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, USA

October 31, 2008

Abstract

We present a randomized work-stealing thread scheduler for fork-join multithreaded jobs that provides continual parallelism feedback to the job scheduler in the form of processor requests. Our 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. Assuming that the job scheduler never allots the 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. For example, the job scheduler can make available a huge number of processors exactly when the job has little use for them. To analyze the performance of our adaptive thread scheduler under this stringent adversarial assumption, we introduce a new technique called "trim analysis," which allows us to prove that our thread scheduler performs poorly on at most a small number of time steps, exhibiting near-optimal behavior on the vast majority.

To be more 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 available, 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.

We measured the performance of A-STEAL on a simulated multiprocessor system using synthetic workloads. For jobs with sufficient parallelism, our experiments indicate that A-STEAL provides almost perfect linear speedup across a variety of processor availability profiles. In these experiments, A-STEAL typically wasted less than 20% of the processor cycles allotted to the job. We compared A-STEAL with the ABP algorithm, an adaptive work-stealing thread scheduler developed by Arora, Blumofe, and Plaxton that does not employ parallelism feedback. On moderate- to heavy-loaded large machines with predetermined availability profiles, A-STEAL typically completed jobs more than twice as quickly, despite being allotted the same or fewer processors on every step, while wasting only 10% of the processor cycles wasted by ABP.

1 Introduction

Multiprocessors are often used for multiprogrammed workloads where many parallel applications share the same machine. As Feitelson describes in his excellent survey [33], schedulers for these machines can be implemented using two levels: 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. 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 ***nonadaptive*** (called "static" in [21]), where a job runs on a fixed number of processors for its lifetime.

In this paper we present an adaptive thread scheduler, A-STEAL, which provides feedback about the job's parallelism to a space-sharing job scheduler. A-STEAL communicates this ***parallelism feedback*** by requesting pro-

This research was supported in part by the Singapore-MIT Alliance and NSF Grants ACI-0324974 and CNS-0305606. Yuxiong He is a Visiting Scholar at MIT CSAIL and a Ph.D. candidate at the National University of Singapore.

cessors from the job scheduler at regular intervals, called ***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.

Prior work on thread scheduling for multithreaded jobs has tended to focus on nonadaptive scheduling [7,8,13,19, 37, 52] or adaptive scheduling without parallelism feedback [3]. For jobs whose parallelism is unknown in advance and which may change during execution, nonadaptive scheduling may waste processor cycles [61], because a job with low parallelism may be allotted more processors than it can productively use. Moreover, in a multiprogrammed environment, nonadaptive scheduling may not allow a new job to start, because existing jobs may already be using most of the processors. Although adaptive scheduling without parallelism feedback allows jobs to enter the system, the job still may waste processor cycles if it is alloted more processors than it can use. Using an adaptive thread scheduler with parallelism feedback, if a job cannot use many processors effectively, the job scheduler can repurpose those processors to other jobs that can use them.

In previous work [2] with Wen Jing Hsu from Nanyang Technological University in Singapore, we described an adaptive greedy task scheduler, A-GREEDY, which uses a centralized greedy algorithm to schedule tasks on allotted processors while providing parallelism feedback to the job scheduler. A-GREEDY ensures that the job completes quickly without wasting many processor cycles. A-GREEDY is a centralized task scheduler, however, and although it is suitable for task scheduling of, for example, data-parallel jobs, where the central scheduler can be aware of all the available tasks at the current moment, it does not directly extend to decentralized thread scheduling.

In the decentralized setting of randomized work-stealing [3, 13, 35], a thread scheduler is not aware of all the available threads to execute at a given moment. Nevertheless, we show that the A-STEAL algorithm offers provable guarantees on both time and waste. We also present empirical evidence of the effectiveness of A-STEAL.

Like prior work on scheduling of multithreaded jobs [6,8,9,12,13,30,44,52], we model the execution of a multithreaded job as a dynamically unfolding directed acyclic graph (dag), where each node in the dag represents a unit-time instruction and an edge represents a serial dependence between nodes. 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. Each job has its own thread scheduler, which operates in an online manner, oblivious to the future characteristics of the dynamically unfolding dag.

Our scheduling model is similar to that in [2]. 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, or the number of processors the job is entitled to get for the quantum q. Our analysis of A-STEAL assumes that the job scheduler decides the availability of processors as an adversary in order to make the thread scheduler robust to different system environments and administrative policies. The number of processors the job receives for quantum q is the job's ***allotment*** $a_q = \min\{d_q, p_q\}$, which is the smaller of its 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. But, an adversarial job scheduler can prevent any thread scheduler from providing good speedup with respect to the mean availability $\overline{P}$ 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. [1]

We introduce 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

[1] Let $\bar{A}$ denote the mean of the processor allotment during the computation. We know that $T_1/\bar{A}$ is also a lower bound on the running time. However, a trivial thread scheduler that always requests one processor can achieve linear speedup respect to the mean of the processor allotment and waste none of processor cycles.

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.

This paper proves 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 average 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.

We implemented A-STEAL in a simulation environment using the DESMO-J [25] 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 [3]. We ran single jobs using both A-STEAL and ABP with the same availability profiles. We found that on moderate- to heavy-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 most 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 performance with respect to time and waste. Section 4 describes simulation setup and results for the performance of A-STEAL. Section 5 describes related work in adaptive and nonadaptive scheduling, and puts this paper in perspective of the prior work in the area. Finally, Section 6 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 estimates processor desire based on the job's history of utilization to provide parallelism feedback to the job scheduler. In this section, we describe A-STEAL and its desire-estimation heuristic.

To describe a thread scheduler, we require a few more definitions. In our job model, a ***thread*** is a chain of nodes with no branches. In addition, a node may ***spawn*** or create another thread, in which case there is an edge from the spawning node to the first node of the spawned thread. A node becomes ***ready*** when all its parents have been executed and a thread becomes ready when its first node becomes ready.

During a quantum, A-STEAL uses ***work-stealing*** [3, 13, 49] 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 [3][2]. In a work-stealing thread scheduler, every processor alloted to the job maintains a double-ended queue, or ***deque***, of ready threads for the job. When the current thread spawns a new thread, the processor pushes the current thread onto the top of the deque and begins working on the new thread. When the current thread completes or blocks, the processor pops the topmost thread off the deque to work on. If the deque of a processor is empty, however, the processor becomes a ***thief***, randomly picking a ***victim*** processor and ***stealing*** work from the bottom of the victim's deque. 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.

Making work-stealing adaptive

This 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 job scheduler obtains $a_q - a_{q-1}$ additional processors. Since the deques 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 deques may be nonempty. To deal with these deques, we use the concept of "mugging" [14]. When a processor runs out of work, instead of stealing immediately, it looks for a ***muggable*** deque, a nonempty deque that has no associated processor working on it. Upon finding a muggable deque, the thief ***mugs*** the deque by taking over the entire deque as its own. Thereafter, it works on the deque as if it were its own. If there are no

[2] These algorithms require some additional restrictions on the job. For example, they require that each node has an out-degree of at most 2. These restrictions apply to A-STEAL as well, depending on the work-stealing algorithm being used.

muggable deques, the thief steals normally. Data structures can be set up between quanta so that stealing and mugging can be accomplished in $O(1)$ time [57].

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. We assume without loss of generality that work-cycles, steal-cycles and mug-cycles all take single time step. We bound time and waste in terms of these elemental processor cycles. Cycles spent stealing and mugging are ***wasted***, and the total waste is the sum of the number of steal-cycles and mug-cycles during the execution of the job.

A-STEAL*'s desire-estimation heuristic*

A-STEAL's desire-estimation algorithm is based on the desire-estimation heuristic for the A-GREEDY algorithm 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, like A-GREEDY, 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 $q+1$.

The classification of quanta by A-STEAL as satisfied versus deprived is identical to A-GREEDY's strategy. A-STEAL compares 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.

Classifying a quantum as efficient versus inefficient is more complicated and varies from the way A-GREEDY classifies quanta. 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 $u_q \leq n_q$. A-GREEDY classifies quanta as efficient or inefficient based on the usage, but A-STEAL uses nonsteal usage instead. In addition, 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 90% to 95%. We call a quantum q ***efficient*** if $n_q \geq \delta L a_q$, 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.

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.

```
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.

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. The algorithm uses a desire-estimation heuristic similar to A-GREEDY's, except that A-STEAL uses nonsteal usage instead of usage. A-STEAL takes as input the quantum q, the utilization parameter δ, and the responsiveness parameter ρ. It then operates as follows:

- 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.
- 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 in-

creases the desire (line 6) for quantum q.

- 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.

3 Trim Analysis of A-STEAL

This section uses a trim analysis to analyze A-STEAL with respect to both time and waste. 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 the utilization parameter, and L is the quantum length, A-STEAL completes the job in time $T = O\left(T_1/\widetilde{P} + T_\infty + L\lg P + L\ln(1/\epsilon)\right)$, with probability at least $1-\epsilon$, where $\widetilde{P}$ is the $O(T_\infty + L\lg P + L\ln(1/\epsilon))$-trimmed availability. This bound implies that A-STEAL achieves linear speed-up on all the time steps excluding $O(T_\infty + L\lg P + L\ln(1/\epsilon))$ time steps with highest processor availability. Moreover, A-STEAL guarantees that the total number of processor cycles wasted during the job's execution is $W = O(T_1)$.

We prove these bounds using a trim analysis [2]. We label each quantum as either ***accounted*** or ***deductible***. Accounted quanta are those with $n_q \geq L\delta p_q$, 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 possibly available during the quantum. Conversely, the deductible quanta are those where $n_q < L\delta p_q$. 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 to a three-way classification of quanta as inefficient, efficient and satisfied, and efficient and deprived:

- **Inefficient:** In 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. Thus, 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.

Time analysis

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 in 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 deques, 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,* A-STEAL *reduces the length of a job's remaining critical path by at least $(1-\delta)L/6$ with probability more than $1/4$ in an inefficient quantum.*

Proof. Let q be an inefficient quantum. A processor with an empty deque steals only when it cannot mug a deque, and hence, all the steal-cycles in quantum q occur when the number of nonempty deques is at most the allotment a_q. Therefore, by Lemma 1, $3a_q$ steal-cycles suffice to reduce the critical path 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.[3] 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 in 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 [23, p. 1111], which says that for a nonnegative

[3] 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.

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, the probability exceeds $1/4$ that quantum q contains at least $j/2$ good rounds. Since each good round reduces the critical path by at least 1, with probability greater than $1/4$, the critical path is reduced during quantum q by at least $j/2 = ((1-\delta)L/3)/4 = (1-\delta)L/6$. □

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

Proof. Let I be the set of inefficient quanta. Define an inefficient quantum q as ***productive*** if it reduces the critical path 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 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 $\Pr\{Y < (1-\lambda)E[Y]\} < \exp(-\lambda^2 E[Y]/2)$ [51, 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 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 desire.

Lemma 4 *Suppose that* A-STEAL *schedules a job on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, before any quantum q, the desire d_q of the job is at most ρP.* □

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 had at least 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 the utilization parameter, and L is the quantum length, then the schedule produces at most $48T_\infty/((1-\delta)L) + \log_\rho P + 16\ln(1/\epsilon)$ 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 on a machine. Let I and C denote the set of inefficient quanta and the set of efficient and satisfied quanta 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 reduced the desire to d_q/ρ. Since the desire later increased 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 the 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 deques are declared muggable. The total number M of mug-cycles is at most the number of muggable deques during the job's execution. Since the allotment reduces by at most $a_q - 1$ from quantum q to quantum $q+1$, there are $M \le \sum_q (a_q - 1) < \sum_q a_q$ 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 $n_r \ge L\delta a_r$ and allotment $a_r = d_r$. Thus, we have $n_r + n_q \ge L\delta a_r = ((L\delta)/(1+\rho))(a_r + \rho a_r) = ((L\delta)/(1+\rho))(a_r + \rho d_r) \ge ((L\delta)/(1+\rho))(a_r + a_q)$, since $a_q \le d_q$ 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 $n_q \ge L\delta a_q$ for these quanta. Let $N = \sum_q n_q$ be the total number of nonsteal-cycles during the job's execution. We have $N = \sum_q n_q \ge ((L\delta)/(1+\rho)) \sum_q a_q \ge ((L\delta)/(1+\rho))M$. 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, $T_1 = N - M \ge ((L\delta)/(1+\rho))M - M = ((L\delta - 1 - \rho)/(1+\rho))M$, which yields $M \le ((1+\rho)(L\delta - 1 - \rho))T_1$. □

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 the 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 mean availability on 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 $n_q \ge L\delta a_q$. Thus, we have $N = \sum_{q \in A \cup D} n_q \ge \sum_{q \in A} n_q \ge \sum_{q \in A} \delta L p_q = \delta L |A| P_A$, and hence, we obtain

$$|A| \le N/(L\delta P_A) \,. \tag{1}$$

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 $M \le ((1+\rho)(L\delta - 1 - \rho))T_1$ mug-cycles, we have $N \le T_1 + M < T_1(1 + (1+\rho)/(L\delta - 1 - \rho))$. 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 the 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*

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

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 is a trim analysis. Let A be the set of accounted quanta, and D be the set of deductible quanta. Lemmas 3 and 6 show that there are at most $|D| = O(T_\infty/((1-\delta)L) + \log_\rho P + \ln(1/\epsilon))$ deductible quanta, and hence at most $L|D| = O(T_\infty/(1-\delta) + L \log_\rho P + L \ln(1/\epsilon))$ time steps belong to deductible quanta. We have that $P_A \ge \widetilde{P}$, 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 $(T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$, 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 the utilization parameter, and L is the quantum length, then* A-STEAL *completes the job in expected time $\mathrm{E}[T] \le O(T_1/\widetilde{P} + T_\infty + L \lg P)$, 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. □

The analysis leading to Theorem 10 and its corollary makes two assumptions. First, we assume that the scheduler knows exactly how many steal-cycles have occurred in the quantum. Second, we assume that the processors can find the muggable deques instantaneously. We shall now relax these assumptions and show that they do not change the asymptotic running time of A-STEAL.

A scheduling system can implement the counting of steal-cycles in several ways that impact our theoretical bounds only minimally. For example, if the number of processors in the machine P is smaller than the quantum length L, then the system can designate one processor to collect all the information from the other processors at the end of each quantum. Collecting this information increases the time bound by a multiplicative factor of only $1 + P/L$. As a practical matter, one would expect that $P \ll L$, since scheduling quanta tend to be measured in tens of milliseconds and processor cycle times in nanoseconds or less, and thus the slowdown would be negligible. Alternatively, one might organize the processors for the job into a tree structure so that it takes $O(\lg P)$ time to collect the total number of steal-cycles at the end of each quantum. The tree implementation introduces a multiplicative factor of $1+(\lg P)/L$ to the job's execution time, an even less significant overhead.

The assumption that it takes constant time to find a muggable deque, can be relaxed in a similar manner. One option is to mug serially, that is, while there is a muggable deque, all processors try to mug according to a fixed linear order. 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 again does not affect the running time bound by much. Alternatively, to obtain a better theoretical bound, we could use a counting network [4] with width P to implement the list of muggable deques, 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.

Waste analysis

The next theorem bounds the waste, which is the total number of mug- and steal-cycles.

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 the utilization parameter, and L is the quantum length, then* A-STEAL *wastes at most*

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

processor cycles in the course of computation.

Proof. Let M be the total number of mug-cycles, and let S be the total number of steal-cycles, and hence, we have $W = S + M$. Since Lemma 8 bounds M, we only need to bound S, which we do using an accounting argument 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 innefficient and efficient quanta, respectively.

Inefficient quanta: Lemma 7 shows that every inefficient quantum q with desire d_q has a distinct corresponding 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 $n_r \ge L\delta a_q/\rho$ and its allocation is $a_r = d_r$. Therefore, we have $n_r \ge L\delta a_r = L\delta d_r = L\delta d_q/\rho \ge L\delta a_q/\rho$. Let s_q be the number of steal-cycles in quantum q. Since there are at most La_q total processor cycles in the quantum, we have $s_q \le La_q \le \rho n_r/\delta$, 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 $S_{\text{ineff}} \le (\rho/\delta)(T_1 + M)$.

Efficient quanta: On any efficient quantum q, the job has 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 $S_{\text{eff}} \le ((1-\delta)/\delta)(T_1 + M)$.

The total waste is therefore $W = S + M = S_{\text{ineff}} + S_{\text{eff}} + M \le (T_1 + M)(1+\rho-\delta)/\delta + M$. Since Lemma 8 provides $M < T_1(1+\rho)/(L\delta - 1 - \rho)$, the theorem follows. □

Interpretation of the bounds

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

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

This reformulation allows us to more easily see the tradeoffs due to the setting of the δ and ρ parameters.

In the time bound, 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 at the same time. In addition, 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.

To see how these settings affect the waste bound, consider the waste bound as two parts, where the waste due to steal-cycles is $S \leq ((1+\rho-\delta)T_1)/\delta$ and the is the waste due to mug-cycles is $M = O(1/L)T_1$. We can see that the waste due to mug-cycles is just a tiny fraction compared to the work T_1. Thus, these bounds indicated that adaptive scheduling with parallelism feedback can be supported without too much regard for the bookkeeping efficiency 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, however, that the number of steal-cycles in efficient steps is bounded by $((1-\delta)/\delta)T_1$, which is a small fraction of S. Thus, most of the waste bound is accounted for by steal-cycles in inefficient quanta. Our analysis assumes that the job scheduler is an adversary, creating as many inefficient quanta as possible. Of course, job schedulers are generally not adversarial. Thus, in practice, we might expect the waste to be a much smaller fraction of T_1 than our bounds show. Section 4 describes some experiments that validate this hypothesis.

4 Experimental Evaluation

We built a discrete-time simulator using DESMO-J [25] to evaluate the performance of A-STEAL. Some of our experiments benchmarked A-STEAL against ABP [3], an adaptive thread scheduler that does not supply parallelism feedback to the job scheduler. This section describes our simulation setup and the results of the experiments.

We conducted four sets of experiments on the simulator with synthetic jobs. Our results are summarized below:

- The ***time experiments*** investigated the performance of A-STEAL on over 2300 job runs. A linear-regression analysis of the results provides evidence that the coefficients on the number of accounted and deductible steps are considerably smaller than those in our theoretical bounds. A second linear-regression analysis indicates that A-STEAL completes jobs on average in at most twice the optimal number of time steps, which is the same bound provided by offline greedy scheduling [19, 37].
- The ***waste experiments*** are designed to measure the waste incurred A-STEAL in practice and compare the observed waste to the theoretical bounds. Our experiments indicate that the waste is almost insensitive to the parameter settings and is a tiny fraction (less than 10%) of the work for jobs with high parallelism.
- The ***time-waste experiments*** compare the completion time and waste of A-STEAL with ABP [3] by running single jobs with predetermined availability profiles. These experiments indicate that on large machines, when the mean availability $\overline{P}$ is considerably smaller than the number of processor P in the machine, A-STEAL completes jobs faster then ABP while wasting fewer processor cycles than ABP. On medium-sized machines, when $\overline{P}$ is of the same order as P, ABP completes jobs somewhat faster than A-STEAL, but it still wastes many more processor cycles than A-STEAL.
- The ***utilization experiments*** compare the utilization of A-STEAL and ABP when many jobs with varying characteristics are using the same multiprocessor resource. The experiments provide evidence that on moderately to heavily loaded large machines, A-STEAL consistently provides a higher utilization than ABP for a variety of job mixes.

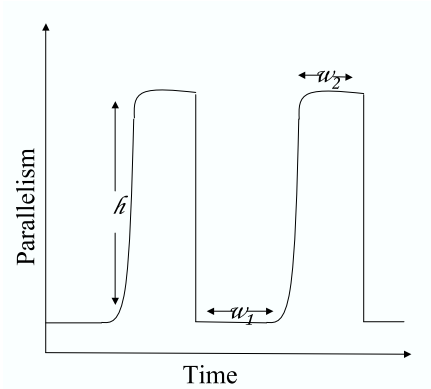


Figure 2: The parallelism profile (for 2 iterations) of the jobs used in the simulation.

Simulation setup

We built a Java-based discrete-time simulator using DESMO-J [25]. The simulator implements four major entities — processors, jobs, thread schedulers, and job schedulers — and simulates their interactions in a two-level scheduling environment. We modeled jobs as dags, which are executed by the thread scheduler. When a job is submitted to the simulated multiprocessor system, an instance of a thread scheduler is created for the job. The job scheduler allots processors to the job, and the thread scheduler simulates the execution of the job using work-stealing. The simulator operates in discrete time steps: a processor can complete either a work-cycle, steal-cycle, or mug-cycle during each time step. We ignored the overheads due to the reallocation of processors in the simulation.

We tested synthetic multithreaded jobs with the parallelism profile shown in Figure 2. Each job alternates between a serial phase of length w_1 and a parallel phase

(with h-way parallelism) of length w_2. The average parallelism of the job is approximately $(w_1 + hw_2)/(w_1 + w_2)$. By varying the values of w_1, w_2, h, and the number of iterations, we can generate jobs with different work, critical-path lengths, and frequency of the change of the parallelism.

In the time-waste experiments and the utilization experiments, we compared the performance of A-STEAL with that of another thread scheduler, ABP [3], an adaptive thread scheduler that does not provide parallelism feedback to the job scheduler. In these experiments, ABP is always allotted all the processors available to the job. ABP uses a nonblocking implementation of work stealing and always maintains P deques. When the job scheduler allots $a_q = p_q$ processors in quantum q, ABP selects a_q deques uniformly at random from the P deques, and the allotted processors start working on them. Arora, Blumofe, and Plaxton [3] prove that ABP completes a job in expected time

$$T = O(T_1/\overline{P} + PT_\infty/\overline{P}) \ , \tag{6}$$

where $\overline{P}$ is the average number of processors allotted to the job by the job scheduler. Although Arora *et al.* provide no bounds on waste, one can prove that ABP may waste $\Omega(T_1 + PT_\infty)$ processor cycles in an adversarial setting.

We implemented three kinds of job schedulers — profile-based, equipartitioning [48], and dynamic-equipartitioning [48]. A profile-based job scheduler was used in the first four sets of experiments, and both equipartitioning and dynamic-equipartitioning job schedulers were used in the utilization experiment. An ***equipartitioning*** (EQ) job scheduler simply allots the same number of processors to all the active jobs in the system. Since ABP provides no parallelism feedback, EQ is a suitable job scheduler for ABP's scheduling model. ***Dynamic equipartitioning*** (DEQ) is a dynamic version of the equipartitioning policy, but it requires parallelism feedback. A DEQ job scheduler maintains an equal allotment of processors to all jobs with the constraint that no job is allotted more processors than it requests. DEQ is compatible with A-STEAL's scheduling model, since it can use the feedback provided by A-STEAL to decide the allotment.

For the first three experiments — time, waste, and time-waste — we ran a single job with a predetermined ***availability profile***: the sequence of processor availabilities p_q for all the quanta while the job is executing. For the ***profile-based*** job scheduler, we precomputed the availability profile, and during the simulation, the job scheduler simply used the precomputed availability for each quantum. We generated three kinds of profiles:

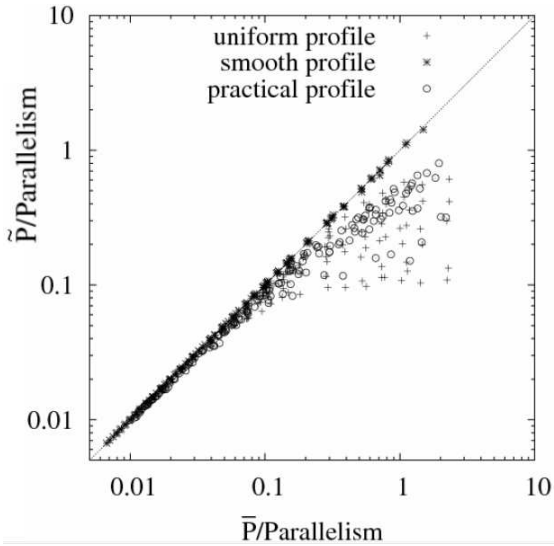
- **Uniform profiles:** The processor availabilities in these profiles follow the uniform distribution in the range from 1 to P, the maximum number of processors in the system. These profiles represent near-adversarial conditions for A-STEAL, because the availability for one quantum is unrelated to the availability for the previous quantum.
- **Smooth profiles:** In these profiles, the change of processor availabilities from one scheduling quantum to the next follows a standard normal distribution. Thus, the processor availability is unlikely to change significantly over two consecutive quanta. These profiles attempt to model situations where new arrivals of jobs are rare, and the availability changes significantly only when a new job arrives.
- **Practical profiles:** These availability profiles were generated from the workload archives [31] of various computer clusters. We computed the availability at every quantum by subtracting the number of processors that were being used at the start of the quantum from the number of processors in the machine. These profiles are meant to capture the processor availability in practical systems.

A-STEAL requires certain parameters as input. The responsiveness parameter is $\rho = 1.5$ for all the experiments. For all experiments except the waste experiments, the utilization parameter is $\delta = 0.8$. We vary δ in the waste experiments. The quantum length L represents the time between successive reallocations of processors by the job scheduler and is selected to amortize the overheads due to the communication between the job scheduler and the thread scheduler and the reallocation of processors. In conventional computer systems, a scheduling quantum is typically between 10 and 20 milliseconds. Our experience with the Cilk runtime system [38] indicated that a steal/mug-cycle takes approximately 0.5 to 5 microseconds, indicating that the quantum length L should be set to values between 10^3 and 10^5 time steps. Our theoretical bounds indicate that as long as $T_\infty \gg L \log P$, the length of L should have little effect on our results. Due to the performance limitations of our simulation environment, however, we were unable to run very long jobs: most have a critical-path length on the order of only a few thousand time steps. Therefore, to satisfy the condition that $T_\infty \gg L \log P$, we set $L = 200$.

Time experiments

The running-time bounds proved in Section 3, though asymptotically strong, have weak constants. The time experiments were designed to investigate what constants would occur in practice and how A-STEAL performs compared to an optimal scheduler. We performed linear-regression analysis on the results of 2331 job runs using many availability profiles of all three kinds to answer these questions.

Our first time experiment uses the bounds in Inequality (2) as a simple model, as in the study [11]. Assuming that equality holds and disregarding smaller terms, the



R7cm

Figure 3: Comparing the (true) mean availability $\overline{P}$ with the trimmed availability $\widetilde{P}$ using three availability profiles. Each data point represents a job execution for which the mean availability and trimmed availability were measured. These values were normalized by dividing by the parallelism T_1/T_∞ of the job. When the parallelism satisfies $T_1/T_\infty > 5\overline{P}$, the experiments indicate that for all profiles, the trimmed availability is a good approximation of the mean availability. All these experiments used $\delta = 0.8$ and $\rho = 1.5$.

model estimates performance as

$$T \approx c_1 T_1/\widetilde{P} + c_\infty T_\infty \ , \tag{7}$$

where $c_1 > 0$ is the ***work overhead*** and $c_\infty > 0$ is the ***critical-path overhead***. When $\delta = 0.8$, $\rho = 1.5$, and $L = 200$, the coefficients for the asymptotic bounds in Inequality (2) turn out to be $1.26 < c_1 < 1.27$ and $c_\infty = 480$, but a direct analysis of expectation can improve the bound on critical-path overhead to $c_\infty = 60$. Since the critical-path overhead c_∞ is large, the bound indicates that A-STEAL may not provide linear speedup except when $T_1/T_\infty \gg 60\widetilde{P}$. Moreover, on accounted time steps, A-STEAL might not provide perfect linear speedup, since the work overhead is $1.26 > 1$.

In practice, however, we should not expect these large overheads to materialize. First, our analysis is focused on asymptotic bounds, and hence we use bounding techniques such as Markov's inequality and Chernoff bounds which are not necessarily tight. Second, our analysis assumes that the job completes the minimum number of work-cycles in each quantum, specifically, 0 on a deductible quantum and $\delta L a_q$ on an accounted quantum with allotment a_q.

Our first linear-regression analysis fits the running time of the 2331 job runs to Equation (7). The least-squares fit to the data to minimize relative error yields $c_1 = 0.960 \pm 0.003$ and $c_\infty = 0.812 \pm 0.009$ with 95% confidence. The R^2 correlation coefficient of the fit is 99.4%. Since $c_\infty = 0.812 \pm 0.009$, on average the jobs achieved linear speedup when $T_1/T_\infty \gg \widetilde{P}$. In addition, since $c_1 = 0.960 \pm 0.003$ A-STEAL achieves almost perfect linear speedup on the accounted steps. The fact that $c_1 < 1$ stems from the fact that jobs performed some work during the deductible steps.

We performed a second set of regression tests on the same set of jobs to compare the performance of A-STEAL with an optimal scheduler. We fit the job data to the curve

$$T = \hat{c}_1 T_1/\overline{P} + \hat{c}_\infty T_\infty \ . \tag{8}$$

The analysis yields $\hat{c}_1 = 0.992 \pm 0.003$ and $c_\infty = 0.911 \pm 0.008$ with an R^2 correlation coefficient of 99.4%. Both $T_1/\overline{P}$ and T_∞ are lower bounds on the job's running time, and thus an optimal scheduler requires at least $\max\{T_1/\overline{P}, T_\infty\} \geq (T_1/\overline{P} + T_\infty)/2 \geq (\hat{c}_1 T_1/\overline{P} + \hat{c}_\infty T_\infty)/2$ time steps, since $\hat{c}_1 < 1$ and $\hat{c}_\infty < 1$. Thus, on average A-STEAL completed the jobs within at most twice the time of an optimal scheduler.

The two models 7 and 8 both predict performance with high accuracy, and yet $\widetilde{P}$ and $\overline{P}$ can diverge significantly. To resolve this paradox, we compared $\widetilde{P}$ and $\overline{P}$ on the job runs. Figure 3 shows a graph of the results, where $\widetilde{P}$ and $\overline{P}$ are each normalized by dividing by the parallelism T_1/T_∞ of the job. The diagonal line is the curve $\widetilde{P} = \overline{P}$.

If a job has parallelism $T_1/T_\infty > 5\overline{P}$ (data points on the left), the experiment indicates that for all three kinds of availability profiles, we have $\widetilde{P} \approx \overline{P}$. In this case, we have $T_1/\widetilde{P} \approx T_1/\overline{P}$ and $T_1/\overline{P} \gg T_\infty$, which implies that the first terms in Equations (7) and (8) are nearly identical and dominate the running time. On the other hand, if a job has small parallelism (data points on the right), the values of $\widetilde{P}$ and $\overline{P}$ diverge and the divergence depends on the availability profile used. In this region, the running time is dominated by the critical-path length T_∞, however, and thus, the divergence of $\widetilde{P}$ and $\overline{P}$ has little influence on the running time.

Waste experiments

Our theoretical analysis shows that the waste exhibited by A-STEAL is at most $O(T_1)$. The constant hidden in the O-notation depends on the parameter settings. In our first waste experiment, we varied the value of the utilization parameter δ to understand the relationship between the waste and the setting of δ. For our second experiment, we investigated whether the waste incurred by a job depends on the job's parallelism.

The proof of Theorem 12 shows that the number of processor cycles wasted by a job is $((1-\delta)/\delta)T_1$ on efficient quanta and approximately $(\rho/\delta)T_1$ on inefficient quanta. Substituting $\delta = 0.8$ and $\rho = 1.5$, A-STEAL could waste as many as $0.25T_1$ processor cycles on efficient quanta and as many as $1.875T_1$ processor cycles on inefficient

quanta. Since this analysis assumes that the job scheduler is an adversary and that the job completes the minimum number of work-cycles in each quantum, we should not expect these constants to materialize in practice.

We measured the waste for 300 jobs, most of which had parallelism $T_1/T_\infty > 5\overline{P}$, for $\delta = 0.5, 0.6, \ldots, 1.0$. The job runs used many availability profiles drawn equally from the three kinds. Figure 4 shows the average of waste normalized by the work T_1 of the job. For comparison we plotted the normalized theoretical bound Inequality (5) for the total waste and the normalized bound $((1-\delta)/\delta)T_1$ for the waste on efficient quanta. As the figure shows (although the curve is barely distinguishable from the x-axis), the observed waste is less than 10% of the work T_1 for most values of δ and is considerably less than the theoretical bounds predict. Moreover, the waste seems to be quite insensitive to the particular value of δ.

We also ran an experiment to determine whether parallelism has an effect on waste. The bound in Inequality (5) does not depend on the parallelism T_1/T_∞ of the job, but only on the work T_1. For the 2331 job runs used in the time experiments, we measured the waste versus parallelism. Since waste is insensitive to δ, all jobs used the value $\delta = 0.8$. Figure 5 graphs the results. As can be seen in the figure, the higher the parallelism, the lower the waste-to-work ratio. The reason is that when the parallelism is high, the job can usually use most of the available processors without readjusting its desire. When the parallelism is low, however, the job's desire must track its parallelism closely to avoid waste. This situation is where A-STEAL is most effective, as the job pushes the theoretical waste bounds to their limit.

Time-waste experiments

The time-waste experiments were designed to compare A-STEAL with ABP, an adaptive thread scheduler with no parallelism feedback. For our first experiment, we ran A-STEAL and ABP to execute 756 job runs on a simulated machine with $P = 512$ processors. Each head-to-head run used one of two practical availability profiles, one with $\overline{P} = 30$ and one with $\overline{P} = 60$. We measured the time and waste of A-STEAL and ABP for each run. Our second experiment was similar, but it used only $P = 128$ processors in the simulated machine over 330 job runs. Whenever the availability exceeded 128, which was not often, we chopped the availability to 128.

Figure 6 shows the ratio of ABP to A-STEAL with respect to both time and waste as a function of the mean availability $\overline{P}$, normalized by dividing by the parallelism T_1/T_∞. This experiment shows that A-STEAL completed jobs about twice as fast as ABP while wasting only about 10% of the processor cycles wasted by ABP. Not surprisingly, A-STEAL wastes fewer processor cycles than ABP, since A-STEAL uses parallelism feedback to limit possible excessive allotment. Paradoxically, however, A-STEAL completes jobs faster than ABP, even though A-STEAL's allotment in every quantum is at most that of ABP, which is always allotted all the available processors.

ABP's slow completion is due to how ABP manages its ready deques. In particular, ABP has no mechanism for increasing and decreasing the number r of ready deques, and it maintains $r = P$ deques throughout the execution. Randomized work-stealing algorithms require $\Theta(r)$ steal-cycles to reduce the length of the critical path by 1 in expectation. Consequently, if r is large, each steal-cycle becomes less effective, and the job's progress along its critical path slows. Thus, if the job has small or moderate parallelism (data points on the right), the critical-path length dominates the running time. If the job has large parallelism (data points on the left), however, the impact is less. In contrast, A-STEAL continues to make good progress along the critical path, regardless of parallelism, by reducing the number of deques according to its allotment.

This paradox can also be understood by using the model from Equation (7) for A-STEAL and an analogous model based on Equation (6) for ABP. Let us consider three cases:

- $T_1/T_\infty < \overline{P} \ll P$ (data points on the right): Whereas A-STEAL completes the job in $\Theta(T_\infty)$ time, ABP requires $\Theta(PT_\infty/\overline{P})$ time.
- $\overline{P} < T_1/T_\infty \ll P$ (data points in the middle): A-STEAL provides linear speedup since $T_1/T_\infty > \widetilde{P}$, but ABP does not, since $T_1/T_\infty \ll P$.
- $P < T_1/T_\infty$ (data points on the left): Both provide linear speedup in this range.

Since ABP performed relatively poorly when P is large compared to $\overline{P}$, our second experiment investigated the case when P is closer to $\overline{P}$. Figure 7 shows the results on 330 job runs on a simulated machine with $P = 128$. In this case, ABP performs slightly better than A-STEAL with respect to time and slightly worse with respect to waste. Since $\overline{P} \approx P$, the two models coincide, and ABP and A-STEAL perform comparably. Therefore, on small machines, where the disparity between $\overline{P}$ and P cannot be very great, the advantage of parallelism feedback is diminished, and ABP may yet be an effective thread-scheduling algorithm.

Utilization experiments

The utilization experiments compared A-STEAL with ABP on a large server where many jobs are running simultaneously and jobs arrive and leave dynamically. We implemented job schedulers to allocate processors amongst various jobs: dynamic equipartitioning [48] for A-STEAL and equipartitioning [63] for ABP. We simulated a 1000-processor machine for about 10^6 time steps, where jobs had a mean interarrival time of 1000 time steps. We compared the utilization provided by A-STEAL and ABP over

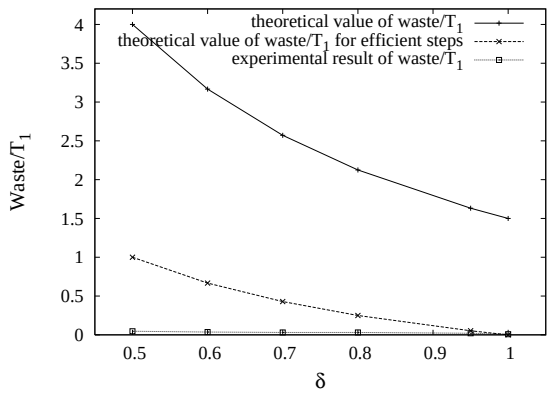


Figure 4: Comparing the theoretical and practical waste (normalized by T_1) using A-STEAL for various values of the utilization parameter δ. The top line shows the total theoretical waste, the next line shows the theoretical waste on efficient quanta, and the bottom line shows the observed waste. The observed waste appears to be almost insensitive to the value of δ and is much smaller than the theoretical waste.

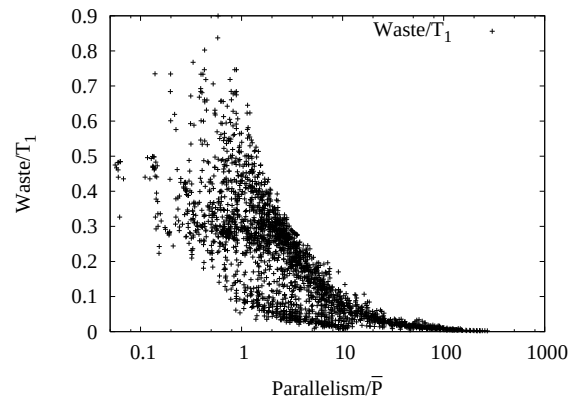


Figure 5: How waste varies with parallelism. When $T_1/T_\infty > 10\overline{P}$, that is, the job's parallelism significantly exceeds the average availability, the observed waste was only a tiny fraction of the work T_1. For jobs with small parallelism, the waste showed a large variance but never exceeded the work T_1 in any of our runs. The utilization parameter was $\delta = 0.8$ for all job runs.

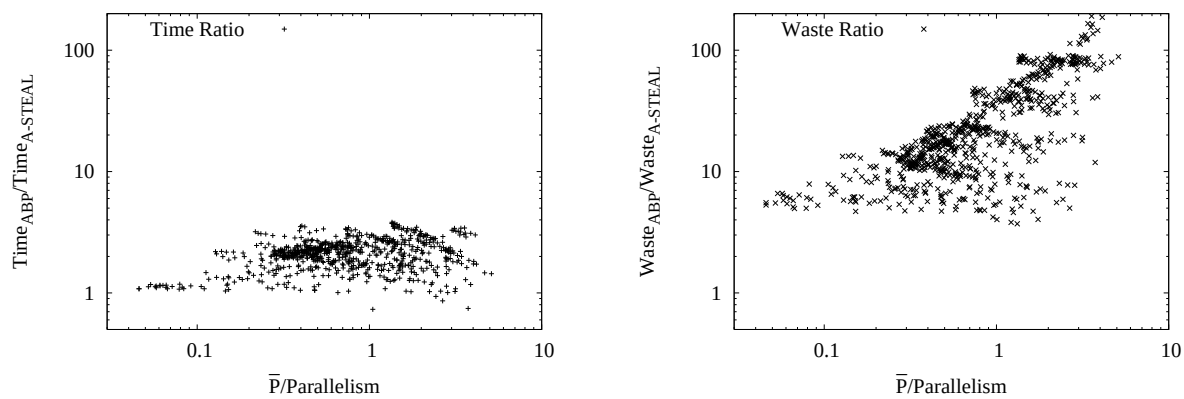


Figure 6: Comparing the time and waste of A-STEAL against ABP when $P = 512$ and $\overline{P} = 30, 60$. In this experiment, where P exceeds $\overline{P}$ by a significant margin, A-STEAL completes jobs about twice as fast as ABP while wasting less than 10% of the processor cycles wasted by ABP.

time.

It was unclear to us what distribution the parallelism and the critical-path lengths should follow. Although many workload models for parallel jobs have been studied [22, 26, 32, 47, 59], none appear to apply directly to multithreaded jobs. Some studies [42, 43, 46] claim that the sizes of Unix jobs follow a heavy-tailed distribution. Without clear guidance, we decided to try various distributions, and as it turned out, our results were fairly insensitive to which we chose.

We considered 9 sets of jobs using three distributions on each of the parallelism and the critical-path length. The means of the distributions were chosen so that jobs arrive faster than they complete and the load on the machine progressively increases. Thus, we were able to measure the utilization of the machine under various loads. The three distributions we explored were the following:

- **Uniform distribution (U):** The critical-path length is picked uniformly from the range $1,000$ to $99,000$. The parallelism is generated uniformly in the range $[1, 80]$.

- **Heavy-tailed distribution 1 (HT1):** We used a Zipf's-like [65] heavy-tailed distribution where the probability of generating x is proportional to $1/x$. In

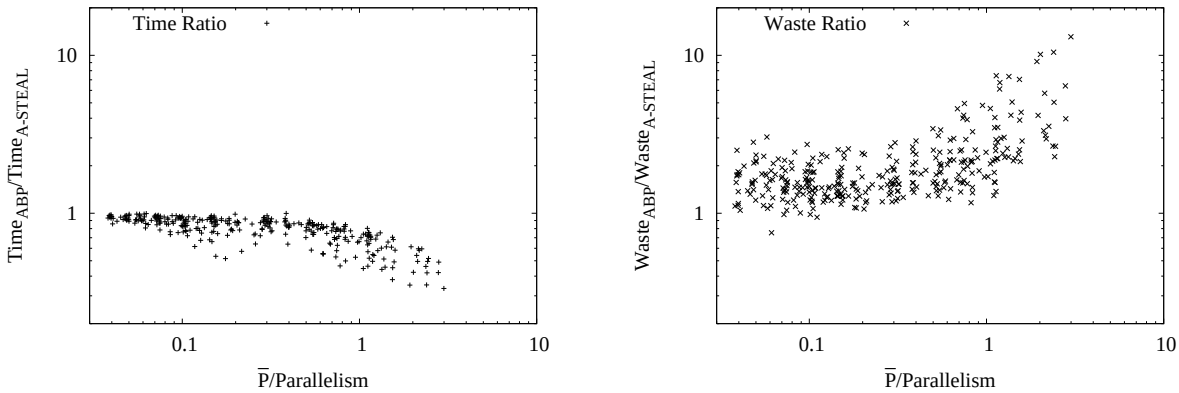


Figure 7: Comparing the time and waste of A-STEAL against ABP when $P = 128$ and $\overline{P} = 30, 60$. In this experiment, where P and $\overline{P}$ are closer in magnitude, A-STEAL runs slightly slower than ABP, but it still tends to waste fewer processor cycles than ABP.

our experiments, the distribution for parallelism has mean about 36, and the distribution for critical-path length has mean $50,000$.

- **Heavy-tailed distribution 2 (HT2):** In this distribution, the probability of generating x is proportional to $1/\sqrt{x}$. In our experiments, the distribution for parallelism has mean 36, and the distribution for critical-path length has mean $50,000$.

Of the 9 possible sets of jobs, we ran 6 experiments using parallelism and critical-path lengths drawn from U/U, U/HT1, HT1/U, HT1/HT1, HT2/U, and HT2/HT2. For all these experiments, the comparison between A-STEAL+DEQ and ABP+EQ followed the same qualitative trends. We broke time into intervals of 2000 time steps and measured the utilization — the fraction of processor cycles spent working — for each interval. Figure 8 shows the utilization as a function of time (log-scale) for the U/U experiment on the left and for HT1/HT1 on the right. As can be seen in both figures, ABP+EQ starts out with a higher utilization, since A-STEAL+DEQ initially requests just one processor. Before 10% of the simulation has elapsed, however, A-STEAL+DEQ overtakes ABP+EQ with respect to the utilization and then consistently provides a higher utilization. Although the figure does not show it, the mean completion time of jobs under ABP+EQ is nearly 50% slower than those under A-STEAL+DEQfor both these distributions.

5 Related work

This section discusses related work on adaptive and nonadaptive schedulers for multithreaded jobs. Work in the area has centered on either job schedulers or on nonadaptive thread schedulers. 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 [41]. Many variants have been implemented since then [34, 40, 49], and it has been analyzed in the context of load balancing [56], backtrack search [45] 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 [3] 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 [10,17,35] and empirical studies show that work-stealing schedulers are scalable and practical [16, 35].

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading, primarily by Blumofe and his coauthors [3,15,16,18]. In this work, the thread scheduler uses randomized work-stealing strategy to schedule threads on available processors but does does not provide the feedback about the job's parallelism to the job scheduler. The work 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 [3] show that the ABP task 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. In all these experiments, the job parallelism T_1/T_∞ is much greater than 8.

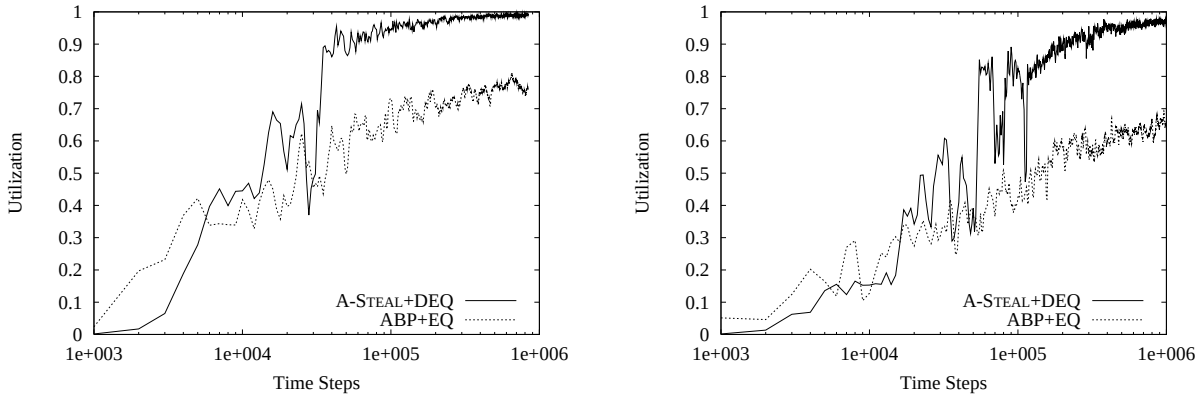


Figure 8: Comparing the utilization over time of A-STEAL+DEQ and ABP+EQ. In the left figure, both the critical-path length and the parallelism follow the uniform distribution, and in the right figure, they follow the HT1 distribution.

Adaptive task scheduling with parallelism feedback has been studied empirically in [57, 60, 62]. 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 thread scheduler with parallelism feedback.

In contrast to adaptive thread schedulers, adaptive job schedulers have been studied extensively, both empirically [21, 27, 36, 48, 53–55, 58, 64] and theoretically [5, 24, 28, 29, 39, 50]. McCann, Vaswani, and Zahorjan [48] 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 [39] 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 [24] proved a similar result for mean response time for multiphase jobs regardless of their arrival times. Song [60] proves that a randomized distributed strategy can implement dynamic equipartitioning.

6 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 empirical studies provide evidence that A-STEAL performs better than ABP when the machine has a large number of processors and has many jobs running on it. The reason is that A-STEAL uses parallelism feedback and the mugging mechanism to reclaim abandoned deques. One can imagine implementing ABP, which does not use parallelism feedback, but which does use a mugging mechanism. Although adding a mugging mechanism to ABP may not improve its performance theoretically, such a modification to ABP might improve its performance as a matter of practice. We are currently studying ABP with this modification in order to evaluate the importance of parallelism feedback itself in adaptive work-stealing.

Acknowledgments

We thank the members of the Supercomputing Technologies group at MIT CSAIL for helpful discussion and comments. Siddhartha Sen, also of MIT CSAIL, contributed to the implementation of an adaptive scheduler for Cilk, which inspired this research.

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. 2005.

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

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

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

[6] 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.

[7] 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.

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

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

[10] 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.

[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. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[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] Walfredo Cirne and Francine Berman. A model for moldable supercomputer jobs. In *IPDPS*, page 59, Washington, DC, USA, 2001. IEEE Computer Society.

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

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

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

[26] Allen B. Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.

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

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

[29] 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.

[30] 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.

[31] Dror Feitelson. Parallel workloads archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[32] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, volume 1162, pages 89–110. Springer, 1996.

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

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

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

[36] 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.

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

[38] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual*. MIT Laboratory for Computer Science, 2001.

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

[40] 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.

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

[42] Mor Harchol-Balter. The effect of heavy-tailed job size. distributions on computer system design. In *Conference on Applications of Heavy Tailed Distributions in Economics*, 1999.

[43] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

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

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

[46] Will Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. In *SIGMETRICS*, pages 54–69, New York, NY, USA, 1986. ACM Press.

[47] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[48] 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.

[49] 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.

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

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

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

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

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

[59] Kenneth C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, 1994.

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

[61] Mark S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *IPPS*, pages 219–238, 1995.

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

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

[64] 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.

[65] George K. Zipf. *Human Behaviour and the Principle of Least-Effort*. Addison-Wesley, 1949.