SALVATORE DI GIROLAMO <DIGIROLS@INF.ETHZ.CH>

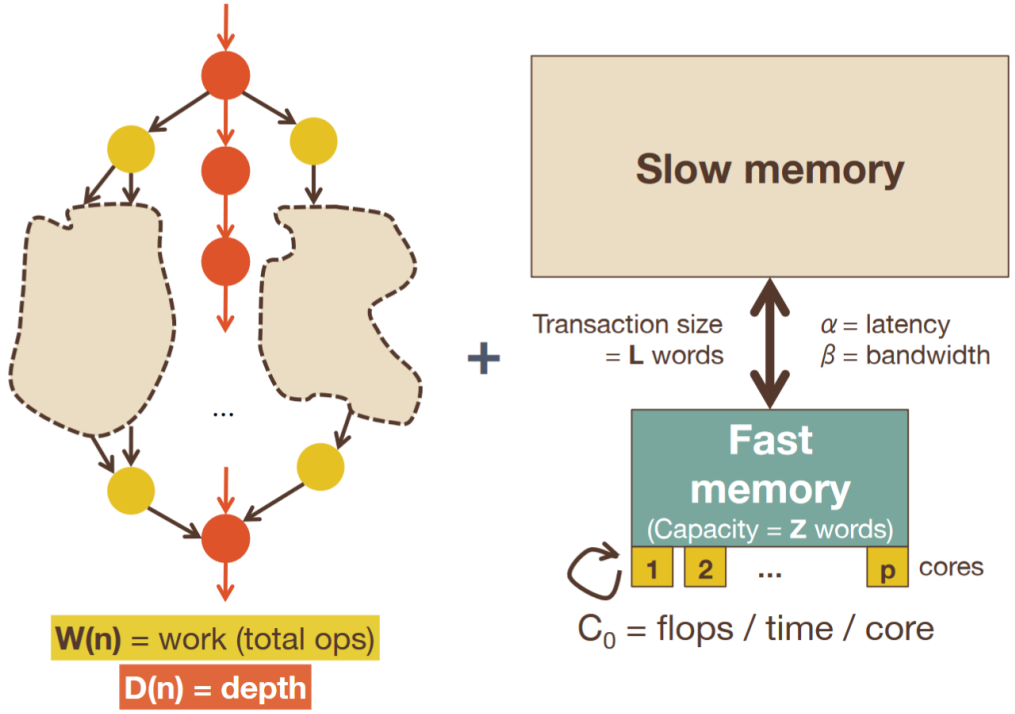
DPHPC: Balance Principles & Scheduling

Recitation session

Deriving a Balance Principle

- **Concept of balance:** a computation running on some machine is efficient if the compute-time dominates the I/O time. *[Kung, 1986]*

- **Deriving a balance principle:**
  - Algorithmically analyze the parallelism
  - Algorithmically analyze the I/O behavior (i.e., number of memory transfers)
  - Combine these two analyses with a cost model for an abstract machine.

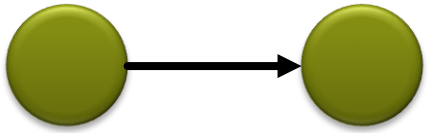


- **Goal: say precisely and analytically how**
  - Changes to the architecture might affect the scaling of a computation
  - Identify what classes of computation might execute efficiently on a given architecture

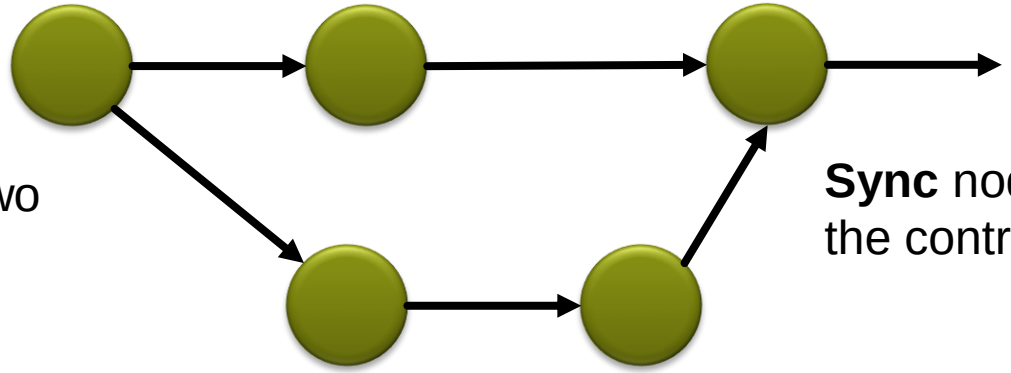
Czechowski, Kent, et al. "Balance Principles for Algorithm-Architecture Co-Design." HotPar 11 (2011): 9-9.

The DAG Model

Strand: chain of serially executed instructions.



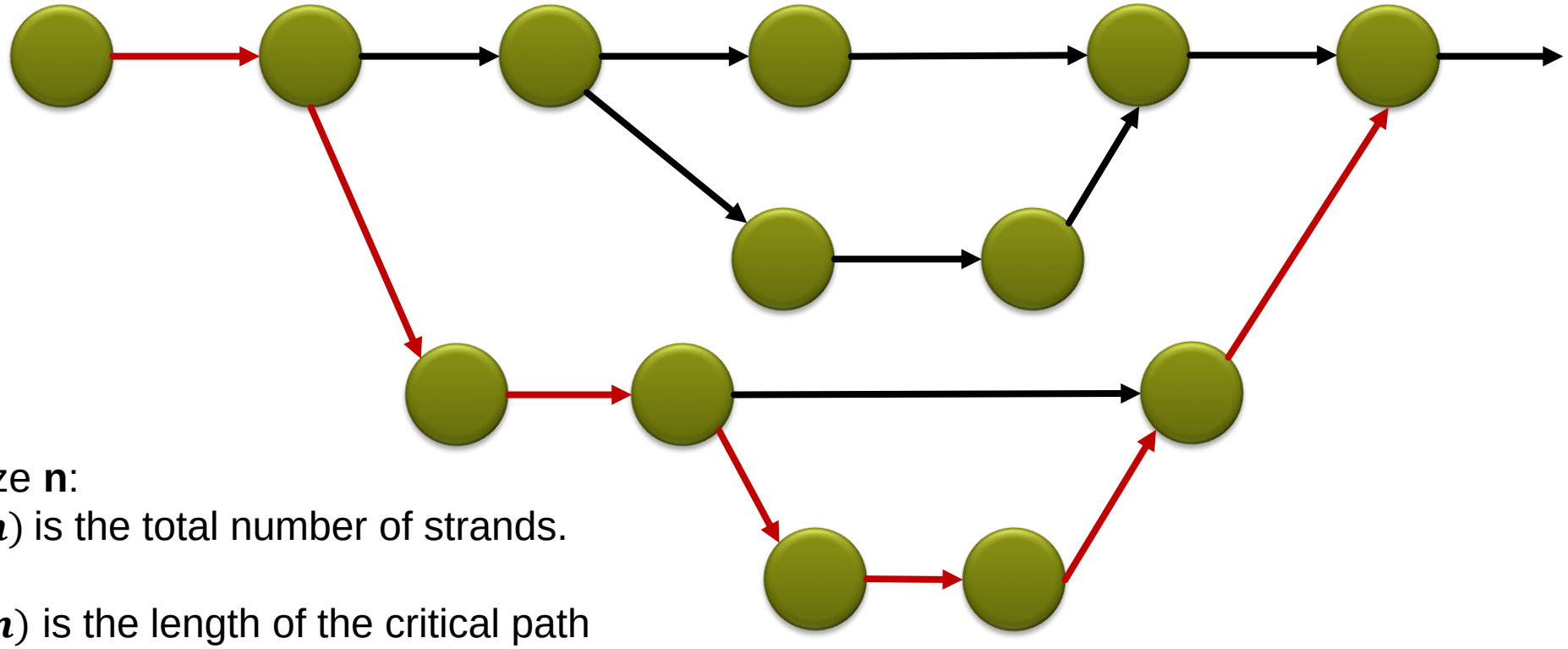
Strands are partially ordered with **dependencies**



Spawn nodes have two successors

Sync nodes are where the control flow merges

The DAG Model



Given an input size **n**:

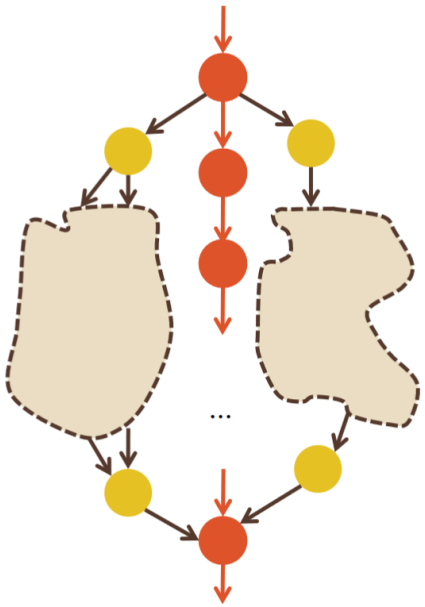
- The **work** $\boldsymbol{W(n)}$ is the total number of strands.
  - W(n)=13
- The **depth** $\boldsymbol{D(n)}$ is the length of the critical path (measured in number of strands).
  - Defines the minimum execution time of the computation
  - D(n)=8

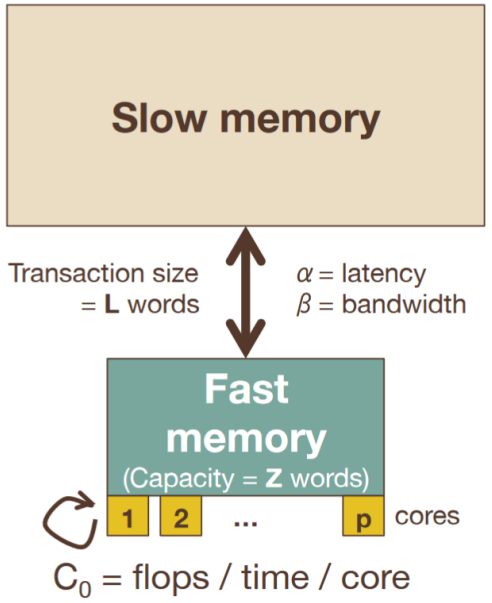
The ratio $\frac{W(n)}{D(n)}$ measures the average available parallelism

Analyzing I/Os

- **We use the classical external memory model**
- **Two level memory**
  - One large&slow
  - The other small&fast (capacity: Z words)

  It can be an automatic cache or a software-controlled scratchpad
- **Work operations can be performed only on data in fast memory**
- **Slow<->Fast memory transfers occur in blocks of L words**
- **$Q_{Z,L}(n)$ is the number of L-sized transfers between slow and fast memory for an input of size n**

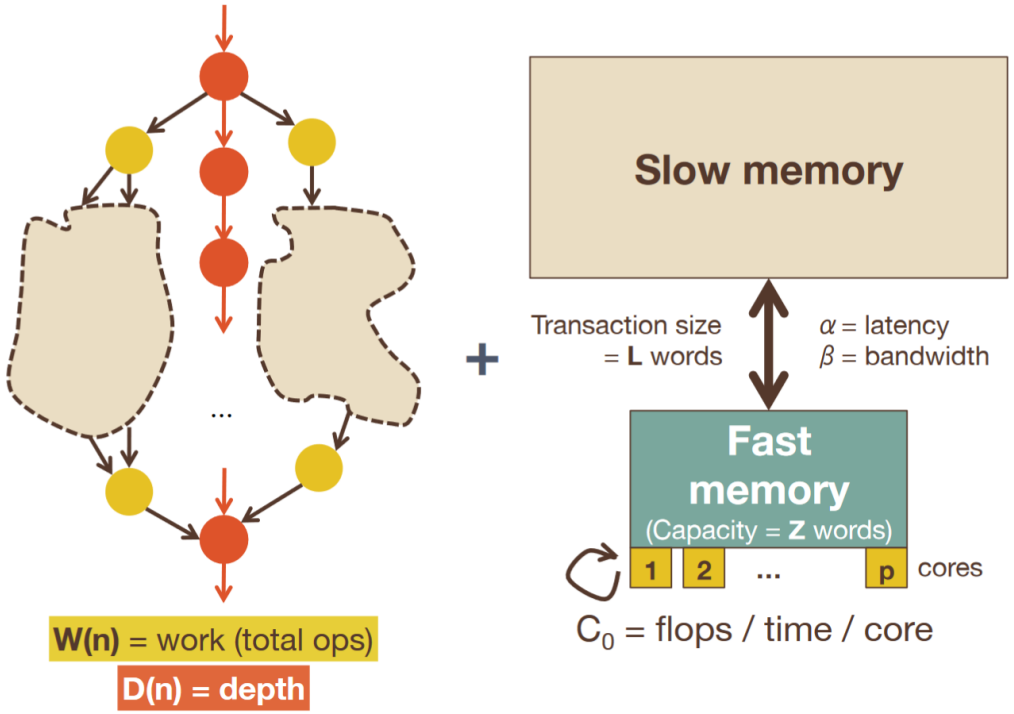




Goal is to optimize the computational intensity: $\frac{W(n)}{Q_{Z,L}(n)\cdot L}$

Architecture-Specific Cost Model

- **We need to introduce the time**
  - This depends on the specific architecture
- **p cores**
- **Each core can deliver C_0 operations per unit time**
- **The time to transfer $m \cdot L$ words is:**
  - $\alpha + m \cdot L/\beta$
  - α is the latency
  - β is the bandwidth in units of words per time

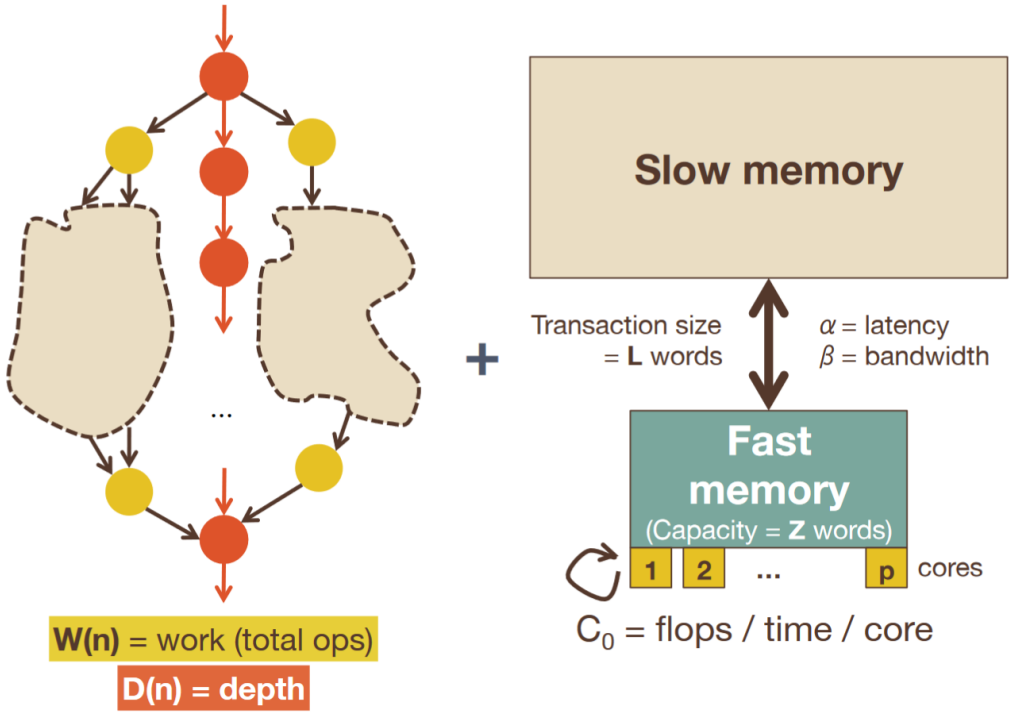


- **The best possible compute time is (Brent's theorem):**

$$T_{\text{comp}}(n; p, C_0) = \left(D(n) + \frac{W(n)}{p} \right) \cdot \frac{1}{C_0}$$

Architecture-Specific Cost Model

- $Q_{Z,L}(n)$ **is for the sequential case**
- **We need to move to the parallel case** $Q_{p;Z,L}(n)$
  - We can bound $Q_{p;Z,L}(n)$ in terms of $Q_{Z,L}(n)$
    Blelloch et al, 2009, need to select a specific scheduler
  - Compute it directly
- **Assumptions:**
  - the latency is accounted for each node in the critical path
  - all the $Q_{p;Z,L}(n)$ are aggregated and pipelined by the memory system
    Hence they are delivered at the peak bandwidth



- **We can estimate the memory cost as:**

$$T_{\text{mem}}(n; p, Z, L, \alpha, \beta) = \alpha \cdot D(n) + \frac{Q_{p;Z,L}(n) \cdot L}{\beta}$$

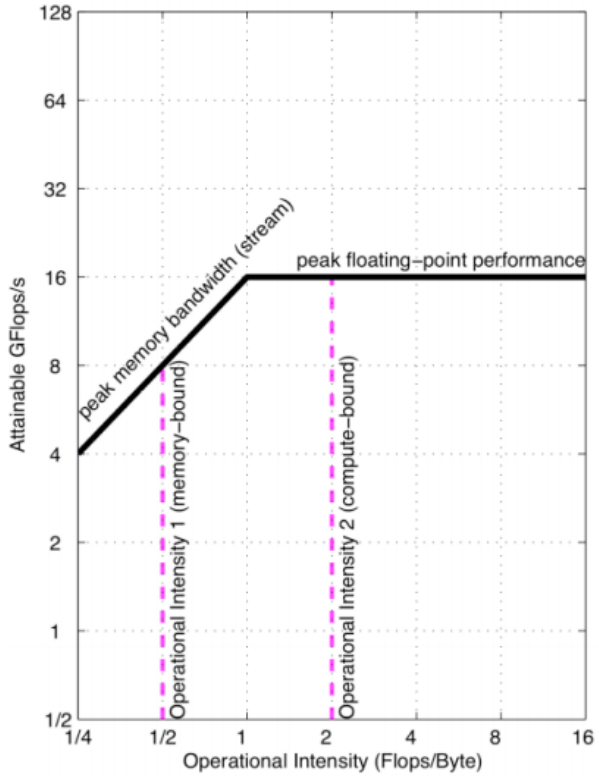
The Balance Principle

- **The balance principle follows by imposing $T_{mem} \leq T_{comp}$**

$$T_{\text{mem}}(n; p, Z, L, \alpha, \beta) = \alpha \cdot D(n) + \frac{Q_{p;Z,L}(n) \cdot L}{\beta}$$

$$T_{\text{comp}}(n; p, C_0) = \left(D(n) + \frac{W(n)}{p} \right) \cdot \frac{1}{C_0}$$

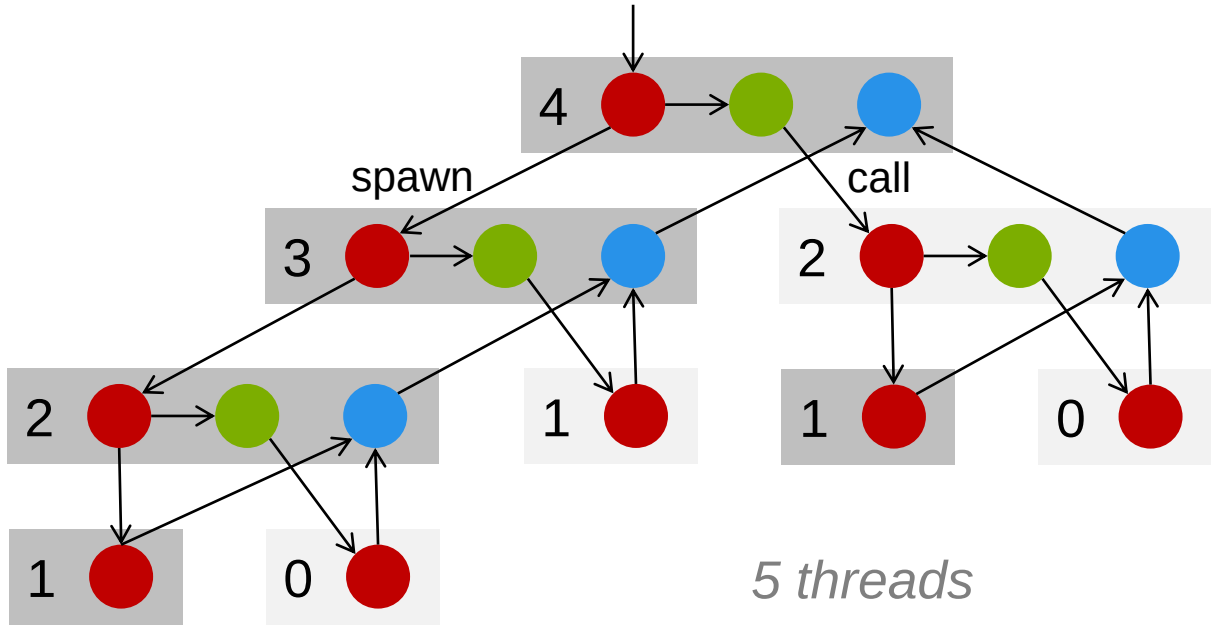
$$\underbrace{\frac{pC_0}{\beta}}_{\text{balance}} \left(1 + \underbrace{\frac{\alpha\beta/L}{Q/D}}_{\text{Little's}} \right) \leq \underbrace{\frac{W}{QL}}_{\text{intensity}} \left(1 + \underbrace{\frac{p}{W/D}}_{\text{Amdahl's}} \right)$$



Scheduling

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

The DAG unfolds dynamically:

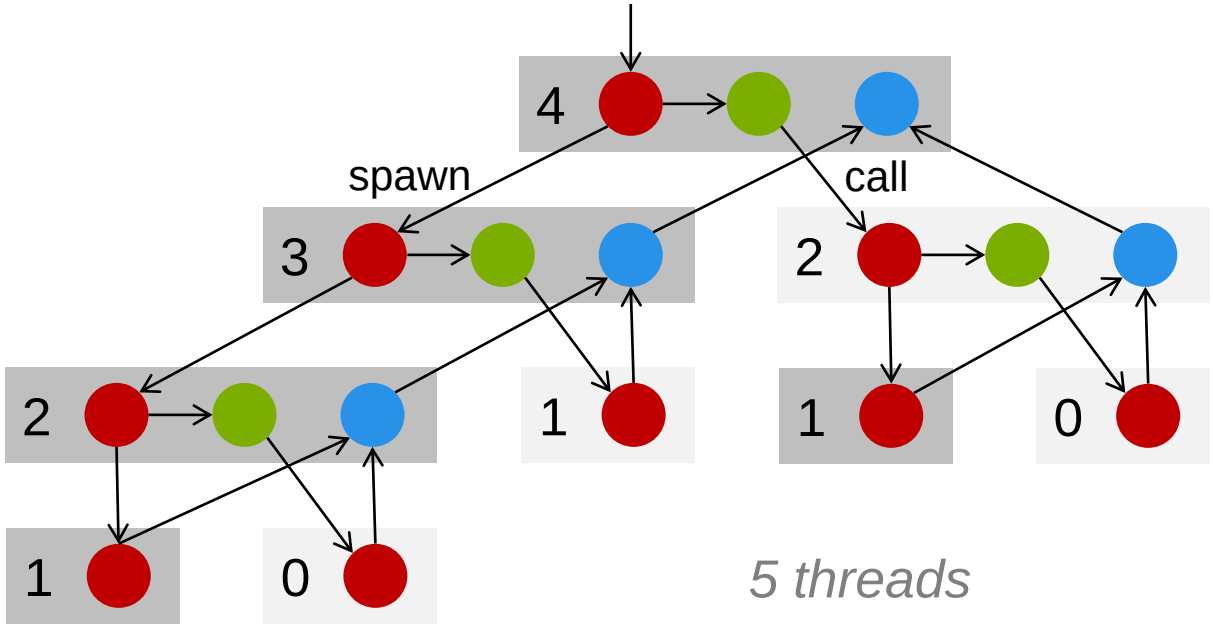


5 threads

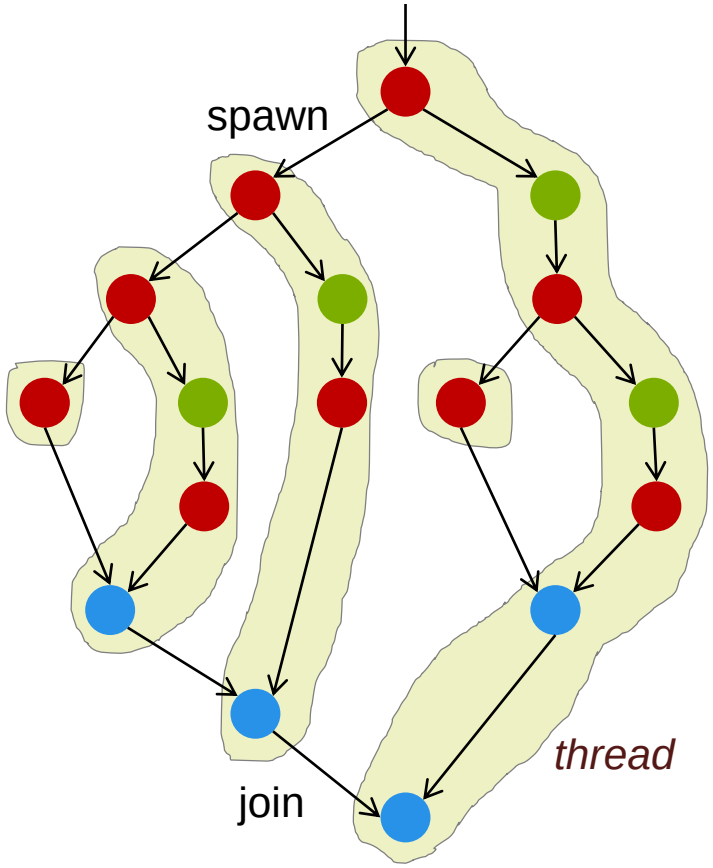
Node: Sequence of instructions without call, spawn, sync, return
Edge: Dependency

Scheduling

The DAG unfolds dynamically:

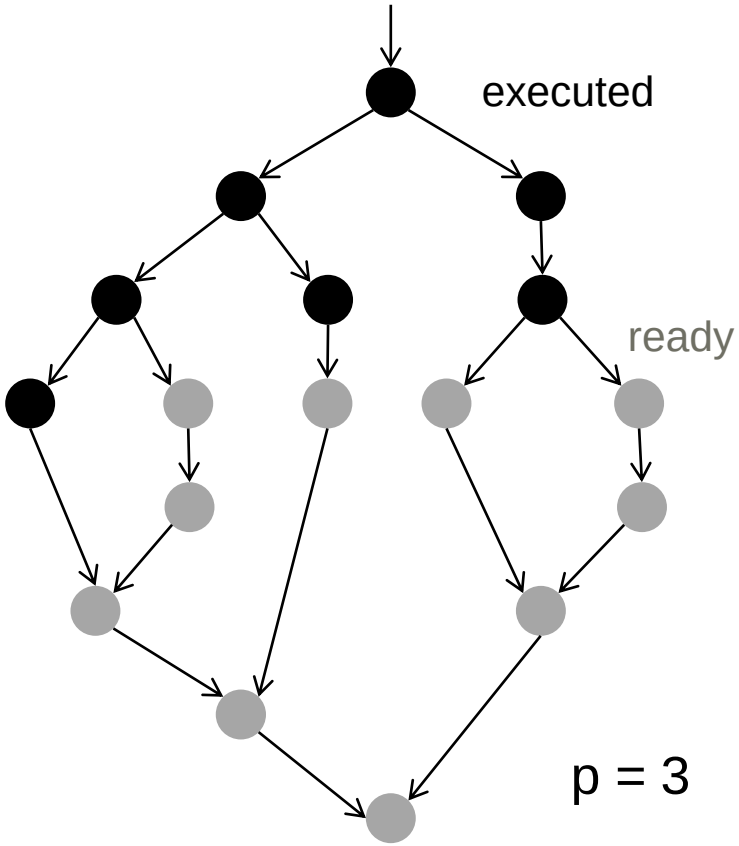


5 threads



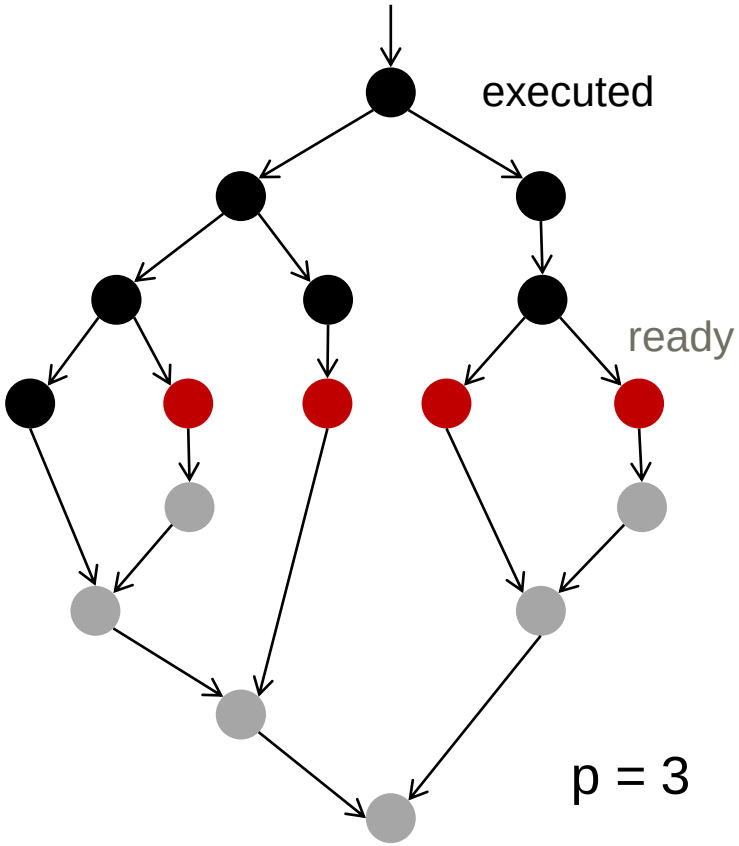
Greedy Scheduler

- *Idea:* **Do as much as possible in every step**
- *Definition:* **A node is ready if all predecessors have been executed**



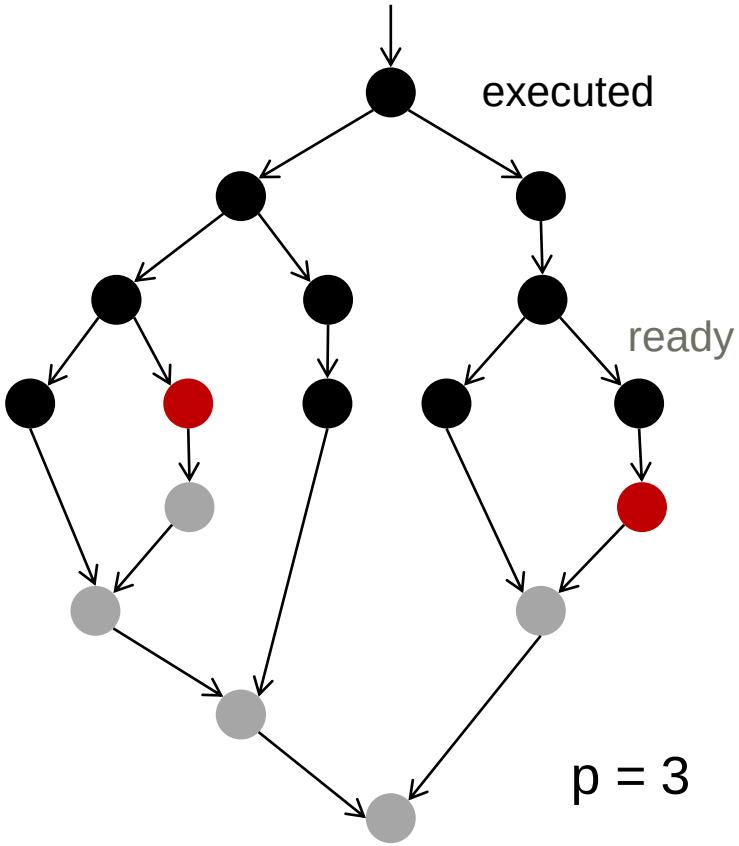
Greedy Scheduler

- *Idea:* **Do as much as possible in every step**
- *Definition:* **A node is ready if all predecessors have been executed**
- **Complete step:**
  - ≥ p nodes are ready
  - run any p



Greedy Scheduler

- *Idea:* **Do as much as possible in every step**
- *Definition:* **A node is ready if all predecessors have been executed**
- **Complete step:**
  - ≥ p nodes are ready
  - run any p
- **Incomplete step:**
  - < p nodes ready
  - run all



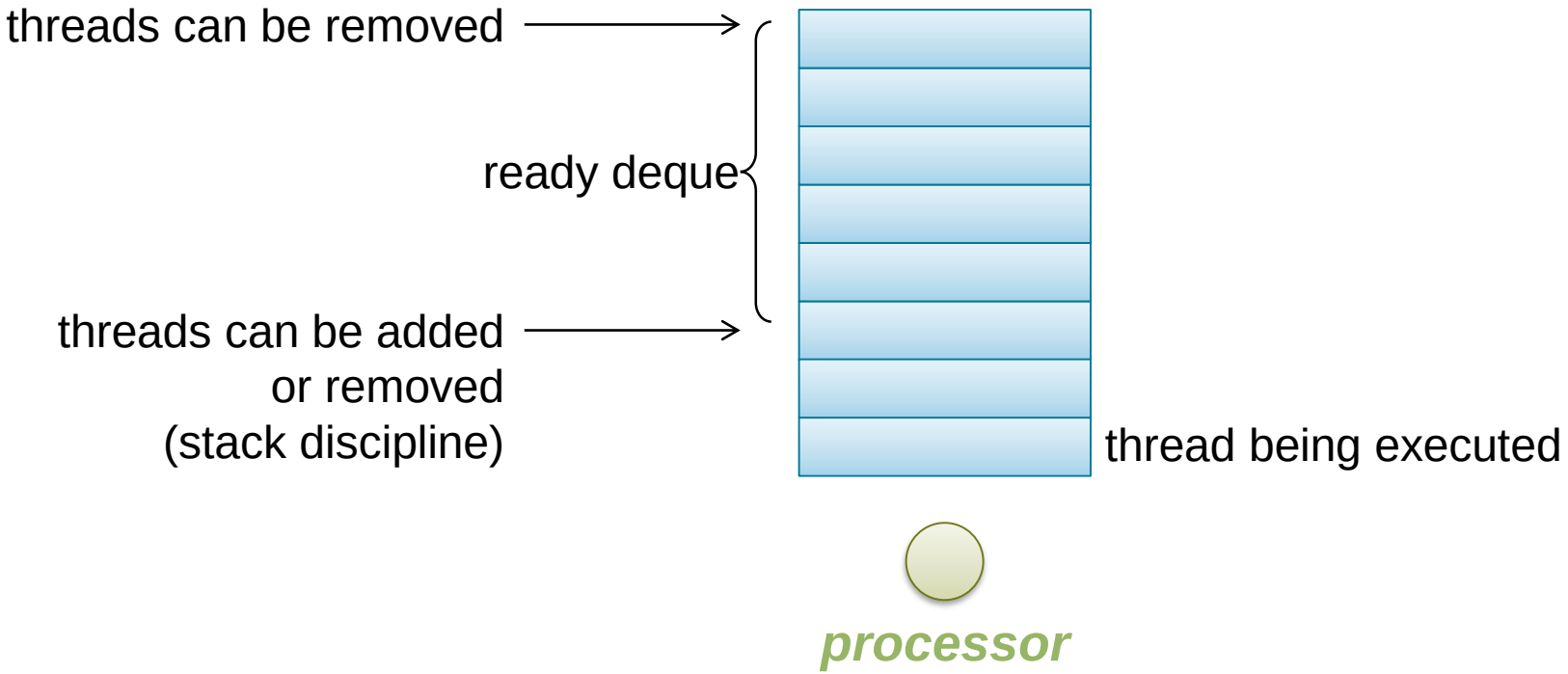
Greedy Scheduler

Maintain thread pool of live threads, each is ready or not

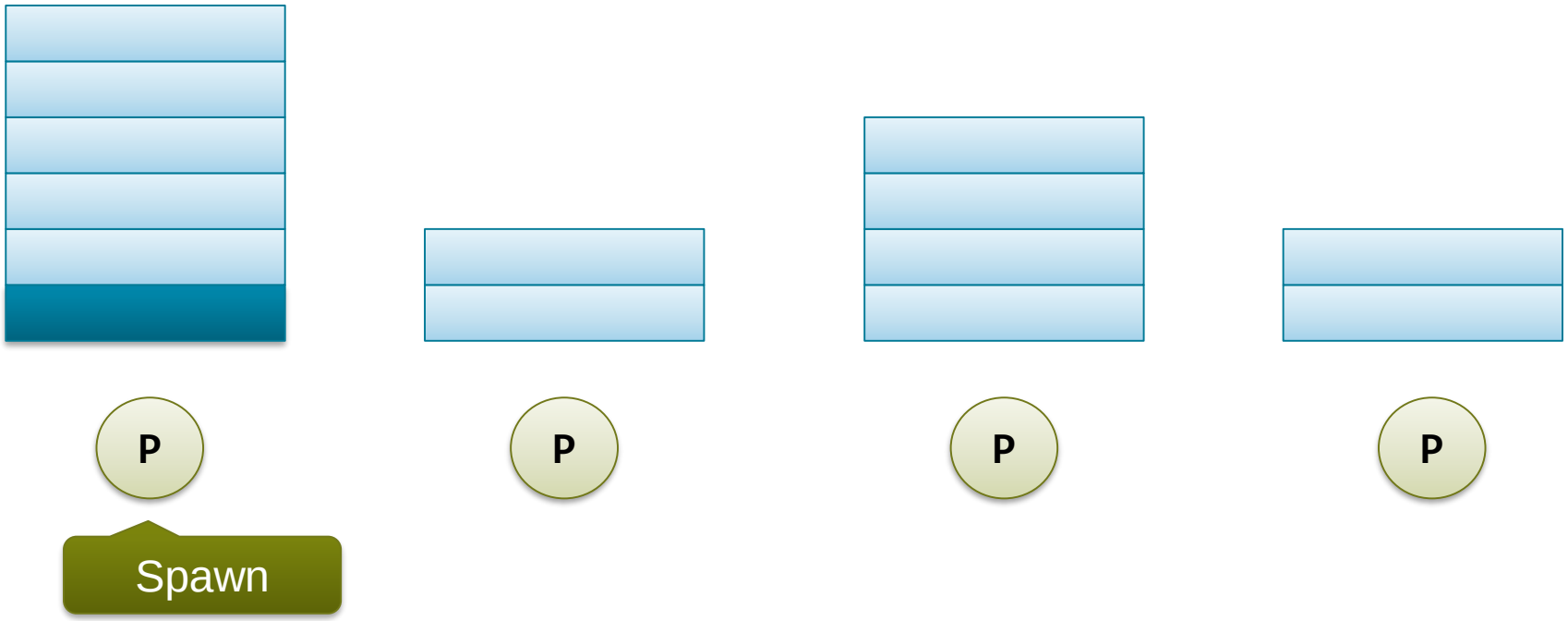
- **Initial: Root thread in thread pool, all processors idle**
- **At the beginning of each step each processor is idle or has a thread T to work on**
- **If idle**
  - *Get ready thread from pool*
- **If has thread T**
  - Case 0: T has another instruction to execute
    execute it
  - Case 1: thread T spawns thread S
    return T to pool, continue with S
  - Case 2: T stalls
    return T to pool, then idle
  - Case 3: T dies
    if parent of T has no living children, continue with the parent, otherwise idle

Work Stealing Scheduler

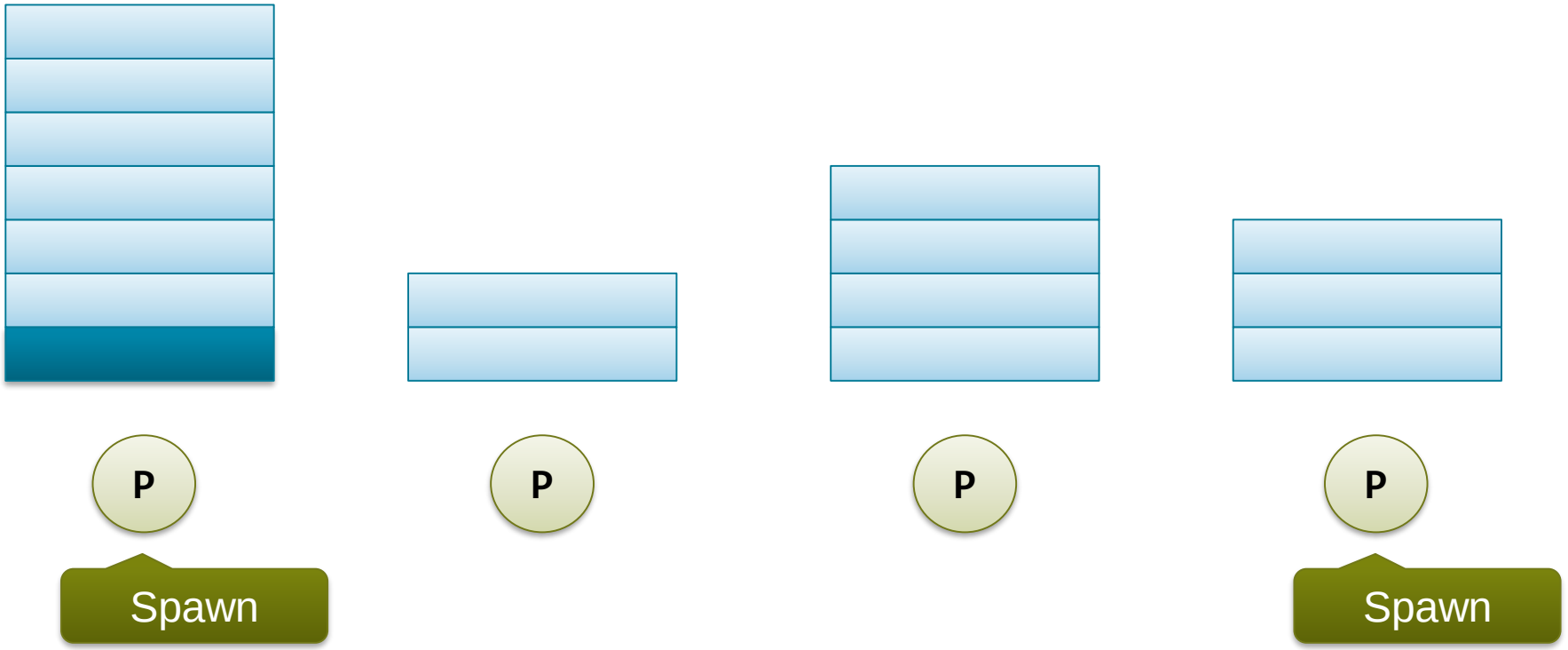
- **Each processor maintains a "ready deque:" deque of threads ready for execution; bottom is manipulated as a stack**



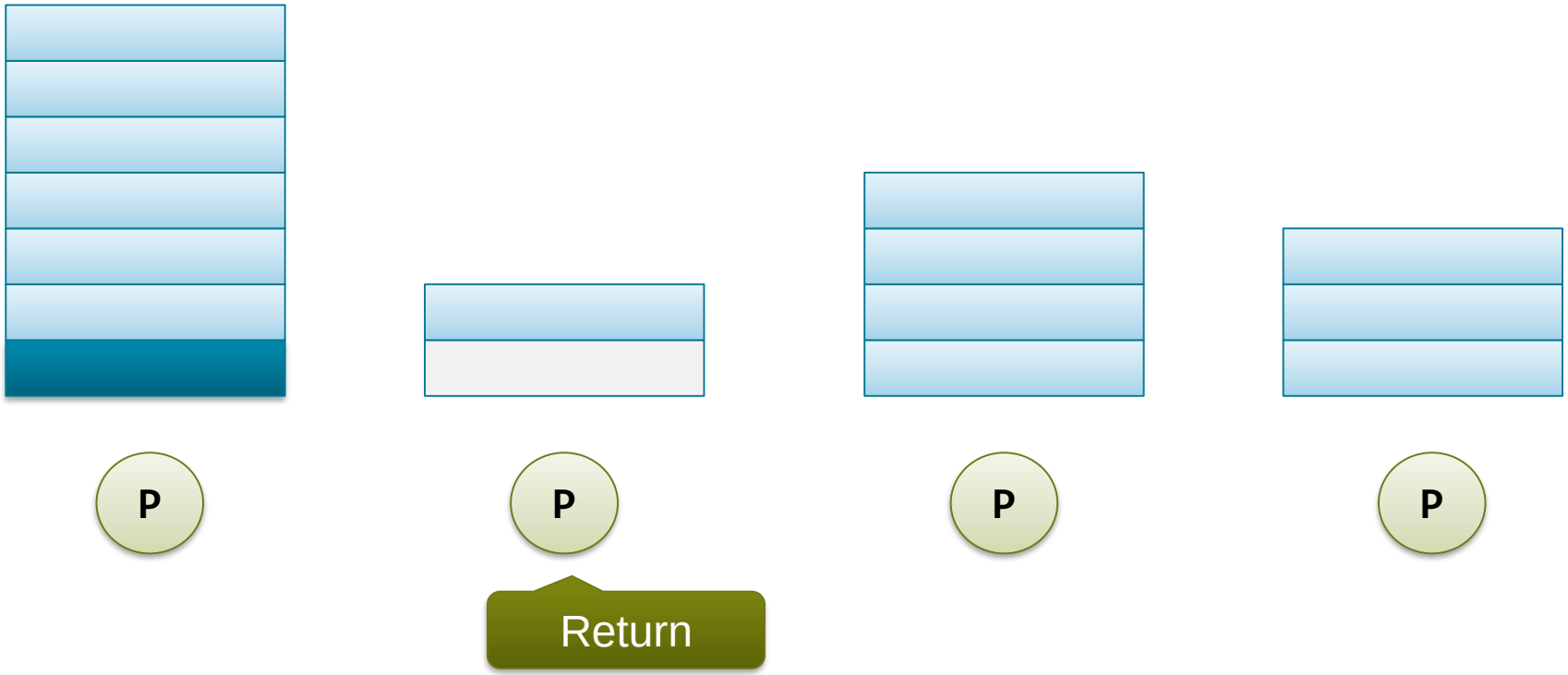
Work Stealing Scheduler



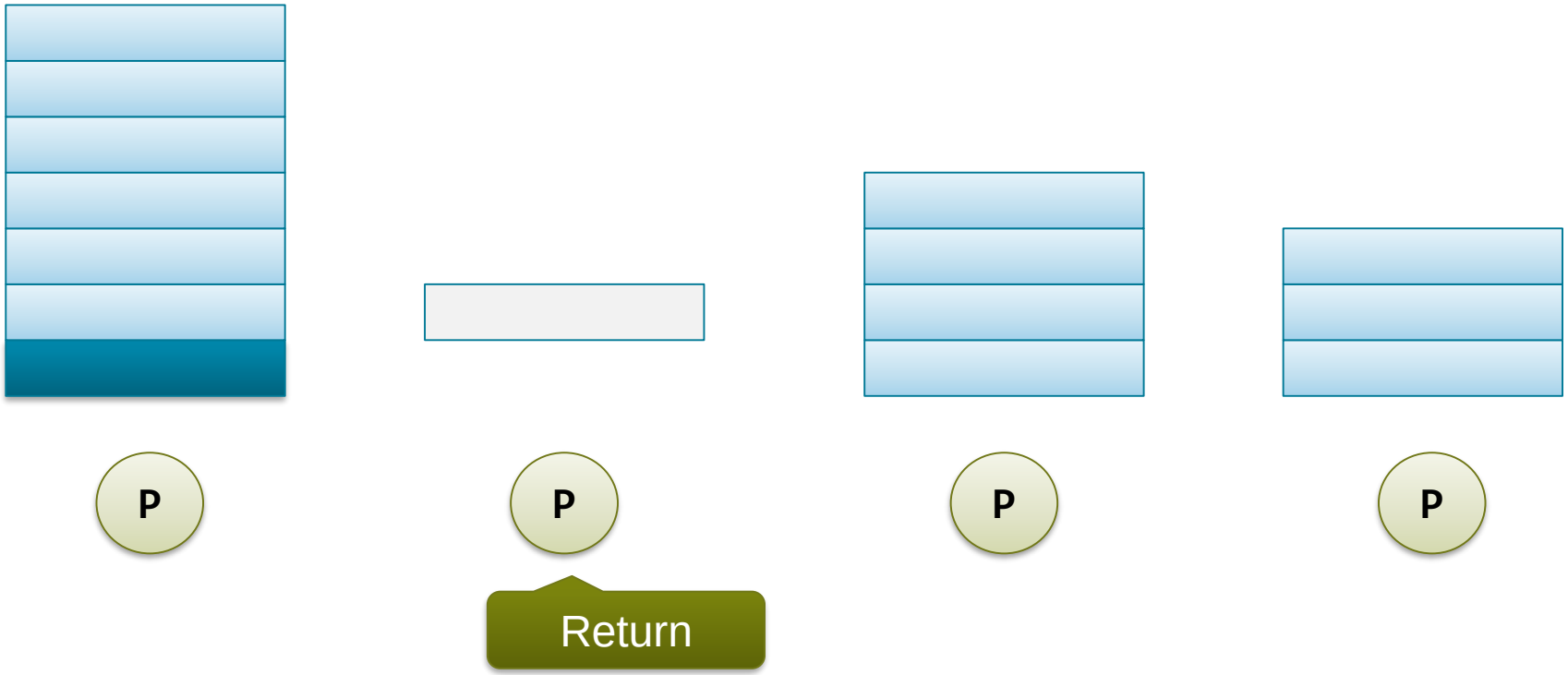
Work Stealing Scheduler



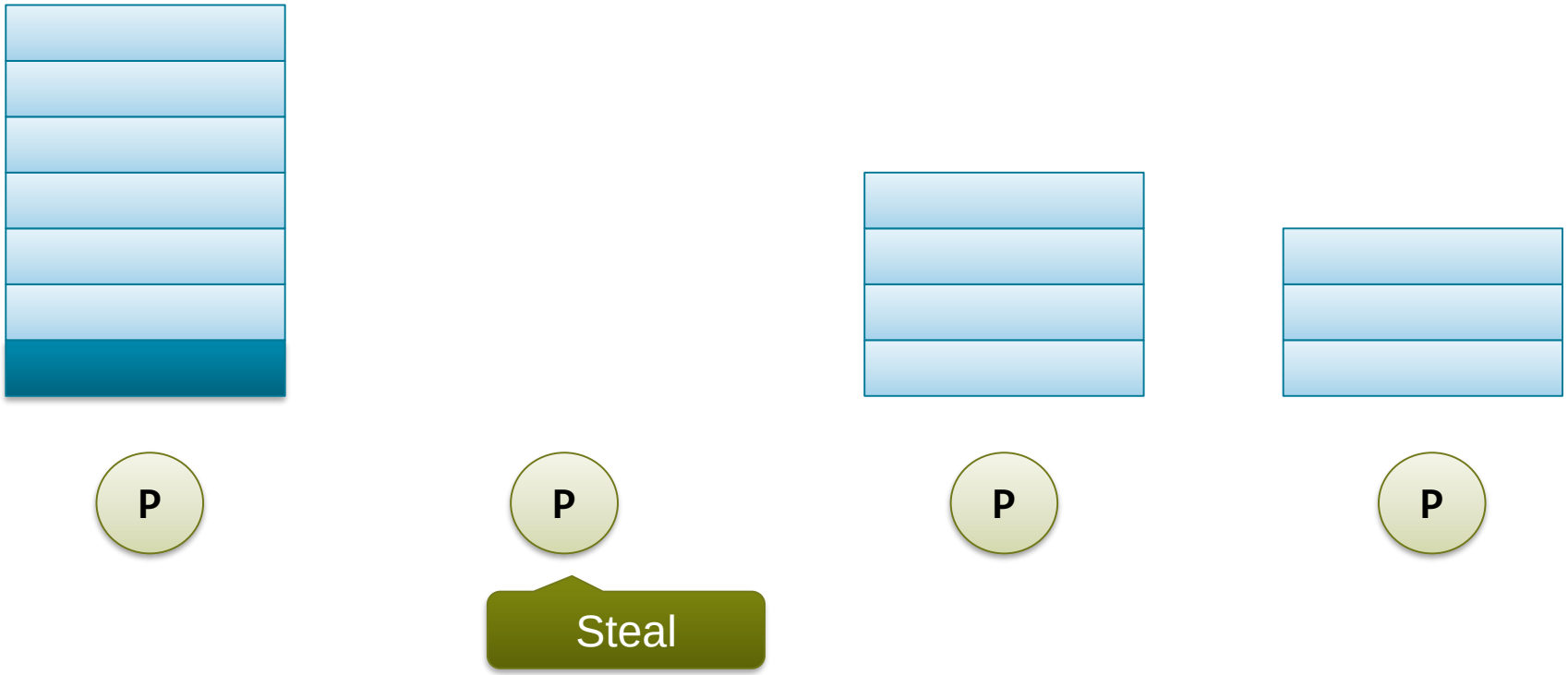
Work Stealing Scheduler



Work Stealing Scheduler

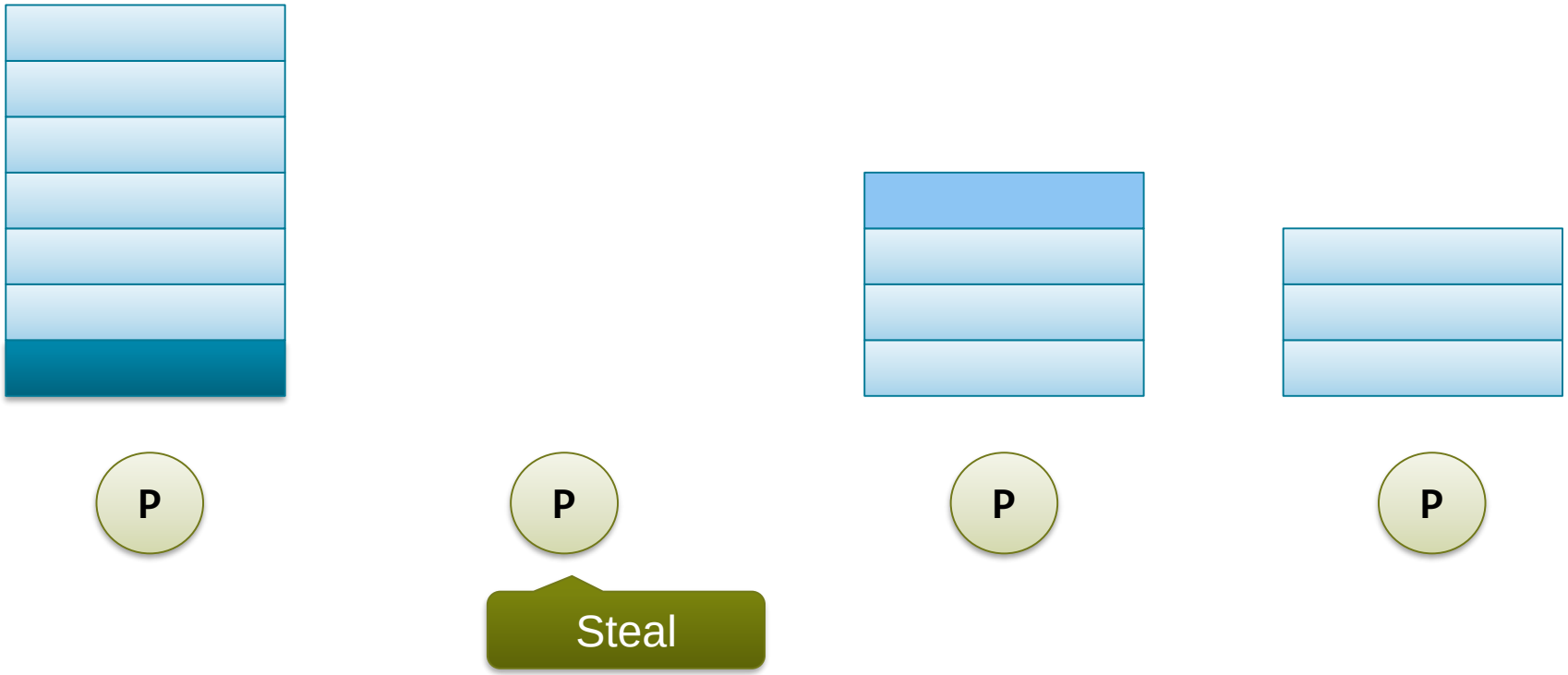


Work Stealing Scheduler

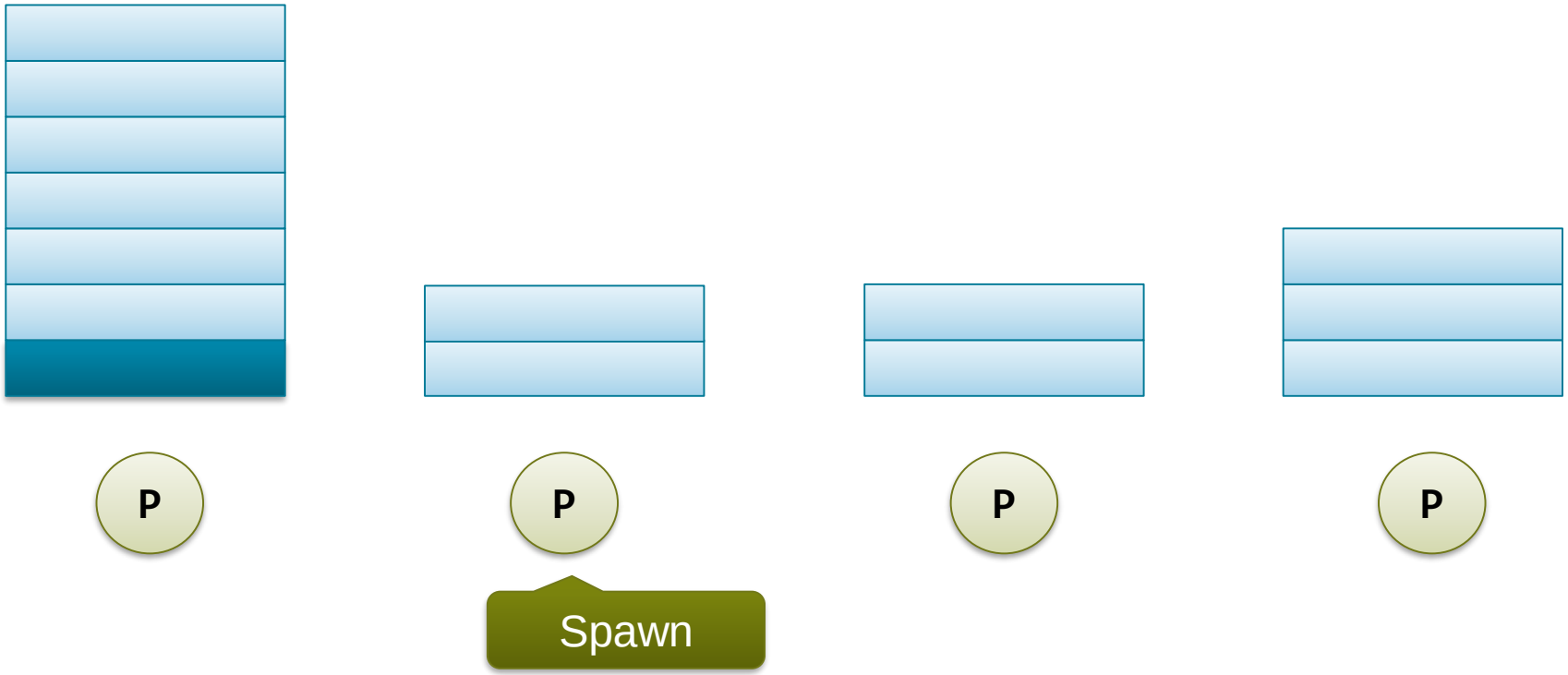


- When a processor runs out of work, it steals a task from the top of a random victim's deque.

Work Stealing Scheduler



Work Stealing Scheduler



Work Stealing Scheduler

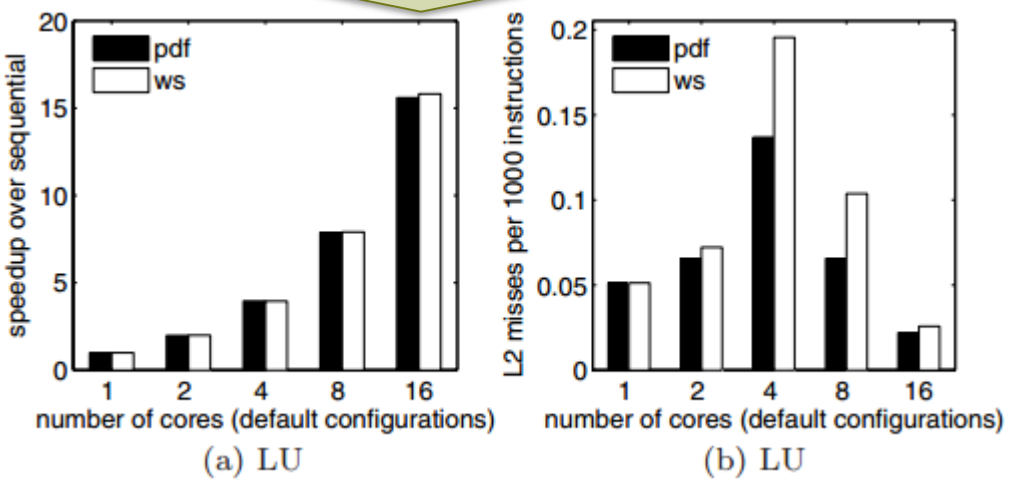
Each processor maintains a ready deque, bottom treated as stack

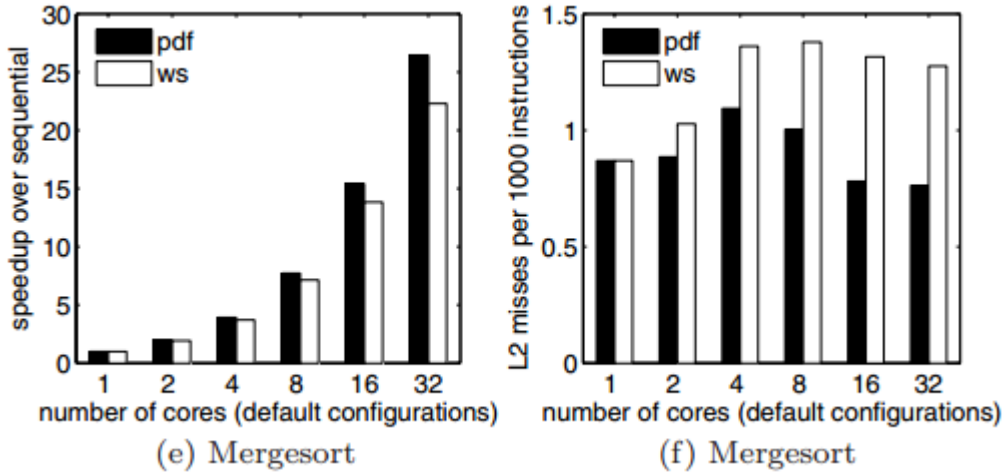
- **Initial: Root thread in deque of a random processor**
- **Deque not empty:**
  - Processor takes thread T from bottom and starts working
  - T spawns S: Put T on stack, continue with S
  - T stalls: Take next thread from stack
  - T dies: Take next thread from stack
  - If T enables a stalled thread S, S is put on the stack of T's processor
- **Deque empty:**
  - Steal thread from the top of a random (uniformly) processor's deque

Parallel Depth First Scheduler

- **Based on the following insight:**
  - *Important (sequential) programs have already been highly tuned to get a good cache performance on a single score*
  - *Small working set*
  - *Good spatial and temporal reuse*
- **Wh ed the ready-to-execute task that the sequential pro**

Why the speedup is not that different?
Low miss/instruction ratio => High Operational Intensity





Liaskovitis, Vasileios, et al. "Parallel depth first vs. work stealing schedulers on CMP architectures." Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2006.

Introducing Cilk

- **Cilk extends the C language with just a handful of keywords**
  - **cilk:** identifies a cilk procedure
  - **spawn:** spawns a new task
  - **sync:** synchronization point
- **It provides *performance guarantees* based on performance abstractions.**
- **Cilk is *processor-oblivious*.**

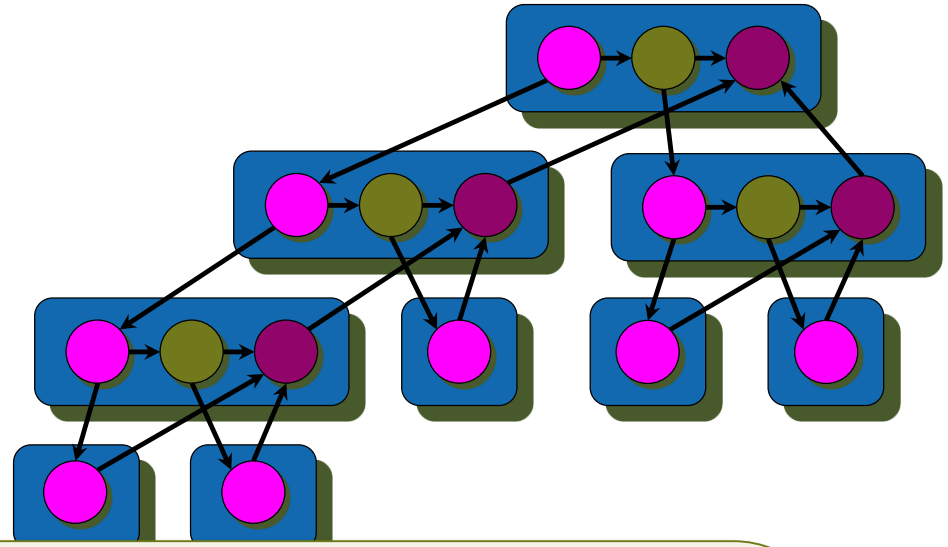
- **Cilk developed at MIT**
- **Cilk++ developed at Cilk Arts**
- **Cilk Plus based on Cilk and Cilk++**
  - Maintained by Intel

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

http://supertech.csail.mit.edu/cilk/lecture-1.ppt

Cilk Example: fib(4)

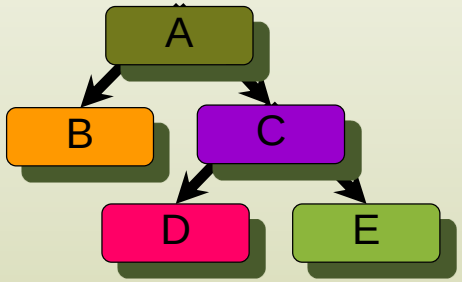
```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```

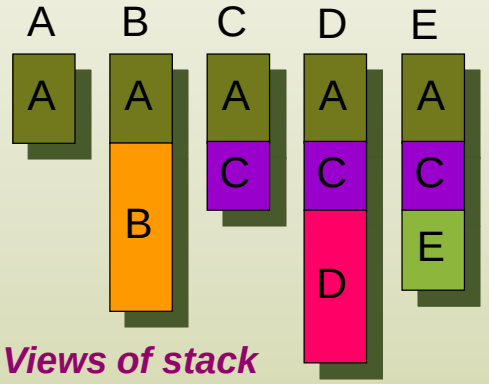


Assume for sim

Work: $T_1 = 17$

What about pointers? *A pointer to stack space can be passed from parent to child, but not from child to parent.*





Views of stack

Cilk Example: Vector Addition

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

How to parallelize?

```
void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    vadd (A, B, n/2);
    vadd (A+n/2, B+n/2, n-n/2);
  }
}
```

http://supertech.csail.mit.edu/cilk/lecture-1.ppt

Cilk Example: Vector Addition

```
void vadd (real *A, real *B, int n){
  int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

How to parallelize?

```
cilk void vadd (real *A, real *B, int n){
  if (n<=BASE) {
    int i; for (i=0; i<n; i++) A[i]+=B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
  } sync;
}
```

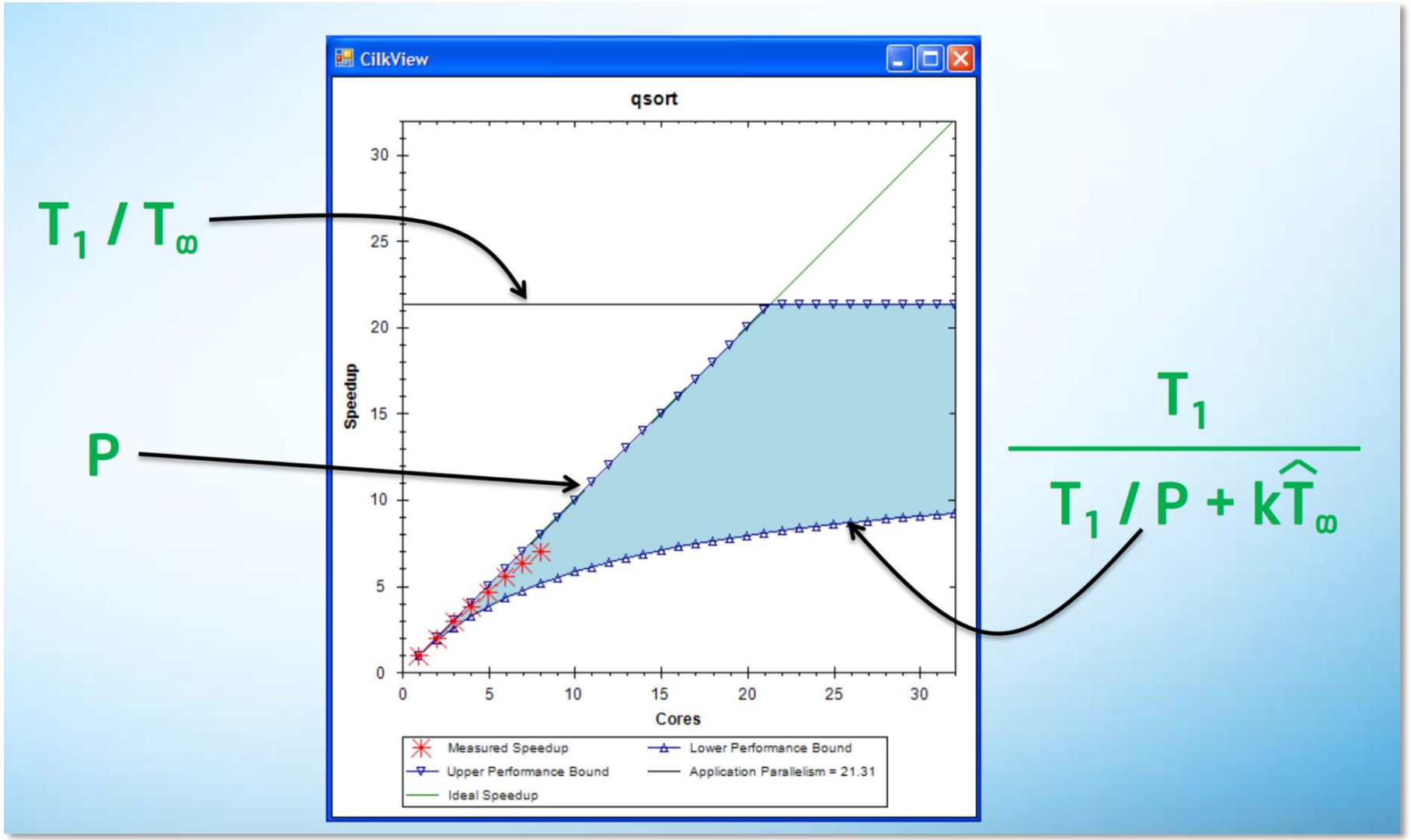
http://supertech.csail.mit.edu/cilk/lecture-1.ppt

Cilk Plus: Scalability Estimation

- Cilkview reads from metadata embedded by the Cilk Plus compiler to perform its calculations.
- Cilkview generates rough (but repeatable) performance measures by counting instructions rather than reading from a clock.
- Despite the coarseness of measurements, Cilkview accurately estimates scalability.

http://www.ckluk.org/ck/talks/cilkplus-tutorial-ppopp11.pdf

Cilk Plus: Scalability Estimation



http://www.ckluk.org/ck/talks/cilkplus-tutorial-ppopp11.pdf

Cilk Plus: Race Detection

salvodg@einstein:~/cilktools-linux-004421/bin

```
 * otherwise, except as expressly provided in the license
 * provided with the Materials.
 *
 * This file implements a simple Intel Cilk Plus program with a known race
 * condition to demonstrate the Cilkscreen race detector.
 */

#include <stdlib.h>
#include <stdio.h>
#include <cilk/cilk.h>


int x;
void race(void)
{
     x = 0;
}

void race1(void)
{
     x = 1;
}

void test(void)
{
     cilk_spawn race();
     cilk_spawn race1();
}
```

salvodg@einstein:~/cilktools-linux-004421/bin

```
[salvodg@einstein bin]$
[salvodg@einstein bin]$
[salvodg@einstein bin]$ ./cilkscreen ../examples/simple-race/simple-race
Cilkscreen Race Detector V2.0.0, Build 4421

Race condition on location 0x603a90
  write access at 0x400e97: (/home/salvodg/cilktools-linux-004421/examples/simple-race/simple-race.cpp:34, test()::__cilk_spawn_1::operator()+0x67)
  write access at 0x400f27: (/home/salvodg/cilktools-linux-004421/examples/simple-race/simple-race.cpp:39, test()::__cilk_spawn_2::operator()+0x67)
    called by 0x400cdb: (/home/salvodg/cilktools-linux-004421/examples/simple-race/simple-race.cpp:45, test+0xab)
    called by 0x400c0a: (/home/salvodg/cilktools-linux-004421/examples/simple-race/simple-race.cpp:50, main+0x2a)
done: x = 1
1 error found by Cilkscreen
[salvodg@einstein bin]$
```