Adaptive Scheduling with Parallelism Feedback

Kunal Agrawal[1] Yuxiong He[2] Wen-Jing Hsu[2] Charles E. Leiserson[1]

[1]Massachusetts Institute of Technology
32 Vassar Street
Cambridge, MA 02139, USA
kunal_ag@mit.edu
cel@mit.edu

[2]Nanyang Technological University
Nanyang Avenue 639798
Singapore
yxhe@mit.edu
hsu@ntu.edu.sg

Abstract

Multiprocessor scheduling in a shared multiprogramming environment can be structured as two-level scheduling, where a kernel-level job scheduler allots processors to jobs and a user-level thread scheduler schedules the work of a job on the allotted processors. In this context, the number of processors allotted to a particular job may vary during the job's execution, and the thread scheduler must adapt to these changes in processor resources. For overall system efficiency, the thread scheduler should also provide parallelism feedback to the job scheduler to avoid allotting a job more processors than it can use productively.

This paper provides an overview of several adaptive thread schedulers we have developed that provide provably good history-based feedback about the job's parallelism without knowing the future of the job. These thread schedulers complete the job in near-optimal time while guaranteeing low waste. We have analyzed these thread schedulers under stringent adversarial conditions, showing that the thread schedulers are robust to various system environments and allocation policies. To analyze the thread schedulers under this adversarial model, we have developed a new technique, called ***trim analysis****, which can be used to show that the thread scheduler provides good behavior on the vast majority of time steps, and performs poorly on only a few. When our thread schedulers are used with dynamic equipartitioning and other related job scheduling algorithms, they are $O(1)$-competitive against an optimal offline scheduling algorithm with respect to both mean response time and makespan for batched jobs and nonbatched jobs, respectively. Our algorithms are the first nonclairvoyant scheduling algorithms to offer such guarantees.*

This research was supported in part by NSF Grants ACI-0324974 and CNS-0615215 and the Singapore-MIT Alliance.

1 Introduction

The scheduling of a collection of parallel jobs onto a multiprocessor is an old and well-studied topic of research [14, 17, 18, 22, 27, 33, 36, 45, 47, 48]. Schedulers for multiprogrammed multiprocessors can be implemented using a two-level strategy [24]: a kernel-level ***job scheduler*** which allots processors to jobs, and a user-level ***thread scheduler*** which schedules the threads belonging to a given job onto the allotted processors. Our research [1–3] has focused on how the thread scheduler for a job can provide provably effective feedback to the job scheduler on the job's parallelism. We also have studied the system behavior of two-level schedulers that employ this kind of adaptive thread scheduler [29, 30]. This paper overviews this research.

Most prior work on thread scheduling for multithreaded jobs deals with ***nonadaptive*** scheduling [6, 7, 9, 13, 26, 40], where the job scheduler allots a fixed number of processors to the job for its entire lifetime. For jobs whose parallelism is unknown in advance and which may change during execution, this strategy may waste processor cycles [45], 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.

With ***adaptive*** scheduling [4] (called "dynamic" scheduling in many papers), the job scheduler can change the number of processors allotted to a job while the job is executing. 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 job. Unfortunately, as with a nonadaptive scheduler, this strategy may cause waste, because a job with low parallelism may still be allotted more processors than it can productively use.

The solutions we have studied [1–3, 29, 30] all employ an adaptive scheduling strategy where the thread scheduler

provides ***parallelism feedback*** to the job scheduler so that when a job cannot use many processors, those processors can be reallotted to jobs with ample need. Based on this parallelism feedback, the job scheduler adaptively changes the allotment of processors according to the availability of processors in the current system environment and the job scheduler's administrative policy.

Various researchers [17, 18, 27, 36, 49] have used the notion of ***instantaneous parallelism***,[1] the number of processors the job can effectively use at the current moment, as the parallelism feedback to the job scheduler. Although using instantaneous parallelism for parallelism feedback is simple, it can cause gross misallocation of processor resources [43]. 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 thread 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 systematically waste processor cycles on the one hand or take too long to complete on the other.

Our studies explore ***history-based strategies*** to provide parallelism feedback, and investigates adaptive scheduling schemes that offer both fairness and provable efficiency without entailing large overhead and requiring prior job information. This paper provides an overview of our adaptive scheduling schemes.

We have developed 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, such as the scheduling of data-parallel jobs. A-STEAL is a distributed thread scheduler, where each job is executed by decentralized work-stealing [9, 15, 28, 41]. 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 only the past behavior of the job, and we assume that the job's future parallelism can be completely uncorrelated with its history of parallelism, our analysis shows that they schedule the job well with respect to both waste and completion time.

By combining the ***dynamic equipartitioning*** job scheduler [36, 47], which uses space sharing, the round-robin job scheduler, which employs time-sharing, we obtain the RAD algorithm [29, 30]. When RAD is used in conjunction with A-GREEDY or A-STEAL, we obtain adaptive two-level schedulers that guarantee both fairness and efficiency. Specifically, these schedulers guarantee $O(1)$-competitiveness with respect to makespan and mean response time for nonbatched and batched jobs, respectively.

The remainder of this paper is organized as follows. Section 2 describes the job model and scheduling model. Section 3 describes the thread scheduling and job scheduling algorithms. Section 4 shows the theoretical and empirical results of our scheduling algorithms. Section 5 summarizes related work, and Section 6 gives conclusion remarks.

2 Scheduling Model

Our scheduling input consists of a collection of independent jobs $\mathcal{J} = \left\{J_1, J_2, \ldots, J_{|\mathcal{J}|}\right\}$ to be scheduled on a collection of P identical processors. 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. In this section, we formalize the job model, and define the scheduling model.

We model the execution of a multithreaded job J_i as a dynamically unfolding directed acyclic graph (***DAG***, for short). Each vertex of the DAG 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. Each edge represents a dependency between the two vertices. 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.

The job scheduler and the thread schedulers interact as follows. The job scheduler may reallocate processors between scheduling 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.

3 Algorithms

In this section, we present the thread scheduling algorithms A-GREEDY and A-STEAL, as well as the job scheduling algorithm RAD. We obtain two-level schedulers GRAD

[1] These researchers actually use the general term "parallelism," but we prefer the more descriptive term.

A-GREEDY(J_i, q, δ, ρ)
1 **if** q is J_i's first quantum
2 **then** $d(J_i, q) \leftarrow 1$ ▷ base case
3 **elseif** $u(J_i, q-1) < L\delta a(J_i, q-1)$
4 **then** $d(J_i, q) \leftarrow d(J_i, q-1)/\rho$ ▷ inefficient
5 **elseif** $a(J_i, q-1) = d(J_i, q-1)$
6 **then** $d(J_i, q) \leftarrow \rho d(J_i, q-1)$ ▷ eff-and-sat
7 **else** $d(J_i, q) \leftarrow d(J_i, q-1)$ ▷ eff-and-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 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.

and WRAD by combining RAD with A-GREEDY and A-STEAL respectively.

The 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 in a greedy fashion [8, 13, 26]. 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 three-way classification of the quantum $q-1$ as inefficient, efficient and satisfied, and efficient and deprived. 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. The algorithm takes as input the utilization parameter δ, and the responsiveness parameter ρ. Intuitively, it operates as follows:

- If quantum $q-1$ was inefficient, A-GREEDY has overestimated the desire. In this case, disregarding to the quantum is satisfied or deprived, A-GREEDY decreases the desire (line 4) for quantum q.
- If quantum $q-1$ was efficient and satisfied, the job has effectively utilized the processors that A-GREEDY requested on its behalf. Thus, A-GREEDY speculates that the job can use more processors and increases the desire (line 6) for quantum q.
- If quantum $q-1$ was efficient but deprived, the job has used all the processors it was allotted, but A-GREEDY had requested more processors for the job than the job actually received from the job scheduler. Since A-GREEDY has no evidence whether the job could have used all the processors requested, it maintains the same desire (line 7) for quantum q.

A-GREEDY is a centralized thread scheduler, however, and although it is suitable for scheduling of, for example, data-parallel jobs, where the central scheduler can be aware of the available work at the current moment, it does not directly extend to decentralized thread scheduling.

The A-STEAL thread scheduler

A-STEAL [2, 3] is a thread scheduler that works in a decentralized fashion, using randomized work-stealing [4, 9, 25] to schedule the threads on allotted processors. Unlike A-GREEDY, it does not need a global view of all the available work to schedule. A-STEAL applies the same desire-estimation algorithm as A-GREEDY to calculate its job's desire.

We now describe A-STEAL's adaptive work-stealing algorithm. Each processor allotted to a job maintains a local ***deque*** (double-ended queue) of those threads that are ready for execution. When the allotment of a job increases, the A-STEAL thread scheduler creates an empty deque for each newly allotted processor. When the allotment decreases, it marks the deques from deallotted processors as ***muggable deques***. Whenever a processor creates new work, it places the work in its local deque. Whenever a processor finishes a piece of work, it looks for new work in its deque. If it looks for work in its deque, but the deque is empty, the processor becomes a ***thief***. The thief first looks around the system for a muggable deque. If one is found, the thief ***mugs*** the deque by taking over the entire deque as its own. Otherwise, it randomly picks a ***victim*** processor and ***steals*** work from the bottom of the victim's deque. If the victim has no available work, the steal is ***unsuccessful***, and the thief continues to steal at random from the other processors until it is ***successful*** and finds work. At all time steps, every processor is either working, stealing, or mugging.

The RAD job scheduler

The job scheduler RAD [30] unifies the space-sharing dynamic-equipartitioning job-scheduling algorithm [36, 47]

with the time-sharing round-robin algorithm. When the number of jobs is greater than the number of processors, RAD 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, RAD uses dynamic equipartitioning to allot processors to jobs. Dynamic equipartitioning gives all jobs the same allotment, unless the job requests less than its fair share, in which case dynamic equipartitioning distributes the unneeded processors to jobs that need them.

4 Statement of Results

This section overviews our theoretical and experimental contributions. We use a new analysis technique, called "trim analysis," to analyze the behavior of adaptive thread schedulers. Using trim analysis, we can prove that both A-GREEDY and A-STEAL complete a job quickly while using the allotted processors efficiently. We also show that the RAD algorithm, when used with A-GREEDY or A-STEAL, is competitive in terms of both makespan for jobs with arbitrary release times and mean response time for batched jobs.

Analysis of A-GREEDY

To make the thread scheduler robust to different system environments and administrative policies, our analysis of A-GREEDY assumes that the job scheduler decides the availability of processors as an adversary. Suppose that A-GREEDY schedules a 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 average 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.

Our studies introduced a technique called ***trim analysis*** to analyze the time bound of adaptive thread schedulers under these adversarial conditions. From the field of statistics, trim analysis borrows the idea of ignoring a few "outliers." 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.

Specifically, our research 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 A-GREEDY algorithm and its performance are documented in [1].

Analysis and simulation of A-STEAL

Like A-GREEDY, we analyze the job completion time produced by A-STEAL by trim analysis. 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.

We have implemented A-STEAL in a simulation environment using the DESMO-J [19] Java simulator. On a large set of jobs running with a variety of availability profiles, our experiments indicate that A-STEAL provides nearly perfect linear speedup when the jobs have ample parallelism. Moreover, A-STEAL typically wastes less than 20% of the allotted processor cycles. We also compared the performance of A-STEAL with the performance of the adaptive scheduler (we call it the ABP scheduler) presented by Arora, Blumofe and Plaxton [4]. We ran single jobs using both A-STEAL and ABP with the same availability profiles. We found that on moderately to heavily loaded large machines, when $\overline{P} \ll P$, A-STEAL paradoxically completes almost all jobs about twice as fast as ABP on average, despite the fact that ABP's allotment on any quantum is never less than A-STEAL's allotment on the same quantum. In most of these job runs, A-STEAL wastes less than 10% of the processor cycles wasted by ABP.

Analysis and simulation of RAD

The efficiency of RAD can be quantified in terms of makespan and mean response time. The ***makespan*** of a job set $\mathcal{J}$is the time to complete all jobs in $\mathcal{J}$. The ***response time*** of a job is the duration between its release time and the completion time. The ***mean response time*** of a job set $\mathcal{J}$is the average response time of all jobs in $\mathcal{J}$.

We use competitive analysis as a tool to evaluate and compare the scheduling algorithm. Competitive analysis compares an on-line scheduling algorithm with an optimal offline algorithm. Let $\mathrm{T}^*(\mathcal{J})$ denote the makespan of an arbitrary jobset $\mathcal{J}$ scheduled by an optimal scheduler, and let $\mathrm{T}(\mathcal{J})$ denote the the makespan produced by an algorithm A for the job set $\mathcal{J}$. An algorithm A is said to be ***c-competitive***

if there exists a constant b such that $\mathrm{T}(\mathcal{J}) \leq c \cdot \mathrm{T}^*(\mathcal{J}) + b$ holds for the schedule of any job set.

Intuitively, if each job provides good parallelism feedback and makes productive use of allotted processors, a good job scheduler can ensure that *all* jobs make good progress. The analysis of RAD confirms this intuition. Based on the "equalized allotment" scheme for processor allocation, and by using the utilization in the past quantum as feedback, we can show that our two-level schedulers are provably efficient in terms of both makespan and mean response time. RAD achieves $O(1)$-competitiveness with respect to makespan for job sets with arbitrary release times. It also achieves $O(1)$-competitiveness with respect to mean response time for batched job sets, where all jobs are released simultaneously. Unlike previous results, [16, 31, 32, 34, 39, 42, 48], our analysis does not assume prior knowledge of jobs' parallelisms. It also does not assume, as in [12, 18], that the parallelism remains relatively constant across a quantum. Since the quantum length can be adjusted to amortize the cost of context-switching during processor reallocation, RAD also provides effective control over the scheduling overhead and ensures efficient utilization of processors.

Our simulation results suggest that RAD performs well in practice. In our experiments, for job sets with arbitrary release times, the average (geometric mean) ratio of RAD's makespan to the optimal makespan was 1.39, and the maximum ratio was less than 4.5. For batched job sets, the average ratio of RAD's mean response time to the optimal mean response time was 2.37, and it the maximum ratio was less than 5.5.

5 Related Work

This section discusses related work on adaptive scheduling of multithreaded jobs. We first discuss adaptive thread schedulers and followed with a brief summary of research on adaptive job schedulers.

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading, primarily by Blumofe and his coauthors [4, 10, 11]. In this work, the thread scheduler schedules threads using a "work-stealing" [9, 37] strategy, but it does not provide feedback about the job's parallelism to the job scheduler. The research in [10, 11] addresses networks of workstations where processors may fail, or they may join and leave a computation while the job is running, showing that work-stealing provides a good foundation for adaptive thread scheduling. In theoretical work, Arora, Blumofe, and Plaxton [4] exhibit a work-stealing thread scheduler that provably completes a job in $O(T_1/\overline{P} + PT_\infty/\overline{P})$ expected time, where $\overline{P}$ is the average number of processor allotted to the job by the job scheduler. Although they provide no bounds on waste, one can prove that their algorithm may waste $\Omega(T_1 + PT_\infty)$ processor cycles in an adversarial setting.

Adaptive thread scheduling without parallelism feedback has also been studied empirically in the context of data-parallel languages [20, 21]. This work focuses on compiler and runtime support for environments where the number of processors changes while the program executes.

Adaptive thread scheduling with parallelism feedback has been studied empirically in [43, 44, 46]. These researchers use a job's history of processor utilization to provide feedback to dynamic-equipartitioning job schedulers. These studies use a variety of 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.

Adaptive job schedulers have been studied empirically [33, 35, 36, 47, 49] and theoretically [5, 17, 22, 23, 27, 38]. McCann, Vaswani, and Zahorjan [36] studied many different job schedulers and evaluated them on a set of benchmarks. They also introduced the notion of dynamic equipartitioning, which gives each job a fair allotment of processors, while allowing processors that cannot be used by a job to be reallocated to other jobs. Their studies indicate that dynamic equipartitioning may be an effective strategy for adaptive job scheduling. Gu [27] proved that dynamic equipartitioning with instantaneous parallelism feedback is 4-competitive with respect to makespan for batched jobs with multiple phases, where the parallelism of the job remains constant during the phase and the phases are relatively long compared with the length of a scheduling quantum. Deng and Dymond [17] proved a similar result for mean response time for multiphase jobs regardless of their arrival times. Song [44] proves that a randomized distributed strategy can implement dynamic equipartitioning. Even though using instantaneous parallelism as feedback is intuitive, it can either cause gross misallocation of processor resources [43] or introduce significant scheduling overhead.

6 Conclusions

Currently, our adaptive multithreading model does not support a full range of multithreaded-programming features — such as I/O, pipelines, master-slaves. We are currently tackling these problems and hope to explore feedback-driven policies for making scheduling and resource-allocation decisions in adaptive environments. Our research to date makes us optimistic that history-based feedback mechanisms can be developed which are theoretically good and provide practically efficient performance.

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. An empirical evaluation of work stealing with parallelism feedback. In *ICDCS*, pages 19 – 29, Lisboa, Portugal, 2006.

[3] K. Agrawal, Y. He, and C. E. Leiserson. Work stealing with parallelism feedback. In *PPoPP*, San Jose, California, Mar. 2007.

[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] N. Bansal, K. Dhamdhere, J. Konemann, and A. Sinha. Nonclairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, 2004.

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *SPAA*, pages 1–12, Santa Barbara, California, 1995.

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

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

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

[10] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *USENIX*, pages 133–147, Anaheim, California, 1997.

[11] R. D. Blumofe and D. S. Park. Scheduling large-scale parallel computations on networks of workstations. In *HPDC*, pages 96–105, San Francisco, California, 1994.

[12] 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.

[13] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, pages 201–206, 1974.

[14] J. L. Bruno, J. Edward G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17(7):382–387, 1974.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and runtime support for programming in adaptive parallel environments. Technical Report CS-TR-3510, University of Maryland, 1995.

[21] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz. Data parallel programming in an adaptive environment. Technical Report CS-TR-CS-TR-3350, University of Maryland, 1994.

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

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

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

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

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

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

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

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

[30] Y. He, W. J. Hsu, and C. E. Leiserson. Provably efficient online non-clairvoyant adaptive scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium*, Long Beach, California, USA, Mar. 2007.

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

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

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

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

[35] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. In *SIGMETRICS*, pages 104–113, Santa Fe, New Mexico, United States, 1988.

[36] 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.

[37] E. Mohr, D. A. Kranz, and J. Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *LFP*, pages 185–197, 1990.

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

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

[40] 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.

[41] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *SPAA*, pages 237–245, Hilton Head, South Carolina, 1991.

[42] 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.

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

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

[45] 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.

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

[47] 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.

[48] 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.

[49] K. K. Yue and D. J. Lilja. Implementing a dynamic processor allocation policy for multiprogrammed parallel applications in the Solaris operating system. *Concurrency and Computation-Practice and Experience*, 13(6):449–464, 2001.