Provably Efficient Adaptive Scheduling For Parallel Jobs

Yuxiong HE[1], Wen Jing HSU[1], Charles E. LEISERSON[2]
[1] Nanyang Technological University
[2] Massachusetts Institute of Technology

***Abstract*— Scheduling competing jobs on multiprocessors has always been an important issue for parallel and distributed systems. The challenge is to ensure global, system-wide efficiency while offering a level of fairness to user jobs. Various degrees of successes have been achieved over the years. However, few existing schemes address both efficiency and fairness over a wide range of work loads. Moreover, in order to obtain analytical results, most of them require prior information about jobs, which may be difficult to obtain in real applications.**

This paper presents two novel adaptive scheduling algorithms – GRAD for centralized scheduling, and WRAD for distributed scheduling. Both GRAD and WRAD ensure fair allocation under all levels of workload, and they offer provable efficiency without requiring prior information of job's parallelism. Moreover, they provide effective control over the scheduling overhead and ensure efficient utilization of processors. To the best of our knowledge, they are the first non-clairvoyant scheduling algorithms that offer such guarantees. We also believe that our new approach of resource request-allotment protocol deserves further exploration.

Specifically, both GRAD and WRAD are $O(1)$-competitive with respect to mean response time for batched jobs, and $O(1)$-competitive with respect to makespan for non-batched jobs with arbitrary release times. The simulation results show that, for non-batched jobs, the makespan produced by GRAD is no more than 1.39 times of the optimal on average and it never exceeds 4.5 times. For batched jobs, the mean response time produced by GRAD is no more than 2.37 times of the optimal on average, and it never exceeds 5.5 times.

***Index Terms*— Adaptive scheduling, Competitive analysis, Data-parallel computing, Greedy scheduling, Instantaneous parallelism, Job scheduling, Makespan, Mean response time, Multiprocessing, Multiprogramming, Parallelism feedback, Parallel computation, Processor allocation, Span, Thread scheduling, Two-level scheduling, Space sharing, Trim analysis, Work, Work-stealing.**

I. Introduction

Parallel computers are expensive resource that often must be shared among a large community of users. One major issue of parallel job scheduling is how to efficiently share resources of multiprocessors among a number of competing jobs, while ensuring each job a required quality of services (see e.g. [6], [7], [9], [11], [14], [16]–[19], [24], [27], [29], [31]–[35], [37], [43], [44]). Efficiency and fairness are two important design goals, where the efficiency is often quantified in terms of makespan and mean response time. This paper summaries several scheduling algorithms we developed. For scheduling of individual jobs, our algorithms ensure short completion time and small waste, for scheduling of job sets, they offer provable efficiency in terms of the makespan and mean response time by allotting each job a fair share of processor resources. Moreover, our algorithms are ***non-clairvoyant*** [9], [14], [16], [24], i.e. they assume nothing about the release time, the execution time, and the parallelism profile of jobs.

Parallel job scheduling can be implemented using a two-level framework [19]: a kernel-level ***job scheduler*** which allots processors to jobs, and a user-level ***thread scheduler*** which maps the threads of a given job to the allotted processors. The job schedulers may implement either ***space-sharing***, where jobs occupy disjoint processor resources, or ***time-sharing***, where different jobs may share the same processor resources at different times. Moreover, both the thread scheduler and the job scheduler may be either ***adaptive***, allowing the number of processors allotted to a job to vary while the job is running, or ***nonadaptive*** (called "static" in [12]), where a job runs on a fixed number of processors over its lifetime. Our schedulers apply two-level structure in the context of adaptive scheduling.

With ***adaptive*** scheduling [4] (called "dynamic" scheduling in [19], [30], [32], [46], [47]), the job scheduler can change the number of processors allotted to a job while the job executes. Thus, new jobs can enter the system, because the job scheduler can simply recruit processors from the already executing jobs and allot them to the new jobs. Without a suitable feedback mechanism, however, both adaptive and nonadaptive schedulers may waste processor cycles, because a job with low parallelism may be allotted more processors than it can productively use. If individual jobs provide proper ***parallelism feedback*** to the job scheduler, waste can be reduced. Therefore, at regular intervals (called quanta), a thread scheduler estimates the desire and provides it to the job scheduler; the job scheduler allots the processors to the jobs based on the request. This feedback mechanism is called ***request-allotment*** protocol. Since the future parallelism of jobs is generally unknown, the challenge here is to develop a request-allotment protocol, which gives an effective way to estimate desire and allocate processors.

Various researchers [13], [14], [22], [32] have used the notion of instantaneous parallelism — the number of processors the job can effectively use at the current

This research was supported in part by the Singapore-MIT Alliance, and NSF Grants ACI-0324974 and CNS-0305606.

moment, as the parallelism feedback to the job scheduler. Although using instantaneous parallelism for parallelism feedback is simple, it is only applicable to the situations where the parallelism of jobs does not change as frequent as the rate of processor reallocation. When jobs' parallelism change fast, using instantaneous parallelism as feedback can cause gross misallocation of processor resources [38]. For example, the parallelism of a job may change substantially during a scheduling quantum, alternating between parallel and serial phases. The sampling of instantaneous parallelism at a scheduling event between quanta may lead the task scheduler to request either too many or too few processors depending on which phase is currently active, whereas the desirable request might be something in between. Consequently, the job may waste processor cycles or take too long to complete.

For thread scheduling, this paper makes use of two adaptive thread schedulers A-GREEDY [1] and A-STEAL [2], [3], which provide parallelism feedback. A-GREEDY is a greedy thread scheduler suitable for centralized scheduling, where each job's thread scheduler can dispatch all the ready threads to the allotted processors in a centralized manner, e.g. the scheduling of data-parallel jobs. A-STEAL is a distributed thread scheduler, where each job is executed by decentralized work-stealing [8], [10], [23]. A-GREEDY and A-STEAL were originally presented in our joint work [1]–[3] with Kunal Agrawal from MIT. Both of them guarantee not to waste many processor cycles while simultaneously ensuring that the job completes quickly. Instead of using instantaneous parallelism, A-GREEDY and A-STEAL provide parallelism feedback to the job scheduler based on a single summary statistic and the job's behavior on the previous quantum. Even though they provide parallelism feedback using the past behavior of the job and we do not assume that the job's future parallelism is correlated with its history of parallelism, our analysis shows that they schedule the job well with respect to both waste and completion time.

For job scheduling, this paper also introduces an "adaptive" job scheduler - RAD, which combines the space-sharing job scheduling algorithm "dynamic equipartitioning" [32], [43] (DEQ) with the time-sharing round robin (RR) algorithm. When the total number of jobs is smaller than or equal to the total number of processors, it uses DEQ as job scheduler. DEQ allots each job with an equal number of processors unless the job requests for less. DEQ was introduced by McCann, Vaswani, and Zahorjan [32] based on earlier work on equipartitioning by Tucker and Gupta [43]. When the total number of jobs is greater than the total number of processors, RAD applies time-sharing round robin algorithm, which also ensures that each job gets an equal slice of scheduling time. RAD was originally presented in our paper [25].

Intuitively, if each job provides good parallelism feedback and makes productive use of available processors, a good job scheduler can ensure that *all* the jobs perform well. To affirm this intuition, we combine the thread schedulers A-GREEDY and A-STEAL with the job scheduler RAD, and get two adaptive two-level scheduler — GRAD and WRAD correspondingly [24], [25]. GRAD, which couples RAD with A-GREEDY, is suitable for centralized thread scheduling. WRAD, which couples RAD with A-STEAL, is more suitable for scheduling in the distributed manner.

Based on the "equalized allotment" scheme for processor allocation, and by using the utilization in the past quantum as feedback, we show that our two-level schedulers are provably efficient. The performance is measured in terms of both makespan and mean response time. Both GRAD and WRAD achieves $O(1)$-competitiveness with respect to makespan for job sets with arbitrary release times, and $O(1)$-competitiveness with respect to mean response time for batched job sets where all jobs are released simultaneously. Unlike many previous results, which either assume clairvoyance [11], [26], [27], [31], [34], [37], [44] or use instantaneous parallelism [9], [14], our schedulers remove these restrictive assumptions. Moreover, because the quantum length can be adjusted to amortize the cost of context-switching during processor reallocation, our schedulers provide effective control over the scheduling overhead and ensures efficient utilization of processors.

Our simulation results also suggest that GRAD performs well in practice [1]. For job sets with arbitrary release time, their makespan scheduled by GRAD is no more than 1.39 times of the optimal on average and it never exceeds 4.5 times. For batched job sets, their mean response time scheduled by GRAD is no more than 2.37 times of the optimal on average, and it never exceeds 5.5 times.

The remainder of this paper is organized as follows. Section II describes the job model, scheduling model, and objective functions. Section III describes the thread scheduling and job scheduling algorithms of GRAD. Section IV and Section V analyze the theoretical performance and present the empirical results for GRAD respectively. Section VI presents a distributed two-level adaptive scheduling algorithm WRAD, and states its performance. Section VII discusses the related work, and Section VIII concludes the paper by raising issues for future research.

II. Scheduling Model and Objective Functions

Our scheduling problem consists of a collection of independent jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_{|\mathcal{J}|}\}$ to be scheduled on a collection of P identical processors. In this section, we formalize the job model, define the scheduling model, and present the optimization criteria of makespan and mean response time.

[1]The experimental study of WRAD is still in progress, and therefore we only present the simulation results of GRAD in the paper.

Job Model

We model the execution of a multithreaded job J_i as a dynamically unfolding directed acyclic graph (dag) such that $J_i = (V_i, E_i)$ where V_i and E_i represent the sets of J_i's vertices and edges respectively. Each vertex $v \in V_i$ represents a unit-time instruction. The ***work*** $T_1(J_i)$ of the job J_i corresponds to the total number of vertices in the dag, i.e. $T_1(J_i) = |V_i|$. Each edge $e \in E_i$ from vertex u to v represents a dependency between the two vertices. The precedence relationship $u \prec v$ holds if and only if there exists a path from vertex u to v in E_i. The ***span*** $T_\infty(J_i)$ corresponds to the number of nodes on the longest chain of the precedence dependencies. The ***release time*** $r(J_i)$ of the job J_i is the time at which J_i becomes first available for processing. Each job is handled by a dedicated thread scheduler, which operates in an online manner, oblivious to the future characteristics of the dynamically unfolding dag.

Scheduling Model

Our scheduling model assumes that time is broken into a sequence of equal-sized ***scheduling quanta*** $1, 2, \ldots$, each of length L, where each quantum q includes the interval $[L{\cdot}q, L{\cdot}q{+}1, \ldots, L(q{+}1){-}1]$ of time steps. The quantum length L is a system configuration parameter chosen to be long enough to amortize scheduling overheads.

The job scheduler and thread schedulers interact as follows. The job scheduler may reallocate processors between quanta. Between quantum $q-1$ and quantum q, the thread scheduler of a given job J_i determines the job's ***desire*** $d(J_i, q)$, which is the number of processors J_i wants for quantum q. Based on the desire of all running jobs, the job scheduler follows its processor-allocation policy to determine the ***allotment*** $a(J_i, q)$ of the job with the constraint that $a(J_i, q) \leq d(J_i, q)$. Once a job is allotted its processors, the allotment does not change during the quantum.

A schedule $\chi = (\tau, \pi)$ of a job set $\mathcal{J}$ is defined as two mappings $\tau : \cup_{J_i \in \mathcal{J}} V_i \rightarrow \{1, 2, \ldots, \infty\}$, and $\pi : \cup_{J_i \in \mathcal{J}} V_i \rightarrow \{1, 2, \ldots, P\}$, which map the vertices of the jobs in the job set $\mathcal{J}$ to the set of time steps, and the set of processors on the machine respectively. A valid mapping must preserve the precedence relationship of each job. For any two vertices $u, v \in V_i$ of the job J_i, if $u \prec v$, then $\tau(u) < \tau(v)$, i.e. the vertex u must be executed before the vertex v. A valid mapping must also ensure that one processor can only be assigned to one job at any given time. For any two vertices u and v, both $\tau(u) = \tau(v)$ and $\pi(u) = \pi(v)$ are true iff $u = v$.

Objective Functions

Our scheduler uses makespan and mean response time as the performance measurement, which are defined as follows.

Definition 1: The ***completion time*** $T_\chi(J_i)$ of a job J_i under a schedule χ is the time at which the schedule completes the execution of the job, i.e. $T_\chi(J_i) = \max_{v \in V_i} \tau(v)$. The ***makespan*** of a given job set $\mathcal{J}$ under the schedule χ is the time taken to complete all jobs in the job set $\mathcal{J}$, i.e. $\mathrm{T}_\chi(\mathcal{J}) = \max_{J_i \in \mathcal{J}} T_\chi(J_i)$.

Definition 2: The ***response time*** of a job J_i under a schedule χ is $T_\chi(J_i) - r(J_i)$, which is the duration between its release time $r(J_i)$ and the completion time $T_\chi(J_i)$. The ***total response time*** of a job set $\mathcal{J}$ under a schedule χ is given by $\mathrm{R}_\chi(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} (T_\chi(J_i) - r(J_i))$ and the ***mean response time*** is $\overline{\mathrm{R}}_\chi(\mathcal{J}) = \mathrm{R}_\chi(\mathcal{J}) / |\mathcal{J}|$.

The goal of the paper is to show that our scheduler optimizes the makespan and mean response time, and we use competitive analysis as a tool to evaluate and compare the scheduling algorithm. The competitive analysis of an online scheduling algorithm is to compare the algorithm against an optimal clairvoyant algorithm. Let $\mathrm{T}^*(\mathcal{J})$ denote the makespan of the jobset $\mathcal{J}$ scheduled by an optimal scheduler, and $\chi(A)$ denote the schedule produced by an algorithm A for the job set $\mathcal{J}$. A deterministic algorithm A is said to be ***c-competitive*** if there exists a constant b such that $\mathrm{T}_{\chi(A)}(\mathcal{J}) \leq c \cdot \mathrm{T}^*(\mathcal{J}) + b$ holds for the schedule $\chi(A)$ of each job set. We will show that our algorithm is c-competitive in terms of the makespan, where c is a small constant. Similarly, let $\mathrm{R}_{\chi(A)}(\mathcal{J})$ and $\overline{\mathrm{R}}_{\chi(A)}(\mathcal{J})$ denote the total response time and mean response time in terms of the schedule $\chi(A)$ produced by an algorithm A. For the mean response time, we will show that our algorithm is also constant-competitive for any batched jobs.

III. GRAD Algorithms

GRAD uses A-GREEDY as thread scheduler, and RAD as job scheduler. We present these two algorithms in this section.

A-GREEDY Thread Scheduler

A-GREEDY [1] is an adaptive greedy thread scheduler with parallelism feedback. Between quanta, it estimates its job's desire, and requests processors from the job scheduler. During the quantum, it schedules the ready threads of the job onto the allotted processors greedily [7], [21]. For a job J_i, if there are more than $a(J_i, q)$ ready threads, A-GREEDY schedules any $a(J_i, q)$ of them. Otherwise, it schedules all of them.

A-GREEDY classifies quanta as "satisfied" versus "deprived" and "efficient" versus "inefficient." A quantum q is ***satisfied*** if $a(J_i, q) = d(J_i, q)$, in which case J_i's allotment is equal to its desire. Otherwise, the quantum is ***deprived***. The quantum q is ***efficient*** if A-GREEDY's utilization $u(J_i, q)$ is no less than a δ fraction of the total allotted processor cycles during the quantum, where δ is named as ***utilization parameter***. Typical values for δ might be 90–95%. Otherwise, the quantum is ***inefficient***.

A-GREEDY calculates the desire $d(J_i, q)$ of the current quantum q based on the previous desire $d(J_i, q-1)$ and the three-way classification of quantum $q-1$ as inefficient, efficient and satisfied, and efficient and deprived.

```
A-GREEDY(q, δ, ρ)
1  if q = 1
2     then d(J_i, q) ← 1                          ▷ base case
3  elseif u(J_i, q−1) < Lδa(J_i, q−1)
4     then d(J_i, q) ← d(J_i, q−1)/ρ              ▷ ineff.
5  elseif a(J_i, q−1) = d(J_i, q−1)
6     then d(J_i, q) ← ρd(J_i, q−1)               ▷ eff.+ sat.
7  else d(J_i, q) ← d(J_i, q−1)                   ▷ eff.+ dep.
8  Report desire d(J_i, q) to the job scheduler.
9  Receive allotment a(J_i, q) from the job scheduler.
10 Greedily schedule on a(J_i, q) processors
   for L time steps.
```

Figure 1: Pseudocode for the adaptive greedy algorithm. A-GREEDY provides parallelism feedback of job J_i to a job scheduler in the form of a desire for processors. Before quantum q, A-GREEDY uses the previous quantum's desire $d(J_i, q-1)$, allotment $a(J_i, q-1)$, and usage $u(J_i, q-1)$ to compute the current quantum's desire d_q based on the utilization parameter δ and the responsiveness parameter ρ.

The initial desire is $d(J_i, 1) = 1$. A-GREEDY uses a ***responsiveness parameter*** $\rho > 1$ to determine how quickly the scheduler responds to changes in parallelism. Typical values of ρ might range between 1.2 and 2.0. Figure 1 shows the pseudo-code of A-GREEDY for one quantum.

RAD Job Scheduler

The job scheduler RAD [25] unifies the space-sharing job scheduling algorithm DEQ [32], [43] with the time-sharing RR algorithm. When the number of jobs is greater than the number of processors, GRAD schedules the jobs in batched round robin fashion, which allocates one processor to each job with an equal share of time. When the number of jobs is at most the number of processors, GRAD uses DEQ as job scheduler. DEQ gives each job an equal share of spatial allotments unless the job requests for less.

When jobs are scheduled in batched round robin fashion, RAD maintains a queue of jobs. At the beginning of each quantum, if there are more than P jobs, it pops P jobs from the top of the queue, and allots one processor to each of them during the quantum. At the end of the quantum, RAD pushes the P jobs back to the bottom of the queue if they are incomplete. The new jobs can be put into the queue once they are released.

DEQ attempts to give each job a fair share of processors. If a job requires less than its fair share, however, DEQ distributes the extra processors to the other jobs. More precisely, upon receiving the desires $\{d(J_i, q)\}$ from the thread schedulers of all jobs $J_i \in \mathcal{J}$, DEQ executes the following ***processor-allocation algorithm***:

1. Set $n = |\mathcal{J}|$. If $n = 0$, return.
2. If the desire for every job $J_i \in \mathcal{J}$ satisfies $d(J_i, q) \geq P/n$, assign each job $a(J_i, q) = P/n$ processors.
3. Otherwise, let $\mathcal{J}' = \{J_i \in \mathcal{J} : d(J_i, q) < P/n\}$. Assign $a(J_i, q) = d(J_i, q)$ processors to each $J_i \in \mathcal{J}'$. Update $\mathcal{J} = \mathcal{J} - \mathcal{J}'$, and $P = P - \sum_{J_i \in \mathcal{J}'} d(J_i, q)$. Go to Step 1.

IV. GRAD Theoretical Results

GRAD is a two-level scheduler, whose performance is usually measured in terms of the global properties such as makespan and mean response time. Intuitively, if each job provides good parallelism feedback and makes productive use of available processors, a good job scheduler can ensure that all the jobs perform well. Therefore, the efficiency of GRAD depends on the effectiveness of A-GREEDY. Specially, we want individual jobs scheduled by A-GREEDY to have short completion time and small waste. In this section, we first introduce A-GREEDY's time and waste bound, then we analyze the performance of GRAD in terms of both makespan and mean response time.

A-GREEDY's Time and Waste Bound

A-GREEDY is a thread scheduler responsible for scheduling individual job J_i. In an adaptive setting where the number of processors allotted to a job can change during execution, both $T_1(J_i)/\overline{P}$ and $T_\infty(J_i)$ are lower bounds on the running time, where $\overline{P}(J_i)$ is the mean of the processor availability for job J_i during the computation. An adversarial job scheduler, however, can prevent any thread scheduler from providing good speedup with respect to the mean availability $\overline{P}(J_i)$ in the worst case. For example, if the adversary chooses a huge number of processors for the job's processor availability just when the job has little instantaneous parallelism, no adaptive scheduling algorithm can effectively utilize the available processors on that quantum.

We introduce a technique called ***trim analysis*** to analyze the time bound of adaptive thread schedulers under these adversarial conditions. Trim analysis borrows the idea of ignoring a few "outliers" from statistics. A ***trimmed mean***, for example, is calculated by discarding a certain number of lowest and highest values and then computing the mean of those that remain. For our purposes, it suffices to trim the availability from just the high side. For a given value R, we define the ***R-high-trimmed mean availability*** as the mean availability after ignoring the R steps with the highest availability. A good thread scheduler should provide linear speedup with respect to an R-trimmed availability, where R is as small as possible.

The following theorem shows that, for each job J_i, A-GREEDY completes the job in $O(T_1(J_i)/\widetilde{P}(J_i) + T_\infty(J_i) + L\lg P)$ time steps, where $\widetilde{P}$ denotes the $O(T_\infty + L\lg P)$-trimmed availability. Thus, job J_i achieves linear speed up with respect to $\widetilde{P}(J_i)$ when $T_1(J_i)/T_\infty(J_i) \gg \widetilde{P}(J_i)$, that is, when its parallelism dominates the $O(T_\infty(J_i) + L\lg P)$-trimmed availability. In addition, we prove that the total number of processor cycles wasted

by the job is $O(T_1(J_i))$, representing at most a constant factor overhead. The details of the proof are documented in [1].

Theorem 1: Suppose that A-GREEDY schedules a job J_i with work $T_1(J_i)$ and critical-path length $T_\infty(J_i)$ on a machine with P processors. If ρ is A-GREEDY's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then A-GREEDY completes the job in

$$T(J_i) \le \frac{T_1(J_i)}{\delta \widetilde{P}(J_i)} + \frac{2T_\infty(J_i)}{1-\delta} + L\log_\rho P + L$$

time steps, where $\widetilde{P}(J_i)$ is the $(2T_\infty(J_i)/(1-\delta) + L\log_\rho P + L)$-trimmed availability. Moreover, A-GREEDY wastes at most

$$(1+\rho-\delta)T_1(J_i)/\delta$$

processor cycles in the course of the computation. □

Makespan

Makespan is the time to complete all jobs in the job set. Given a job set $\mathcal{J}$ and P processors, lower bounds on the makespan of any job scheduler can be obtained based on release time, work, and span. Recall that for a job $J_i \in \mathcal{J}$, the quantities $r(J_i)$, $T_1(J_i)$, and $T_\infty(J_i)$ represent the release time, work, and span of J_i, respectively. Let $\mathrm{T}^*(\mathcal{J})$ denote the makespan produced by an optimal scheduler on a job set $\mathcal{J}$ scheduled on P processors. Let $T_1(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} T_1(J_i)$ denote the total work of the job set. The following two inequalities give two lower bounds on the makespan [9]:

$$\mathrm{T}^*(\mathcal{J}) \ge \max_{J_i \in \mathcal{J}} \{r(J_i) + T_\infty(J_i)\} , \quad (1)$$

$$\mathrm{T}^*(\mathcal{J}) \ge T_1(\mathcal{J})/P . \quad (2)$$

The following theorem bounds the makespan of a job set $\mathcal{J}$ scheduled by GRAD.

Theorem 2: Suppose that GRAD schedules a job set $\mathcal{J}$ on a machine with P processors. It completes the job set in

$$\mathrm{T}(\mathcal{J}) \le \frac{\rho+1}{\delta}\frac{T_1(\mathcal{J})}{P} + \frac{2}{1-\delta}\max_{J_i \in \mathcal{J}}\{T_\infty(J_i) + r(J_i)\} + L\log_\rho P + 2L$$

time steps.

Proof Sketch. Suppose job J_k is the last job completed among the jobs in $\mathcal{J}$. Let $S(J_k)$ denote the set of satisfied steps for J_k, and $D(J_k)$ denote its set of deprived steps. The job J_k is scheduled to start its execution at the beginning of the quantum q where $Lq < r(J_k) \le L(q+1)$, which is the quantum immediately after J_k's release. Therefore, we have $\mathrm{T}(\mathcal{J}) \le r(J_k) + L + |S(J_k)| + |D(J_k)|$.

We now bound $|S(J_k)|$ and $|D(J_k)|$. According to [1], we know that the number of satisfied steps is given by $|S(J_k)| \le 2T_\infty(J_k)/(1-\delta) + L\log_\rho P + L$. To bound the total number of deprived steps, we use the work-conservative property of RAD to make a connection between the total allotment with the total work of the job set. The key observation is that RAD must have allotted all processors to jobs whenever J_k is deprived. Once we get $|D(J_k)|$, a simple summation gives us the desired bound. □

Since both the quantum length L and the number of processors P are independent variables with respect to any job set $\mathcal{J}$, and both $T_1(\mathcal{J})/P$ and $\max_{J_i \in \mathcal{J}}\{T_\infty(J_i) + r(J_i)\}$ are lower bounds of $\mathrm{T}^*(\mathcal{J})$, GRAD is $O(1)$-competitive with respect to makespan. Specially, when $\delta = 0.5$ and ρ approaches 1, the competitiveness ratio $(\rho+1)/\delta + 2/(1-\delta)$ approaches its minimum value 8. Thus, GRAD is $(8+\epsilon)$-competitive with respect to makespan for any constant $\epsilon > 0$.

Mean Response Time

Mean response time is an important measure for multiuser environments where we desire as many users as possible to get fast response from the system. To introduce the lower bounds of mean response time for batched jobs, we need to introduce two definitions – squashed work area and aggregate span. Consider the jobs in the job set $\mathcal{J}$ are ordered according to their work, i.e. $T_1(J_1) \le T_1(J_2) \le \cdots \le T_1(J_{|\mathcal{J}|})$. The ***squashed work area*** of $\mathcal{J}$ is

$$\mathrm{swa}(\mathcal{J}) = (1/P)\sum_{i=1}^{n}(n-i+1)T_1(J_i) .$$

The ***aggregate span*** of $\mathcal{J}$ is

$$T_\infty(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} T_\infty(J_i) ,$$

where $T_\infty(J_i)$ is the span of job $J_i \in \mathcal{J}$. The research in [14], [44], [45] establishes two lower bounds for the mean response time:

$$\overline{\mathrm{R}^*}(\mathcal{J}) \ge T_\infty(\mathcal{J})/|\mathcal{J}| , \quad (3)$$

$$\overline{\mathrm{R}^*}(\mathcal{J}) \ge \mathrm{swa}(\mathcal{J})/|\mathcal{J}| , \quad (4)$$

where $\overline{\mathrm{R}^*}(\mathcal{J})$ denotes the mean response time of $\mathcal{J}$ scheduled by an optimal clairvoyant scheduler. Both the aggregate span $T_\infty(\mathcal{J})$ and the squashed work area $\mathrm{swa}(\mathcal{J})$ are lower bounds for the total response time $\mathrm{R}^*(\mathcal{J})$ under an optimal clairvoyant scheduler.

Theorem 3: Suppose that a job set $\mathcal{J}$ is scheduled by GRAD on P processors. Let $C = 2 - 2/(|\mathcal{J}|+1)$. The total response time $\mathrm{R}(\mathcal{J})$ of the schedule is at most

$$\mathrm{R}(\mathcal{J}) = C\left(\frac{\rho+1}{\delta}\mathrm{swa}(\mathcal{J}) + \frac{2}{1-\delta}T_\infty(\mathcal{J})\right) + O(|\mathcal{J}|L\log_\rho P) , \quad (5)$$

where $\mathrm{swa}(\mathcal{J})$ is the squashed work area of $\mathcal{J}$, and $T_\infty(\mathcal{J})$ is the aggregate span of $\mathcal{J}$.

Proof Sketch. The proof of the competitiveness of mean response time is more complex than that of makespan. The analysis can be divided into two parts. In the first

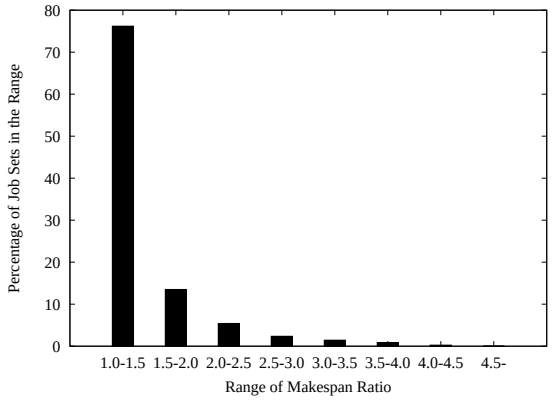

Figure 2: Comparing the makespan of GRAD with the theoretical lower bound for job sets with arbitrary job release time.

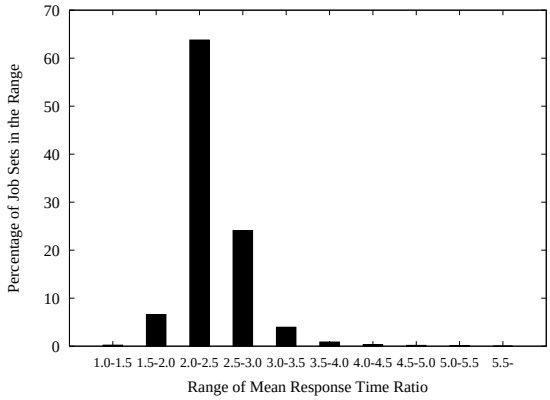

Figure 3: Comparing the mean response time of GRAD with the theoretical lower bound for batched job sets.

part where $|\mathcal{J}| \leq P$, GRAD always uses DEQ as job scheduler. In this case, we use mathematical induction to show local c-competitiveness argument, which asserts Inequality (5) is always true at any time step t during the execution of the job set. In the second part of the proof where $|\mathcal{J}| > P$, GRAD uses both RR and DEQ. Since we consider batched jobs, the number of incomplete jobs decreases monotonically. When the number of incomplete jobs drops to P, GRAD switches its job scheduler from RR to DEQ. Therefore, we prove the second case based on the properties of round robin scheduling and the results of the first case. □

Since both $\mathrm{swa}(\mathcal{J}) / |\mathcal{J}|$ and $T_\infty(\mathcal{J}) / |\mathcal{J}|$ are lower bounds on $\overline{\mathrm{R}}(\mathcal{J})$, we obtain the following corollary.

Corollary 4: Suppose that a job set $\mathcal{J}$ is scheduled by GRAD. The mean response time $\overline{\mathrm{R}}(\mathcal{J})$ of the schedule satisfies

$$\overline{\mathrm{R}}(\mathcal{J}) = \left(2 - \frac{2}{|\mathcal{J}|+1}\right)\left(\frac{\rho+1}{\delta} + \frac{2}{1-\delta}\right)\overline{\mathrm{R}^*}(\mathcal{J}) + O(L\log_\rho P) ,$$

where $\overline{\mathrm{R}^*}(\mathcal{J})$ denotes the mean response time of $\mathcal{J}$ scheduled by an optimal clairvoyant scheduler. □

Since both the quantum length L and the number of processors P are independent variables with respect to any job set $\mathcal{J}$, Corollary 4 shows that GRAD is $O(1)$-competitive with respect to mean response time for batched jobs. Specifically, when $\delta = 1/2$ and ρ approaches 1, GRAD's competitiveness ratio approaches the minimum value 16. Thus, GRAD is $(16+\epsilon)$-competitive with respect to mean response time for any constant $\epsilon > 0$.

V. GRAD Experimental Results

GRAD's competitive ratio with respect to makespan and mean response time, though asymptotically strong, has a relatively large constant multiplier. Our experiments were designed to evaluate the constants that would occur in practice and compare GRAD to an optimal scheduler. We build a Java-based discrete-time simulator using DESMO-J [15]. Our simulator models four major entities — processors, jobs, thread schedulers, and job schedulers, and simulates their interactions in a two-level scheduling environment. The simulator operates in discrete time steps, and we ignore the overheads incurred in the reallocation of processors.

Our benchmark application is the Fork-Join jobs, which alternate between serial and parallel phases. Fork-Join jobs arise naturally in jobs that exhibit "data parallelis", meaning those that apply the same computation to a number of different data points. Many computationally intensive applications can be expressed in a data-parallel fashion [36]. The repeated fork-join cycle in the job reflects the often iterative nature of these computations. We generate jobs with different work, spans, and phase lengths. The experiments are conducted on more than 10000 runs of jobs sets using many combinations of jobs and different loads.

Figure 2 shows how GRAD performs compared to an optimal scheduler with respect to makespan. The makespan of a job set $\mathcal{J}$ has two lower bounds $\max_{J_i \in \mathcal{J}}(r(J_i)+T_\infty(J_i))$ and $T_1(\mathcal{J})/P$. The makespan produced by an optimal scheduler is at least the larger of these two lower bounds. The makespan ratio in Figure 2 is defined as the makespan of a job set scheduled by GRAD divided by the large of the two lower bounds. In Figure 2, the X-axis represents the ranges of the makespan ratio, while the histogram shows the percentage of the job sets whose makespan ratio falls into each range. Among more than 10000 runs, 76.19% of them use less than 1.5 times of the theoretical lower bound, 89.70% uses less than 2.0 times, and none uses more than 4.5 times. The average makepsan ratio is 1.39, which suggests that in practice GRAD has a small competitive ratio with respect to the makespan.

Figure 3 shows the distribution of the mean response

time normalized w.r.t the larger of the two lower bounds – the squashed work bound $\mathrm{swa}(\mathcal{J})/|\mathcal{J}|$ and the aggregated critical path bound $T_\infty(\mathcal{J})/|\mathcal{J}|$. The histogram in Figure 3 shows that, among more than 8000 runs, 94.65% of them use less than 3 times of the theoretical lower bound, and none of them uses more than 5.5 times. The average mean response time ratio is 2.37.

We now interpret the relation between the theoretical bounds and experimental results as follows. When $\rho = 2$ and $\delta = 0.8$, from Theorem 2, GRAD is 13.75-competitive in the worst case. However, we anticipate that GRAD's makespan ratio would be small in practical settings, especially when many jobs have total work much larger than span and the machine is moderately or highly loaded. In this case, the term on $T_1(\mathcal{J})/P$ in Inequality (3) of Theorem 2 is much larger than the term $\max_{J_i \in \mathcal{J}} \{T_\infty(i) + r(i)\}$, which is to say, the term on $T_1(\mathcal{J})/P$ generally dominates the makespan bound. The proof of Theorem 2 calculates the coefficient of $T_1(\mathcal{J})/P$ as the ratio of the total allotment (total work plus total waste) versus the total work. When the job scheduler is RAD, which is not a true adversary, our simulation results indicate that the ratio of the waste versus the total work is only about $1/10$ of the total work. Thus, the coefficient of $T_1(\mathcal{J})/P$ in Inequality (3) is about 1.1. It explains that the makespan produced by GRAD is less than 2-times of the lower bound in average as shown in Figure 2.

Similar to makepsan, we can relate the theoretical mean resposne time bounds with experimental results as follows. When $\rho = 2$ and $\delta = 0.8$, from Theorem 3, GRAD is 27.60-competitive. However, we expect that GRAD should perform closer to optimal in practice. In particular, when the job set $\mathcal{J}$ exhibits reasonably large total parallelism, we have $\mathrm{swa}(\mathcal{J}) \gg T_\infty(\mathcal{J})$, and thus, the term involving $\mathrm{swa}(\mathcal{J})$ in Theorem 3 dominates the total response time. More importantly, RAD is not an adversary of A-GREEDY, as mentioned before, the waste of a job is only about $1/10$ of the total work in average for over $100,000$ job runs we tested. Based on this waste, the squashed area bound $\mathrm{swa}(\mathcal{J})$ in Inequality (5) of Theorem 3 has a coefficient to be around 2.2. It explains that the mean response time produced by GRAD is less than 3 times of the lower bound as shown in Figure 3.

VI. WRAD – Distributed Adaptive Scheduler

WRAD is a distributed two-level adaptive scheduler that uses the A-STEAL algorithm [2], [3] as its thread scheduler and RAD as its job scheduler. While A-GREEDY uses a centralized algorithm to schedule tasks on the allotted processors, our new thread scheduling algorithm A-STEAL works in a decentralized fashion, using randomized work-stealing [4], [8], [20] to schedule the threads on allotted processors. A-STEAL is unaware of all available threads at a given moment. Whenever a processor runs out of work, it "steals" work from another processor chosen at random. To the best of our knowledge, A-STEAL is the first work-stealing thread scheduler that provides provably effective parallelism feedback to a job scheduler. In this section, we briefly review A-STEAL, and present WRAD's results.

A-STEAL is a decentralized adaptive thread scheduler with parallelism feedback, and like A-GREEDY, A-STEAL performs two functions. Between quanta, it estimates its job's desire and requests processors from the job scheduler. A-STEAL applies the same desire-estimation algorithm as A-GREEDY to calculate its job's desire. During the quantum, A-STEAL schedules the ready threads of the job onto the allotted processors using an adaptive work-stealing algorithm. For a job J_i, A-STEAL guarantees linear speedup with respect to $O(T_\infty(J_i) + L\lg P)$-trimmed availability. In addition, A-STEAL wastes at most $O(T_1(J_i))$ processor cycles. The precise statements of A-STEAL's time and waste bound are given by the following theorem. Please refer to [2] for its proof.

Theorem 5: Suppose that A-STEAL schedules a job J_i with work $T_1(J_i)$ and critical-path length $T_\infty(J_i)$ on a machine with P processors. For any $\epsilon > 0$, with probability at least $1-\epsilon$, A-STEAL completes the job in

$$\begin{aligned} T &\leq \frac{T_1(J_i)}{\delta \widetilde{P}(J_i)} \left(1 + \frac{1+\rho}{L\delta - 1 - \rho}\right) \\ &\quad + O\left(\frac{T_\infty(J_i)}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right) \end{aligned}$$

time steps, where $\widetilde{P}(J_i)$ is the $O(T_\infty(J_i)/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability. Moreover, it wastes at most

$$W \leq \left(\frac{1+\rho-\delta}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right) T_1(J_i)$$

processor cycles in the course of the computation. □

WRAD is $O(1)$-competitive with respect to both makespan and mean response time. The methods used for analyzing WRAD are similar to those for GRAD. However, since A-STEAL and WRAD are randomized scheduling algorithms, we show that the makespan (or the expected mean response time) is within a factor c of that incurred in an optimal clairvoyant algorithm in expectation, not in the worst case. The following theorem presents the makespan, and mean response time bound of WRAD respectively. WRAD is $O(1)$-competitive for both makespan and, in the batch setting, mean response time.

Theorem 6: Suppose that a job set $\mathcal{J}$ is scheduled by WRAD on a machine with P processors. The expected competition of the schedule is given by

$$\begin{aligned} \mathrm{E}[\mathrm{T}(\mathcal{J})] &= \left(\frac{\rho+1}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right) \frac{T_1(\mathcal{J})}{P} \\ &\quad + O\left(\frac{\max_{J_i \in \mathcal{J}} \{r(J_i) + T_\infty(J_i)\}}{1-\delta}\right) \\ &\quad + L\log_\rho P + 2L \end{aligned}$$

time steps. Moreover, let $C = 2 - 2/(|\mathcal{J}| + 1)$. The expected response time of the schedule is given by

$$\mathrm{E}\left[\mathrm{R}(\mathcal{J})\right] = C\left(\frac{\rho+1}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right)\mathrm{swa}\left(\mathcal{J}\right) + O\left(\frac{T_\infty\left(\mathcal{J}\right)}{1-\delta} + |\mathcal{J}|\, L \log_\rho P\right) ,$$

where $\mathrm{swa}\left(\mathcal{J}\right)$ denotes the squashed work area, and $T_\infty\left(\mathcal{J}\right)$ denotes the aggregate span. □

Both GRAD and WRAD follow the same resource request-allotment protocol, which uses jobs' utilization in the past quantum as feedback, and makes fair processor allotments among jobs. Even though their thread scheduler A-GREEDY and A-STEAL apply fairly different ways to schedule ready threads on processors, both GRAD and WRAD ensure constant competitiveness with respect to makespan, and, in batched setting, mean response time. We believe that this new approach of resource request-allotment protocol can be useful in many other resource management problems, and it deserves further exploration.

VII. Related Work

This section discusses related work in job scheduling that minimizes makespan and mean response time. In the offline version of the problem, all the jobs' resource requirements and release times are known in advance. In the online clairvoyant version of the problem, the algorithm knows the resource requirements of a job when it is released, but it must base its decisions only on jobs that have been released. In this paper, we have studied the online nonclairvoyant version of the problem, where the resource requirements and release times are unknown to the scheduling algorithm.

The online nonclairvoyant scheduling of parallel jobs includes the scheduling of a single parallel job, multiple serial jobs, and multiple parallel jobs.

Prior work on scheduling a single parallel job tends to focus on nonadaptive scheduling [6], [8], [21], [35] or adaptive scheduling without parallelism feedback [4]. For jobs whose parallelism is unknown in advance and which may change during execution, nonadaptive scheduling is known to waste processor cycles [41], because a job with low parallelism may be allotted more processors than it can productively use. Moreover, in a multiprogrammed environment, nonadaptive scheduling may not allow a new job to start, because existing jobs may already be using most of the processors. Although adaptive scheduling without parallelism feedback allows jobs to enter the system, jobs may still waste processor cycles if they are allotted more processors than they can use.

Adaptive thread scheduling with parallelism feedback has been studied empirically in [38], [40], [42]. These researchers use a job's history of processor utilization to provide feedback to dynamic-equipartitioning job schedulers. These studies use different strategies for parallelism feedback, and all report better system performance with parallelism feedback than without, but it is not apparent which strategy is superior.

Some researchers [5], [28], [33] have studied the online non-clairvoyant scheduling of serial jobs to minimize the mean response time on single or multiple processors. For jobs with arbitrary release times, Motwani, Phillips, and Torng [33] show that every deterministic algorithm can achieve competitiveness no better than $\Omega(n^{1/3})$, and any randomized algorithm can achieve competitiveness no better than $\Omega(\log n)$ for mean response time. Becchetti and Leonardi [5] present a version of the randomized multilevel feedback algorithm (RMLF) and prove an $O(\log n \log(n/P))$-competitiveness result against any oblivious adversary on a machine with P processors.

Shmoys, Wein and Williamson in [39] study the lower bounds of online nonclairvoyant scheduling of serial jobs with respect to makespan. They show that the competitive ratio is at least $(2-1/P)$ for any preemptive deterministic online algorithm, and at least $(2 - 1/\sqrt{P})$ for any non-preemptive randomized online algorithm with an oblivious adversary.

Adaptive parallel job scheduling has been studied both empirically [29], [32], [43] and theoretically [13], [16], [17], [22], [33]. McCann, Vaswani, and Zahorjan [32] study many different job schedulers and evaluated them on a set of benchmarks. They also introduce the notion of dynamic equipartitioning (DEQ), which gives each job a fair allotment of processors based on the job's request, while allowing processors that cannot be used by a job to be reallocated to other jobs. Brecht, Deng, and Gu [9] prove that DEQ with instantaneous parallelism as feedback is 2-competitive with respect to the makespan. Later, Deng and Dymond [14] prove that DEQ with instantaneous parallelism is also 4-competitive for batched jobs with respect to the mean response time. Since DEQ only addresses the case where there are more processors than active jobs, a scheduling algorithm that uses DEQ as job scheduler, can only be applied to the case where the total number of jobs in the job set is less than or equal to the total number of processors.

VIII. Conclusions

We have presented two new adaptive scheduling algorithms GRAD and WRAD that ensure fair allocation under all levels of workload, and they offer provable efficiency without requiring prior information of job's parallelism. Moreover, they provide effective control over the scheduling overhead and ensure efficient utilization of processors. To the best of our knowledge, GRAD and WRAD are the first non-clairvoyant scheduling algorithms that offer such guarantees.

The request-allotment framework discussed in this paper

can be applied to application-specific schedulers. Here, GRAD combines RAD with A-GREEDY thread scheduler, and WRAD combines RAD with A-STEAL. Analogously, one can develop a two-level scheduler by applying the request-allotment protocol, and application-specific thread schedulers. Such a two-level scheduler may provide both system-wide performance guarantees such as minimal makespan and mean response time, and optimization of individual applications.

References

[1] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive task scheduling with parallelism feedback. In *PPoPP*, pages 100 – 109, New York City, NY, USA, 2006.

[2] K. Agrawal, Y. He, and C. E. Leiserson. Work stealing with parallelism feedback. To appear in PPoPP 2007.

[3] K. Agrawal, Y. He, and C. E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *ICDCS*, pages 19 – 29, Lisboa, Portugal, 2006.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129, Puerto Vallarta, Mexico, 1998.

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

[6] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, Philadelphia, Pennsylvania, 1996.

[7] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[9] T. Brecht, X. Deng, and N. Gu. Competitive dynamic multiprocessor allocation for parallel applications. In *Parallel and Distributed Processing*, pages 448 – 455, San Antonio, TX, 1995.

[10] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA*, pages 187–194, Portsmouth, New Hampshire, 1981.

[11] J. Chen and A. Miranda. A polynomial time approximation scheme for general multiprocessor job scheduling (extended abstract). In *STOC*, pages 418–427, New York, NY, USA, 1999.

[12] S.-H. Chiang and M. K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *JSSPP*, pages 200–223, Honolulu, Hawaii, United States, 1996.

[13] X. Deng and P. Dymond. On multiprocessor system scheduling. In *SPAA*, pages 82–88, Padua, Italy, 1996.

[14] X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *SODA*, pages 159–167, Philadelphia, PA, USA, 1996.

[15] DESMO-J: A framework for discrete-event modelling and simulation. http://asi-www.informatik.uni-hamburg.de/desmoj/.

[16] J. Edmonds. Scheduling in the dark. In *STOC*, pages 179–188, Atlanta, Georgia, United States, 1999.

[17] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6(3):231–250, 2003.

[18] Z. Fang, P. Tang, P.-C. Yew, and C.-Q. Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, 1990.

[19] D. G. Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). Technical report, IBM Research Report RC 19790 (87657) 2nd Revision, 1997.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[21] R. L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal on Applied Mathematics*, pages 17(2):416–429, 1969.

[22] N. Gu. Competitive analysis of dynamic processor allocation strategies. Master's thesis, York University, 1995.

[23] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *LFP*, pages 9–17, Austin, Texas, Aug. 1984.

[24] Y. He, W. J. Hsu, and C. E. Leiserson. Provably efficient two-level adaptive scheduling. In *JSSPP*, Saint-Malo, France, 2006.

[25] Y. He, W. J. Hsu, and C. E. Leiserson. Provably efficient adaptive scheduling through equalized allotments. Unpublished manuscripts, 2007.

[26] K. Jansen and L. Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *SODA*, pages 490–498, Philadelphia, PA, USA, 1999.

[27] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *SPAA*, pages 86–95, New York, USA, 2005.

[28] B. Kalyanasundaram and K. R. Pruhs. Minimizing flow time nonclairvoyantly. *Journal of the ACM*, 50(4):551–567, 2003.

[29] S. T. Leutenegger and M. K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *SIGMETRICS*, pages 226–236, Boulder, Colorado, United States, 1990.

[30] S. Lucco. A dynamic scheduling method for irregular parallel programs. In *PLDI*, pages 200–211, New York, NY, USA, 1992.

[31] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA*, pages 167–176, Philadelphia, PA, USA, 1994.

[32] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[33] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. In *SODA*, pages 422–431, Austin, Texas, United States, 1993.

[34] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA*, pages 23–32, New York, NY, USA, 1999.

[35] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, 1999.

[36] L. S. Nyland, J. F. Prins, A. Goldberg, and P. H. Mills. A design methodology for data-parallel applications. *IEEE Transactions on Software Engineering*, 26(4):293–314, 2000.

[37] U. Schwiegelshohn, W. Ludwig, J. L. Wolf, J. Turek, and P. S. Yu. Smart smart bounds for weighted response time scheduling. *SIAM Journal of Computing*, 28(1):237–253, 1998.

[38] S. Sen. Dynamic processor allocation for adaptively parallel jobs. Master's thesis, Massachusetts Institute of technology, 2004.

[39] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines online. In *FOCS*, pages 131–140, San Juan, Puerto Rico, 1991.

[40] B. Song. Scheduling adaptively parallel jobs. Master's thesis, Massachusetts Institute of Technology, 1998.

[41] M. S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *IPPS*, pages 219–238, Oakland, California, United States, 1995.

[42] K. G. Timothy B. Brecht. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27-28:519–539, 1996.

[43] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *SOSP*, pages 159–166, New York, NY, USA, 1989.

[44] J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu. Scheduling parallelizable tasks to minimize average response time. In *SPAA*, pages 200–209, Cape May, New Jersey, United States, 1994.

[45] J. Turek, U. Schwiegelshohn, J. L. Wolf, and P. S. Yu. Scheduling parallel tasks to minimize average response time. In *SODA*, pages 112–121, Philadelphia, PA, USA, 1994.

[46] P. Yang, D. Desmet, F. Catthoor, and D. Verkest. Dynamic scheduling of concurrent tasks with cost performance trade-off. In *CASES*, pages 103–109, New York, NY, USA, 2000.

[47] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *SIGMETRICS*, pages 214–225, Boulder, Colorado, United States, 1990.