6.172
Performance
Engineering of
Software Systems

LECTURE 13
Parallelism and Performance

Charles E. Leiserson

October 26, 2010

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.*

Photograph of Gene Amdahl removed due to copyright restrictions.

*In general, if a fraction α of an application can be run in parallel and the rest must run serially, the speedup is at most $1/(1-\alpha)$.

But whose application can be decomposed into just a serial part and a parallel part? For *my* application, what speedup should I expect?

OUTLINE

- What Is Parallelism?
- Scheduling Theory
- Cilk++ Runtime System
- A Chess Lesson

OUTLINE

- What Is Parallelism?
- Scheduling Theory
- Cilk++ Runtime System
- A Chess Lesson

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.

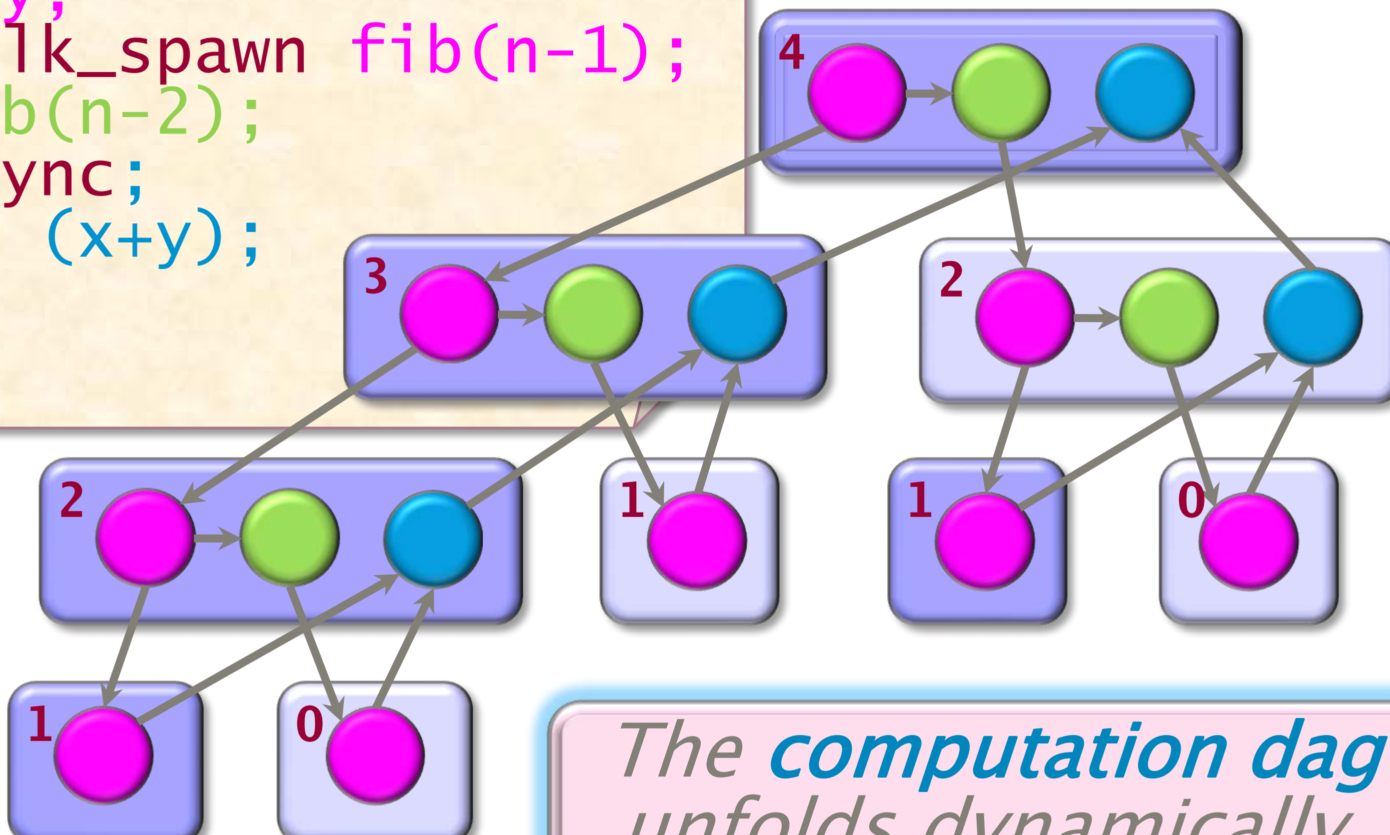
Cilk++ keywords *grant permission* for parallel execution. They do not *command* parallel execution.

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:

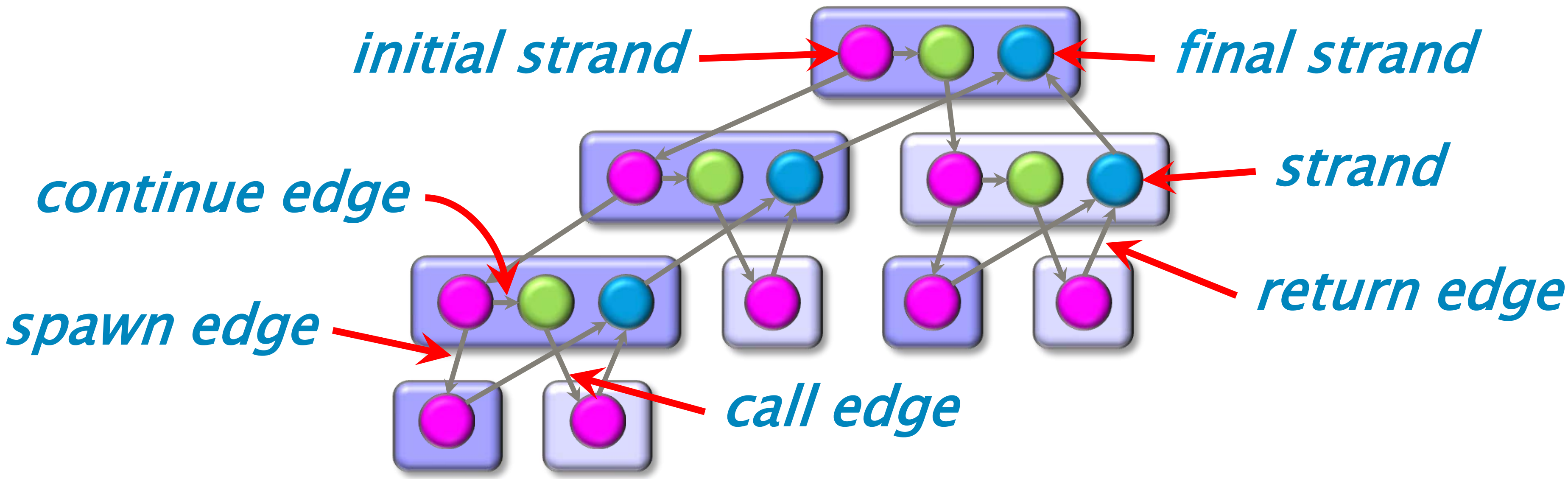
`fib(4)`



"Processor oblivious"

The ***computation dag*** *unfolds dynamically.*

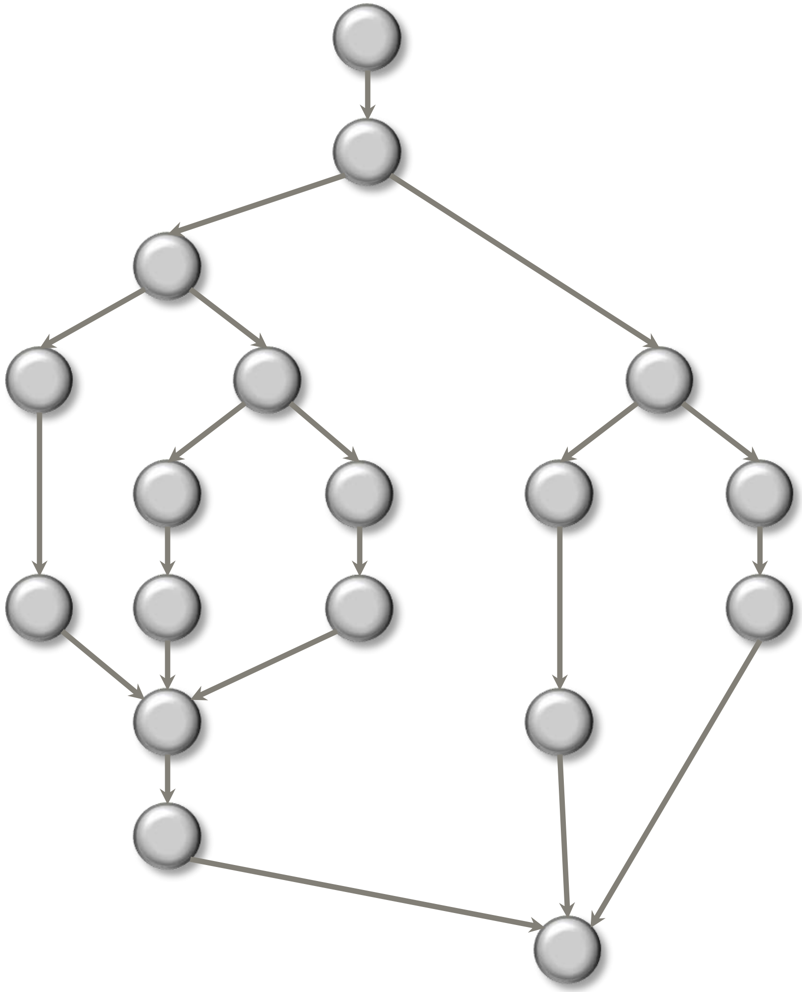
Computation Dag



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

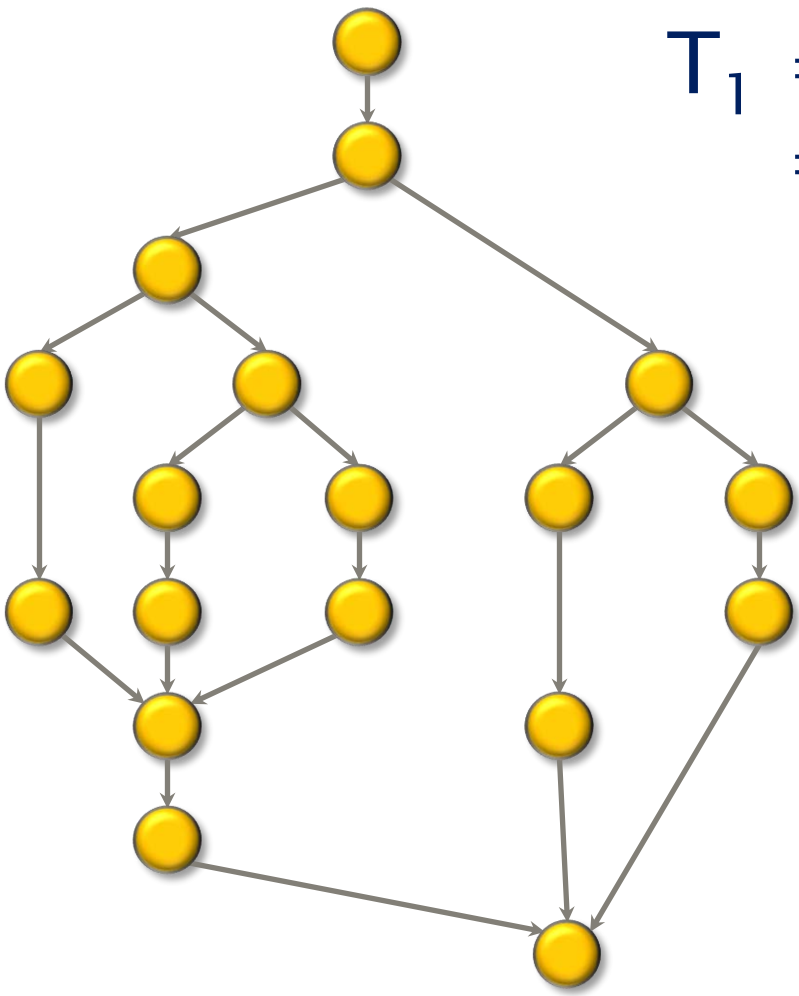
Performance Measures

T_P = execution time on P processors



Performance Measures

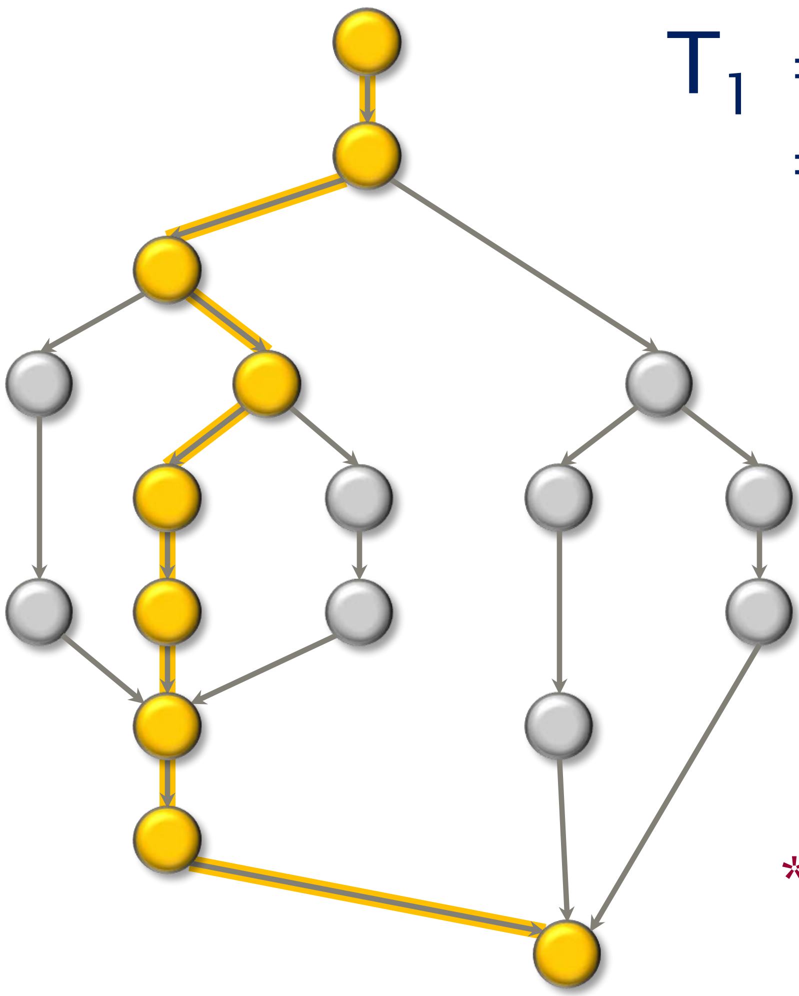
T_P = execution time on P processors



T_1 = ***work***
$= 18$

Performance Measures

T_P = execution time on P processors



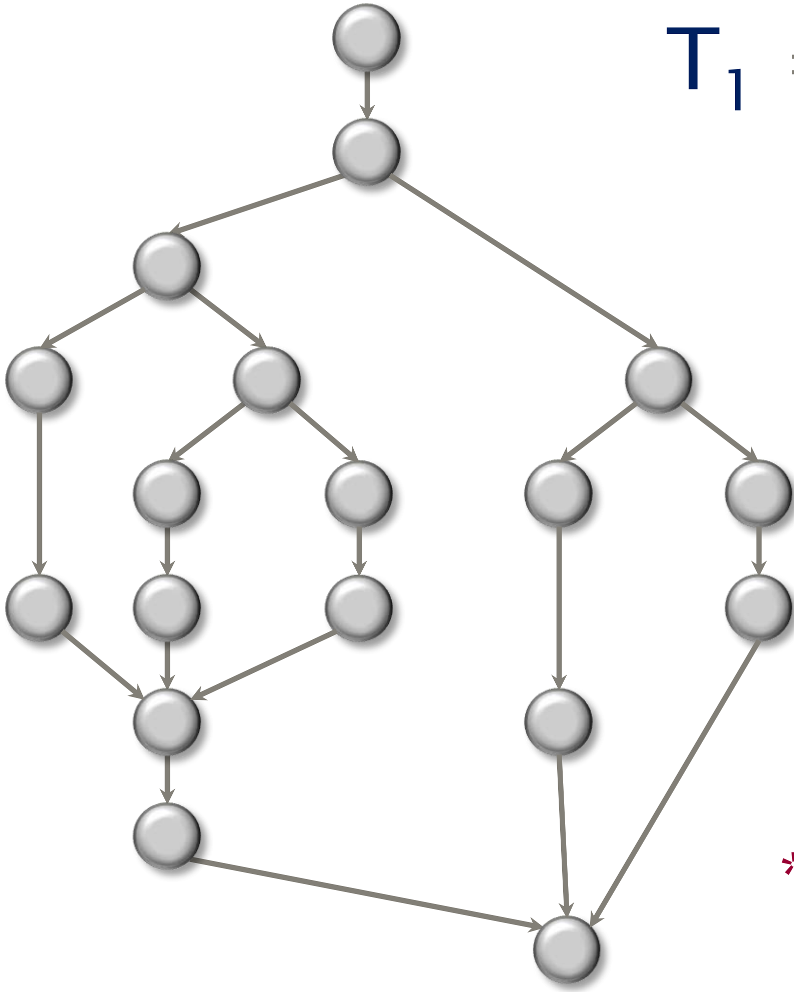
T_1 = ***work*** $= 18$

T_∞ = ***span**** $= 9$

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

Performance Measures

T_P = execution time on P processors



T_1 = ***work*** T_∞ = ***span****

WORK LAW

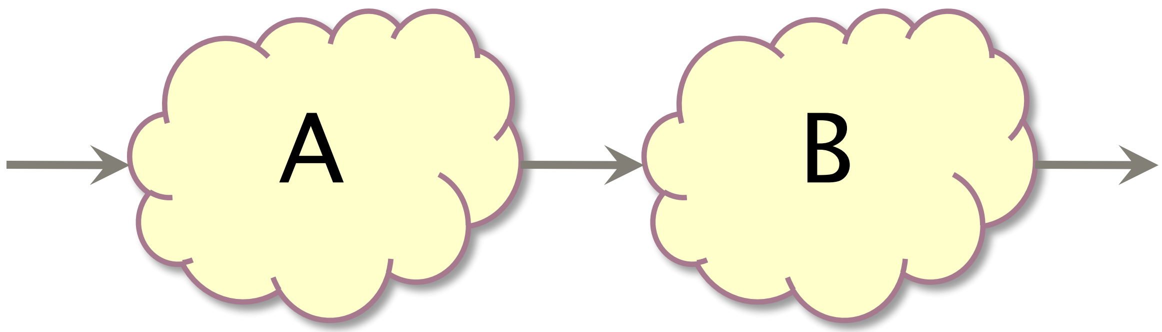
- $T_P \geq T_1/P$

SPAN LAW

- $T_P \geq T_\infty$

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

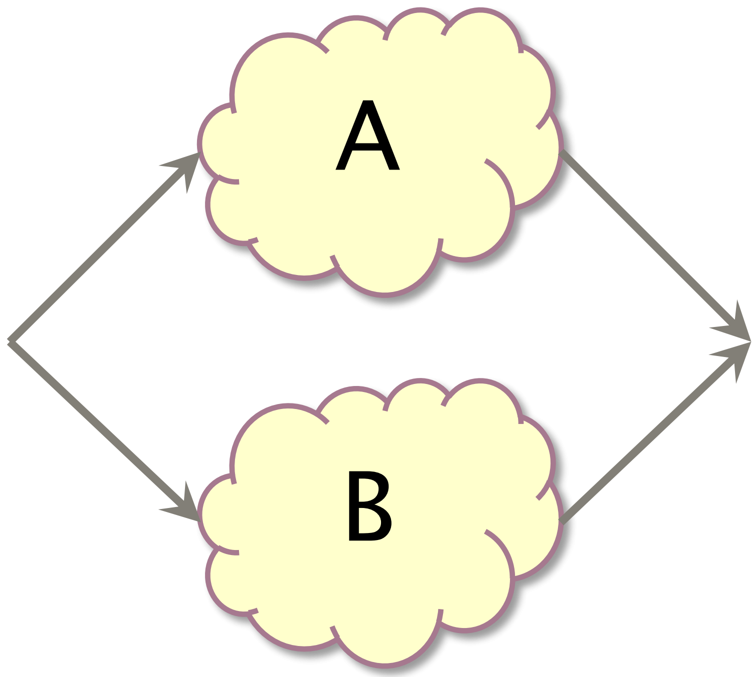
Series Composition



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

Def. T_1/T_P = ***speedup*** on P processors.

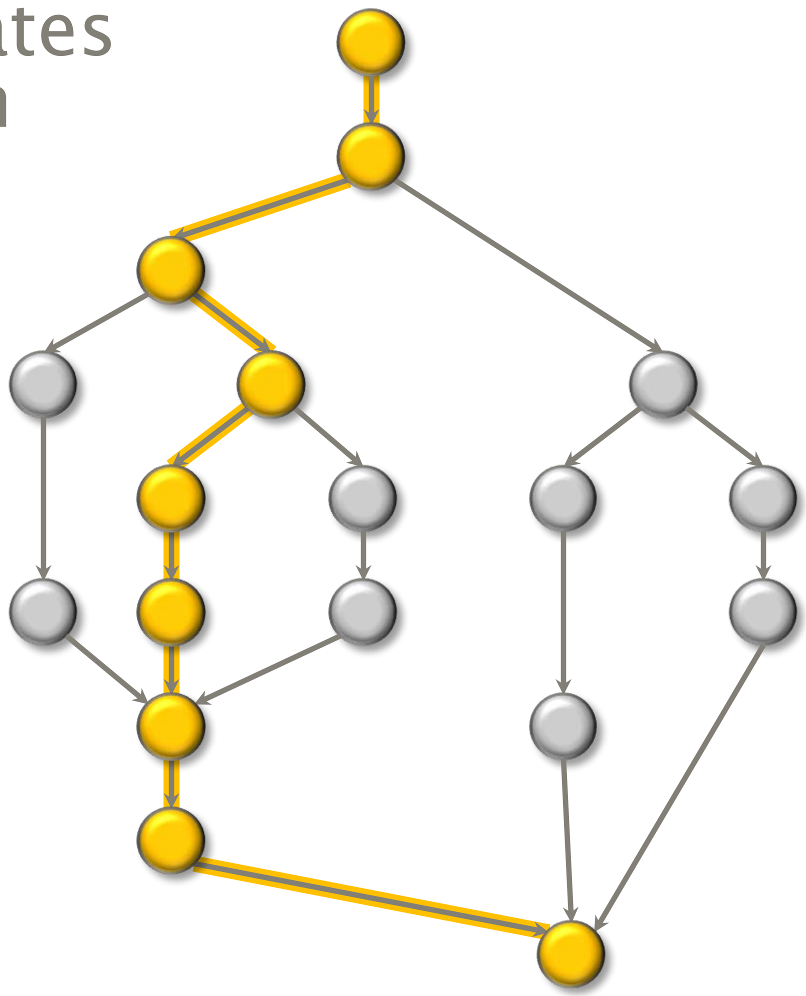
If $T_1/T_P = P$, we have ***(perfect) linear speedup***.

If $T_1/T_P > P$, we have ***superlinear speedup***, which is not possible in this performance model, because of the Work Law $T_P \geq T_1/P$.

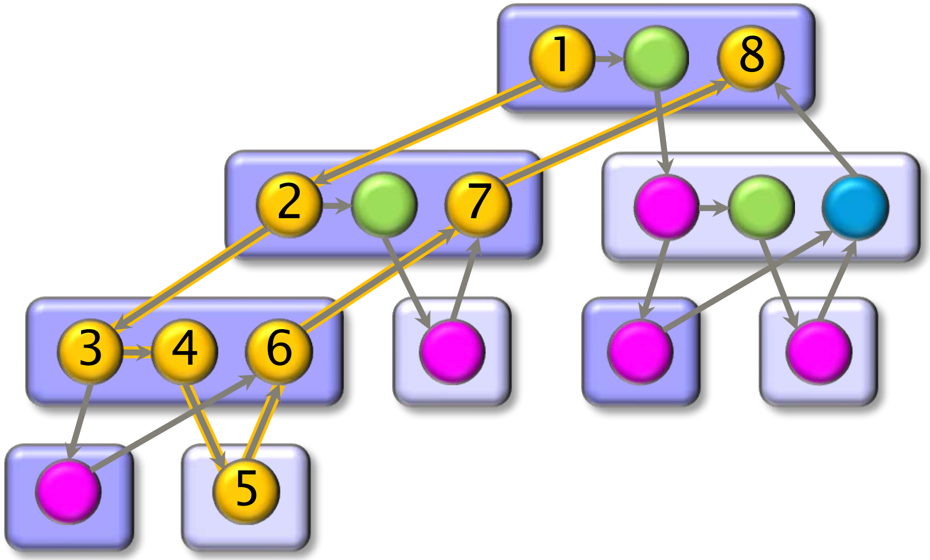
Parallelism

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

$$\begin{aligned} T_1/T_\infty &= \textit{parallelism} \\ &= \text{the average amount of work per step along the span.} \\ &= 18/9 \\ &= 2\ . \end{aligned}$$



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.

Analysis of Parallelism

- The Cilk++ tool suite provides a *scalability analyzer* called *Cilkview*.
- Like the Cilkscreen race detector, Cilkview uses *dynamic instrumentation*.
- Cilkview computes *work* and *span* to derive upper bounds on parallel performance.
- Cilkview also estimates scheduling overhead to compute a *burdened span* for lower bounds.

Quicksort Analysis

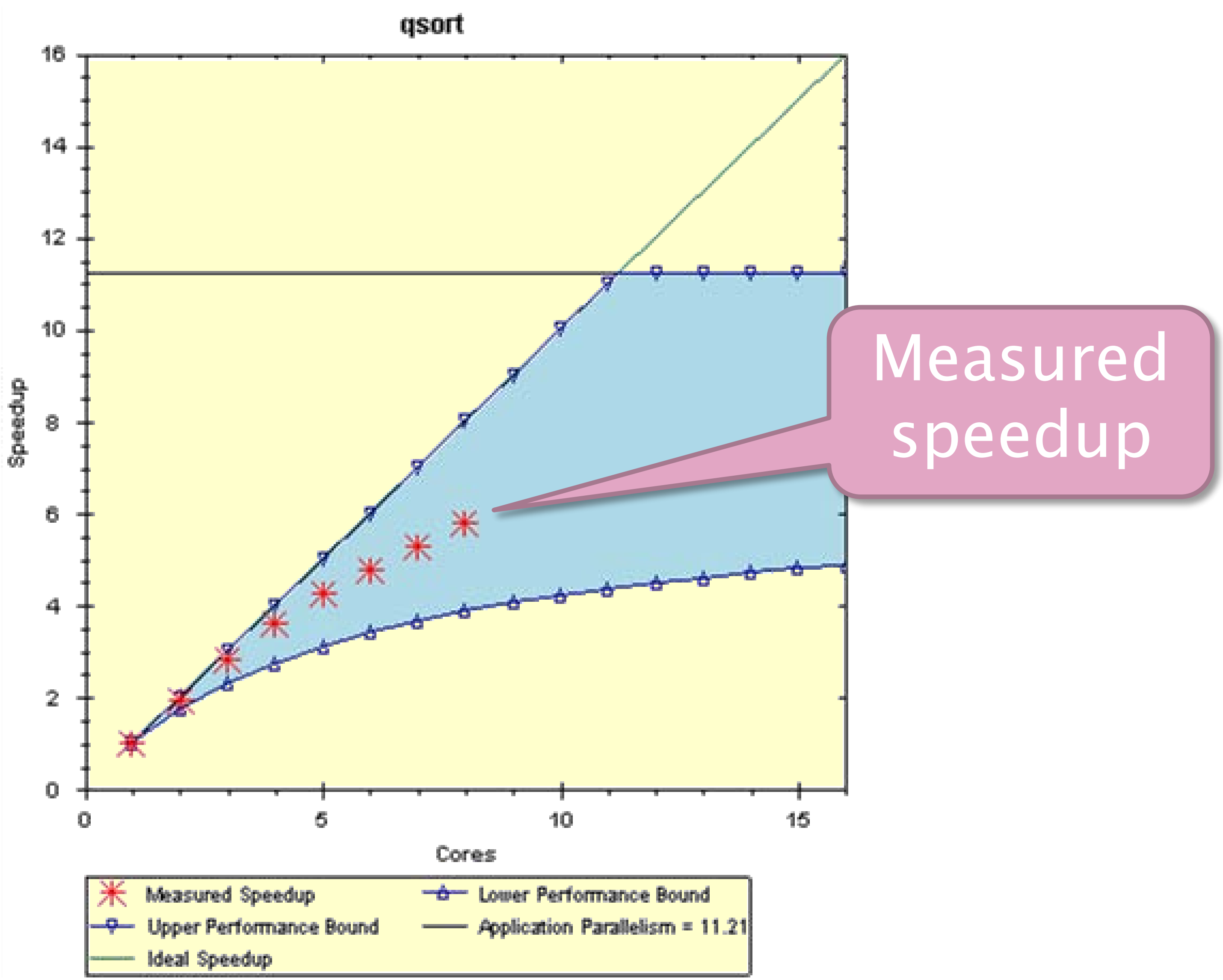
Example: Parallel quicksort

```
template <typename T>
void qsort(T begin, T end) {
  if (begin != end) {
     T middle = partition(
                   begin,
                   end,
                   bind2nd( less<typename iterator_traits<T>::value_type>(),
                            *begin )
                );
     cilk_spawn qsort(begin, middle);
     qsort(max(begin + 1, middle), end);
     cilk_sync;
  }
}
```

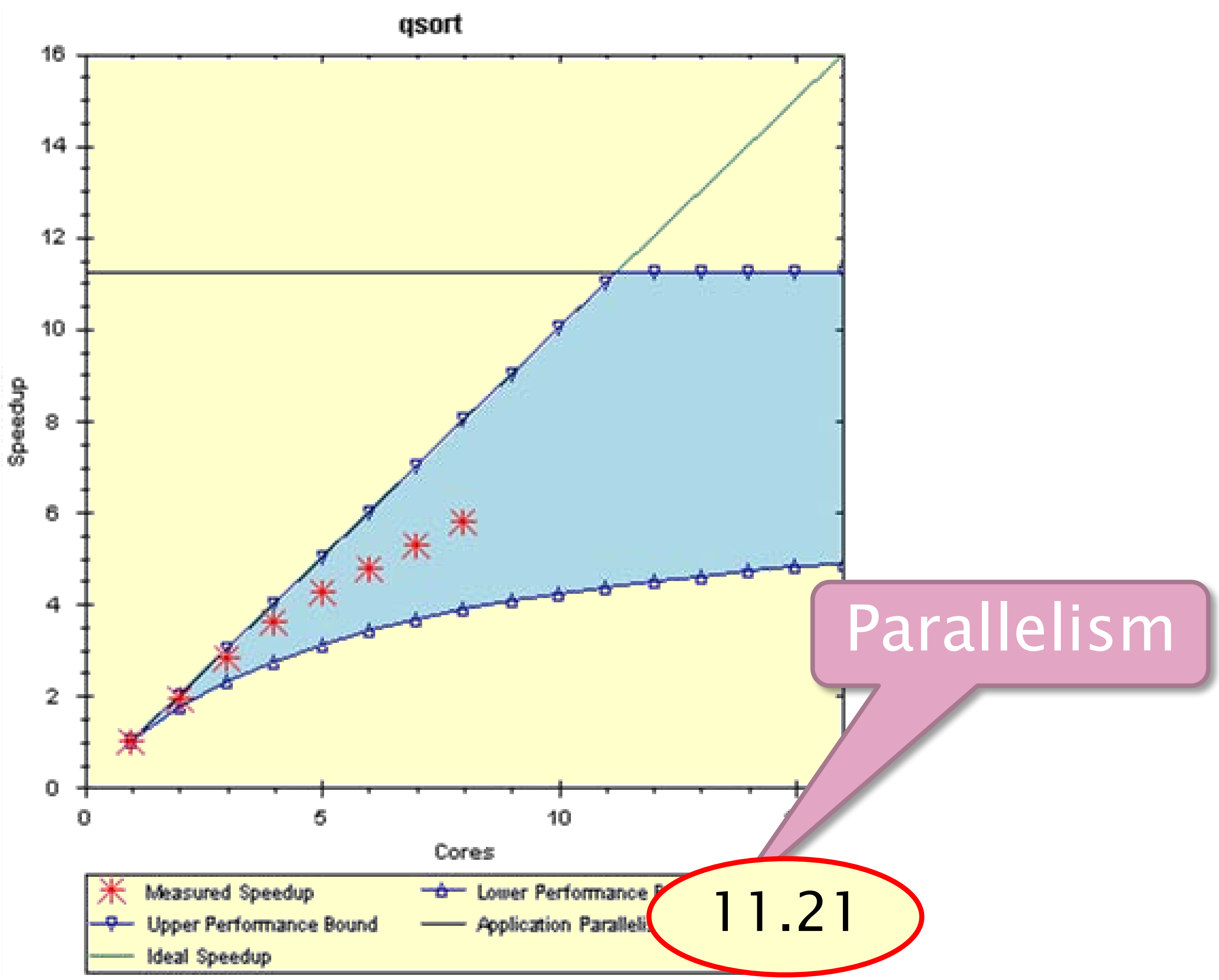
Analyze the sorting of 100,000,000 numbers.

★ ★ ★ ***Guess the parallelism!*** ★ ★ ★

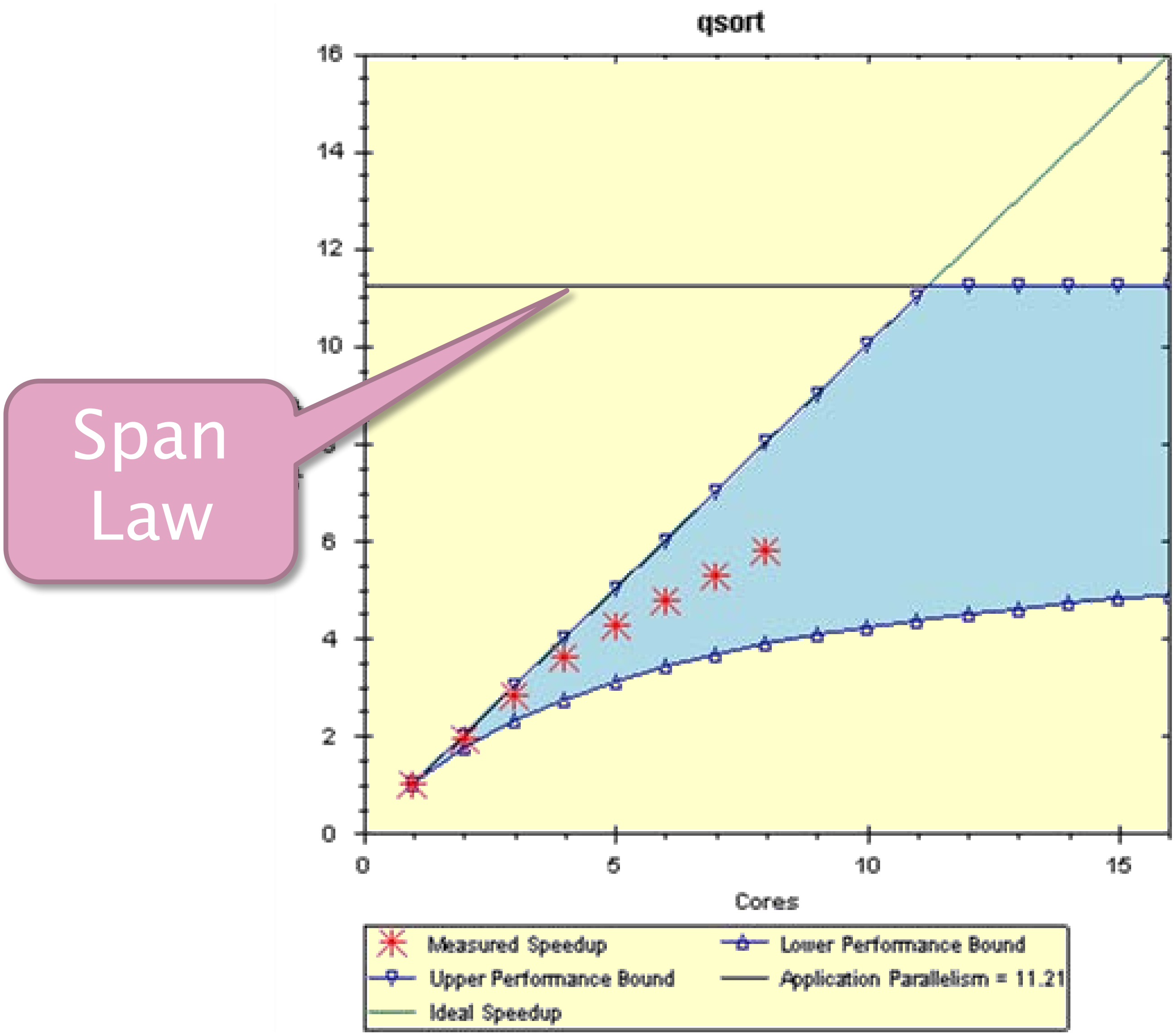
Cilkview Output



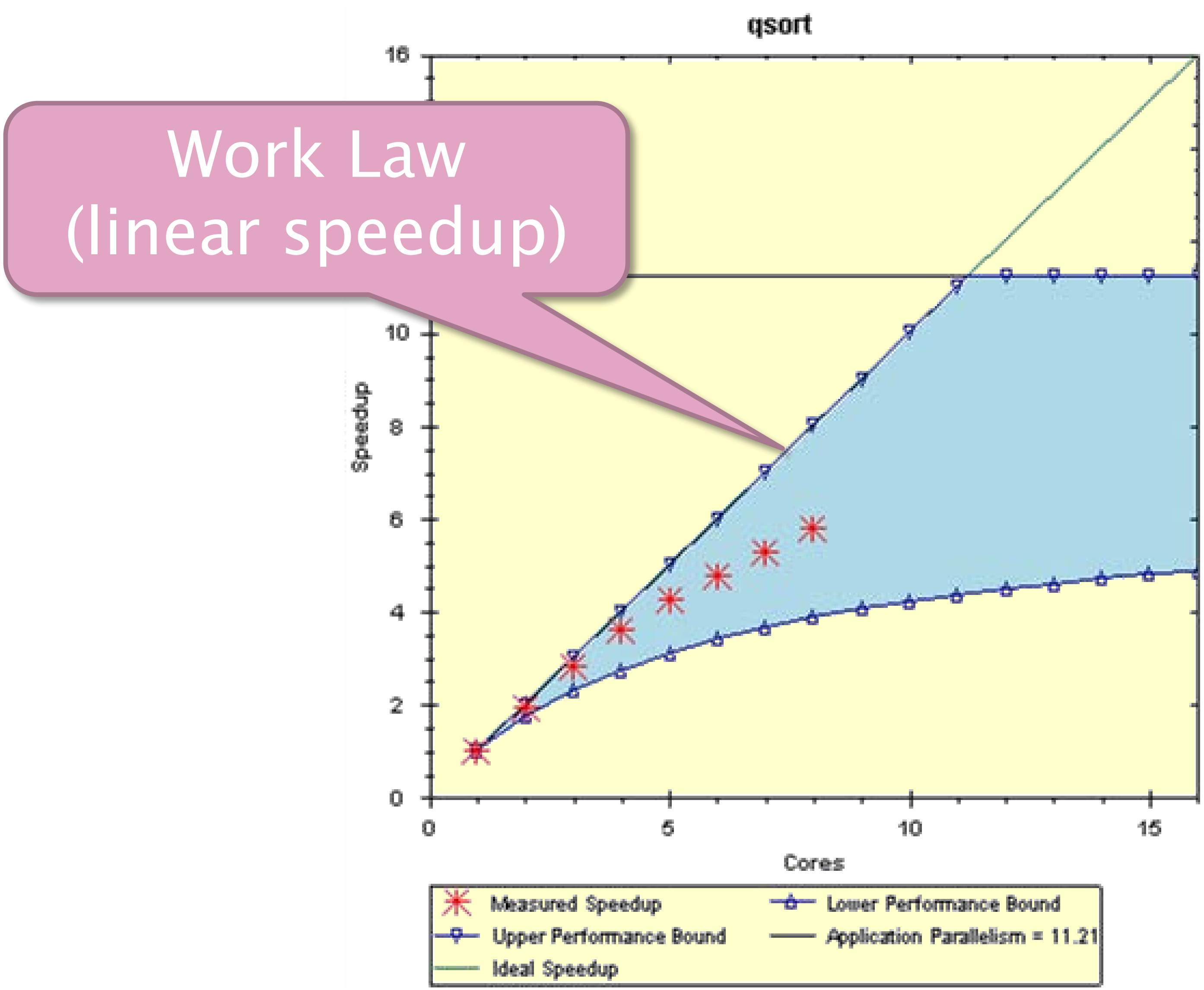
Cilkview Output



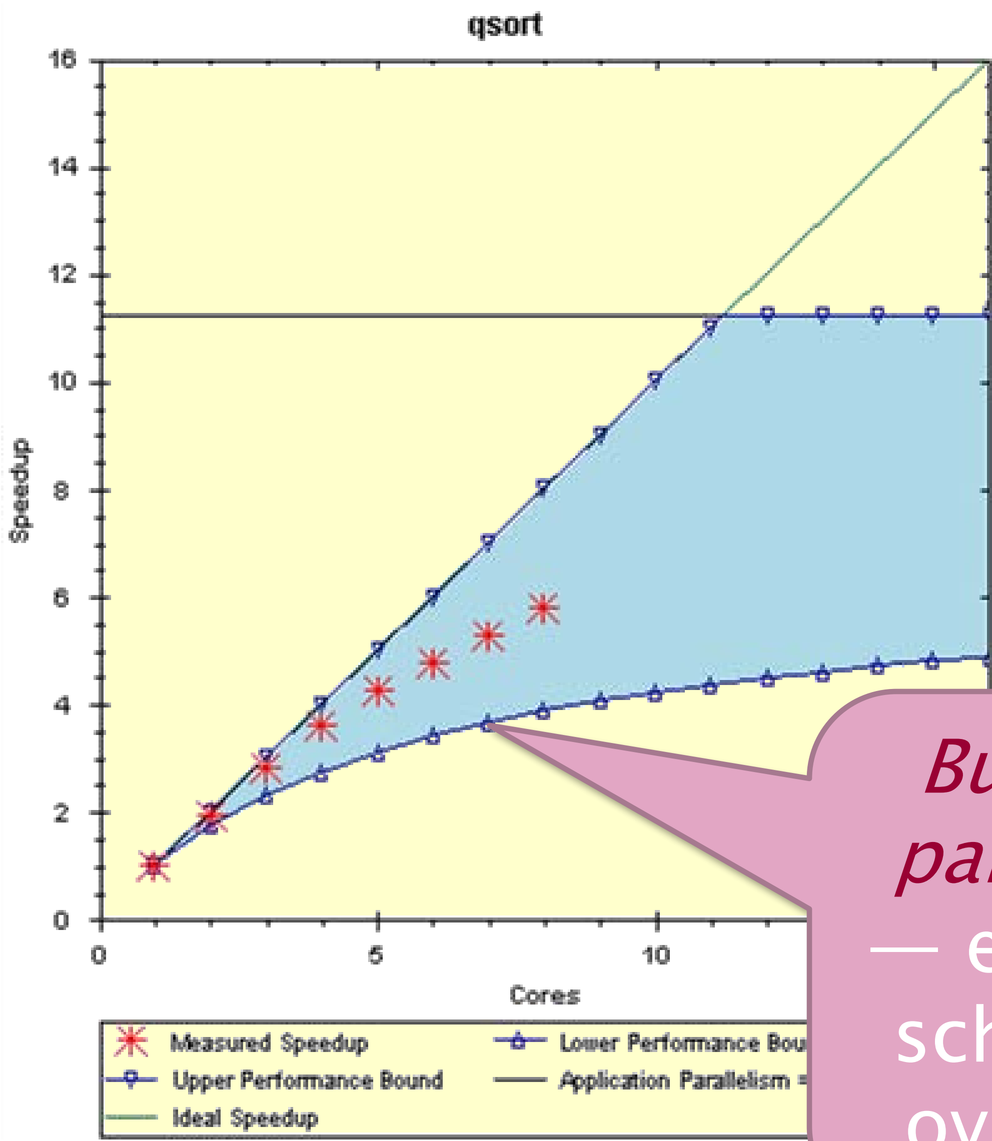
Cilkview Output



Cilkview Output



Cilkview Output



Burdened parallelism — estimates scheduling overheads

Theoretical Analysis

Example: Parallel quicksort

```
template <typename T>
void qsort(T begin, T end) {
  if (begin != end) {
     T middle = partition(
                  begin,
                  end,
                  bind2nd( less<typename iterator_traits<T>::value_type>(),
                           *begin )
               );
     cilk_spawn qsort(begin, middle);
     qsort(max(begin + 1, middle), end);
     cilk_sync;
  }
}
```

Expected work = $O(n \lg n)$
Expected span = $\Omega(n)$

Parallelism = $O(\lg n)$

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	$\Theta(E)$	$\Theta(\Delta \lg V)$	$\Theta(E/\Delta \lg V)$

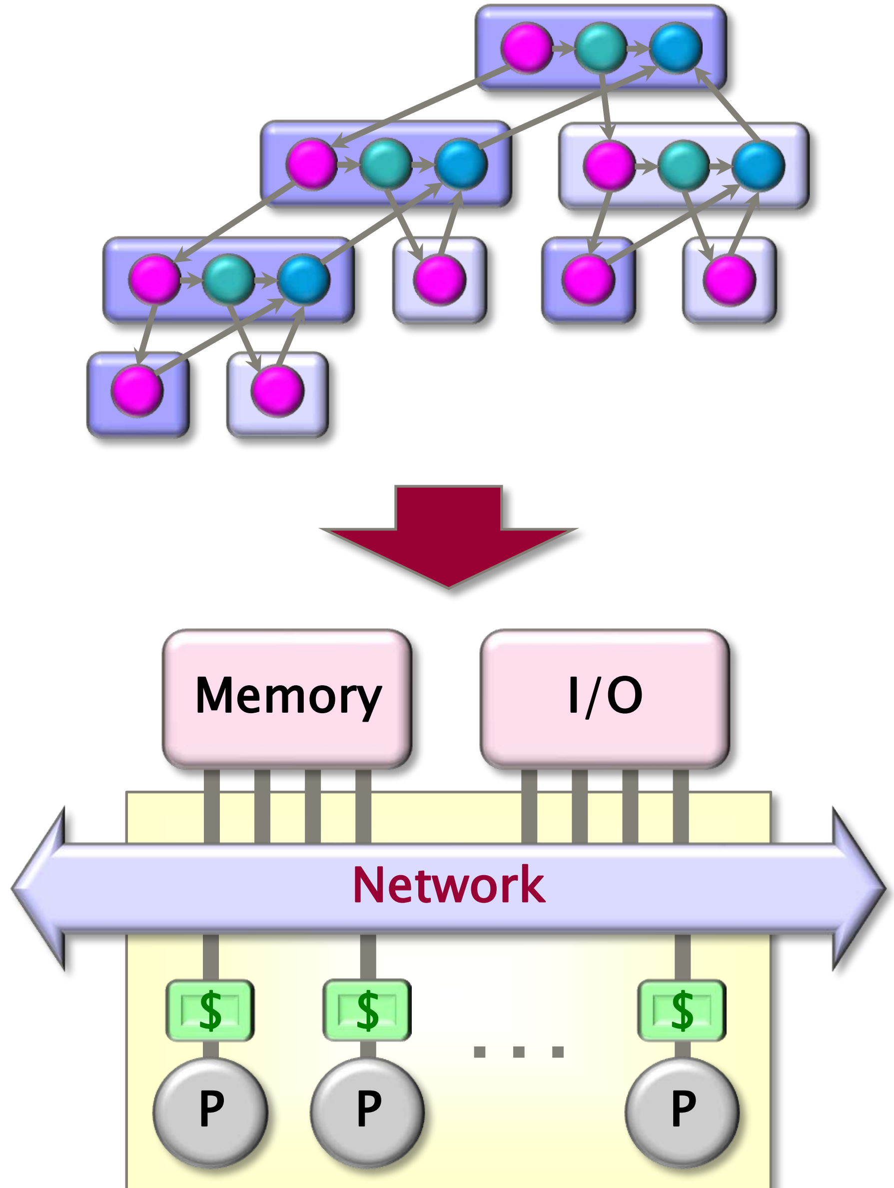
*Cilk++ on 1 processor competitive with the best C++.

OUTLINE

- What Is Parallelism?
- Scheduling Theory
- Cilk++ Runtime System
- A Chess Lesson

Scheduling

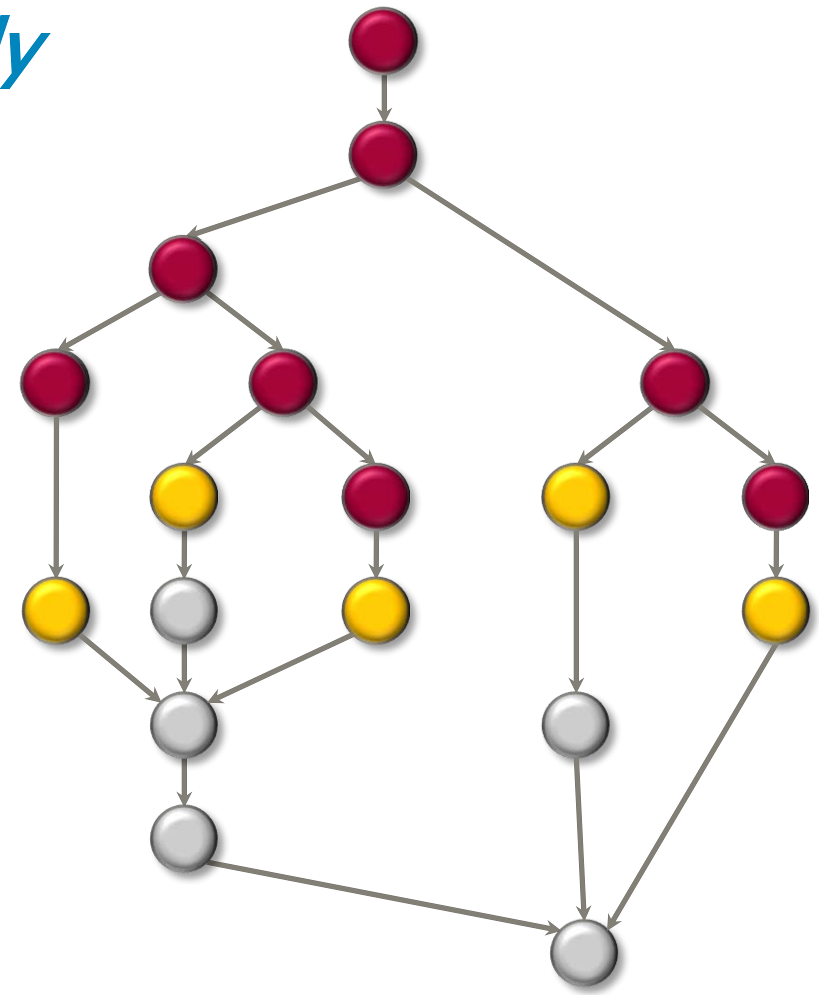
- Cilk++ allows the programmer to express *potential* parallelism in an application.
- The Cilk++ ***scheduler*** maps strands onto processors dynamically at runtime.
- Since the theory of ***distributed*** schedulers is complicated, we'll explore the ideas with a ***centralized*** scheduler.



Greedy Scheduling

IDEA: Do as much as possible on every step.

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



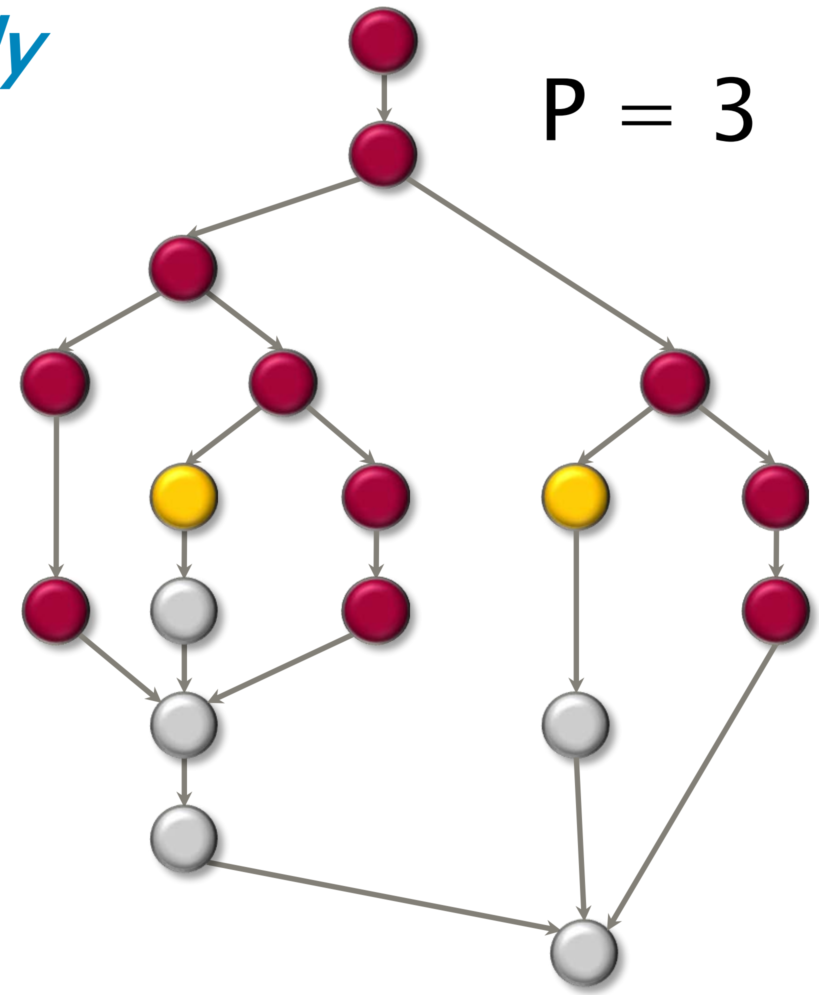
Greedy Scheduling

IDEA: Do as much as possible on every step.

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

Complete step

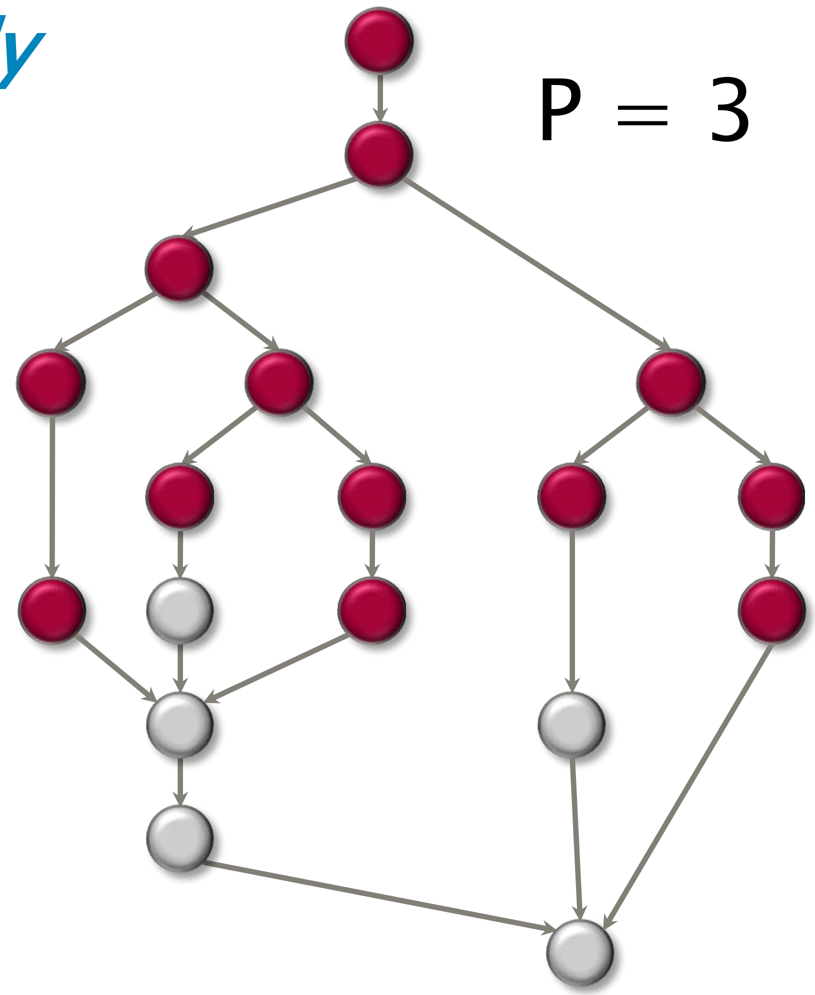
- $\geq$ P strands ready.
- Run any P.



Greedy Scheduling

IDEA: Do as much as possible on every step.

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



Complete step

- ≥ P strands ready.
- Run any P.

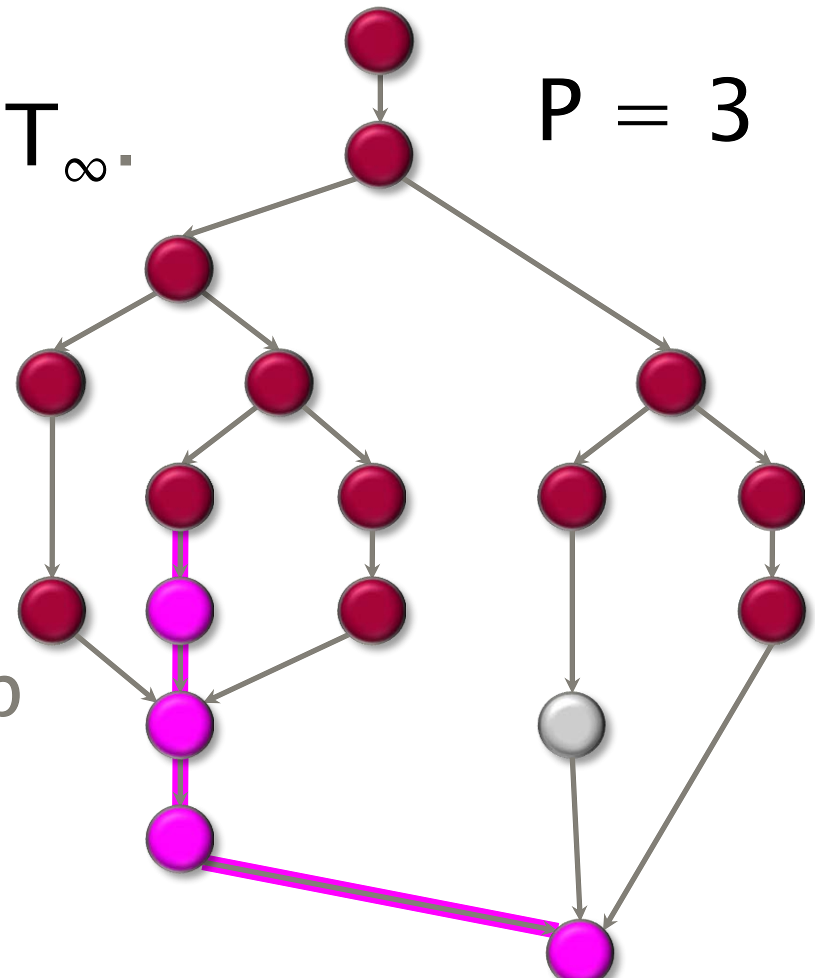
Incomplete step

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

Analysis of Greedy

Theorem [G68, B75, EZL89]**.** 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 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

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

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

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

Cilk++ Performance

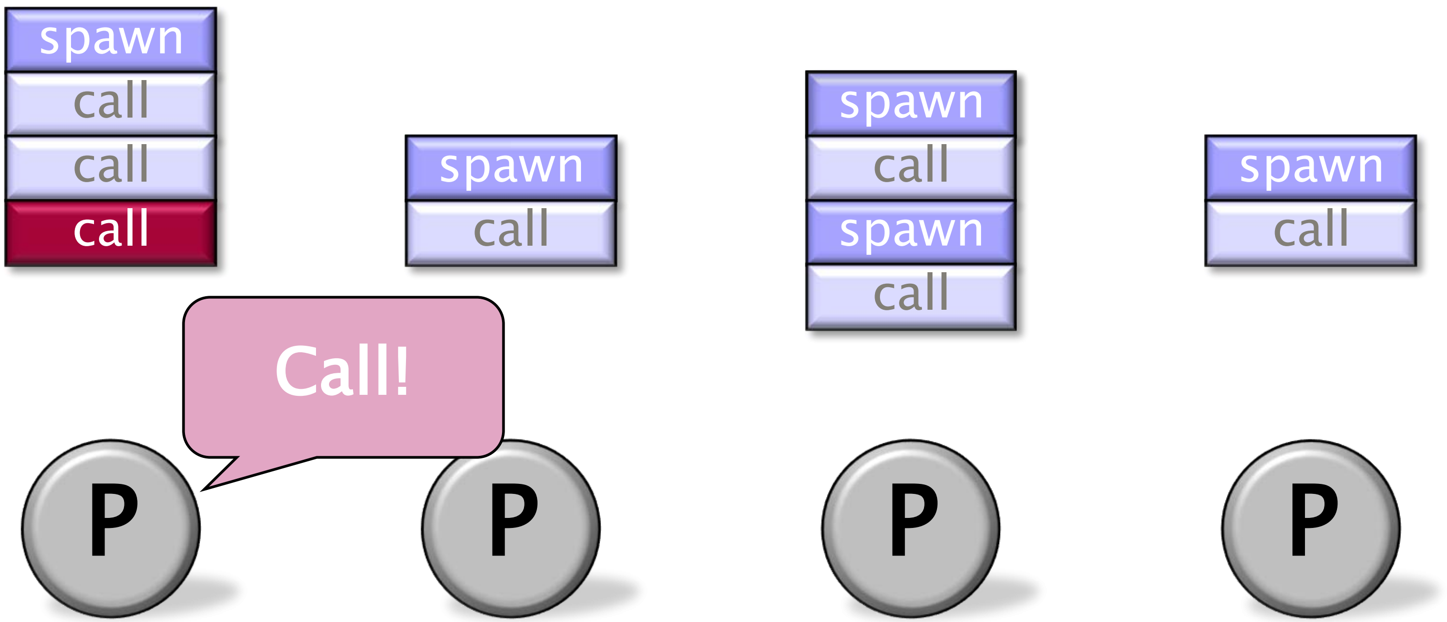
- Cilk++'s work-stealing scheduler achieves
  - $T_P = T_1/P + O(T_\infty)$ expected time (provably);
  - $T_P \approx T_1/P + T_\infty$ time (empirically).
- Near-perfect linear speedup as long as $P \ll T_1/T_\infty$.
- Instrumentation in Cilkview allows the programmer to measure T_1 and T_∞ .

OUTLINE

- What Is Parallelism?
- Scheduling Theory
- Cilk++ Runtime System
- A Chess Lesson

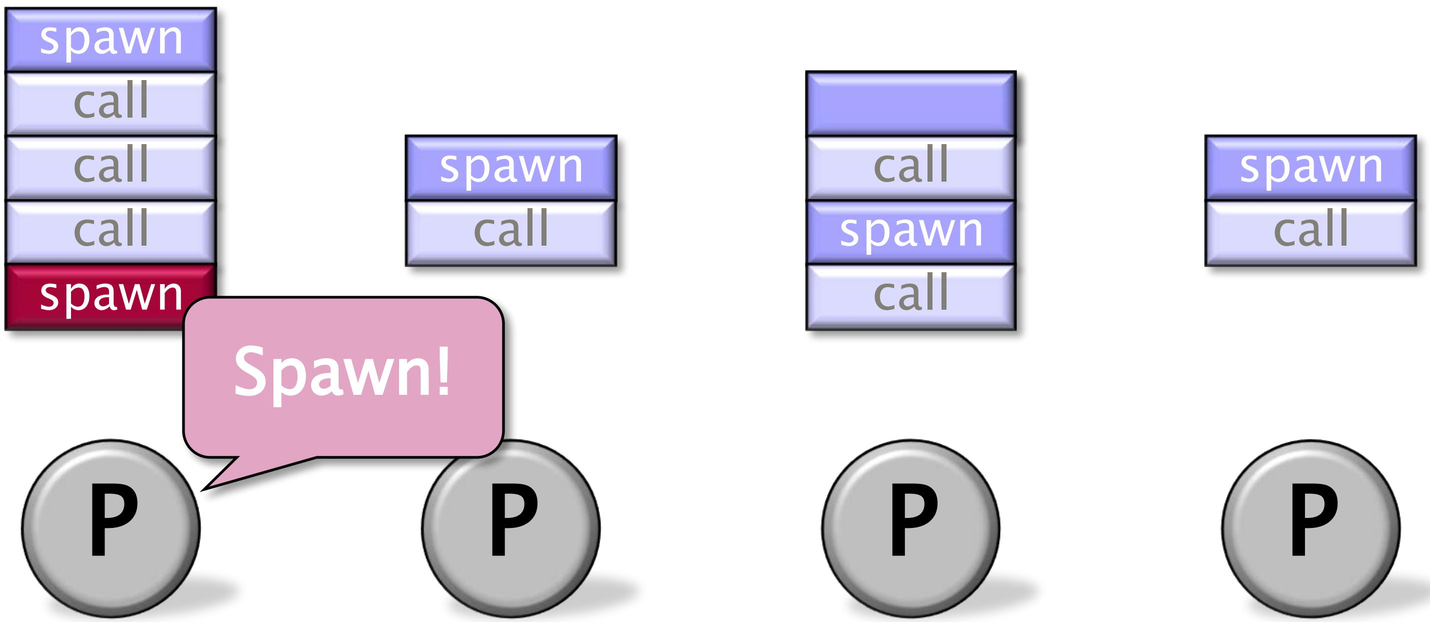
Cilk++ Runtime System

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



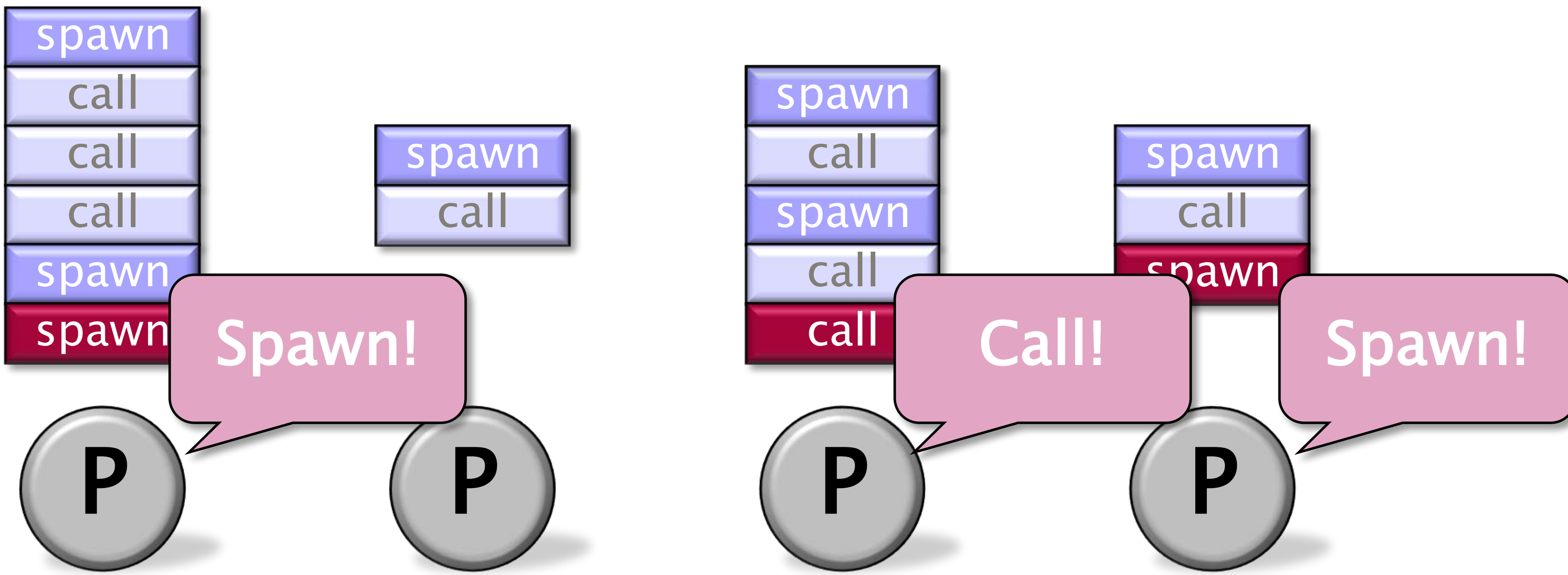
Cilk++ Runtime System

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



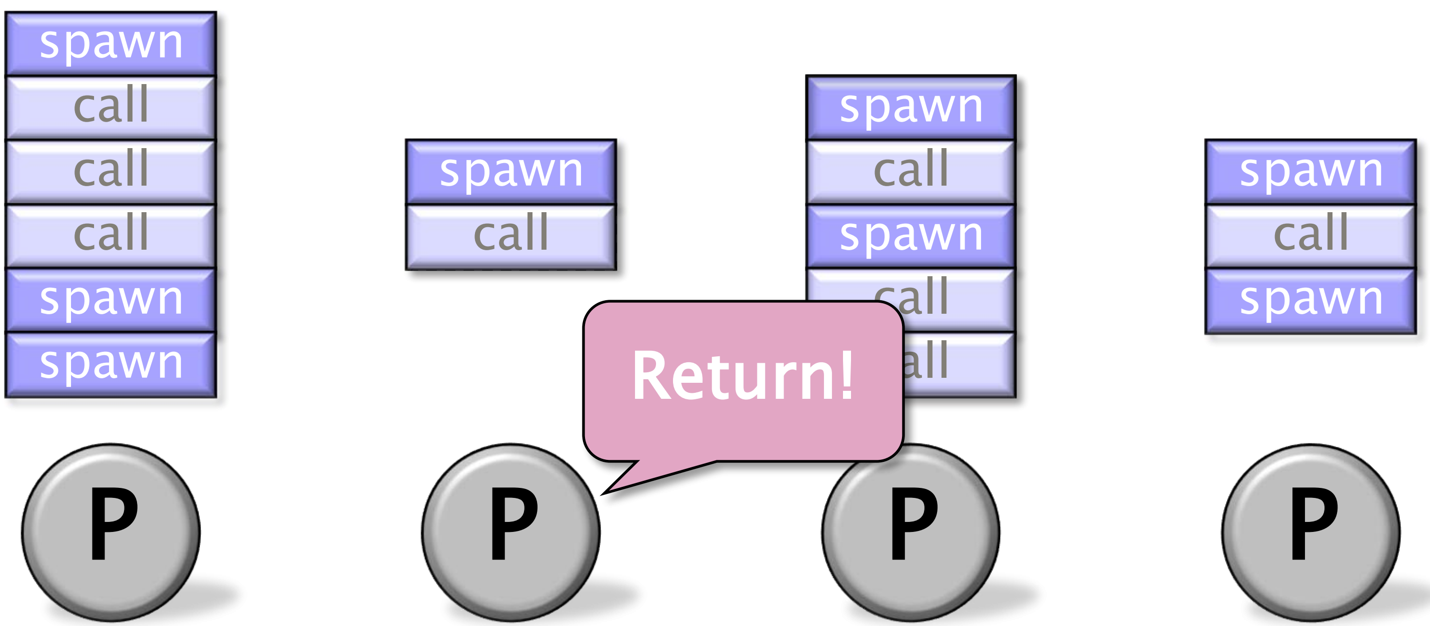
Cilk++ Runtime System

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



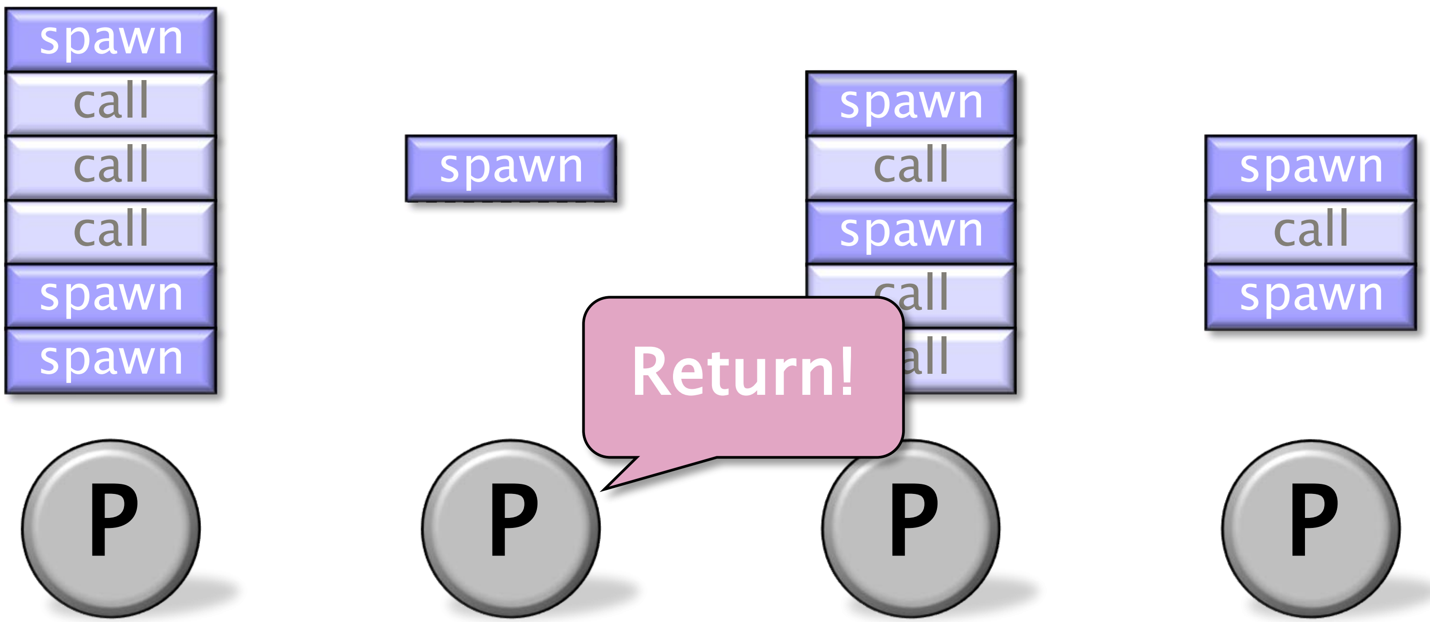
Cilk++ Runtime System

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



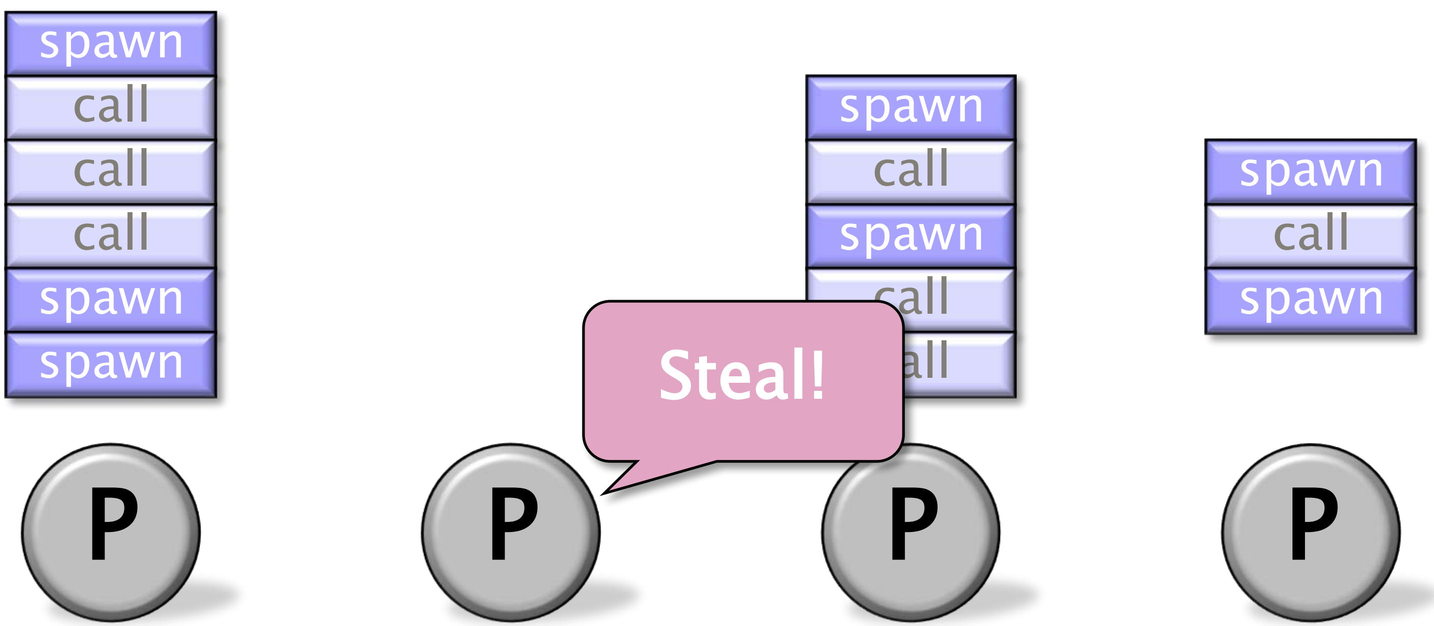
Cilk++ Runtime System

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

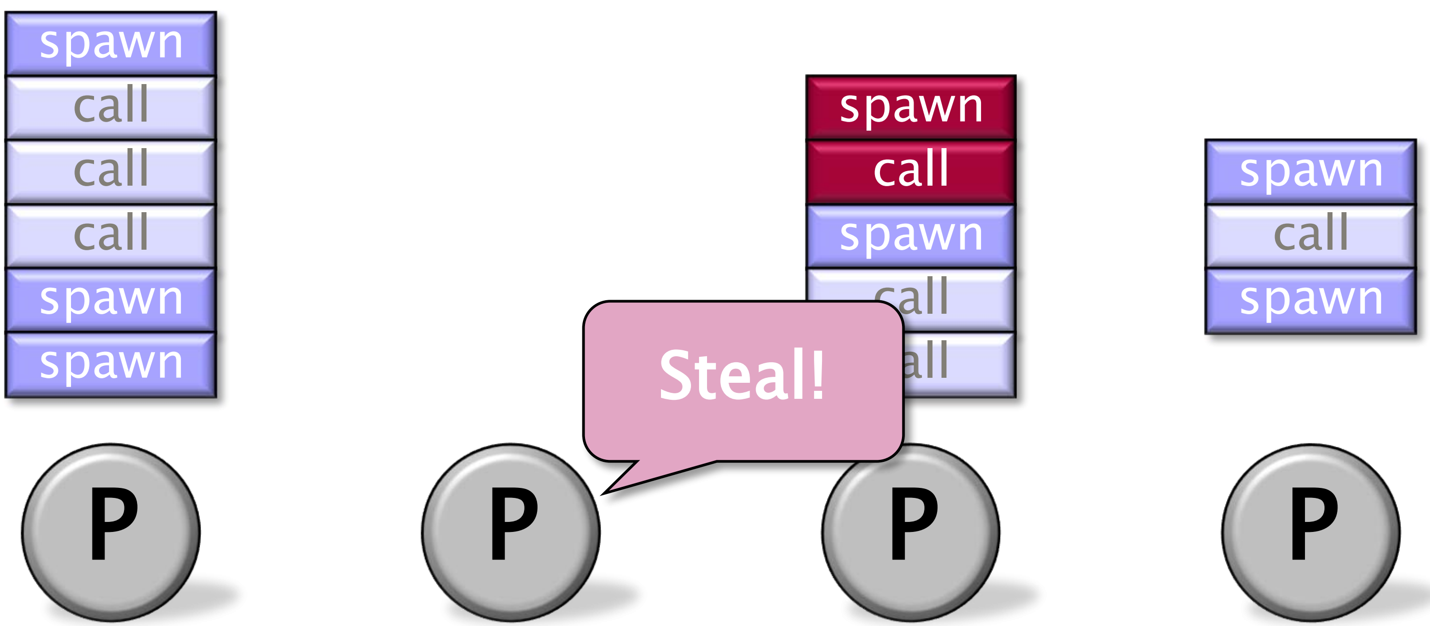
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

Cilk++ Runtime System

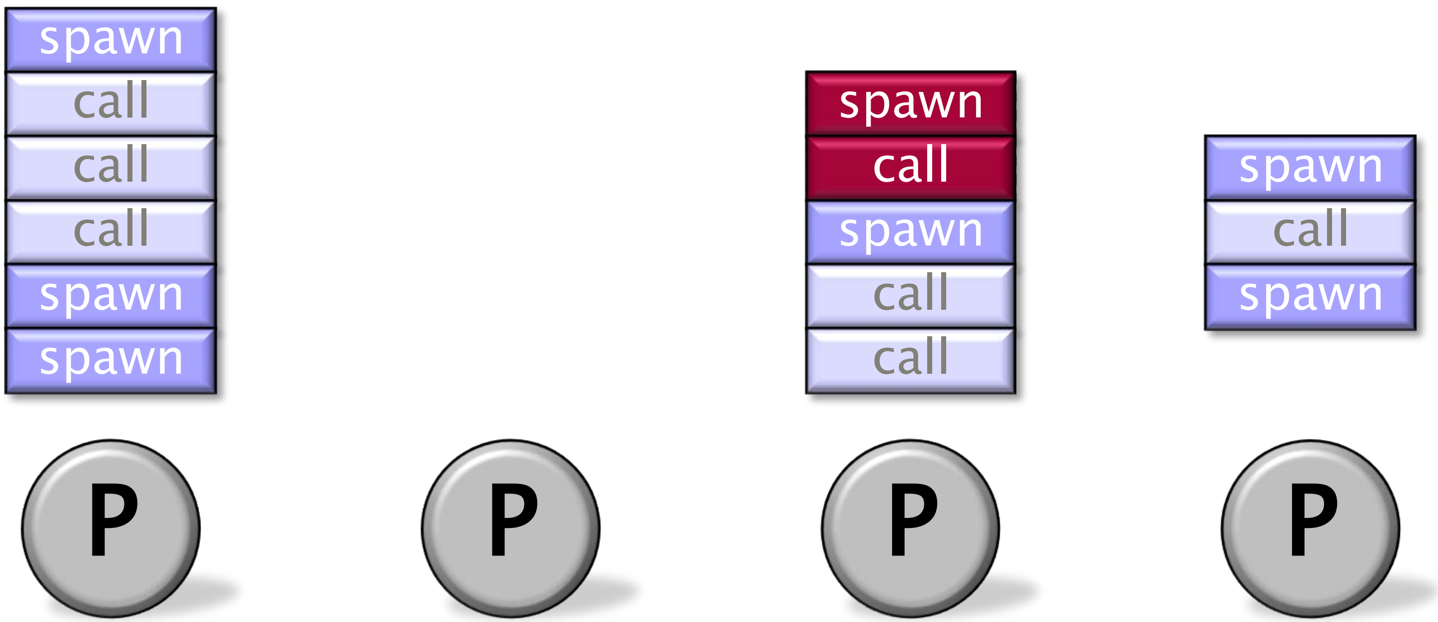
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

Cilk++ Runtime System

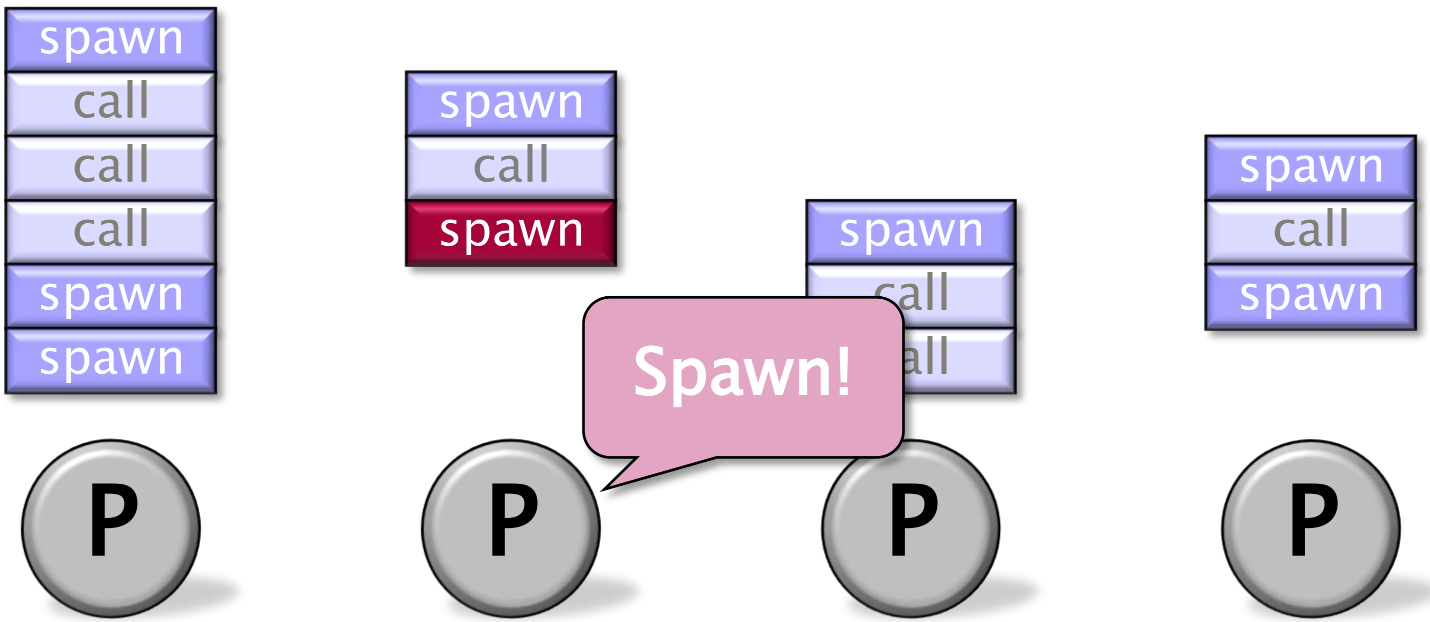
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

Cilk++ Runtime System

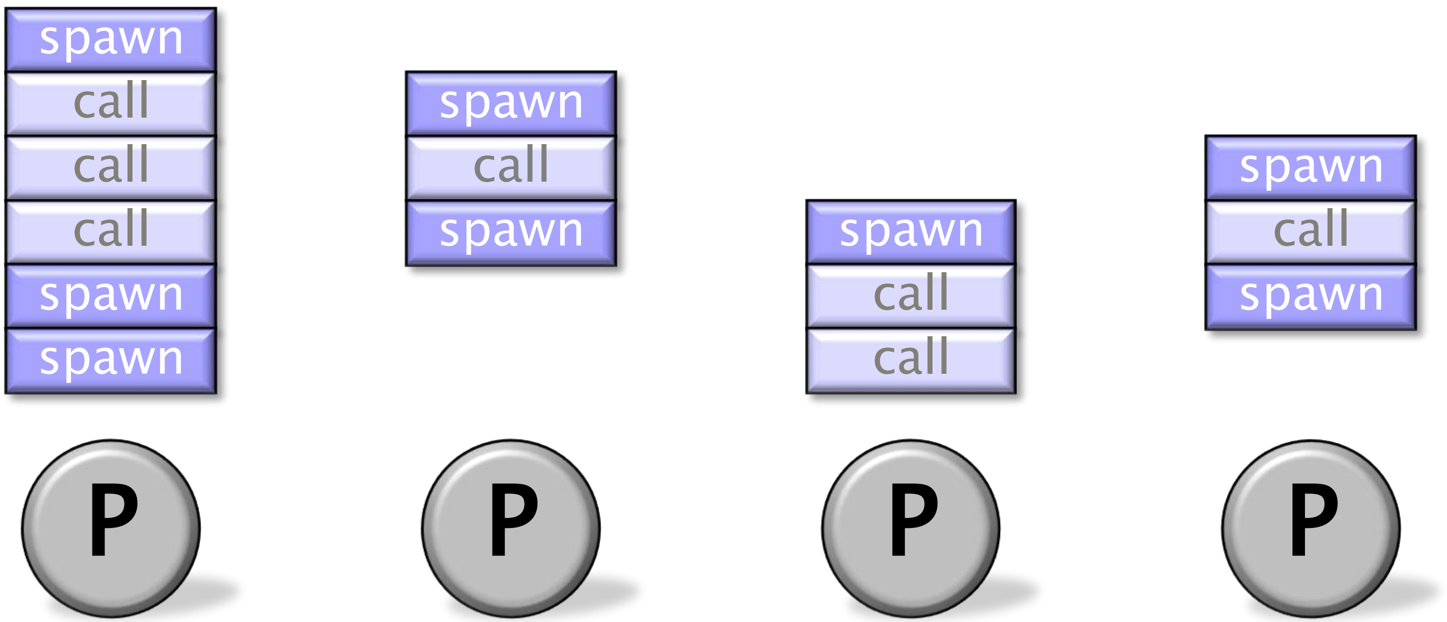
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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

Cilk++ Runtime System

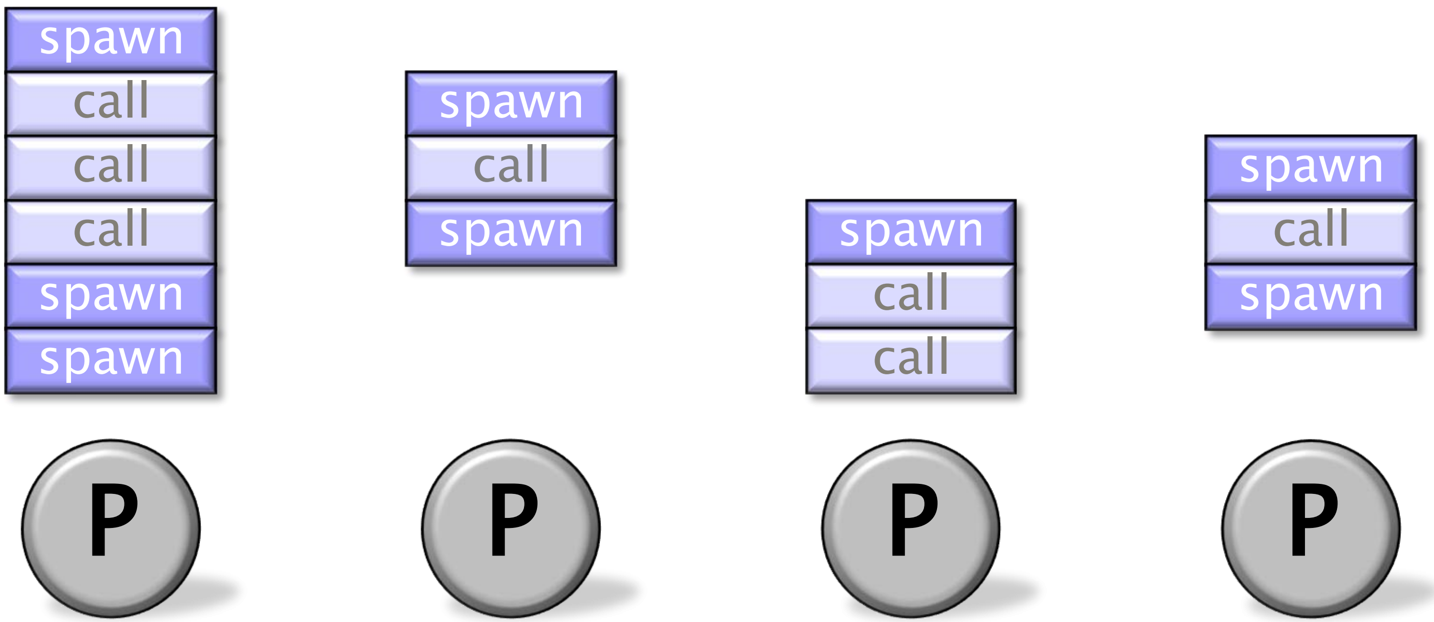
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].



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 ⇒ ***linear speed-up***.

Work-Stealing Bounds

Theorem. The Cilk++ work-stealing scheduler achieves expected running time

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

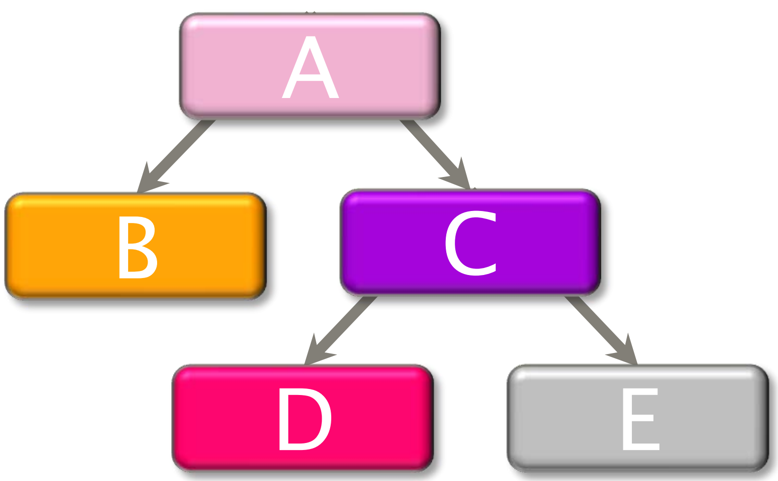
on P processors.

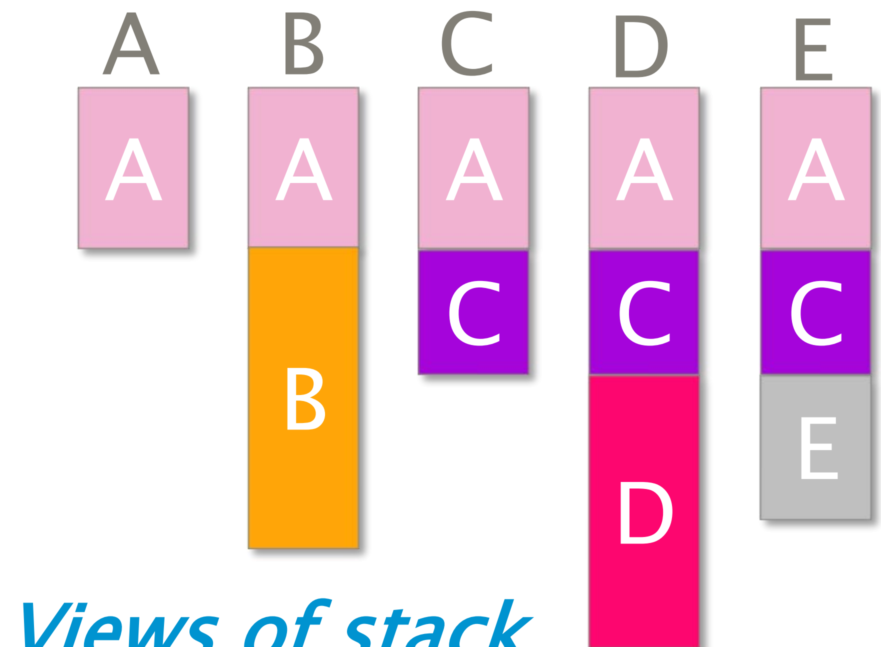
Pseudoproof. A processor is either *working* or *stealing*. The total time all processors spend working is T_1. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(PT_\infty)$. Since there are P processors, the expected time is

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

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.



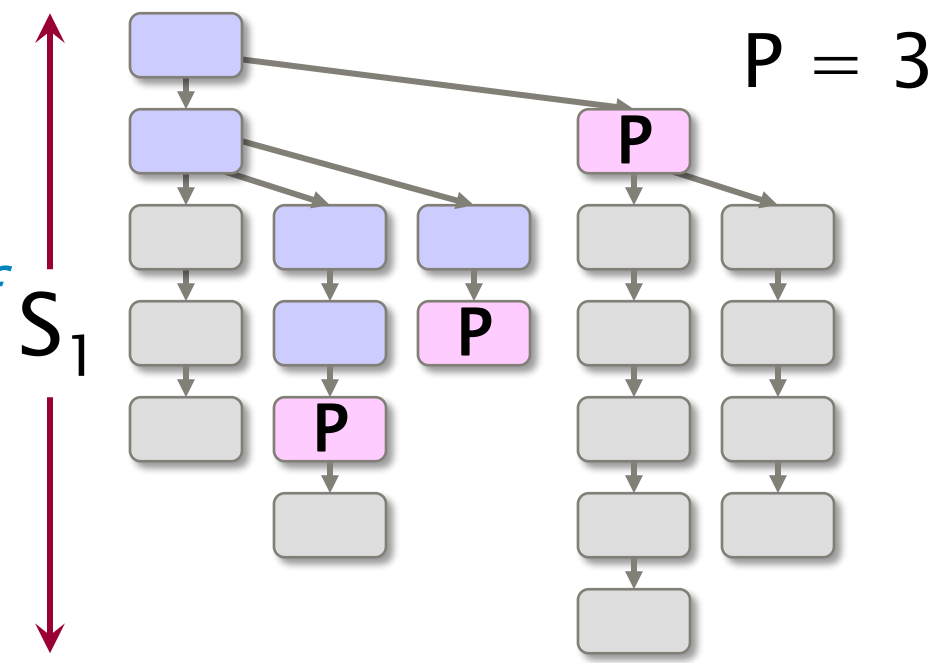


Cilk++'s ***cactus stack*** supports multiple views in parallel.

Space Bounds

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 $S_P \le PS_1$.

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



Linguistic Implications

Code like the following executes properly without any risk of blowing out memory:

```
for (int i=1; i<1000000000; ++i) {
  cilk_spawn foo(i);
}
cilk_sync;
```

MORAL: *Better to steal parents from their children than children from their parents!*

OUTLINE

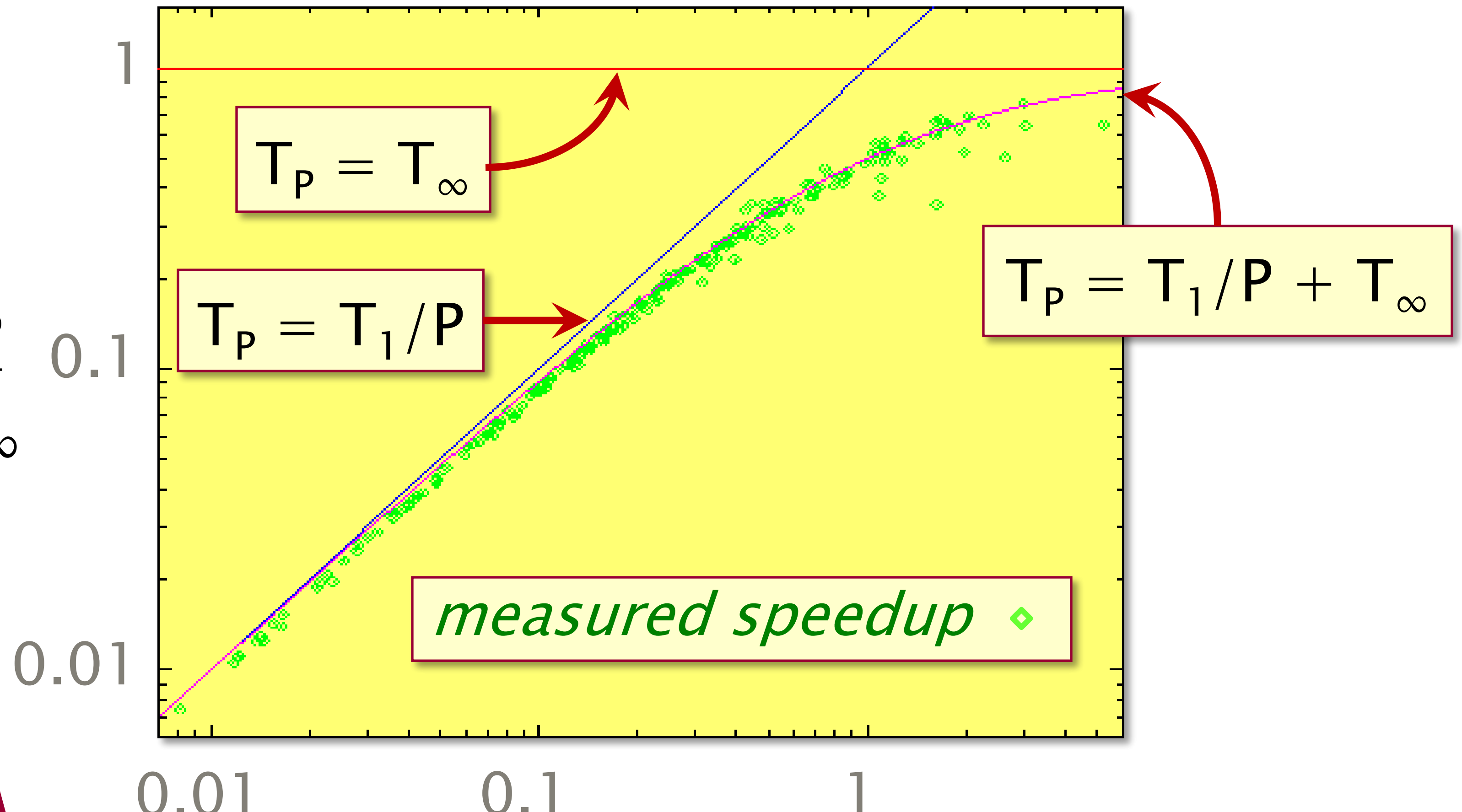
- What Is Parallelism?
- Scheduling Theory
- Cilk++ Runtime System
- A Chess Lesson

Cilk Chess Programs

- ★Socrates placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node Connection Machine CM5.
- ★Socrates 2.0 took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs' 1824-node Intel Paragon.
- Cilkchess placed 1st in the 1996 Dutch Open running on a 12-processor Sun Enterprise 5000. It placed 2nd in 1997 and 1998 running on Boston University's 64-processor SGI Origin 2000.
- Cilkchess tied for 3rd in the 1999 WCCC running on NASA's 256-node SGI Origin 2000.

★Socrates Speedup

$$\frac{T_1/T_P}{T_1/T_\infty}$$



Normalize by parallelism

$$\frac{P}{T_1/T_\infty}$$

Developing ★Socrates

- For the competition, ★Socrates was to run on a 512-processor Connection Machine Model CM5 supercomputer at the University of Illinois.
- The developers had easy access to a similar 32-processor CM5 at MIT.
- One of the developers proposed a change to the program that produced a speedup of over 20% on the MIT machine.
- After a back-of-the-envelope calculation, the proposed "improvement" was rejected!

★Socrates Paradox

Original program	*Proposed program*
$T_{32} = 65$ seconds	$T'_{32} = 40$ seconds

$$T_P \approx T_1/P + T_\infty$$

$T_1 = 2048$ seconds	$T'_1 = 1024$ seconds
$T_\infty = 1$ second	$T'_\infty = 8$ seconds
$T_{32} = 2048/32 + 1$	$T'_{32} = 1024/32 + 8$
$= 65$ seconds	$= 40$ seconds
$T_{512} = 2048/512 + 1$	$T'_{512} = 1024/512 + 8$
$= 5$ seconds	$= 10$ seconds

Moral of the Story

Work and span beat running times for predicting scalability of performance.

6.172 Performance Engineering of Software Systems
Fall 2010