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

DPHPC: Work-Depth

Recitation session

Reference:
Guy E. Blelloch and Bruce M. Maggs. 2010. Parallel algorithms. In *Algorithms and theory of computation handbook* (2 ed.), Mikhail J. Atallah and Marina Blanton (Eds.). Chapman & Hall/CRC 25-25.

RAM model

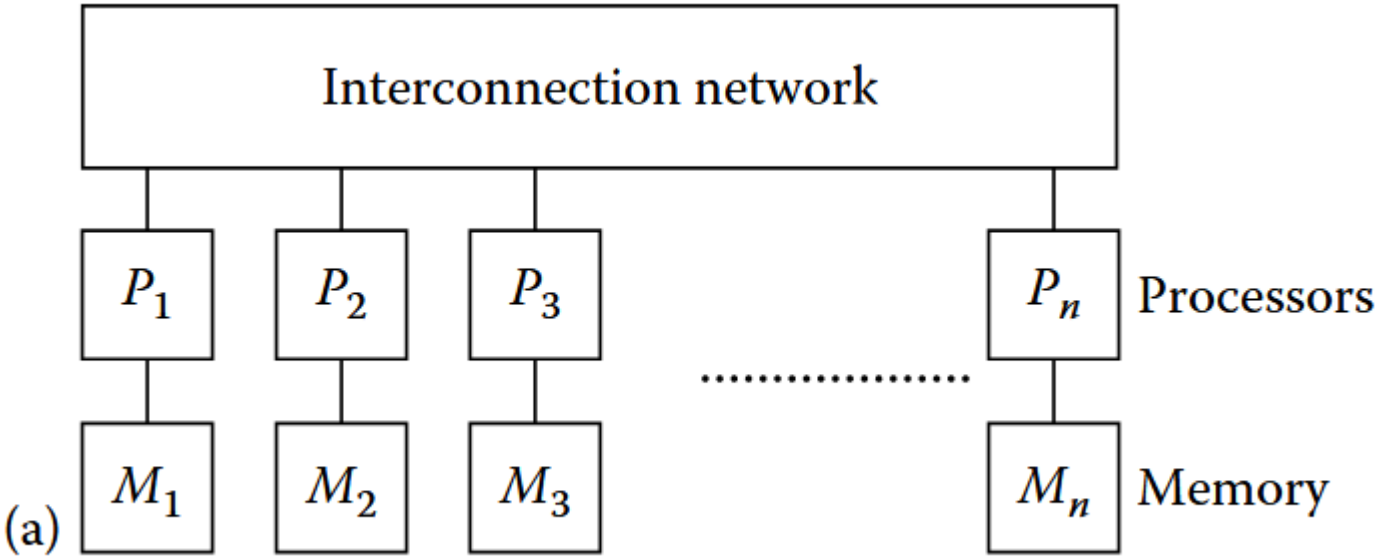
- Memory consists of infinite number of cells
- Instructions exectued sequentially, one at time
- All instructions take unit time

How to model parallel applications?

PRAM is a generalization of RAM

However, parallel computers tend to vary more in organization than do sequential computers

Local-Memory Machine



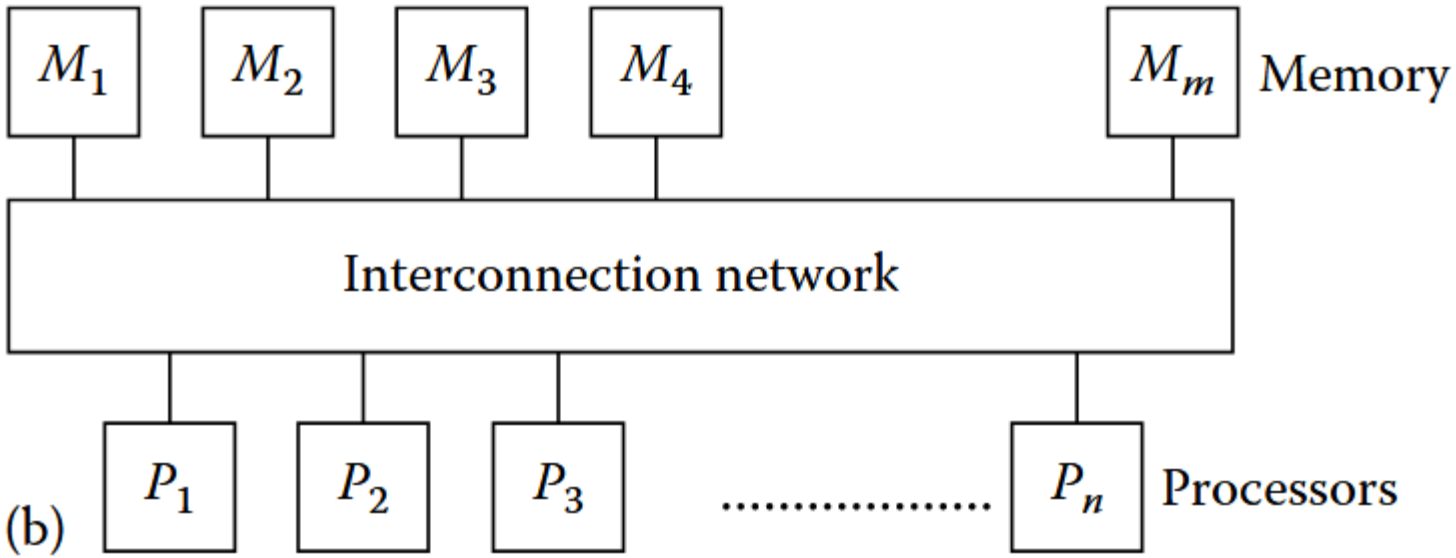
How do processors access memory?

Local memory is accessed directly, remote memory is accessed by sending messages through the interconnection network.

What is the cost of the memory accesses?

Local accesses are made in unit time. Remote accesses cost depends on (a) the interconnection network and (b) on the access pattern of the other processors (may congest the network).

Modular Memory Machine



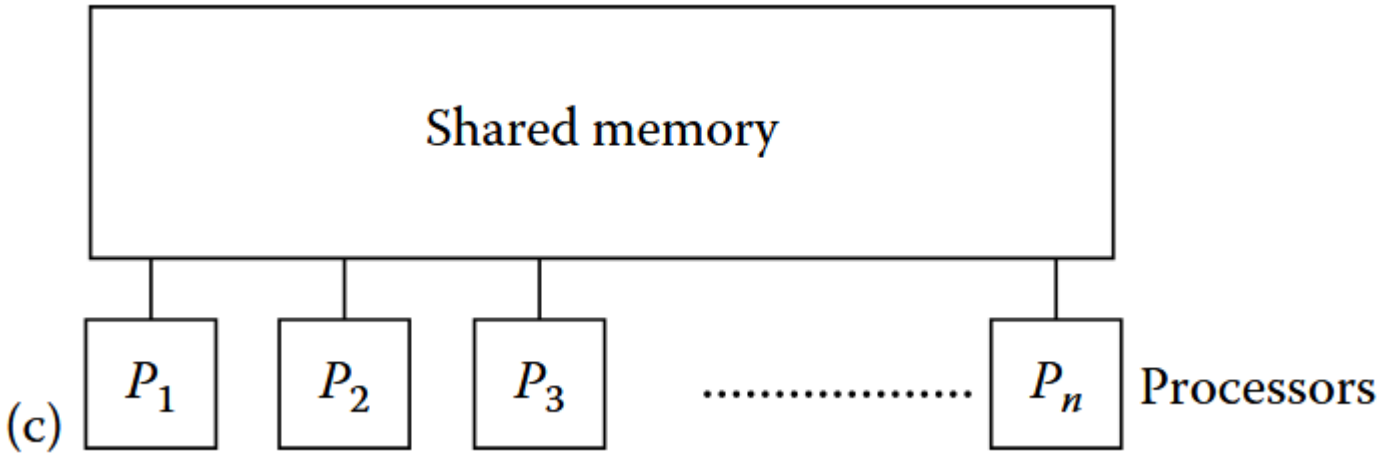
How do processors access memory?

Memory access requests are sent through the interconnection network.

What is the cost of the memory accesses?

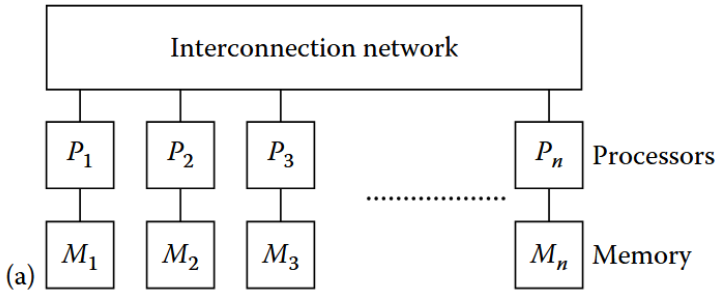
Typically, memory and processors are arranged such that the memory access cost is roughly uniform. Exact time depends on the interconnection network and on the access pattern of other processors.

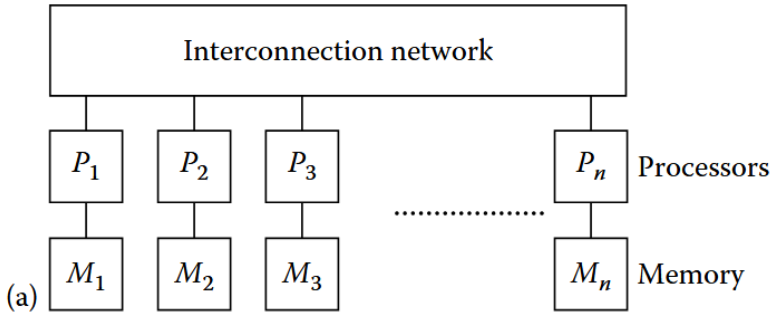
Shared-Memory Machine (PRAM model)



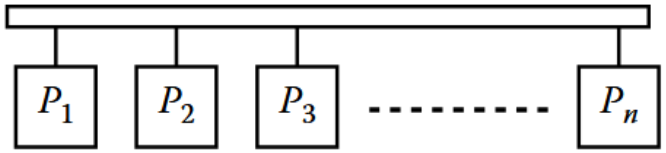
How do processors access memory?
Processors are directly attached to the memory.

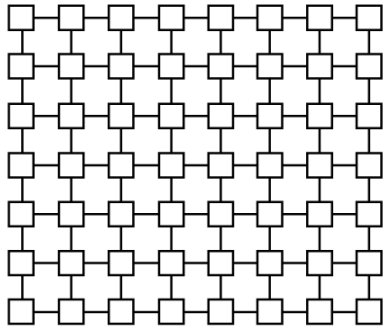
What is the cost of the memory accesses?
Unit cost, parallel accesses are possible (write-conflicts have to be managed according with the model).

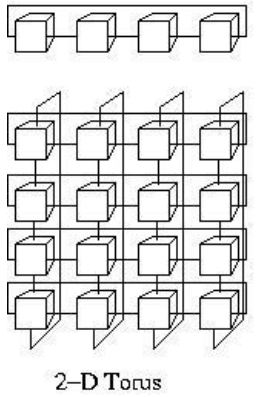


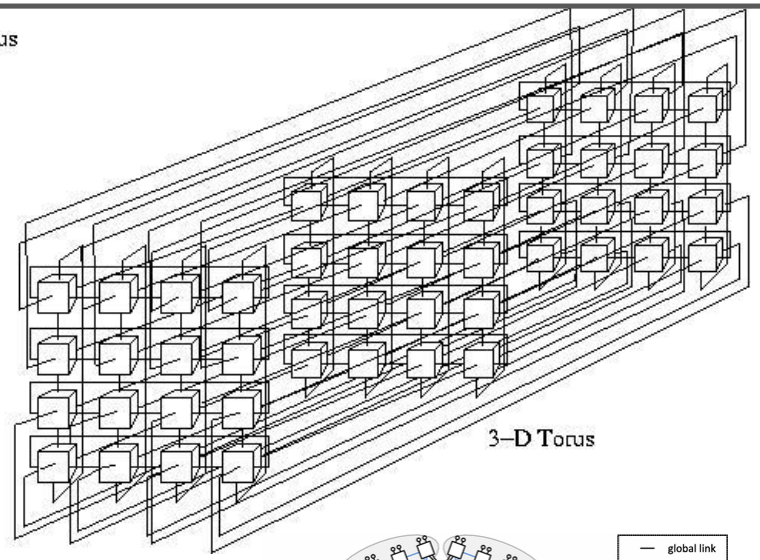


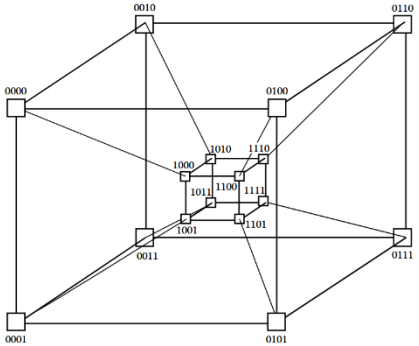
Interconnection Networks

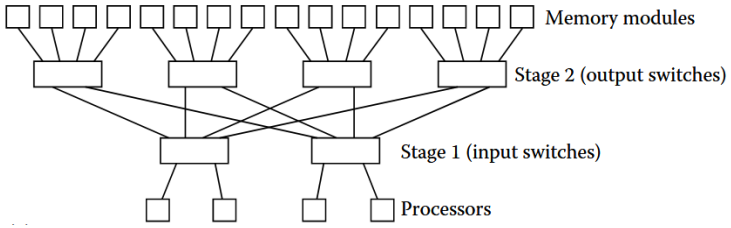


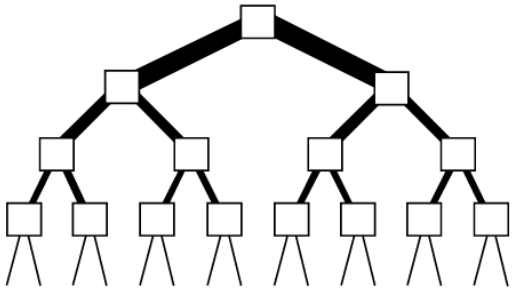


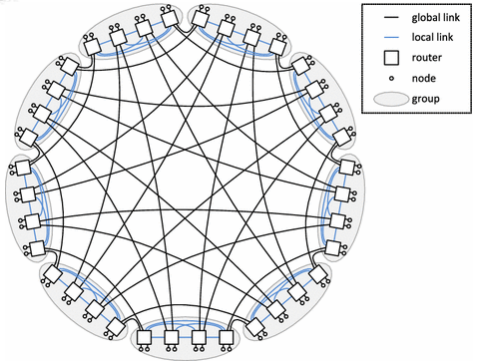


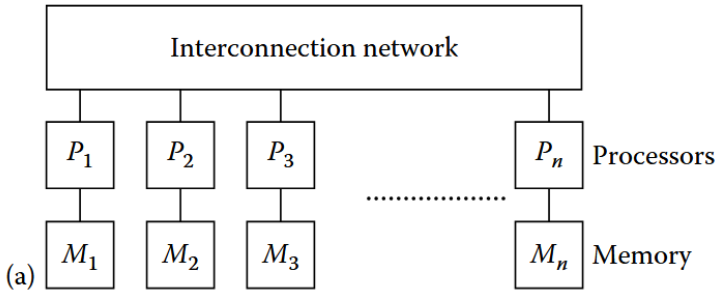


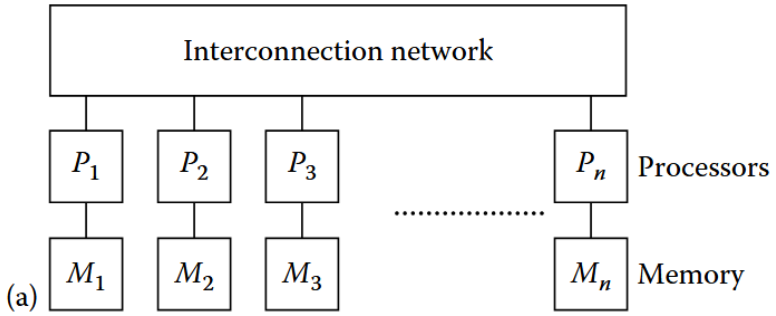




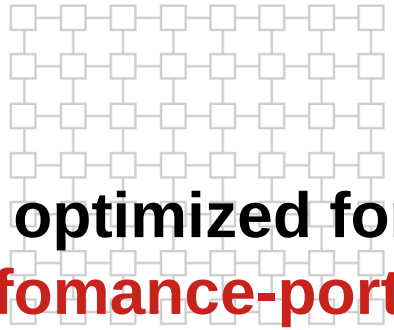


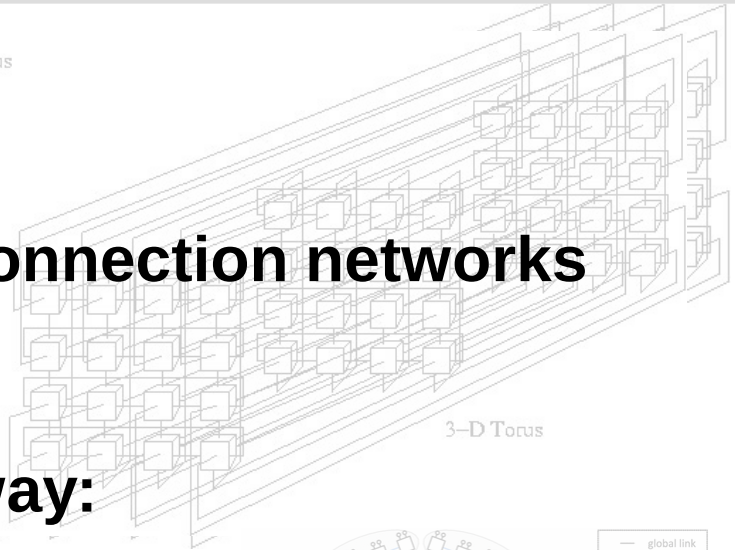




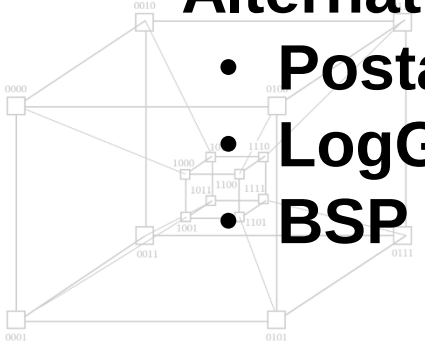


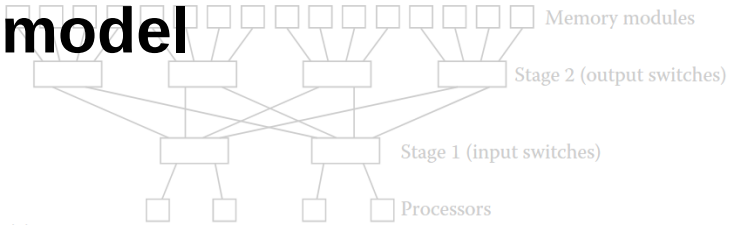
Interconnection Networks





- **Algorithms can be optimized for specific interconnection networks**
  - **Difficult for perfomance-portability!**
- **Alternative is to abstract the network details away:**
  - **Postal model**
  - **LogGP**
  - **BSP**





Work-Depth Model

- **Idea:** Instead of focusing on the architecture (too many different ways to organize a parallel computer), let's focus on the algorithm.

Circuit model:
Nodes and directed arcs.

Work: total number of nodes.

Depth: longest directect path from an input arc to an output arc

Vector model:
Sequence of steps. A step is a vector operation.

Work: work of a step is the length of it's input vector. Work of the algorithm is sum of works of its steps.

Depth: number of steps

Language model:
Programming language constructs.

Work: Number of constructs.

Depth: maximum depth of call sequences.

Algorithm Cost

Work and depth can be viewed as the running time of an algorithm at two limits: one processor (work) and an unlimited number of processors (depth).

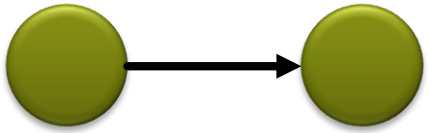
Brent's theorem provides bounds to the running time:

$$\frac{W}{P} \leq T \leq \frac{W}{P} + D$$

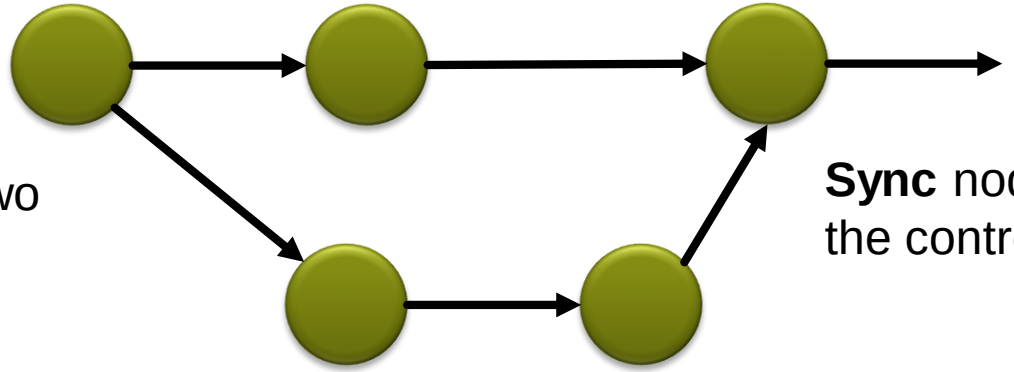
Richard P. Brent. The parallel evaluation of general arithmetic expressions. Journal of the Association for Computing Machinery, 21(2):201-206, 1974.

Defining a DAG

Strand: chain of serially executed instructions.



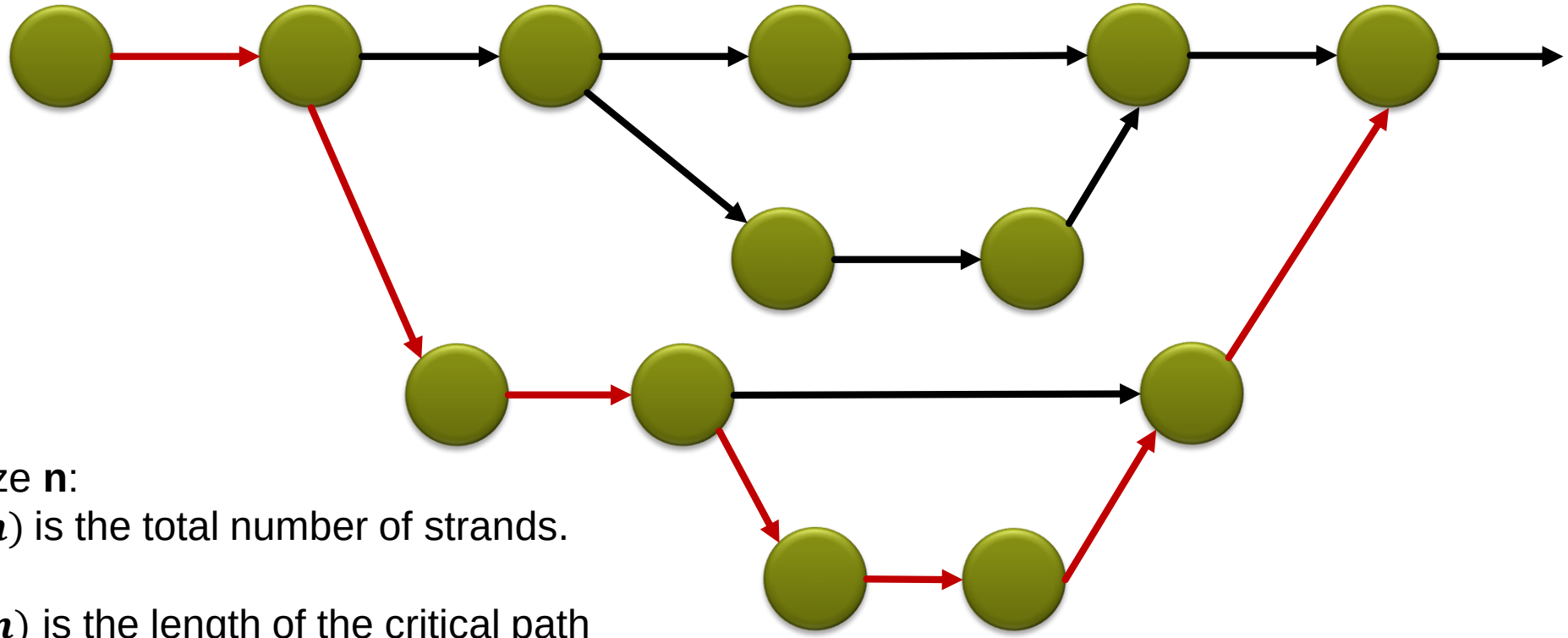
Strands are partially ordered with **dependencies**



Spawn nodes have two successors

Sync nodes are where the control flow merges

Defining a DAG



Given an input size **n**:

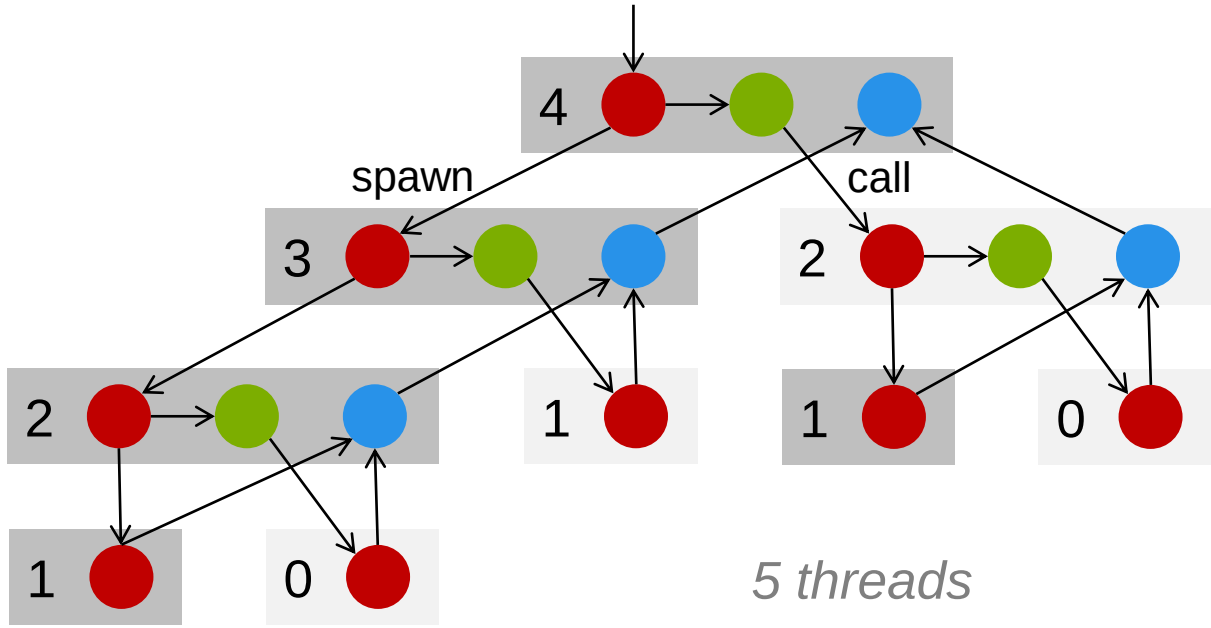
- The **work** $W(n)$ is the total number of strands.
  - W(n)=13
- The **depth** $D(n)$ is the length of the critical path (measured in number of strands).
  - Defines the minimum execution time of the computation
  - D(n)=8

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

Scheduling a DAG

```
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:

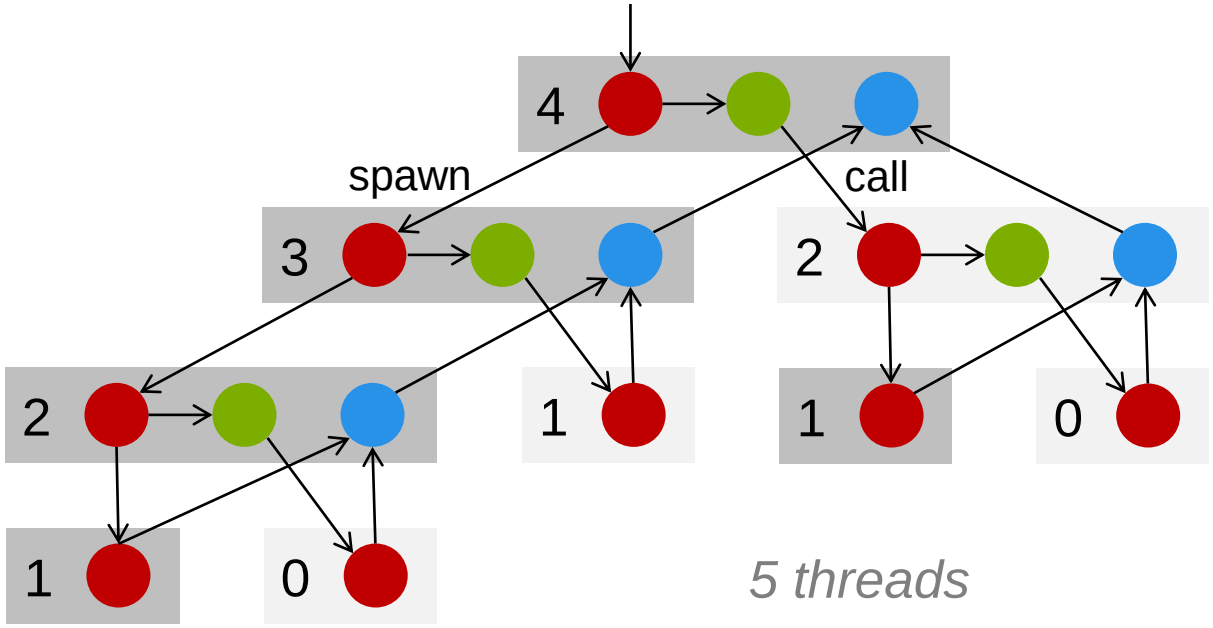


5 threads

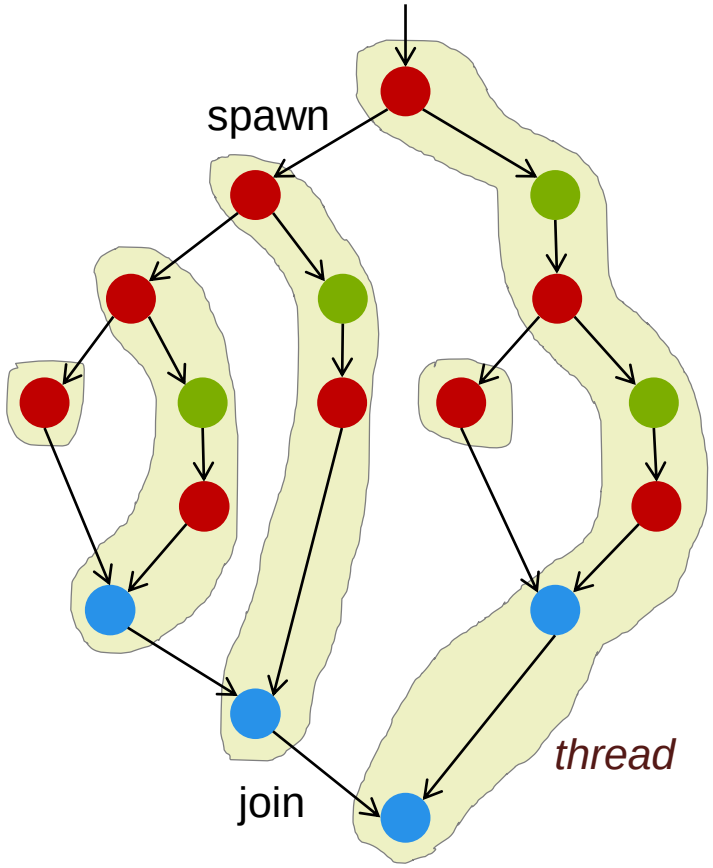
Node: Sequence of instructions without call, spawn, sync, return
Edge: Dependency

Scheduling a DAG

The DAG unfolds dynamically:



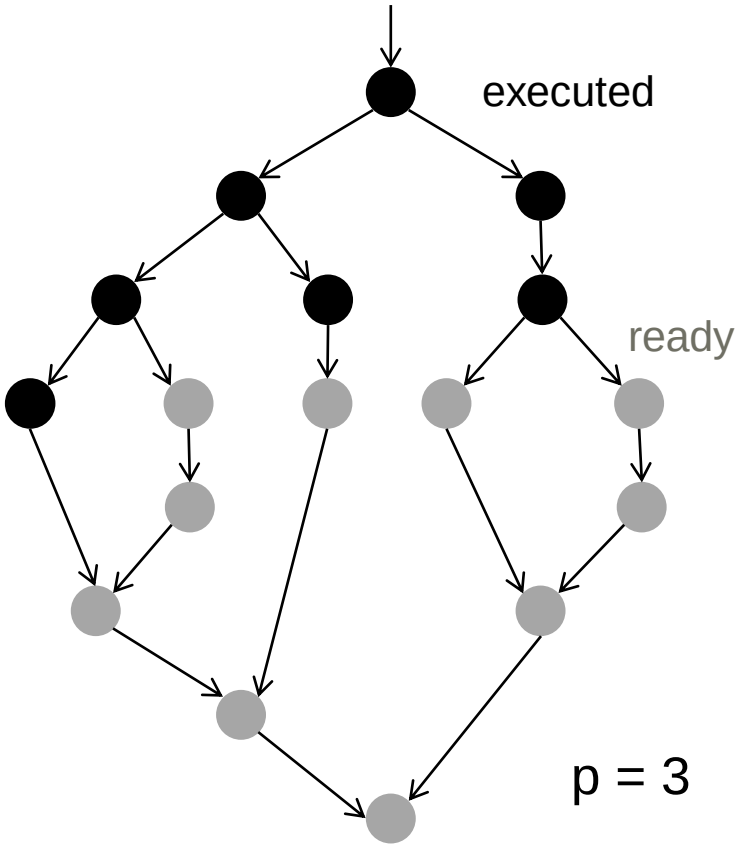
5 threads



Remember oblivious algorithms?

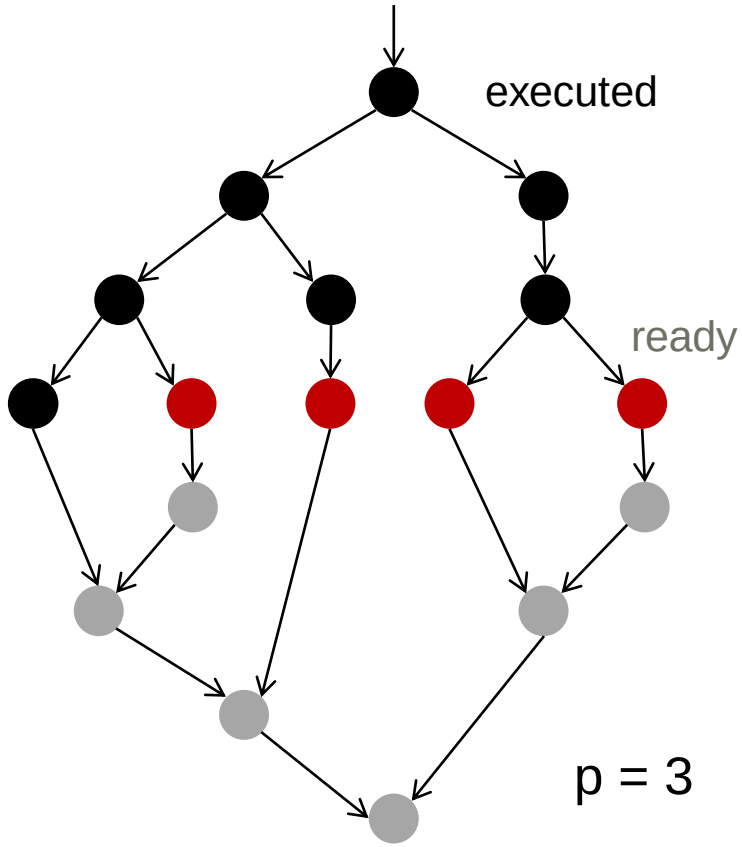
Greedy Scheduler

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



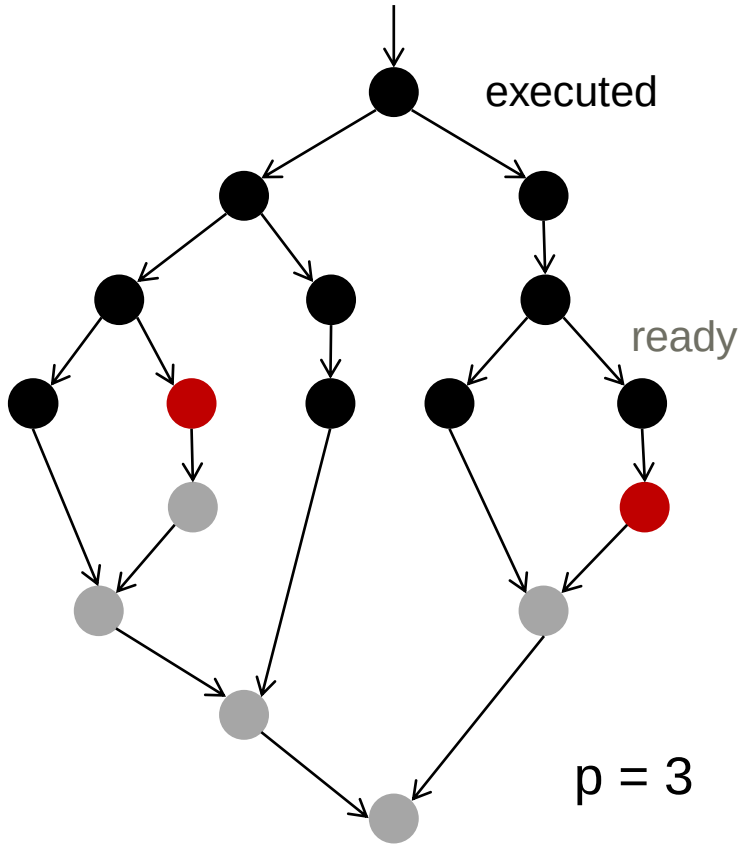
Greedy Scheduler

- *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



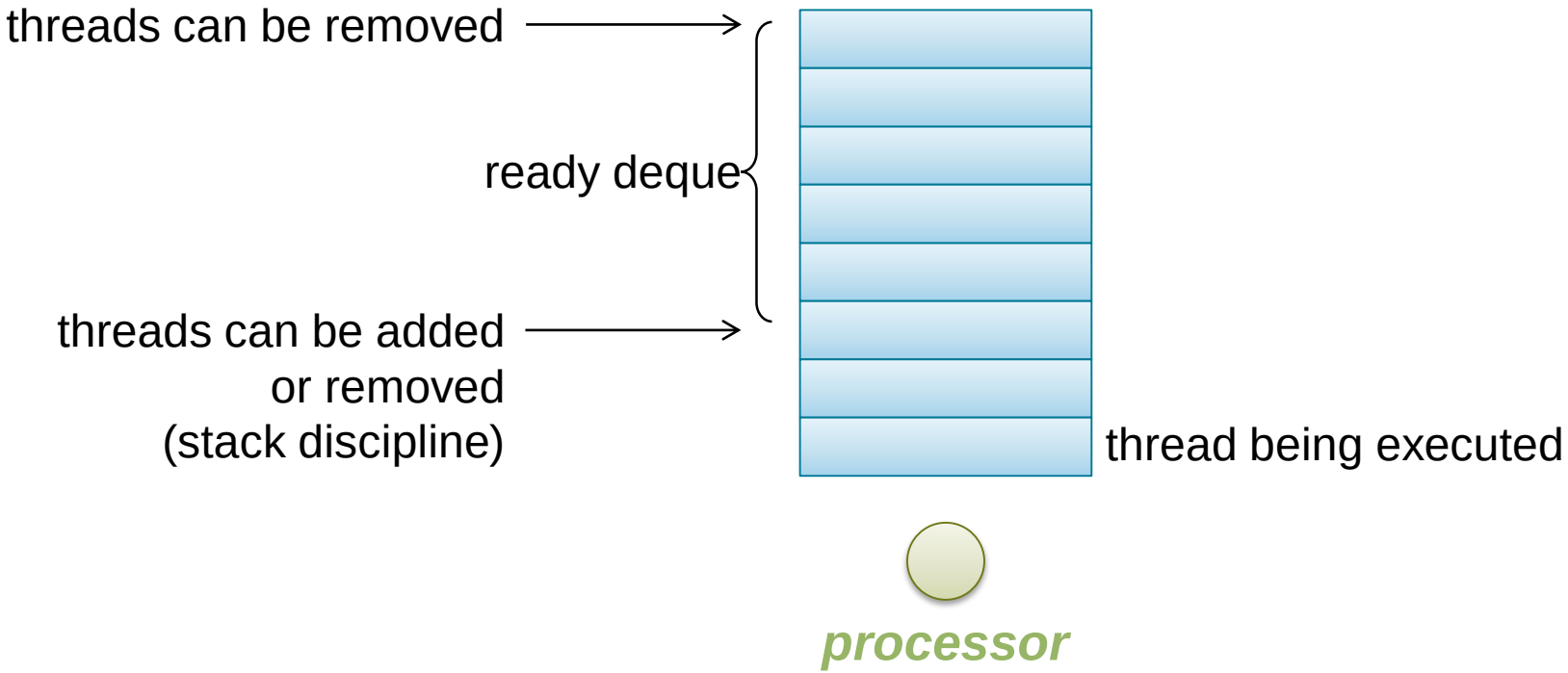
Greedy Scheduler

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

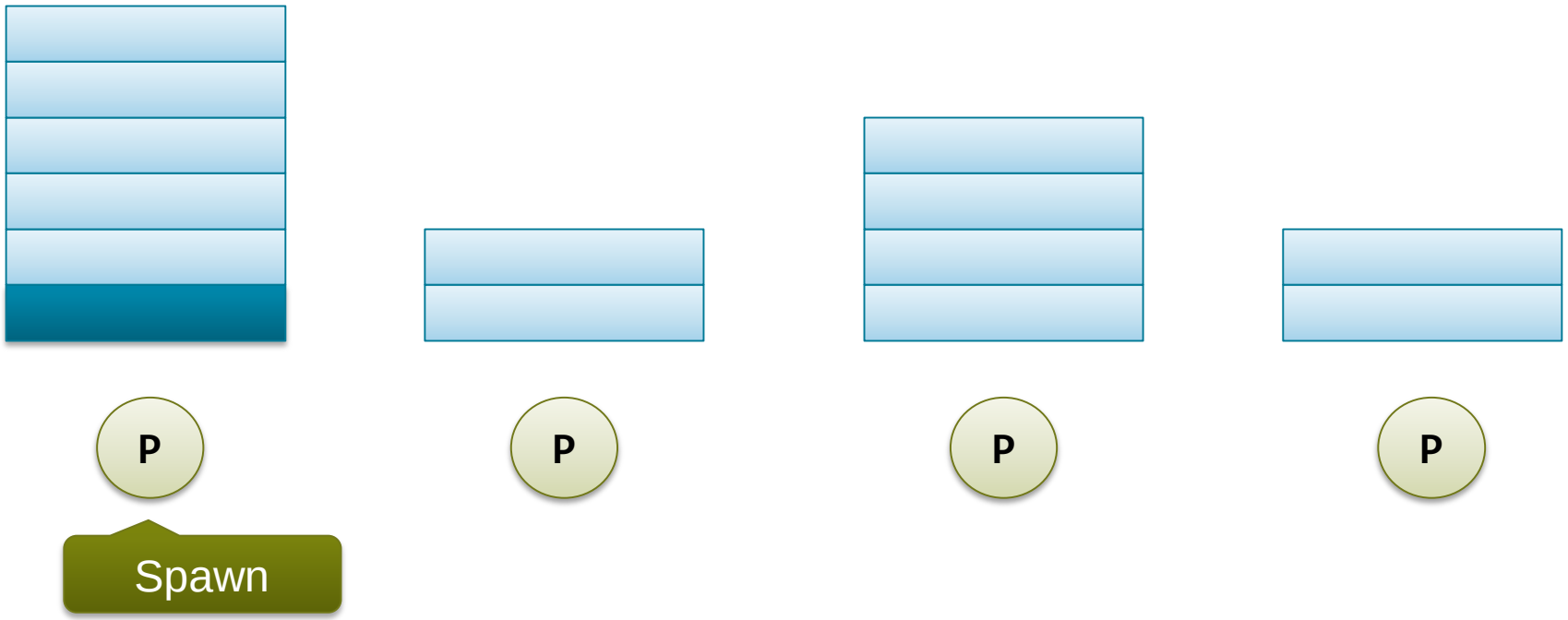
- **Initial: Root thread in thread pool, all processors idle**
- **At the beginning of each step each processor is idle or has a thread T to work on**
- **If idle**
  - *Get ready thread from pool*
- **If has thread T**
  - Case 0: T has another instruction to execute
    execute it
  - Case 1: thread 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

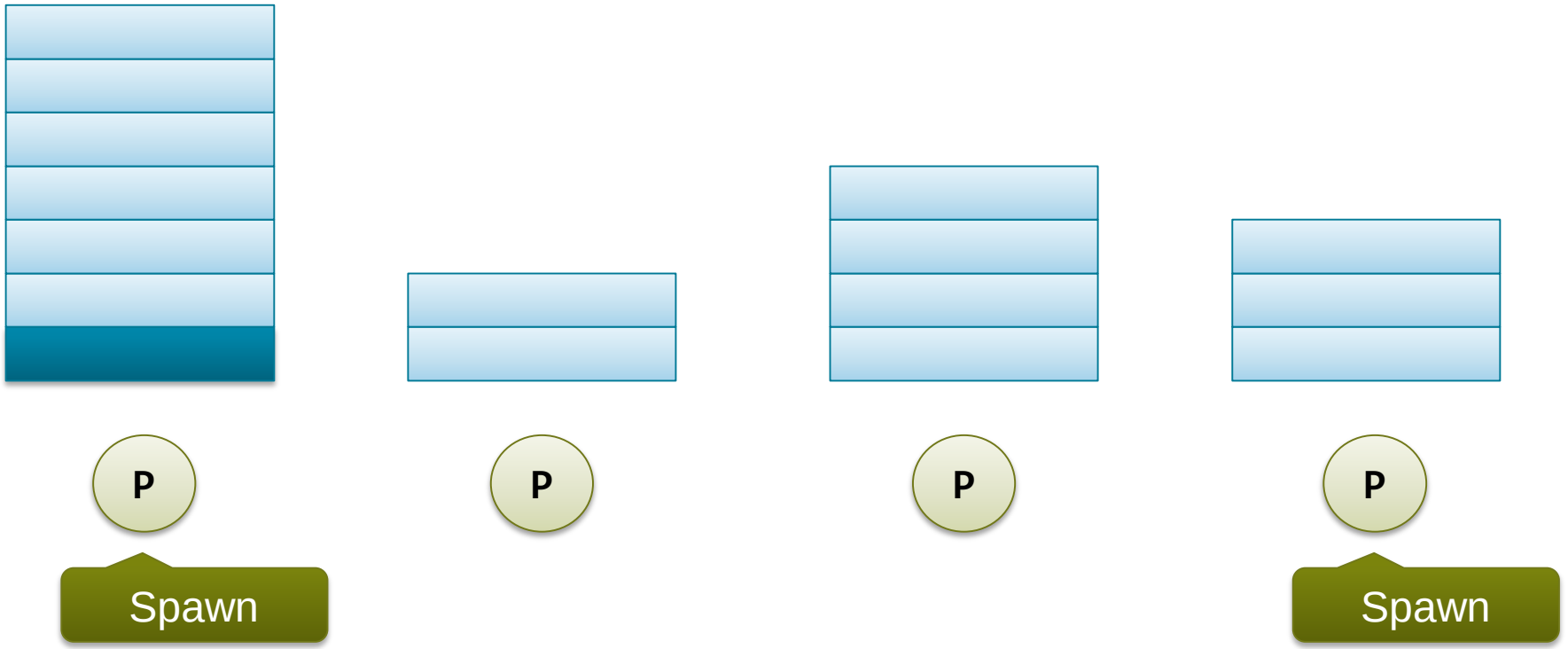
- **Each processor maintains a "ready deque:" deque of threads ready for execution; bottom is manipulated as a stack**



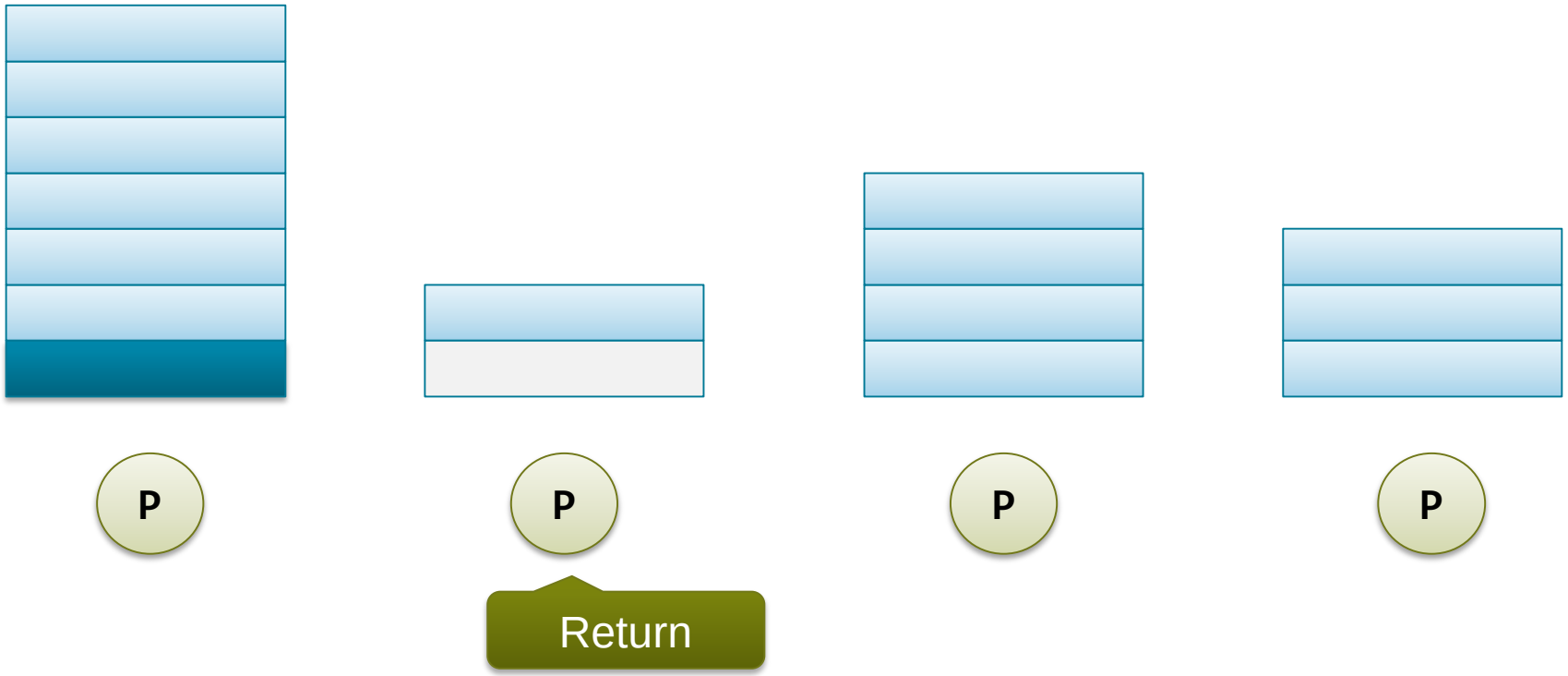
Work Stealing Scheduler



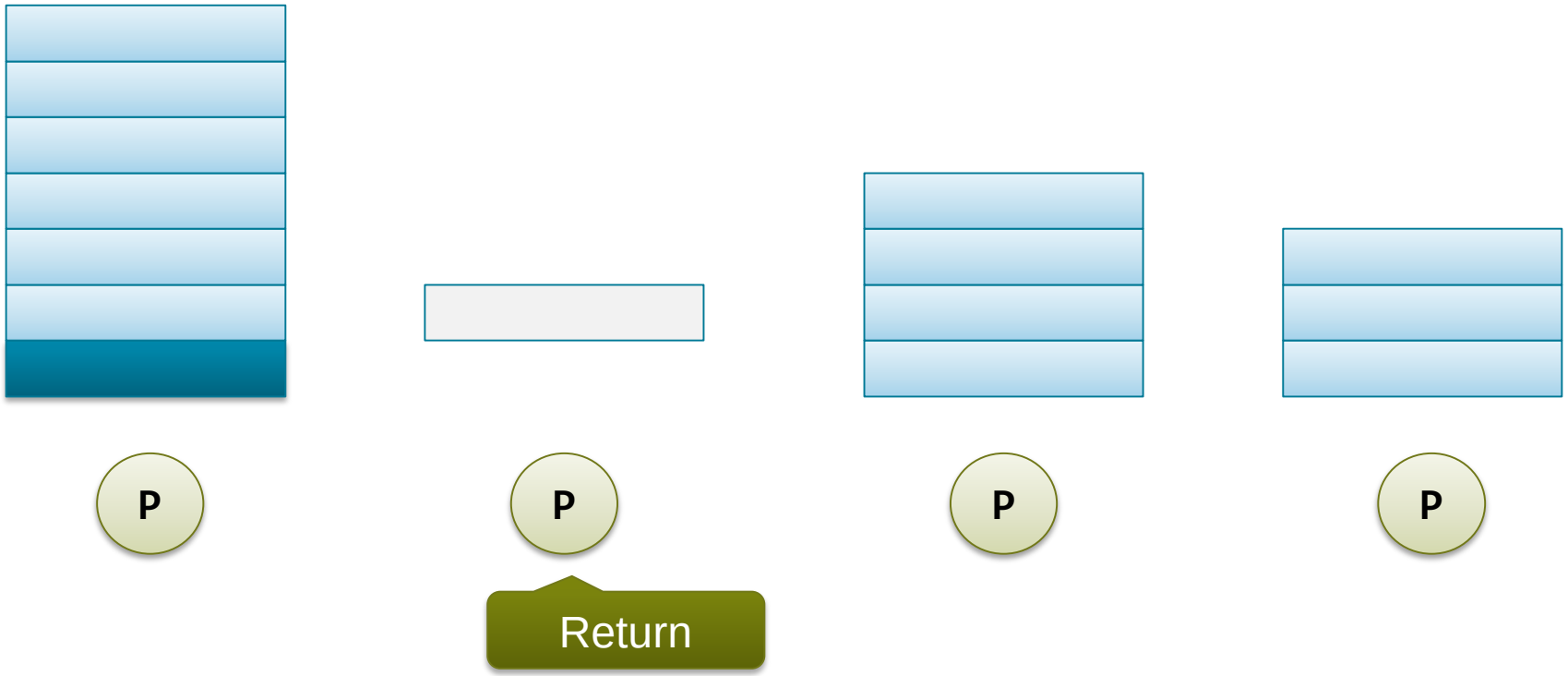
Work Stealing Scheduler



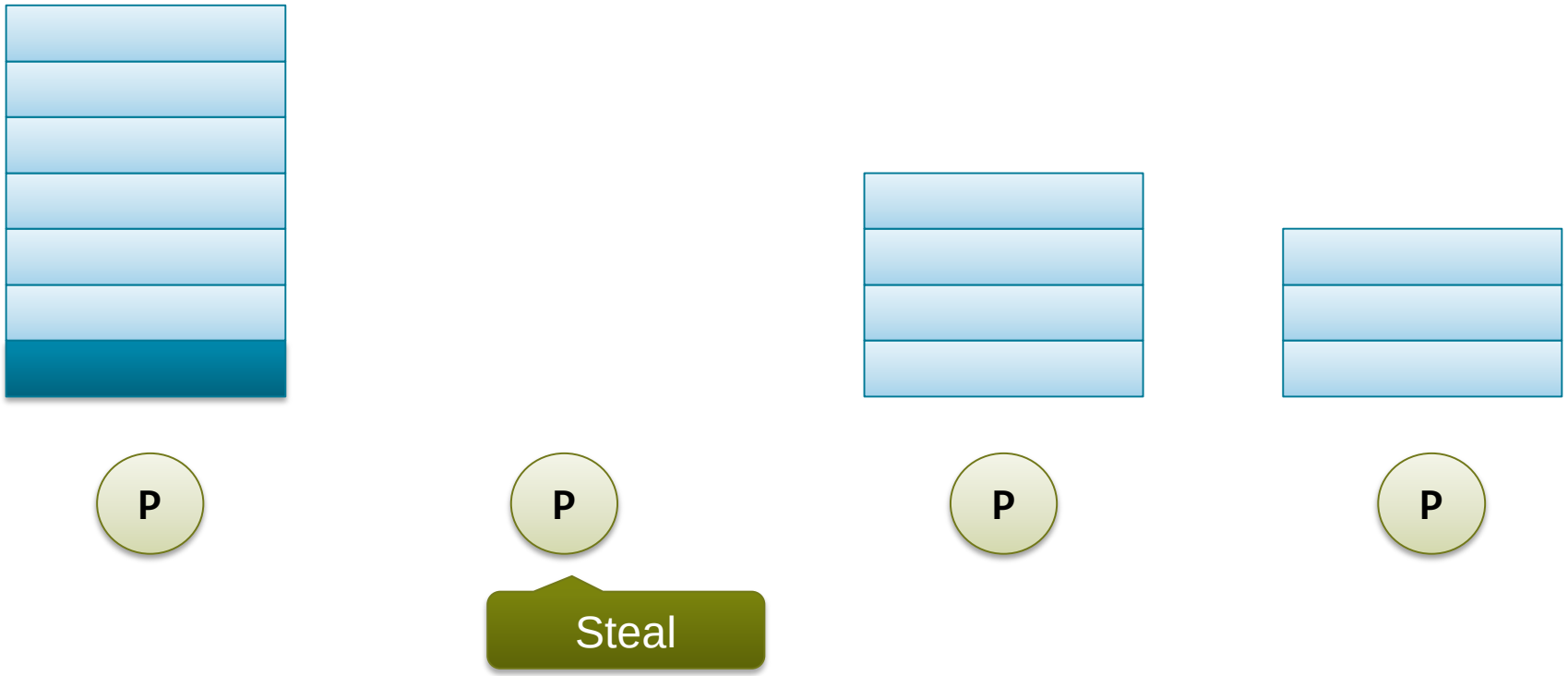
Work Stealing Scheduler



Work Stealing Scheduler

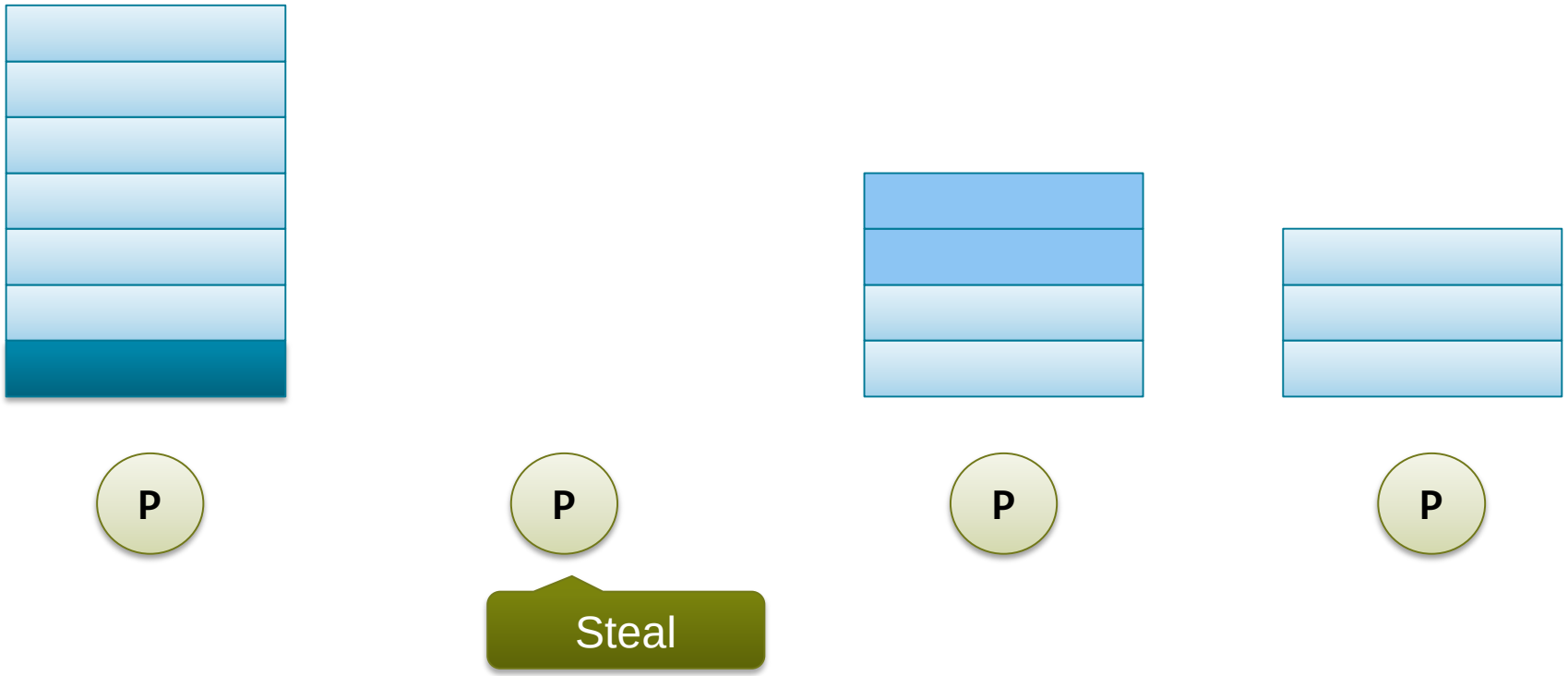


Work Stealing Scheduler

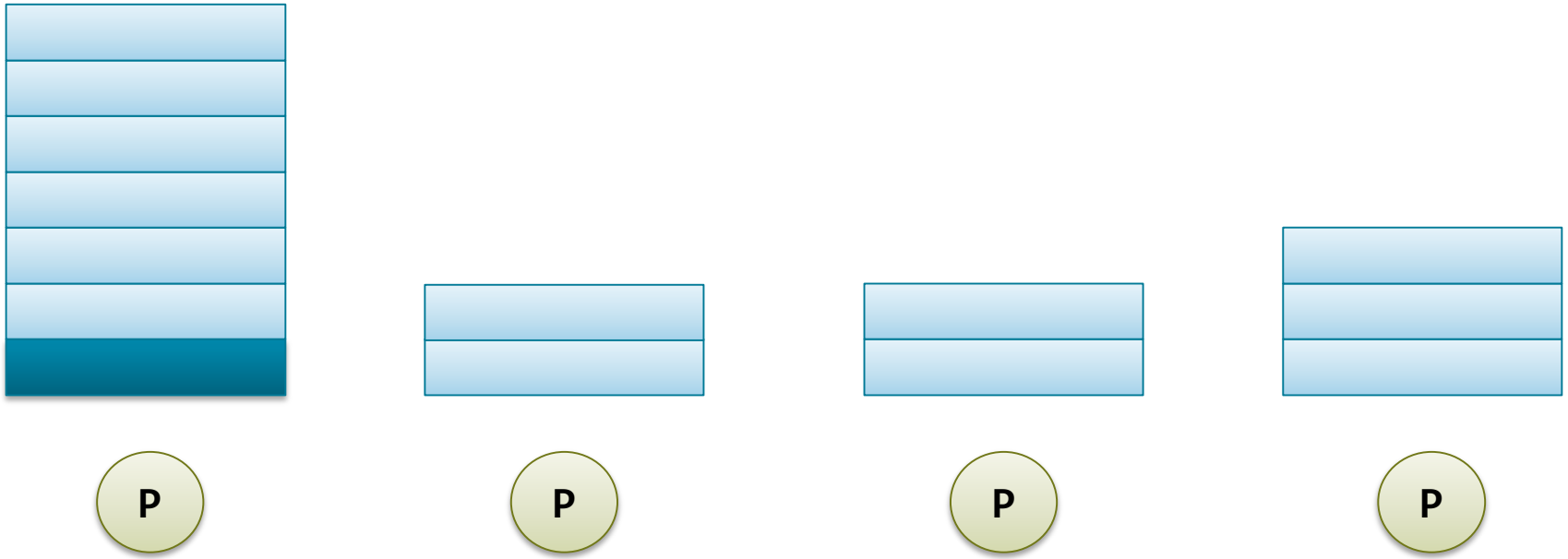


- When a processor runs out of work, it steals a task from the top of a random victim's deque.

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

Matrix Multiplication

ALGORITHM: MATRIX_MULTIPLY(A, B)

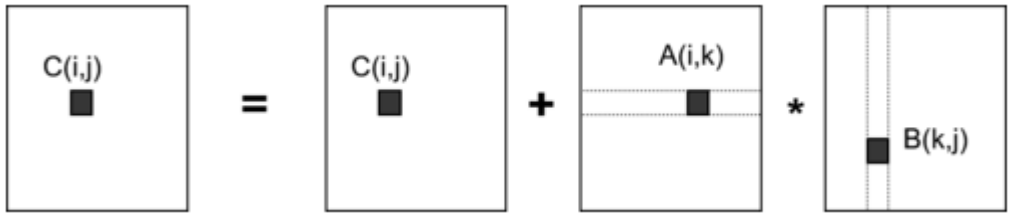
```
1  (l, m) := dimensions(A)
2  (m, n) := dimensions(B)
3  in parallel for i ∈ [0..l) do
4      in parallel for j ∈ [0..n) do
5          R_ij := sum({A_ik * B_kj : k ∈ [0..m)})
6  return R
```

Work?
$O(n^3)$
Depth?
$O(logn)$

Can we do better?
E.g., Stressen $O(n^{2.81})$

Much more parallelism than needed!

```
for i = 1 to n
    {read row i of A into fast memory}              - n² reads
    for j = 1 to n
        {read C(i,j) into fast memory}              - n² reads
        {read column j of B into fast memory}       - n³ reads
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
        {write C(i,j) back to slow memory}          - n² writes
```



Partition A by rows and B by cols

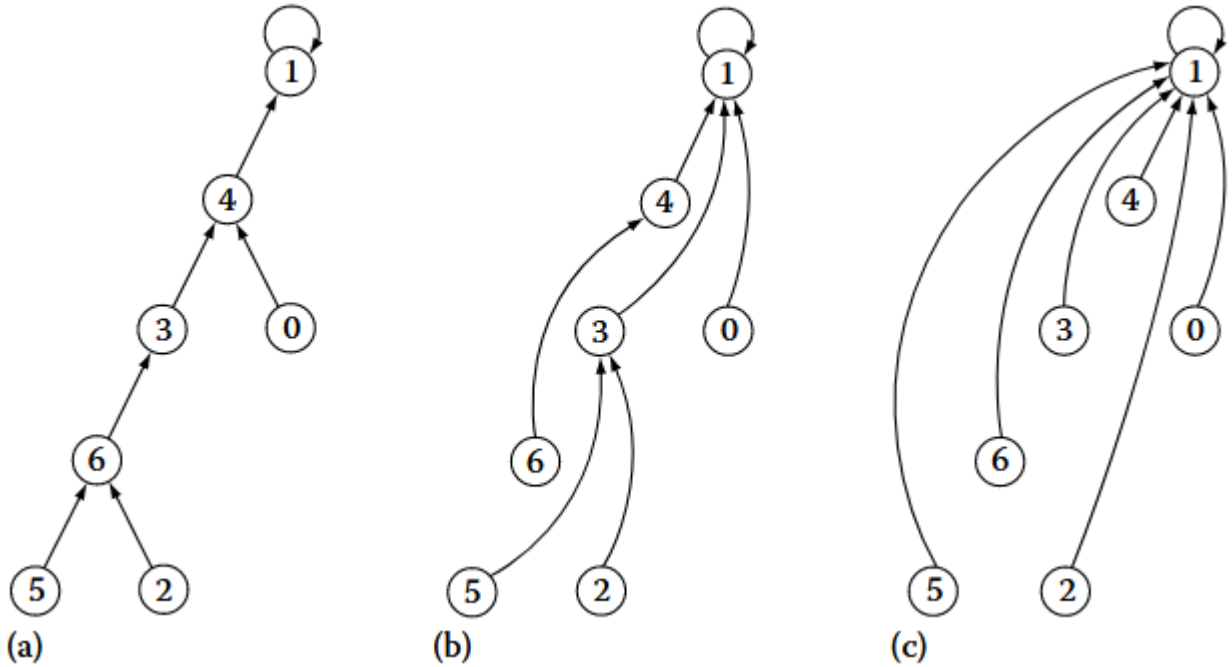
Research focused on how to reduce communication: Communication Avoiding Algorithms

Pointer Jumping

ALGORITHM: POINT_TO_ROOT(P)
1 **for** j **from** 1 **to** $\lceil \log |P| \rceil$
2 $P := \{P[P[i]] : i \in [0..|P|)\}$

Applicable to lists or trees.
At the end of the loop all nodes point to their root.

Many use cases: e.g., connected components.



Work? $O(nlogn)$

Depth? $O(logn)$