CS 140 : Feb 19, 2015
Cilk Scheduling & Applications

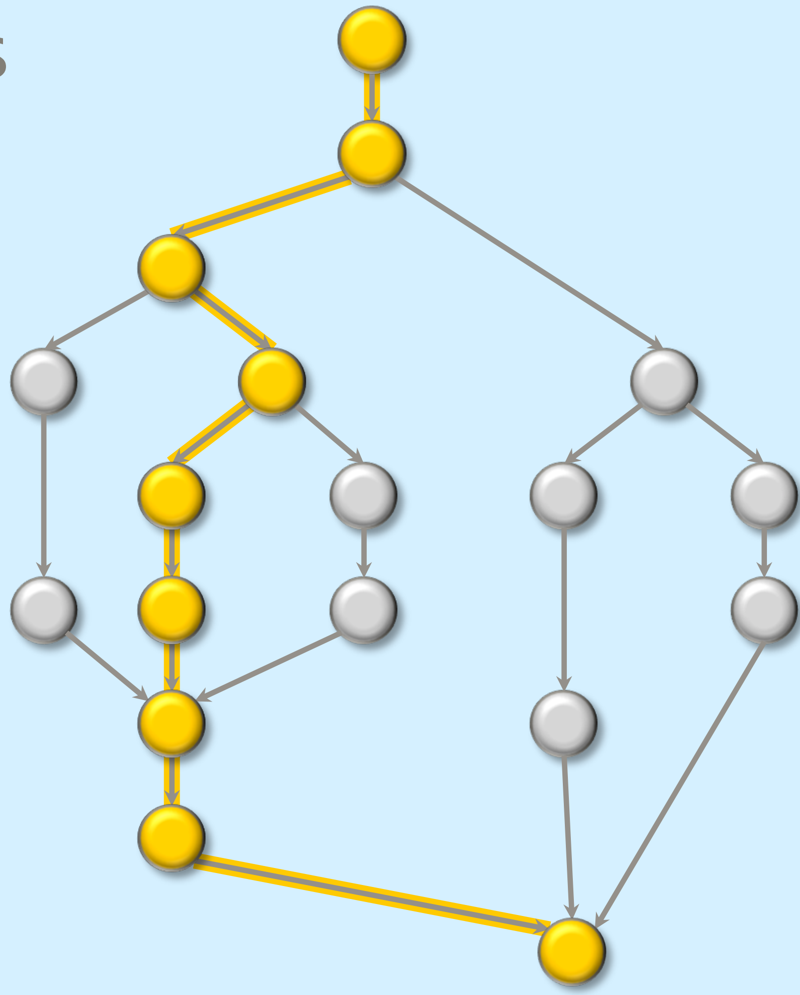
- Analyzing quicksort
- Optional: Master method for solving divide-and-conquer recurrences
- Tips on parallelism and overheads
- Greedy scheduling and parallel slackness
- Cilk runtime

Thanks to Charles E. Leiserson for some of these slides

Potential Parallelism

Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = ***potential parallelism***
= the average amount of work per step along the span.



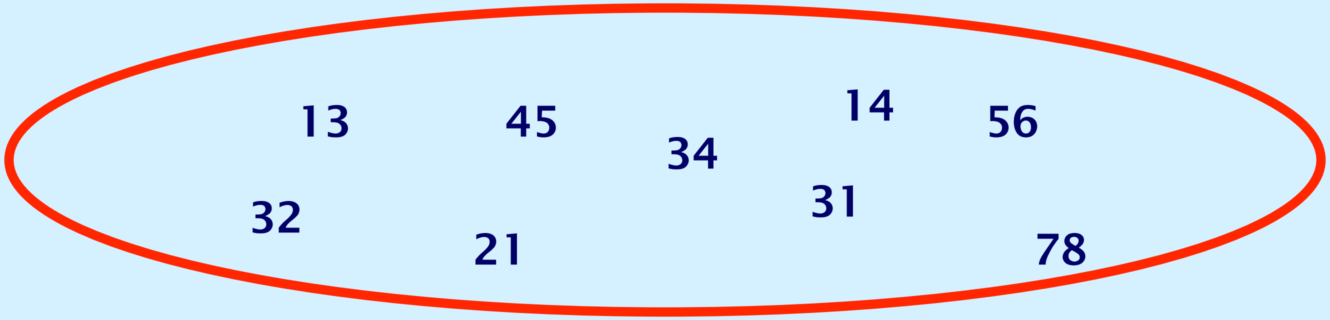
Sorting

- Sorting is possibly the most frequently executed operation in computing!
- **Quicksort** is the fastest sorting algorithm in practice with an average running time of O(N log N), (but $O(N^2)$ worst case performance)
- **Mergesort** has worst case performance of O(N log N) for sorting N elements
- Both based on the recursive **divide-and-conquer** paradigm

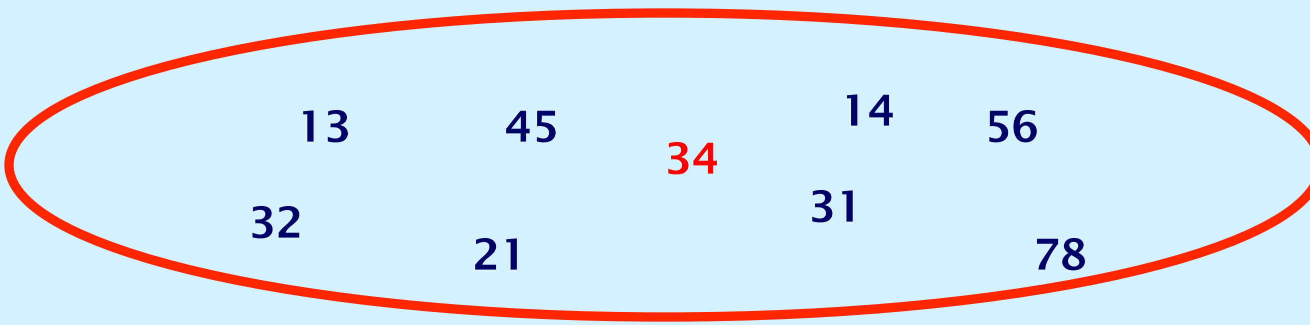
QUICKSORT

- Basic Quicksort sorting an array S works as follows:
  - If the number of elements in S is 0 or 1, then return.
  - Pick any element *v* in S. Call this pivot.
  - Partition the set S–{v} into two disjoint groups:
    - $S_1 = \{x \in S-\{v\} \mid x \leq v\}$
    - $S_2 = \{x \in S-\{v\} \mid x \geq v\}$
  - Return quicksort(S_1) followed by v followed by quicksort(S_2)

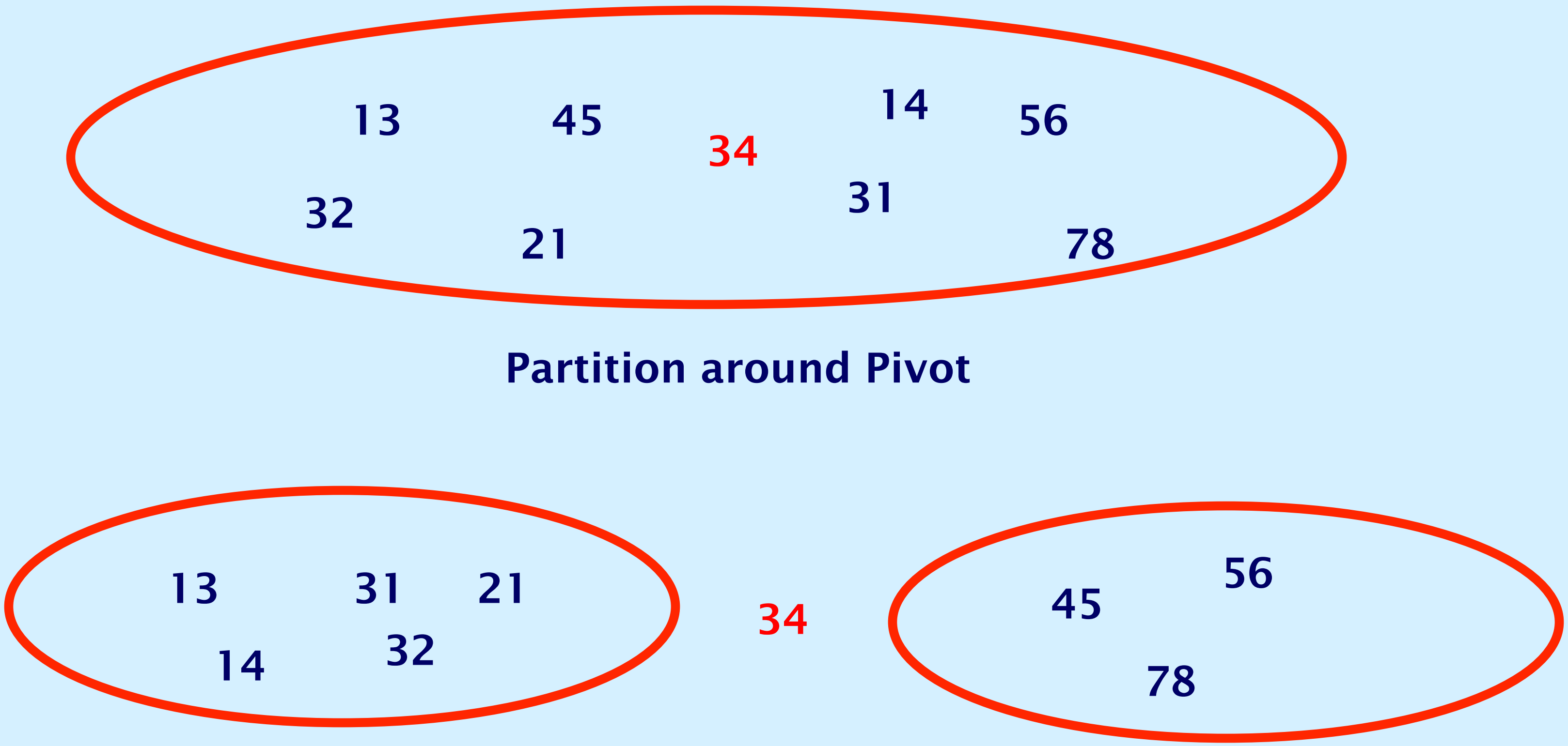
QUICKSORT



Select Pivot

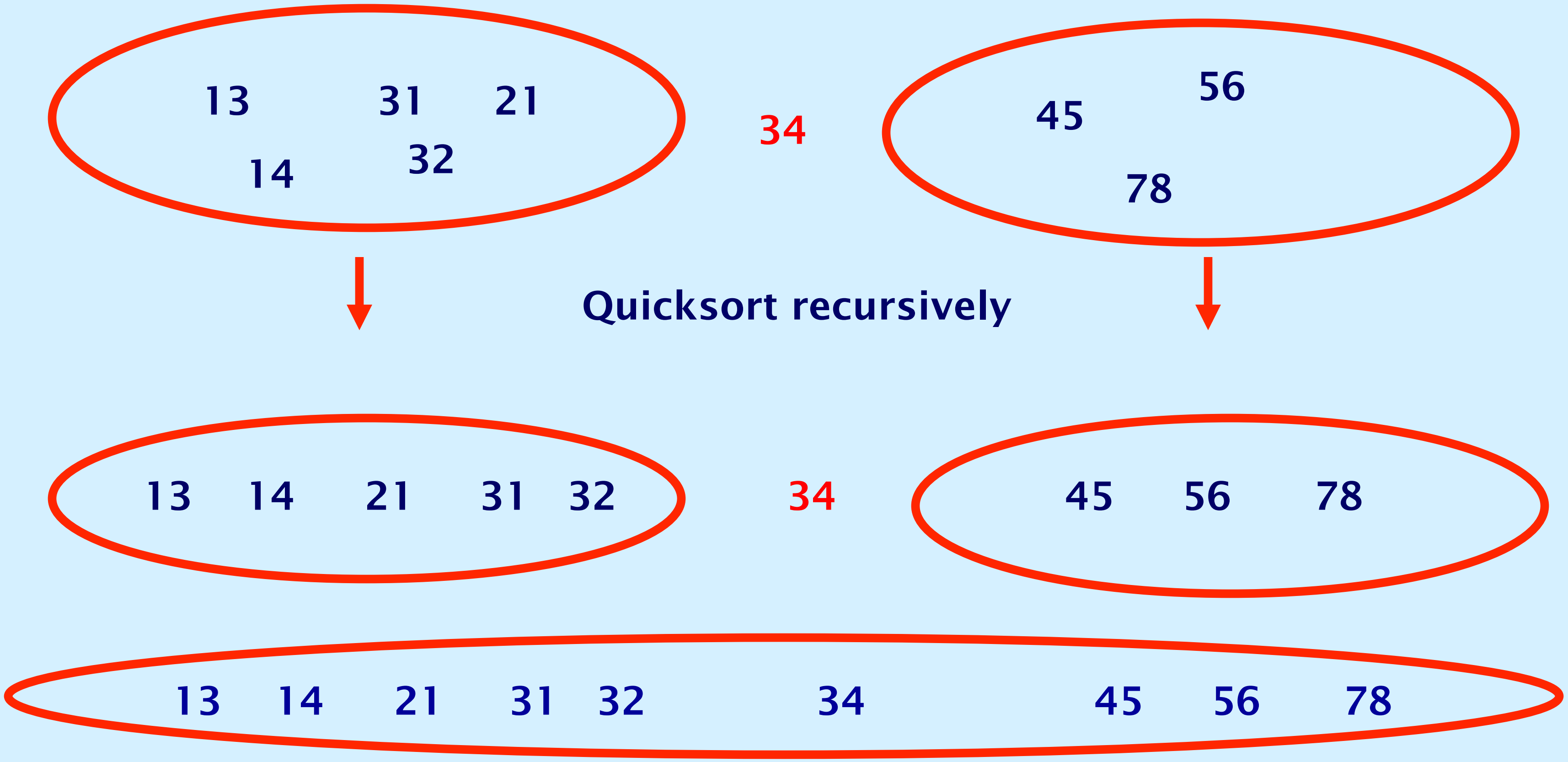


QUICKSORT



13
45
34
14
56
32
21
31
78
Partition around Pivot
13
31
21
14
32
34
45
56
78

QUICKSORT



13 31 21
14 32
34
56
45
78
Quicksort recursively
13 14 21 31 32
34
45 56 78
13 14 21 31 32 34 45 56 78

Parallelizing Quicksort

- Serial Quicksort sorts an array S as follows:
  - If the number of elements in S is 0 or 1, then return.
  - Pick any element *v* in S. Call this **pivot**.
  - Partition the set S−{v} into two disjoint groups:
    - $S_1 = \{x \in S-\{v\} \mid x \leq v\}$
    - $S_2 = \{x \in S-\{v\} \mid x \geq v\}$
  - Return **quicksort(S_1)** followed by **v** followed by **quicksort(S_2)**

Parallel Quicksort (Basic)

- The second recursive call to *qsort* does not depend on the results of the first recursive call
- We have an opportunity to speed up the call by making both calls in parallel.

```
template <typename T>
void qsort(T begin, T end) {

  if (begin != end) {

     T middle = partition(begin, end, …);

     cilk_spawn qsort(begin, middle);
     qsort(max(begin + 1, middle), end);   // No cilk_spawn on this line!
     cilk_sync;
  }
}
```

Actual Performance

- ./qsort 500000 -cilk_set_worker_count 1
  >> 0.083 seconds
- ./qsort 500000 -cilk_set_worker_count 16
  >> 0.014 seconds
- Speedup = T_1/T_{16} = 0.083/0.014 = **5.93**

Actual Performance

- ./qsort 500000 -cilk_set_worker_count 1
  >> 0.083 seconds
- ./qsort 500000 -cilk_set_worker_count 16
  >> 0.014 seconds
- Speedup = T_1/T_{16} = 0.083/0.014 = **5.93**

- ./qsort 50000000 -cilk_set_worker_count 1
  >> 10.57 seconds
- ./qsort 50000000 -cilk_set_worker_count 16
  >> 1.58 seconds
- Speedup = T_1/T_{16} = 10.57/1.58 = **6.67**

Why not better???

Measure Work/Span Empirically

- cilkview -w ./qsort 50000000

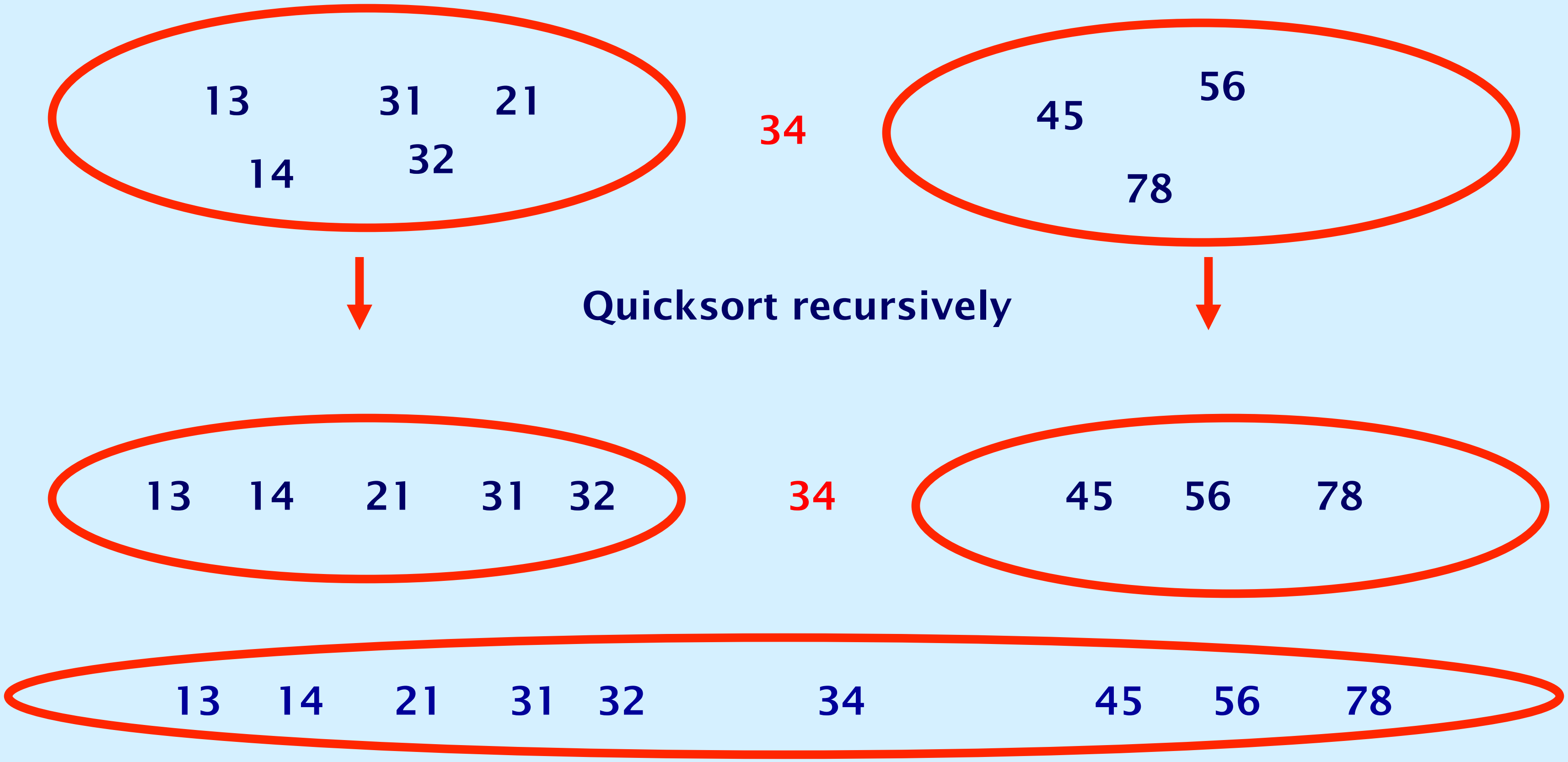
  Work = 21593799861
  Span = 1261403043
  Burdened span = 1261600249
  Parallelism = ***17.1189***
  Burdened parallelism = 17.1162
  #Spawn = 50000000
  #Atomic instructions = 14

- cilkview -w ./qsort 500000

  Work = 178835973
  Span = 14378443
  Burdened span = 14525767
  Parallelism = ***12.4378***
  Burdened parallelism = 12.3116
  #Spawn = 500000
  #Atomic instructions = 8

```
workspan ws;
ws.start();
sample_qsort(a, a + n);
ws.stop();
ws.report(std::cout);
```

Analyzing Quicksort



Assume we have a “great” partitioner that always generates two balanced sets

Analyzing Quicksort

- Work:

$$T_1(n) = 2T_1(n/2) + \Theta(n)$$
$$2T_1(n/2) = 4T_1(n/4) + 2\,\Theta(n/2)$$
....
....
$$+\quad n/2\, T_1(2) = n\, T_1(1) + n/2\, \Theta(2)$$

$$T_1(n) = \Theta(n \lg n)$$

- Partitioning
- not parallel !

- Span recurrence: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

  Solves to $T_\infty(n) = \Theta(n)$

Analyzing Quicksort

Parallelism: $\frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$ **Not much !**

- Indeed, partitioning (i.e., constructing the array $S_1 = \{x \in S-\{v\} \mid x \leq v\}$) can be accomplished in parallel in time $\Theta(\lg n)$
- Which gives a span $T_\infty(n) = \Theta(\lg^2 n)$
- And parallelism $\Theta(n/\lg n)$ **Way better !**

- Basic parallel qsort can be found under $cilkpath/examples/qsort

The Master Method (Optional)

The ***Master Method*** for solving recurrences applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\ ,^{*}$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

IDEA: Compare $n^{\log_b a}$ with $f(n)$.

* The base case is always $T(n) = \Theta(1)$ for sufficiently small n.

Master Method — CASE 1

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \gg f(n)$$

Specifically, $f(n) = O(n^{\log_b a - \epsilon})$ for some const $\epsilon > 0$

Solution: $T(n) = \Theta(n^{\log_b a})$

Strassen matrix multiplication: $a = 7$, $b = 2$, $f(n) = n^2$

➔ $T_1(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$

Master Method — Case 2

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \approx f(n)$$

Specifically, $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some const $k \geq 0$

Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n))$

quicksort work: $a=2$, $b=2$, $f(n)=n$, $k=0$ ➔ $T_1(n) = \Theta(n \lg n)$
qsort span: $a=1$, $b=2$, $f(n)=\lg n$, $k=1$ ➔ $T_\infty(n) = \Theta(\lg^2 n)$

Master Method — CASE 3

$$T(n) = a\,T(n/b) + f(n)$$

$$n^{\log_b a} \ll f(n)$$

Specifically, $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some const $\epsilon > 0$,

and *(regularity)* $a \cdot f(n/b) \le c \cdot f(n)$ for some const $c < 1$

Solution: $T(n) = \Theta(f(n))$

Eg: qsort span (bad version): $a=1$, $b=2$, $f(n)=n$ ➔ $T_\infty(n) = \Theta(n)$

Master Method Summary

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \epsilon})$, constant $\epsilon > 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$

$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, constant $\epsilon > 0$, and regularity condition

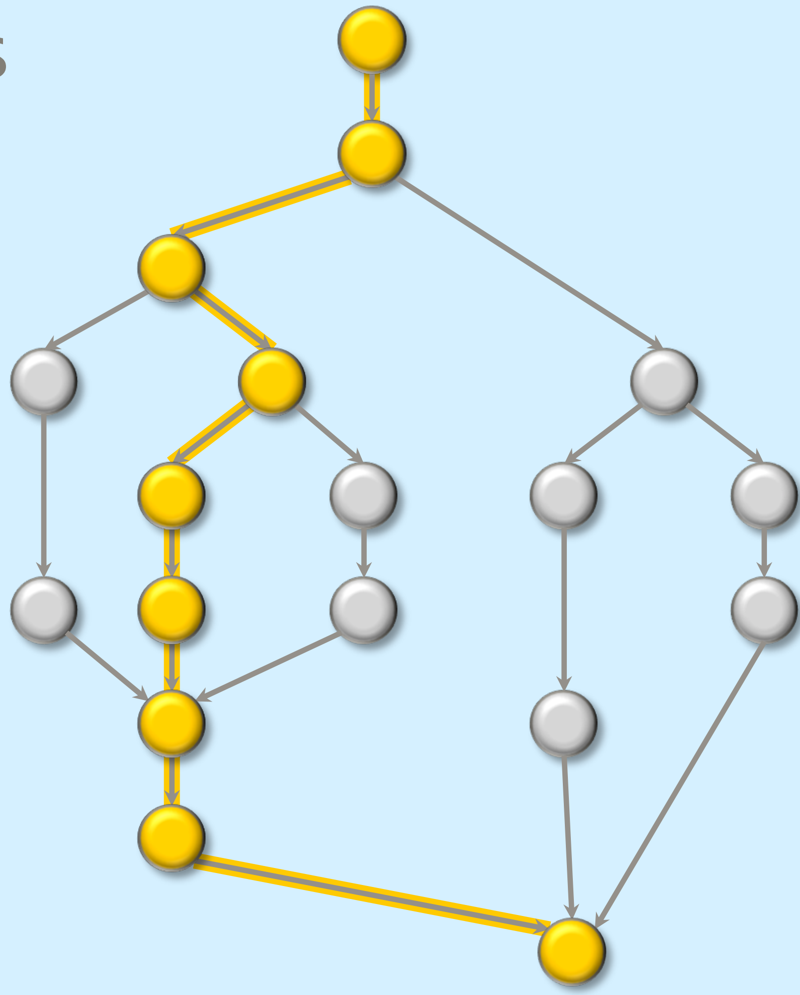
$\Rightarrow T(n) = \Theta(f(n))$.

Potential Parallelism

Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = ***potential parallelism***

= the average amount of work per step along the span.



Three Tips on Parallelism

1. *Minimize span* to maximize parallelism. Try to generate 10 times more parallelism than processors for near-perfect linear speedup.
2. If you have plenty of parallelism, try to trade some if it off for *reduced work overheads*.
3. Use *divide-and-conquer recursion* or *parallel loops* rather than spawning one small thing off after another.

Do this:

```
cilk_for (int i=0; i<n; ++i) {
    foo(i);
}
```

Not this:

```
for (int i=0; i<n; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```

Three Tips on Overheads

1. Make sure that work/#spawns is not too small.
   - Coarsen by using function calls and *inlining* near the leaves of recursion rather than spawning.

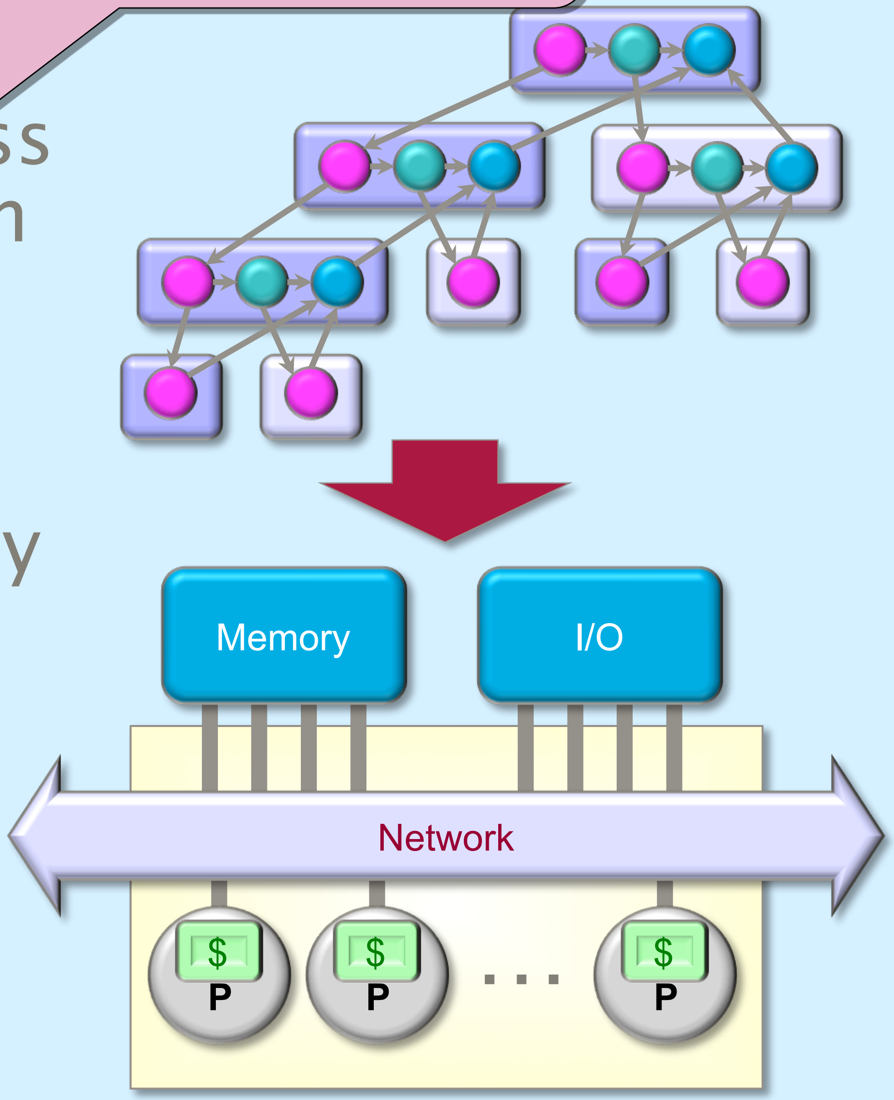
2. Parallelize *outer loops* if you can, not inner loops (otherwise, you'll have high *burdened parallelism*, which includes runtime and scheduling overhead). If you must parallelize an inner loop, coarsen it, but not too much.
   - 500 iterations should be plenty coarse for even the most meager loop. Fewer iterations should suffice for "fatter" loops.

3. Use *reducers* only in sufficiently fat loops.

Scheduling

A strand is a sequence of instructions that doesn't contain any parallel constructs

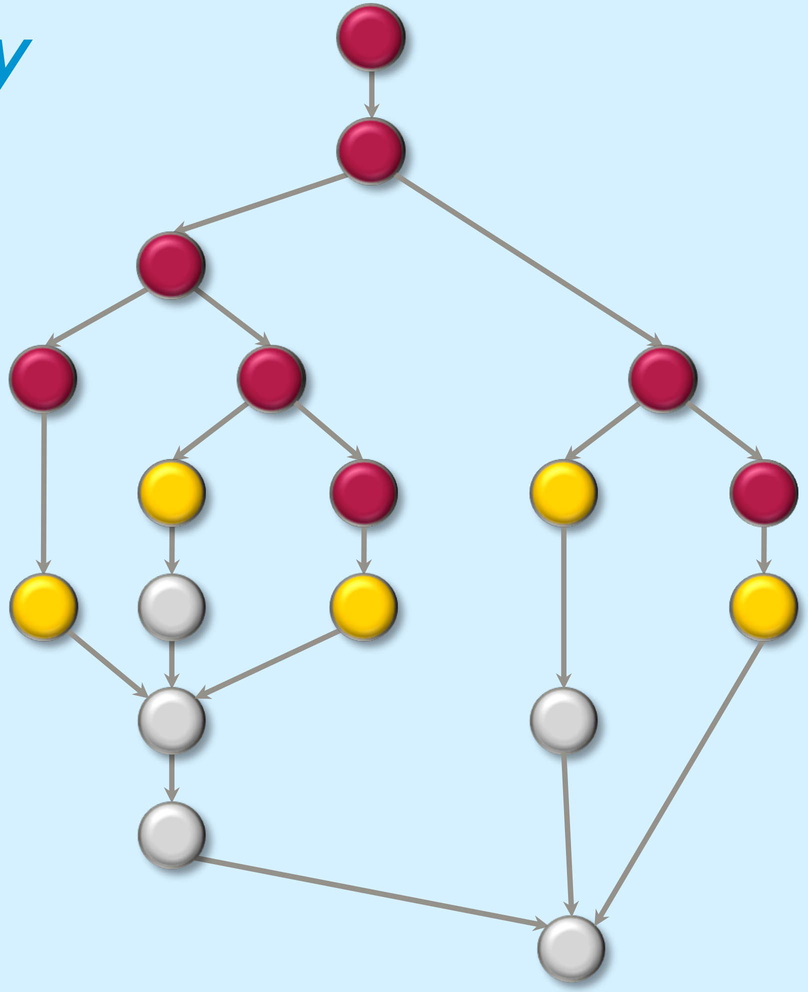
- Cilk allows the programmer to express *potential* parallelism in an application.
- The Cilk ***scheduler*** maps strands onto processors dynamically at runtime.
- Since ***on-line*** schedulers are complicated, we'll explore the ideas with an ***off-line*** scheduler.



Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A strand is ***ready*** if all its predecessors have executed.



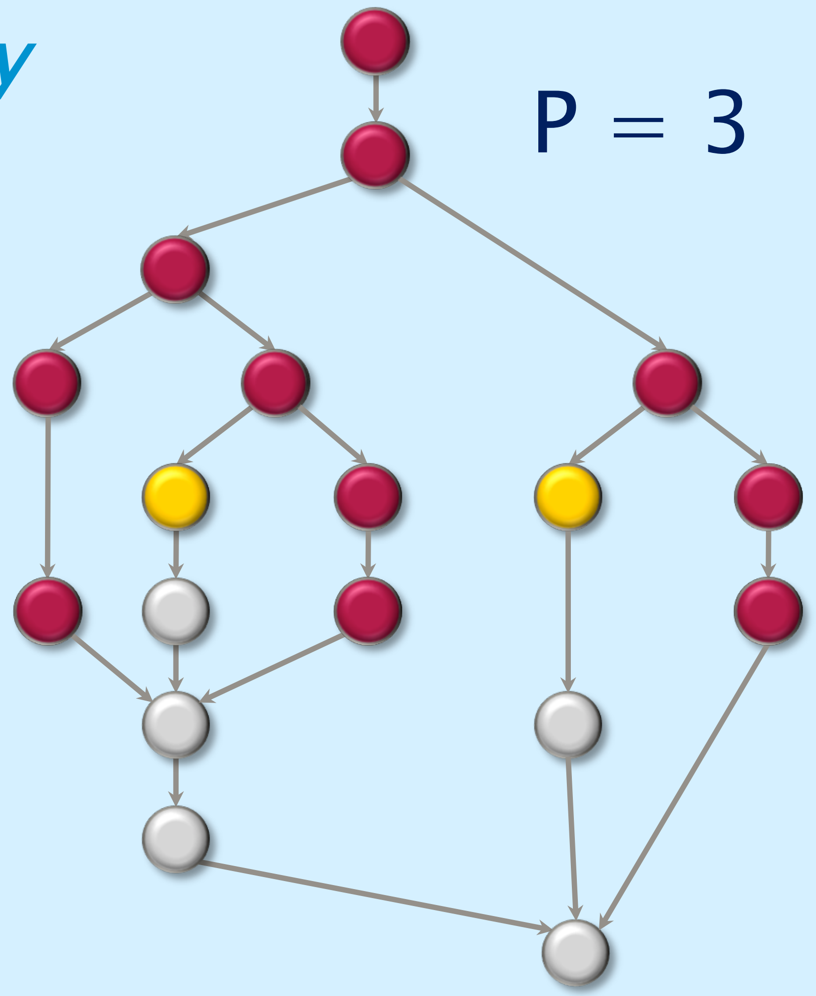
Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A strand is ***ready*** if all its predecessors have executed.

Complete step

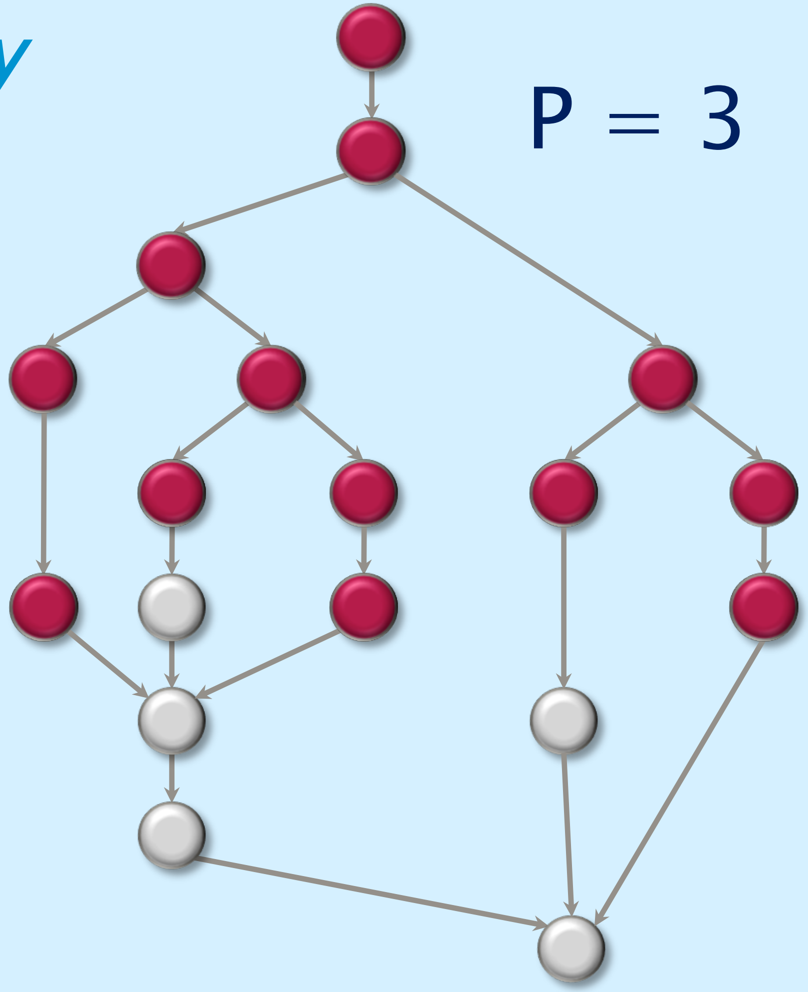
- ≥ P strands ready.
- Run any P.



Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A strand is ***ready*** if all its predecessors have executed.



Complete step

- $\geq$ P strands ready.
- Run any P.

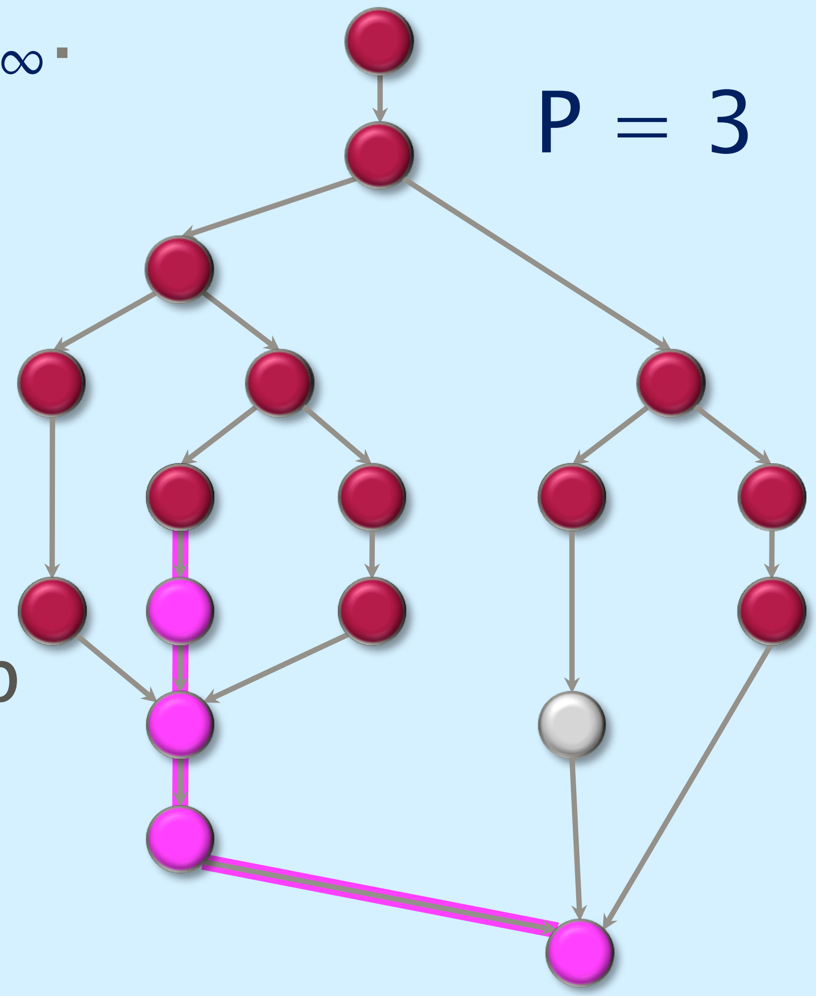
Incomplete step

- $<$ P strands ready.
- Run all of them.

Analysis of Greedy

Theorem : Any greedy scheduler achieves

$$T_P \le T_1/P + T_\infty .$$



Proof.

- \# complete steps $\le T_1/P$, since each complete step performs P work.
- \# incomplete steps $\le T_\infty$, since each incomplete step reduces the span of the <u>unexecuted dag</u> by 1. ■

Optimality of Greedy

Theorem. Any greedy scheduler achieves within a factor of 2 of optimal.

Proof. Let T_P^* be the execution time produced by the optimal scheduler. Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ by the Work and Span Laws, we have

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max\{T_1/P, T_\infty\} \\ &\leq 2T_P^* . \quad \blacksquare \end{aligned}$$

Linear Speedup

Theorem. Any greedy scheduler achieves near-perfect linear speedup whenever $P \ll T_1/T_\infty$.

Proof. Since $P \ll T_1/T_\infty$ is equivalent to $T_\infty \ll T_1/P$, the Greedy Scheduling Theorem gives us

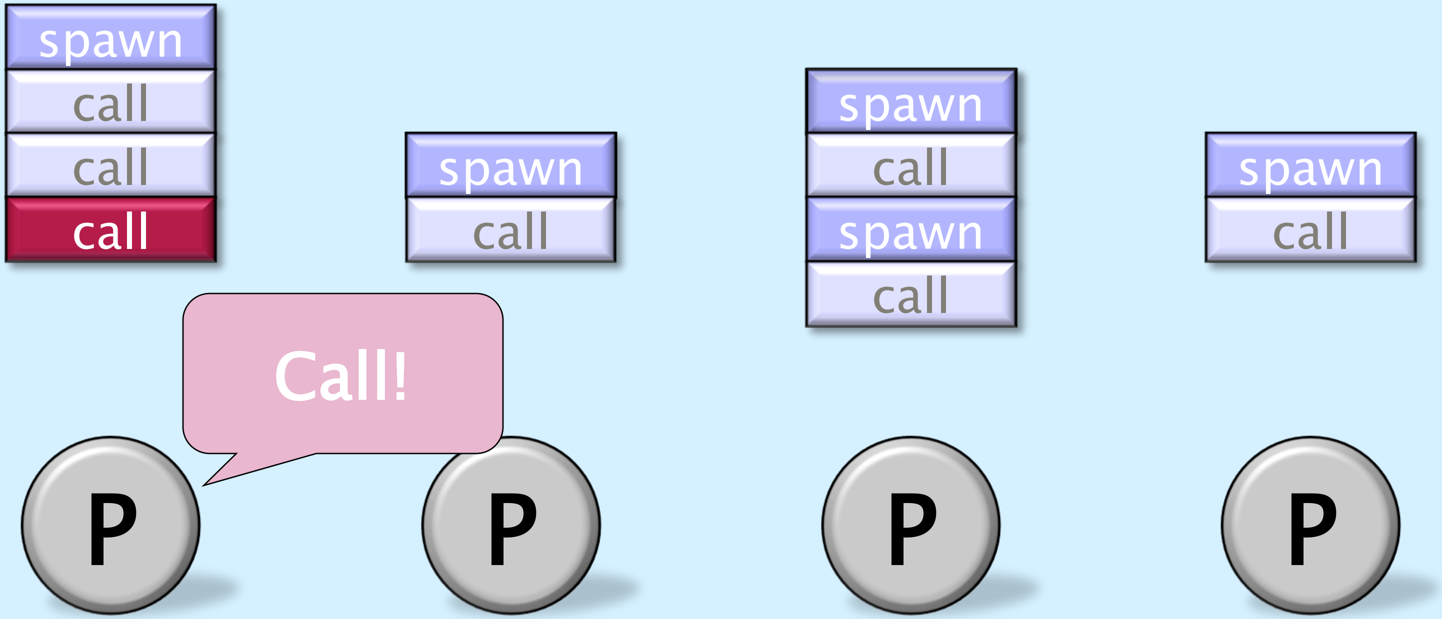
$$\begin{aligned} T_P &\le T_1/P + T_\infty \\ &\approx T_1/P . \end{aligned}$$

Thus, the speedup is $T_1/T_P \approx P$. ■

Definition. The quantity T_1/PT_∞ is called the ***parallel slackness***.

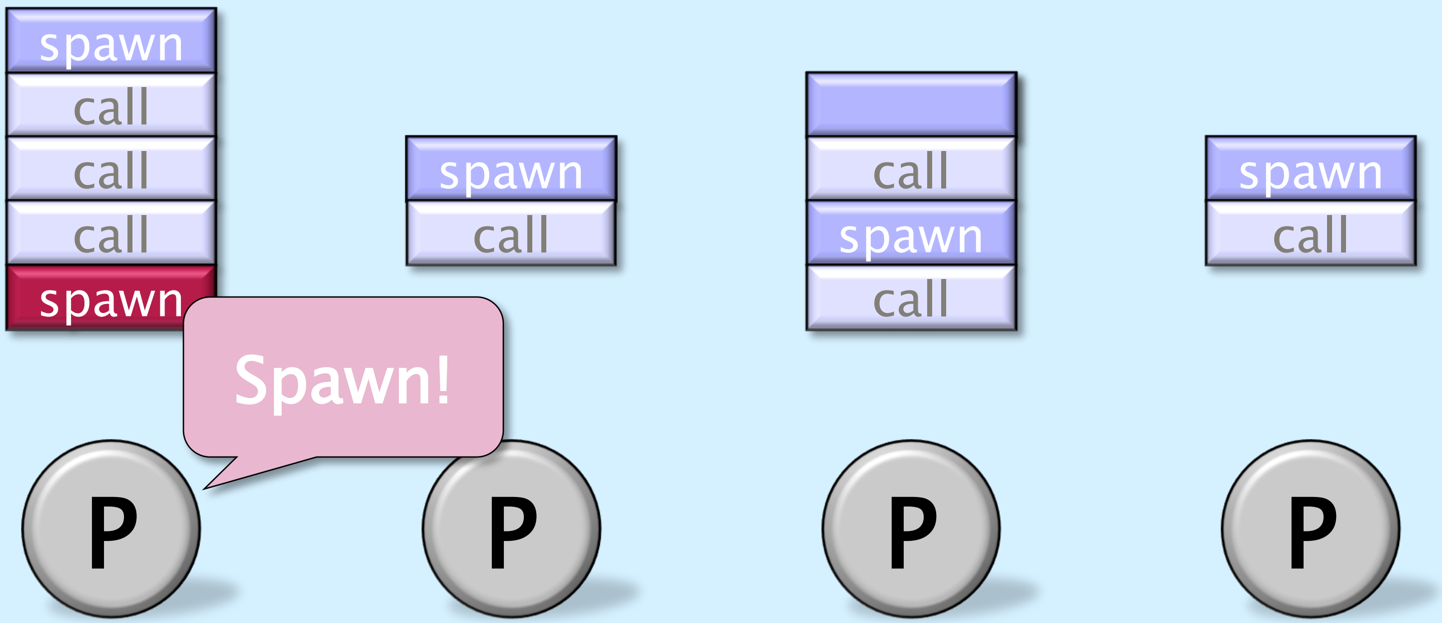
Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



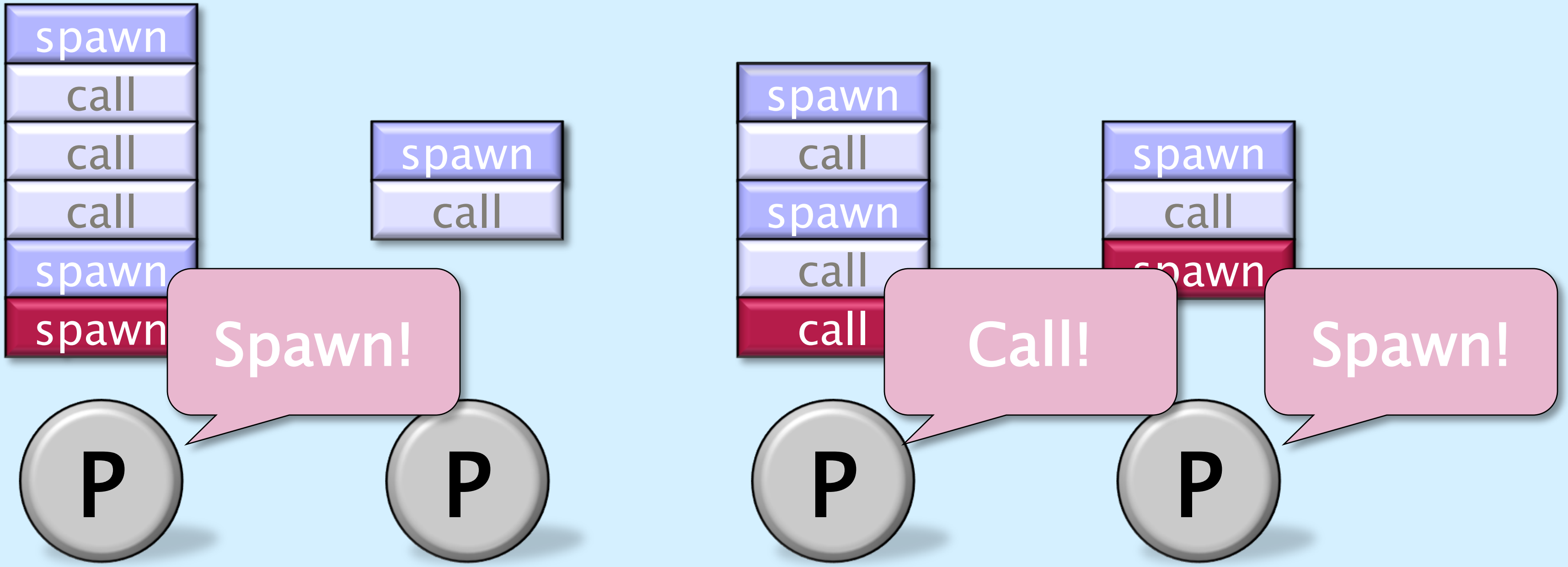
Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



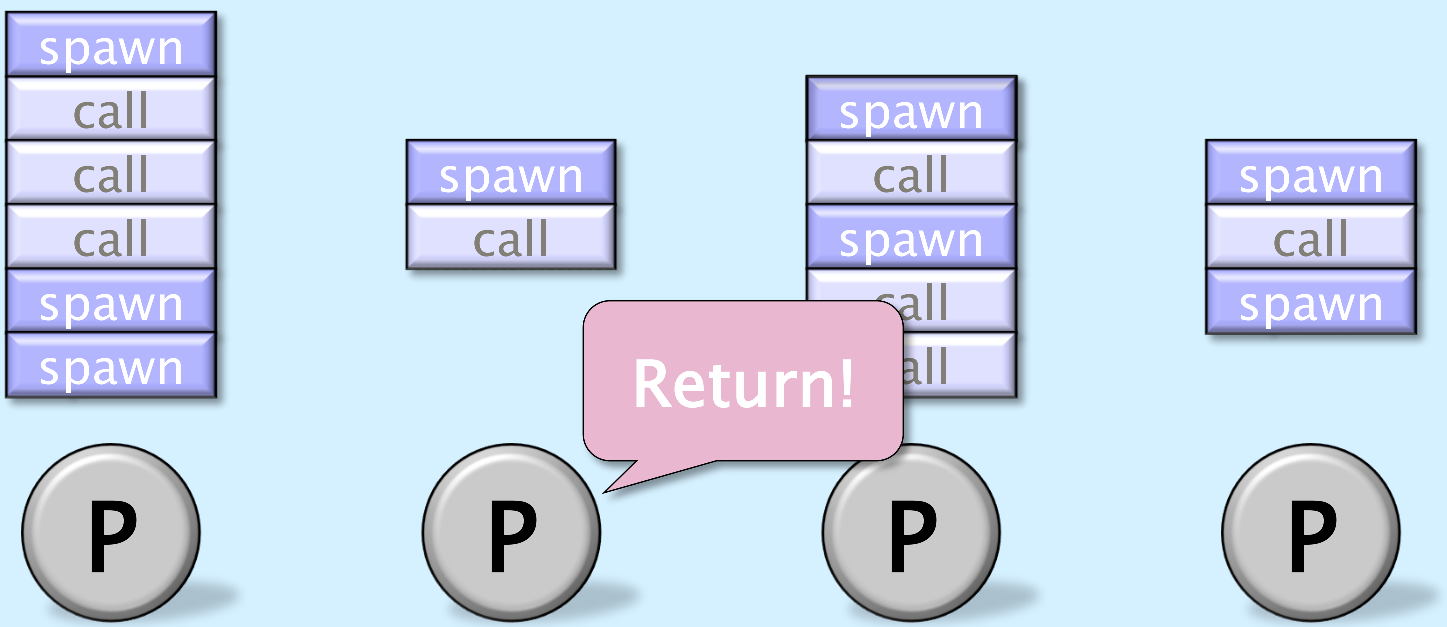
Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



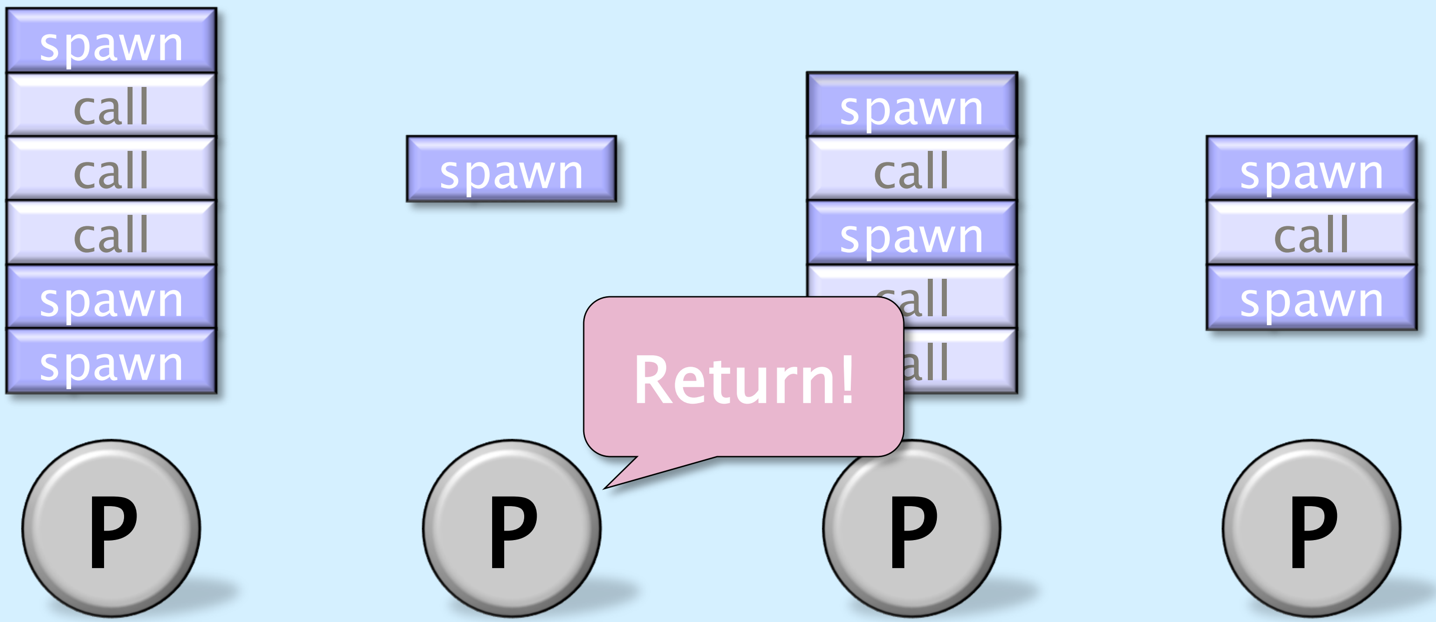
Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



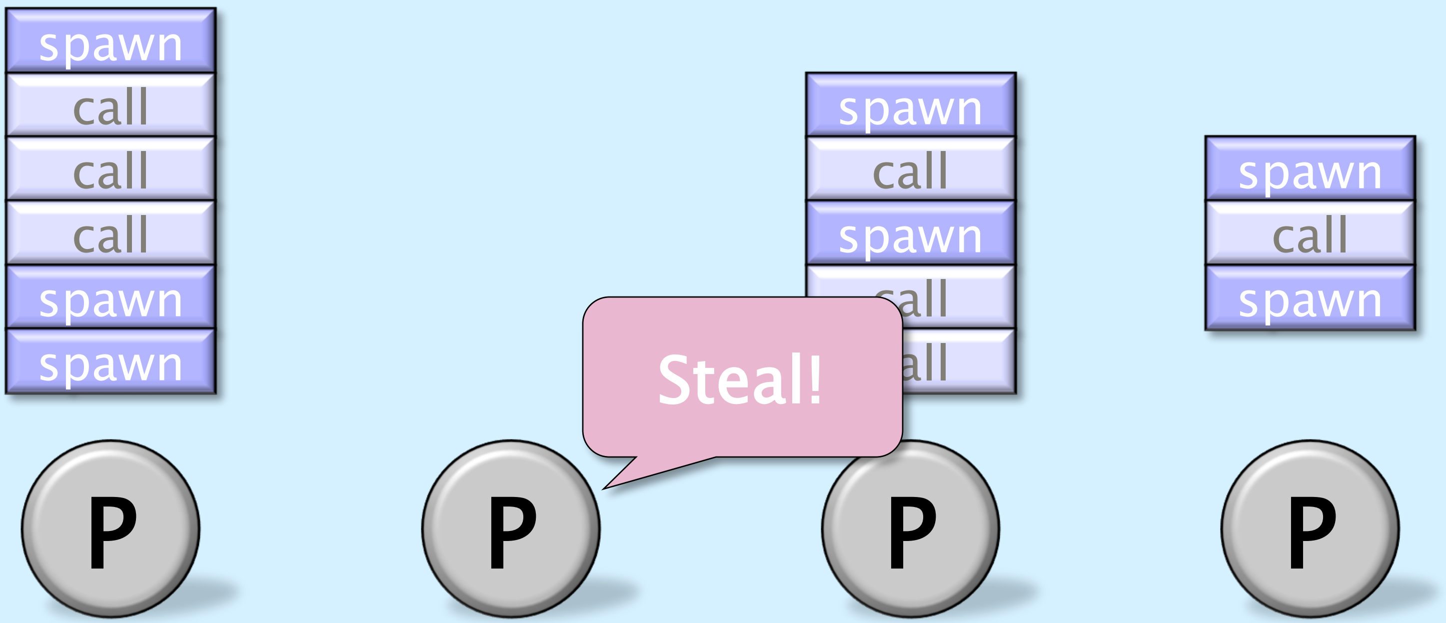
Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack

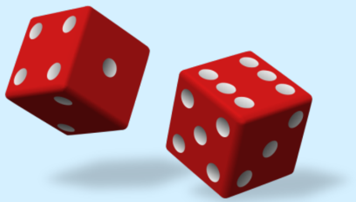


Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack

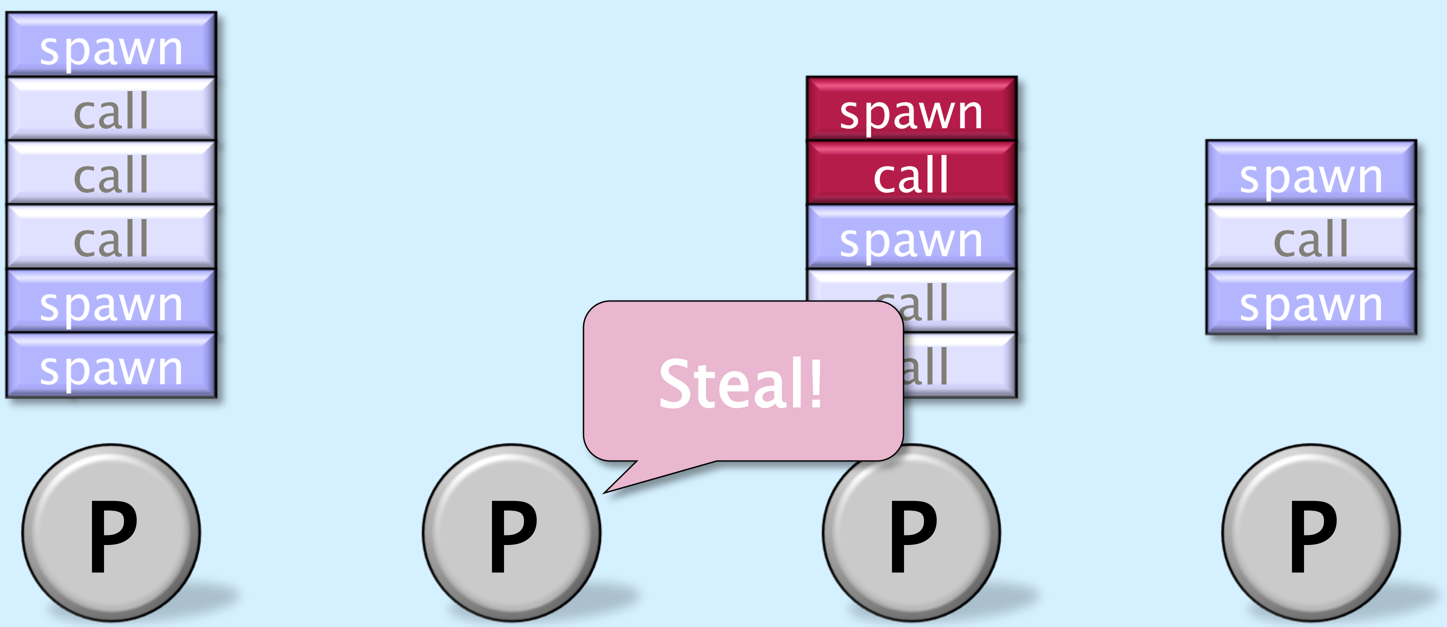


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.

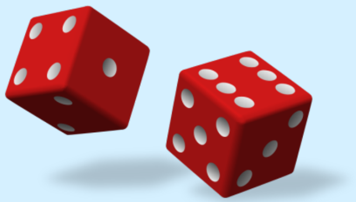


Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack

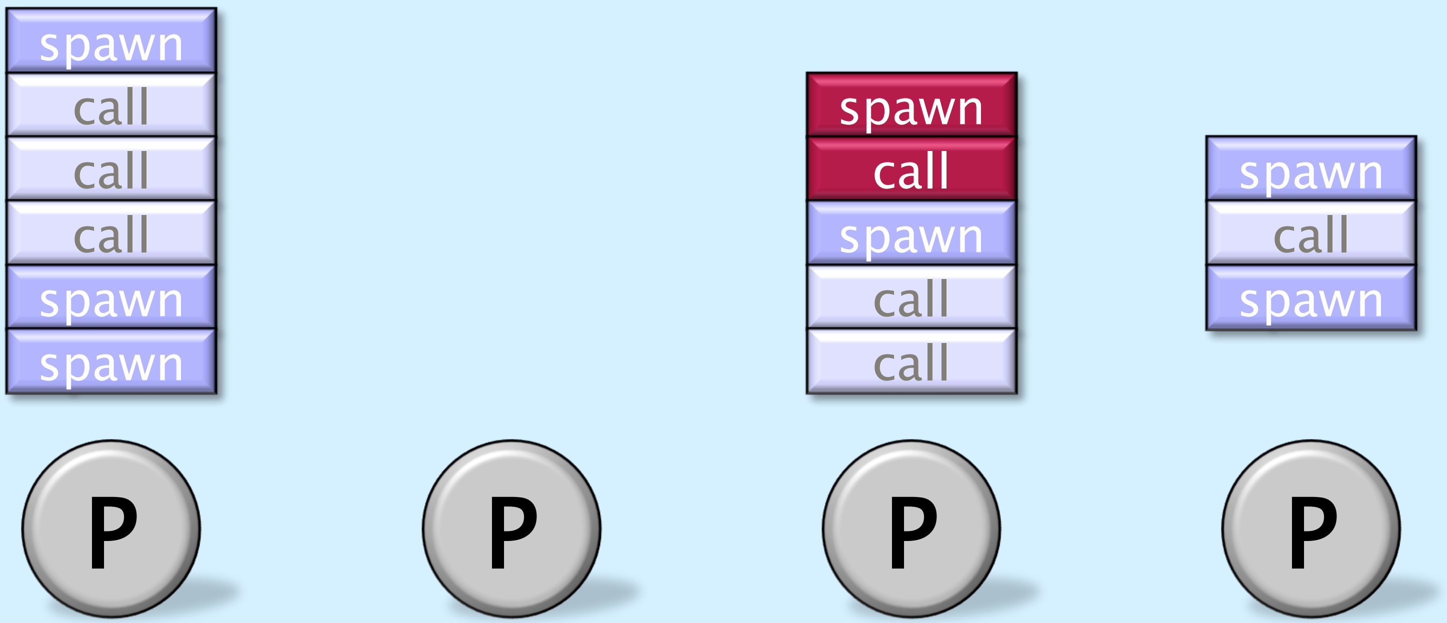


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.

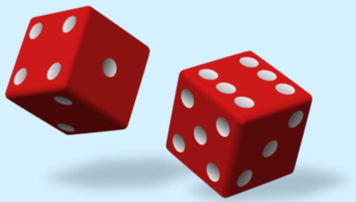


Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack

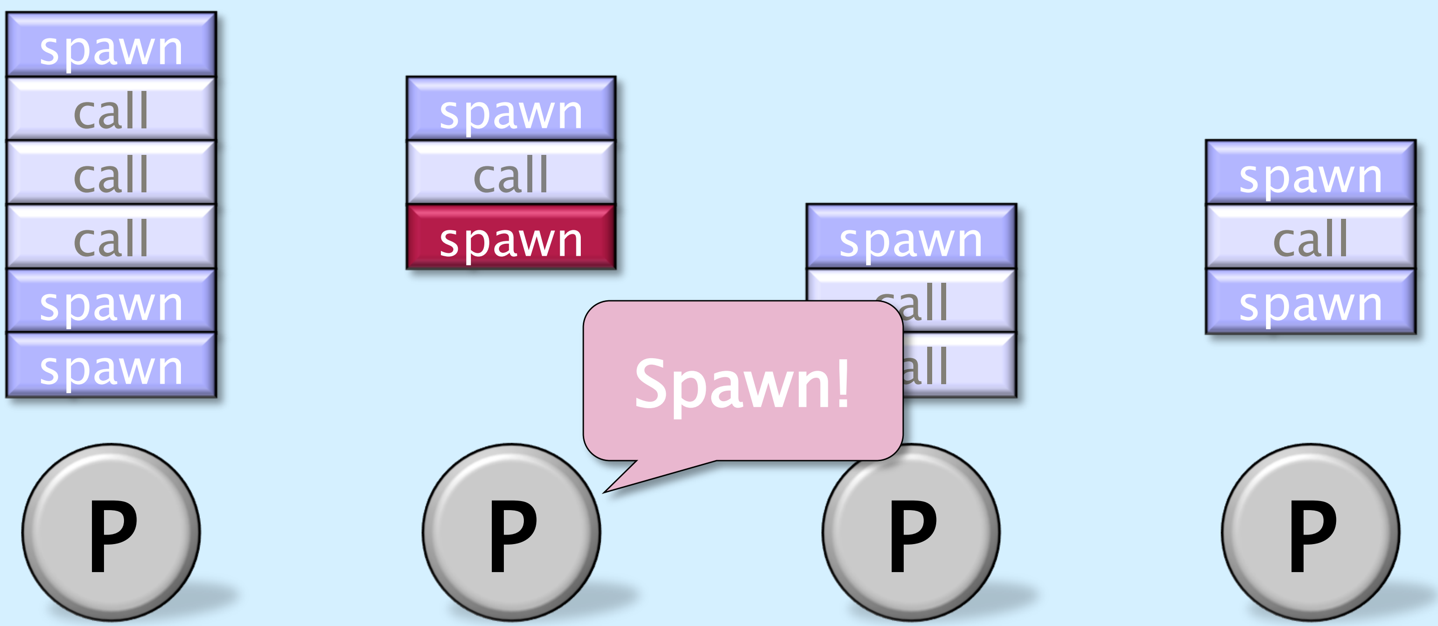


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.



Cilk Runtime System

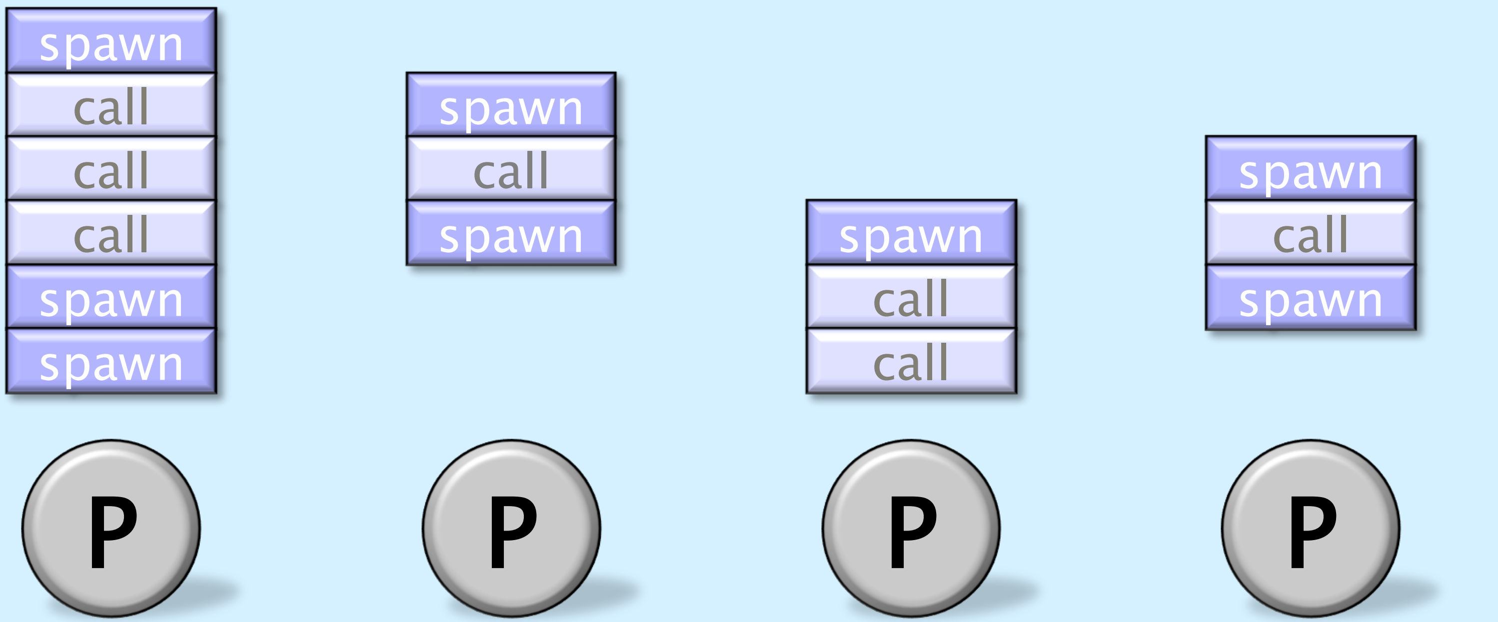
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



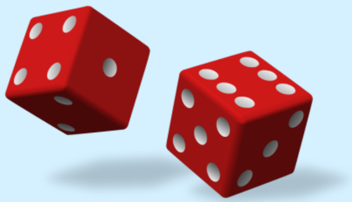
When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.

Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack

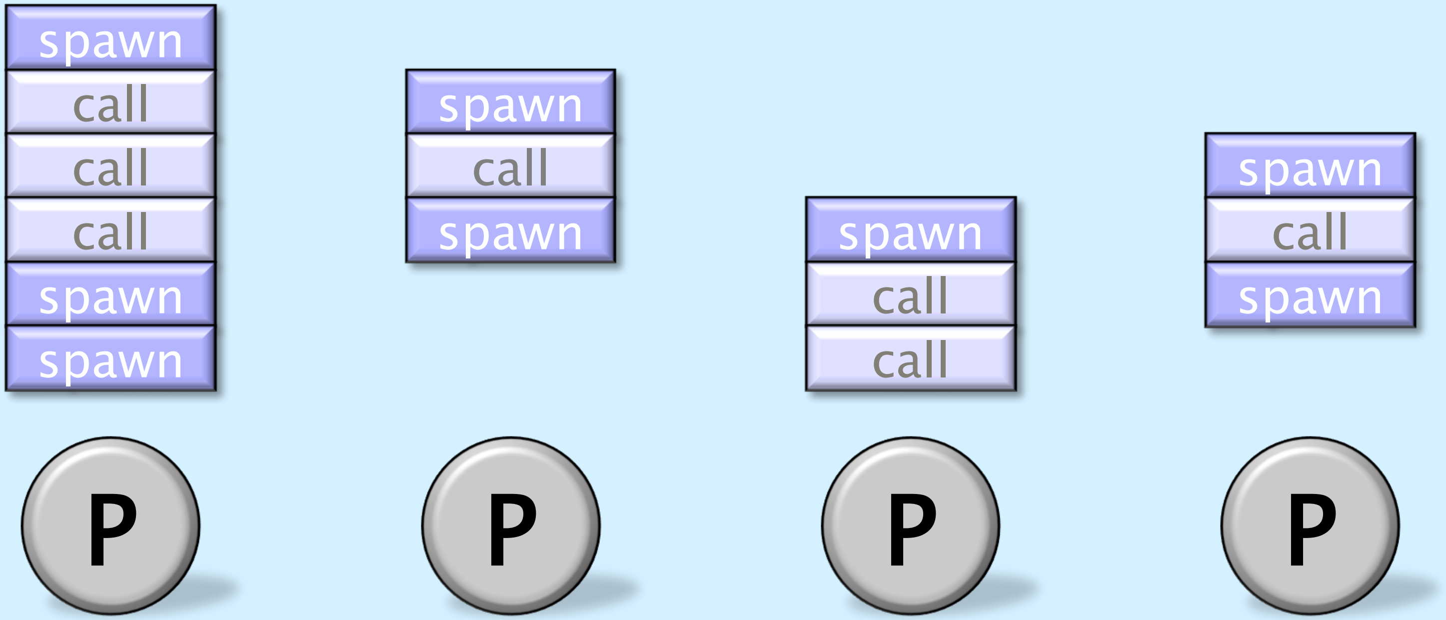


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.



Cilk Runtime System

Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



Theorem: With sufficient parallelism, workers steal infrequently ⇒ ***linear speed-up***.