CSE 613: Parallel Programming

Lectures 6 – 8
(Analysis of a Work Stealing Scheduler)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2015

Work-Sharing and Work-Stealing Schedulers

Work-Sharing

- Whenever a processor generates new tasks it tries to distribute some of them to underutilized processors
- Easy to implement through centralized (global) task pool
- The centralized task pool creates scalability problems
- Distributed implementation is also possible (but see below)

Work-Stealing

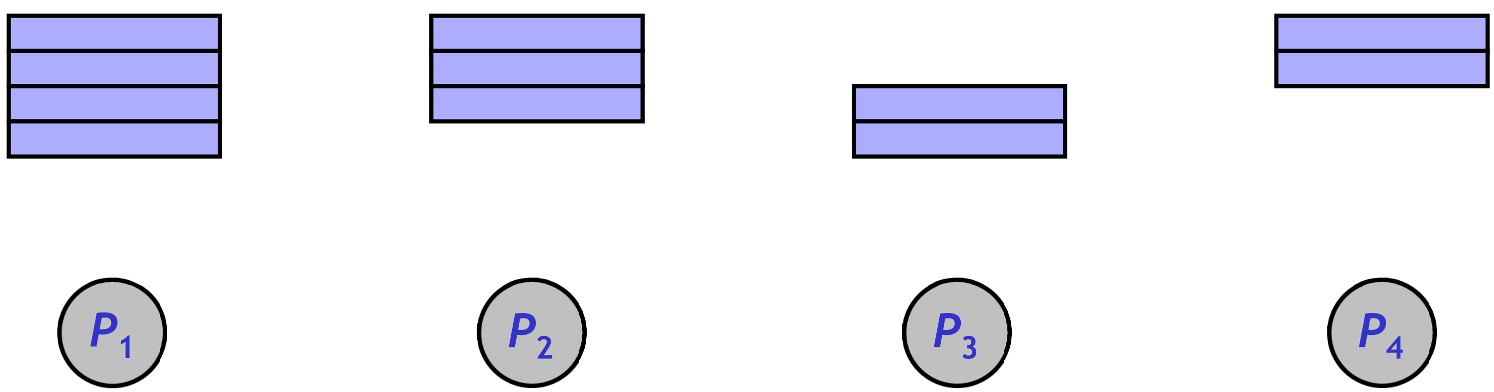
- Whenever a processor runs out of tasks it tries steal tasks from other processors
- Distributed implementation
- Scalable
- Fewer task migrations compared to work-sharing (why?)

Cilk++'s Work-Stealing Scheduler

- A *randomized distributed* scheduler
- Time bounds
  - Provably: $T_p = \frac{T_1}{p} + \mathrm{O}(T_\infty)$ (expected time)
  - Empirically: $T_p \approx \frac{T_1}{p} + T_\infty$
- Space bound: $\leq p$ × serial space bound
- Has provably good *cache performance*

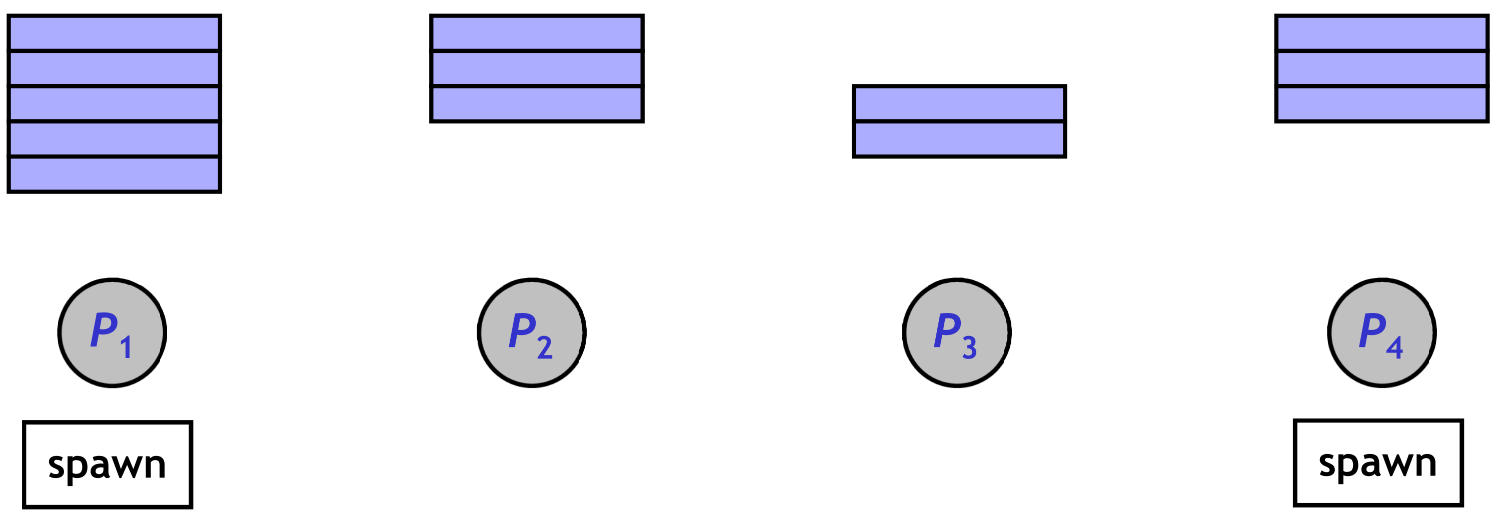
Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



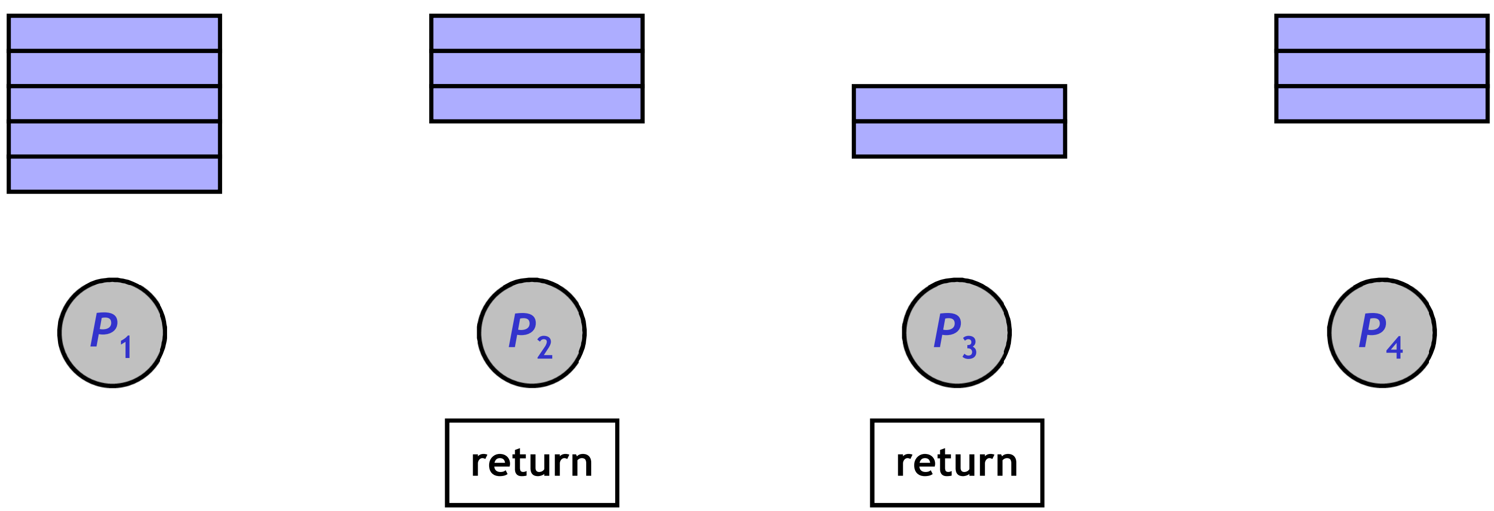
Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



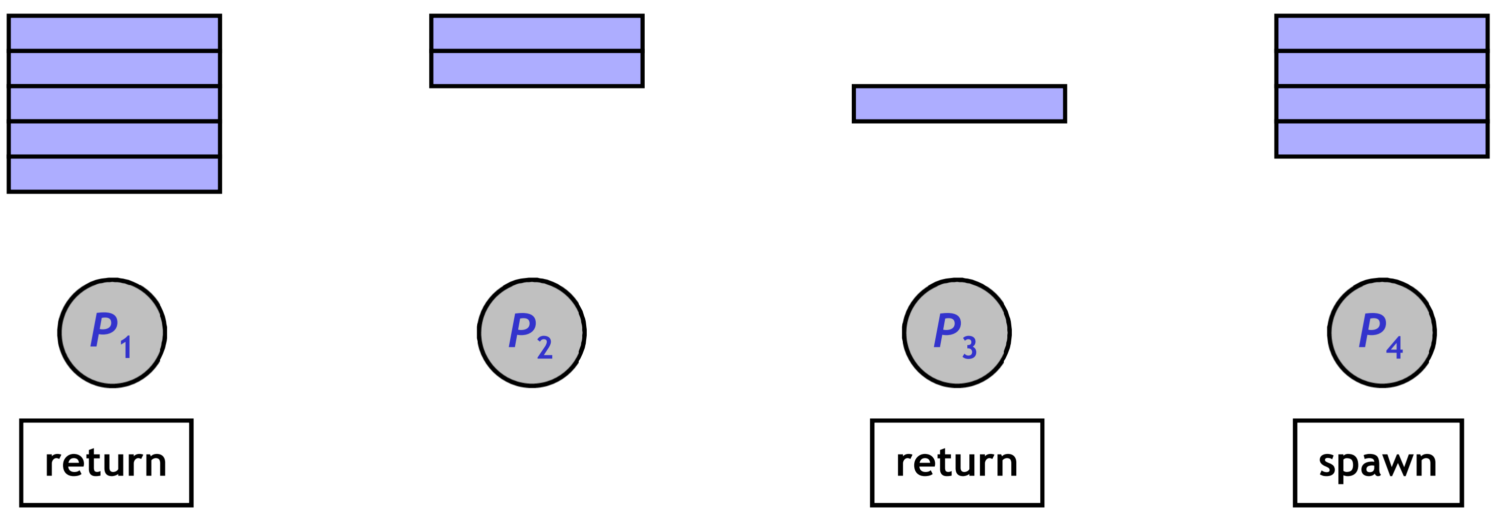
Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



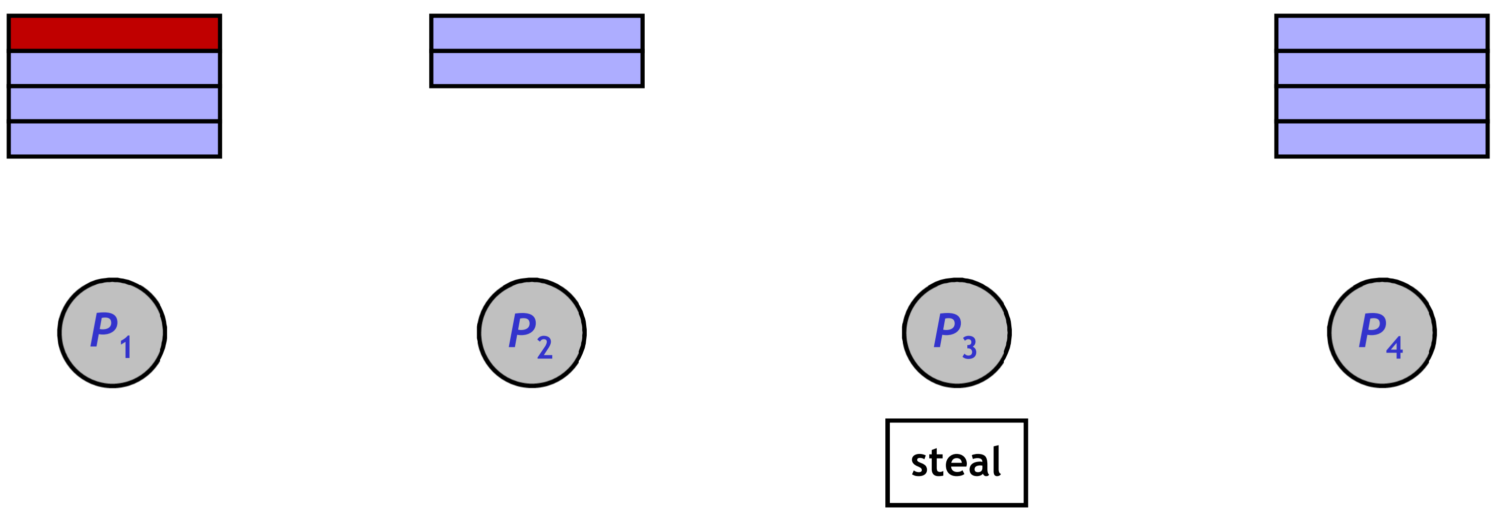
Cilk++'s Work-Stealing Scheduler

– Each core maintains a *work dqueue* of ready threads

– A core manipulates the bottom of its dqueue like a stack

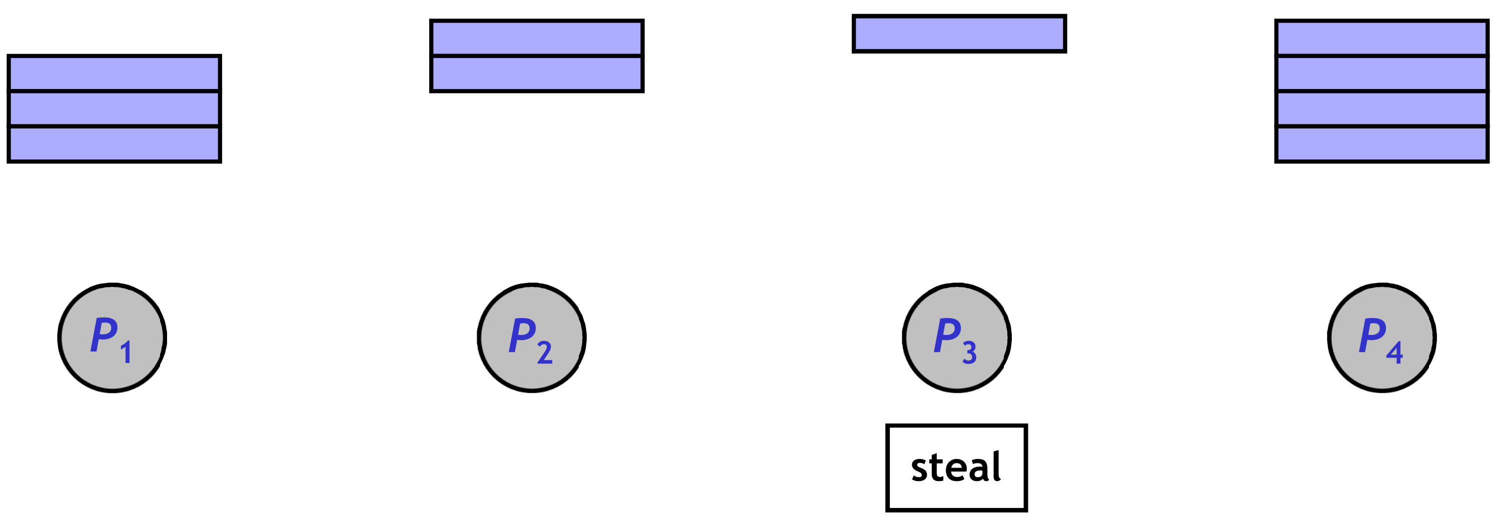
  - Pops ready threads for execution
  - Pushes new/spawned threads

– Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



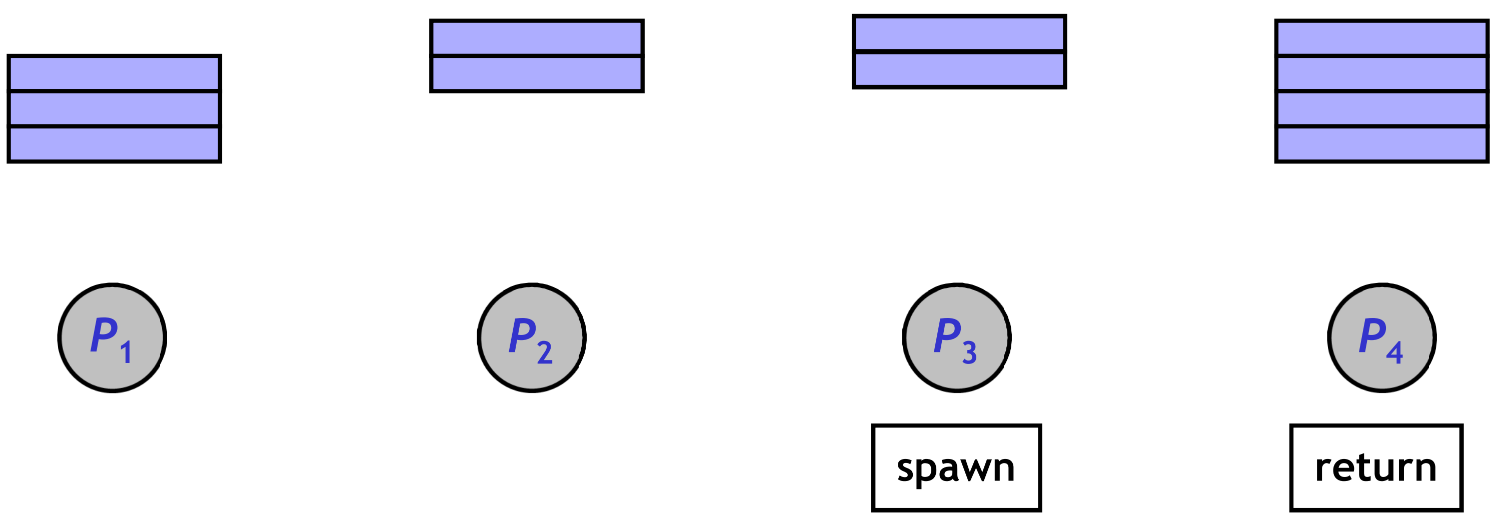
Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



Bound on the Number of Attempted Steals

Let T_p be the running time on p processors. Then T_1= total work.

Let S be the number of attempted steals.

Since each processor is either working or stealing, we have,

$$T_p = \mathrm{O}\left(\frac{T_1 + S}{p}\right)$$

We will show that $S = \mathrm{O}(pT_\infty)$. Then

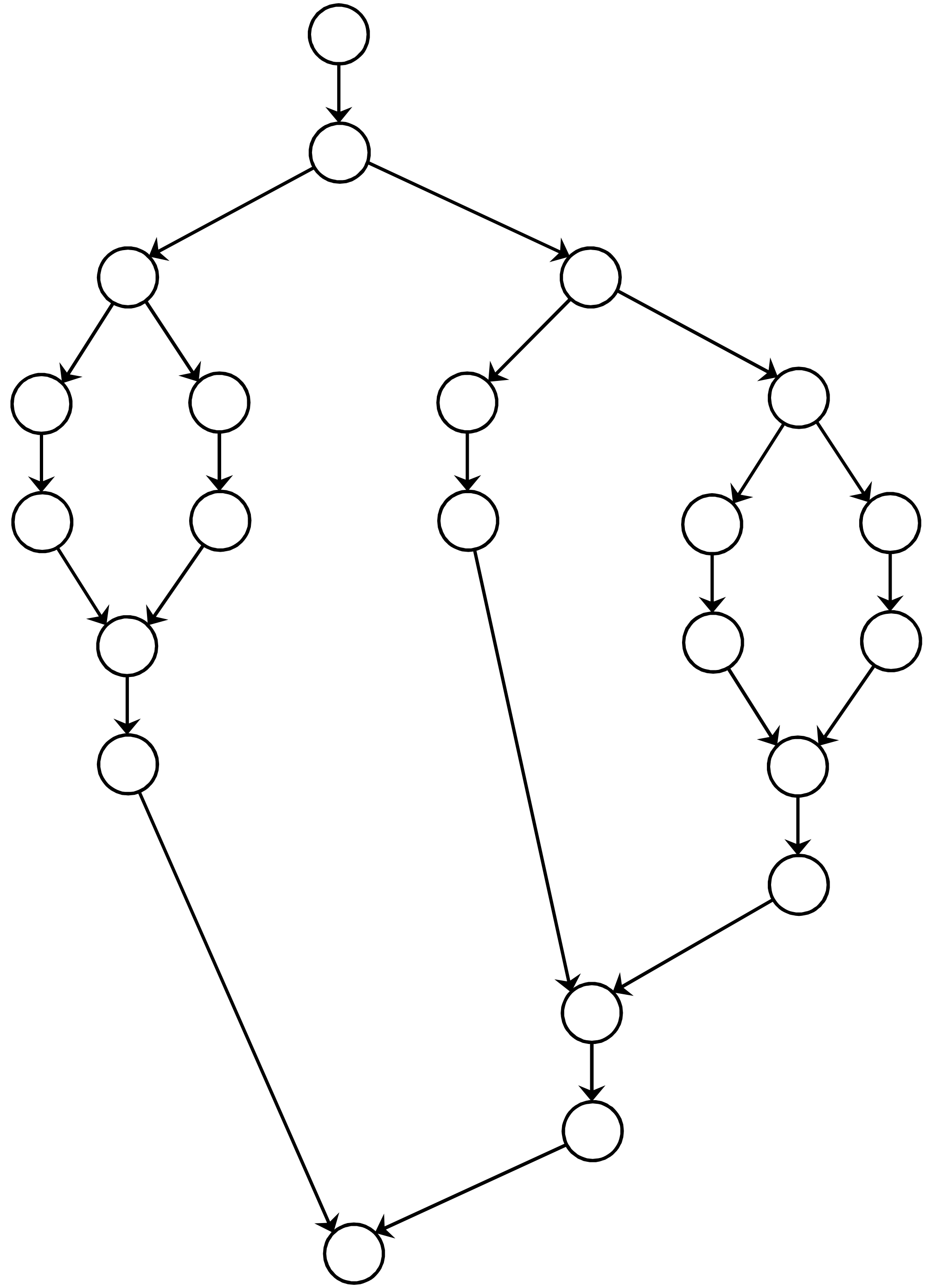
$$T_p = \mathrm{O}\left(\frac{T_1 + S}{p}\right) = \mathrm{O}\left(\frac{T_1}{p} + T_\infty\right)$$

Assumptions

DAG: We treat the multithreaded computation as a DAG, where each node corresponds to one instruction.

Deques: The deques contain ready nodes, that is, each ready thread in a deque is replaced with its currently ready node.

Assigned Node: The node a processor is currently executing.



Assumptions

Scheduler: Operates on nodes instead of threads as follows.

- if a processor does not have an assigned node,
  - **Deque nonempty:** pops the bottom-most node off its deque, and that node becomes the assigned node
  - **Deque empty:** pops the top-most node off the deque of a random victim, and that node becomes the assigned node
- if the execution of an assigned node enables
  - **Two child nodes:** one is pushed onto the bottom of the deque, and the other child becomes the assigned node
  - **One child node:** that child becomes the assigned node
  - **No child node:** the processor returns to the deque or becomes a thief to obtain an assigned node

Assumptions

Enabling Edge: If execution of node *u* enables node *v* we call edge (*u*, *v*) an *enabling edge.*

Designated Parent: If *u* enables *v,* we call *u* the *designated parent* of *v*.

Each node except the root has exactly one designated parent.

Enabling Tree: Subgraph of the DAG consisting of only enabling edges form a rooted tree that we call the *enabling tree*.

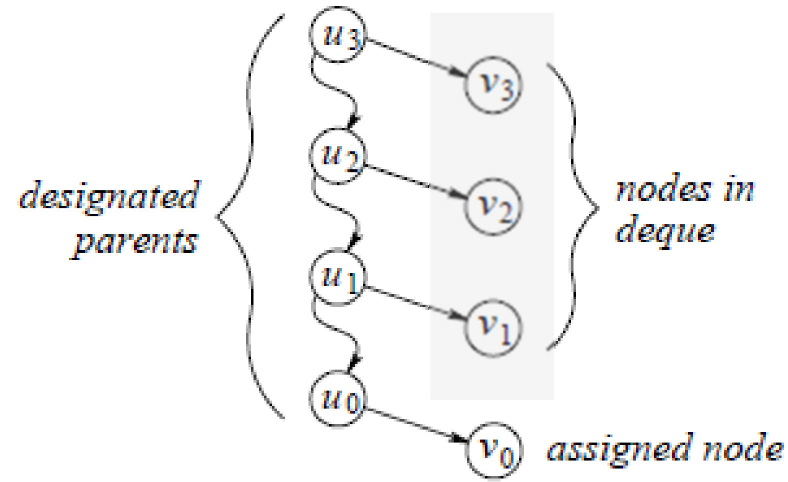
Each execution of the computation DAG may have a different enabling tree.

Node depth: The depth $d(u)$ of node *u* in the enabling tree is the number of edges on the path from the root of the tree to *u*

Node weight: The weight of node *u* is defined as $w(u) = T_\infty - d(u)$

A Structural Lemma

Lemma 1: For a given processor, if v_0 is the assigned node, $v_1, \ldots, v_k$ are the nodes in deque, u_i is the designated parent of v_i, then u_{i+1} is an ancestor of u_i, and for $i \geq 1$, $u_{i+1} \neq u_i$.



Proof: By induction on the number of assigned node executions.

The claim holds vacuously when execution begins.

Assume that the claim holds before an assigned node execution. Show that the claim continues to hold if the number of child node enabled by the execution is

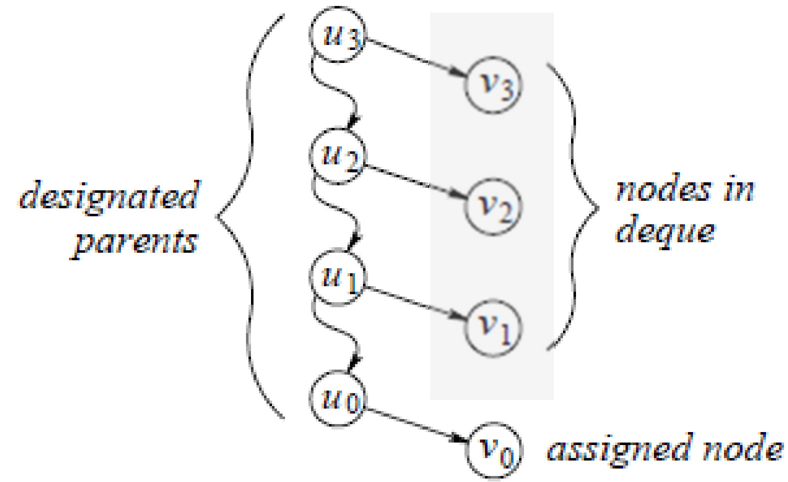
0: two cases: deque empty & deque nonempty

1: straight-forward

2: gives rise to the possibility that $u_0 = u_1$

A Structural Lemma

Corollary 1: For a given processor, if v_0 is the assigned node, and $v_1, \dots, v_k$ are the nodes in deque, then $w(v_0) \le w(v_1) < \dots < w(v_{k-1}) < w(v_k)$.



Proof: From Lemma 1 we know that if u_i is the designated parent of v_i, then u_{i+1} is an ancestor of u_i, and for $i \ge 1$, $u_{i+1} \ne u_i$.

Hence, $d(u_0) \ge d(u_1) > \dots > d(u_{k-1}) > d(u_k)$.

But $d(v_i) = d(u_i) + 1$. So,

$$d(v_0) \ge d(v_1) > \dots > d(v_{k-1}) > d(v_k).$$

Since $w(v_i) = T_\infty - d(v_i)$, we have,

$$w(v_0) \le w(v_1) < \dots < w(v_{k-1}) < w(v_k).$$

Potential Function

To simplify analysis we assume that each operation (i.e., execution of a node or a steal attempt) takes one time step to complete.

The potential of a ready node u at time step t is:

$$\phi_t(u) = \begin{cases} 3^{2w(u)-1}, & \text{if } u \text{ is being executed;} \\ 3^{2w(u)}, & \text{otherwise (} u \text{ is in deque).} \end{cases}$$

Let $R_t(q)$ be the set of ready nodes associated with processor q at time step t. Then potential of q at time t:

$$\phi_t(q) = \sum_{u \in R_t(q)} \phi_t(u)$$

Then the total potential at time step t:

$$\Phi_t = \sum_q \phi_t(q)$$

Potential Function

Initial Value: The only ready node is the root node at depth 0. Hence,

$$\Phi_0 = 3^{2T_\infty - 1}$$

Final Value: No ready nodes. Hence,

$$\Phi_{final} = 0$$

Intermediate Values: Throughout the entire execution the potential never increases, that is, for each time step t:

$$\Phi_{t+1} \leq \Phi_t$$

Potential Function

Lemma 2: For each time step t, $\Phi_{t+1} \leq \Phi_t$.

Proof: Only the following two actions may change the potential.

– Removal of any node u from deque to assign to a processor:

$$\begin{aligned}\text{Decrease in potential} &= \phi_t(u) - \phi_{t+1}(u) \\ &= 3^{2w(u)} - 3^{2w(u)-1} \\ &= \frac{2}{3}\phi_t(u) \\ &> 0\end{aligned}$$

– Execution of an assigned node u:

The execution may enable 0, 1 or 2 child nodes.

Potential Function

Lemma 2: For each time step t, $\Phi_{t+1} \leq \Phi_t$.

Proof: Only the following two actions may change the potential.

– Removal of any node u from deque to assign to a processor:

$$\text{Decrease in potential} = \phi_t(u) - \phi_{t+1}(u) = \frac{2}{3}\phi_t(u) > 0$$

– Execution of an assigned node u:

The execution may enable 0, 1 or 2 child nodes.

Suppose u enables x (dequed) and y (assigned). Then potential drop:

$$\begin{aligned}
\phi_t(u) - \phi_{t+1}(x) - \phi_{t+1}(y) &= 3^{2w(u)-1} - 3^{2w(x)} - 3^{2w(y)-1} \\
&= 3^{2w(u)-1} - 3^{2(w(u)-1)} - 3^{2(w(u)-1)-1} \\
&= 3^{2w(u)-1}\left(1 - \frac{1}{3} - \frac{1}{9}\right) = \frac{5}{9}\phi_t(u) > 0
\end{aligned}$$

For fewer than 2 child nodes the potential drops even more.

Top-Heavy Deques

Lemma 3: If the deque of processor q is nonempty, and v is the top node of the deque, then $\phi_t(v) \geq \frac{3}{4}\phi_t(q)$.

Proof: Let $v_k (= v), v_{k-1}, \ldots, v_1$ be the nodes in the deque from top to bottom, and let v_0 be the assigned node. Then

$$\phi_t(v_0) \leq \frac{1}{3}\phi_t(v_1) \text{ and } \phi_t\left(v_{j-1}\right) \leq \frac{1}{9}\phi_t\left(v_j\right) \text{ for } 2 \leq j \leq k.$$

Hence,

$$\phi_t(q) = \sum_{0 \leq j \leq k} \phi_t\left(v_j\right) \leq \phi_t(v_k)\left(1 + \frac{1}{9} + \frac{1}{9^2} + \cdots + \frac{1}{9^{k-1}} + \frac{1}{9^{k-1}} \cdot \frac{1}{3}\right)$$

$$\leq \frac{4}{3}\phi_t(v_k) = \frac{4}{3}\phi_t(v)$$

Successful Steals

Lemma 4: A successful steal by processor r from processor q at time step t decreases Φ_t by at least $\frac{1}{2}\phi_t(q)$.

Proof: Let v be the top node in q's deque. Then from Lemma 3:

$$\phi_t(v) \geq \frac{3}{4}\phi_t(q)$$

The potential of q decreases by $\phi_t(v)$.

The potential of r increases by $\frac{1}{3}\phi_t(v)$.

Hence, the total potential drop $= \phi_t(v) - \frac{1}{3}\phi_t(v) = \frac{2}{3}\phi_t(v) \geq \frac{1}{2}\phi_t(q)$

Attempted Steals

Corollary 2: An attempted steal from processor q at time step t decreases Φ_t by at least $\frac{1}{2}\phi_t(q)$.

Proof: If $\phi_t(q) = 0$, the claim is vacuously true. So, let $\phi_t(q) > 0$.

If the steal succeeds, then Φ_t drops by at least $\frac{1}{2}\phi_t(q)$. [Lemma 4]

If fails, then Φ_t drops by at least $\frac{5}{9}\phi_t(q) \geq \frac{1}{2}\phi_t(q)$. [Proof of Lemma 2]

Balls and Weighted Bins

Lemma 5: Suppose p balls are thrown independently and uniformly at random into p bins, where bin i has weight W_i, for $i = 1, \dots, p$. Define

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

If $W = \sum_{i=1}^{p} W_i$ and $X = \sum_{i=1}^{p} X_i$, then for any β s.t. $0 < \beta < 1$,

$$\Pr[X < \beta W] < \frac{1}{(1-\beta)e}.$$

Proof: $\Pr[X_i = 0] = \left(1 - \frac{1}{p}\right)^p < \frac{1}{e}$.

Then $E[X_i] = 0 \times \Pr[X_i = 0] + W_i \times \Pr[X_i \neq 0] > \left(1 - \frac{1}{e}\right) W_i$.

Hence, $E[X] > \left(1 - \frac{1}{e}\right) W \Rightarrow E[W - X] < \frac{W}{e}$.

Using Markov's inequality,

$$\Pr[X < \beta W] = \Pr[W - X > (1-\beta)W] < \frac{E[W - X]}{(1-\beta)W} < \frac{1}{(1-\beta)e}.$$

Potential Drops in Phases

Lemma 6: Consider time steps i and $j > i$ such that at least p steal attempts occur between time steps i (inclusive) and j (exclusive). Then

$$\Pr\left[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i\right] > \frac{1}{4}.$$

Proof: Each processor q is a bin, and each attempted steal is a throw of a ball. Let Q be the set of processors that were victims of attempted steals during this phase.

Let $X_q = \phi_i(q)$ for each $q \in Q$, and $X_q = 0$ otherwise.

Then setting $\beta = \frac{1}{2}$ in Lemma 5, we get,

$$\Pr\left[\sum_{q \in Q} \phi_i(q) < \frac{1}{2}\Phi_i\right] < \frac{2}{e} \Rightarrow \Pr\left[\sum_{q \in Q} \phi_i(q) \geq \frac{1}{2}\Phi_i\right] > 1 - \frac{2}{e} > \frac{1}{4}$$

But from Corollary 2: $\Phi_i - \Phi_j \geq \frac{1}{2}\sum_{q \in Q} \phi_i(q)$

Combining: $\Pr\left[\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i\right] > \frac{1}{4}$

Expected Number of Steal Attempts

Theorem 1: The expected number of steal attempts during the entire computation is $\mathrm{O}(pT_\infty)$.

Proof: We say that a phase is successful if the potential drops by a factor of at least $\frac{1}{4}$ during that phase.

Since $\Phi_0 = 3^{2T_\infty - 1}$, the computation terminates after at most $\log_{\frac{4}{3}}\Phi_0 = (2T_\infty - 1)\log_{\frac{4}{3}}3 < 8T_\infty$ successful phases.

Since a phase is successful with probability at least $\frac{1}{4}$, the expected number of phases required for $8T_\infty$ successes is at most $32T_\infty$.

Each phase consists of p steal attempts. Hence, the expected number of steals before termination is $\mathrm{O}(pT_\infty)$.

High Probability Bound on Steal Attempts

Theorem: The number of steal attempts is $O\left(p\left(T_\infty + \log\frac{1}{\epsilon}\right)\right)$ with probability at least $1-\epsilon$, for $0 < \epsilon < 1$.

Proof: Suppose the execution takes $n = 32T_\infty + m$ phases. Each phase succeeds with probability $\geq \frac{1}{4}$. Then $\mu = n \times \frac{1}{4} = 8T_\infty + \frac{m}{4}$.

Chernoff bound 3 for lower tail with $\gamma = \frac{m}{4}$ and $m = 32T_\infty + 16\ln\frac{1}{\epsilon}$.

$$Pr[X \leq 8T_\infty] < e^{-\frac{(m/4)^2}{16T_\infty + m/2}} \leq e^{-\frac{(m/4)^2}{m/2+m/2}} = e^{-\frac{m}{16}} \leq e^{-\frac{16\ln\frac{1}{\epsilon}}{16}} = \epsilon$$

Thus the probability that the execution takes $64T_\infty + 16\ln\frac{1}{\epsilon}$ phases or more is less than ϵ.

Hence, the number of steal attempts is $O\left(p\left(T_\infty + \log\frac{1}{\epsilon}\right)\right)$ with probability at least $1-\epsilon$.