**Parallel and Sequential Algorithms** **Fall 2014**

# Work Stealing Analysis

*Student Presentation 1- Rob Utterback* *September 5, 2014*

## Scheduling Multithreaded Computations

We now know how to model multithreaded computations using directed acyclic graphs (DAGs). But now we want to find good ways to *schedule* these onto processing units.

Many current parallel programming systems/languages rely on the programmer to manually schedule tasks. Instead, we'll look at a dynamic scheduler for multithreaded computations that not only frees the programmer of scheduling duties but is also provably efficient.

## Getting Greedy

We previously learned about greedy schedulers, and proved that they execute a DAG with work $T_1$ and span $T_\infty$ in time

$$\frac{T_1}{P} + T_\infty,$$

which is optimal within a factor of 2.

Thus we want a scheduler that is greedy (or close to it). Doing so will make our scheduler asymptotically optimal!

## Work Stealing

Guided by this intuition, let's learn about a distributed scheduling algorithm known as *work stealing*.

### Structures

Each processing unit (which we'll sometimes call a worker) keeps a structure called a deque (double-ended queue), illustrated in Figure 1.

### Scheduling Rules

In addition to maintaining the deque, each worker follows a set of scheduling rules, as follows.

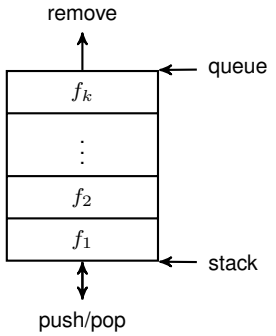



Figure 1: One worker's deque. The worker pushes and pops from the bottom of their deque, while other workers may remove from the top.

1. Node `x` spawns node `y`
   $\Longrightarrow$ Push `x` on bottom of deque, start working on `y`.

2. Node `x` returns.

   **(a)** If deque non-empty $\Longrightarrow$ Pop from the bottom and work.

   **(b)** If deque empty $\Longrightarrow$ attempt to get and execute `x`'s parent.

   **(c)** If deque empty and parent busy $\Longrightarrow$ perform work stealing.

3. `x` joins (syncs)
   $\Longrightarrow$ If possible, proceed. If stalled (children exist), then perform work stealing.
   Note that if the node is stalled, the deque *must* be empty. Do you see why?

### Note

An implementation of a work stealing scheduler (such as Cilk), will actually push, pop, and steal functions (and their continuations) on each worker's deque. In the analysis, however, we'll consider each unit-time "instruction" as its own node.

## Analysis of Running Time

**Theorem.** *Given a multithreaded computation with work $T_1$ and span $T_\infty$, work stealing executes this computation in expected time*

$$O\left(\frac{T_1}{P} + T_\infty\right).$$

## High Level Idea

At any time step, there are only two possible things a worker can be doing: working on a node, or attempting to steal work. So our approach will be to count the number of work steps and the number of steal attempts separately. Then, since a worker takes one of these steps each time step, we divide by $P$ to get the bound.

Clearly there must be exactly $T_1$ work steps, so the question is how many steal steps (attempts) are there? To match the bound above we need to bound the expected number of steal attempts by $O(T_\infty P)$.

## Phases

To count the steal attempts we will actually count the number of *phases*. We define a phase as a time period in which $\theta(P)$ steal attempts occur. Thus, if we can prove that the expected number of phases is $O(T_\infty)$, then we will have shown that the expected number of steal attempts is $O(T_\infty P)$, proving the bound.

## Potential Function

Now, to bound the number of phases we will define a potential function. This potential function will be monotonically decreasing (it will never increase). Given an initial amount of potential, we will bound the expected number of rounds until the potential reaches 0, which corresponds to the end of the computation.

First, we define the weight $w(\mathtt{u})$ of a node $\mathtt{u}$ as $T_\infty - d(\mathtt{u})$, where $d(\mathtt{u})$ is the depth of node $\mathtt{u}$ in the DAG. That is, $d(\mathtt{u})$ is the length of the longest path from the root node to $\mathtt{u}$, or

$$\max_{\mathtt{p} \in \text{parents}(\mathtt{u})} \{d(\mathtt{p})\} + 1$$

We must also have the notion of a "ready" node, which just means that all its dependencies have been met. We let $R_i$ denote all ready nodes at the beginning of phase $i$. Now, for every $\mathtt{u} \in R_i$, the potential of $\mathtt{u}$ is

$$\phi_i(u) = \begin{cases} 3^{2w(\mathtt{u})-1} & \text{if } \mathtt{u} \text{ is assigned} \\ 3^{2w(\mathtt{u})} & \text{otherwise} \end{cases}$$

So the potential of a phase $i$ is

$$\Phi_i = \sum_{\mathtt{u} \in R_i} \phi_i(\mathtt{u})$$

Note that at the beginning of the computation we have $\Phi_0 = 3^{2T_\infty - 1}$, and we want to count the number of (expected) phases until the potential reaches 0.

## Proof of Bound

To prove the bound, we will use two key lemmas. These will initially be stated without proof to show the structure of the main proof, after which we will prove the lemmas.

**Lemma 1.** $\Phi$ *is monotonically decreasing.*

**Lemma 2.** $\mathbf{Pr}\left\{\Phi_i - \Phi_{i+1} \geq \frac{1}{4}\Phi_i\right\} > \frac{1}{4}$
*In other words, the potential decreases by a constant fraction each phase with constant probability.*

*Proof of Time Bound.* We call a phase *successful* if the potential drops by at least $\frac{1}{4}$. We know that the potential will never increase (Lemma 1), and since the potential at phase 0 is $3^{2T_\infty - 1}$, the maximum number of successful steals is the number of times we can divide $\Phi_0$ by $\frac{4}{3}$ until we reach 0. That is,

$$\begin{aligned}\log_{\frac{4}{3}} 3^{2T_\infty - 1} &= (2T_\infty - 1)\log_{4/3} 3 \\ &< 8T_\infty.\end{aligned}$$

By Lemma 2, a successful phase happens with probability $\frac{1}{4}$, so the expected number of total phases is thus $32T_\infty = O(T_\infty)$.

Now we can put everything together. We have $O(T_\infty)$ expected phases, each of which has $\theta(P)$ steal attempts. Hence there are $O(T_\infty P)$ total expected steal attempts. Recall that we must have exactly $T_1$ work steps, and $P$ processors executing either a work step or a steal attempt at every time step. Thus the expected time to execute is

$$O\left(\frac{T_1 + T_\infty P}{P}\right) = O\left(\frac{T_1}{P} + T_\infty\right).$$

□

## Proof of Lemmas

*Proof of Lemma 1.* Workers perform two actions that can change the potential.

*Case* 1. A worker removes a node from a deque (either by a steal of a pop). Then that node $\mathtt{u}$ becomes assigned, hence we have

$$\phi_i(\mathtt{u}) - \phi_{i+1}(\mathtt{u}) = 3^{2w(\mathtt{u})} - 3^{2w(\mathtt{u})-1} = \frac{2}{3}\phi_i(\mathtt{u})$$

and the rest of the nodes are unchanged.

*Case* 2. A worker executes a node. The potential of that node, u, will disappear, but in the worst case two children are enabled, which adds potential. Assume that child x is pushed and y becomes assigned. Then x and y both have weight $w(\mathtt{u}-1)$, so the decrease in potential is

$$\begin{aligned}\phi_i(\mathtt{u}) - \phi_{i+1}(\mathtt{x}) - \phi_{i+1}(\mathtt{y}) &= 3^{2w(\mathtt{u})-1} - 3^{2w(\mathtt{x})} - 3^{2w(\mathtt{y})-1} \\ &= 3^{2w(\mathtt{u})-1} - 3^{2(w(\mathtt{u})-1)} - 3^{2(w(\mathtt{u})-1)-1} \\ &= 3^{2w(\mathtt{u})-1}\left(1 - \frac{1}{3} - \frac{1}{9}\right) \\ &= \frac{5}{9}\phi_i(\mathtt{u})\end{aligned}$$

In either case the potential decreases. □

The proof of the other lemma is broken into two other pieces and a final step to combine them. Let $D_i$ denote the non-empty deques at the start of phase $i$, not including an assigned node, and $A_i$ denote the empty deques. So if a deque $q$ has one node assigned, but nothing on the deque that can be stolen, then $q \in A_i$.

**Claim 1** (Top-Heavy Deques)**.** *Consider a round $i$, a process $q \in D_i$, and $q$'s topmost node $\mathtt{u}$. Then*

$$\phi_i(u) \geq \frac{3}{4}\Phi_i(q)$$

*That is, the top-most node of any deque holds $3/4$ of the potential of that entire deque!*

*Proof of Claim 1.* Let u be the only node in $q$'s deque and y be the assigned node of $q$. Assume that both u and y have the same parent, x. Then u and y have the same weight, so

$$\begin{aligned}\Phi_i(q) &= \phi_i(\mathtt{u}) + \phi_i(\mathtt{y}) \\ &= 3^{2w(\mathtt{u})} + 3^{2w(\mathtt{y})-1} \\ &= 3^{2w(\mathtt{u})} + 3^{2w(\mathtt{u})-1} \\ &= \frac{4}{3}\phi_i(\mathtt{u}).\end{aligned}$$

So $\phi_i(\mathtt{u}) = \frac{3}{4}\Phi_i(q)$.

In any other case, every node has strictly higher weight than the node below it (this can be proved formally using induction with the scheduling rules). So in those cases the topmost node contributes even more than $3/4$ of the potential of $q$. □

**Claim 2** (Balls and Weighted Bins)**.** *Suppose $P$ balls (steal attempts) are thrown randomly (independently and uniformly) into $P$ bins (worker deques). Let each bin $i$ have weight $W_i$ (the potential of a deque). Let $W$ denote the sum of all the bin weights and $X_i$ denote a random variable:*

$$X_i = \begin{cases} W_i & \textit{if some ball lands in bin } i \\ 0 & \textit{otherwise} \end{cases}$$

*If $X$ is the sum of all $X_i$ and $0 < \beta < 1$, then*

$$\mathbf{Pr}\{X \geq \beta W\} > 1 - \frac{1}{((1-\beta)\boldsymbol{e})}$$

$X_i$ should be interpreted as some fraction of the potential of a deque that will be lost upon a steal.

*Proof of Claim 2.* For each bin $i$, $W_i - X_i$ is $W_i$ if a ball lands in the bin and 0 otherwise. Thus

$$\begin{aligned} \mathbf{E}\left[W_i - X_i\right] &= W_i \left(1 - \frac{1}{P}\right)^P \\ &\leq \frac{W_i}{\boldsymbol{e}} \end{aligned}$$

Using linearity of expectation, this implies $\mathbf{E}\left[W - X\right] \leq {}^{W}/_{\boldsymbol{e}}$. Applying Markov's inequality yields

$$\begin{aligned} &\mathbf{Pr}\left\{W - X > (1-\beta)W\right\} < \frac{\mathbf{E}[W - X]}{(1-\beta)W} \\ \implies &\mathbf{Pr}\left\{X < \beta W\right\} < \frac{1}{((1-\beta)\boldsymbol{e})} \\ \implies &\mathbf{Pr}\left\{X \geq \beta W\right\} > 1 - \frac{1}{((1-\beta)\boldsymbol{e})}. \end{aligned}$$

□

*Proof of Lemma 2.* Claim 2 is really saying that with a constant probability, the potential of all (non-empty) deques will decrease by a constant fraction whenever $P$ steal attempts happen (in each phase).

Consider a steal attempt from worker $q \in D_i$. Then $q$'s topmost node $\mathtt{u}$ is at least *assigned* to another worker. (Note that a failed steal attempt on a non-empty deque implies that some other worker's steal attempt succeeded.) This node $\mathtt{u}$ is at least ${}^{3}/_{4}$ of $q$'s potential, as assigning it decreases $\mathtt{u}$'s potential by ${}^{2}/_{3}$ (from Case 1 of Lemma 1).

Thus the potential of $q$ drops by

$$\frac{2}{3}\phi_i(u) \geq \frac{2}{3}\frac{3}{4}\Phi_i(q) = \frac{1}{2}\Phi_i(q)$$

Now consider each deque $q \in D_i$ as a bin and assign it weight $W_q = \frac{1}{2}\Phi_i(q)$. For each $q \in A_i$, assigned $W_q = 0$. Applying the Ball and Bins claim with $\beta = \frac{1}{2}$,

$$\mathbf{Pr}\left\{\Phi_i - \Phi_{i+1} \geq \frac{1}{2}\frac{1}{2}\Phi_i(D_i)\right\} > 1 - \frac{2}{e} > \frac{1}{4}$$

Now we must consider the case where deque $q$ is empty at the beginning of phase $i$ ($q \in A_i$). In this case either $\Phi_i(q) = 0$ or $q$ has one node assigned to it. In the latter case $\Phi_i(q)$ drops by $5/9$ from Case 2 of Lemma 1 because that node is executed.

So for all workers that actually contribute potential to the total, their potential drops by $\frac{1}{4}$ with a constant probability. Thus

$$\mathbf{Pr}\left\{\Phi_i - \Phi_{i+1} \geq \frac{1}{4}\Phi_i\right\} > \frac{1}{4}$$

□

# Space Bounds

Running time is not our only concern, however. We don't want our scheduler to use much more space than the sequential computation, if possible. The following property provides a good space usage bound if a scheduler can achieve it. The design of work stealing was influenced by this property.

Instead of considering a DAG made up of unit-time nodes, we will now consider a DAG with *strands*, each of which is made up of one or more sequential instructions. It is these strands that are actually pushed, popped, and stolen. See the DAG figure from Wednesday's notes for an example. We consider a strand as *alive* if the instruction that spawns it has been executed.

**Property 1** (Busy Leaves)**.** *Define the spawn subtree as the tree of strands that are alive. The Busy Leaves property states that every leaf in the spawn subtree is being worked on. That is, every living strand with no descendants is being worked on.*

**Theorem.** *Any scheduler achieving the Busy Leaves property uses space $O(PS_1)$, where $S_1$ is the space used when the computation is run sequentially.*

*Proof.* Since the BL property says that all leaves of the spawn tree are being executed, we can have at most $P$ leaves. The space used by a leaf is the space of all activation frames from that strand up to the root. So in the worst case a leaf uses space $S_1$.

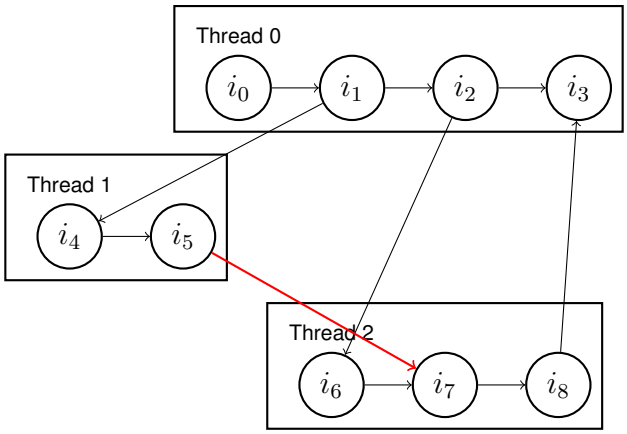



Figure 2: This DAG is not strict because of the red edge.

Thus the space used in the worst case is $P \cdot S_1$. □

We would like for work stealing to satisfy the busy leaves property. But it turns out that for general DAGs work stealing may not achieve this goal. Instead we will define a subclass of DAGs called "strict" and confine work stealing to executing strict computations.

**Definition 1** (Strict)**.** A strict computation (DAG) is one in which join (sync) edges go only to ancestors. The intuition is that we don't want to start a strand until its arguments are ready. Figure 2 shows a DAG that is not strict.

**Theorem.** *The work stealing scheduler uses space* $O(PS_1)$.

*Proof.* We need only prove that work stealing satisfies the busy leaves property because of Theorem . We proceed by induction on execution time, using each of the scheduling rules separately.

*Case* 1 (x spawns y.). x is no longer a leaf, and y is now a leaf. Work stealing forces the processor to push x and work on y, so the new leaf is being worked on.

*Case* 2 (x returns.).

**(a)** If the deque is non-empty, we pop from the bottom, so now a new leaf is being worked on.

**(b)** If the deque is empty and we get x's parent, then are again working on a leaf.

**(c)** If the deque is empty and the parent is busy, then x did not enable a new leaf.

*Case* 3 (x joins). If x enabled another strand, it will work on a new leaf (this overlaps with Case 2b). If x is now stalled, the unresolved dependency must have come from a descendant since we have a strict computation. Thus x's parent has another child, so the parent is not a leaf.

□