Adaptive Work Stealing with Parallelism Feedback

Kunal Agrawal Yuxiong He * Charles E. Leiserson

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

We present an adaptive work-stealing thread scheduler, A-STEAL, for fork-join multithreaded jobs, like those written using the Cilk multithreaded language or the Hood work-stealing library. The A-STEAL algorithm is appropriate for large parallel servers where many jobs share a common multiprocessor resource and in which the number of processors available to a particular job may vary during the job's execution. A-STEAL provides continual parallelism feedback to a job scheduler in the form of processor requests, and the job must adapt its execution to the processors allotted to it. Assuming that the job scheduler never allots any job more processors than requested by the job's thread scheduler, A-STEAL guarantees that the job completes in near-optimal time while utilizing at least a constant fraction of the allotted processors.

Our analysis models the job scheduler as the thread scheduler's adversary, challenging the thread scheduler to be robust to the system environment and the job scheduler's administrative policies. We analyze the performance of A-STEAL using "trim analysis," which allows us to prove that our thread scheduler performs poorly on at most a small number of time steps, while exhibiting near-optimal behavior on the vast majority. To be precise, suppose that a job has work T_1 and critical-path length T_∞. On a machine with P processors, A-STEAL completes the job in expected $O(T_1/\widetilde{P} + T_\infty + L \lg P)$ time steps, where L is the length of a scheduling quantum and $\widetilde{P}$ denotes the $O(T_\infty + L \lg P)$-trimmed availability. This quantity is the average of the processor availability over all but the $O(T_\infty + L \lg P)$ time steps having the highest processor availability. When the job's parallelism dominates the trimmed availability, that is, $\widetilde{P} \ll T_1/T_\infty$, the job achieves nearly perfect linear speedup. Conversely, when the trimmed mean dominates the parallelism, the asymptotic running time of the job is nearly the length of its critical path.

Categories and Subject Descriptors D.4.1 [*Software*]: Operating Systems - process management; F.2 [*Theory of Computation*]: Analysis of Algorithms and Problem Complexity.

General Terms Algorithms, Performance, Theory.

Keywords Adaptive scheduling, Adversary, Critical path, Multithreaded Languages, Distributed scheduling, Job scheduling, Multiprocessing, Multiprogramming, Parallelism feedback, Parallel computation, Processor allocation, Space sharing, Thread scheduling, Two-level scheduling, Trim analysis, Work, Work stealing.

* Yuxiong He is a Visiting Scholar at MIT CSAIL and a Ph.D. candidate at the Nanyang Technological University.

1. Introduction

The large expense of high-end multiprocessors makes it attractive for them to run multiprogrammed workloads, where many parallel applications share the same machine. As Feitelson describes in his excellent survey [29], schedulers for these machines can be implemented using two levels: a kernel-level ***job scheduler*** to allot processors to jobs, and a user-level ***thread scheduler*** to schedule the threads belonging to a given job onto 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 ***adaptive*** (called "dynamic" in [21]), where the number of processors allotted to a job may change while the job is running, or ***non-adaptive*** (called "static" in [21]), where a job runs on a fixed number of processors for its lifetime.

Randomized work-stealing [4, 13, 31] has proved to be an effective way to design a thread scheduler, both in theory and in practice. The decentralized thread scheduler is unaware of all the available threads to execute at a given moment. Whenever a processor runs out of work, it "steals" work from another processor chosen a random. To date, however, no work-stealing thread schedulers have been designed that provide provably effective parallelism feedback to a job scheduler.

In this paper we present an adaptive work-stealing thread scheduler, A-STEAL, which provides feedback about the job's parallelism to a space-sharing job scheduler by requesting processors from the job scheduler at regular intervals, called ***scheduling quanta***. Based on this parallelism feedback, the job scheduler can alter the allotment of processors to the job for the upcoming quantum according to the availability of processors in the current system environment and the job scheduler's administrative policy. A-STEAL is inspired by a task-scheduling algorithm, called A-GREEDY, which we developed in previous work [2] with Wen Jing Hsu from Nanyang Technological University in Singapore. Whereas A-GREEDY uses a centralized algorithm to schedule tasks on allotted processors, our new A-STEAL algorithm works in a decentralized fashion, using work-stealing to schedule the threads on allotted pro-

cessors. We prove that A-STEAL is efficient, minimizing both execution time and wasted processor cycles.

While A-STEAL is an extension of classic randomized work stealing and the feedback algorithm of A-GREEDY, combining the algorithms posed novel technical challenges, because unlike classical randomized work-stealing, A-STEAL must deal with dynamic changes in the job's processor allotment. In particular, when the allotment decreases, we use a mechanism called "mugging" [14], which involves a processor stealing all the work of another processor, rather than just a single task. Paradoxically, although muggings are considered as waste in our analysis, they are treated as productive work for the purpose of providing parallelism feedback.

Like prior work on scheduling of multithreaded jobs [7, 9, 10, 12, 13, 28, 38, 44], we model the execution of a multithreaded job as a dynamically unfolding directed acyclic graph (dag). Each node in the dag represents a unit-time instruction, and an edge represents a serial dependence between nodes. A ***thread*** is a chain of nodes with no branches. A node becomes ***ready*** when all its predecessors have been executed, and a thread becomes ready when its first node becomes ready. The ***work*** T_1 of the job corresponds to the total number of nodes in the dag and the ***critical-path length*** T_∞ corresponds to the length of the longest chain of dependencies. The ***parallelism*** of the job is the quantity T_1/T_∞, which represents the average amount of work along each step of the critical path. Each job has its own thread scheduler, which operates in an online manner, oblivious to the future characteristics of the dynamically unfolding dag.

In the scheduling model, we assume that time is broken into a sequence of equal-size ***scheduling quanta*** $1, 2, \ldots$, each consisting of L time steps, and the job scheduler is free to reallocate processors between quanta. The ***quantum length*** L is a system configuration parameter chosen to be long enough to amortize the time to reallocate processors among the various jobs and to perform various other bookkeeping for scheduling, including communication between the thread scheduler and the job scheduler, which typically involves a system call.

The thread scheduler operates as follows. Between quanta $q-1$ and q, it determines its job's ***desire*** d_q, which is the number of processors the job wants for quantum q. The thread scheduler provides the desire d_q to the job scheduler as its parallelism feedback. The job scheduler follows some processor allocation policy to determine the ***processor availability*** p_q — the number of processors to which the job is entitled for the quantum q. In order to make the thread scheduler robust to different system environments and administrative policies, our analysis of A-STEAL assumes that the job scheduler decides the availability of processors as an adversary. The number of processors the job receives for quantum q is the job's ***allotment*** $a_q = \min\{d_q, p_q\}$, the smaller of the job's desire and the processor availability. Once a job is allotted its processors, the allotment does not change during the quantum. Consequently, the thread scheduler must do a good job before a quantum of estimating how many processors it will need for all L time steps of the quantum, as well as do a good job of scheduling the ready threads on the allotted processors.

In an adaptive setting where the number of processors allotted to a job can change during execution, both $T_1/\overline{P}$ and T_∞ are lower bounds on the running time, where $\overline{P}$ is the mean of the processor availability during the computation. In the worst case, however, an adversarial job scheduler can prevent any thread scheduler from providing good speedup with respect to the mean availability $\overline{P}$. 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*** — the number of threads ready to run at a given moment — no adaptive scheduling algorithm can effectively utilize the available processors on that quantum.[1]

We use ***trim analysis*** [2] to analyze the time bound of adaptive thread schedulers under these adversarial conditions. Trim analysis borrows from the field of statistics 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, or just ***R-trimmed availablility***, for short. A good thread scheduler should provide linear speedup with respect to an R-trimmed availability, where R is as small as possible.

We prove that A-STEAL guarantees linear speedup with respect to $O(T_\infty + L \lg P)$-trimmed availability. Specifically, consider a job with work T_1 and critical-path length T_∞ running on a machine with P processors and a scheduling quantum of length L. A-STEAL completes the job in expected $O(T_1/\widetilde{P} + T_\infty + L \lg P)$ time steps, where $\widetilde{P}$ denotes the $O(T_\infty + L \lg P)$-trimmed availability. Thus, the job achieves linear speed up with respect to the trimmed availability $\widetilde{P}$ when the parallelism T_1/T_∞ dominates $\widetilde{P}$. In addition, we prove that the total number of processor cycles wasted by the job is $O(T_1)$, representing at most a constant-factor overhead.

Although this paper concerns itself only with the theoretical performance of A-STEAL, we have also studied its empirical performance using simulations [3]. To summarize that work, a linear-regression analysis using a variety of availability profiles indicates that A-STEAL provides almost perfect linear speedup with respect to the mean availability. The simulations indicate that A-STEAL typically wastes less than 20% of the processor cycles allotted to the job. We also compared A-STEAL with the ABP algorithm, an adaptive work-stealing thread scheduler developed by Arora, Blumofe, and Plaxton which does not employ parallelism feedback. 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 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 virtually all of these job runs, A-STEAL wastes less than 10% of the processor cycles wasted by ABP.

The remainder of this paper is organized as follows. Section 2 describes the A-STEAL algorithm and Section 3 provides a trim analysis of its completion time, while Section 4 provides the waste analysis. Section 5 explains the trade-offs among various parameters. Section 6 describes related work in adaptive and nonadaptive scheduling, putting this paper into an historical perspective. Finally, Section 7 offers some concluding remarks.

2. Adaptive work-stealing

This section presents the adaptive work-stealing thread scheduler A-STEAL. Before the start of a quantum, A-STEAL es-

[1] Using mean processor allotment instead of mean availability does not provide useful results. The trivial thread scheduler that always requests (and receives) 1 processor can achieve perfect linear speedup with respect to its mean allotment (which is 1) while wasting no processor cycles. By using a measure of availability, the thread scheduler must attempt to exploit parallelism.

timates processor desire based on the job's history of utilization. It uses this estimate as its parallelism feedback to the job scheduler, which it provides in the form of a request for processors. In this section, we describe A-STEAL and its desire-estimation heuristic.

During a quantum, A-STEAL uses ***work-stealing*** [4,13,41] to schedule the job's threads on the allotted processors. A-STEAL can use any provably good work-stealing algorithm, such as that of Blumofe and Leiserson [13] or the nonblocking one presented by Arora, Blumofe, and Plaxton [4]. In a work-stealing thread scheduler, every processor alloted to the job maintains a queue of ready threads for the job. When the ready queue of a processor becomes empty, the processor becomes a ***thief***, randomly picking a ***victim*** processor and ***stealing*** work from the victim's ready queue. If the victim has no available work, then the steal is ***unsuccessful***, and the thief continues to steal at random from other processors until it is ***successful*** and finds work. At all times, every processor is either working or stealing.

This basic work-stealing algorithm must be modified to deal with dynamic changes in processor allotment to the job between quanta. Two simple modifications make the work-stealing algorithm adaptive.

Allotment gain: When the allotment increases from quantum $q-1$ to q, the thread scheduler obtains $a_q - a_{q-1}$ additional processors. Since the ready queues of these new processors start out empty, all these processors immediately start stealing to get work from the other processors.

Allotment loss: When the allotment decreases from quantum $q-1$ to q, the job scheduler deallocates $a_{q-1} - a_q$ processors, whose ready queues may be nonempty. To deal with these queues, we use the concept of "mugging" [14]. When a processor runs out of work, instead of stealing immediately, it looks for a ***muggable*** queue, a nonempty queue that has no associated processor working on it. Upon finding a muggable queue, the thief ***mugs*** the queue by taking over the entire queue as its own. Thereafter, it works on the queue as if it were its own. If there are no muggable queues, the thief steals normally.

At all time steps during the execution of A-STEAL, every processor is either working, stealing, or mugging. We call the cycles a processor spends on working, stealing, and mugging as ***work-cycles***, ***steal-cycles***, and ***mug-cycles***, respectively. Cycles spent stealing and mugging are ***wasted***.

The salient part of A-STEAL is its desire-estimation algorithm, which is extended from the desire-estimation heuristic for the A-GREEDY algorithm originally presented in [2]. To estimate the desire for the next quantum $q+1$, A-STEAL classifies the previous quantum q as either "satisfied" or "deprived" and either "efficient" or "inefficient." Of the four possibilities for classification, A-STEAL only uses three: inefficient, efficient-and-satisfied, and efficient-and-deprived. Using this three-way classification and the job's desire for the previous quantum q, it computes the desire for the next quantum $q+1$.

A-STEAL classifies a quantum as satisfied versus deprived by comparing the allotment a_q with the desire d_q. The quantum q is ***satisfied*** if $a_q = d_q$, that is, the job receives as many processors as A-STEAL requested for it from the job scheduler. Otherwise, if $a_q < d_q$, the quantum is ***deprived***, because the job did not receive as many processors as it requested.

A-STEAL classifies a quantum as efficient versus inefficient by comparing the usage with the allotment. We define the ***usage*** u_q of quantum q as the total number of work-cycles in q and the ***nonsteal usage*** n_q as the sum of the number of work-cycles and mug-cycles. Therefore, we have $u_q \le n_q$. A-STEAL uses a ***utilization parameter*** δ as the threshold to differentiate between efficient and inefficient quanta. The utilization parameter δ in A-STEAL is a lower bound on the fraction of available processors used to work or mug on accounted steps. Typical values for δ might be in the range of 80% to 95%. We call a quantum q ***efficient*** if $n_q \ge \delta L a_q$, that is, the nonsteal usage is at least a δ fraction of the total processor cycles allotted. A quantum is ***inefficient*** otherwise. Inefficient quanta contain at least $(1-\delta)La_q$ steal-cycles.

A-STEAL (q, δ, ρ)

```
1  if q = 1
2     then d_q ← 1                    ▷ base case
3  elseif n_{q-1} < Lδa_{q-1}
4     then d_q ← d_{q-1}/ρ            ▷ inefficient
5  elseif a_{q-1} = d_{q-1}
6     then d_q ← ρd_{q-1}             ▷ efficient-and-satisfied
7  else d_q ← d_{q-1}                 ▷ efficient-and-deprived
8  Report d_q to the job scheduler.
9  Receive allotment a_q from the job scheduler.
10 Schedule on a_q processors using randomized work
   stealing for L time steps.
```

Figure 1: Pseudocode for the adaptive work-stealing thread scheduler A-STEAL, which provides parallelism feedback to a job scheduler in the form of processor desire. Before quantum q, A-STEAL uses the previous quantum's desire d_{q-1}, allotment a_{q-1}, and nonsteal usage n_{q-1} to compute the current quantum's desire d_q based on the utilization parameter δ and the responsiveness parameter ρ.

It might seem counterintuitive for the definition of "efficient" to include mug-cycles. After all, mug-cycles are wasted. The rationale is that mug-cycles arise as a result of an allotment loss. Thus, they do not generally indicate that the job has a surplus of processors. Therefore, A-STEAL does not penalize jobs for too many mug cycles in a quantum.

A-STEAL calculates the desire d_q of the current quantum q based on the previous desire d_{q-1} and the three-way classification of quantum $q-1$ as inefficient, efficient-and-satisfied, and efficient-and-deprived. The initial desire is $d_1 = 1$. Like A-GREEDY, A-STEAL 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 pseudocode of A-STEAL for one quantum. A-STEAL takes as input the quantum q, the utilization parameter δ, and the responsiveness parameter ρ. For the first quantum, it requests 1 processor. Thereafter, it operates as follows:

- **Inefficient:** If quantum $q-1$ was inefficient, it contained many steal-cycles, which indicates that most of the processors had insufficient work to do. Therefore, A-STEAL overestimated the desire for quantum $q-1$. In this case, A-STEAL does not care whether quantum $q-1$ was satisfied or deprived. It simply decreases the desire (line 4) for quantum q.
- **Efficient-and-satisfied:** If quantum $q-1$ was efficient-and-satisfied, the job effectively utilized the processors that A-STEAL requested on its behalf. In this case, A-STEAL speculates that the job can use more processors. It increases the desire (line 6) for quantum q.
- **Efficient-and-deprived:** If quantum $q-1$ was efficient-and-deprived, the job used all the processors it was allotted, but A-STEAL had requested more processors for the job

than the job actually received from the job scheduler. Since A-STEAL has no evidence whether the job could have used all the processors requested, it maintains the same desire (line 7) for quantum q.

After determining the job's desire, A-STEAL requests that many processors from the job scheduler, receives its allotment, and then schedules the job on the allotted processors for the L time steps of the quantum.

3. Time analysis

This section uses a trim analysis to analyze A-STEAL with respect to the completion time. Suppose that A-STEAL schedules a job with work T_1 and critical-path length T_∞ on a machine with P processors. Let ρ denote A-STEAL's responsiveness parameter, δ its utilization parameter, and L the quantum length. For any constant $0 < \epsilon < 1$, A-STEAL completes the job in time $T = O\left(T_1/\widetilde{P} + T_\infty + L\log_\rho P + L\ln(1/\epsilon)\right)$, with probability at least $1-\epsilon$, where $\widetilde{P}$ is the $O(T_\infty + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability. This bound implies that A-STEAL achieves linear speedup on all the time steps excluding $O(T_\infty + L\log_\rho P + L\ln(1/\epsilon))$ time steps with the highest processor availability.

We make two assumptions to simply the presentation of the analysis. First, we assume that there is no contention for steals and that a steal can be completed on a single time step. Second, we assume that a mug can be completed on one time step as well. That is, if there is a muggable queue, then a thief processor can find it instantly and mug it. If there is no muggable queue, then a thief processor knows instantly that there is no muggable queue and it should start stealing. We shall relax these assumptions at the end of this section.

We prove our completion-time bound using a trim analysis, which calculates the performance by discarding a few outliers and measures that for the remaining majority. We label each quantum as either ***accounted*** or ***deductible***. Accounted quanta are those for which $n_q \geq L\delta p_q$, where n_q is the nonsteal usage. That is, the job works or mugs for at least a δ fraction of the Lp_q processor cycles available during the quantum. Conversely, the deductible quanta are those for which $n_q < L\delta p_q$. Our trim analysis will show that when we ignore the relatively few deductible quanta, we obtain linear speedup on the more numerous accounted quanta.

We can relate this labeling of quanta as accounted versus deductible to a three-way classification of quanta as inefficient, efficient-and-satisfied, and efficient-and-deprived:

- **Inefficient:** On an inefficient quantum q, we have $n_q < L\delta a_q \leq L\delta p_q$, since the allotment a_q never exceeds the availability p_q. We label all inefficient quanta as deductible, irrespective of whether they are satisfied or deprived.
- **Efficient-and-satisfied:** On an efficient quantum q, we have $n_q \geq L\delta a_q$. Since we have $a_q = \min\{p_q, d_q\}$ for a satisfied quantum, it follows that $a_q = d_q \leq p_q$. Despite these two bounds, we may nevertheless have $n_q < L\delta p_q$. Since we cannot guarantee that $n_q \geq L\delta p_q$, we pessimistically label the quantum q as deductible.
- **Efficient-and-deprived:** As before, on an efficient quantum q, we have $n_q \geq L\delta a_q$. On a deprived quantum, we have $a_q < d_q$ by definition. Since $a_q = \min\{p_q, d_q\}$, we must have $a_q = p_q$. Hence, it follows that $n_q \geq L\delta a_q = L\delta p_q$, and we label quantum q as accounted.

We now analyze the execution time of A-STEAL by bounding the number of deductible and accounted quanta separately. Two observations provide intuition for the proof. First, each inefficient quantum contains a large number of steal-cycles, which we can expect to reduce the length of the remaining critical path. This observation will help us to bound the number of deductible quanta. Second, most of the processor cycles on an efficient quantum are spent either working or mugging. We shall show that there cannot be too many mug-cycles during the job's execution, and thus most of the processor cycles on efficient quanta are spent doing useful work. This observation will help us to bound the number of accounted quanta.

The following lemma, proved in [13], shows how steal-cycles reduce the length of the job's critical path.

LEMMA 1. *If a job has r ready queues, then $3r$ steal-cycles suffice to reduce the length of the job's remaining critical path by at least 1 with probability at least $1 - 1/e$, where e is the base of the natural logarithm.* □

The next lemma shows that an inefficient quantum reduces the length of the job's critical path, which we shall later use to bound the total number of inefficient quanta.

LEMMA 2. *If ρ is* A-STEAL*'s responsiveness parameter and L is the quantum length, on an inefficient quantum,* A-STEAL *reduces the length of a job's remaining critical path by at least $(1-\delta)L/6$ with probability greater than $1/4$.*

Proof. Let q be an inefficient quantum. A processor with an empty ready queue steals only when it cannot mug a queue, and hence, all the steal-cycles on quantum q occur when the number of nonempty queues is at most the allotment a_q. Therefore, Lemma 1 dictates that $3a_q$ steal-cycles suffice to reduce the critical-path length by 1 with probability at least $1 - 1/e$. Since the quantum q is inefficient, it contains at least $(1-\delta)La_q$ steal-cycles. Divide the time steps of the quantum into ***rounds*** such that each round contains $3a_q$ steal-cycles. Thus, there are at least $j = (1-\delta)La_q/3a_q = (1-\delta)L/3$ rounds.[2] We call a round ***good*** if it reduces the length of the critical path by at least 1; otherwise, the round is ***bad***. For each round i on quantum q, we define the indicator random variable X_i to be 1 if round i is a bad round and 0 otherwise, and let $X = \sum_{i=1}^{j} X_i$. Since $\Pr\{X_i = 1\} < 1/e$, linearity of expectation dictates that $\mathrm{E}[X] < j/e$. We now apply Markov's inequality [22, p. 1111], which says that for a nonnegative random variable, we have $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$ for all $t > 0$. Substituting $t = j/2$, we obtain $\Pr\{X > j/2\} \leq \mathrm{E}[X]/(j/2) \leq (j/e)/(j/2) = 2/e < 3/4$. Thus, with probability greater than $1/4$, the quantum q contains at least $j/2$ good rounds. Since each good round reduces the critical-path length by at least 1, with probability greater than $1/4$, the critical-path length reduces by least $j/2 = ((1-\delta)L/3)/2 = (1-\delta)L/6$ during quantum q. □

LEMMA 3. *Suppose that* A-STEAL *schedules a job with critical-path length T_∞. If L is the quantum length, then for any $\epsilon > 0$, the schedule produces at most $48T_\infty/(L(1-\delta)) + 16\ln(1/\epsilon)$ inefficient quanta with probability at least $1-\epsilon$.*

Proof. Let I be the set of inefficient quanta. Define an inefficient quantum q as ***productive*** if it reduces the critical-path length by at least $(1-\delta)L/6$ and ***unproductive*** otherwise. For each quantum $q \in I$, define the indicator random variable Y_q to be 1 if q is productive and 0 otherwise. By

[2] Actually, the number of rounds is $j = \lfloor (1-\delta)L/3 \rfloor$, but we shall ignore the roundoff for simplicity. A more detailed analysis can nevertheless produce the same constants in the bounds for Lemmas 3 and 6.

Lemma 2, we have $\Pr\{Y_q = 1\} > 1/4$. Let the total number of unproductive quanta be $Y = \sum_{q \in I} Y_q$. For simplicity in notation, let $A = 6T_\infty/(1-\delta)L$. If the job's execution contains $|I| \geq 48T_\infty/(1-\delta)L + 16\ln(1/\epsilon)$ inefficient quanta, then we have $\mathrm{E}[Y] > |I|/4 \geq 12T_\infty/(1-\delta)L + 4\ln(1/\epsilon) = 2A + 4\ln(1/\epsilon)$. Using the Chernoff bound $\Pr\{Y < (1-\lambda)E[Y]\} < \exp(-\lambda^2 E[Y]/2)$ [43, p. 70] and choosing $\lambda = (A + 4\ln(1/\epsilon))/(2A + 4\ln(1/\epsilon))$, we obtain

$$\begin{aligned}
&\Pr\{Y < A\} \\
&= \Pr\left\{Y < \left(1 - \frac{A + 4\ln(1/\epsilon)}{2A + 4\ln(1/\epsilon)}\right)(2A + 4\ln(1/\epsilon))\right\} \\
&= \Pr\{Y < (1-\lambda)(2A + 4\ln(1/\epsilon))\} \\
&\leq \exp\left(-\frac{\lambda^2}{2}(2A + 4\ln(1/\epsilon))\right) \\
&= \exp\left(-\frac{1}{2} \cdot \frac{(A + 4\ln(1/\epsilon))^2}{2A + 4\ln(1/\epsilon)}\right) \\
&< \exp\left(-\frac{1}{2} \cdot 4\ln(1/\epsilon) \cdot \frac{1}{2}\right) \\
&= \epsilon\,.
\end{aligned}$$

Therefore, if the number $|I|$ of inefficient quanta is at least $48T_\infty/(1-\delta)L + 16\ln(1/\epsilon)$, the number of productive quanta is at least $A = 6T_\infty/(1-\delta)L$ with probability at least $1-\epsilon$. By Lemma 2 each productive quantum reduces the critical-path length by at least $(1-\delta)L/6$, and therefore at most $A = 6T_\infty/(1-\delta)L$ productive quanta occur during job's execution. Consequently, with probability at least $1-\epsilon$, the number of inefficient quanta is $|I| \leq 48T_\infty/(1-\delta)L + 16\ln(1/\epsilon)$. □

The following technical lemma proved in [2] bounds the maximum value of the desire.

LEMMA 4. *Suppose that* A-STEAL *schedules a job on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, then before any quantum q, the desire d_q of the job is bounded by $d_q < \rho P$.* □

The next lemma reveals a relationship between inefficient quanta and efficient-and-satisfied quanta.

LEMMA 5. *Suppose that* A-STEAL *schedules a job on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter and the schedule produces m inefficient quanta, then it produces at most $m + \log_\rho P + 1$ efficient-and-satisfied quanta.*

Proof. Assume for the purpose of contradiction that a job's execution has m inefficient quanta, but $k > m + \log_\rho P + 1$ efficient-and-satisfied quanta. Recall that desire increases by ρ after every efficient-and-satisfied quantum, decreases by ρ after every inefficient quantum, and does not change otherwise. Thus, the total increase in desire is ρ^k, and the total decrease in desire is ρ^m. Since the desire starts at 1, the desire at the end of the job is $\rho^{k-m} > \rho^{\log_\rho P + 1} = \rho P$, contradicting Lemma 4. □

The following lemma bounds the number of efficient-and-satisfied quanta.

LEMMA 6. *Suppose that* A-STEAL *schedules a job with critical-path length T_∞ on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then the schedule produces at most $48T_\infty/(1-\delta)L + \log_\rho P + 16\ln(1/\epsilon) + 1$ efficient-and-satisfied quanta with probability at least $1-\epsilon$ for any $\epsilon > 0$.*

Proof. The lemma follows directly from Lemmas 3 and 5. □

The next lemma exhibits the relationship between inefficient quanta and efficient-and-satisfied quanta.

LEMMA 7. *Suppose that* A-STEAL *schedules a job, and let I and C denote the sets of inefficient and efficient-and-satisfied quanta, respectively, produced by the schedule. If ρ is* A-STEAL*'s responsiveness parameter, then there exists an injective mapping $f : I \to C$ such that for all $q \in I$, we have $f(q) < q$ and $d_{f(q)} = d_q/\rho$.*

Proof. For every inefficient quantum $q \in I$, define $r = f(q)$ to be the latest efficient-and-satisfied quantum such that $r < q$ and $d_r = d_q/\rho$. Such a quantum always exists, because the initial desire is 1 and the desire increases only after an efficient-and-satisfied quantum. We must prove that f does not map two inefficient quanta to the same efficient-and-satisfied quantum. Assume for the sake of contradiction that there exist two inefficient quanta $q < q'$ such that $f(q) = f(q') = r$. By definition of f, the quantum r is efficient-and-satisfied, $r < q < q'$, and $d_q = d_{q'} = \rho d_r$. After the inefficient quantum q, A-STEAL reduces the desire to d_q/ρ. Since the desire later increases again to $d_{q'} = d_q$ and the desire increases only after efficient-and-satisfied quanta, there must be an efficient-and-satisfied quantum r' in the range $q < r' < q'$ such that $d(r') = d(q')/\rho$. But then, by the definition of f, we would have $f(q') = r'$. Contradiction. □

We can now bound the total number of mug-cycles executed by processors.

LEMMA 8. *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, the schedule produces at most $((1+\rho)/(L\delta - 1 - \rho))T_1$ mug-cycles.*

Proof. When the allotment decreases, some processors are deallocated, and their ready queues are declared muggable. The total number M of mug-cycles is at most the number of muggable queues during the job's execution. Since the allotment reduces by at most $a_q - 1$ from quantum q to quantum $q+1$, there are $M \leq \sum_q (a_q - 1) < \sum_q a_q$ mug-cycles during the execution of the job.

By Lemma 7, for each inefficient quantum q, there is a distinct corresponding efficient-and-satisfied quantum $r = f(q)$ that satisfies $d_q = \rho d_r$. By definition, each efficient-and-satisfied quantum r has a nonsteal usage $n_r \geq L\delta a_r$ and allotment $a_r = d_r$. Thus, we have $n_r + n_q \geq L\delta a_r = (L\delta/(1+\rho))(a_r + \rho a_r) = (L\delta/(1+\rho))(a_r + \rho d_r) \geq (L\delta/(1+\rho))(a_r + a_q)$, since $a_q \leq d_q$ and $d_q = \rho d_r$. Except for these inefficient quanta and their corresponding efficient-and-satisfied quanta, every other quantum q is efficient, and hence $n_q \geq L\delta a_q$ for these quanta. Let $N = \sum_q n_q$ be the total number of nonsteal-cycles during the job's execution. We have $N = \sum_q n_q \geq (L\delta/(1+\rho)) \sum_q a_q \geq (L\delta/(1+\rho))M$. Since the total number of nonsteal-cycles is the sum of work-cycles and mug-cycles and the total number of work-cycles is T_1, we have $N = T_1 + M$, and hence, $T_1 = N - M \geq (L\delta/(1+\rho))M - M = ((L\delta - 1 - \rho)/(1+\rho))M$, which yields $M \leq ((1+\rho)/(L\delta - 1 - \rho))T_1$. □

LEMMA 9. *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, the schedule produces at most $(T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$ accounted quanta, where P_A is the mean availability on the accounted quanta.*

Proof. Let A and D denote the set of accounted and deductible quanta, respectively. The mean availability on accounted quanta is $P_A = (1/|A|)\sum_{q\in A} p_q$. Let N be the total number of nonsteal-cycles. By definition of accounted quanta, the nonsteal usage satisfies $n_q \geq L\delta a_q$. Thus, we have $N = \sum_{q\in A\cup D} n_q \geq \sum_{q\in A} n_q \geq \sum_{q\in A} \delta L p_q = \delta L\,|A|\,P_A$, and hence, we obtain

$$|A| \leq N/L\delta P_A\ . \tag{1}$$

But, the total number of nonsteal-cycles is the sum of the number of work-cycles and mug-cycles. Since there are at most T_1 work-cycles on accounted quanta and Lemma 8 shows that there are at most $M \leq ((1+\rho)/(L\delta - 1 - \rho))T_1$ mug-cycles, we have $N \leq T_1 + M \leq T_1(1 + (1+\rho)/(L\delta - 1 - \rho))$. Substituting this bound on N into Inequality (1) completes the proof.

□

We are now ready to bound the running time of jobs scheduled with A-STEAL.

THEOREM 10. *Suppose that* A-STEAL *schedules a job with work T_1 and critical-path length T_∞ on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then for any $\epsilon > 0$, with probability at least $1-\epsilon$,* A-STEAL *completes the job in*

$$\begin{aligned} T \quad &\leq \quad \frac{T_1}{\delta\widetilde{P}}\left(1 + \frac{1+\rho}{L\delta - 1 - \rho}\right) \\ &\quad + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right) \end{aligned} \tag{2}$$

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

Proof. The proof uses trim analysis. Let A be the set of accounted quanta, and let D be the set of deductible quanta. Lemmas 3 and 6 show that there are $|D| = O(T_\infty/((1-\delta)L) + \log_\rho P + \ln(1/\epsilon))$ deductible quanta, and hence $L\,|D| = O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$ time steps belong to deductible quanta. We have that $P_A \geq \widetilde{P}$, since the mean availability on the accounted time steps (we trim the $L\,|D|$ deductible steps) must be at least the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability (we trim the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$ steps that have the highest availability). From Lemma 9, the number of accounted quanta is at most $(T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$, and since $T = L(|A| + |D|)$, the desired time bound follows. □

COROLLARY 11. *Suppose that* A-STEAL *schedules a job with work T_1 and critical-path length T_∞ on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then* A-STEAL *completes the job in expected time $\mathrm{E}[T] = O(T_1/\widetilde{P} + T_\infty + L\lg P)$, where $\widetilde{P}$ is the $O(T_\infty + L\lg P)$-trimmed availability.*

Proof. Straightforward conversion of high-probability bound to expectation, together with setting δ and ρ to suitable constants. □

Our analysis made two assumptions to ease the presentation. First, we assumed that there is no contention for steals. Second, we assumed that a thief processor can find a muggable queue and mug it in unit time. Now, let us relax these assumptions.

The first issue dealing with the contention on steals has been addressed by Blumofe and Leiserson in [13]. A balls-and-bins argument can be used to prove that taking the contention of steals into account would increase the running time by at most $O(\lg P)$, which is tiny compared to the other terms in our running time.

Mugging requires more data-structure support. When a processor runs out of work, it needs to find out if there are any muggable queues for the job. As a practical matter, these muggable queues can be placed in a set (using any synchronous queue or set implementations as in [49, 52, 53, 56]). This strategy could increase the number of mug-cycles by a factor of P in the worse case. If $P \ll L$, however, this change does not affect the running time bound by much. Moreover, in practice, the number of muggings is so small that the time spent on muggings is insignificant compared to the total running time of the job. Alternatively, to obtain a better theoretical bound, we could use a counting network [5] with width P to implement the list of muggable queues, in which case each mugging operation would consume $O(\lg^2 P)$ processor cycles. The number of accounted steps in the time bound from Lemma 9 would increase slightly to $(T_1/\delta\widetilde{P})\left(1 + (1+\rho)\lg^2 P/(L\delta - 1 - \rho)\right)$, but the number of deductible steps would not change.

4. Waste analysis

This section proves that when a job is scheduled by A-STEAL, the total number of processor cycles wasted during the job's execution is $W = O(T_1)$ in the worst case.

THEOREM 12. *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then* A-STEAL *wastes at most*

$$W \leq \left(\frac{1+\rho-\delta}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right) T_1 \tag{3}$$

processor cycles during the course of the computation.

Proof. Let M be the total number of mug-cycles, and let S be the total number of steal-cycles. Hence, we have $W = S + M$. Since Lemma 8 provides an upper bound of M, we only need to bound S. We use an accounting argument to calculate S based on whether a quanta is inefficient or efficient. Let S_{ineff} and S_{eff}, where $S = S_{\text{ineff}} + S_{\text{eff}}$, be the numbers of steal-cycles on inefficient and efficient quanta, respectively.

Inefficient quanta: Lemma 7 shows that every inefficient quantum q with desire d_q corresponds to a distinct efficient-and-satisfied quantum $r = f(q)$ with desire $d_r = d_q/\rho$. Thus, the steal-cycles on quantum q can be amortized against the nonsteal-cycles on quantum r. Since quantum r is efficient-and-satisfied, its nonsteal usage satisfies $n_r \geq L\delta a_q/\rho$, and its allocation is $a_r = d_r$. Therefore, we have $n_r \geq L\delta a_r = L\delta d_r = L\delta d_q/\rho \geq L\delta a_q/\rho$. Let s_q be the number of steal-cycles on quantum q. Since the quantum contains at most La_q total processor cycles, we have $s_q \leq La_q \leq \rho n_r/\delta$, that is, the number of steal-cycles in the inefficient quantum q is at most a ρ/δ fraction of the nonsteal-cycles in its corresponding efficient-and-satisfied quantum r. Therefore, the total number of steal-cycles in all inefficient quanta satisfies $S_{\text{ineff}} \leq (\rho/\delta)(T_1 + M)$.

Efficient quanta: On any efficient quantum q, the job performs at least $L\delta a_q$ work- and mug-cycles and at most $L(1-\delta)a_q$ steal-cycles. Summing over all efficient quanta,

the number of steal-cycles on efficient quanta is $S_{\text{eff}} \leq ((1-\delta)/\delta)(T_1 + M)$.

Using the bound $M \leq ((1+\rho)/(L\delta - 1 - \rho))T_1$ from Lemma 8, we obtain

$$\begin{aligned} W &= S + M \\ &= S_{\text{ineff}} + S_{\text{eff}} + M \\ &\leq (\rho/\delta)(T_1 + M) + ((1-\delta)/\delta)(T_1 + M) + M \\ &= (T_1 + M)\frac{1+\rho-\delta}{\delta} + M \\ &\leq \left(T_1 + T_1\frac{1+\rho}{L\delta - 1 - \rho}\right)\frac{1+\rho-\delta}{\delta} + T_1\frac{1+\rho}{L\delta - 1 - \rho} \\ &= T_1\left(\left(1 + \frac{1+\rho}{L\delta - 1 - \rho}\right)\frac{1+\rho-\delta}{\delta} + \frac{1+\rho}{L\delta - 1 - \rho}\right) \\ &= T_1\left(\frac{1+\rho-\delta}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right) , \end{aligned}$$

which proves the theorem. □

5. Interpretation of the bounds

In this section, we simplify the bounds on A-STEAL's behavior to understand the tradeoffs involved in completion time and waste due to parameter settings. Although our bounds are good asymptotically, they contain large constants. Some might wonder whether these large constants might adversely affect A-STEAL's practical utility. We argue that most of the large constants are due to the assumption of the adversarial behavior of the job scheduler, and thus, we should expect the practical performance of A-STEAL to be better than that indicated by our bounds. Moreover, we have implemented A-STEAL in a simulation environment [3] using the DESMO-J [24] Java simulator, which provides strong evidence that A-STEAL should be efficient in practice.

If the utilization parameter δ and responsiveness parameter ρ are constants, the bounds in Inequalities (2) and (3) can be simplified as follows:

$$\begin{aligned} T &\leq \frac{T_1}{\delta\widetilde{P}}(1 + O(1/L)) \\ &\quad + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right) , \quad (4) \\ W &\leq \left(\frac{1+\rho-\delta}{\delta} + O(1/L)\right)T_1 . \quad (5) \end{aligned}$$

This reformulation allows us to see the trade-offs due to the setting of the δ and ρ parameters more easily. As δ increases and ρ decreases, the completion time increases and the waste decreases. Thus, reasonable values for the utilization parameter δ might lie between 80% and 95%, and the responsiveness parameter ρ might be set between 1.2 and 2.0. The quantum length L is a system configuration parameter, which might have values in the range 10^3 to 10^5.

For the time bound in Inequality (4), as δ increases toward 1, the coefficient of $T_1/\widetilde{P}$ decreases toward 1, and the job comes closer to perfect linear speedup on accounted steps, but the number of deductible steps increases as well. The large number of deductible steps is due to the adversarial job scheduler. We have performed simulations [3] which indicate that the jobs usually achieve nearly perfect linear speedup with respect to $\widetilde{P}$ for a variety of availability profiles.

To see how these settings affect the waste bound, consider the waste bound in Inequality (5) as two parts, where the waste due to steal-cycles is $S \leq (1+\rho-\delta)T_1/\delta$, and the waste due to mug-cycles is only $M = O(1/L)T_1$. Since the waste on mug-cycles is just a tiny fraction compared to the work T_1, an implementation of A-STEAL need not overly concern itself with the bookkeeping overhead of adding and removing processors from jobs.

The major part of waste comes from steal-cycles, where S is generally less than $2T_1$ for typical parameter values. The analysis of Theorem 12 shows that the number of steal-cycles on efficient steps is bounded by $((1-\delta)/\delta)T_1$, which is a small fraction of S. Thus, most of the waste that occurs in the bound can be attributed to the steal-cycles on the inefficient quanta. To ensure the robustness of A-STEAL, our analysis assumes that the job scheduler is an adversary, which creates as many inefficient quanta as possible. Since job schedulers are generally not adversarial, we should not expect these large overheads to materialize in practice. The simulations in [3] indicate that the waste is a much smaller fraction of T_1 than that suggested by our theoretical bound. In those experiments, A-STEAL typically wastes less than 20% of the allotted processor cycles.

6. Related work

This section discusses related work on adaptive and nonadaptive schedulers for multithreaded jobs. Prior work on thread scheduling for multithreaded jobs has tended to focus on nonadaptive scheduling [8, 9, 13, 19, 33, 44] or adaptive scheduling without parallelism feedback [4]. We start by discussing nonadaptive work-stealing schedulers. We then discuss empirical and theoretical work on adaptive thread schedulers. Finally, we give a brief summary of research on adaptive job schedulers.

Work-stealing has been used as a heuristic since Burton and Sleep's research [20] and Halstead's implementation of Multilisp [36]. Many variants have been implemented since then [30, 35, 41], and it has been analyzed in the context of load balancing [48], backtrack search [39] etc. Blumofe and Leiserson [13] proved that the work-stealing algorithm is efficient with respect to time, space, and communication for the class of "fully strict" multithreaded computations. Arora, Blumofe and Plaxton [4] extended the time bound result to arbitrary multithreaded computations. In addition, Acar, Blelloch and Blumofe [1] show that work-stealing schedulers are efficient with respect to cache misses for jobs with "nested parallelism." Moreover, variants of work-stealing algorithms have been implemented in many systems [11, 17, 31] and empirical studies show that work-stealing schedulers are scalable and practical [16, 31].

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading, primarily by Blumofe and his coauthors [4, 15, 16, 18]. In this work, the thread scheduler uses randomized work-stealing strategy to schedule threads on available processors but does not provide the feedback about the job's parallelism to the job scheduler. The research in [15, 18] addresses networks of workstations where processors may fail or join and leave a computation while the job is running, showing that work-stealing provides a good foundation for adaptive task scheduling. In theoretical work, Arora, Blumofe, and Plaxton [4] show that their task scheduler (we call it the ABP scheduler) provably completes a job in $O(T_1/\overline{P} + PT_\infty/\overline{P})$ expected time. Blumofe and Papadopoulos [16] perform an empirical evaluation of ABP and show that on an 8-processor machine, ABP provides almost perfect linear speedup for jobs with reasonable parallelism.

Adaptive task scheduling with parallelism feedback has also been studied empirically in [50, 54, 55]. These researchers use a job's history of processor utilization to provide feedback

to dynamic-equipartitioning job schedulers. Their studies use different strategies for parallelism feedback, and all report better system performance with parallelism feedback than without, but it is not apparent which of their strategies is best. Our earlier work [2] appears to be the only theoretical analysis of a task scheduler with parallelism feedback.

In contrast to adaptive thread schedulers, adaptive job schedulers have been studied extensively, both empirically [21, 25, 32, 40, 45–47, 51, 57] and theoretically [6, 23, 26, 27, 34, 42]. McCann, Vaswani, and Zahorjan [40] 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. Gu [34] 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 [23] proved a similar result for mean response time for multiphase jobs regardless of their arrival times. He, Hsu and Leiserson [37] recently proved that two-level schedulers which combine A-STEAL (or A-GREEDY) with dynamic equipartitioning are constant competitive with respect to makespan (for arbitrary job arrivals) and mean completion time (for batched arrivals). Song [54] proves that a randomized distributed strategy can implement dynamic equipartitioning.

7. Conclusions

This section offers some conclusions and directions for future work.

This and previous research [2] has used the technique of trimming to limit a powerful adversary, enabling us to analyze adaptive schedulers with parallelism feedback. The idea of ignoring a few outliers while calculating averages is often used in statistics to ignore anomalous data points. For example, teachers often ignore the lowest score while computing a student's grade, and in the Olympic Games, the lowest and the highest scores are sometimes ignored when computing an athlete's average. In theoretical computer science, when an adversary is too powerful, we sometimes make statistical assumptions about the input to render the analysis tractable, but statistical assumptions may not be valid in practice. Trimming may prove itself of value for analyzing such problems.

A-STEAL needs full information about the previous quantum to estimate the desire of the current quantum. Collecting perfect information might become difficult as the number of processors becomes larger, especially if the number of processors exceeds the quantum length. A-STEAL only estimates the desire, however, and therefore approximate information should be enough to provide feedback. We are currently studying the possibility of using sampling techniques to estimate the number of steal-cycles, instead of counting the exact number.

Our model for jobs and scheduling takes only computation into account and has no performance model for handling input and output operations. But there is a large class of parallel applications which perform a large number of input/output operations. It is unclear how the thread scheduler should respond when a thread is performing I/O even in the case of nonadaptive work-stealing schedulers and the effect of I/O on all kinds of thread schedulers in theory and in practice is an interesting open question.

Acknowledgments

Thanks to the members of the Supercomputing Technologies group at MIT CSAIL and to Wen Jing Hsu of the Nanyang Technological Institute in Singapore for numerous helpful discussions.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA*, pages 1–12, New York, NY, USA, 2000.

[2] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive task scheduling with parallelism feedback. In *PPoPP*, 2006.

[3] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *ICDCS*, 2006.

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

[5] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[6] Nikhil Bansal, Kedar Dhamdhere, Jochen Konemann, and Amitabh Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, 2004.

[7] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[8] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *SPAA*, pages 1–12, Santa Barbara, California, 1995.

[9] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ICFP*, pages 213–225, 1996.

[10] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

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

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

[14] Robert D. Blumofe, Charles E. Leiserson, and Bin Song. Automatic processor allocation for work-stealing jobs. 1998.

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

[16] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. In *SIGMETRICS*, pages 266–267, 1998.

[17] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1999.

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

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

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

[21] Su-Hui Chiang and Mary K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *JSSPP*, pages 200–223, Honolulu, Hawaii, United States, 1996.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[23] Xiaotie Deng and Patrick Dymond. On multiprocessor system scheduling. In *SPAA*, pages 82–88, 1996.

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

[25] Derek L. Eager, John Zahorjan, and Edward D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[26] Jeff Edmonds. Scheduling in the dark. In *STOC*, pages 179–188, 1999.

[27] Jeff Edmonds, Donald D. Chinn, Timothy Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6(3):231–250, 2003.

[28] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, 1990.

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

[30] Raphael Finkel and Udi Manber. DIB—A distributed implementation of backtracking. *TOPLAS*, 9(2):235–256, April 1987.

[31] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[32] Dipak Ghosal, Giuseppe Serazzi, and Satish K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, 1991.

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

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

[35] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

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

[37] Yuxiong He, Hsu Wen Jing, and Charles E. Leiserson. Provably efficient two-level adaptive scheduling for multithreaded jobs on parallel computers. JSSPP, 2006.

[38] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35(5-6):743–765, 1991.

[39] Richard M. Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *STOC*, pages 290–300, Chicago, Illinois, May 1988.

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

[41] Eric Mohr, David A. Kranz, and Jr. Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *LFP*, pages 185–197, 1990.

[42] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. In *SODA*, pages 422–431, 1993.

[43] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1 edition, 1995.

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

[45] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Maximizing speedup through self-tuning of processor allocation. In *IPPS*, pages 463–468, 1996.

[46] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, pages 155–174. Springer-Verlag, 1996.

[47] Eric W. Parsons and Kenneth C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In *IPPS*, pages 127–145, London, UK, 1995. Springer-Verlag.

[48] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *SPAA*, pages 237–245, Hilton Head, South Carolina, July 1991.

[49] William Scherer, Doug Lea, and Michael Scott. Scalable synchronous queues. In *PPoPP*, 2006.

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

[51] K. C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *SIGMETRICS*, pages 171–180, 1989.

[52] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *SPAA*, pages 54–63, 1995.

[53] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions of Computer Systems*, 14(4):385–428, 1996.

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

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

[56] Roger Wattenhofer and Peter Widmayer. The counting pyramid: an adaptive distributed counting scheme. *Journal of Parallel and Distributed Computing*, 64(4):449–460, 2004.

[57] 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.