An Empirical Evaluation of 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

Abstract

A-STEAL *is a provably good adaptive work-stealing thread scheduler that provides parallelism feedback to a multiprocessor job scheduler.* A-STEAL *uses a simple multiplicative-increase, multiplicative-decrease algorithm to provide continual parallelism feedback to the job scheduler in the form of processor requests. Although jobs scheduled by* A-STEAL *can be shown theoretically to complete in near-optimal time asymptotically while utilizing at least a constant fraction of the allotted processors, the constants in the analysis leave it open on whether* A-STEAL *works well in practice. This paper confirms with simulation studies that* A-STEAL *performs well when scheduling adaptively parallel work-stealing jobs on large-scale multiprocessors.*

Our studies monitored the behavior of A-STEAL *on a simulated multiprocessor system using synthetic workloads. We measured the completion time and waste of* A-STEAL *on over* 2300 *job runs using a variety of processor availability profiles. Linear-regression analysis indicates that* A-STEAL *provides almost perfect linear speedup. In addition,* 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 which does not employ parallelism feedback. On moderately to heavily 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. We compared the utilization of* A-STEAL *and ABP when many jobs with varying characteristics are using the same multiprocessor. These experiments provide evidence that* A-STEAL *consistently provides higher utilization than ABP for a variety of job mixes.*

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.

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 [31], 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 [18]), where the number of processors allotted to a job may change while the job is running, or ***nonadaptive*** (called "static" in [18]), where a job runs on a fixed number of processors for its lifetime.

The two-level scheduling model assumes that time is broken into a sequence of equal-size ***scheduling quanta*** $1, 2, \ldots$, each consisting of L time steps. 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. Between quanta $q-1$ and q, the thread scheduler determines its job's ***desire*** d_q, which is the number of processors the job wants for quantum q. Then, it provides the desire d_q to the job scheduler as its ***parallelism feedback***. The job scheduler assesses the various jobs running in the system and based on some administrative policy, determines the ***processor availability*** p_q, or the number of processors the job is entitled to receive for the quantum q. 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 d_q and the processor availability p_q. 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 this paper, we study thread schedulers that employ ***work stealing*** [1,4,8,10,13,14,17,32,33,37,38,42,47,52] to schedule the job's threads onto the allotted processors. Each processor maintains a ***work queue*** containing threads that are ready to execute. Whenever a processor runs out of work, it becomes a ***thief*** and attempts to ***steal*** a thread from another processor. If this ***victim*** processor has no available work in its work queue, the steal is ***unsuccessful***, and the ***thief*** processor makes new attempts to steal elsewhere until it is ***successful*** and finds work. We say that the processor cycles spent stealing are ***wasted***, because they do not contribute directly to the forward progress of completing the job's work.

Work-stealing schedulers can be analyzed in terms of two key measures of a job. The ***work*** T_1 of the job corresponds to the total number of unit-time instructions that the job executes, or equivalently its running time on 1 processor. The ***critical-path length*** T_∞ corresponds to the length of the longest chain of dependencies, or equivalently its running time on an infinite number of processors, assuming no overheads. Blumofe and Leiserson [10] showed that on a fixed number P of processors, a randomized work-stealing scheduler completes a job in $O(T_1/P+T_\infty)$ expected time. They also show that the number of processor cycles wasted is $O(PT_\infty)$.

Arora, Blumofe, and Plaxton extended this basic work-stealing algorithm to an adaptive setting, but notably ***without*** parallelism feedback. ABP always maintains P work queues, where P is the total number of processors in the machine. When the job scheduler allots $a_q = p_q$ processors in quantum q, ABP selects a_q queues uniformly at random from the P queues, and the allotted processors work on them. Arora *et al.* prove that the ***ABP*** algorithm completes a job in expected time

$$T = O(T_1/\overline{P} + PT_\infty/\overline{P}) , \quad (1)$$

where $\overline{P}$ is the mean processor availability as determined 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.

In previous work [3], we introduced a provably good adaptive thread scheduler, called A-STEAL, which is the focus of this paper. A-STEAL employs a provably good desire-estimation heuristic, which we shall review in Section 2, to provide parallelism feedback. A-STEAL adapts to changes in processor allotment by making two simple modifications to the basic work-stealing algorithm:

Allotment gain: When the allotment increases from quantum $q-1$ to q, the job obtains $a_q - a_{q-1}$ additional processors. The newly added processors immediately start stealing to obtain work from the other processors.

Allotment loss: When the allotment decreases from quantum $q-1$ to q, the job loses $a_{q-1} - a_q$ processors. To reallocate the work of these processors, A-STEAL uses the concept of "mugging" [11]. When a processor runs out of work, instead of stealing immediately, it looks for a ***muggable*** work queue: one that has no associated processor working on it and ***mugs*** the queue by taking over the entire work queue as its own. If there are no muggable work queues, the thief reverts to stealing normally. Data structures can be set up between quanta so that a successful steal or mug can be accomplished in $O(1)$ time [53].

At all time steps during the execution of an A-STEAL-scheduled job, every processor is either working, stealing, or mugging, and cycles spent stealing and mugging are ***wasted***.

The theoretical analysis in [3] demonstrates that A-STEAL schedules threads effectively — bounding completion time and wasted processor cycles — even in the face of an adversarial job scheduler. This analysis employs a technique called ***trim analysis*** [2], where we deduct a small number "bad" quanta and guarantee good performance on the remaining ones. Specifically, the ***R-high-trimmed mean availability*** (or ***R-trimmed availability***, for short) is defined to be the value obtained by removing the R time steps with the highest availability and taking the arithmetic average of the remaining. In [3], we show that for any given job with work T_1 and critical path T_∞ scheduled by A-STEAL, the job completes in $O(T_1/\widetilde{P} + T_\infty + L\lg P)$ expected time steps, where $\widetilde{P}$ denotes the $O(T_\infty + L\lg P)$-trimmed availability. Moreover, the total waste is at most $O(T_1)$, a constant fraction of the total work.

Despite the guarantees of these theoretical bounds, A-STEAL might nevertheless operate poorly in practice for three potential reasons:

- The constants hidden by the asymptotic notation might be too large to be practical.
- The assumption of an adversarial job scheduler might be too stringent, leading a simpler algorithm such as ABP to perform as well or better in realistic settings.
- The true mean availability $\overline{P}$ might diverge considerably in practice from the trimmed availability $\widetilde{P}$, leading A-STEAL to perform poorly on average.

In this paper, we describe simulation studies which show that A-STEAL is indeed an effective thread-scheduling algorithm, and that the empirical evidence refutes these three potential reasons for poor performance. We built a discrete-time simulator using DESMO-J [23] to evaluate the performance of A-STEAL. Some of our experiments evaluated the constants hidden in the asymptotic notations, while others benchmarked A-STEAL against ABP [4]. 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 constants hidden by the asymptotic notation are small. 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 [9, 16, 35].

The ***waste experiments*** were designed to measure the waste incurred by 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*** compared the completion time and waste of A-STEAL with ABP [4] 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*** compared 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.

The remainder of this paper is organized as follows. Section 2 describes the A-STEAL algorithm and its theoretical bounds. Section 3 describes our simulation setup. Sections 4, 5, 6, and 7 describe the four sets of experiments in detail. Section 8 describes related work, and Section 9 offers concluding remarks.

2. Overview of A-STEAL

This section briefly reviews the A-STEAL algorithm originally presented in [3], including its desire-estimation algorithm and its theoretical guarantees.

A-STEAL's simple desire-estimation algorithm is inspired by that in [2]. During each quantum q, A-STEAL monitors how each processor allotted to the job spends its cycles: working, stealing, or mugging. At the end of the quantum, it determines the job's ***nonsteal usage*** n_q for the job: the total number of cycles the jobs spends working or mugging. To estimate the desire for the next quantum $q+1$, A-STEAL uses the nonsteal usage n_q, as well as the desire d_q and allotment a_q from the previous quantum.

In addition, A-STEAL employs two tuning parameters. The ***utilization parameter*** $\delta \leq 1$ determines whether a quantum q is deemed ***efficient***, namely, whether $n_q \geq \delta L a_q$. That is, the nonsteal usage is at least a δ fraction of the La_q total processor cycles allotted for the quantum. A quantum is deemed ***inefficient*** otherwise. Typical values for δ might lie in the range 80–95%. The ***responsiveness parameter*** $\rho > 1$ determines how quickly the scheduler responds to changes in parallelism. Typical values for ρ might lie in the range 1.2–2.0.

Figure 1 shows the pseudocode for A-STEAL. The theoretical paper [3] contains more details about the algorithm.

```
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}         ▷ efficient
 6     then d_q ← ρd_{q−1}
 7  else d_q ← d_{q−1}
 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 a processor desire. Just before quantum q begins, 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.

The trim analysis presented in [3] provides the following bounds on the performance of A-STEAL with respect to time and waste. Suppose that a job with work T_1 and critical-path length T_∞ is scheduled on a machine with P processors, and let $\widetilde{P}$ be the $O(T_\infty + L\lg P)$-trimmed availability. Then, A-STEAL completes the job in time T while wasting at most W processor cycles, where

$$\mathrm{E}[T] \leq \frac{T_1}{\delta\widetilde{P}}(1+O(1/L)) + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P\right), \quad (2)$$

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

3. Simulation setup

To study A-STEAL, we built a Java-based discrete-time simulator using DESMO-J [23]. Our simulator implements four major entities — processors, jobs, thread schedulers, and job schedulers — and simulates their interactions in a two-level scheduling environment. Like prior work on scheduling of multithreaded jobs [5–7, 9, 10, 28, 41, 49], we modeled 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. 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

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

during each time step. We ignored the overheads due to the reallocation of processors.

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.

We implemented three kinds of job schedulers — profile-based, equipartitioning [46], and dynamic-equipartitioning [46]. A profile-based job scheduler was used in the first three 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 during job's execution. 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 [29] 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 varied 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 [57] 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 $L = 200$.

4. Time Experiments

The running-time bounds proved in [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 [8]. Assuming that equality holds and disregarding smaller terms, the model estimates performance as

$$T \approx c_1 T_1/\widetilde{P} + c_\infty T_\infty \ , \qquad (4)$$

where $c_1 > 0$ is the ***work overhead*** and $c_\infty > 0$ is

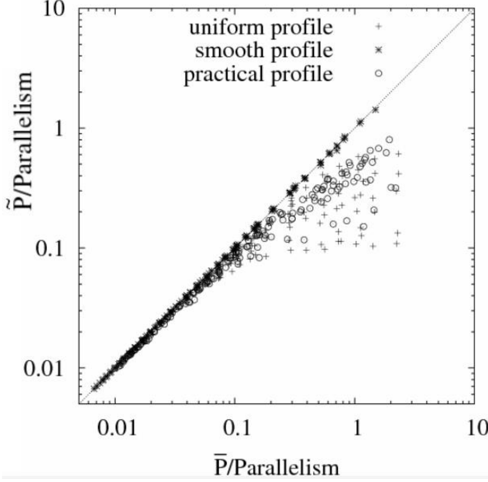


Figure 3: Comparing the (true) mean availability $\overline{P}$ with the trimmed availability $\widetilde{P}$ using three availability profiles. Each data point represents a job execution for which the mean availability and trimmed availability were measured. These values were normalized by dividing 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 $\rho = 1.5$.

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 $T_1/T_\infty \gg 60\widetilde{P}$. In practice, however, we should not expect these large overheads to materialize.

Our first linear-regression analysis fits the running time of the 2331 job runs to Equation (4). 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 work overhead c_1 is less than 1, because the jobs performed work during some of the steps that were trimmed.

We performed a second set of regressions 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

$$T = \hat{c}_1 T_1/\overline{P} + \hat{c}_\infty T_\infty \ . \tag{5}$$

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 4 and 5 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 (4) and (5) 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.

5. Waste Experiments

The theoretical analysis in [3] shows that the waste incurred 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.

Inequality (3) gives the total waste by A-STEAL, but it can be shown that 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 (3) 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 (3) 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.

6. 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 work queues. In particular, ABP has no mechanism for increasing and decreasing the number r of work queues, and it maintains $r = P$ queues 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 queues according to its allotment.

This paradox can also be understood by using the model from Equation (4) for A-STEAL and an analogous model based on Equation (1) for ABP. 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.

7. 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 among various jobs: dynamic equipartitioning [46] for A-STEAL and equipartitioning [59] 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 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 [19, 24, 30, 45, 55], none appears to apply directly to multithreaded jobs. Some studies [39,40,44] 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 [61] heavy-tailed distribution where the probabil-

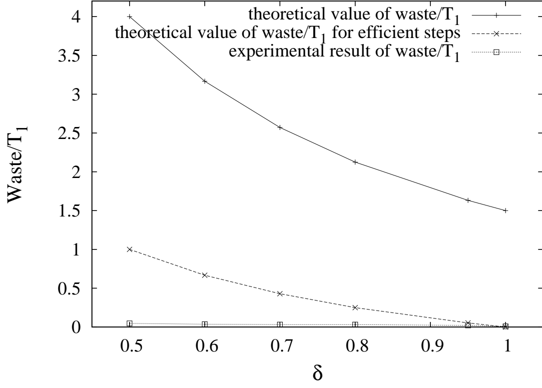


Figure 4: Comparing the theoretical and practical waste (normalized by T_1) using A-STEAL for various values of the utilization parameter δ. The top line shows the total theoretical waste, the next line shows the theoretical waste on efficient quanta, and the bottom line shows the observed waste. The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

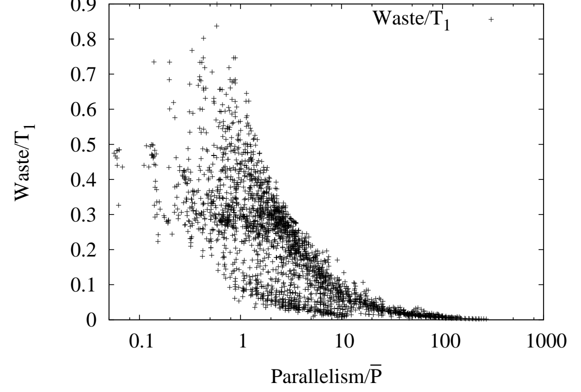
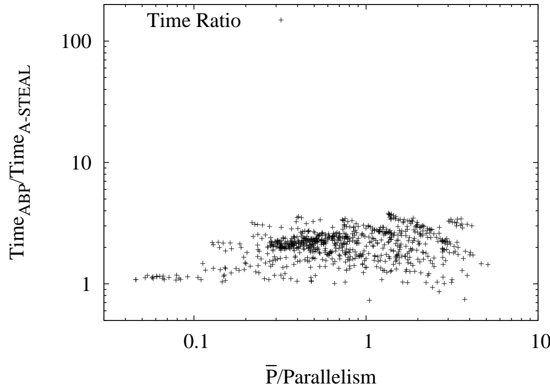


Figure 5: How waste varies with parallelism. When $T_1/T_\infty > 10\overline{P}$, that is, the job's parallelism significantly exceeds the average availability, the observed waste was only a tiny fraction of the work T_1. For jobs with small parallelism, the waste showed a large variance but never exceeded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



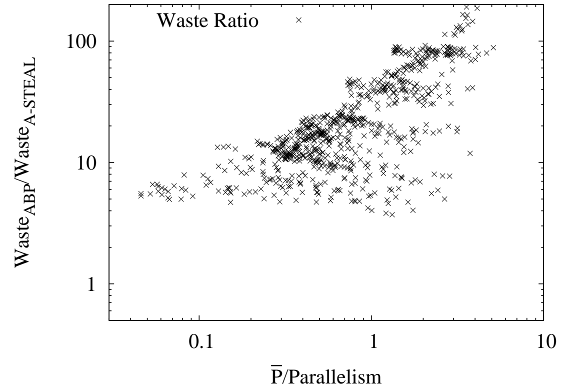


Figure 6: Comparing the time and waste of A-STEAL against ABP when $P = 512$ and $\overline{P} = 30, 60$. In this experiment, where P exceeds $\overline{P}$ by a significant margin, A-STEAL completes jobs about twice as fast as ABP while wasting less than 10% of the processor cycles wasted by ABP.

ity of generating x is proportional to $1/x$. In 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+DEQ for both these distributions.

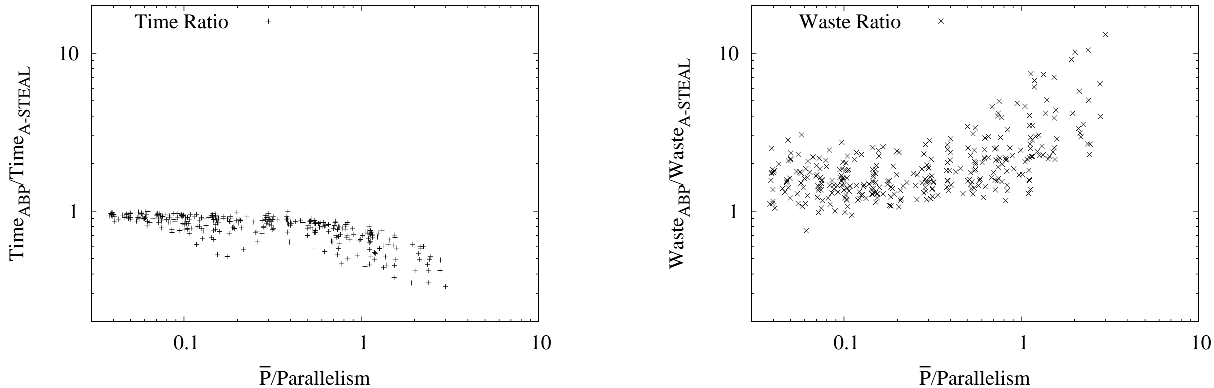


Figure 7: Comparing the time and waste of A-STEAL against ABP when $P = 128$ and $\overline{P} = 30, 60$. In this experiment, where P and $\overline{P}$ are closer in magnitude, A-STEAL runs slightly slower than ABP, but it still tends to waste fewer processor cycles than ABP.

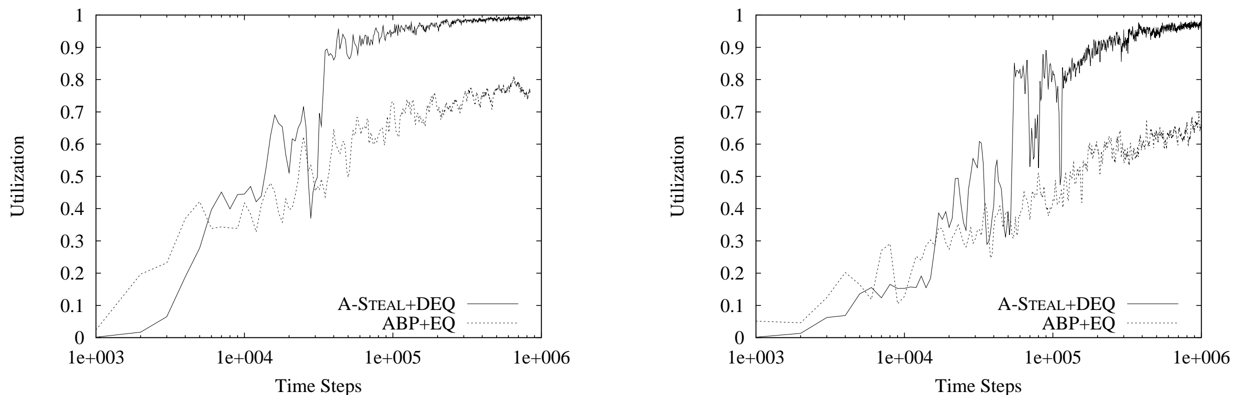


Figure 8: Comparing the utilization over time of A-STEAL+DEQ and ABP+EQ. In the left figure, both the critical-path length and the parallelism follow the uniform distribution, and in the right figure, they follow the HT1 distribution.

8. Related work

This section discusses related work on adaptive and non-adaptive scheduling of multithreaded jobs.

Work-stealing has been used as a heuristic since Burton and Sleep's research [17] on scheduling functional programs and Halstead's implementation of Multilisp [38]. Many variants have since been implemented [32, 37, 47], and work-stealing algorithms have been analyzed in the context of load balancing [52] and backtrack search [42]. Blumofe and Leiserson [10] proved that work stealing is efficient with respect to time, space, and communication for scheduling multithreaded computations on a fixed number of processor, and Arora, Blumofe, and Plaxton [4] extended the result to varying numbers of processors. Acar, Blelloch, and Blumofe [1] showed that work-stealing schedulers are efficient with respect to cache misses for jobs with "nested parallelism." Work-stealing algorithms have been implemented in many systems [8, 14, 33], and empirical studies show that work-stealing schedulers are scalable and practical [13, 33].

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading, primarily by Blumofe and his coauthors [4, 12, 13, 15]. 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 work in [12, 15] 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. Arora, Blumofe, and Plaxton [4] show that the ABP thread scheduler is provably efficient, and they give a nonblocking implementation of their algorithm. Blumofe and Papadopoulos [13] 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 parallelism of jobs is much greater than 8.

Adaptive task scheduling with parallelism feedback has been studied empirically in [53, 56, 58]. 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 feed-

back than without, but it is not apparent which of their strategies is best. Our earlier work [2, 3] appears to be the only theoretical analysis of a thread scheduler with parallelism feedback.

Adaptive job schedulers have been studied extensively, both empirically [25, 34, 46, 50, 51, 54, 60] and theoretically [22, 26, 27, 36, 48]. McCann, Vaswani, and Zahorjan [46] 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.

9. Conclusions

This section offers some conclusions and directions for future work.

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 queues. 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.

The empirical studies presented in this paper use a simulated multiprocessor. We ignore all scheduling overheads due to the communication between the job scheduler and the thread scheduler and due to the reallocation of processors. We plan to implement A-STEAL in the Cilk [8, 33] or JCilk [20, 21, 43] programming environments to evaluate parallelism feedback in the context of real multiprocessors.

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

[2] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson. Adaptive task scheduling with parallelism feedback. In *PPoPP*, 2006.

[3] K. Agrawal, Y. He, and C. E. Leiserson. A theoretical analysis of work stealing with parallelism feedback. 2006.

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

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

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

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

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

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

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

[11] R. D. Blumofe, C. E. Leiserson, and B. Song. Automatic processor allocation for work-stealing jobs. 1998.

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

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

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

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

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

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

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

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

[20] J. S. Danaher. The JCilk-1 runtime system. Master's thesis, MIT Department of Electrical Engineering and Computer Science, June 2005.

[21] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. The JCilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, Oct. 2005.

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

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

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

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

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

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

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

[29] D. Feitelson. Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[30] D. G. Feitelson. Packing schemes for gang scheduling. In D. G. Feitelson and L. Rudolph, editors, *JSSPP*, volume 1162, pages 89–110. Springer, 1996.

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

[32] R. Finkel and U. Manber. DIB—A distributed implementation of backtracking. *TOPLAS*, 9(2):235–256, Apr. 1987.

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

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

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

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

[37] M. Halbherr, Y. Zhou, and C. 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, Sept. 1994.

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

[39] M. 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.

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

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

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

[43] I.-T. A. Lee. The JCilk multithreaded language. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Aug. 2005.

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

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

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

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

[48] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. In *SODA*, pages 422–431, 1993.

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

[50] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing speedup through self-tuning of processor allocation. In *IPPS*, pages 463–468, 1996.

[51] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In D. G. Feitelson and L. Rudolph, editors, *JSSPP*, pages 155–174. Springer-Verlag, 1996.

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

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

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

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

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

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

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

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

[60] 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.

[61] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.