them over time.

II. The Description of the Algorithm

In this section we describe the nonclairvoyant algorithm, which we will denote by A, in greater detail. Like mentioned in the introduction (Section I-B), A consists of two components, (i) the speed scaling policy which at any time t determines the power to run each processor at, and (ii) the job selection policy which decides which job is run on which processor.

At any time instant t, let $n_a(t)$ denote the number of unfinished jobs for A, our online algorithm, that have already been released. Also let $N_a(t)$ denote the set of these unfinished jobs. The greedy speed scaling policy, which we denote by GreedySS, intuitively incrementally invests $n_a(t)$ total units of energy into the processors that will give it the greatest increase in aggregate speed. For completeness, we give pseudo-code for GreedySS, which takes an integer input parameter W:

```
GreedySS($W$):
• Initially set $E_i := 0$ for all processors $i$. $E_i$ will
  eventually be the power used by processor $i$.
• For $j = 1$ to $W$ do
    – Let $k = \arg\max_i Q_i(E_i + 1) - Q_i(E_i)$
    – Increment $E_k$ to $E_k + 1$.
• Set the speed $s_i$ of each processor $i$ to be $Q_i(E_i)$
```

Recall that $Q_i(y)$ is the inverse power function for processor i and specifies the maximum speed that processor i can run subject to a power constraint of y. Intuitively, the algorithm partitions the power budget into units, and assigns each unit to the machine which offers the best increase to the total speed that can be extracted. The speed scaling policy for A is then just GreedySS(n_a).

The job selection policy for A is *Equipartition*, which equally shares the speed among the processors. That is, each job is effectively run at speed $\sum_i s_i/n_a$. Furthermore, since there are only at most $n_a(t)$ machines which are powered on at an time t, it is possible to equally share the total resources among the jobs without having to schedule any job on two different machines at the the same time. We remark that this seems like the most natural nonclairvoyant job selection policy, since the scheduler is not aware of the initial job sizes, and hence remaining unprocessed sizes.

Before we analyze the algorithm A, we now show that GreedySS optimally solves the following *speed extraction optimization problem*:

Speed Extraction Optimization Problem: Assign a power budget of E_i to each processor i so as to maximize the total extracted speed $\sum_i Q_i(E_i)$ subject to the constraints:

- Each E_i is a nonnegative integer,
- $\sum_i E_i \leq n_a(t)$, and
- the number of machines that are powered, i.e. have $E_i > 0$, is at most n_a.

Lemma 2.1: The greedy algorithm GreedySS optimally solves the speed extraction problem.

Proof: Imagine the optimal speed scaling policy assigns a power budget of $E_i^* \in \mathbb{Z}_{\geq 0}$ to machine i, such that $\sum_i E_i^* = n_a(t)$ and the number of machines for which $E_i^* > 0$ is at most $n_a(t)$. Also let $s^* = \sum_i Q_i(E_i^*)$ be the speed achieved by this power distribution.

Firstly, notice that we can view s^* as the following sum:

$$s^* = \sum_{i=1}^{m} \sum_{j=0}^{E_i^*} Q_i(j+1) - Q_i(j) \tag{2}$$

But notice that we could also express the total speed extracted by the greedy algorithm as a sum of such increments, with each increment being selected at some point in the loop iteration in step (2). Now, suppose for the sake of contradiction the greedy selection differs from the optimal policy. Then, consider the first point j in the loop iteration where the greedy algorithm picked a machine i and incremented power from E to $E+1$ for some value of E, such that the optimal solution does not include the term $Q_i(E+1) - Q_i(E)$ in s^* (in equation 2).

Let the values of the E_i's (in the greedy algorithm GreedySS) at the beginning of this iteration be denoted by $E_i^{j^*}$, for $1 \leq i \leq m$. Since the first $j^* - 1$ choices made by the greedy policy are all included in the optimal solution as well, and both the greedy algorithm and the optimal algorithm have the constraint on the total power budget to be $n_a(t)$, it must mean that the optimal solution includes some other term of the form $Q_{i'}(E'+1) - Q_{i'}(E')$ which the greedy algorithm does not choose in its $n_a(t)$ iterations. Furthermore, it must be that $E' > E_{i'}^{j^*}$ otherwise the greedy algorithm must have also made this selection in one of the first $j^* - 1$ iterations.

However, the nature of the greedy selection, and the concavity of the functions Q_i ensure that

$$\begin{aligned} Q_i(E+1) - Q_i(E) &\geq Q_{i'}(E_{i'}^{j^*}+1) - Q_{i'}(E_{i'}^{j^*})) \\ &\geq Q_{i'}(E'+1) - Q_{i'}(E') \end{aligned}$$

The first inequality above is due to the greedy choice of the selection in iteration j^* and the second follows from the concavity of the function $Q_{i'}$. Therefore the optimal solution would become no worse in terms of total speed it can extract by switching the latter selection with the former. But we could apply this swapping step repeatedly until the optimal solution and the greedy policy are identical without decreasing the speed extracted by the optimal policy. This completes the proof. ■

III. The Analysis of A

In this section, we show using an amortized local-competitiveness analysis that the algorithm A is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for the objective of flow plus energy. We will assume that the optimal schedule OPT is the schedule output by the GKP algorithm defined in [GKP], and then show that A is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive relative to this OPT. This is sufficient, as [GKP] shows that the GKP algorithm is $(1+\epsilon)$-speed $O(1/\epsilon)$-competitive. Therefore, we could combine these two results to get $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitiveness of our nonclairvoyant algorithm A.

We first reduce the analysis of A on an instance $\mathcal{I}$ to the analysis of A an associated instance $\mathcal{I}'$ where there is only a single processor. We show the cost for the online cost is preserved by the reduction and the optimal cost doesn't increase, that is:

$$\mathsf{A}(\mathcal{I}) = \mathsf{A}(\mathcal{I}')$$

and

$$\text{OPT}(\mathcal{I}) \geq \text{OPT}(\mathcal{I}')$$

Then it is sufficient to show that A is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for the objective of flow plus energy on the instance $\mathcal{I}'$ to get the desired result.

In subsection III-A we give the reduction, and note that it is obvious that $\mathsf{A}(\mathcal{I}) = \mathsf{A}(\mathcal{I}')$. In subsection III-B we show that $\text{OPT}(\mathcal{I}) \geq \text{OPT}(\mathcal{I}')$. In subsection III-C we show that A is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive (relative to the GKP schedule) for the objective of flow plus energy on the instance $\mathcal{I}'$.

A. *The Reduction*

For each job in $\mathcal{I}$, there is a corresponding job in $\mathcal{I}'$ with the same size and release time. The single processor in the instance $\mathcal{I}'$ has the inverse power function

$$Q(W) := \mathsf{GreedySS}(\lceil W \rceil)$$

In other words, $Q(W)$ is simply the total speed used by A with $\lceil W \rceil$ jobs. To define the power function $P(\cdot)$, we simply say that $P(s)$ is the smallest W such that $Q(W) \geq s$. It is clear from the nature of the reduction that the schedules $\mathsf{A}(\mathcal{I})$ and $\mathsf{A}(\mathcal{I}')$ are identical, since at any time t, the speed of the single processor in $\mathcal{I}'$ is set to be $Q(n_a(t))$, which is equal to the total speed extracted in $\mathcal{I}$, which is $\mathsf{GreedySS}(n_a(t))$. Because of this, the extent to which each job is run is identical in both schedules.

B. *Comparing the costs of the optimal solutions*

In this section we show that $\text{OPT} = \text{OPT}(\mathcal{I})$ has cost at least $\text{OPT}' = \text{OPT}(\mathcal{I}')$. We accomplish this by finding a feasible candidate for OPT' of cost at most OPT, which recall we are assuming is the output of the GKP algorithm on $\mathcal{I}$. The following facts about the GKP schedule are useful and are proved in [GKP]:

(a) When a job is released, the GKP algorithm assigns it to some machine which would minimize the total future cost that is to be incurred. This assignment is fixed when a job arrives, and jobs are never migrated.
(b) At any point in time, the GKP algorithm runs machine i at a speed of $Q_i(n_o(i,t))$, where $n_o(i,t)$ is the number of unfinished jobs in GKP that have been assigned to machine i. Notice that the total power used at any time is exactly $n_o(t) = \sum_i n_o(i,t)$.

Then, suppose we set the speed of the single processor in OPT' at time t is be $Q(n_o(t))$. Further, OPT' schedules the same set of jobs OPT schedules on its processors at this time instant, at the same rates. To show that this is feasible, it is enough to show that $Q(n_o(t))$ is at least the total speed used by OPT on its jobs at time t. But this is clear because of property (b) we observed above, that the power distribution of OPT is *one* feasible solution for the problem solved by the greedy scaling policy $\mathsf{GreedySS}(n_o(t))$, and hence the *best* such partition would only extract at least as much speed. Hence, OPT' simply imitates OPT at this time so each job completes at the same time in OPT' as the corresponding job in $\mathcal{I}$ does in OPT, and the power used at any time in OPT' is at most the power used in OPT. Hence, $\text{OPT} \geq \text{OPT}'$.

C. *Comparing* A *and* OPT'

We show that A is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive (relative to the schedule OPT' output by the GKP algorithm on the instance $\mathcal{I}'$) for the objective of flow plus energy using an amortized local competitiveness argument. To this end, let us define the following potential function:

$$\Phi(t) = \sum_{j=1}^{n_a(t)} \frac{\mathsf{rank(j,t)}}{Q(\mathsf{rank(j,t)})} \max\left(0, x_a(j,t) - x_o(j,t)\right) \quad (3)$$

Here, $N_a(t)$ denotes the set of jobs that are unfinished at time t in A, $n_a(t) = |N_a(t)|$, and for all $j \in N_a(t)$, $\mathsf{rank(j,t)}$ denotes the index of how late the job arrived among all unfinished jobs for A at time t. That is the most recent job would have a rank of $|N_a(t)|$, and the earliest arriving job would have a rank of 1. The term $x_a(j,t)$ denotes the amount of processing unfinished by A for job j at time t, and $x_o(j,t)$ denotes the analogous quantity for the optimal solution OPT'.

We now show that all of the conditions in an amortized local competitiveness argument, reviewed in Section I-D3, hold for the potential function Φ. Clearly, at $t = 0$, $x_a(j,t) = x_o(j,t)$ for any job j and hence $\Phi(0) = 0$. Furthermore, when both A and OPT' have completed by some time t, again $\Phi(t) = 0$. Hence the boundary conditions trivially hold. As for the completion condition, notice that when A completes a job j, the term $x_a(j,t) - x_o(j,t)$ is non-positive, and therefore when we remove it from the summation because $j \notin N_a(t + dt)$, the potential only drops. Furthermore, all unfinished jobs which arrived subsequent to j will suffer a decrease in their corresponding terms in the potential function, since their rank drops by 1 and the function $x/Q(x)$ is non-decreasing (since $Q(\cdot)$ is concave). As for arrival conditions, notice that when a job j arrives, say at some time t, it does not change other jobs' indices/ranks, and $x_a(j,t) = x_o(j,t)$ and therefore the potential function does not increase on job arrivals (i.e. $d = 0$ from Section I-D3 in our case).

Now we deal with the most interesting case — the running condition. Consider some arbitrarily small time interval $[t, t + dt)$ when no jobs arrive or are completed by the online algorithm A. We now analyze the change in potential function in this time interval. The potential changes due to the following factors: (i) the optimal solution OPT' works on some jobs thereby decreasing $x_o(j,t)$ in some of the terms in Φ, and (ii) the online algorithm A works on all unfinished jobs and therefore brings down $x_a(j,t)$. We bound these changes in two steps. Recall that our aim is to show the following running condition inequality:

$$\begin{aligned} & (P_a(t) + n_a(t)) + \frac{d\Phi_i(t)}{dt} && (4)\\ = \;& (n_a(t) + n_a(t)) + \frac{d\Phi_i(t)}{dt} && (5)\\ \le \;& c\left(P_o(t) + n_o(t)\right) && (6)\\ = \;& c\left(n_o(t) + n_o(t)\right) && (7)\end{aligned}$$

Note that both A algorithm and the GKP algorithm maintain the invariant that the power is equal to the number of unfinished jobs.

The increase in Φ due to OPT' working. Since OPT' is using the GKP algorithm, its power at time t is exactly $n_o(t)$ and therefore the total speed it can work on any job is at most $Q(n_o(t))$. In the worst case, it dedicates this entire speed to the job with the highest index in Φ, that is the latest arriving job unfinished job in $N_a(t)$. Thus the total increase in Φ can be bounded by

$$\frac{n_a(t)}{Q(n_a(t))} Q(n_o(t)) dt \quad (8)$$

Notice that if $n_o(t) \ge n_a(t)$, then this is at most $n_o(t)$ since $x/Q(x)$ is a non-decreasing function when Q is concave. Therefore, in this case, the running condition holds with $c = 3/2$. Also, if $n_o(t) \le n_a(t) \le \frac{1}{\epsilon} n_o(t)$, then the term in expression 8 is at most

$$\frac{1}{\epsilon} \frac{n_o(t)}{Q(n_o(t))} Q(n_o(t)) dt \le \frac{1}{\epsilon} n_o(t) dt$$

This is because for any $\lambda \ge 1$, the non-decreasing nature of $Q(\cdot)$ implies that $\frac{\lambda n_o(t)}{Q(\lambda n_o(t))} \le \lambda \frac{n_o(t)}{Q(n_o(t))}$. Combining this fact (instantiating $\lambda = (1/\epsilon)$) with the fact that $x/Q(x)$ is also non-decreasing gives us $\frac{n_a(t)}{Q(n_a(t))} \le \frac{1}{\epsilon} \frac{n_o(t)}{Q(n_o(t))}$.

Therefore, in this case, the running condition holds with $c = (3/2\epsilon)$. Therefore, the most interesting case is when $n_a(t) > (1/\epsilon) n_o(t)$. In this case, we argue that Φ drops sufficiently to counter the rise.

The decrease in Φ due to A working. The online algorithm A works on all the unfinished jobs at the same rate, i.e. runs each job at a rate $\sigma Q(n_a(t))/n_a(t)$, where we assume that the online algorithm has a speed-up of σ over the optimal solution. Furthermore, it decreases Φ for all the jobs on which it is lagging behind OPT' (those jobs which it leads have $\max(x_a(j,t) - x_o(j,t), 0) = 0$ and Φ does not drop due to the processing of these jobs). But since $n_o(t) \le \epsilon n_a(t)$, the potential function drops for at least $(1-\epsilon) n_a(t)$ jobs, and in the worst case, these are the ones with ranks 1 through $n_a(t)(1-\epsilon)$.

Therefore, the total drop in Φ is at least

$$\begin{aligned} & \sum_{j=1}^{(1-\epsilon)n_a(t)} \frac{i}{Q(i)} \sigma \frac{Q(n_a(t))}{n_a(t)} dt \\ \ge \;& \frac{1}{2}(1-\epsilon) n_a(t) \frac{(1-\epsilon) n_a(t)}{Q((1-\epsilon) n_a(t))} \sigma \frac{Q(n_a(t))}{n_a(t)} dt \\ \ge \;& \frac{1}{2}(1-\epsilon) n_a(t) \frac{(1-\epsilon) n_a(t)}{Q(n_a(t))} \sigma \frac{Q(n_a(t))}{n_a(t)} dt \\ \ge \;& \frac{1}{2}(1-\epsilon)^2 n_a(t) \frac{(1-\epsilon) n_a(t)}{Q(n_a(t))} \sigma \frac{Q(n_a(t))}{n_a(t)} dt \\ \ge \;& \frac{\sigma}{2}(1-\epsilon)^2 n_a(t) \end{aligned}$$

Here, the second inequality follows because $x/Q(x)$ is a sub-additive function[1] when Q is concave, and for any

[1]A function f is sub-additive if $f(a+b) \le f(a) + f(b)$ for $a, b \ge 0$.

sub-additive function $f(\cdot)$, $\int_{x=0}^{t} f(x)dx \geq \frac{1}{2}tf(t)$. The third inequality holds because Q is non-increasing. Now, the term in the final inequality would be at least $2n_a(t)$ for $\sigma = (2+5\epsilon)$ when ϵ is sufficiently small. Therefore, the total decrease in Φ can counter both the increase in Φ due to OPT$'$ working and pay for the incremental cost of the online algorithm. Thus in this case, the running condition holds with $c = 0$.

Taking maximum over all cases, we see that $c = 3/2\epsilon$ suffices for the following inequality to hold at all times:

$$(n_a(t) + n_a(t)) + \frac{d\Phi_i(t)}{dt} \leq \frac{3}{2\epsilon}(n_o(t) + n_o(t))$$

Plugging this into the details given in Section I-D3 completes the proof of our main result.

IV. Conclusion

The main result of this paper is to show that a natural nonclairvoyant algorithm is bounded-speed bounded-competitive for the objective of flow plus energy on power-heterogeneous processors. This paper is a first step towards determining the theoretically best nonclairvoyant algorithm for scheduling jobs of varying importance on power-heterogeneous processors. The obvious two possible next steps are to either find a scalable algorithm for flow plus energy, or find a bounded-speed bounded-competitive algorithm for weighted flow plus energy. The most obvious way to generalize the algorithm considered in this paper to weighted flow is for each job to get a share of the total speed proportional to its weight. This works for a single processor, but doesn't work for multiprocessors. For example, if you have jobs with weights 10, 5 and 5, and three processors that can only be run at unit speed, then it is not possible for the first job to be run at speed $3 \cdot \left(\frac{10}{20}\right)$.

References

[AF07] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3(4), 2007.

[AWT09] Lachlan L.H. Andrew, Adam Wierman, and Ao Tang. Optimal speed scaling under arbitrary power functions. *SIGMETRICS Performance Evaluation Review*, 37(2):39–41, 2009.

[BC09] Nikhil Bansal and Ho-Leung Chan. Weighted flow time does not admit o(1)-competitive algorithms. In *SODA*, pages 1238–1244, 2009.

[BCLL08] Nikhil Bansal, Ho-Leung Chan, Tak Wah Lam, and Lap-Kei Lee. Scheduling for speed bounded processors. In *ICALP (1)*, pages 409–420, 2008.

[BCP09] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Speed scaling with an arbitrary power function. In *SODA*, pages 693–701, 2009.

[BKN] Nikhil Bansal, Ravishankar Krishnaswamy, and Viswanath Nagarjan. Better scalable algorithms for broadcast scheduling. submitted for publication.

[BL04] 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.

[BPS09] Nikhil Bansal, Kirk Pruhs, and Clifford Stein. Speed scaling for weighted flow time. *SIAM Journal on Computing*, 39(4), 2009.

[BSC08] Fred A. Bower, Daniel J. Sorin, and Landon P. Cox. The impact of dynamically heterogeneous multicore processors on thread scheduling. *IEEE Micro*, 28(3):17–25, 2008.

[BT06] Carl Bussema and Eric Torng. Greedy multiprocessor server scheduling. *Oper. Res. Lett.*, 34(4):451–458, 2006.

[CEL$^+$09] Ho-Leung Chan, Jeff Edmonds, Tak Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. In *STACS*, pages 255–264, 2009.

[CEP09] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. In *SPAA*, pages 1–10, 2009.

[CGKM09] Jivitej S. Chadha, Naveen Garg, Amit Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation. In *STOC*, pages 679–684, 2009.

[EP09] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *SODA*, pages 685–692, 2009.

[GKP] Anupam Gupta, Ravishankar Krishnaswamy, and Kirk Pruhs. Scalably scheduling power-heterogeneous processors. submitted for publication.

[KP00] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[KTJ06] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *International conference on parallel architectures and compilation techniques*, pages 23–32. ACM, 2006.

[KTR$^+$04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Computer Architecture News*, 32(2):64, 2004.

[LLTW08a] Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Competitive non-migratory scheduling for flow time and energy. In *SPAA*, pages 256–264, 2008.

[LLTW08b] Tak Wah Lam, Lap-Kei Lee, Isaac Kar-Keung To, and Prudence W. H. Wong. Speed scaling functions for flow time scheduling based on active job count. In *European Symposium on Algorithms*, pages 647–659, 2008.

[LR07] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *Journal of Computer and Systems Sciences*, 73(6):875–891, 2007.

[Mer08] Rick Merritt. CPU designers debate multi-core future. *EE Times*, February 2008.

[MWK$^+$06] Tomer Y. Morad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):4, 2006.

[Pru07] Kirk Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Performance Evaluation Review*, 34(4):52–58, 2007.

[PST04] Kirk Pruhs, Jiri Sgall, and Eric Torng. Online scheduling. In *Handbook on Scheduling*. CRC Press, 2004.

[PUW08] Kirk Pruhs, Patchrawat Uthaisombut, and Gerhard J. Woeginger. Getting the best response for your erg. *ACM Transactions on Algorithms*, 4(3), 2008.