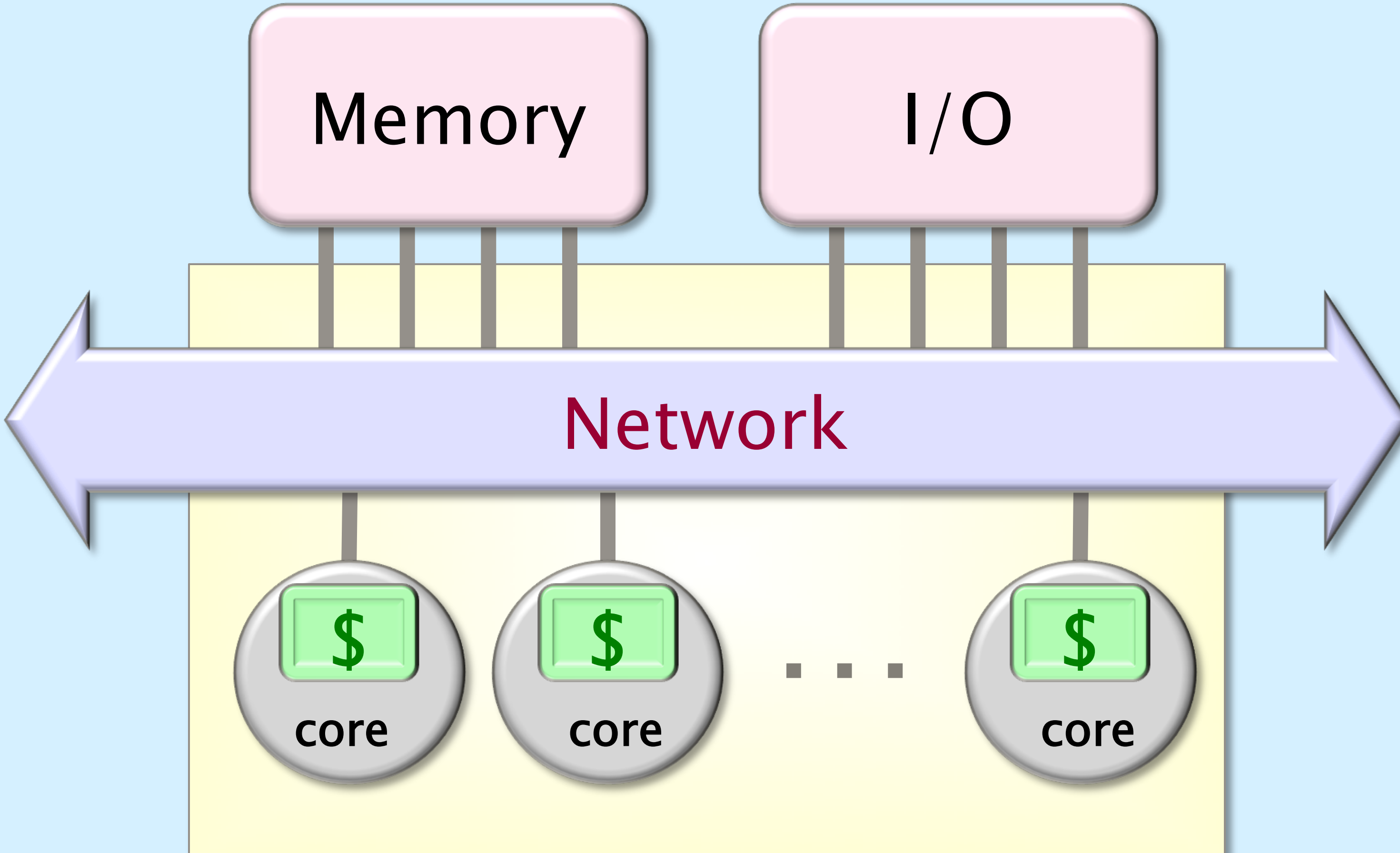
CS 240A: Shared Memory & Multicore Programming with Cilk++

- Multicore and NUMA architectures
- Multithreaded Programming
- Cilk++ as a concurrency platform
- Work, Span, (potential) Parallelism

Thanks to Charles E. Leiserson for some of these slides

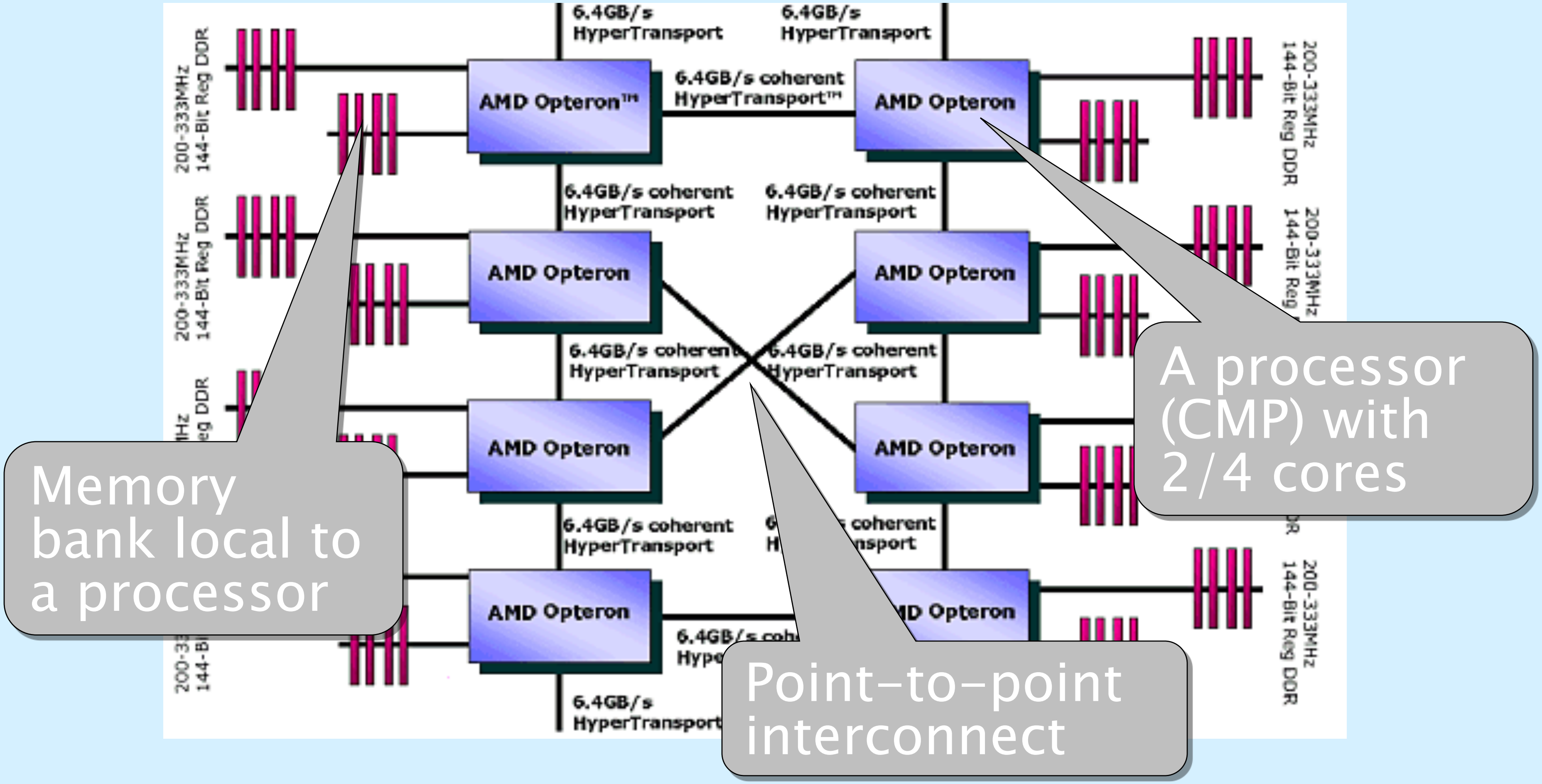
Multicore Architecture



Chip Multiprocessor (CMP)

cc-NUMA Architectures

AMD 8-way Opteron Server (neumann@cs.ucsb.edu)

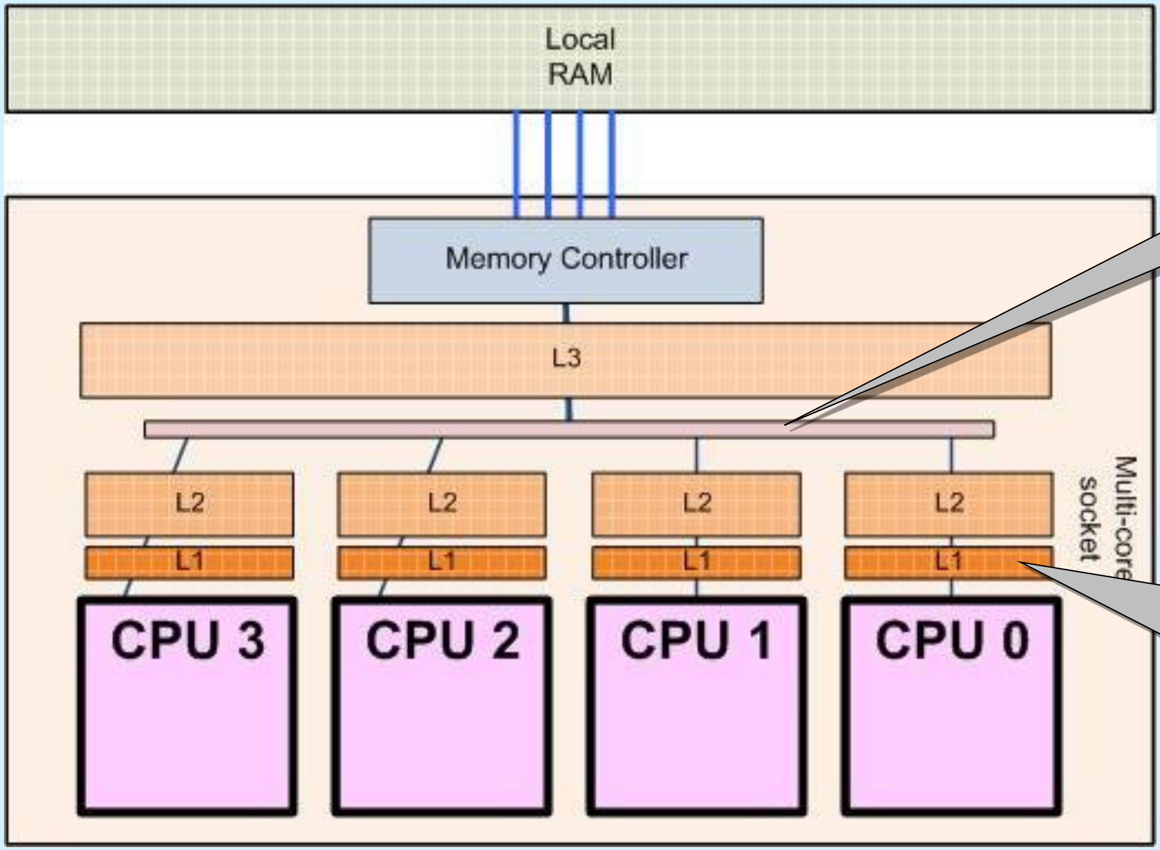


cc-NUMA Architectures

- No Front Side Bus
- Integrated memory controller
- On-die interconnect among CMPs
- Main memory is physically distributed among CMPs (i.e. each piece of memory has an affinity to a CMP)
- NUMA: Non-uniform memory access.
  - For multi-socket servers only
  - Your desktop is safe (well, for now at least)
  - Triton nodes are not NUMA either

Desktop Multicores Today

This is your AMD Barcelona or Intel Core i7 !



On-die interconnect

Private cache: Cache coherence is required

Multithreaded Programming

- POSIX Threads (Pthreads) is a set of threading interfaces developed by the IEEE

- "Assembly language" of shared memory programming

- Programmer has to manually:
  - Create and terminate threads
  - Wait for threads to complete
  - Manage interaction between threads using mutexes, condition variables, etc.

Concurrency Platforms

- Programming directly on PThreads is painful and error-prone.
- With PThreads, you either sacrifice memory usage or load-balance among processors
- A *concurrency platform* provides linguistic support and handles load balancing.
- Examples:
  - Threading Building Blocks (TBB)
  - OpenMP
  - Cilk++

Cilk vs PThreads

How will the following code execute in PThreads? In Cilk?

```
for (i=1; i<1000000000; i++) {
  spawn-or-fork foo(i);
}
sync-or-join;
```

What if foo contains code that waits (e.g., spins) on a variable being set by another instance of foo?

They have different liveness properties:

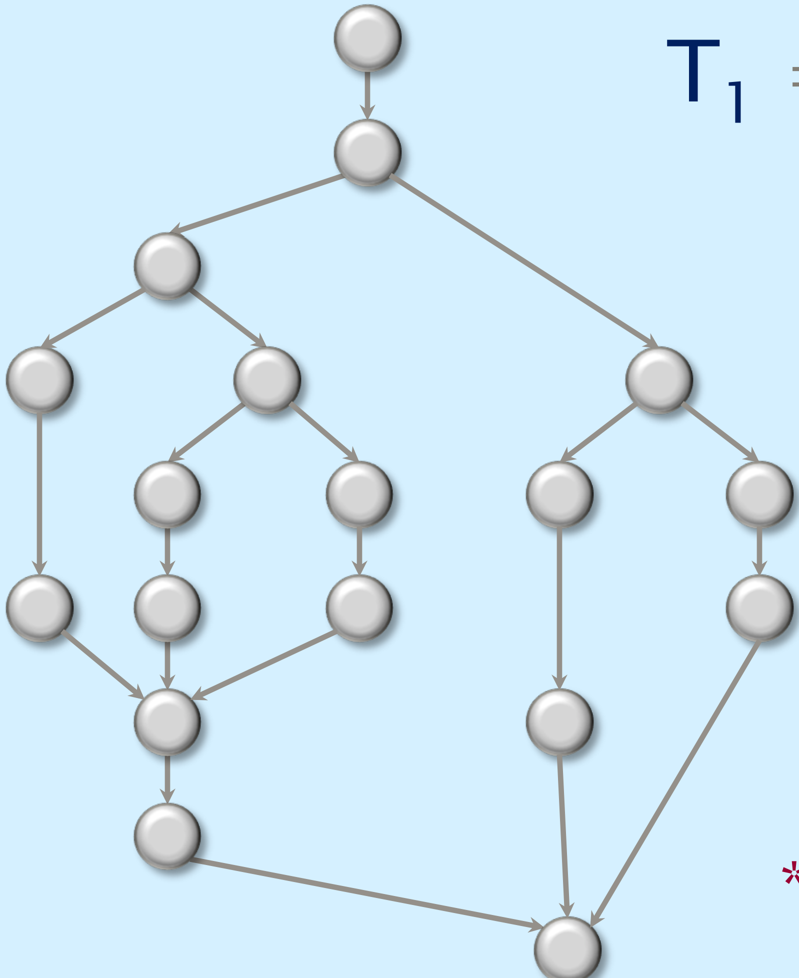
- Cilk threads are spawned lazily, "may" parallelism
- PThreads are spawned eagerly, "must" parallelism

Cilk vs OpenMP

- Cilk++ guarantees space bounds
  - On P processors, Cilk++ uses no more than P times the stack space of a serial execution.
- Cilk++ has a solution for global variables (called “reducers” / “hyperobjects”)
- Cilk++ has nested parallelism that works and provides guaranteed speed-up.
  - Indeed, cilk scheduler is provably optimal.
- Cilk++ has a race detector (cilkscreen) for debugging and software release.
- *Keep in mind that platform comparisons are (always will be) subject to debate*

Complexity Measures

T_P = execution time on P processors



T_1 = ***work*** T_∞ = ***span****

WORK LAW

- $T_P \geq T_1/P$

SPAN LAW

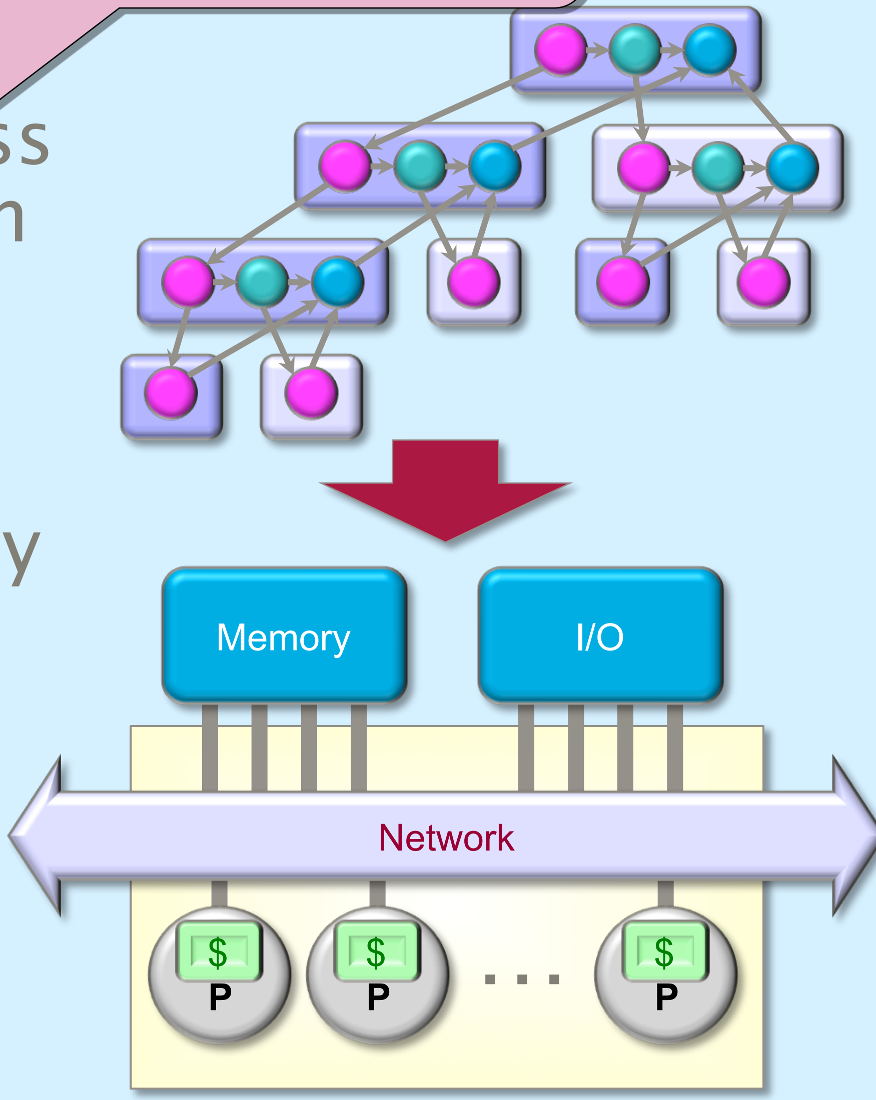
- $T_P \geq T_\infty$

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

Scheduling

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

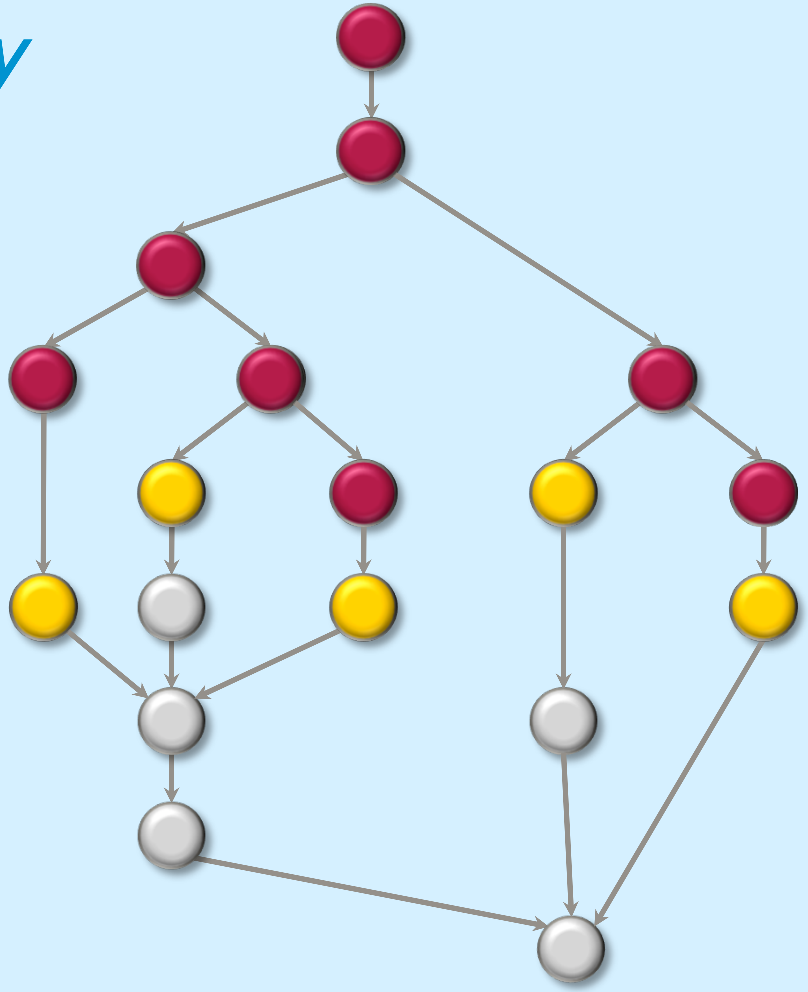
- Cilk++ allows the programmer to express *potential* parallelism in an application.
- The Cilk++ ***scheduler*** maps strands onto processors dynamically at runtime.
- Since ***on-line*** schedulers are complicated, we'll explore the ideas with an ***off-line*** scheduler.



Greedy Scheduling

IDEA: Do as much as possible on every step.

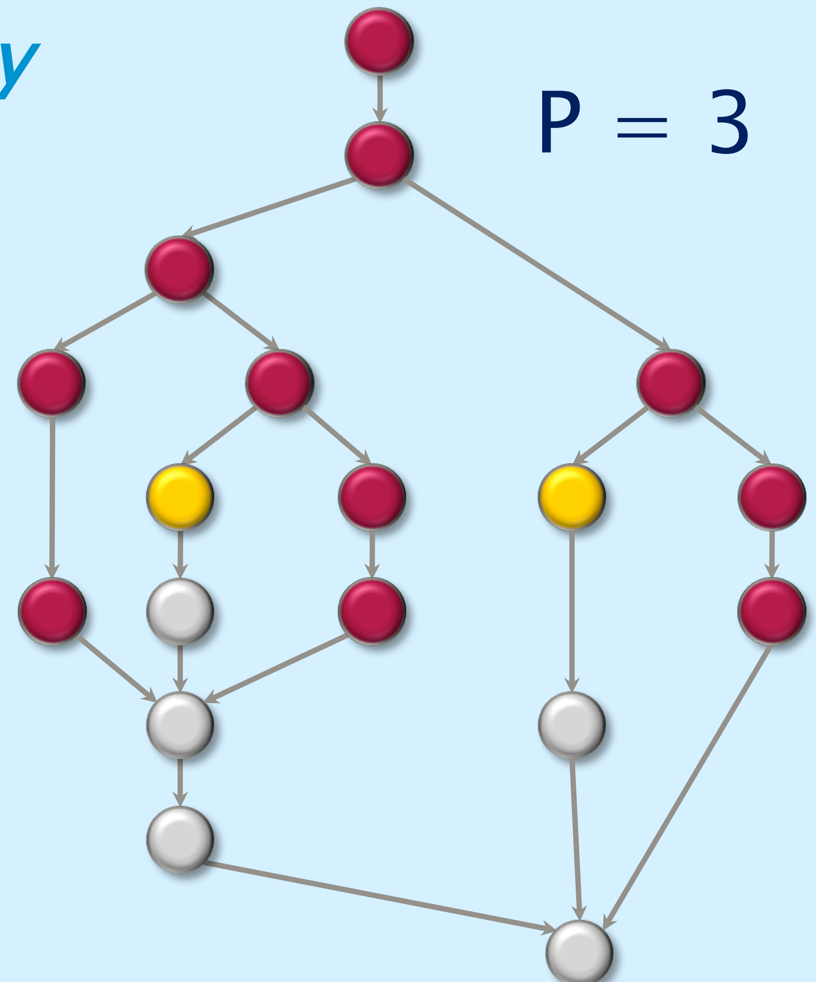
Definition: A strand is ***ready*** if all its predecessors have executed.



Greedy Scheduling

IDEA: Do as much as possible on every step.

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



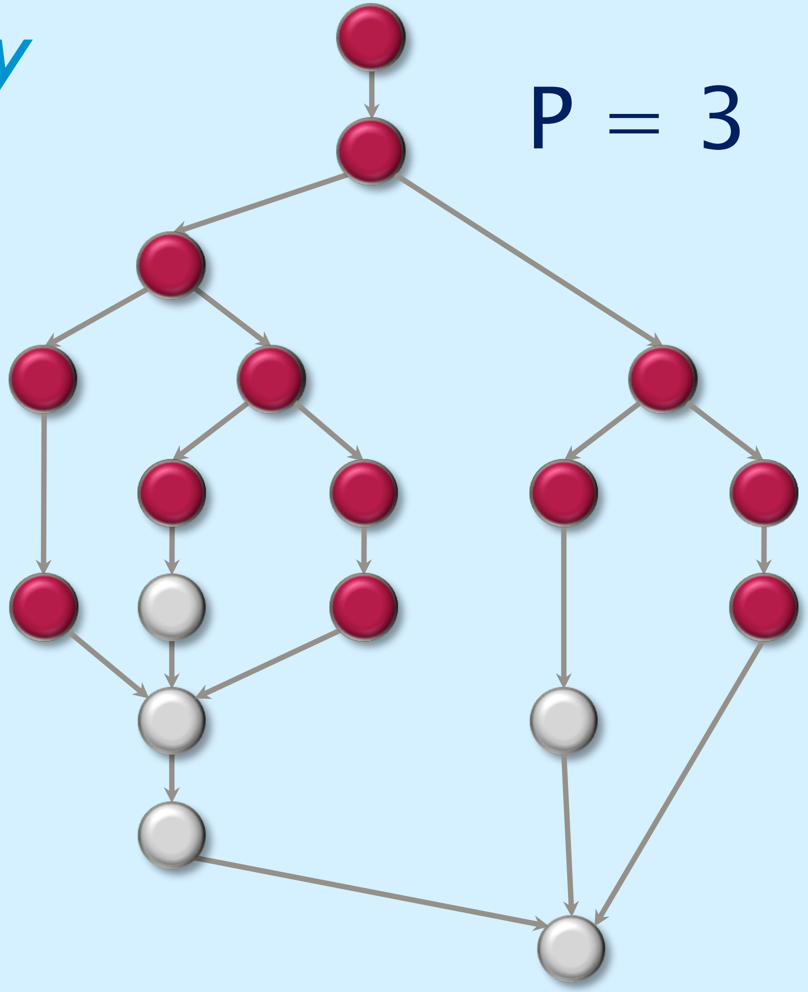
Complete step

- ≥ 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.

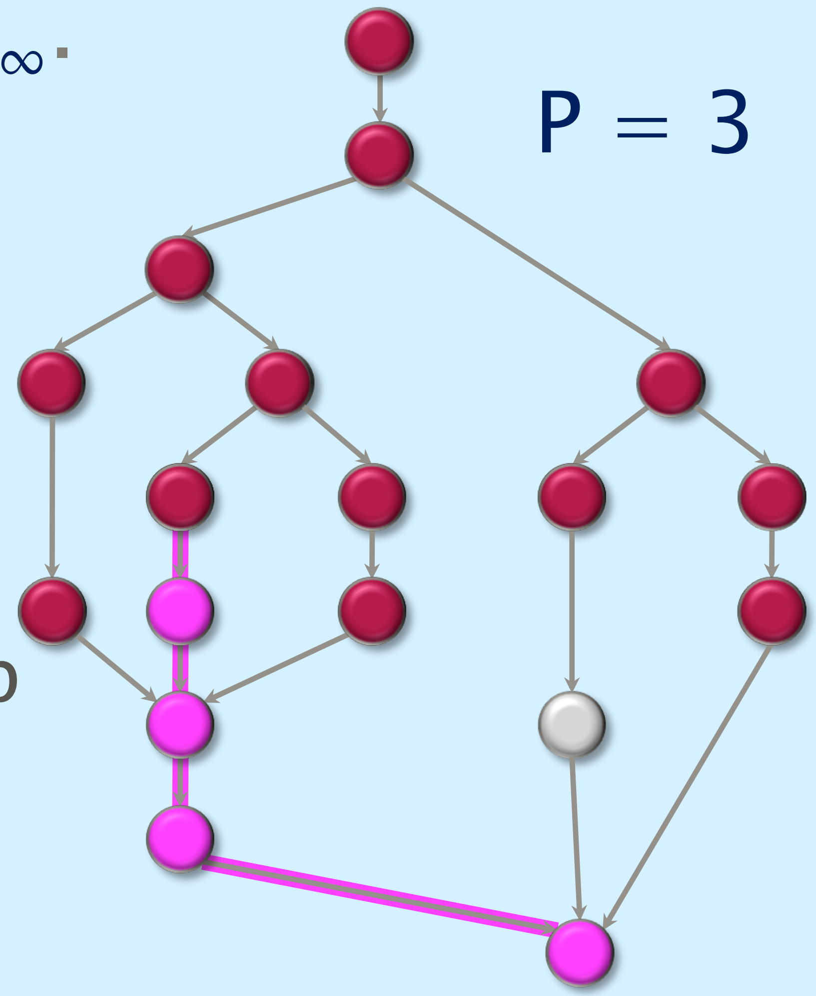
Incomplete step

- $<$ 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 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 $P \ll T_1/T_\infty$.

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 $T_1/T_P \approx P$. ■

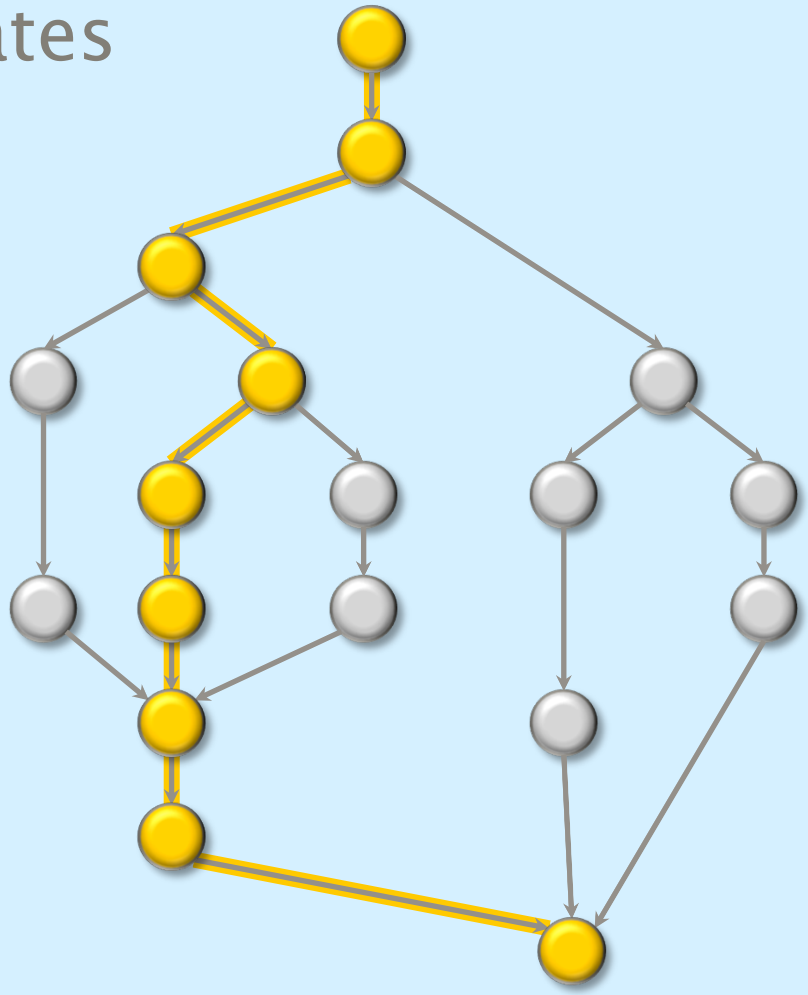
Definition. The quantity T_1/PT_∞ is called the ***parallel slackness***.

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_∞ = ***parallelism***

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



Great, how do we program it?

- Cilk++ is a faithful extension of C++
- Often use **divide-and-conquer**
- Three (really two) hints to the compiler:
  - cilk_spawn: *this function can run in parallel with the caller*
  - cilk_sync: *all spawned children must return before execution can continue*
  - cilk_for: *all iterations of this loop can run in parallel*
  - Compiler translates cilk_for into cilk_spawn & cilk_sync under the covers

Nested Parallelism

Example: 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;
  }
}
```

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

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

Cilk++ Loops

Example: Matrix transpose

```
cilk_for (int i=1; i<n; ++i) {
    cilk_for (int j=0; j<i; ++j) {
        B[i][j] = A[j][i];
    }
}
```

- A `cilk_for` loop's iterations execute in parallel.
- The index must be declared in the loop initializer.
- The end condition is evaluated exactly once at the beginning of the loop.
- Loop increments should be a **const** value

Serial Correctness

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

Cilk++ source

The *serialization* is the code with the Cilk++ keywords replaced by null or **C++** keywords.

Linker

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

Serialization

Binary

Serial correctness can be debugged and verified by running the multithreaded code on a single processor.

Cilk++ Runtime Library

Serialization

How to seamlessly switch between serial c++ and parallel cilk++ programs?

```
#ifdef CILKPAR
   #include <cilk.h>
#else
   #define cilk_for for
   #define cilk_main main
   #define cilk_spawn
   #define cilk_sync
#endif
```

Add to the beginning of your program

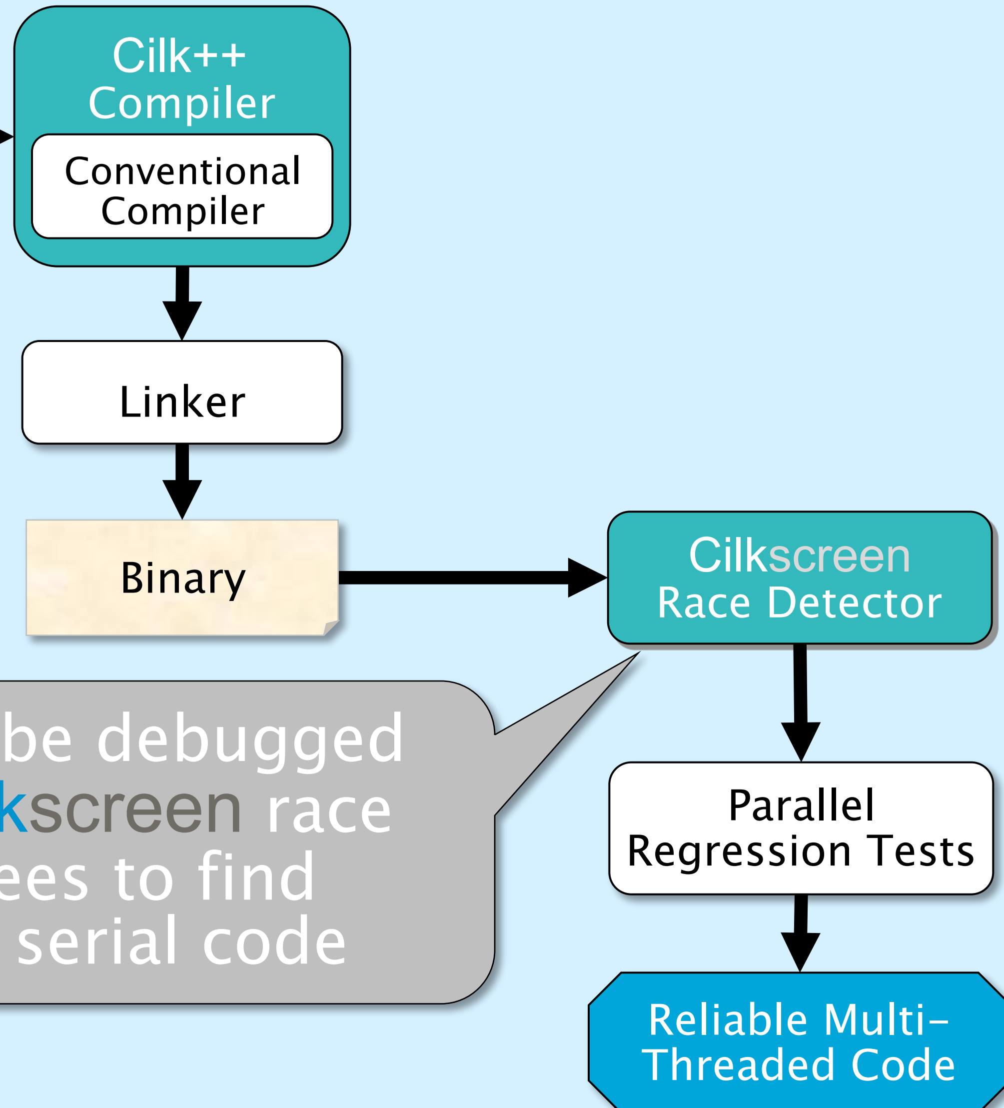
Compile !

- cilk++ -DCILKPAR -O2 -o parallel.exe main.cpp
- g++ -O2 -o serial.exe main.cpp

Parallel Correctness

```
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);
  }
}
```

Cilk++ source



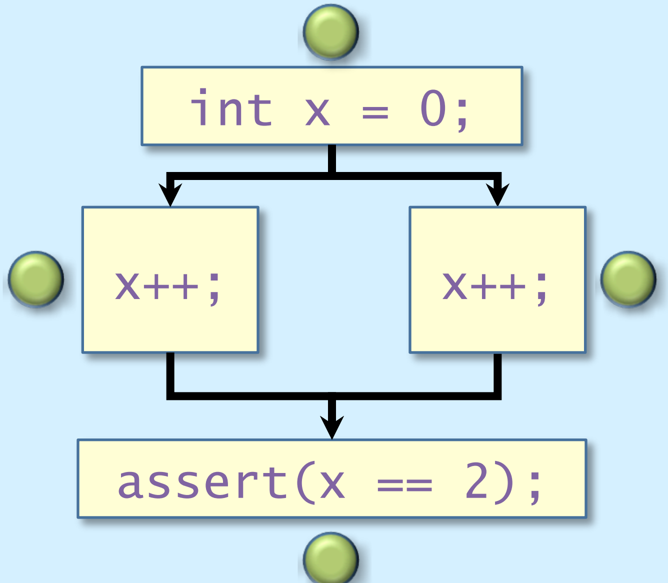
Parallel correctness can be debugged and verified with the Cilkscreen race detector, which guarantees to find inconsistencies with the serial code

Race Bugs

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

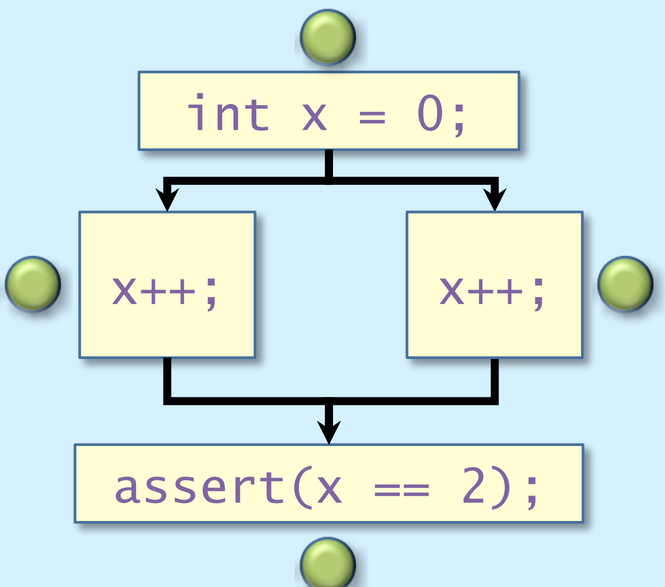
```
int x = 0;
cilk_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

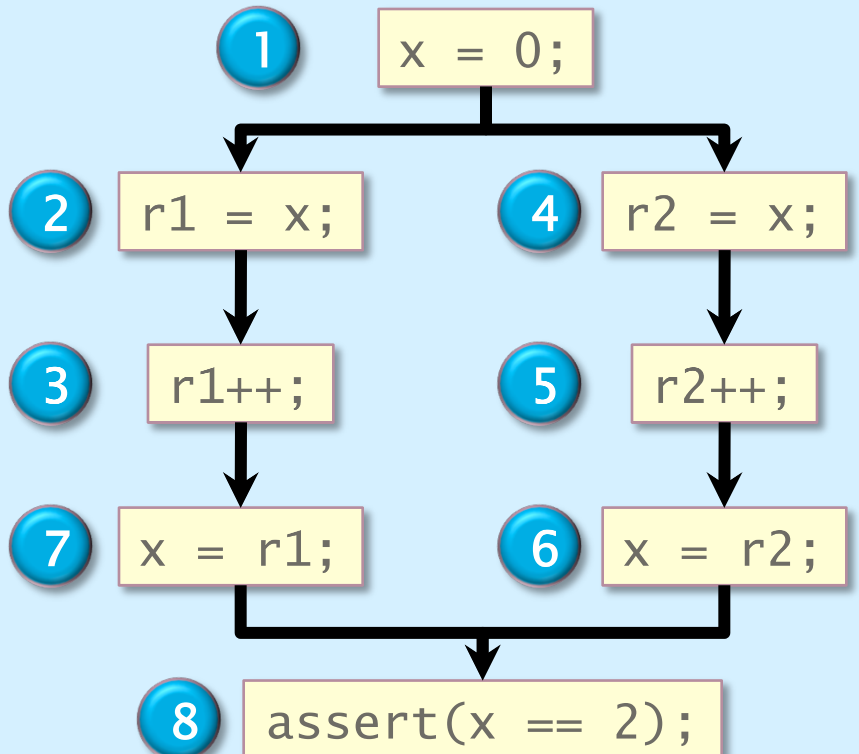


Dependency Graph

Race Bugs

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

- All the 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.

Ex.

```
cilk_spawn qsort(begin, middle);
qsort(max(begin + 1, middle), end);
cilk_sync;
```

Note: The arguments to a spawned function are evaluated in the parent before the spawn occurs.

Cilk++ Reducers

- Hyperobjects: **reducers**, holders, splitters
- Primarily designed as a solution to global variables, but has broader application

```
int result = 0;
cilk_for (size_t i = 0; i < N; ++i) {
    result += MyFunc(i);
}
```

Data race !

```
#include <reducer_opadd.h>
…
cilk::hyperobject<cilk::reducer_opadd<int> > result;
cilk_for (size_t i = 0; i < N; ++i) {
    result() += MyFunc(i);
}
```

Race free !

This uses one of the predefined reducers, but you can also write your own reducer easily

Hyperobjects under the covers

- A reducer **hyperobject**<T> includes an associative_binary operator ⊗ and an identity element.
- Cilk++ runtime system gives each thread a private view of the global variable
- When threads synchronize, their private views are combined with ⊗

Cilkscreen

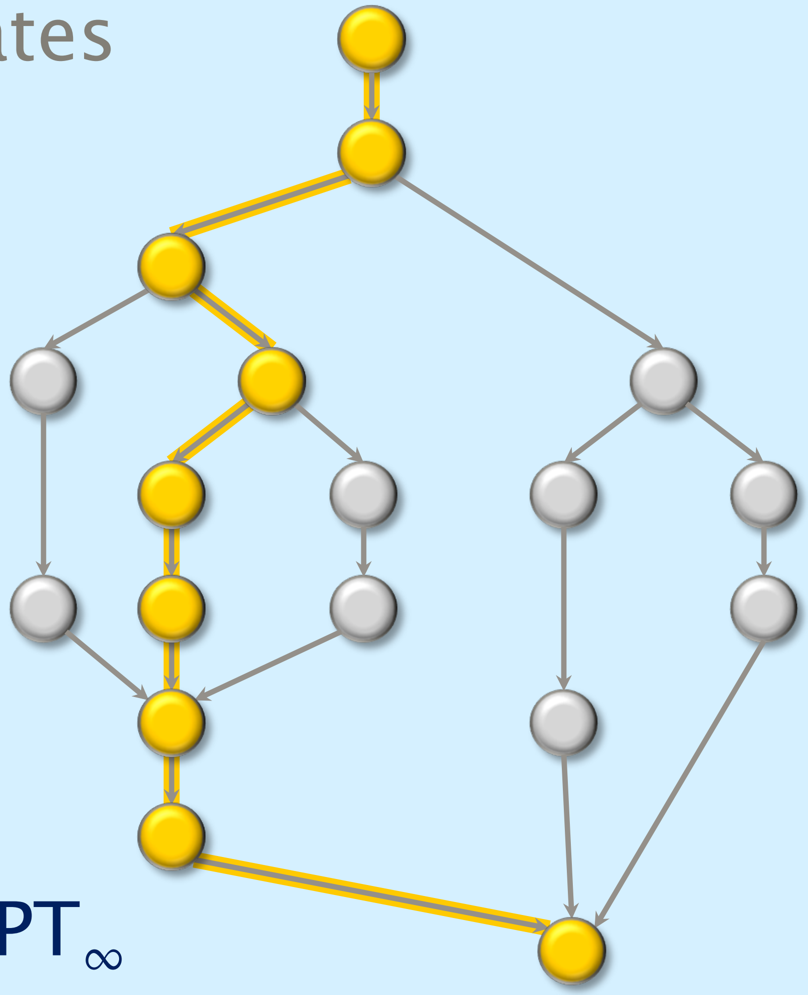
- Cilkscreen runs off the binary executable:
  - Compile your program with `-fcilkscreen`
  - Go to the directory with your executable and say `cilkscreen` *`your_program`* `[`*`options`*`]`
  - Cilkscreen prints info about any races it detects
- Cilkscreen guarantees to report a race if there exists a parallel execution that could produce results different from the serial execution.
- It runs about 20 times slower than single-threaded real-time.

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_∞ = ***parallelism***

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



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

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

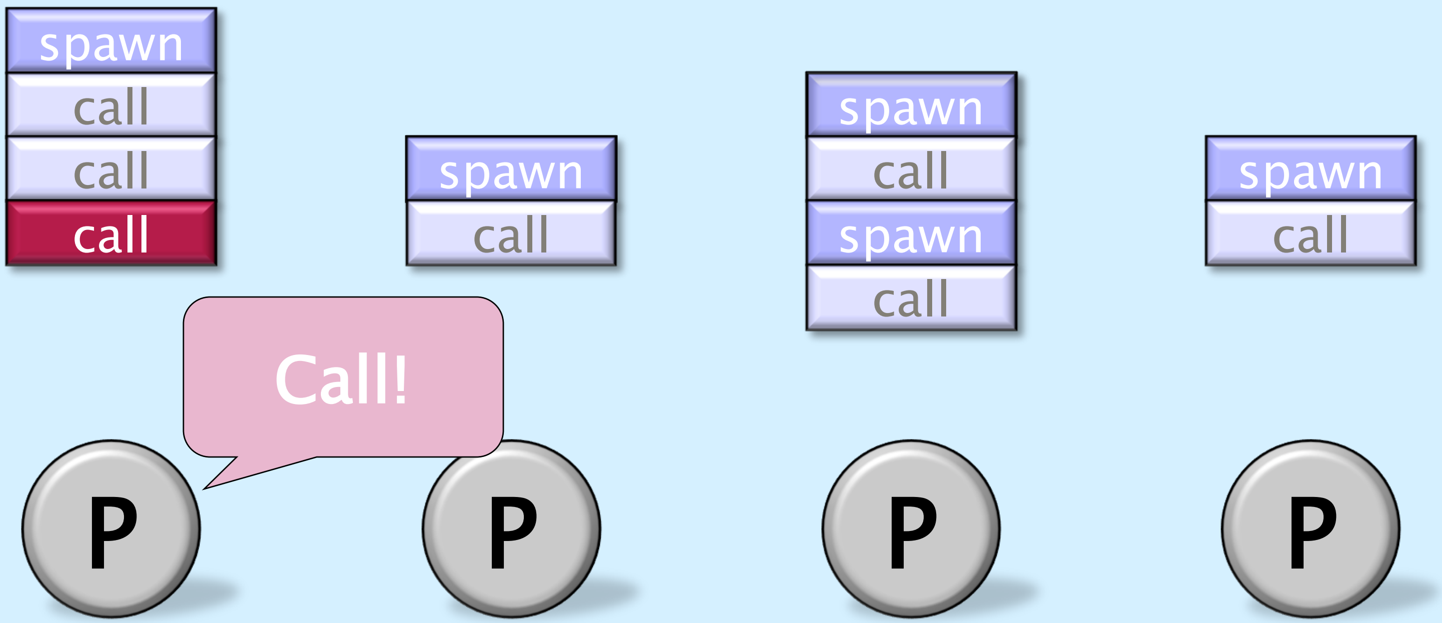
1. Make sure that work/#spawns is not too small.
   - Coarsen by using function calls and *inlining* near the leaves of recursion rather than spawning.

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.

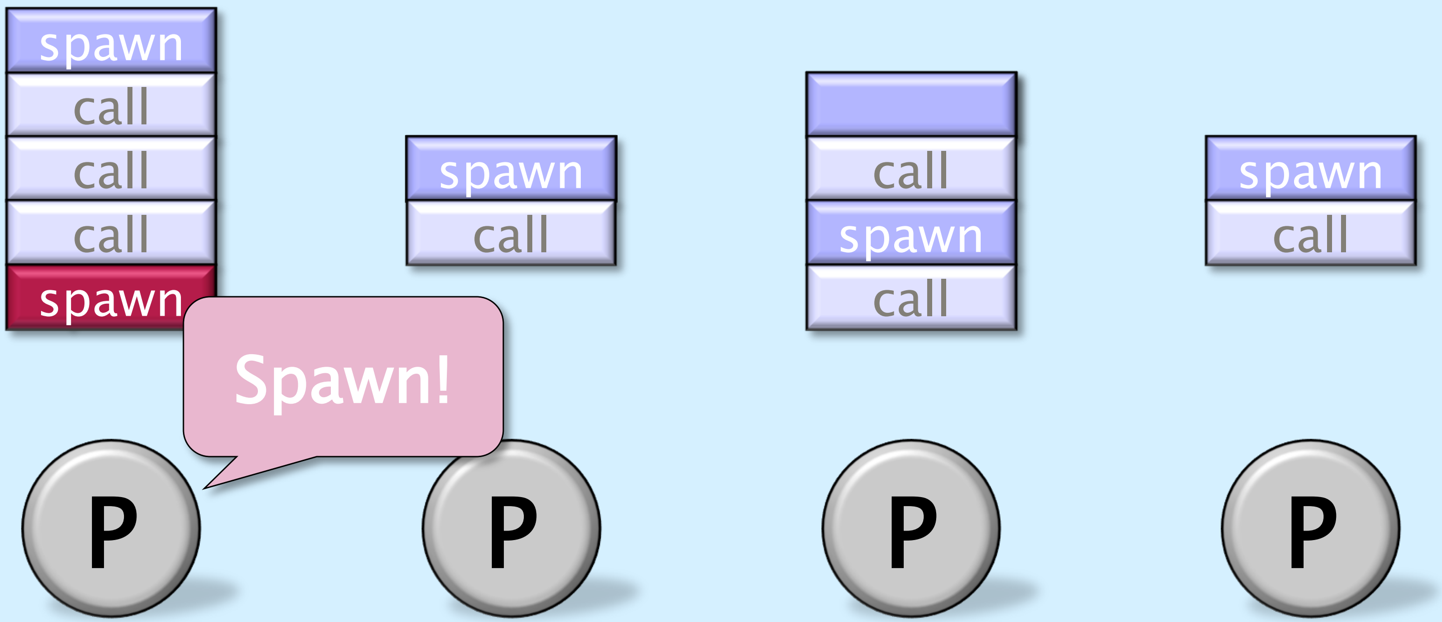
Cilk++ Runtime System

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



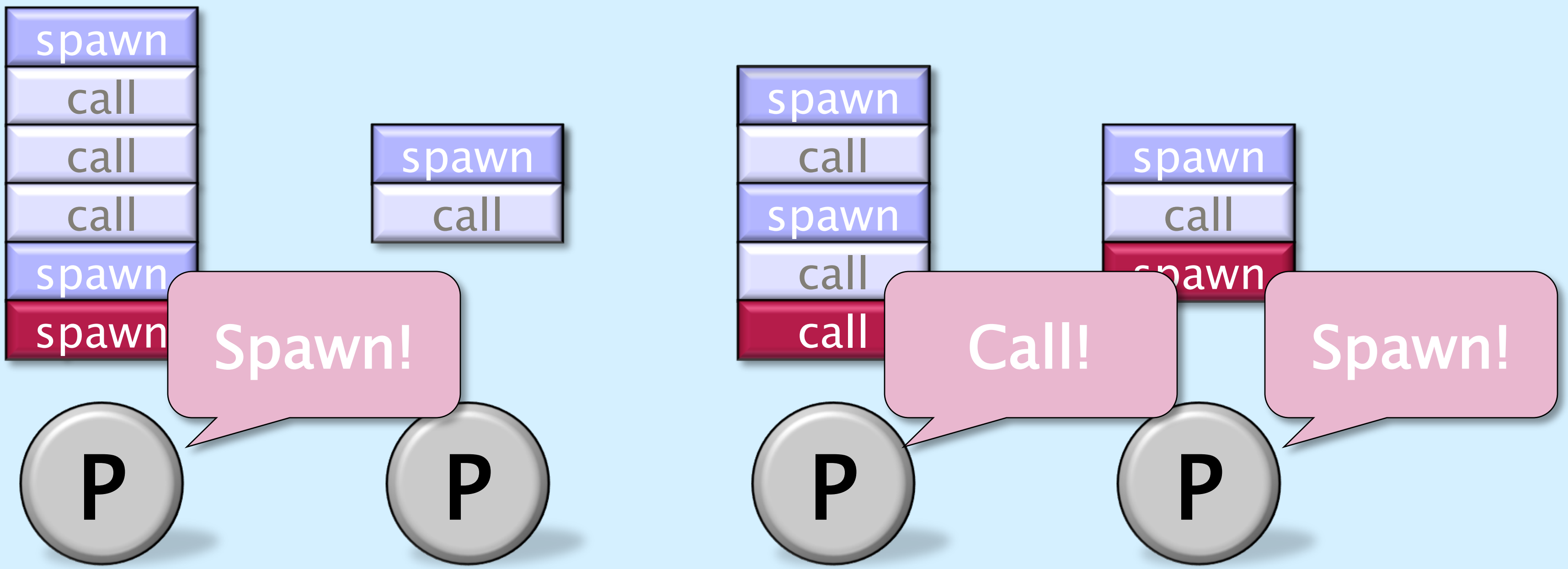
Cilk++ Runtime System

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



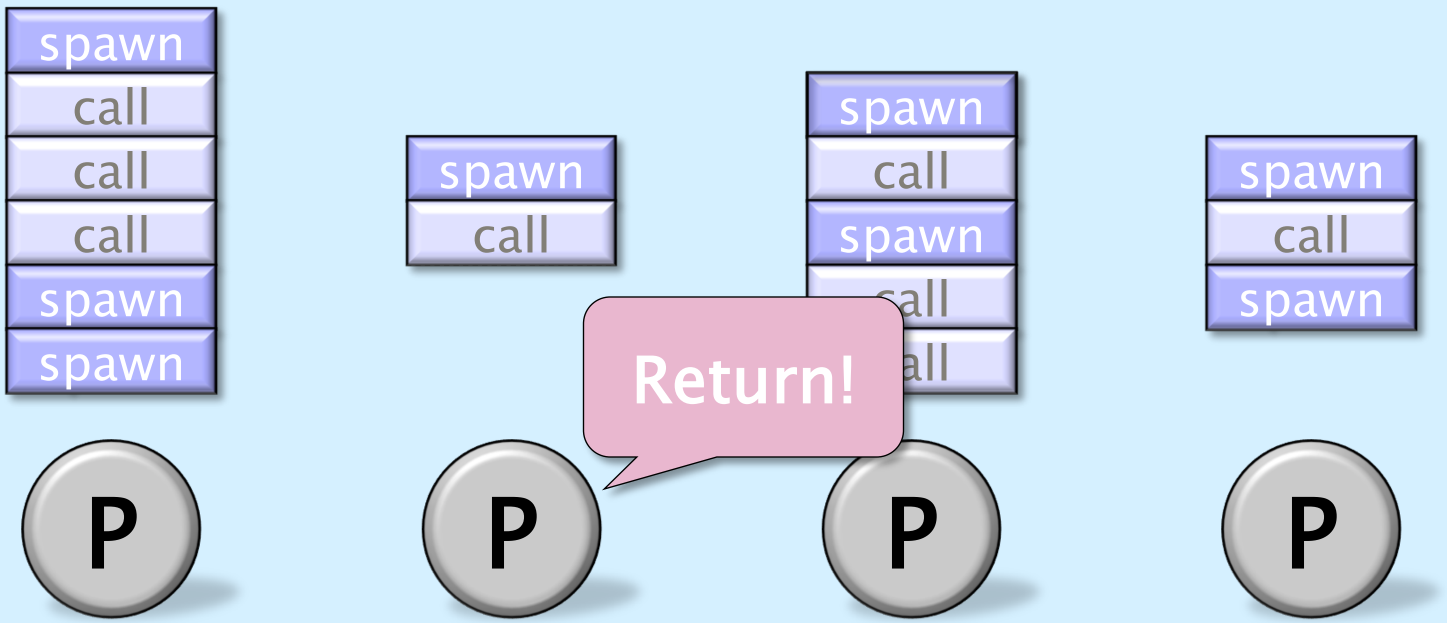
Cilk++ Runtime System

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



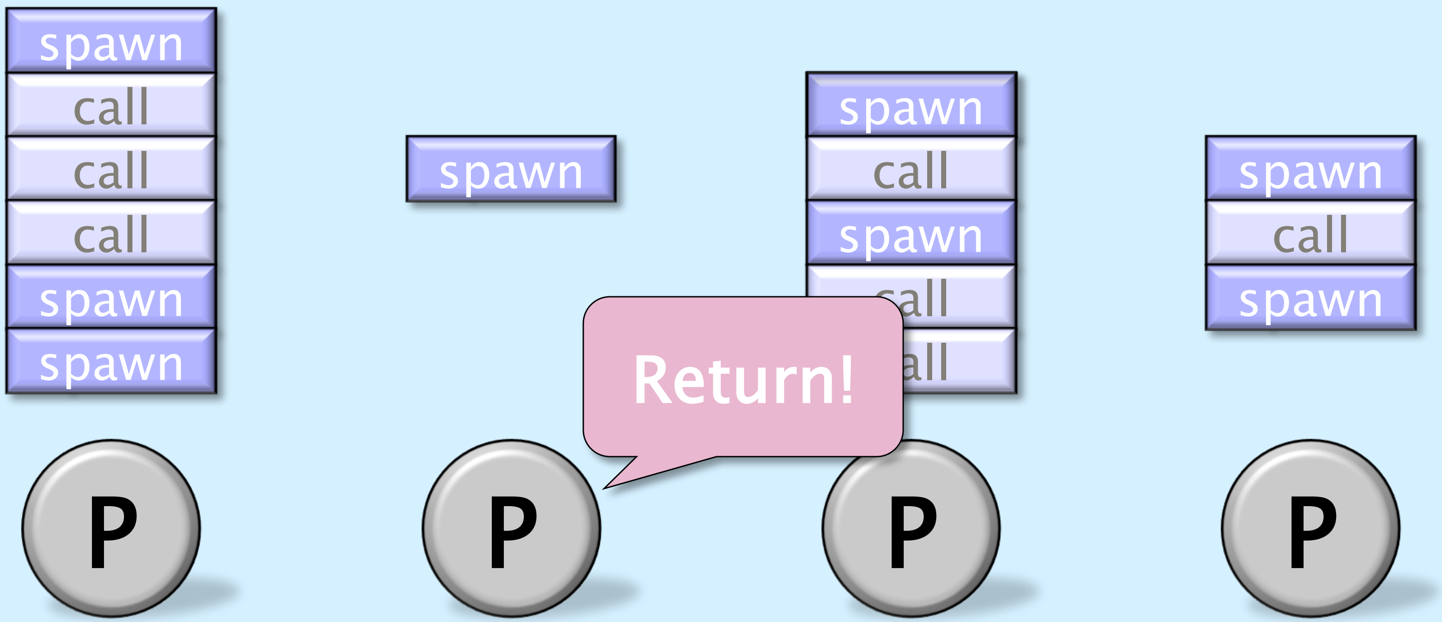
Cilk++ Runtime System

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



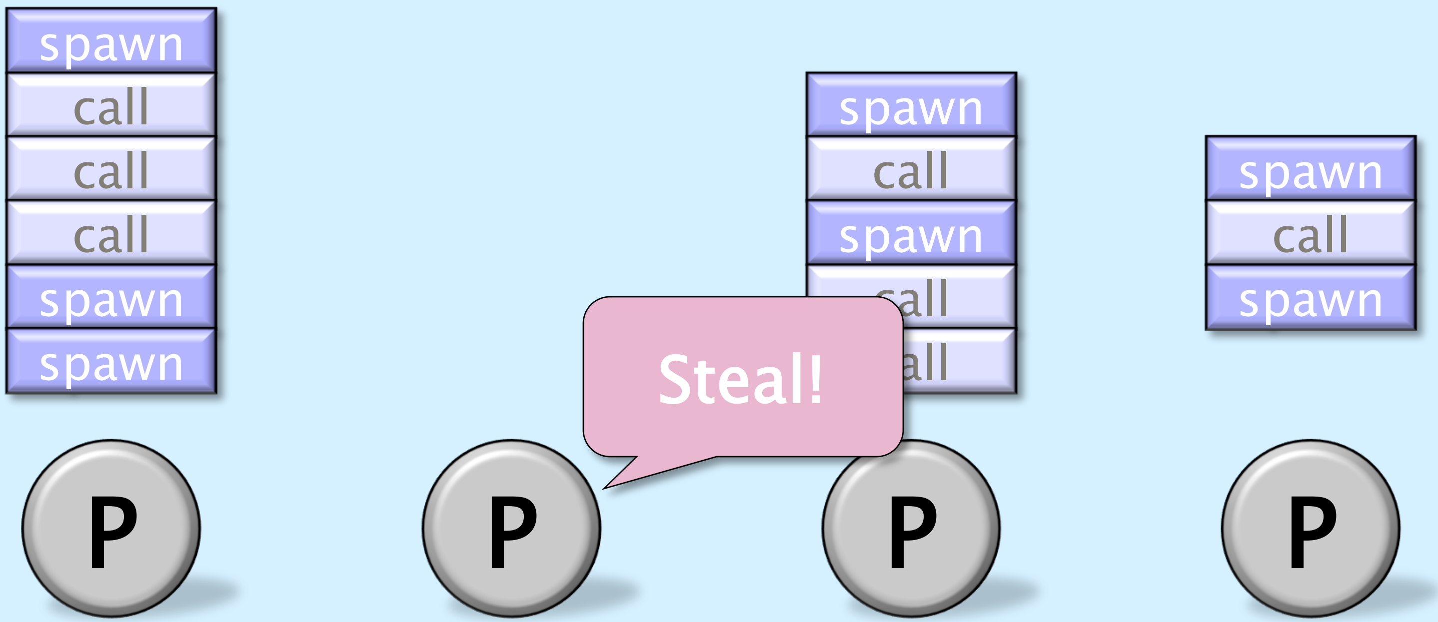
Cilk++ Runtime System

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

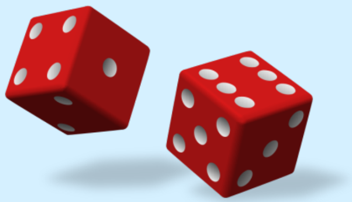


Cilk++ Runtime System

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

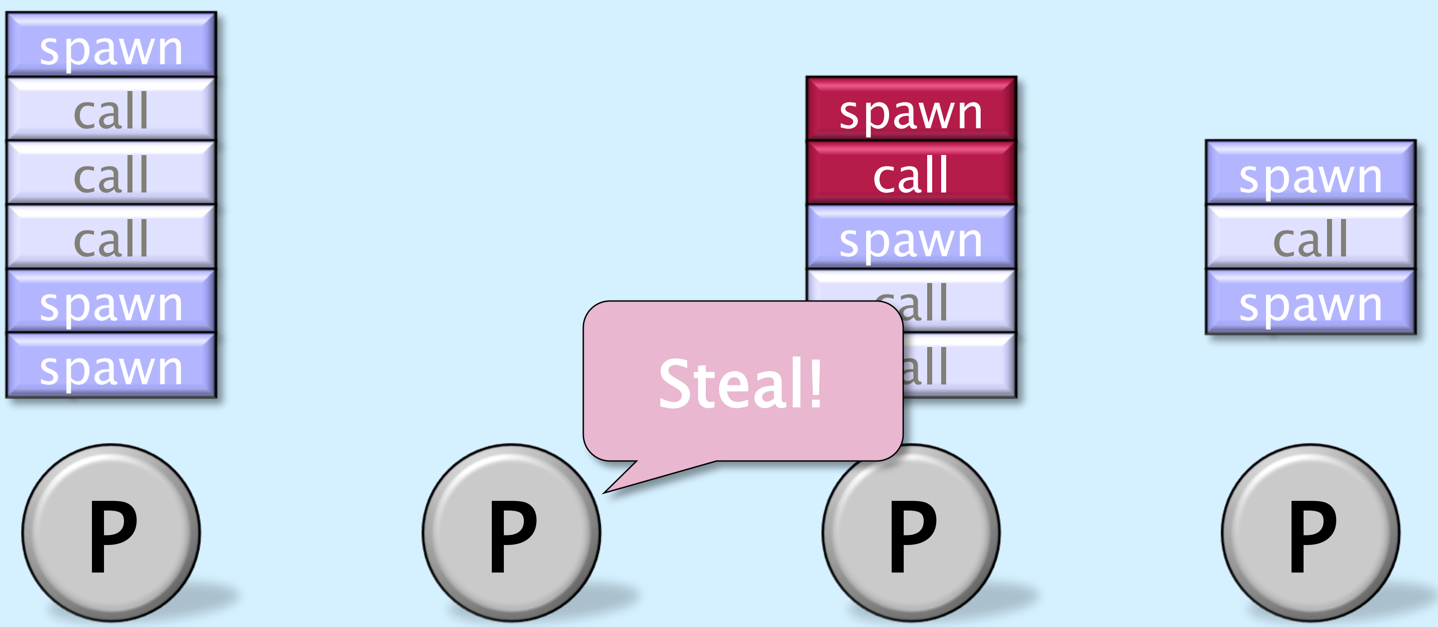


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.

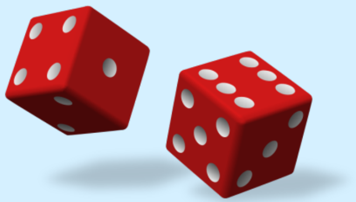


Cilk++ Runtime System

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

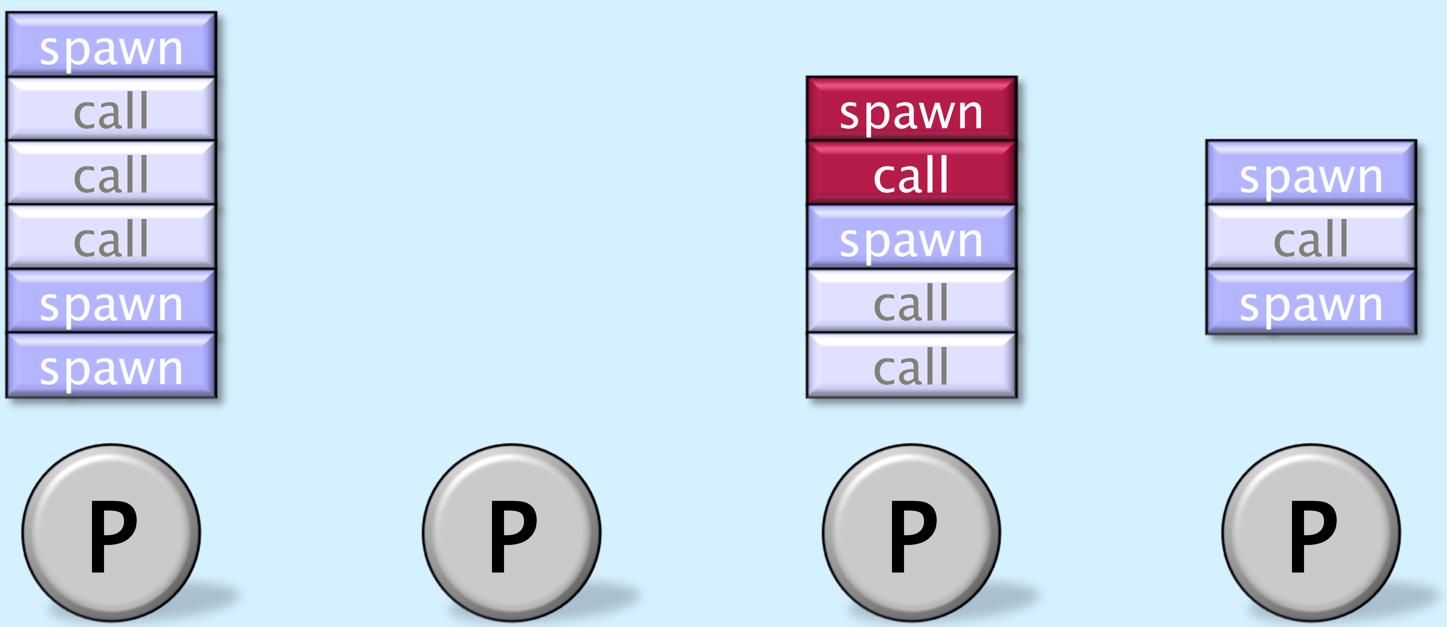


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.

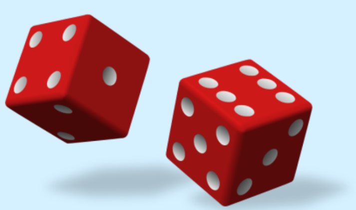


Cilk++ Runtime System

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

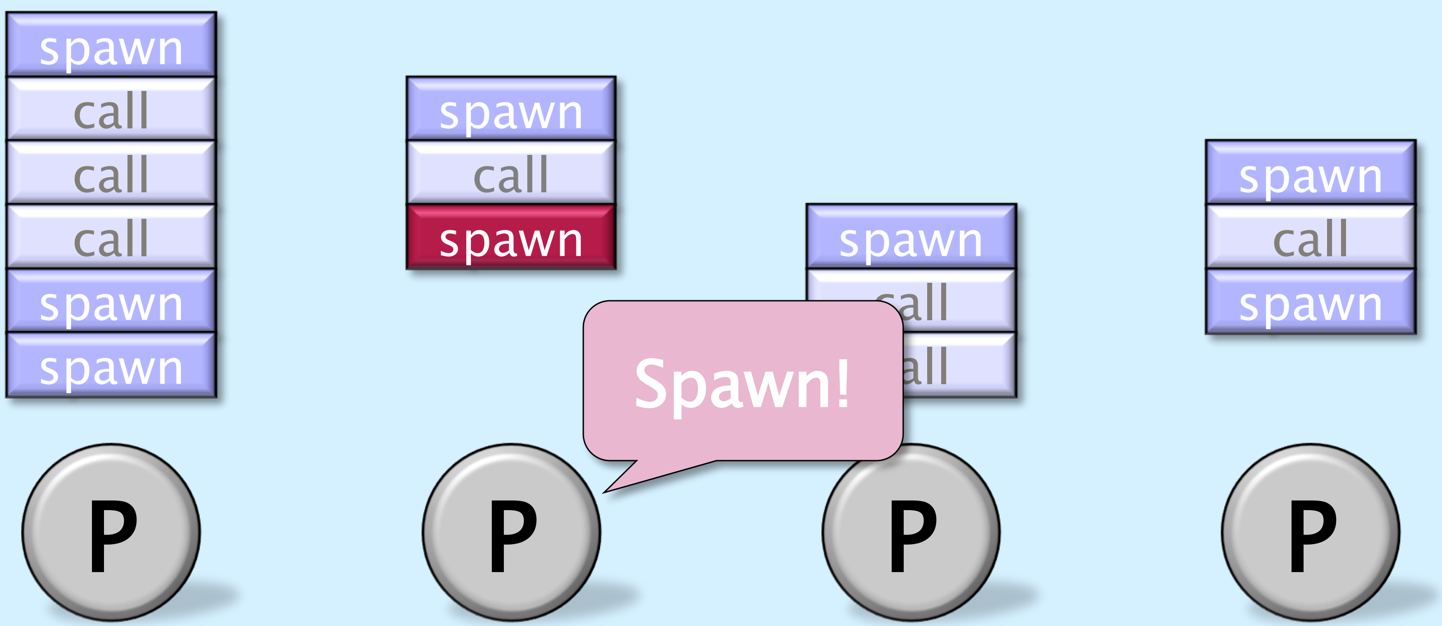


When a worker runs out of work, it ***steals*** from the top of a ***random*** victim's deque.



Cilk++ Runtime System

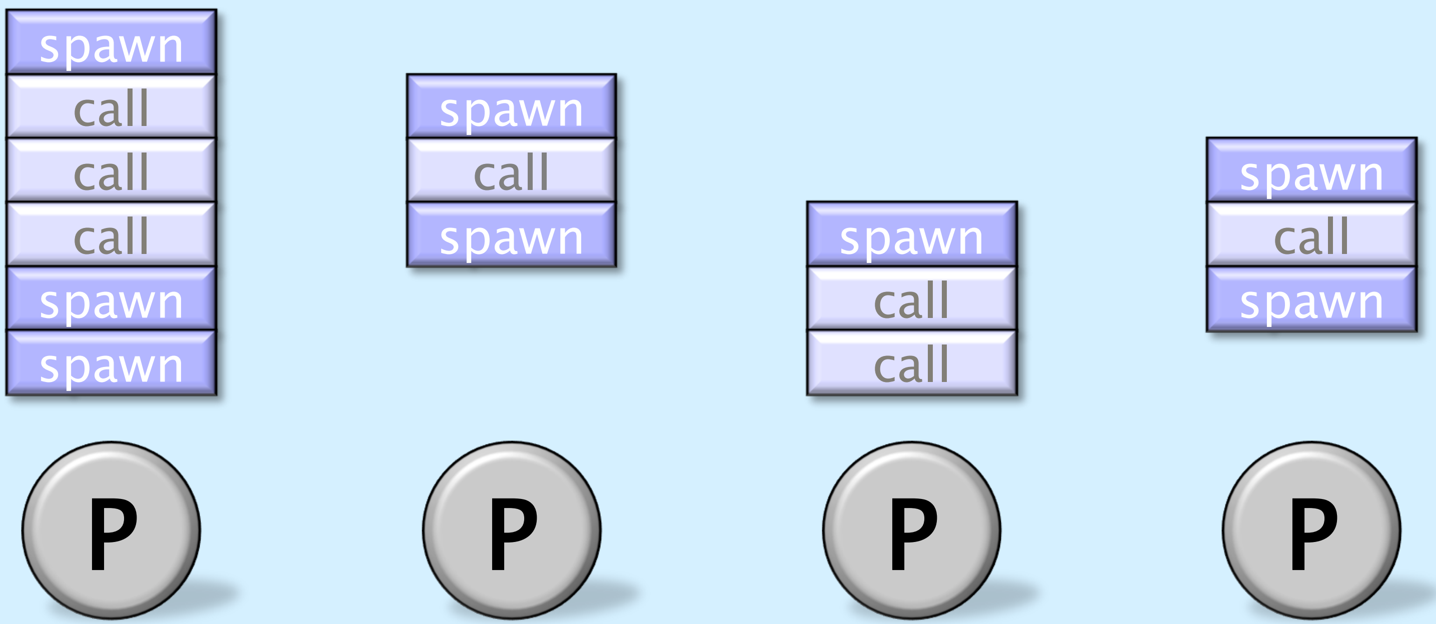
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



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

Cilk++ Runtime System

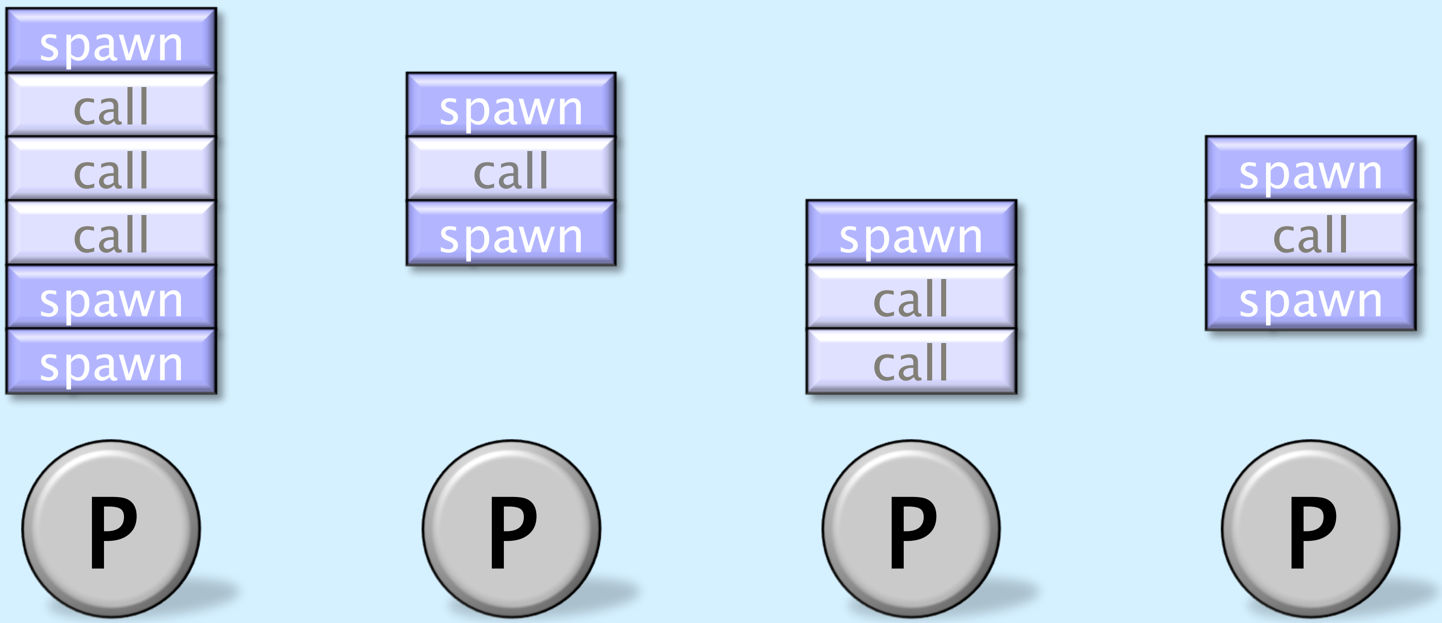
Each worker (processor) maintains a ***work deque*** of ready strands, and it manipulates the bottom of the deque like a stack



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***.