

# 6.172 Performance Engineering of Software Systems



## LECTURE 7 Races and Parallelism

Julian Shun



# Recall: Basics of Cilk

```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

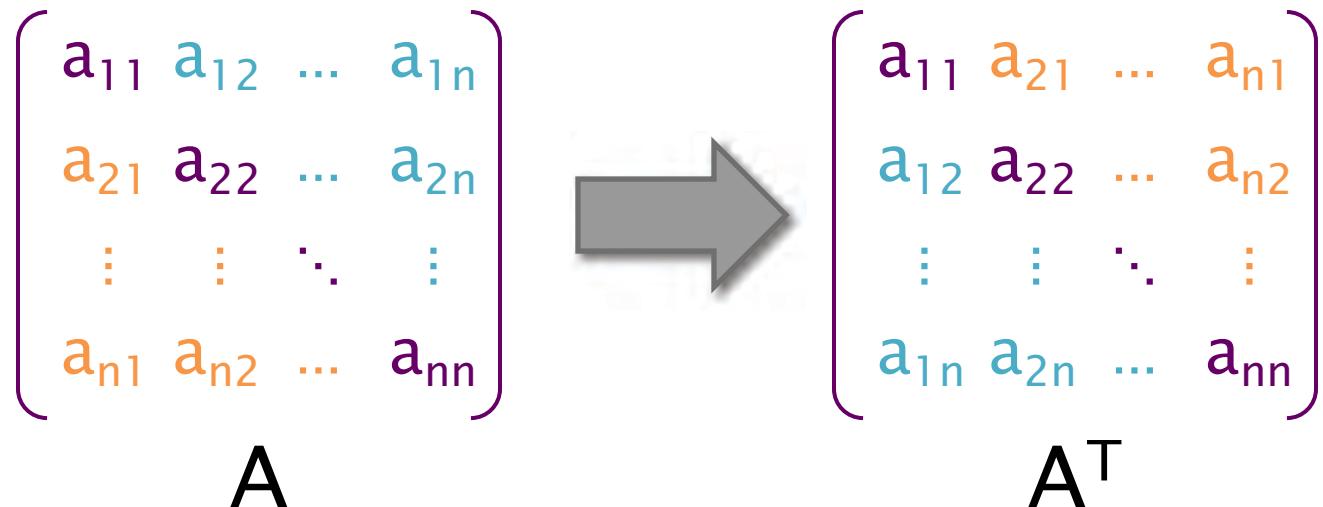
The named *child* function may execute in parallel with the *parent* caller.

Control cannot pass this point until all spawned children have returned.

Cilk keywords grant permission for parallel execution.  
They do not command parallel execution.

# Loop Parallelism in Cilk

Example:  
In-place  
matrix  
transpose

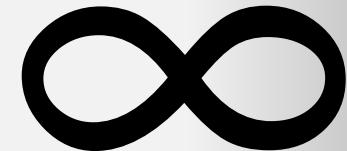


The iterations  
of a `cilk_for`  
loop execute  
in parallel.

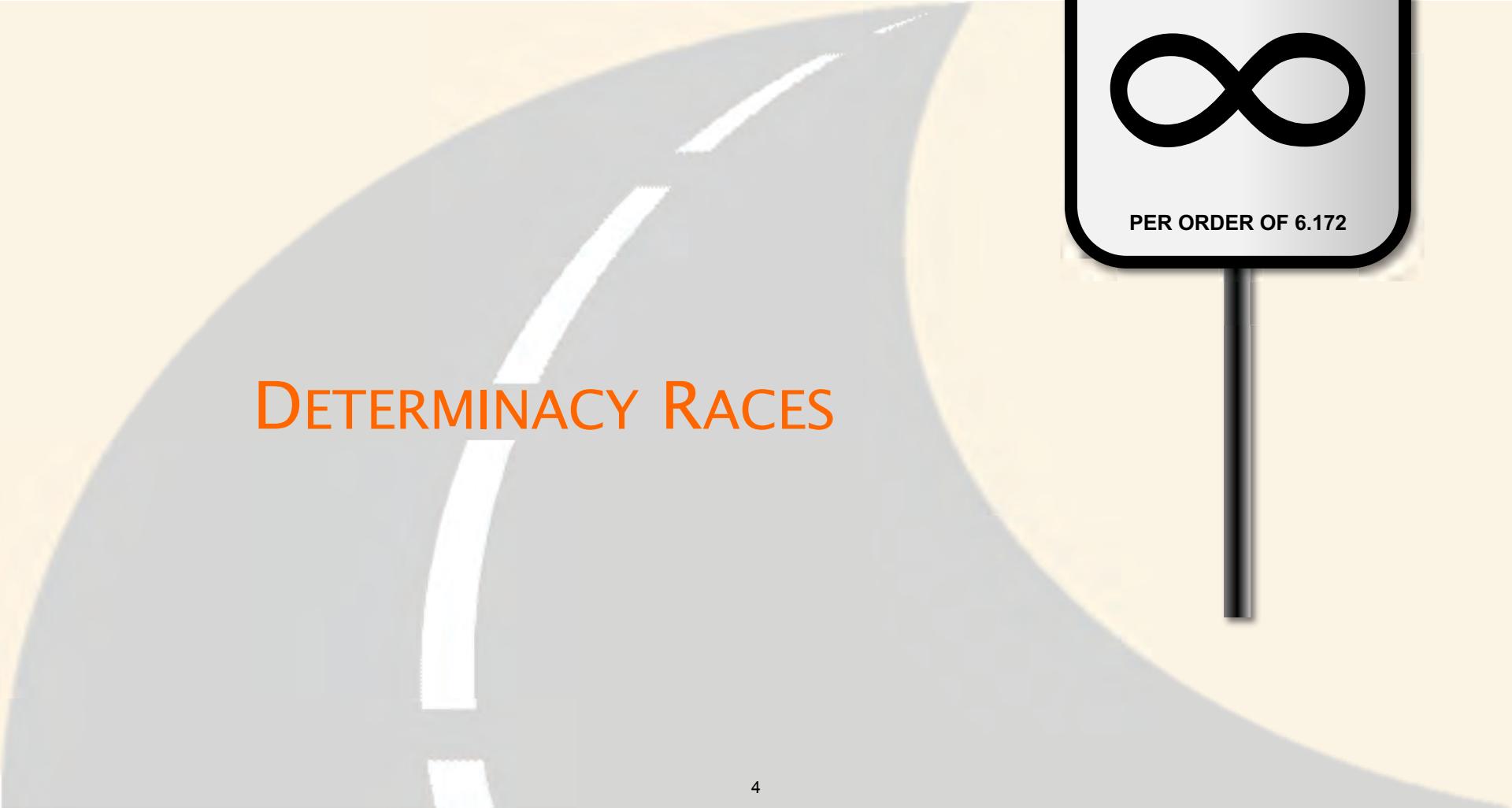
```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```



**SPEED  
LIMIT**



PER ORDER OF 6.172



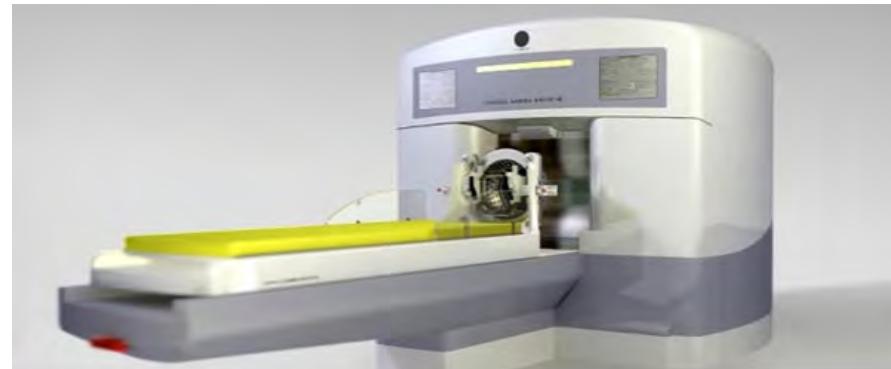
**DETERMINACY RACES**

# Race Conditions

**Race conditions** are the bane of concurrency.  
Famous race bugs include the following:

- ▶ Therac-25 radiation therapy machine — killed 3 people and seriously injured many more.
- ▶ North American Blackout of 2003 — left 50 million people without power.

*Race bugs are notoriously difficult to discover by conventional testing!*



© source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>



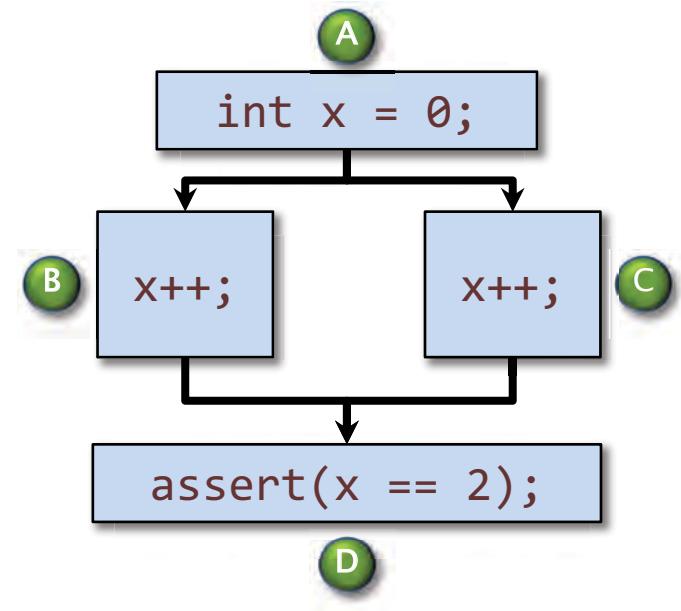
Image created by MIT OpenCourseWare from [public domain image](#).

# Determinacy Races

**Definition.** A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

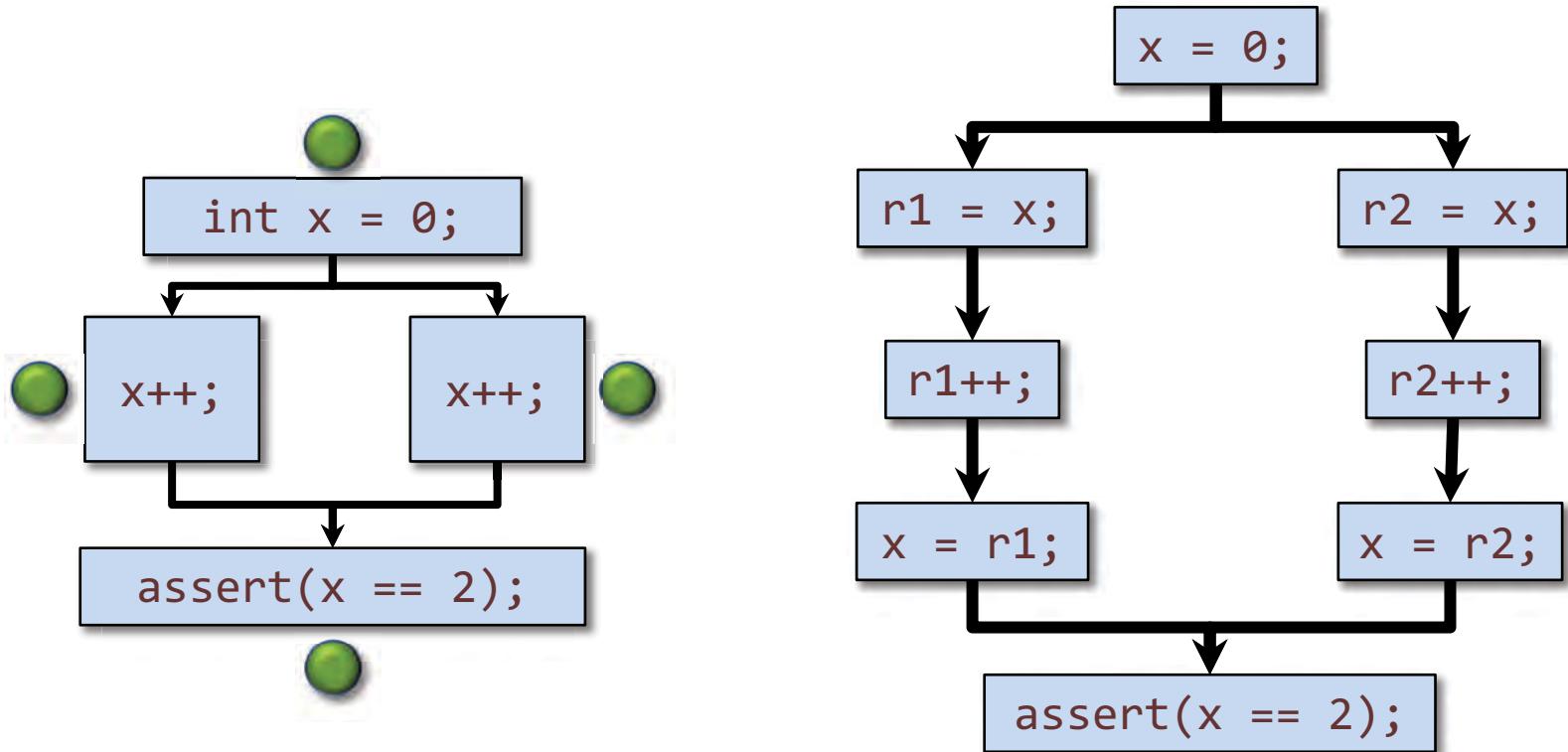
## Example

```
int x = 0;  
cilk_for (int i=0, i<2, ++i) {  
    x++;  
}  
assert(x == 2);
```

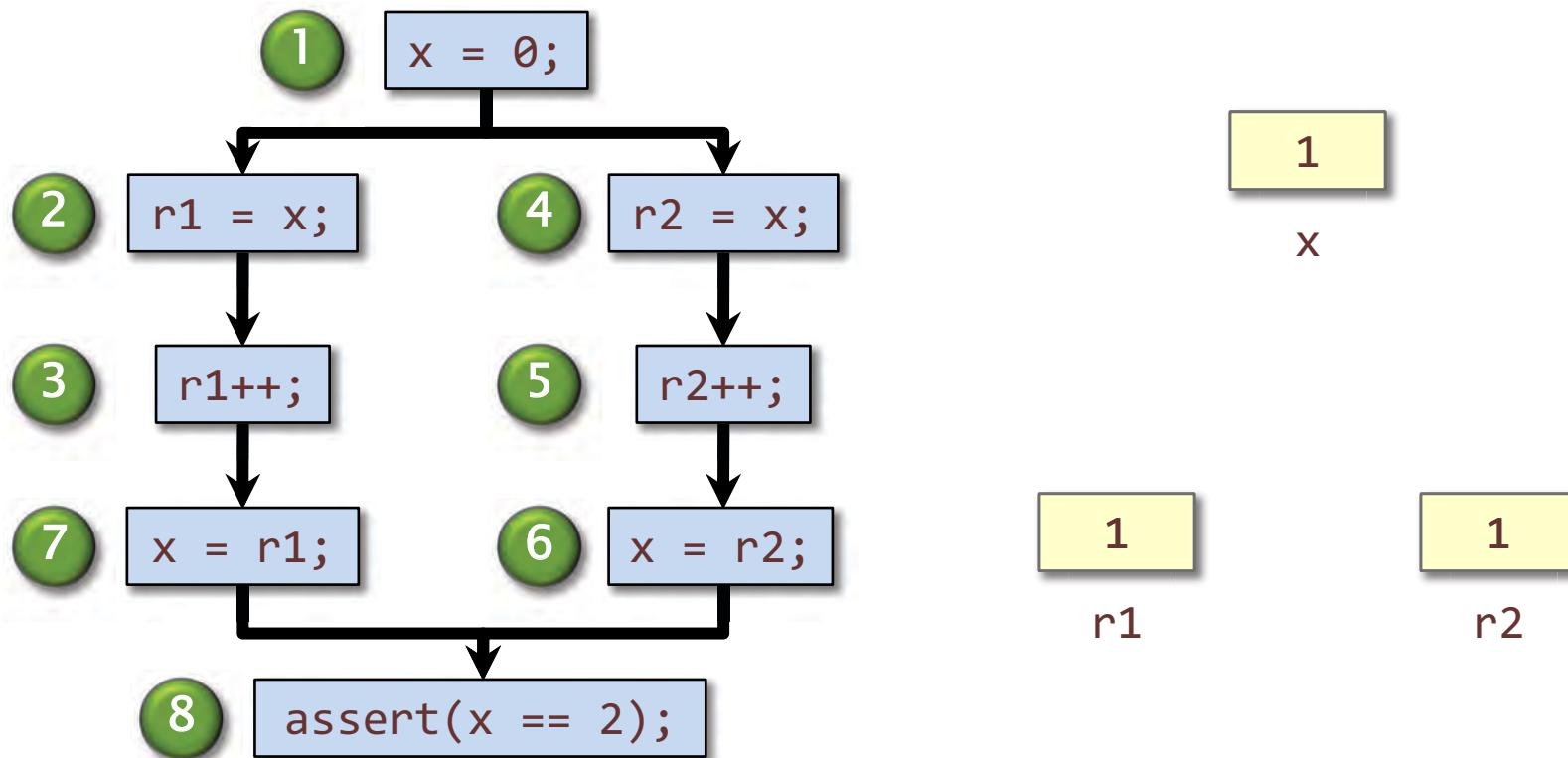


dependency graph

# A Closer Look



**Definition.** A **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.



# Types of Races

Suppose that instruction A and instruction B both access a location x, and suppose that A||B (A is parallel to B).

A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

Two sections of code are independent if they have no determinacy races between them.

# Avoiding Races

- Iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.
  - **Note:** The arguments to a spawned function are evaluated in the parent before the spawn occurs.
- Machine word size matters. Watch out for races in packed data structures:

```
struct {  
    char a;  
    char b;  
} x;
```

**Ex.** Updating `x.a` and `x.b` in parallel may cause a race! Nasty, because it may depend on the compiler optimization level. (Safe on Intel x86-64.)

# Cilksan Race Detector

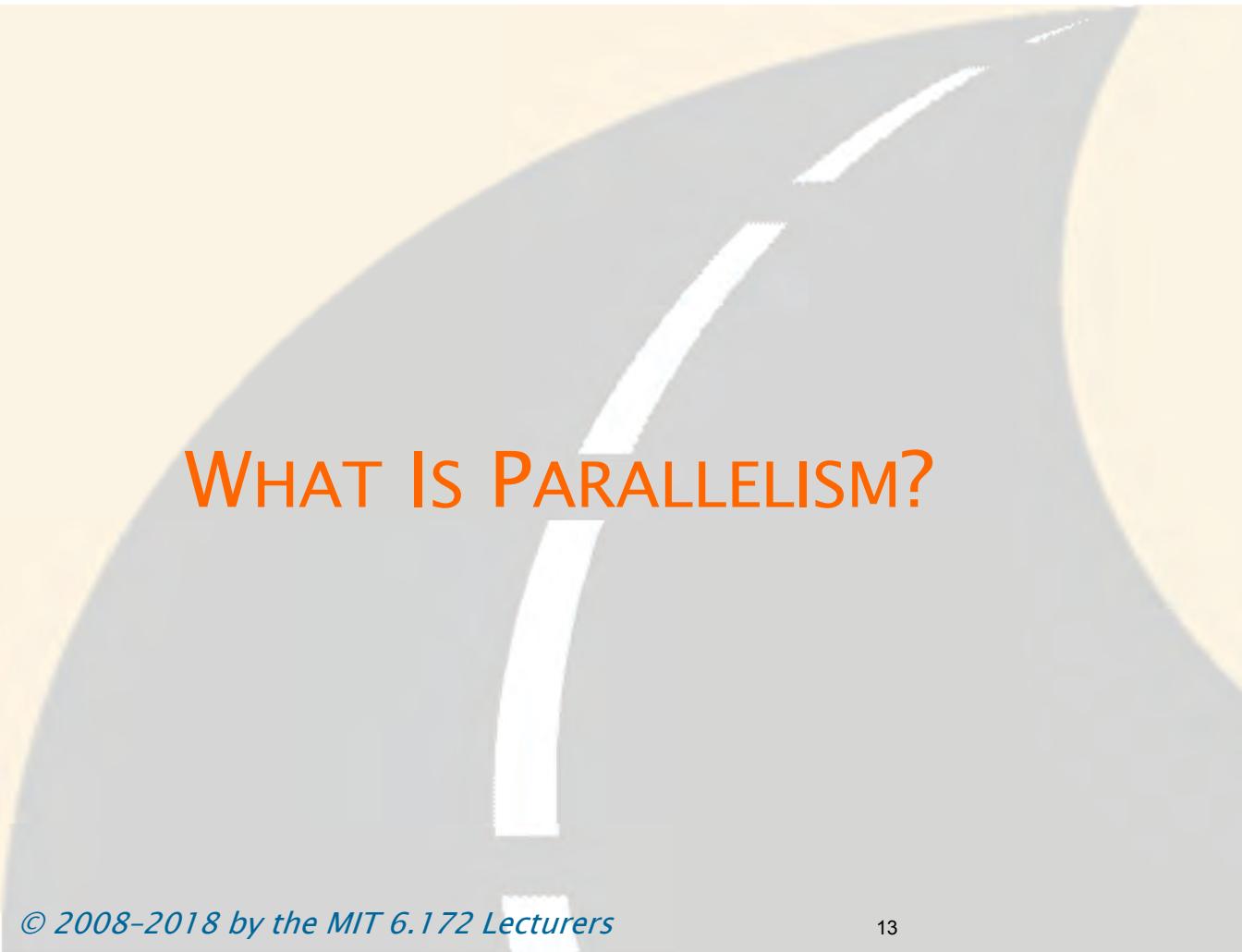
- The Cilksan-instrumented program is produced by compiling with the `-fsanitize=cilk` command-line compiler switch.
- If an ostensibly deterministic Cilk program run on a given input could possibly behave any differently than its serial elision, Cilksan **guarantees** to report and localize the offending race.
- Cilksan employs a **regression-test** methodology, where the programmer provides test inputs.
- Cilksan **identifies** filenames, lines, and variables involved in races, including stack traces.
- Ensure that **all** program files are instrumented, or you'll miss some bugs.
- Cilksan is your **best friend**.

# Cilksan Output

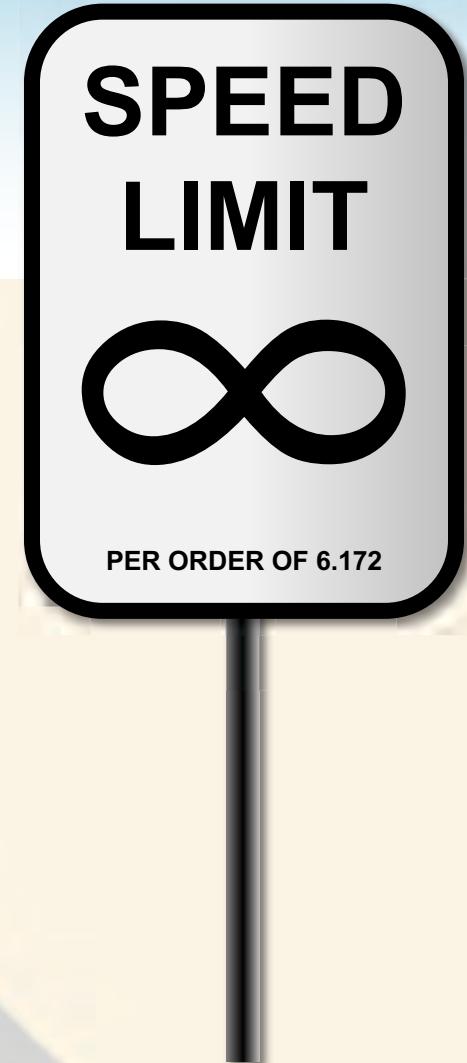
```
$ cilksan ./mm_dac
...
Race detected at address 0x65c070
    Write access to C (declared at mm/mm_dac.c:27)
        from 0x400ed0 mm_base mm/mm_dac.c:34:15
            Called from 0x401868 mm_dac mm/mm_dac.c:57:5
            Called from 0x4025d0 mm_dac mm/mm_dac.c:63:5
            Spawns from 0x401548 mm_dac mm/mm_dac.c:63:5
            Called from 0x4025d0 mm_dac mm/mm_dac.c:63:5
            Spawns from 0x401548 mm_dac mm/mm_dac.c:63:5
    Read access to C (declared at mm/mm_dac.c:27)
        from 0x400e27 mm_base mm/mm_dac.c:34:15
            Called from 0x401868 mm_dac mm/mm_dac.c:57:5
Common calling context
    Called from 0x401c02 main mm/mm_dac.c:85:3
```

0.686637

Race detector detected total of 47 races.  
Race detector suppressed 147409 duplicate error messages



WHAT IS PARALLELISM?



# Execution Model

```
int fib (int n) {  
    if (n < 2) return n;  
    else {  
        int x, y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return x + y;  
    }  
}
```

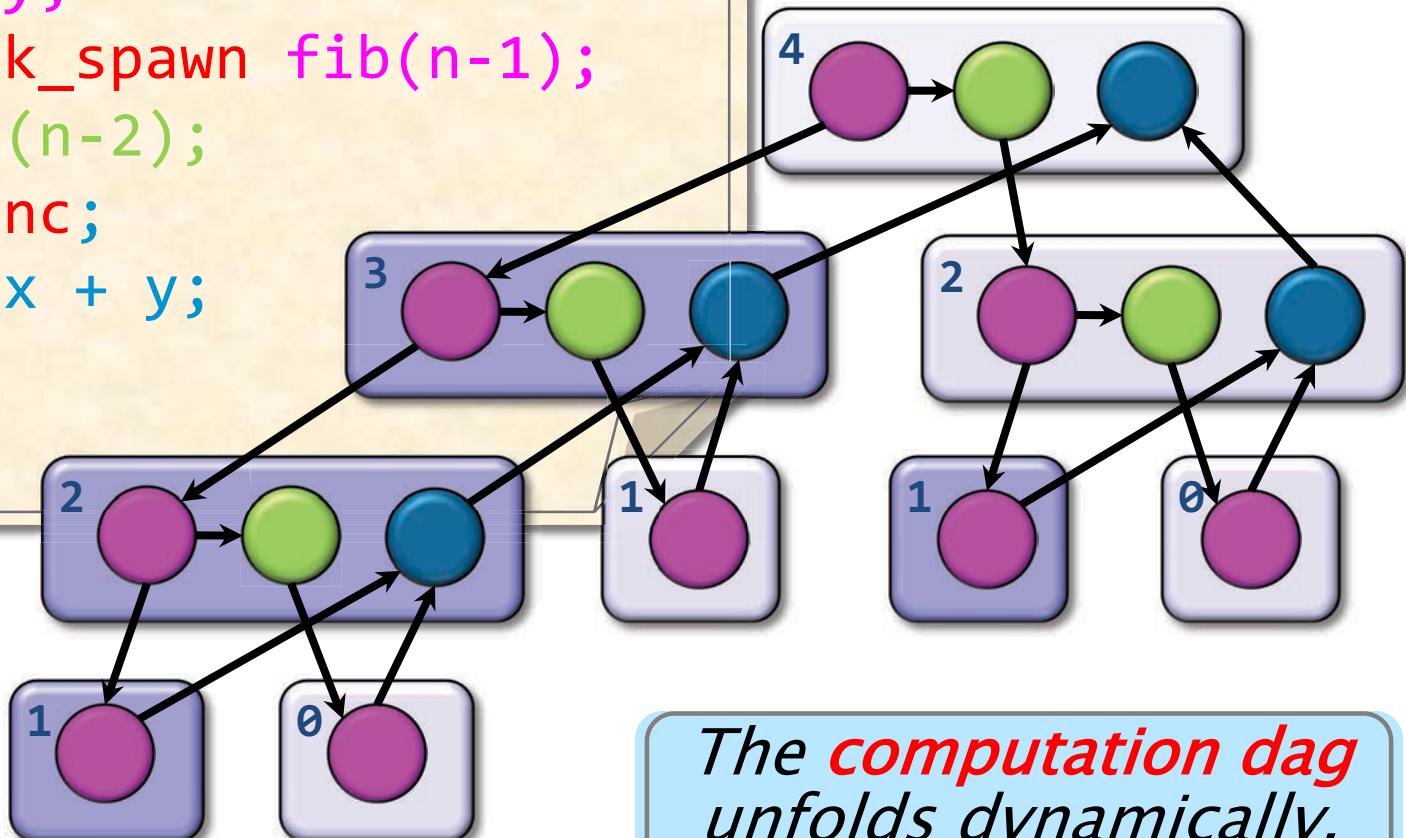
Example:  
`fib(4)`

# Execution Model

```
int fib (int n) {  
    if (n < 2) return n;  
    else {  
        int x, y;  
        x = cilk_spawn fib(n-1);  
        y = fib(n-2);  
        cilk_sync;  
        return x + y;  
    }  
}
```

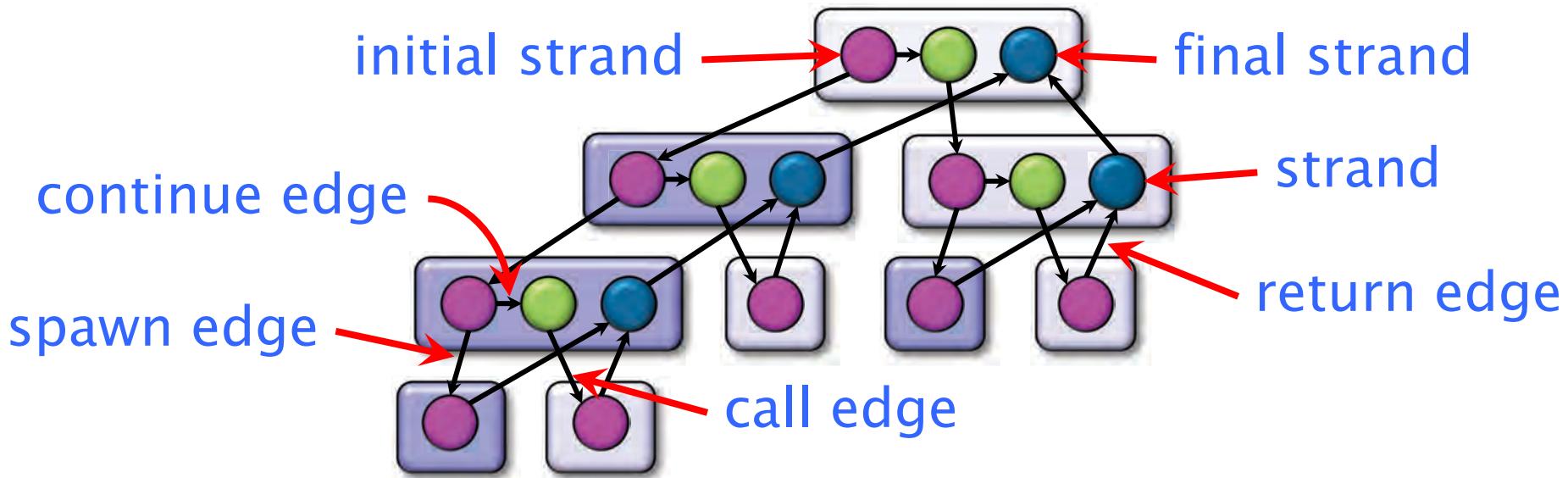
“Processor oblivious”

Example:  
 $\text{fib}(4)$



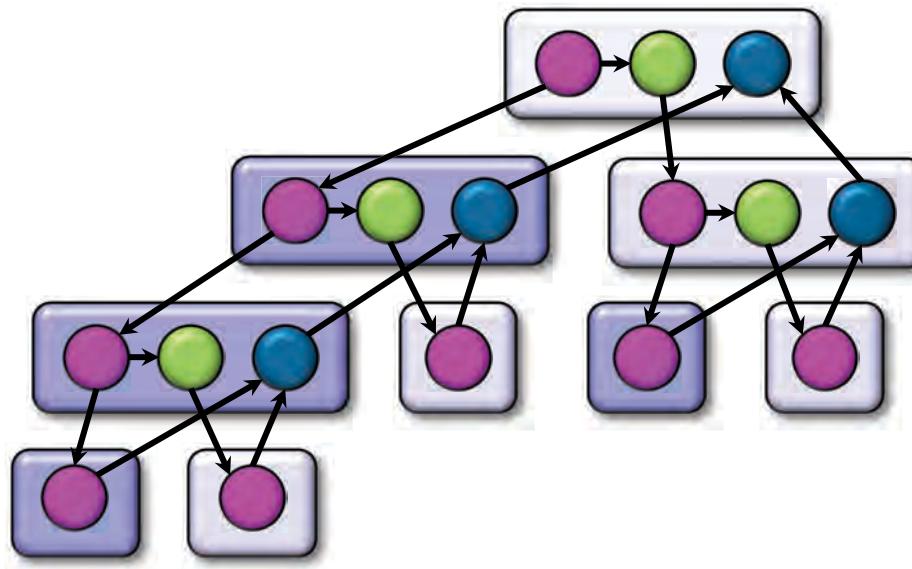
*The computation dag  
unfolds dynamically.*

# Computation Dag



- A parallel instruction stream is a dag  $G = (V, E)$ .
- Each vertex  $v \in V$  is a **strand**: a sequence of instructions not containing a spawn, sync, or return from a spawn.
- An edge  $e \in E$  is a **spawn**, **call**, **return**, or **continue** edge.
- Loop parallelism (`cilk_for`) is converted to spawns and syncs using recursive divide-and-conquer.

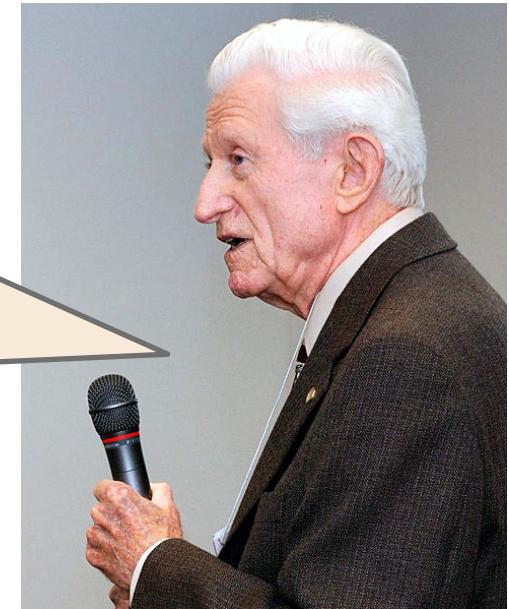
# How Much Parallelism?



Assuming that each strand executes in unit time,  
what is the **parallelism** of this computation?

# Amdahl's “Law”

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on.

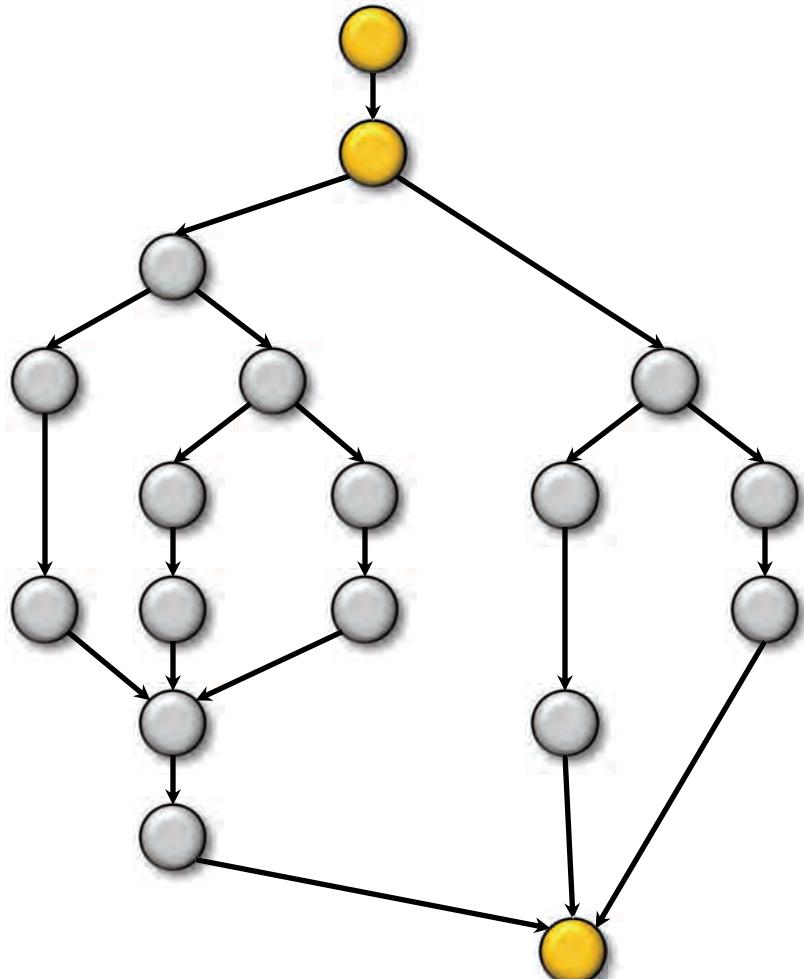


Gene M. Amdahl

In general, if a fraction  $\alpha$  of an application must be run serially, the speedup can be at most  $1/\alpha$ .

# Quantifying Parallelism

What is the **parallelism** of this computation?

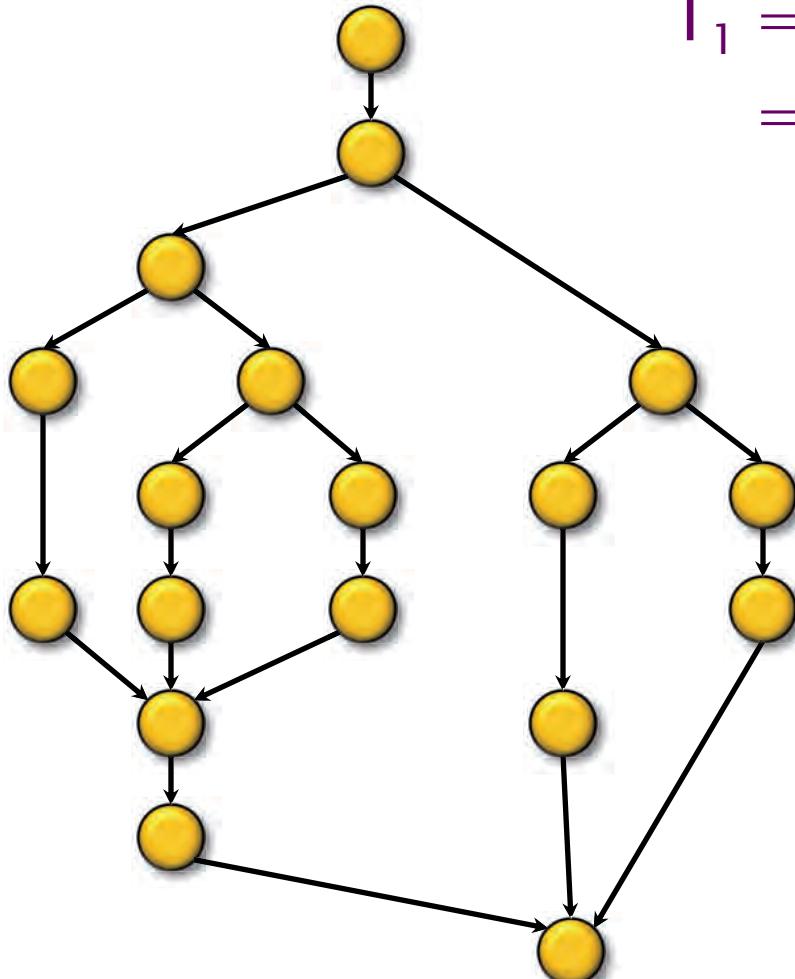


Amdahl's Law says that since the serial fraction is  $3/18 = 1/6$ , the speedup is upper-bounded by 6.

# Performance Measures

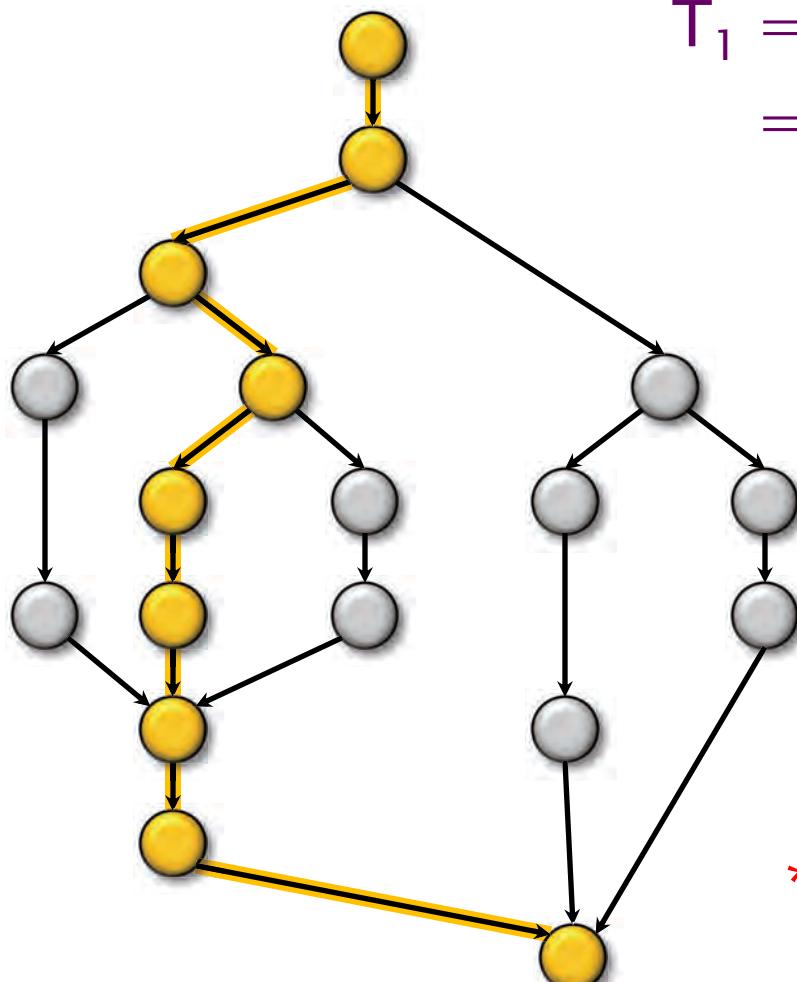
$T_P$  = execution time on  $P$  processors

$$T_1 = \text{work} \\ = 18$$



# Performance Measures

$T_p$  = execution time on  $P$  processors



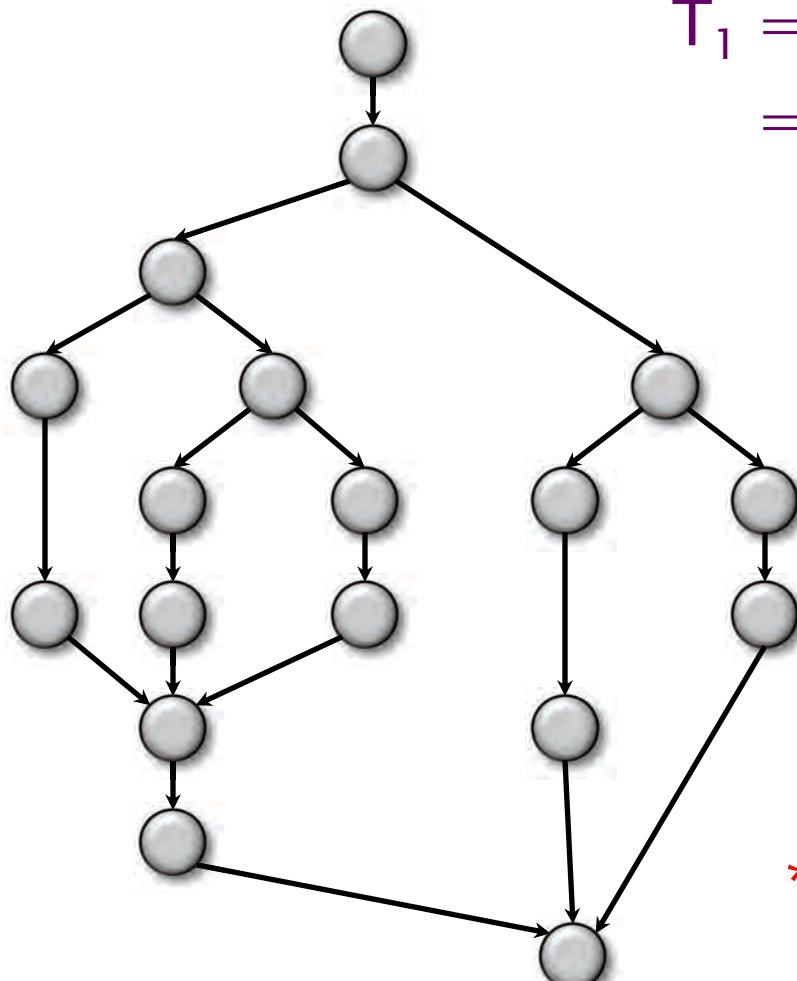
$$T_1 = \text{work} \\ = 18$$

$$T_\infty = \text{span}^* \\ = 9$$

\*Also called critical-path length or computational depth.

# Performance Measures

$T_P$  = execution time on  $P$  processors



$$T_1 = \text{work} \\ = 18$$

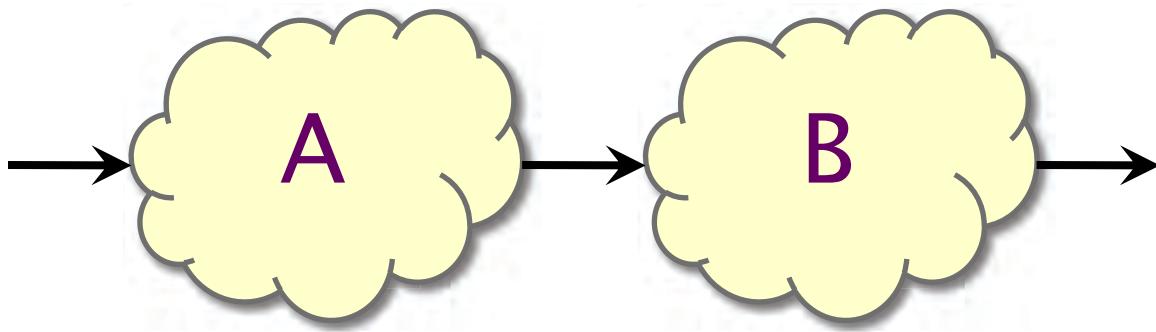
$$T_\infty = \text{span}^* \\ = 9$$

**WORK LAW**  
•  $T_P \geq T_1 / P$

**SPAN LAW**  
•  $T_P \geq T_\infty$

\*Also called critical-path length or computational depth.

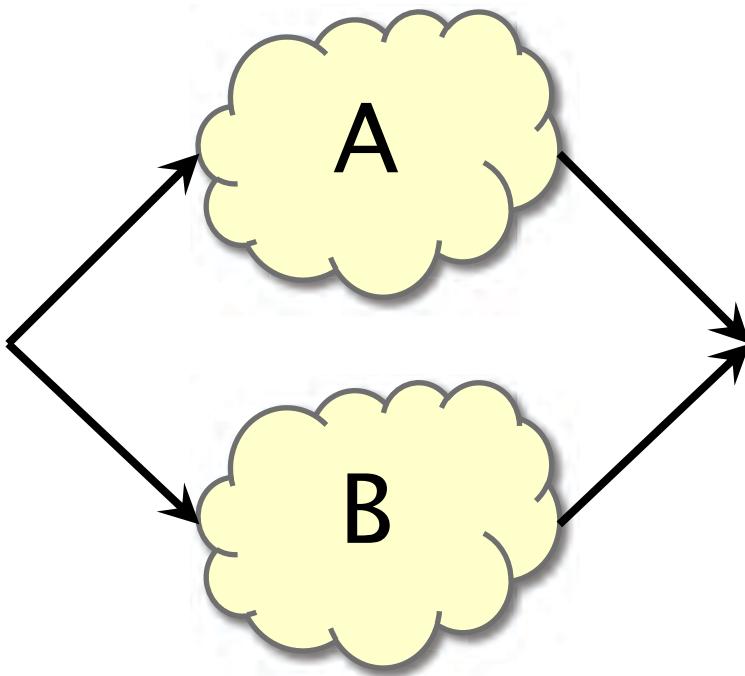
# Series Composition



*Work:*  $T_1(A \cup B) = T_1(A) + T_1(B)$

*Span:*  $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

# Parallel Composition



*Work:*  $T_1(A \cup B) = T_1(A) + T_1(B)$

*Span:*  $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$

# Speedup

Definition.  $T_1/T_P$  = speedup on  $P$  processors.

---

- If  $T_1/T_P < P$ , we have sublinear speedup.
  - If  $T_1/T_P = P$ , we have (perfect) linear speedup.
  - If  $T_1/T_P > P$ , we have superlinear speedup,  
which is not possible in this simple  
performance model, because of the WORK LAW  
 $T_P \geq T_1/P$ .
-

# Parallelism

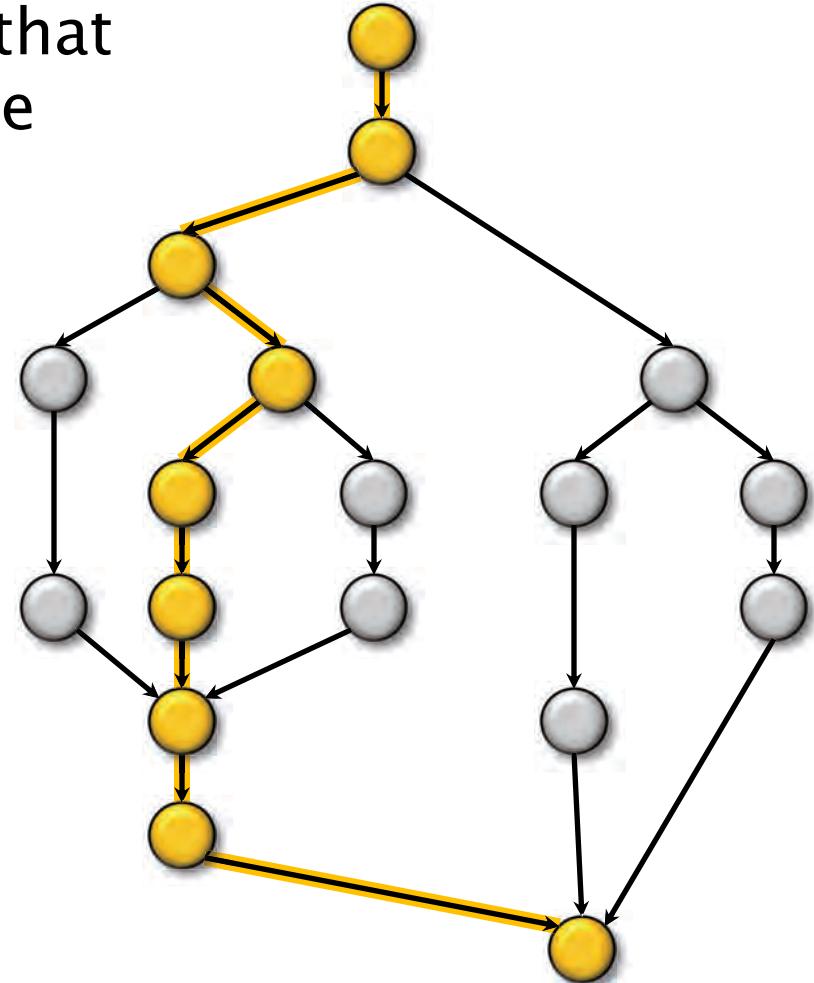
Because the **SPAN LAW** dictates that  $T_P \geq T_\infty$ , the maximum possible speedup given  $T_1$  and  $T_\infty$  is

$T_1/T_\infty = \text{parallelism}$

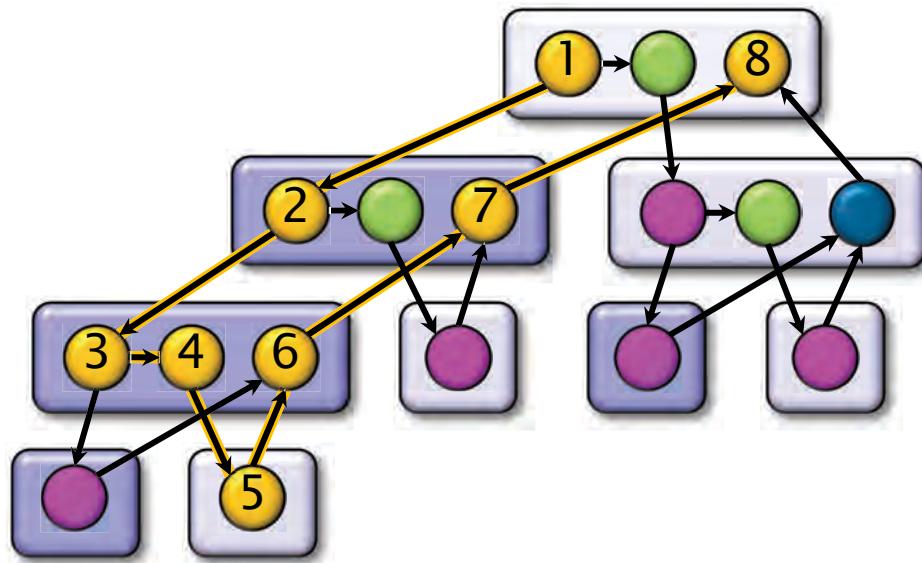
= the average amount of work per step along the span

=  $18/9$

= 2.



# Example: fib(4)



Assume for simplicity that each strand in **fib(4)** takes unit time to execute.

*Work:*  $T_1 = 17$

*Span:*  $T_\infty = 8$

*Parallelism:*  $T_1/T_\infty = 2.125$

Using many more than 2 processors can yield only marginal performance gains.

# THE CILKSCALE SCALABILITY ANALYZER



# Cilkscale Scalability Analyzer

- The Tapir/LLVM compiler provides a **scalability analyzer** called **Cilkscale**.
- Like the Cilksan race detector, Cilkscale uses **compiler-instrumentation** to analyze a serial execution of a program.
- Cilkscale computes **work** and **span** to derive upper bounds on parallel performance.

# Quicksort Analysis

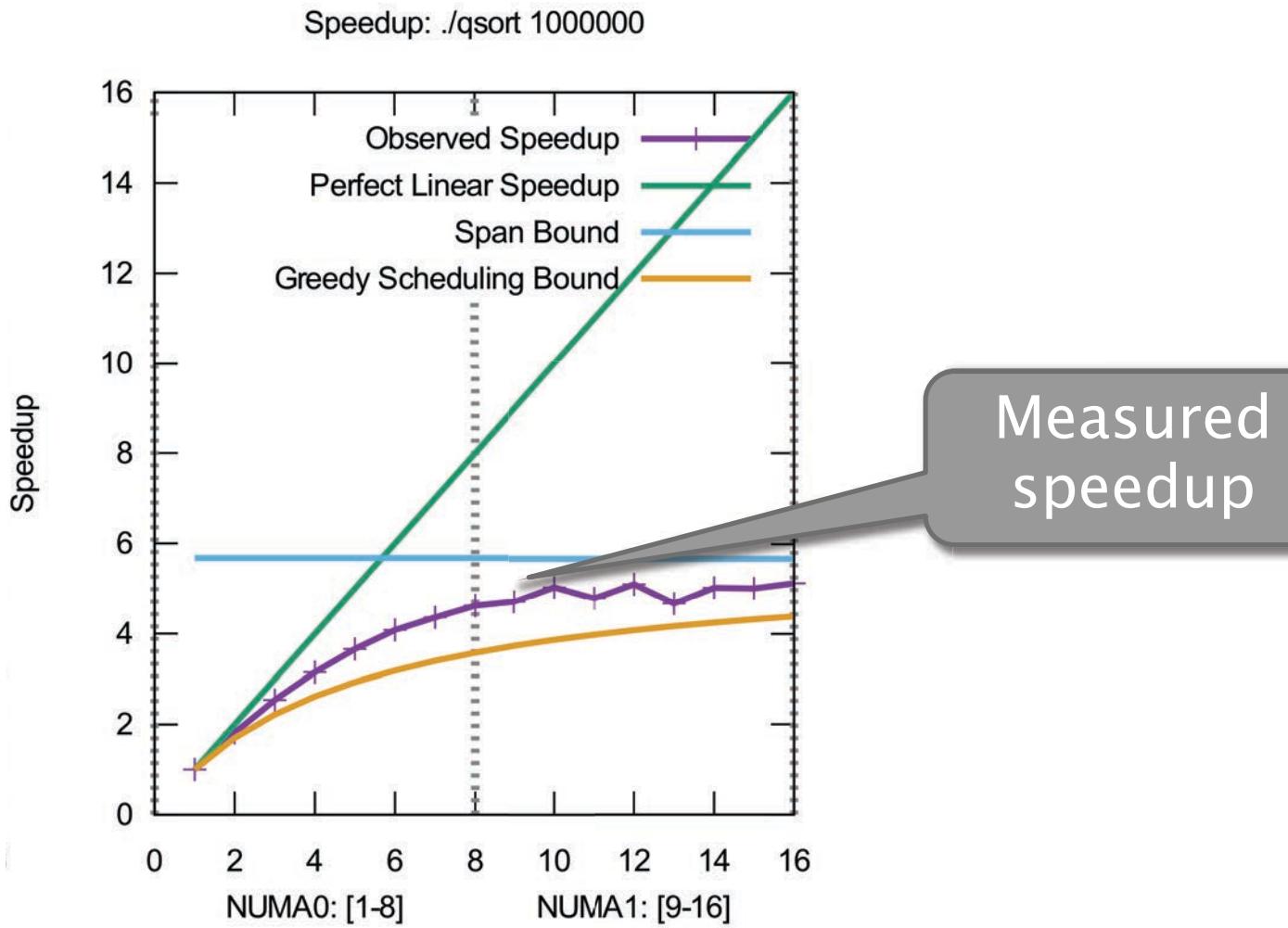
## Example: Parallel quicksort

```
static void quicksort(size_t *left, size_t *right)
{
    if (left == right) return;
    size_t *p = partition(left, right); //run serially
    cilk_spawn quicksort(left, p);
    quicksort(p + 1, right);
    cilk_sync;
}
```

Analyze the sorting of 1,000,000 numbers.

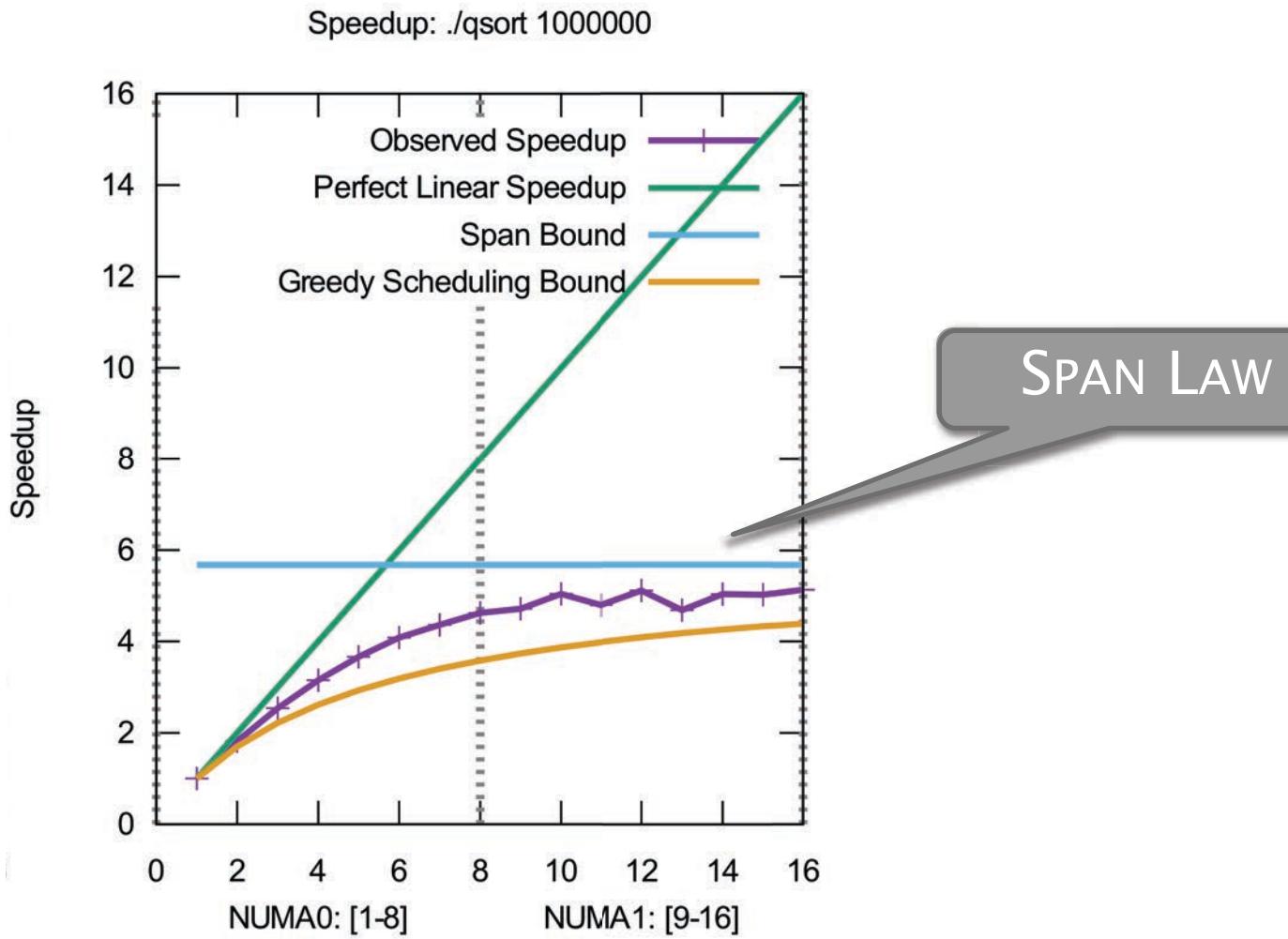
★★★ *Guess the parallelism!* ★★★

# Cilkscale Output

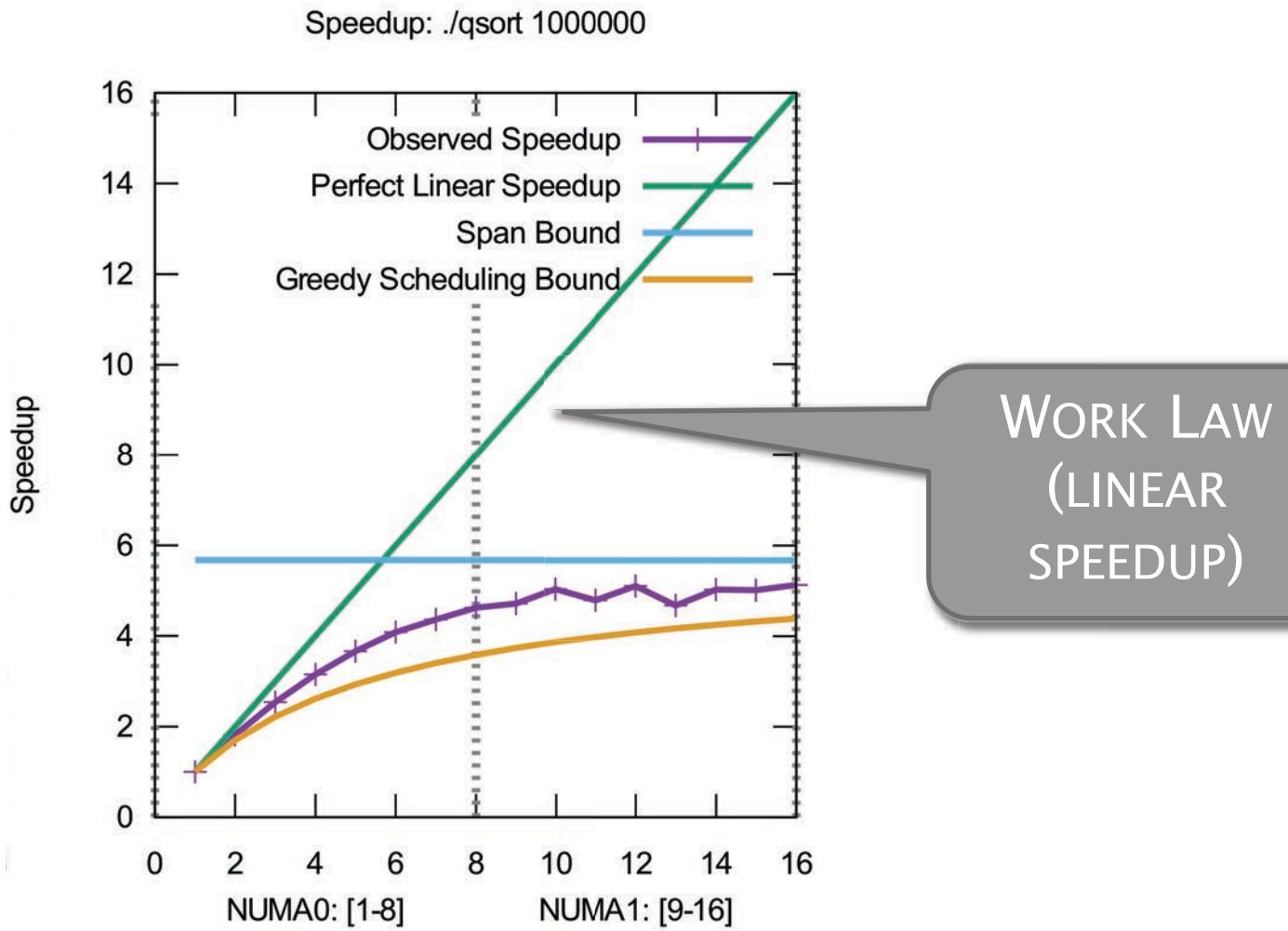


Measured  
speedup

# Cilkscale Output

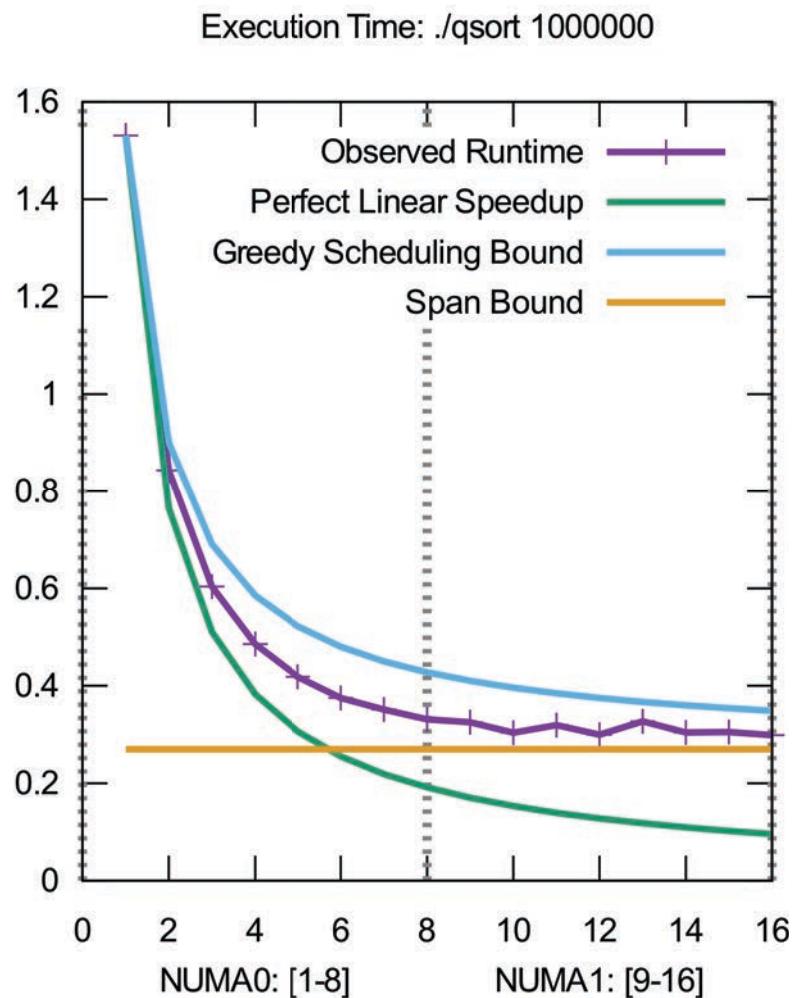


# Cilkscale Output



WORK LAW  
(LINEAR  
SPEEDUP)

# Cilkscale Output



# Theoretical Analysis

## Example: Parallel quicksort

```
static void quicksort(size_t *left, size_t *right)
{
    if (left == right) return;
    size_t *p = partition(left, right); //run serially
    cilk_spawn quicksort(left, p);
    quicksort(p + 1, right);
    cilk_sync;
}
```

Expected work =  $\Theta(n \lg n)$   
Expected span =  $\Theta(n)$



Parallelism =  $\Theta(\lg n)$

# Interesting Practical\* Algorithms

Algorithm	Work	Span	Parallelism
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$	$\Theta(n / \lg^2 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$	$\Theta(n^3 / \lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$	$\Theta(n^{\lg 7} / \lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$	$\Theta(n^2 / \lg n)$
Tableau construction	$\Theta(n^2)$	$\Theta(n^{\lg 3})$	$\Theta(n^{2-\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n / \lg n)$
Breadth-first search	$\Theta(E)$	$\Theta(\Delta \lg V)$	$\Theta(E / \Delta \lg V)$

---

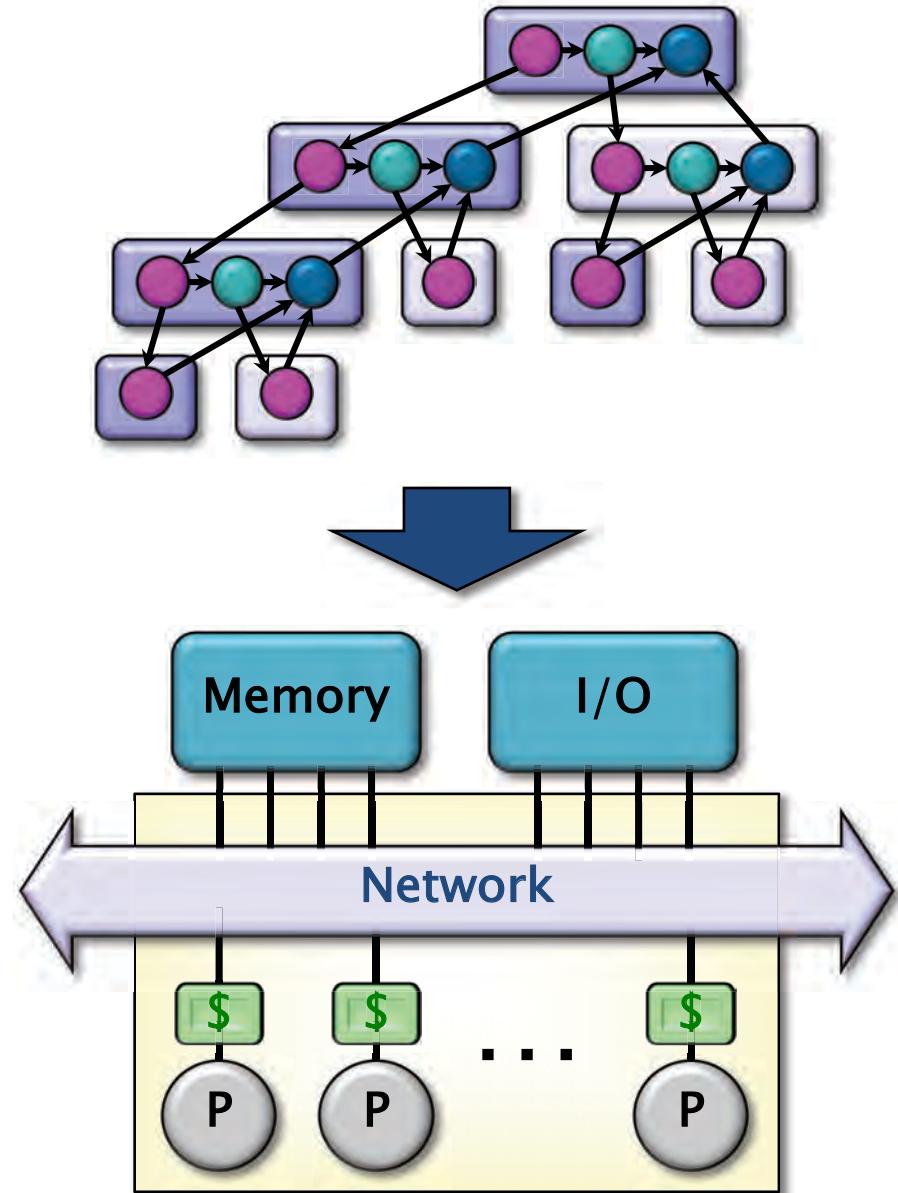
\*Cilk on 1 processor competitive with the best C.

# SCHEDULING THEORY



# Scheduling

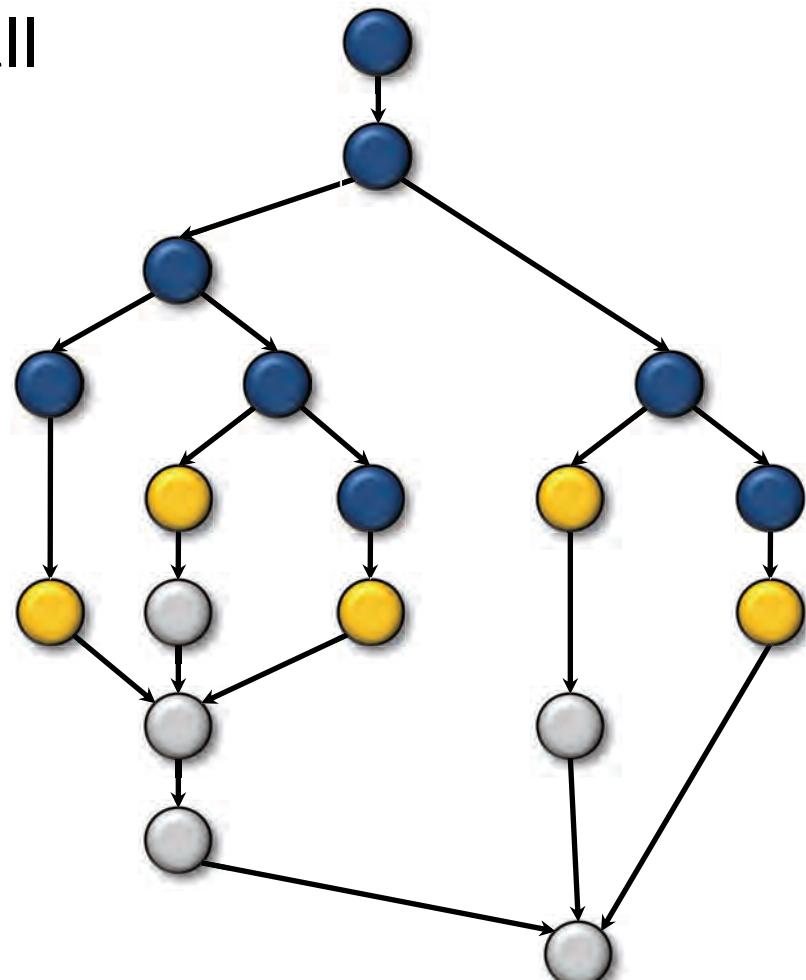
- Cilk allows the programmer to express **potential parallelism** in an application.
- The Cilk **scheduler** maps strands onto processors dynamically at runtime.
- Since the theory of **distributed** schedulers is complicated, we'll explore the ideas with a **centralized** scheduler.



# Greedy Scheduling

IDEA: Do as much as possible on every step.

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



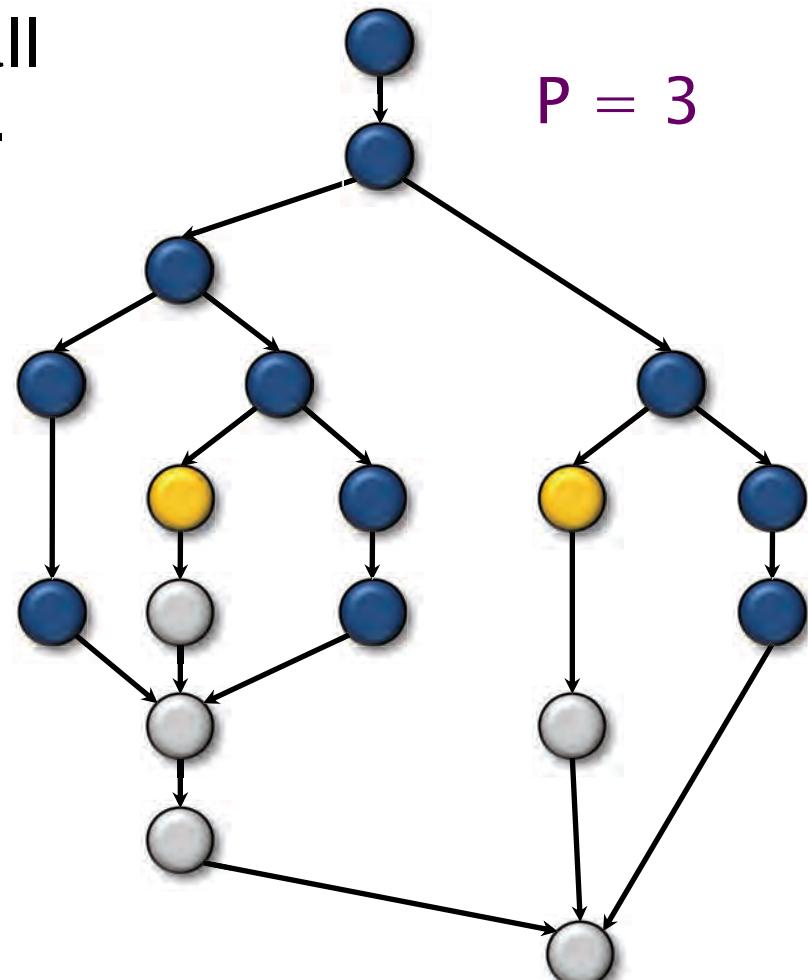
# Greedy Scheduling

IDEA: Do as much as possible on every step.

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

Complete step

- $\geq P$  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.

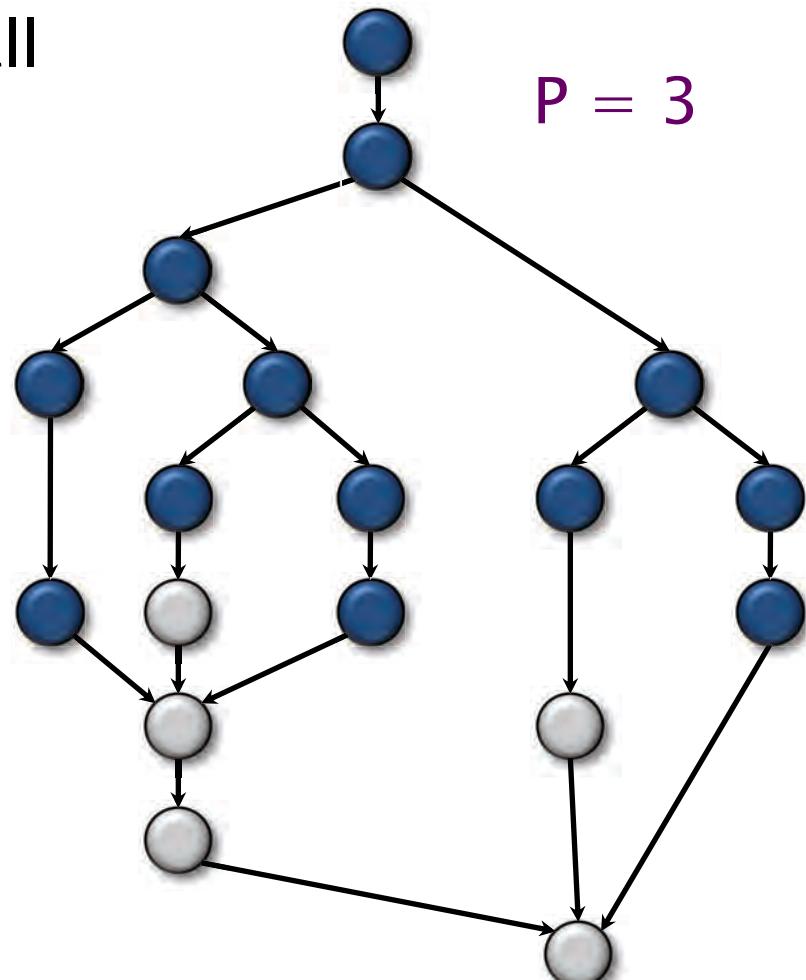
# Complete step

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

# Incomplete step

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

P = 3



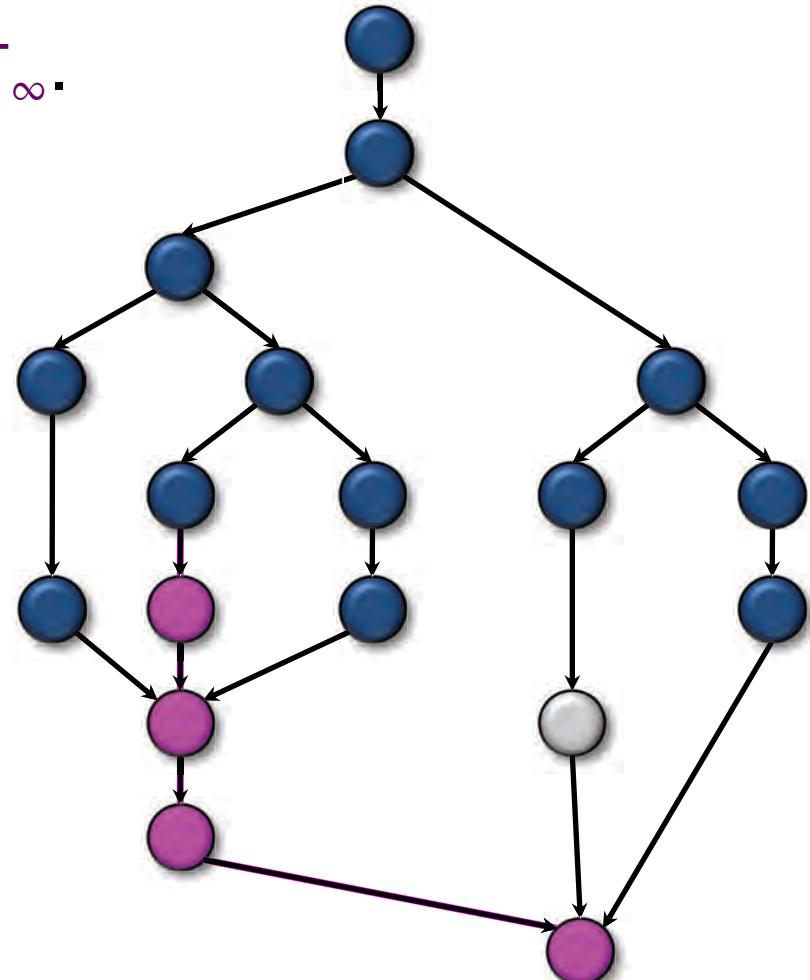
# Analysis of Greedy

Theorem [G68, B75, EZL89]. Any greedy scheduler achieves

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

*Proof.*

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



# Optimality of Greedy

**Corollary.** 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

**Corollary.** Any greedy scheduler achieves near-perfect linear speedup whenever  $T_1/T_\infty \gg P$ .

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

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

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

**Definition.** The quantity  $T_1/(PT_\infty)$  is called the parallel slackness.

# Cilk Performance

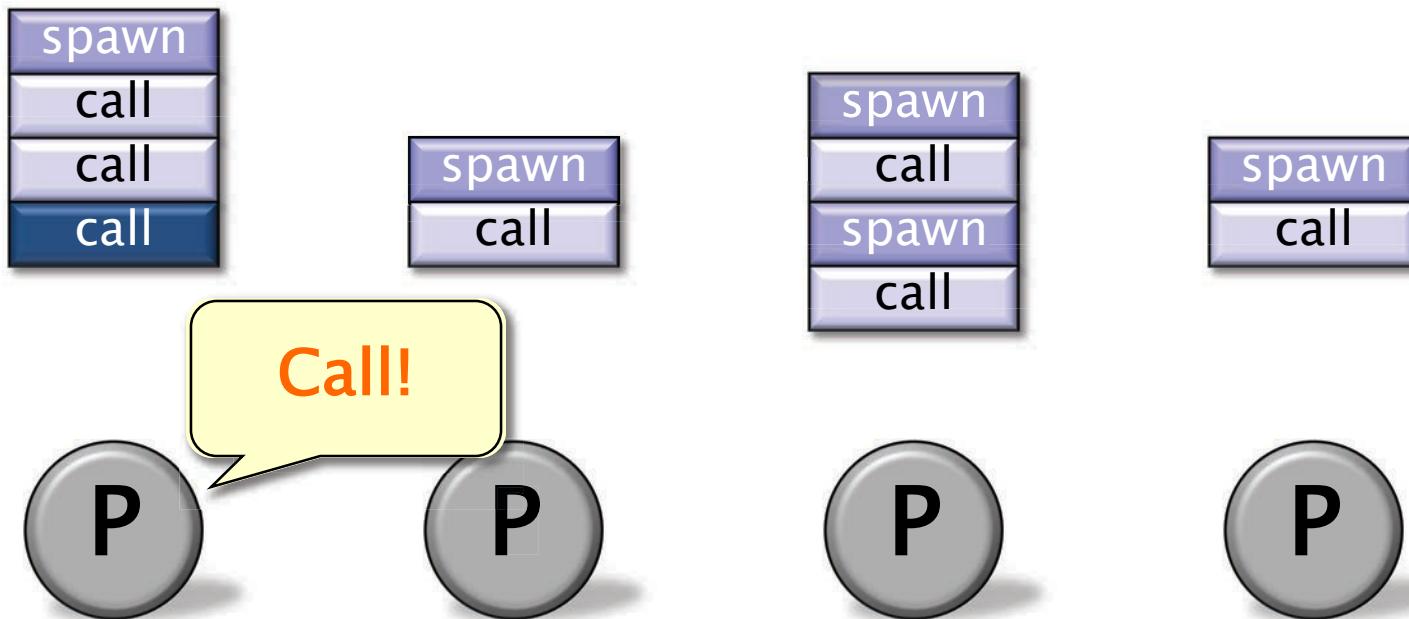
- Cilk's work-stealing scheduler achieves
  - $T_P = T_1/P + O(T_\infty)$  expected time (provably);
  - $T_P \approx T_1/P + T_\infty$  time (empirically).
- Near-perfect linear speedup as long as  $P \ll T_1/T_\infty$ .
- Instrumentation in Cilkscale allows you to measure  $T_1$  and  $T_\infty$ .

# THE CILK RUNTIME SYSTEM



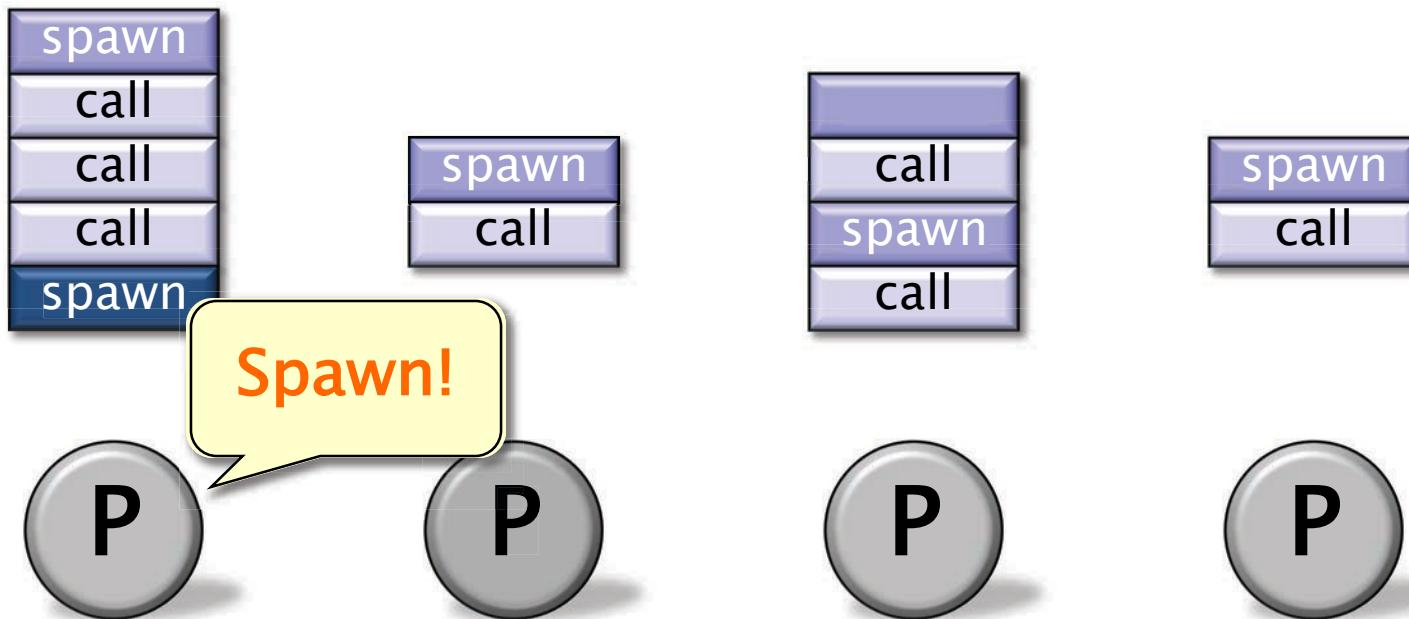
# Cilk Runtime System

Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



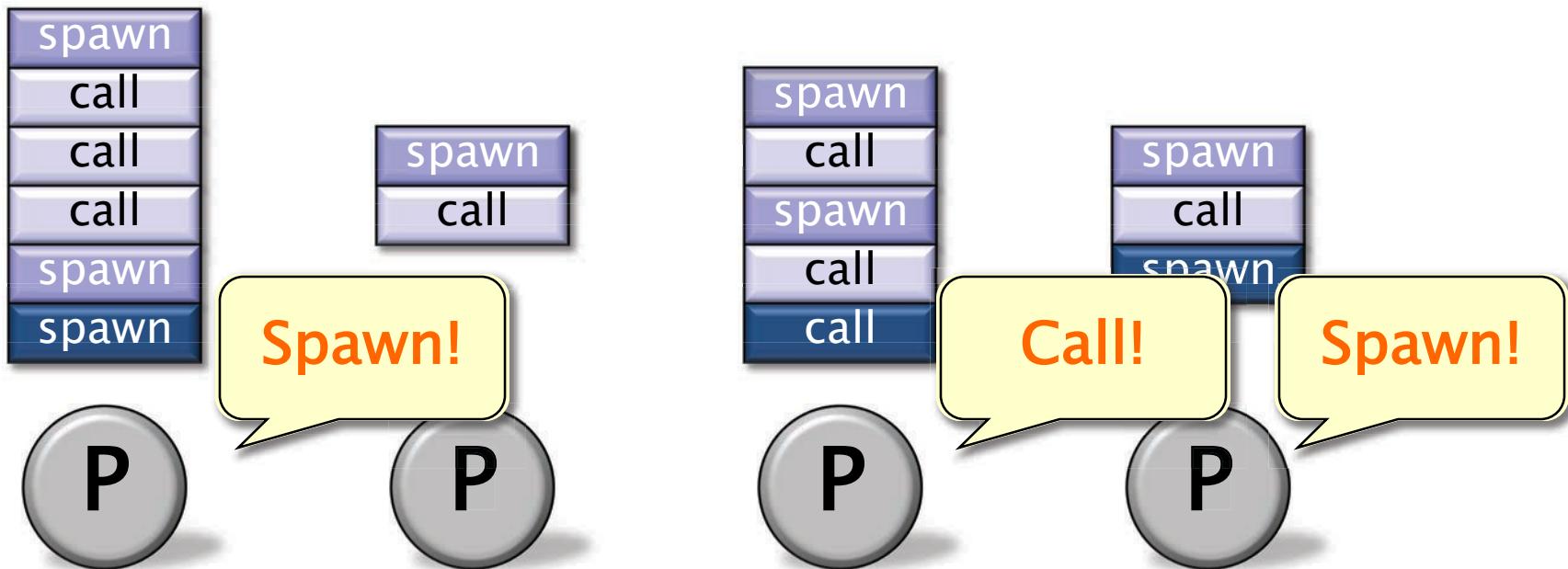
# Cilk Runtime System

Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



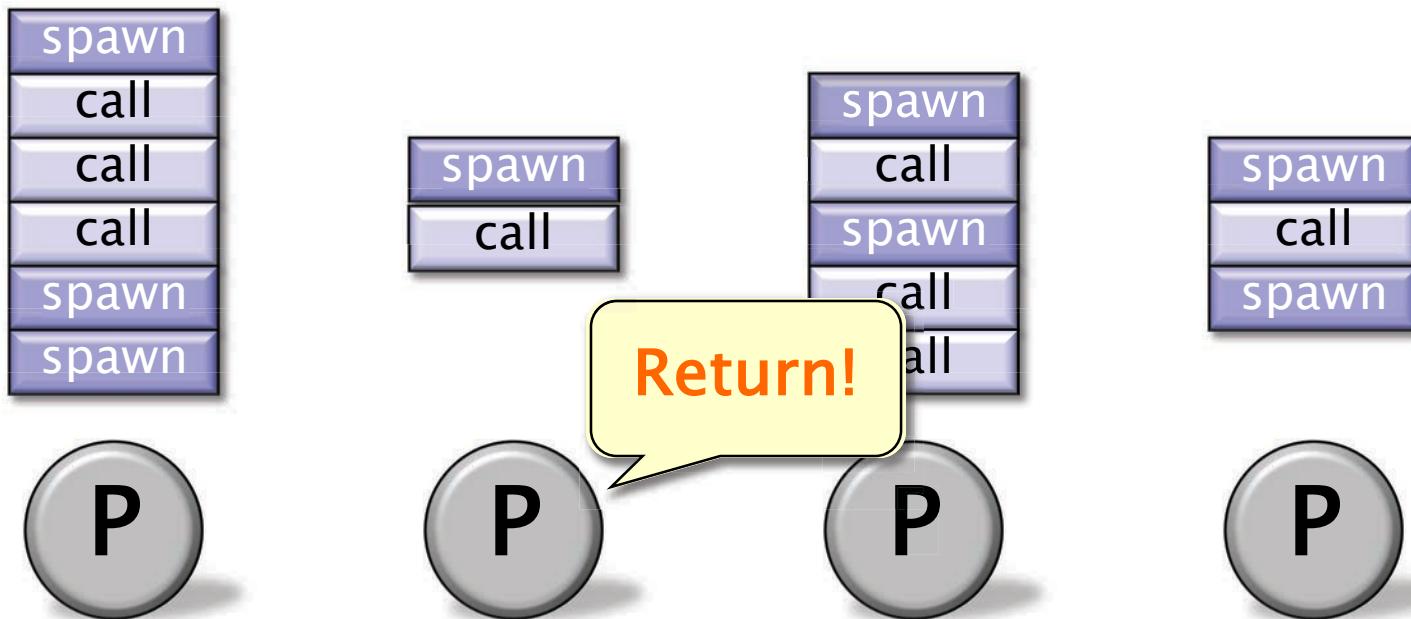
# Cilk Runtime System

Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



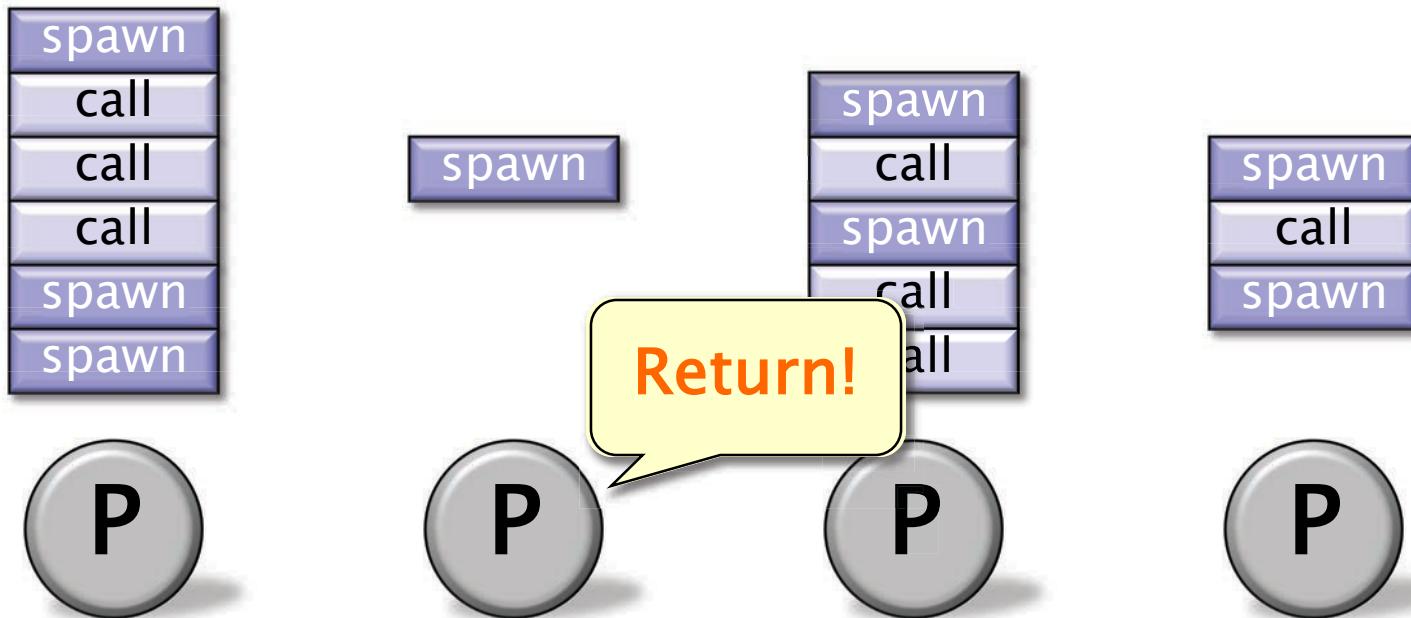
# Cilk Runtime System

Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



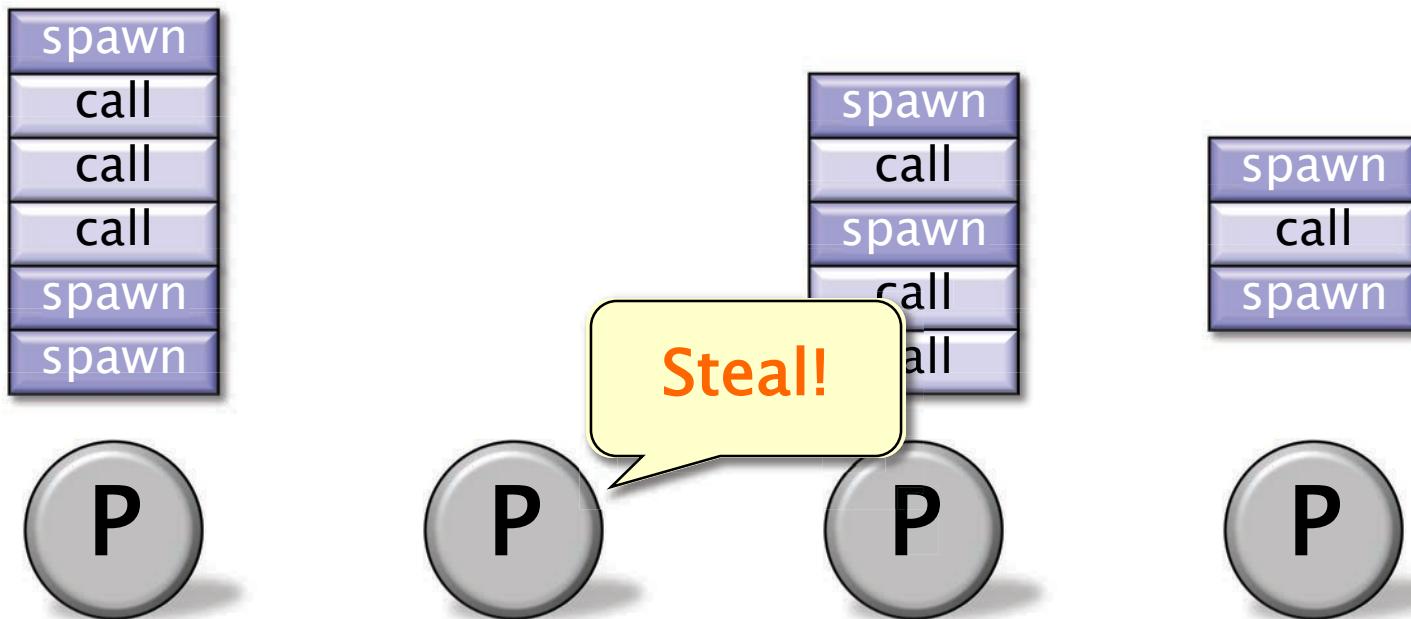
# Cilk Runtime System

Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



# Cilk Runtime System

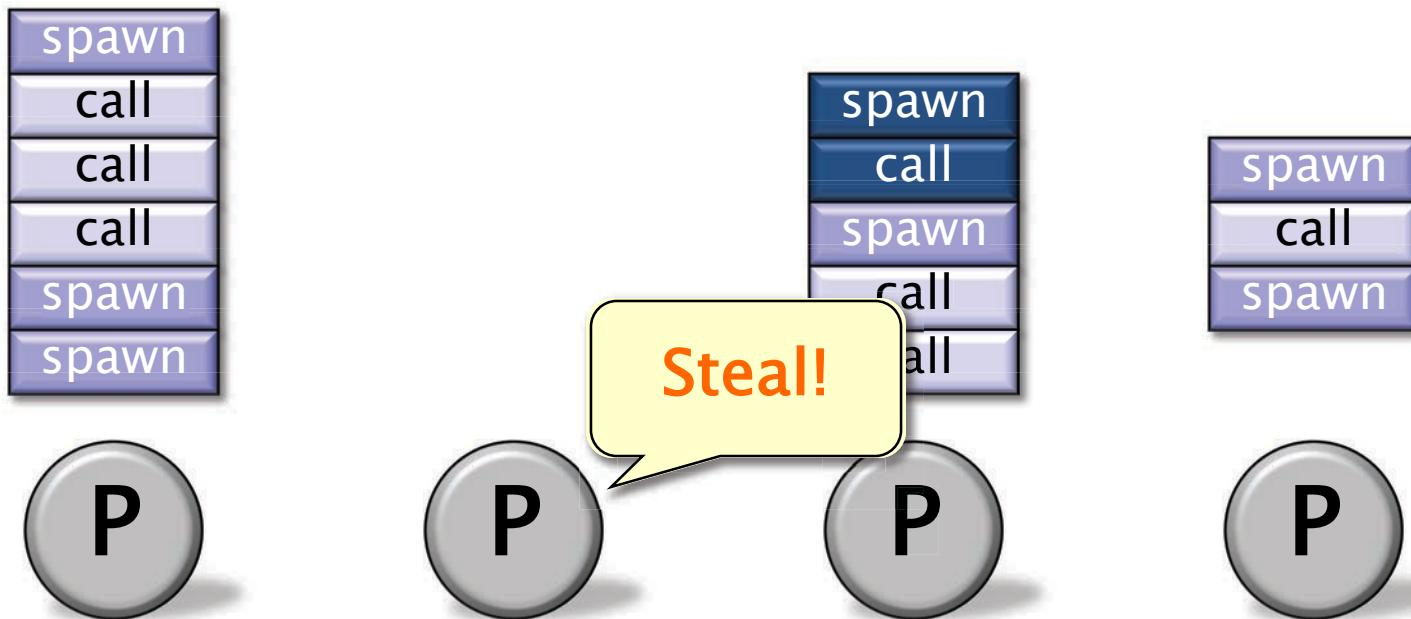
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

# Cilk Runtime System

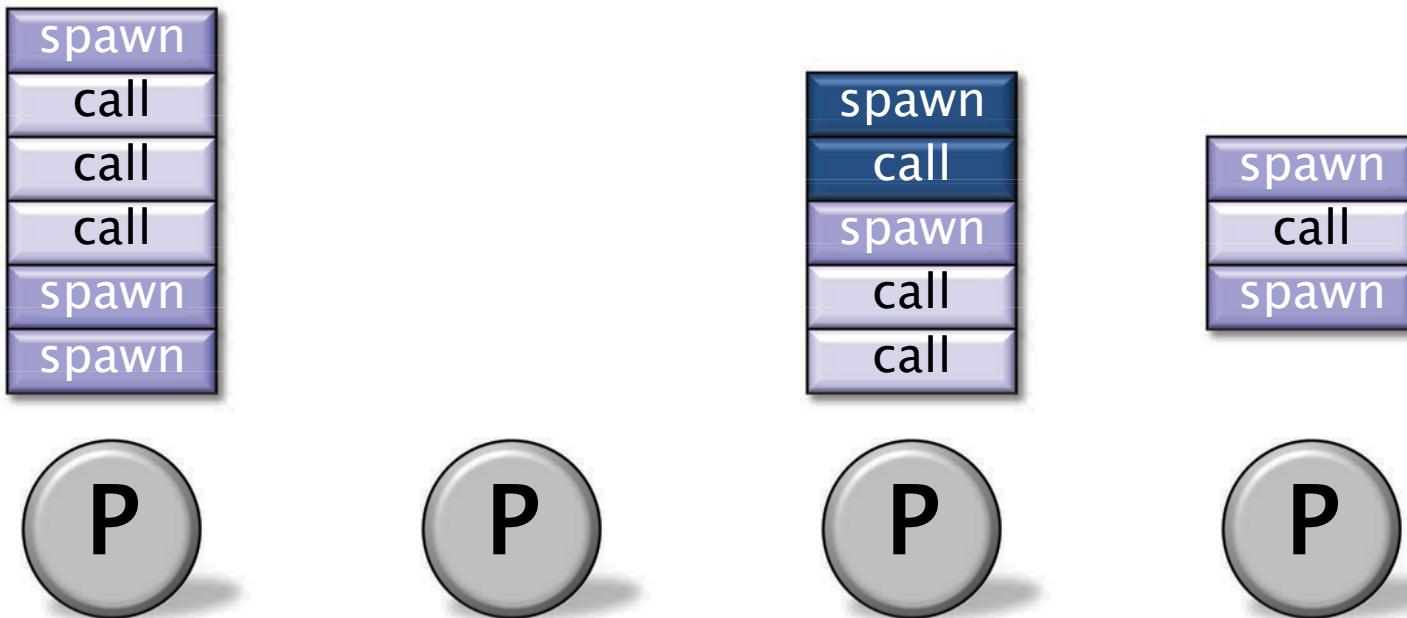
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

# Cilk Runtime System

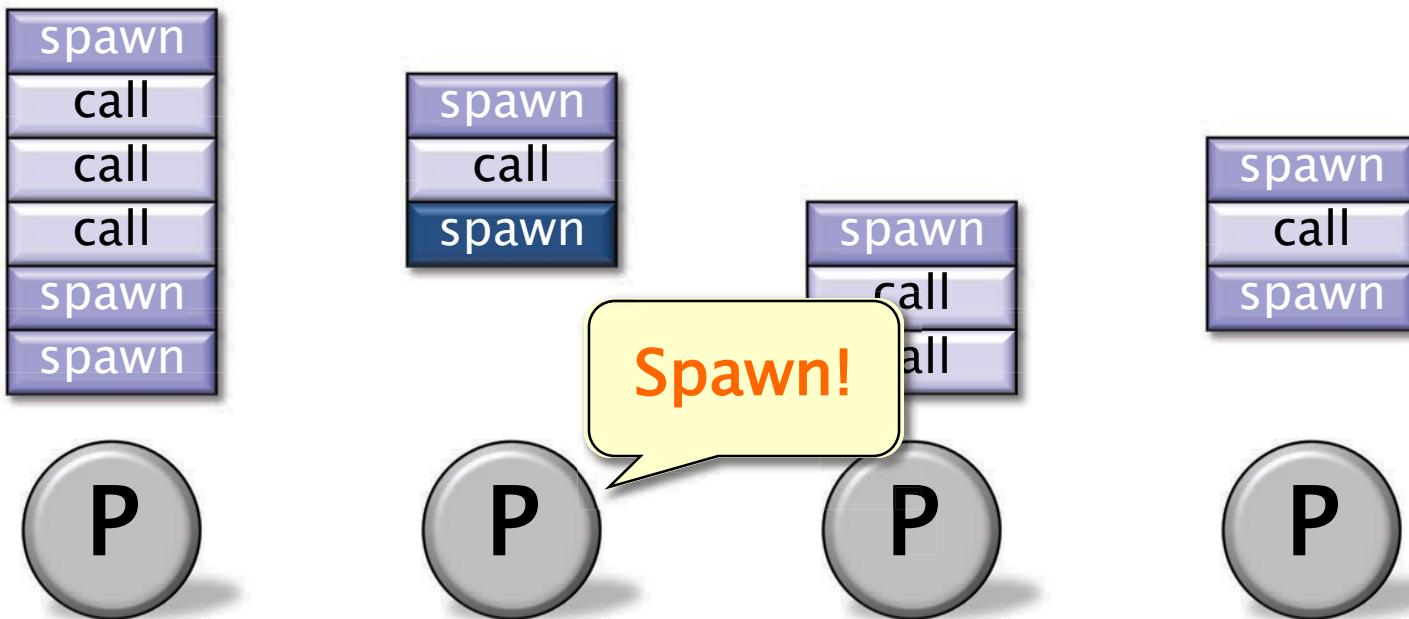
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

# Cilk Runtime System

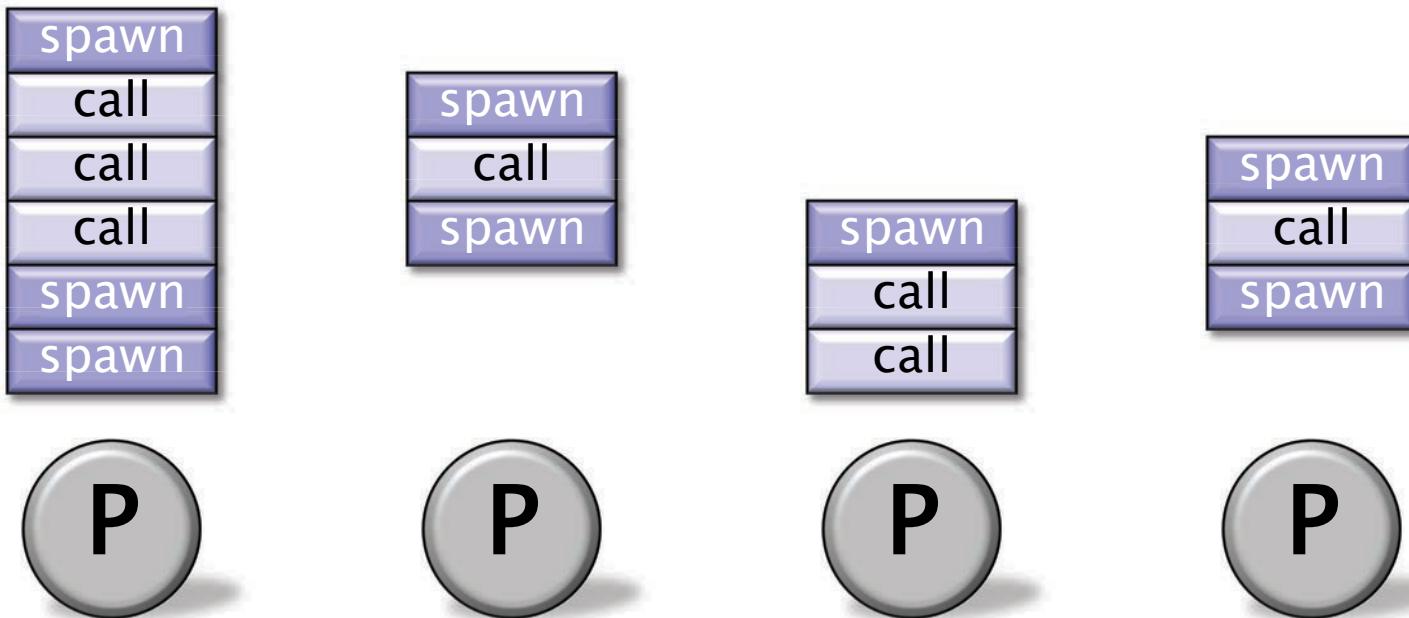
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

# Cilk Runtime System

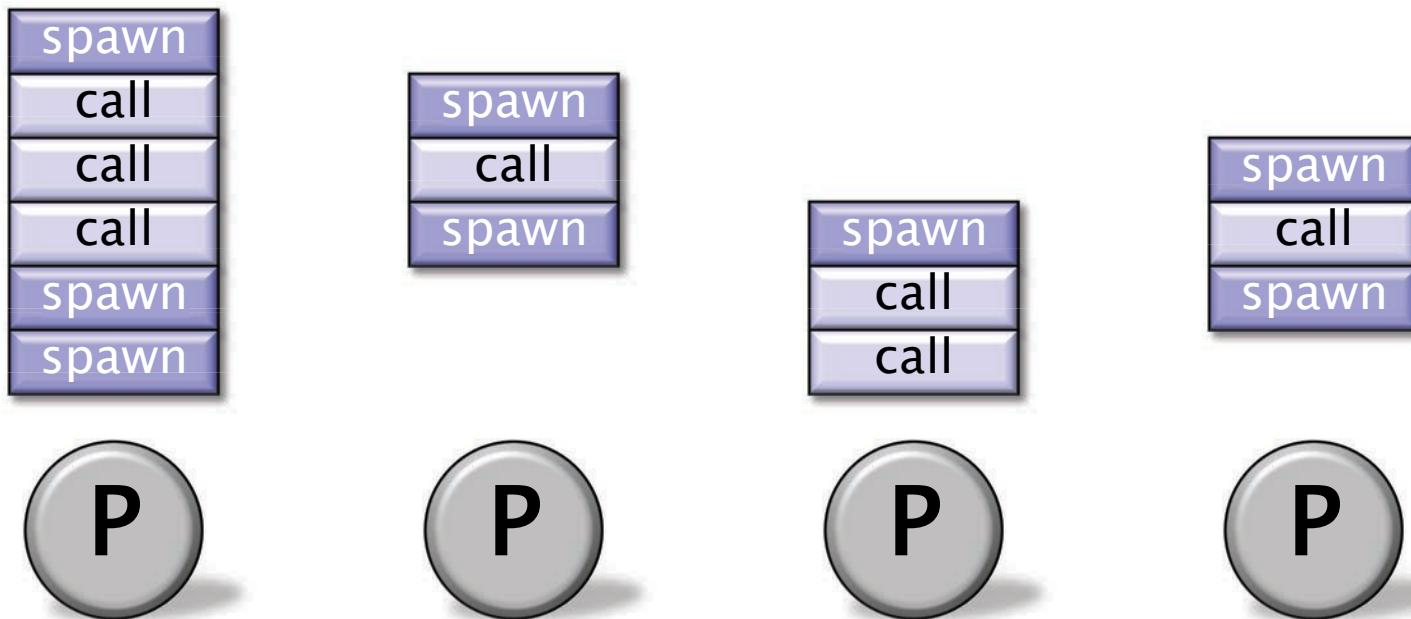
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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 [MKH90, BL94, FLR98].



**Theorem** [BL94]: With sufficient parallelism, workers steal infrequently  $\Rightarrow$  linear speed-up.

# Work–Stealing Bounds

**Theorem** [BL94]. The Cilk work-stealing scheduler achieves expected running time

$$T_P \approx T_1/P + O(T_\infty)$$

on  $P$  processors.

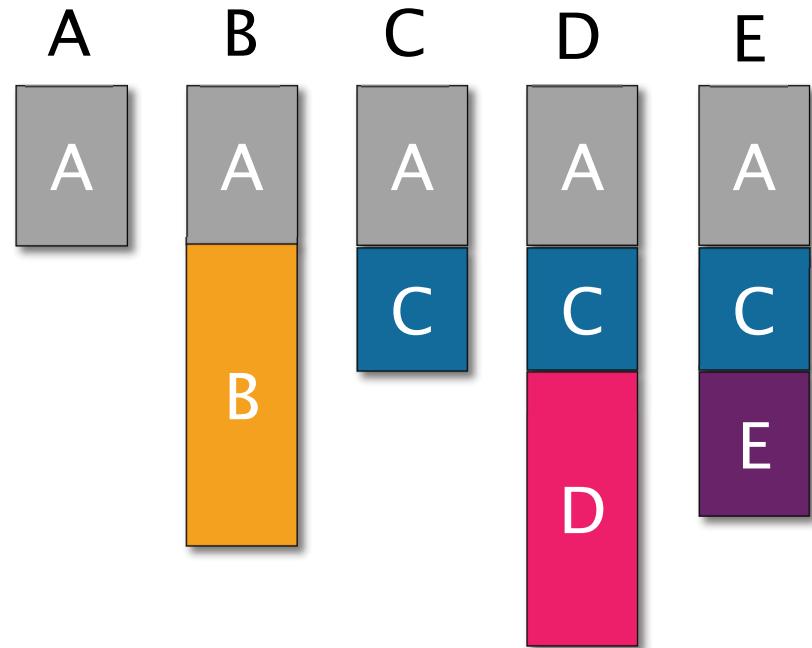
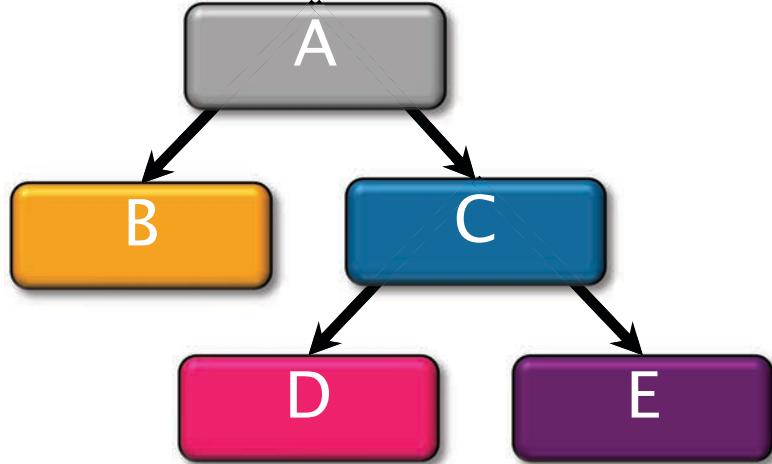
*Pseudoproof.* A processor is either **working** or **stealing**. The total time all processors spend working is  $T_1$ . Each steal has a  $1/P$  chance of reducing the span by 1. Thus, the expected cost of all steals is  $O(PT_\infty)$ . Since there are  $P$  processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty) . \blacksquare$$

# Cactus Stack

Cilk supports C's **rule for pointers**: A pointer to stack space can be passed from parent to child, but not from child to parent.

*Views of stack*

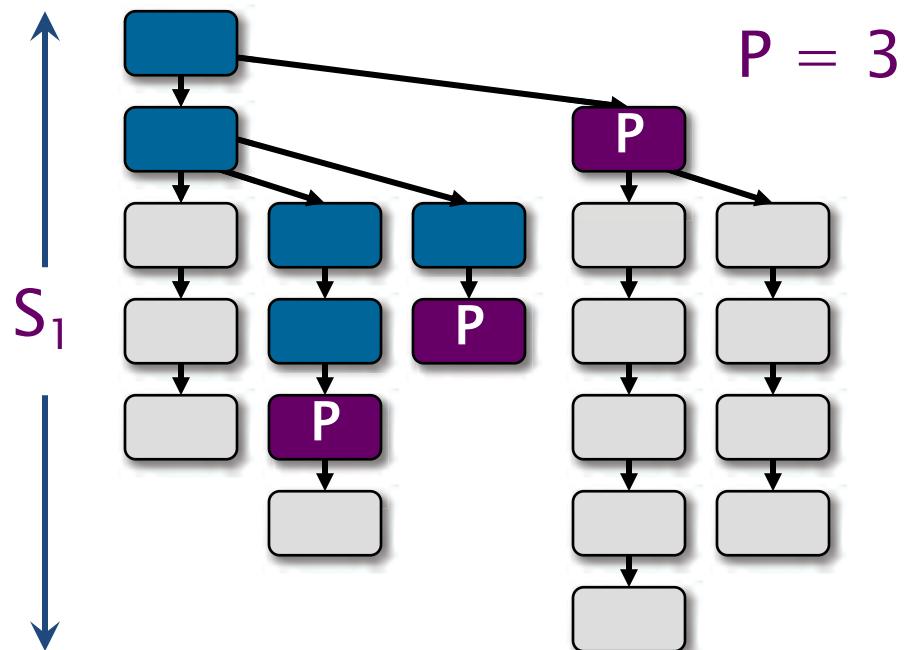


Cilk's **cactus stack** supports multiple views in parallel.

# Bound on Stack Space

**Theorem.** Let  $S_1$  be the stack space required by a serial execution of a Cilk program. Then the stack space required by a  $P$ -processor execution is at most  $S_P \leq PS_1$ .

**Proof** (by induction).  
The work-stealing algorithm maintains the **busy-leaves property**:  
Every extant leaf activation frame has a worker executing it. ■



# Summary

- Determinacy races are often bugs, and they can be detected using Cilksan
- Cilkscale can analyze the work, span, and parallelism of a computation
- A greedy scheduler is within a factor of 2 of the optimal scheduler
- Cilk uses a work-stealing scheduler with strong theoretical bounds on its running time