Shared-memory Parallel Programming with Cilk Plus

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Outline for Today

- Threaded programming models
- Introduction to Cilk Plus
  - —tasks
  - —algorithmic complexity measures
  - —scheduling
  - —performance and granularity
  - —task parallelism examples
    - – vector addition using divide and conquer
    - – nqueens: exploratory search

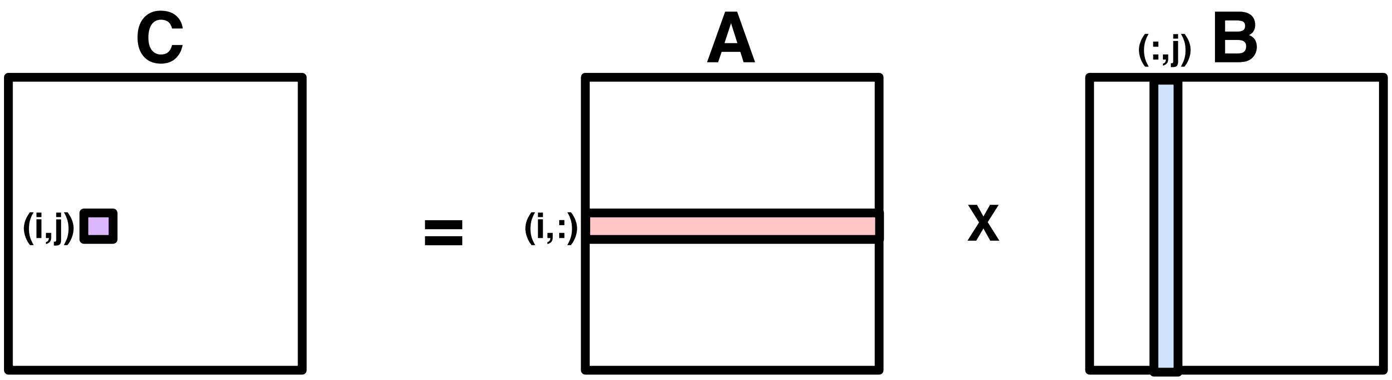
What is a Thread?

- Thread: an independent flow of control
  - — software entity that executes a sequence of instructions
- Thread requires
  - — program counter
  - — a set of registers
  - — an area in memory, including a call stack
  - — a thread id
- A process consists of one or more threads that share
  - — address space
  - — attributes including user id, open files, working directory, ...

An Abstract Example of Threading

A sequential program for matrix multiply

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    c[i][j] = dot_product(get_row(a, i), get_col(b, j))
```



can be transformed to use multiple threads

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    c[i][col] = spawn dot_product(get_row(a, i), get_col(b, j))
```

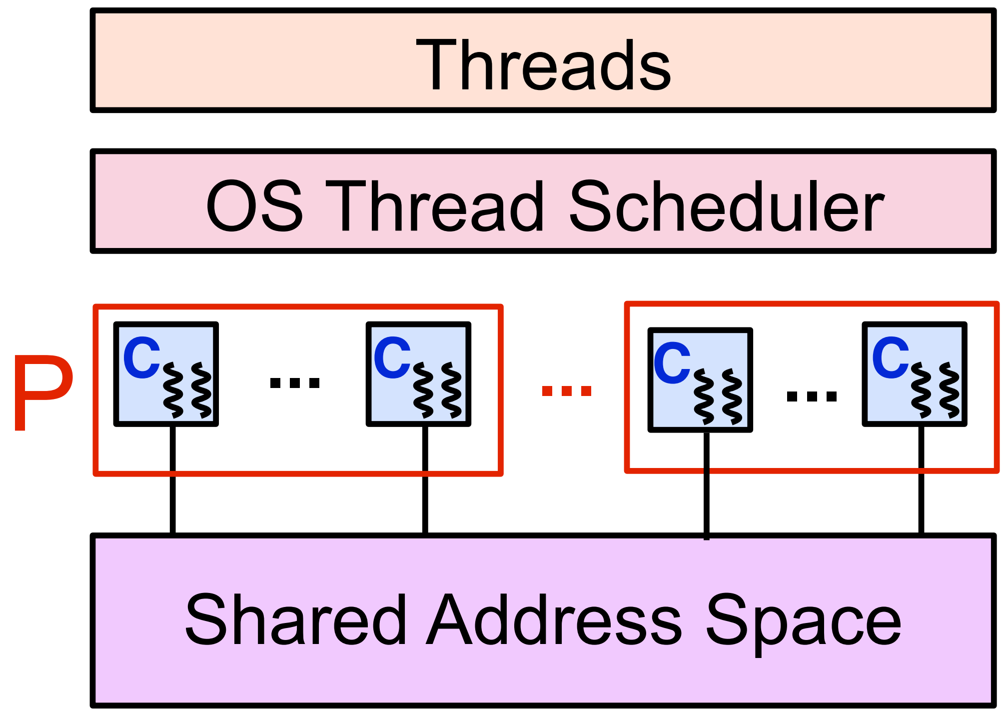
Why Threads?

Well matched to multicore hardware

- **Employ parallelism to compute on shared data**
  - —boost performance on a fixed memory footprint (strong scaling)
- **Useful for hiding latency**
  - —e.g. latency due to memory, communication, I/O
- **Useful for scheduling and load balancing**
  - —especially for dynamic concurrency
- **Relatively easy to program**
  - —easier than message-passing? you be the judge!

Threads and Memory

- **All memory is globally accessible to every thread**
- **Each thread's stack is treated as local to the thread**
- **Additional local storage can be allocated on a per-thread basis**
- **Idealization: treat all memory as equidistant**



Schema for SMP Node

Targets for Threaded Programs

Shared-memory parallel systems

- Multicore processor
- Workstations or cluster nodes with multiple processors
- Xeon Phi manycore processor
  - —about 250 threads
- SGI UV: scalable shared memory system
  - —up to 4096 threads

Threaded Programming Models

- Library-based models
  - —all data is shared, unless otherwise specified
  - —examples: Pthreads, C++11 threads, Intel Threading Building Blocks, Java Concurrency Library, Boost
- Directive-based models, e.g., OpenMP
  - —shared and private data
  - —pragma syntax simplifies thread creation and synchronization
- Programming languages
  - —Cilk Plus (Intel)
  - —CUDA (NVIDIA)
  - —Habanero-Java (Rice/Georgia Tech)

Cilk Plus Programming Model

- A simple and powerful model for writing multithreaded programs
- Extends C/C++ with three new keywords
  - —cilk_spawn: invoke a function (potentially) in parallel
  - —cilk_sync: wait for a procedure's spawned functions to finish
  - —cilk_for: execute a loop in parallel
- Cilk Plus programs specify logical parallelism
  - —what computations can be performed in parallel, i.e., tasks
  - —not mapping of work to threads or cores
- Faithful language extension
  - —if Cilk Plus keywords are elided → C/C++ program semantics
- Availability
  - —Intel compilers
  - —GCC (full in versions 5 — 7; removed in version 8)

Cilk Plus Tasking Example: Fibonacci

Fibonacci sequence

0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 21 34 55 89 144 233 377 610 987

- **Computing Fibonacci recursively**

```
unsigned int fib(unsigned int n)  {
  if (n < 2) return n;
  else {
    unsigned int n1, n2;
    n1 = fib(n-1);
    n2 = fib(n-2);
    return (n1 + n2);
  }
}
```

Cilk Plus Tasking Example: Fibonacci

Fibonacci sequence

0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 … 21 34 55 89 144 233 377 610 987 …

- **Computing Fibonacci recursively in parallel with Cilk Plus**

```
unsigned int fib(unsigned int n)  {
  if (n < 2) return n;
  else {
    unsigned int n1, n2;
    n1 = cilk_spawn fib(n-1);
    n2 = fib(n-2);
    cilk_sync;
    return (n1 + n2);
  }
}
```

Cilk Plus Terminology

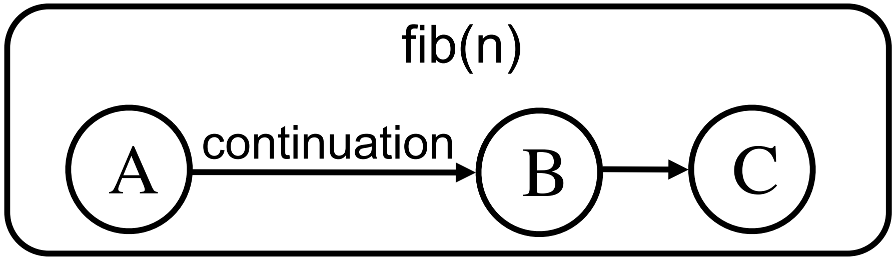
- Parallel control
  - cilk_spawn, cilk_sync
  - return from spawned function
- Strand
  - maximal sequence of instructions not containing parallel control

```
unsigned int fib(n)  {
  if (n < 2) return n;
  else {
    unsigned int n1, n2;
    n1 = cilk_spawn fib(n - 1);
    n2 = cilk_spawn fib(n - 2);
    cilk_sync;
   return (n1 + n2);
  }
}
```

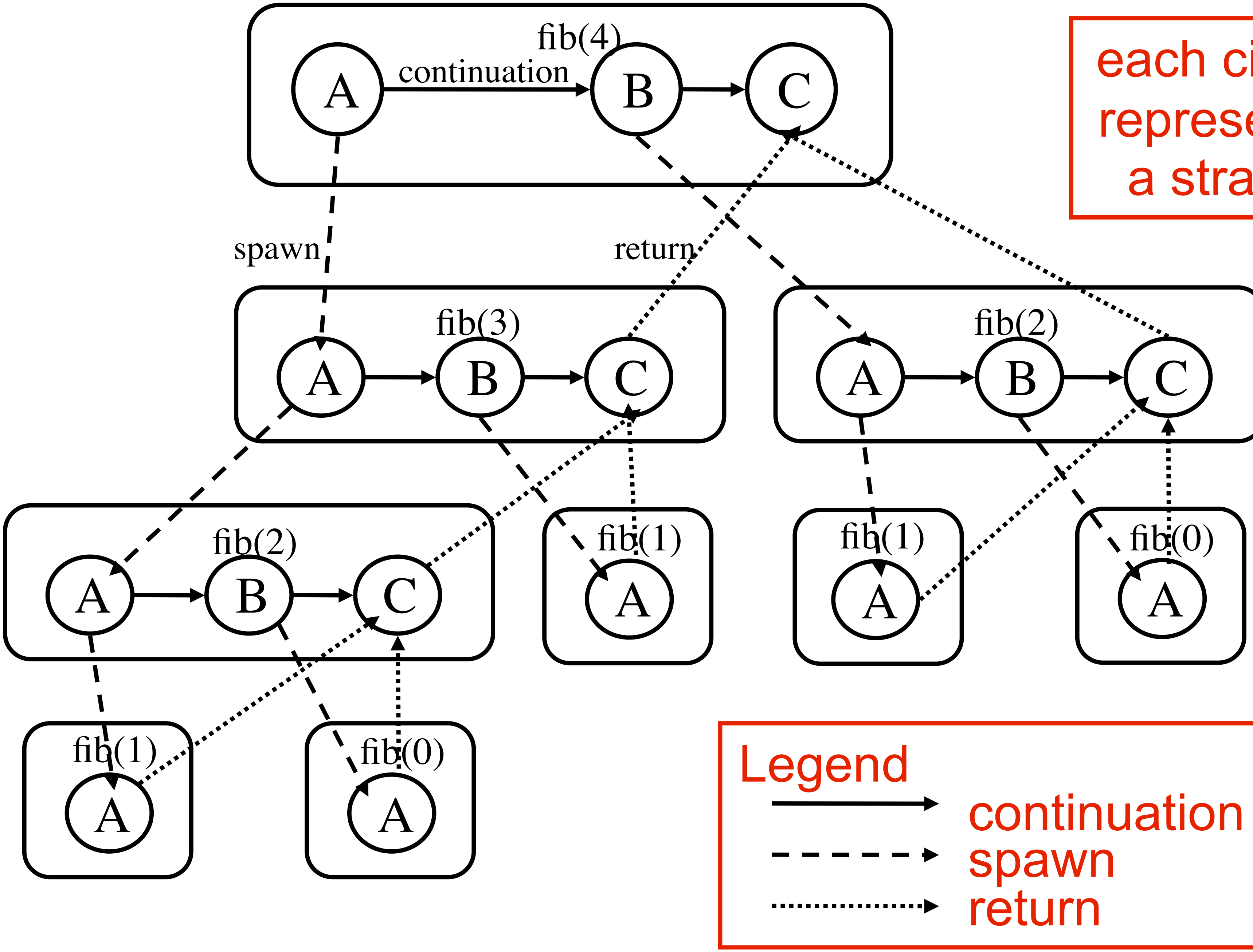
Strand A: code before first spawn

Strand B: compute n-2 before 2nd spawn

Strand C: n1+ n2 before the return



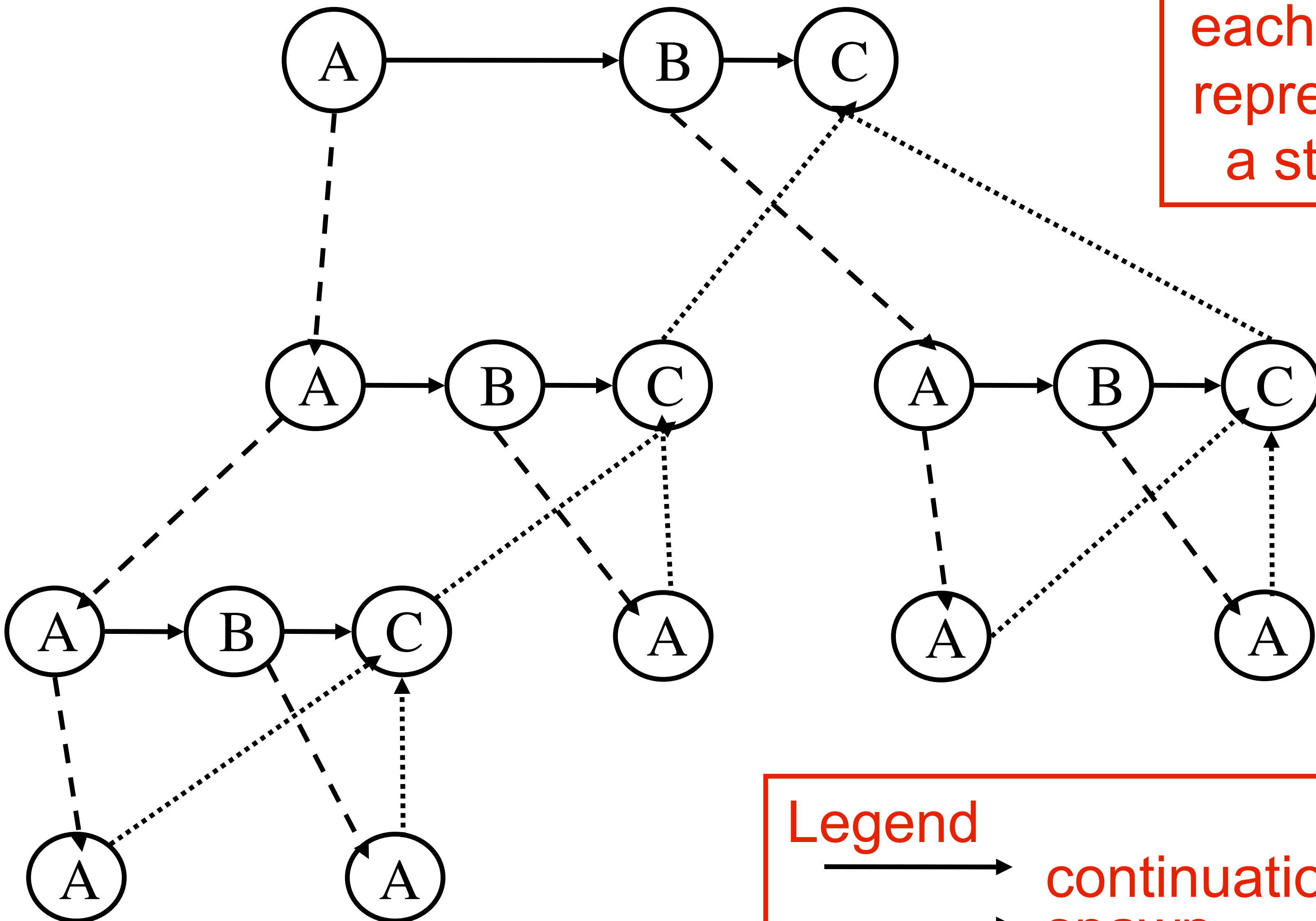
Cilk Program Execution as a DAG



each circle represents a strand

Cilk Program Execution as a DAG

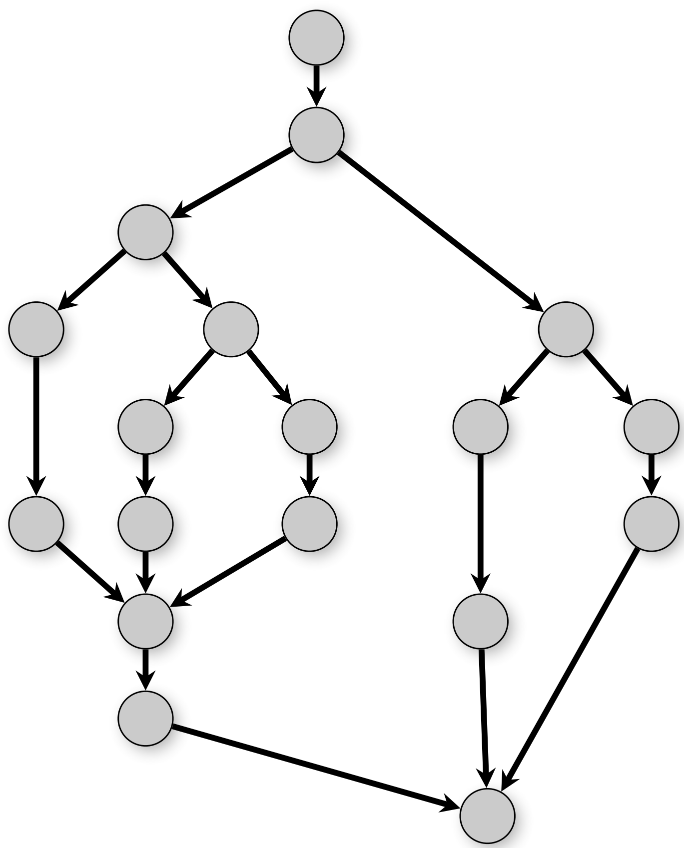
each circle represents a strand

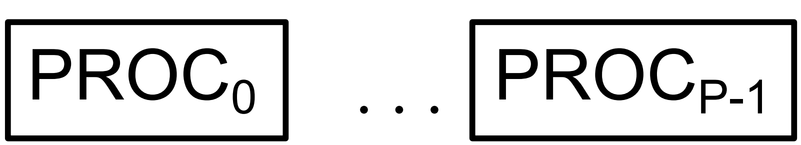


Legend
continuation
spawn
return

Algorithmic Complexity Measures

T_P = execution time on P processors





Computation graph abstraction:

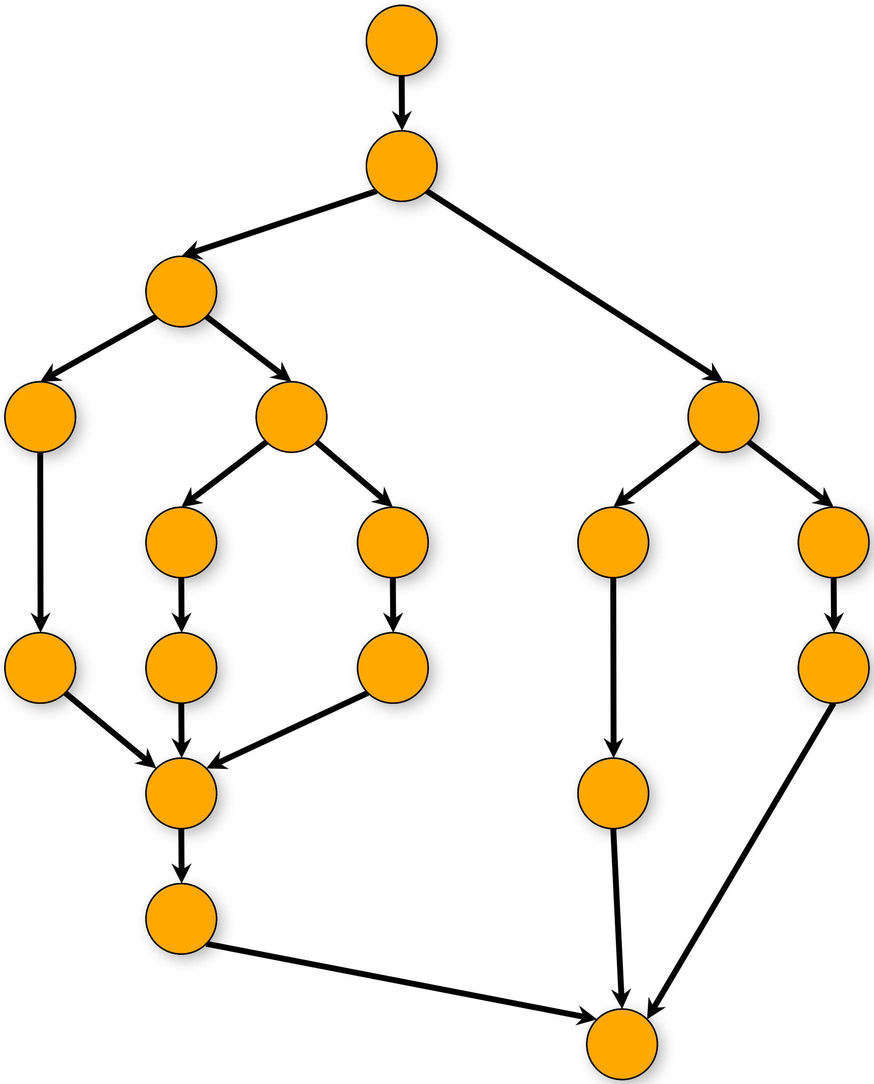
- node = arbitrary sequential computation
- edge = dependence (successor node can only execute after predecessor node has completed)
- Directed Acyclic Graph (DAG)

Processor abstraction:

- P identical processors
- each processor executes one node at a time

Algorithmic Complexity Measures

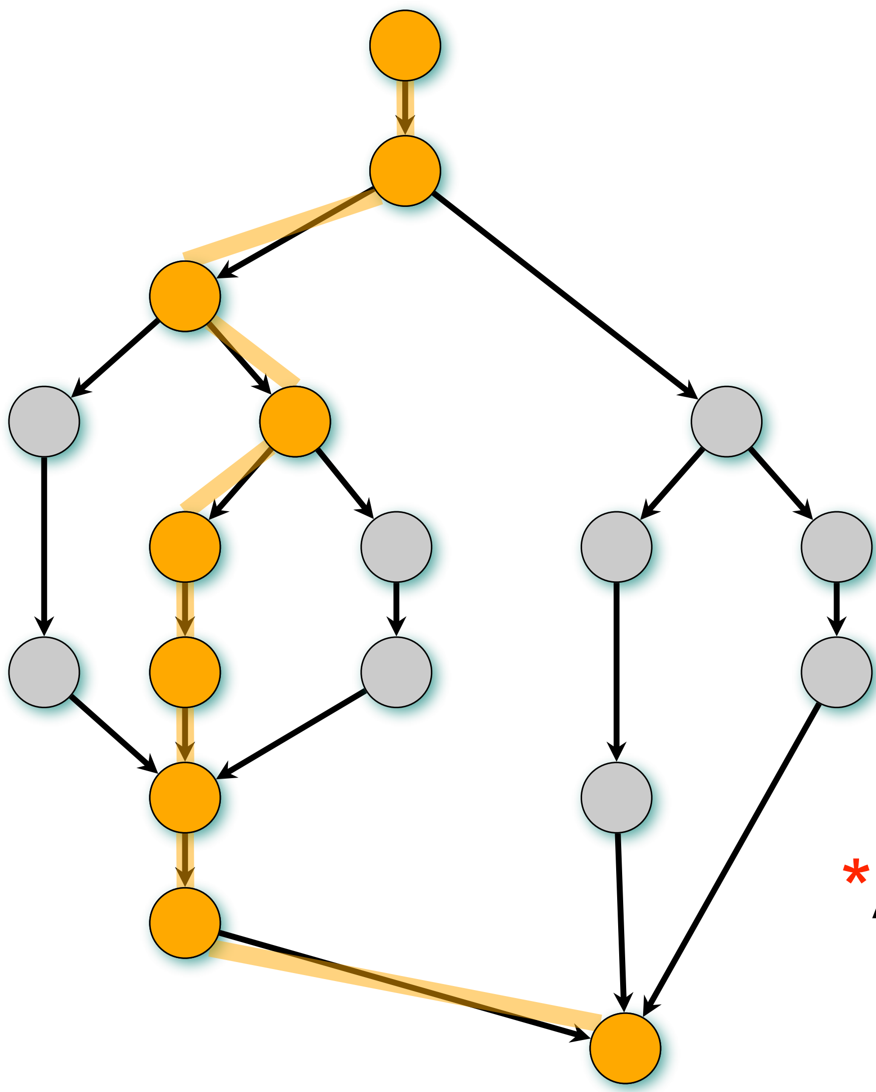
T_P = execution time on P processors



T_1 = *work*

Algorithmic Complexity Measures

T_P = execution time on P processors



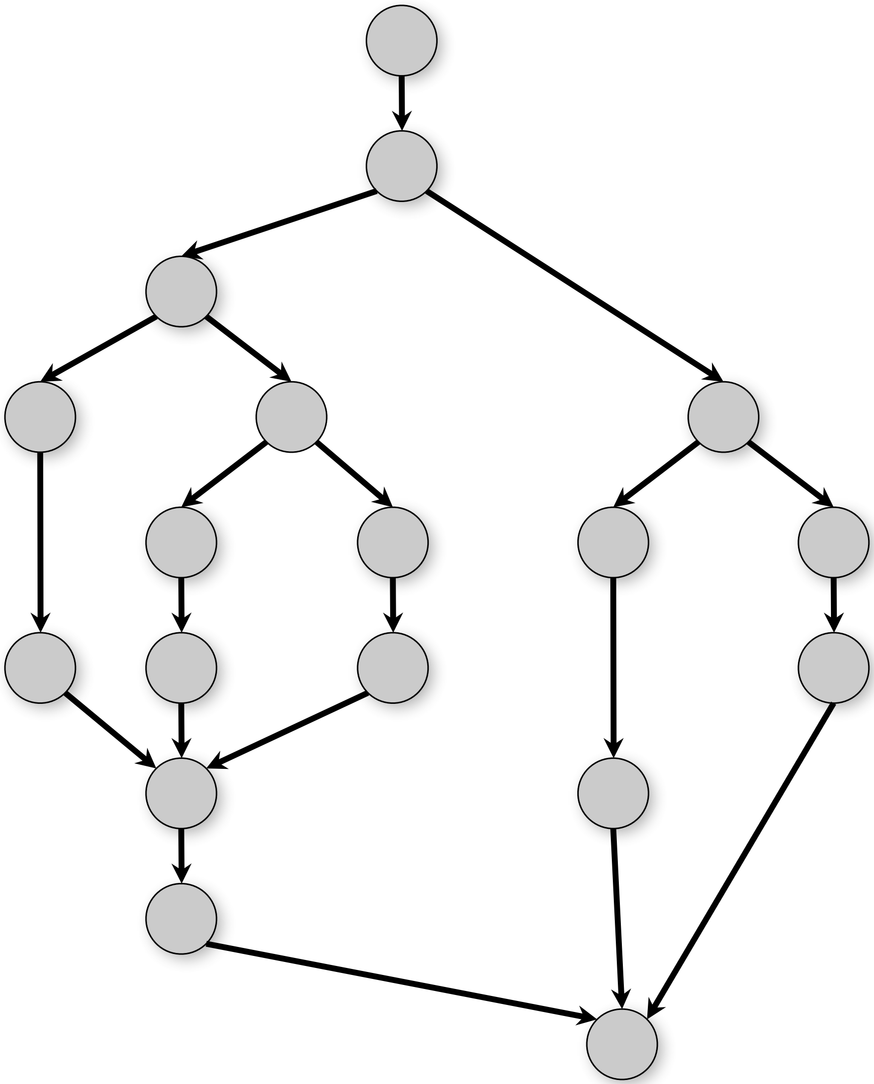
T_1 = *work*

T_∞ = *span**

*Also called *critical-path length*

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = *work*

T_∞ = *span*

LOWER BOUNDS

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

Speedup

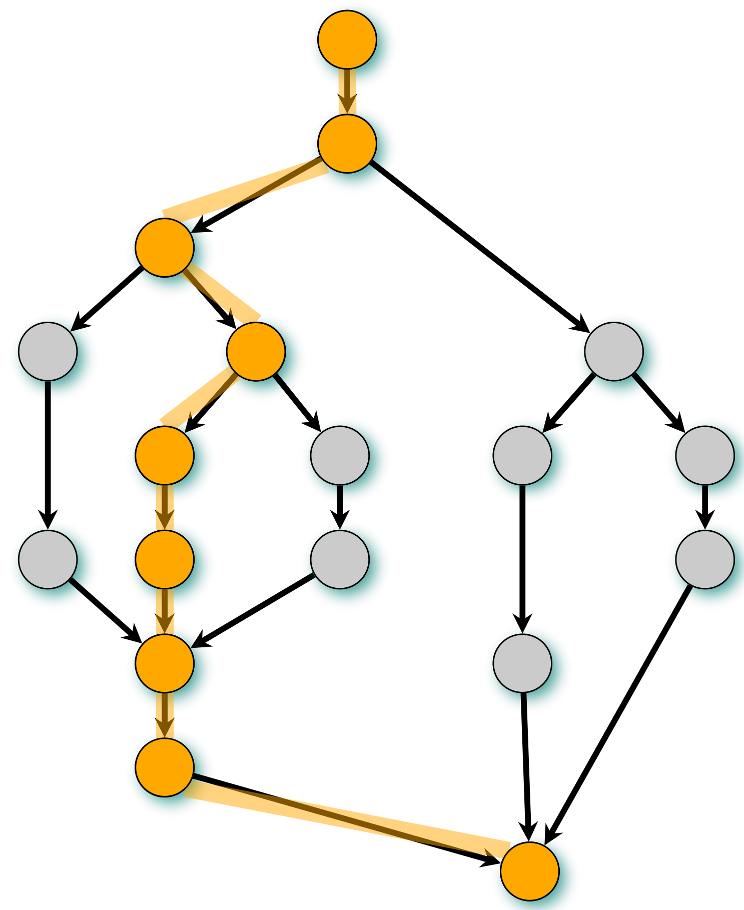
Definition: T_1/T_P **=** ***speedup*** **on** P **processors**

If T_1/T_P = $\Theta(P)$, we have *linear speedup*;
= P, we have *perfect linear speedup*;
> P, we have *superlinear speedup*,

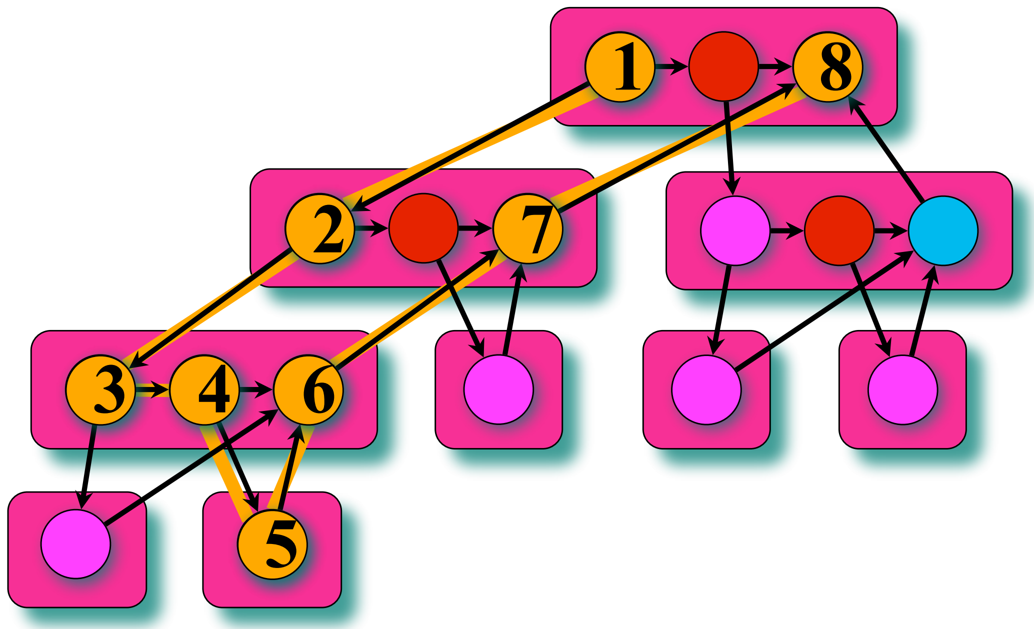
Superlinear speedup is not possible in this model because of the lower bound $T_P \geq T_1/P$, but it can occur in practice (e.g., due to cache effects)

Parallelism ("Ideal Speedup")

- T_P depends on the <u>schedule</u> of computation graph nodes on the processors
  - two different schedules can yield different values of T_P for the same P
- For convenience, define *parallelism* (or ideal speedup) as the ratio T_1/T_∞
- Parallelism is independent of P, and only depends on the computation graph
- Also define *parallel slackness* as the ratio, $(T_1/T_\infty)/P$; the larger the slackness, the less the impact of T_∞ on performance



Example: `fib(4)`

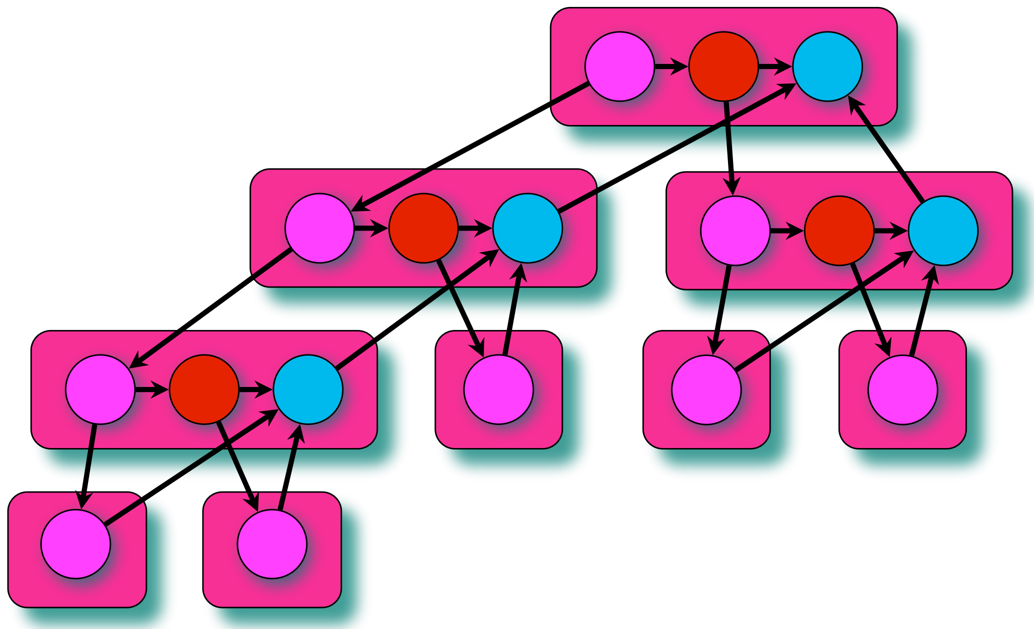


Assume for simplicity that each strand in **fib()** *takes unit time to execute.*

Work: T_1 = 17 (T_P refers to execution time on P processors)

Span: T_∞ = 8 (Span = “critical path length”)

Example: `fib(4)`



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

Work: $T_1 = 17$

Span: $T_\infty = 8$

Ideal Speedup: $T_1 / T_\infty = 2.125$

Using more than 2 processors makes little sense

Task Scheduling

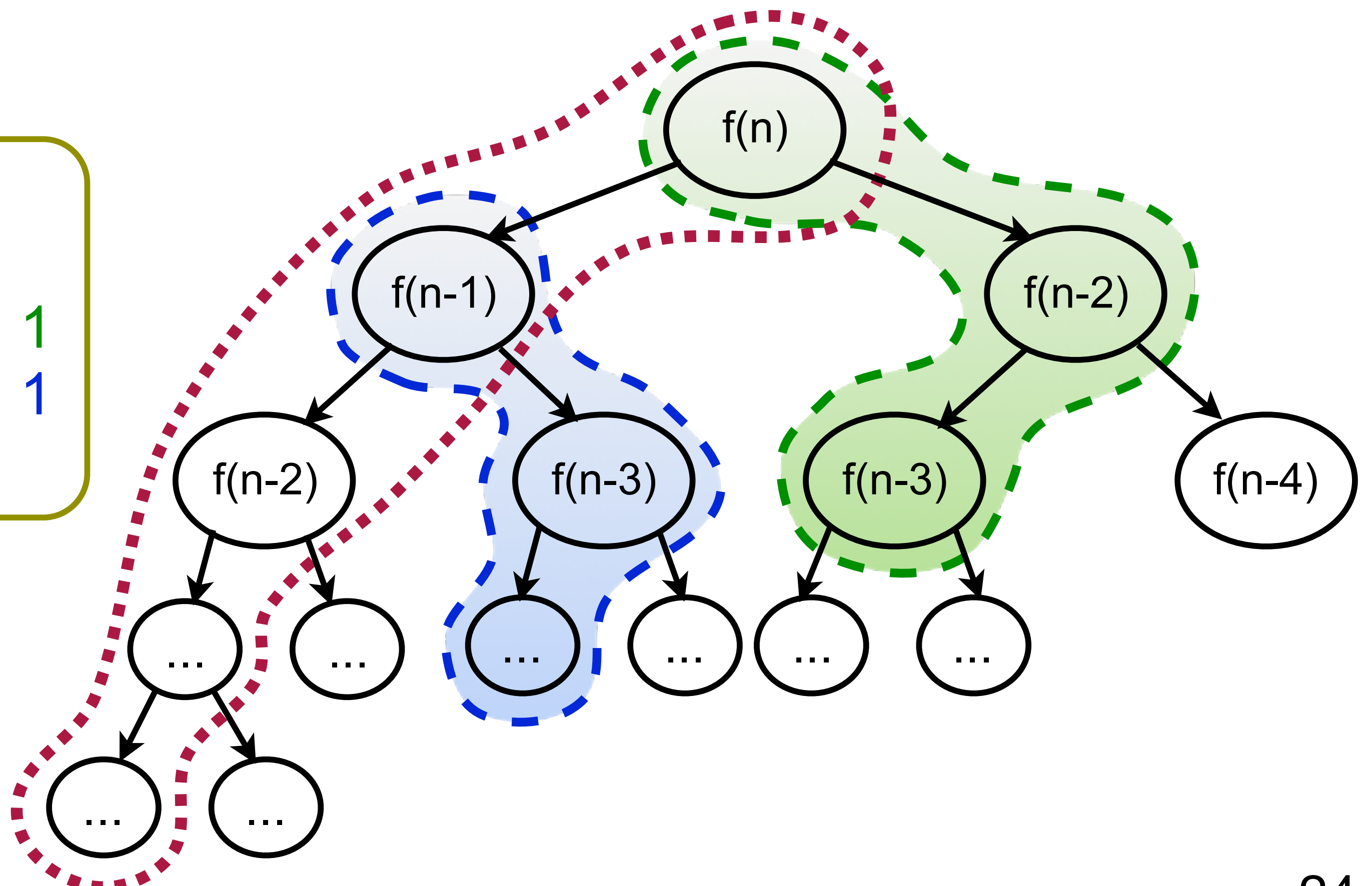
- Popular scheduling strategies
  - —work-sharing: task scheduled to run in parallel at every spawn
    - benefit: maximizes parallelism
    - drawback: cost of setting up new tasks is high → should be avoided
  - —work-stealing: processor looks for work when it becomes idle
    - lazy parallelism: put off setting up parallel execution until necessary
    - benefits: executes with precisely as much parallelism as needed
      minimizes the number of tasks that must be set up
      runs with same efficiency as serial program on uniprocessor
- Cilk uses work-stealing rather than work-sharing

Cilk Execution using Work Stealing

- **Cilk runtime maps logical tasks to compute cores**
- **Approach:**
  - **— lazy task creation plus work-stealing scheduler**
    - **`cilk_spawn`: a potentially parallel task is available**
    - **an idle thread steals a task from a random working thread**

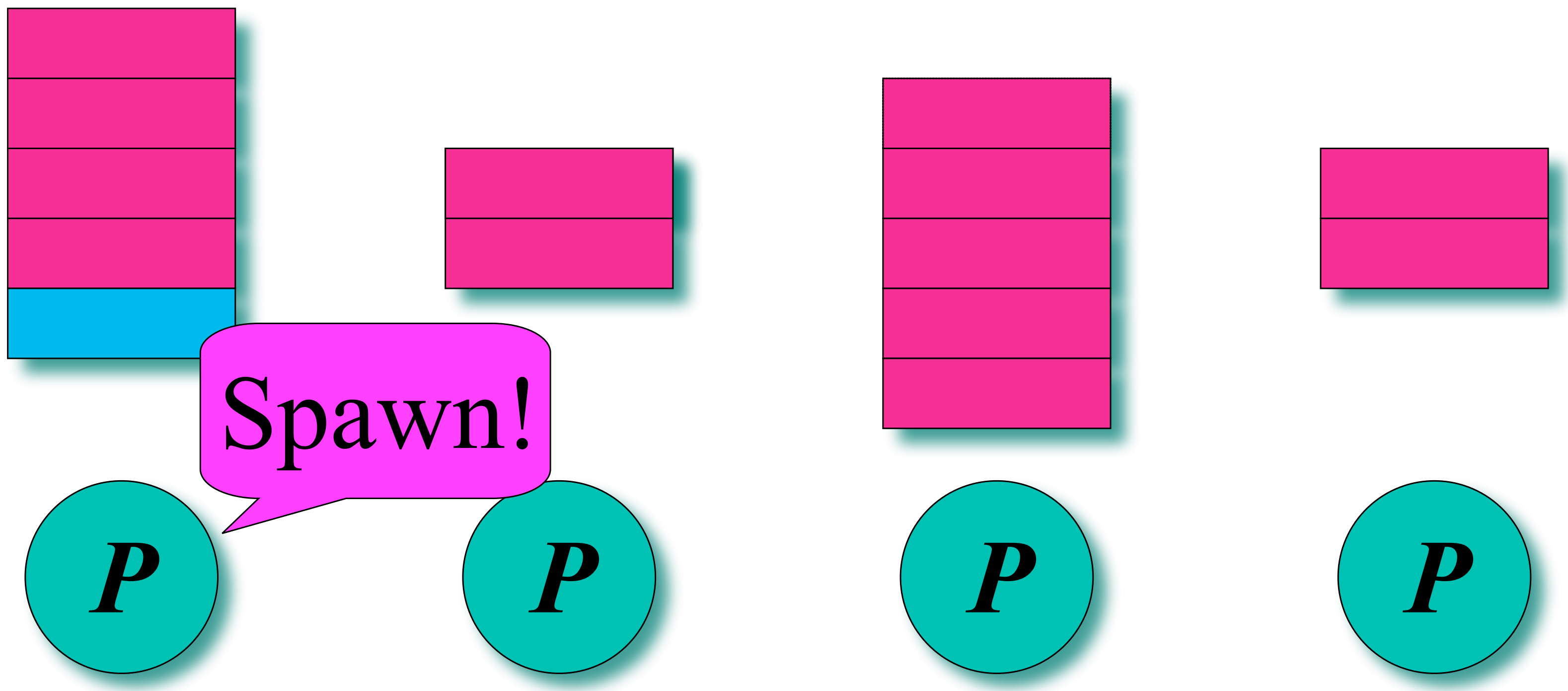
Possible Execution:

thread 1 begins
thread 2 steals from 1
thread 3 steals from 1
etc**...**



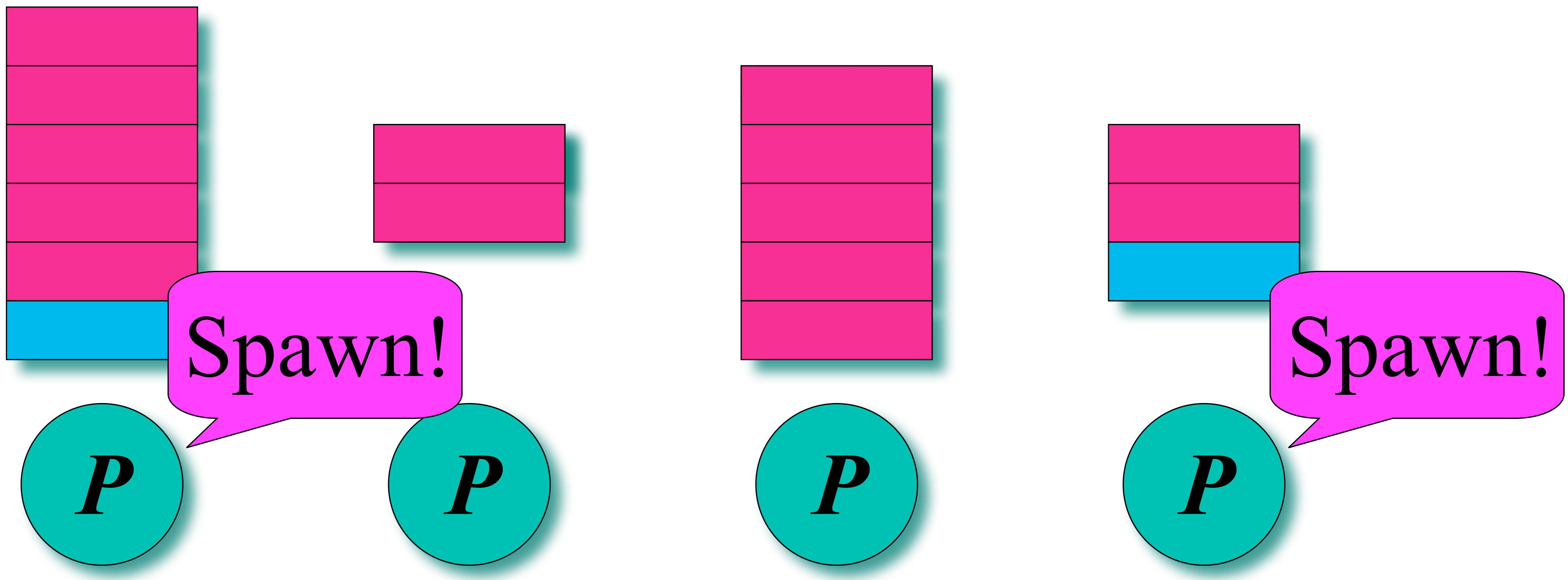
Cilk's Work-Stealing Scheduler

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



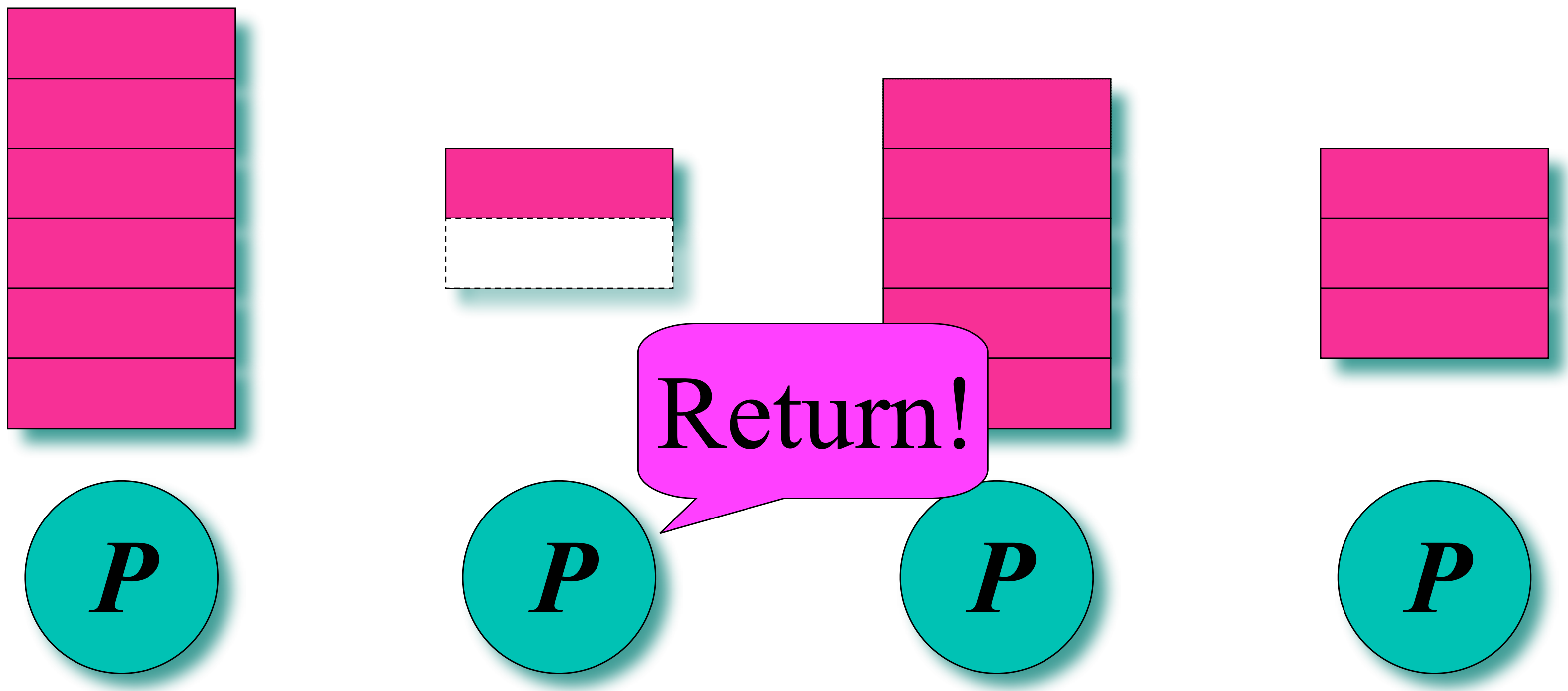
Cilk's Work-Stealing Scheduler

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



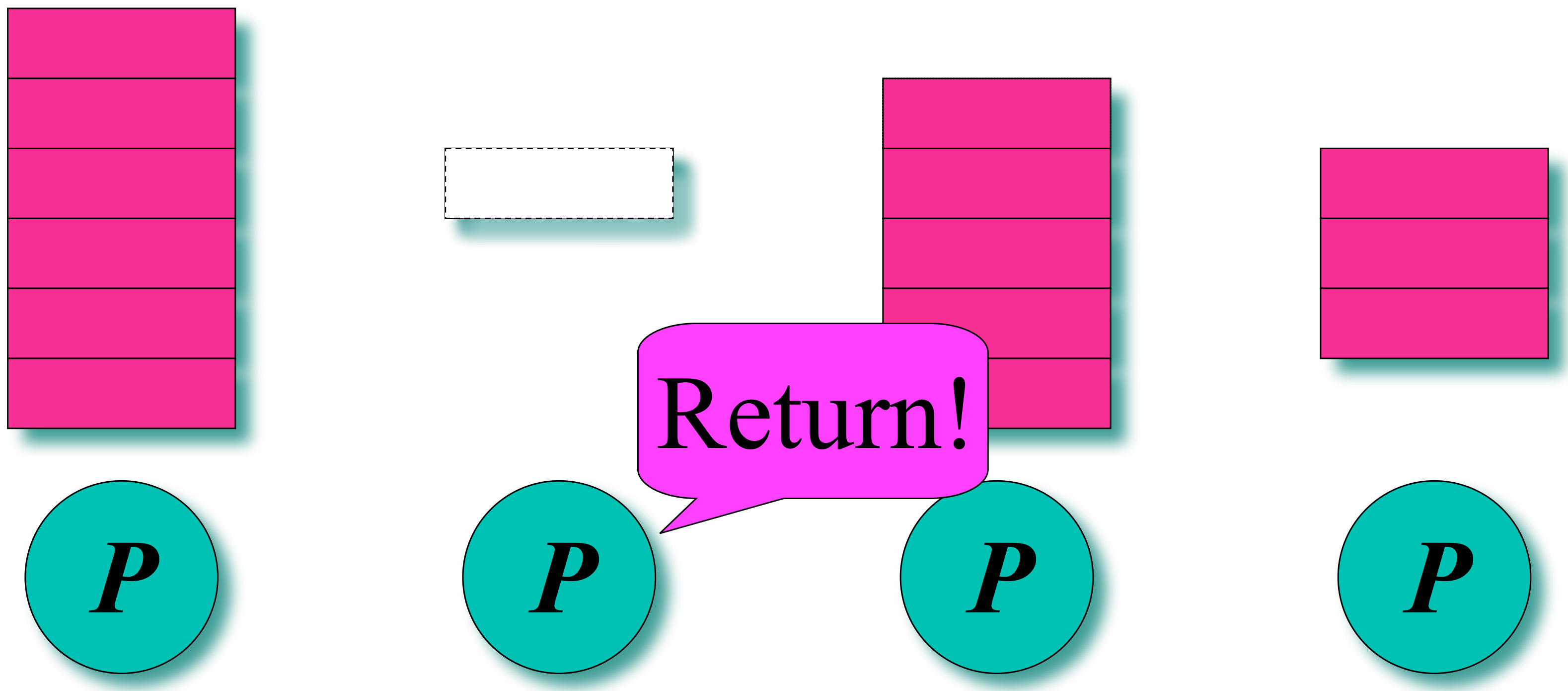
Cilk's Work-Stealing Scheduler

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



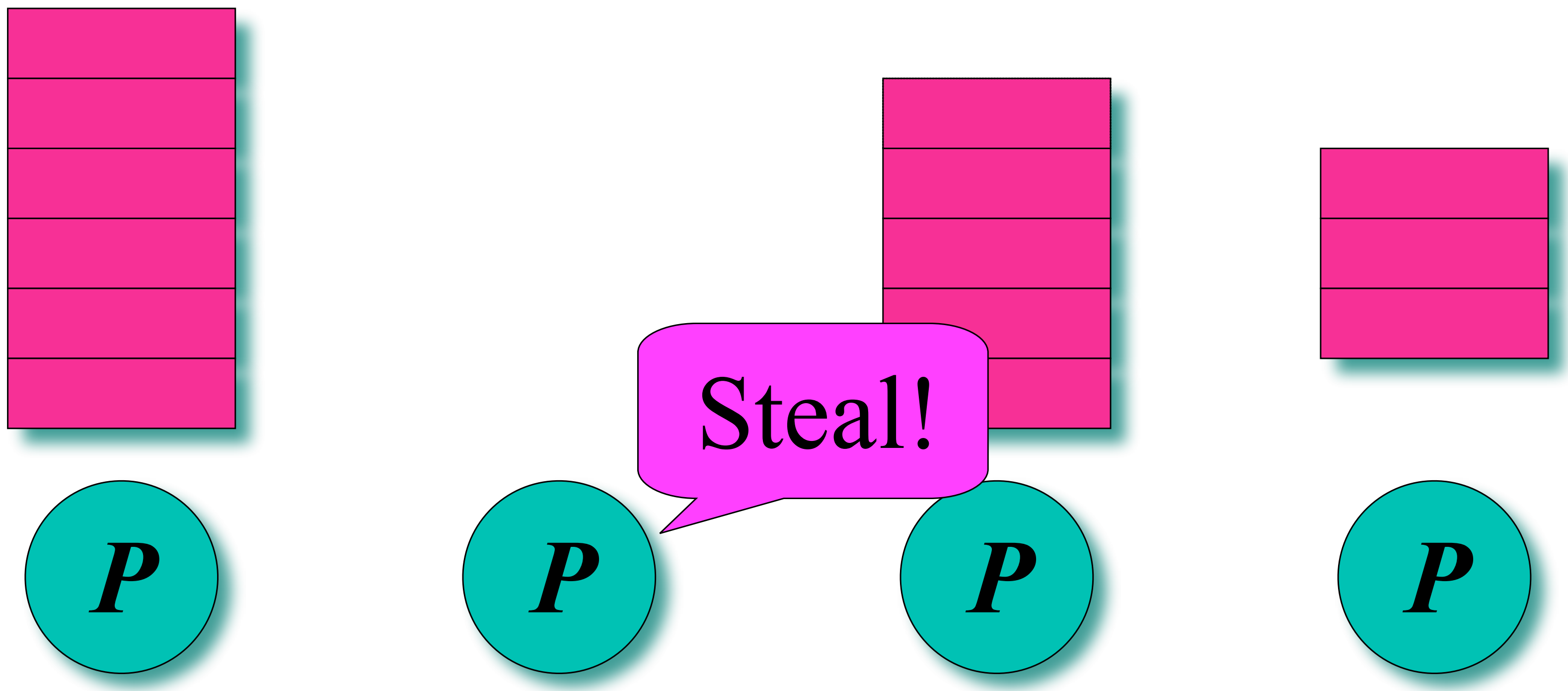
Cilk's Work-Stealing Scheduler

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

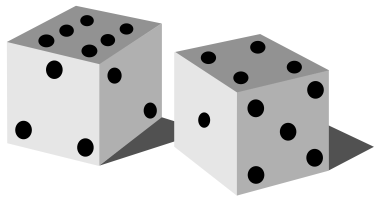


Cilk's Work-Stealing Scheduler

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

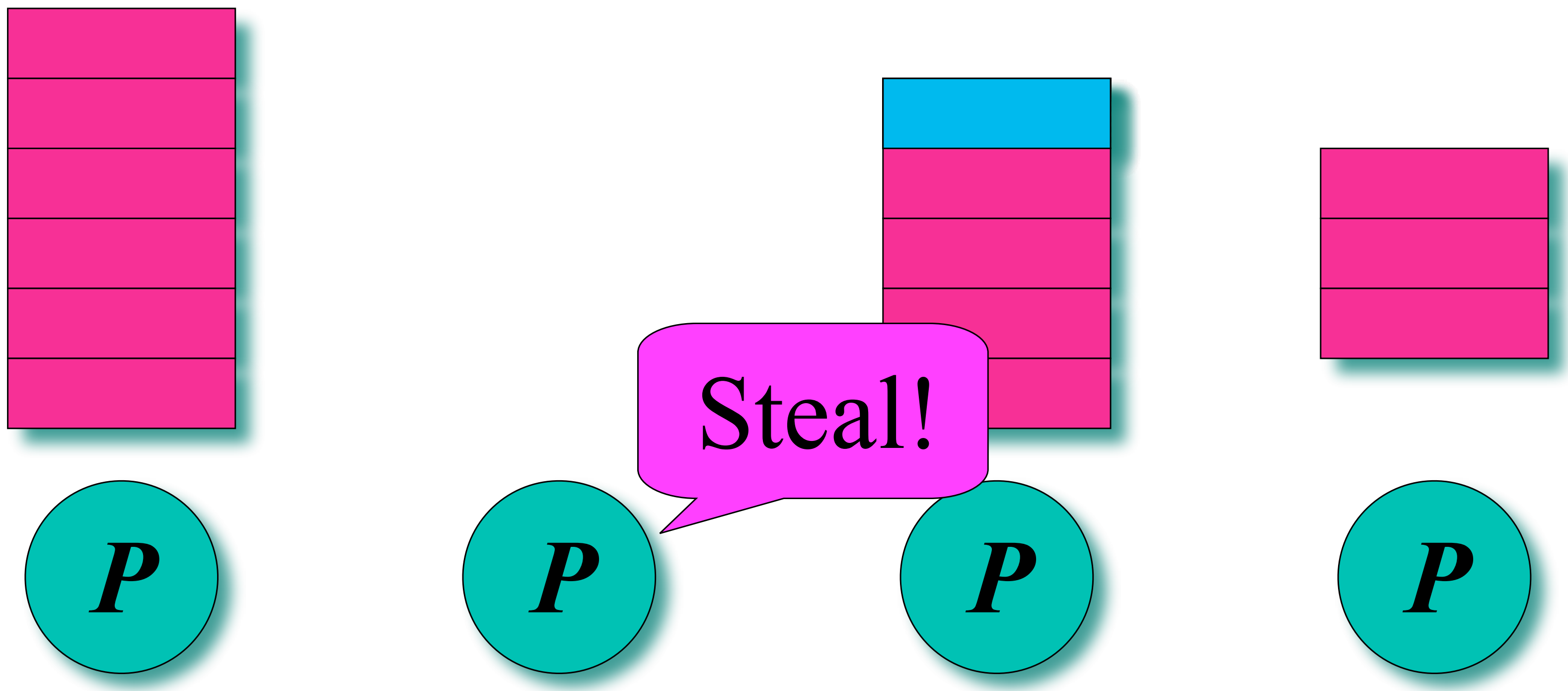


When a processor runs out of work, it ***steals*** a strand from the top of a ***random*** victim's deque

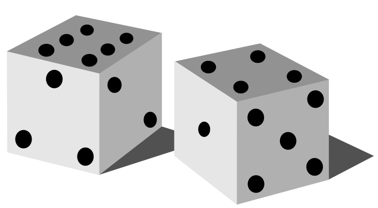


Cilk's Work-Stealing Scheduler

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

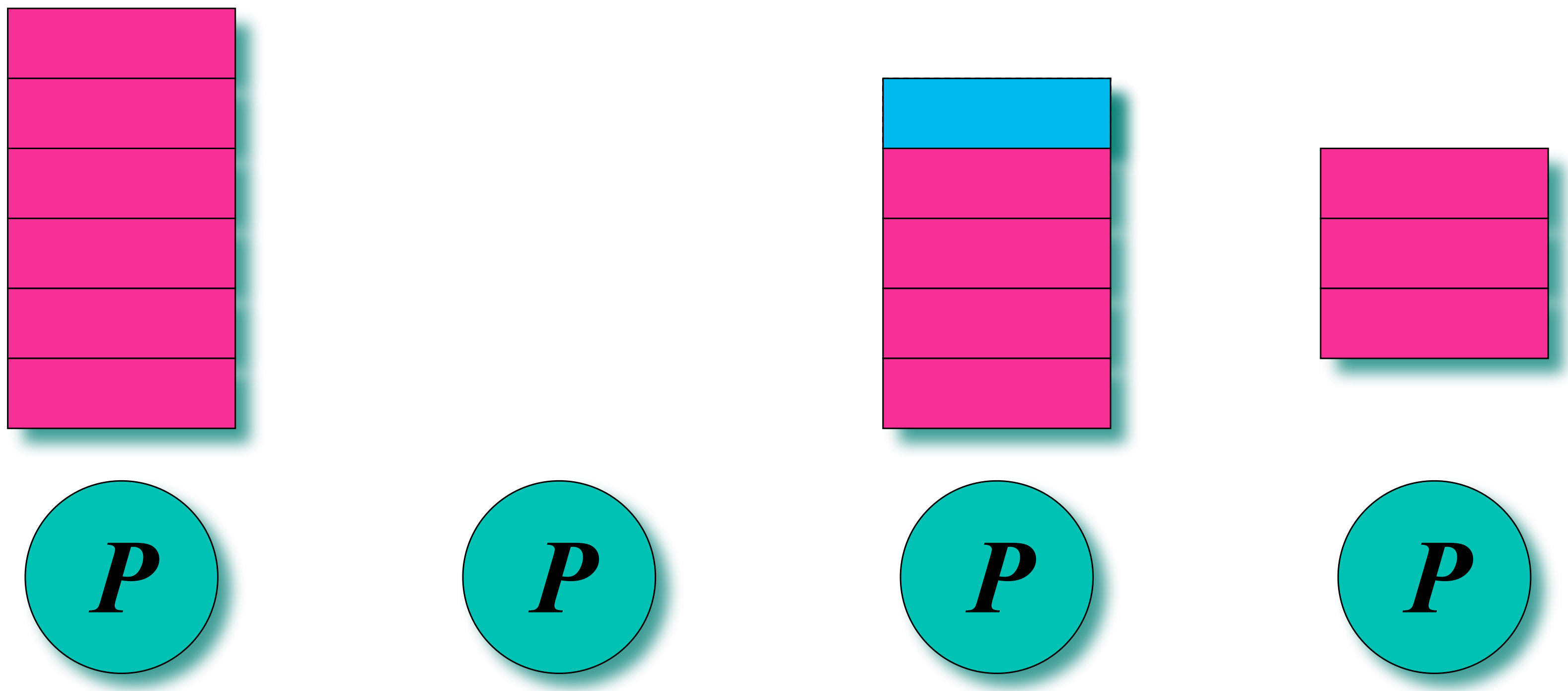


When a processor runs out of work, it ***steals*** a strand from the top of a ***random*** victim's deque

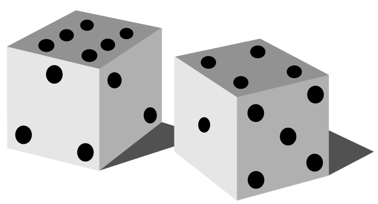


Cilk's Work-Stealing Scheduler

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

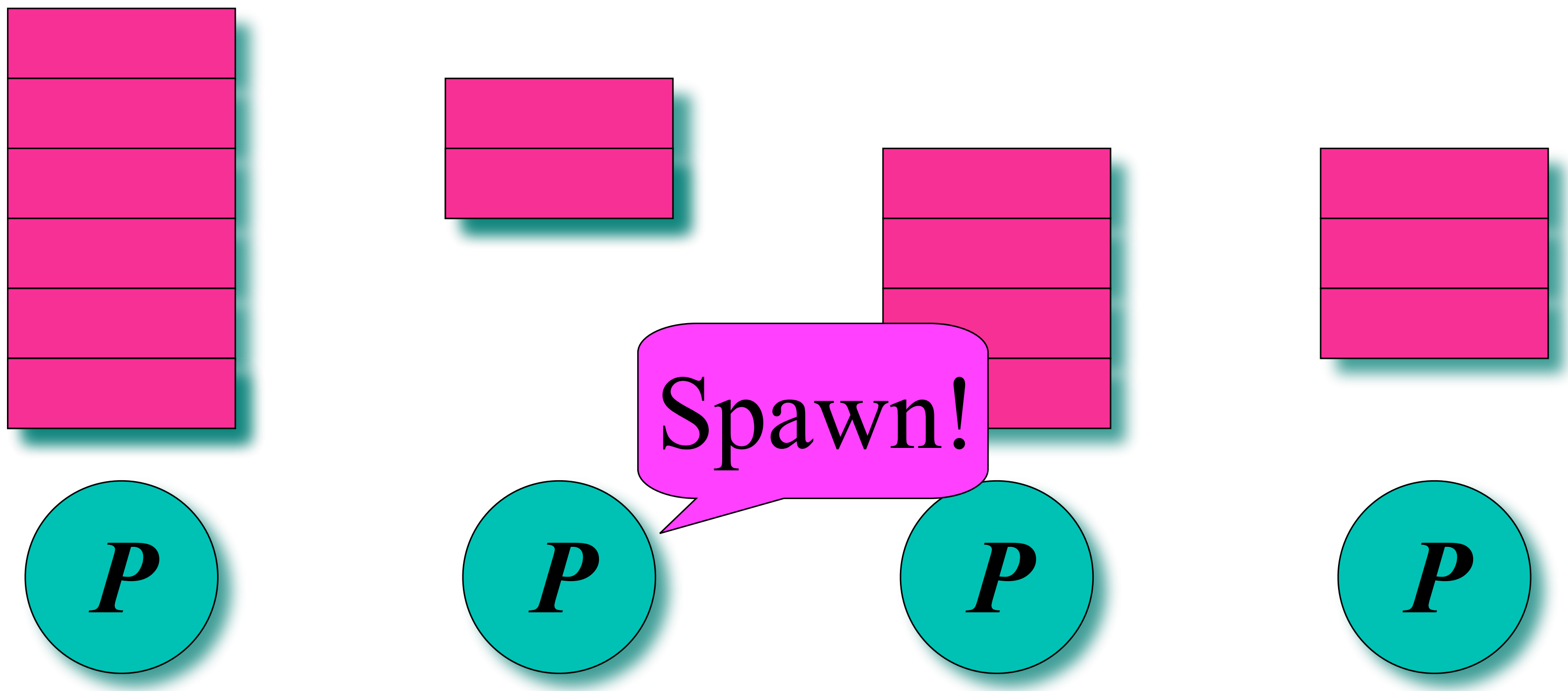


When a processor runs out of work, it ***steals*** a strand from the top of a ***random*** victim's deque

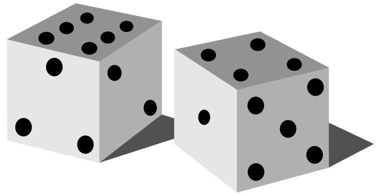


Cilk's Work-Stealing Scheduler

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



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



Performance of Work-Stealing

Theorem: Cilk's work-stealing scheduler achieves an expected running time of $T_P \le T_1/P + O(T_\infty)$ on P processors

Greedy Scheduling Theorem

- Types of schedule steps
  - complete step
    - at least P operations ready to run
    - select any P and run them
  - incomplete step
    - strictly < P operation ready to run
    - greedy scheduler runs them all

Theorem: On P processors, a greedy scheduler executes any computation G with work T_1 and critical path of length T_∞ in time $T_p \leq T_1/P + T_\infty$

Proof sketch

- only two types of scheduler steps: complete, incomplete
- cannot be more than T_1/P complete steps, else work > T_1
- every incomplete step reduces remaining critical path length by 1
  - no more than T_∞ incomplete steps

Parallel Slackness Revisited

critical path overhead = smallest constant c_∞ such that

$$T_p \le \frac{T_1}{P} + c_\infty T_\infty$$

$$T_p \le \left(\frac{T_1}{T_\infty P} + c_\infty\right) T_\infty = \left(\frac{\overline{P}}{P} + c_\infty\right) T_\infty$$

Let $\overline{P}$ = T_1/T_∞ = parallelism = max speedup on ∞ processors

Parallel slackness assumption

$$\overline{P}/P >> c_\infty$$

thus

$$\frac{T_1}{P} >> c_\infty T_\infty$$

$$T_p \approx \frac{T_1}{P}$$

linear speedup

"critical path overhead has little effect on performance when sufficient parallel slackness exists"

Work Overhead

$$c_1 = \frac{T_1}{T_s}$$

work overhead

"Minimize work overhead (c_1) at the expense of a larger critical path overhead (c_∞), because work overhead has a more direct impact on performance"

$$T_p \le c_1 \frac{T_s}{P} + c_\infty T_\infty$$

$$T_p \approx c_1 \frac{T_s}{P}$$

assuming parallel slackness

You can reduce C_1 by increasing the granularity of parallel work

Parallelizing Vector Addition

C

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

Divide and Conquer

- An effective parallelization strategy
  - —creates a good mix of large and small sub-problems
- Work-stealing scheduler can allocate chunks of work efficiently to the cores, as long as
  - —not only a few large chunks
    - – if work is divided into just a few large chunks, there may not be enough parallelism to keep all the cores busy
  - —not too many very small chunks
    - – if the chunks are too small, then scheduling overhead may overwhelm the benefit of parallelism

Parallelizing Vector Addition

C

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

C

```
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);
  }
}
```

Parallelization strategy:

1. Convert loops to recursion.

Parallelizing Vector Addition

C

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

Cilk Plus

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

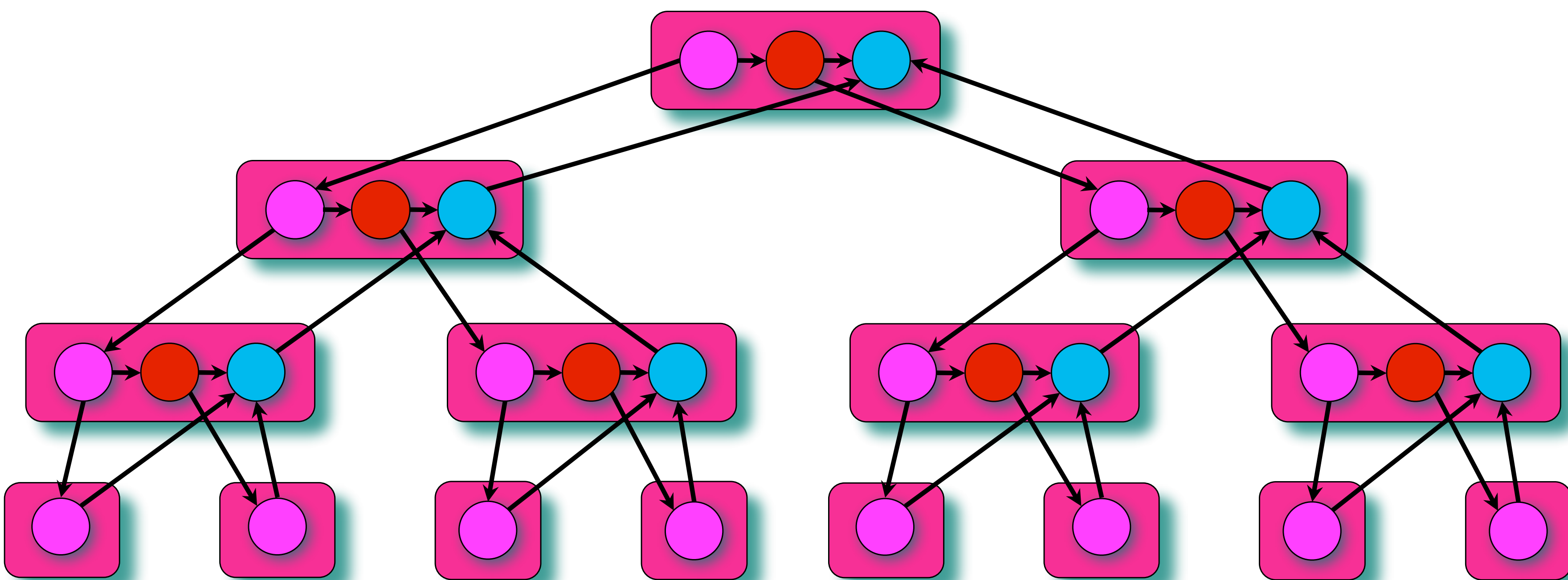
Parallelization strategy:

1. Convert loops to recursion.
2. Insert Cilk Plus keywords.

Side benefit:
D&C is generally good for caches!

Vector Addition

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



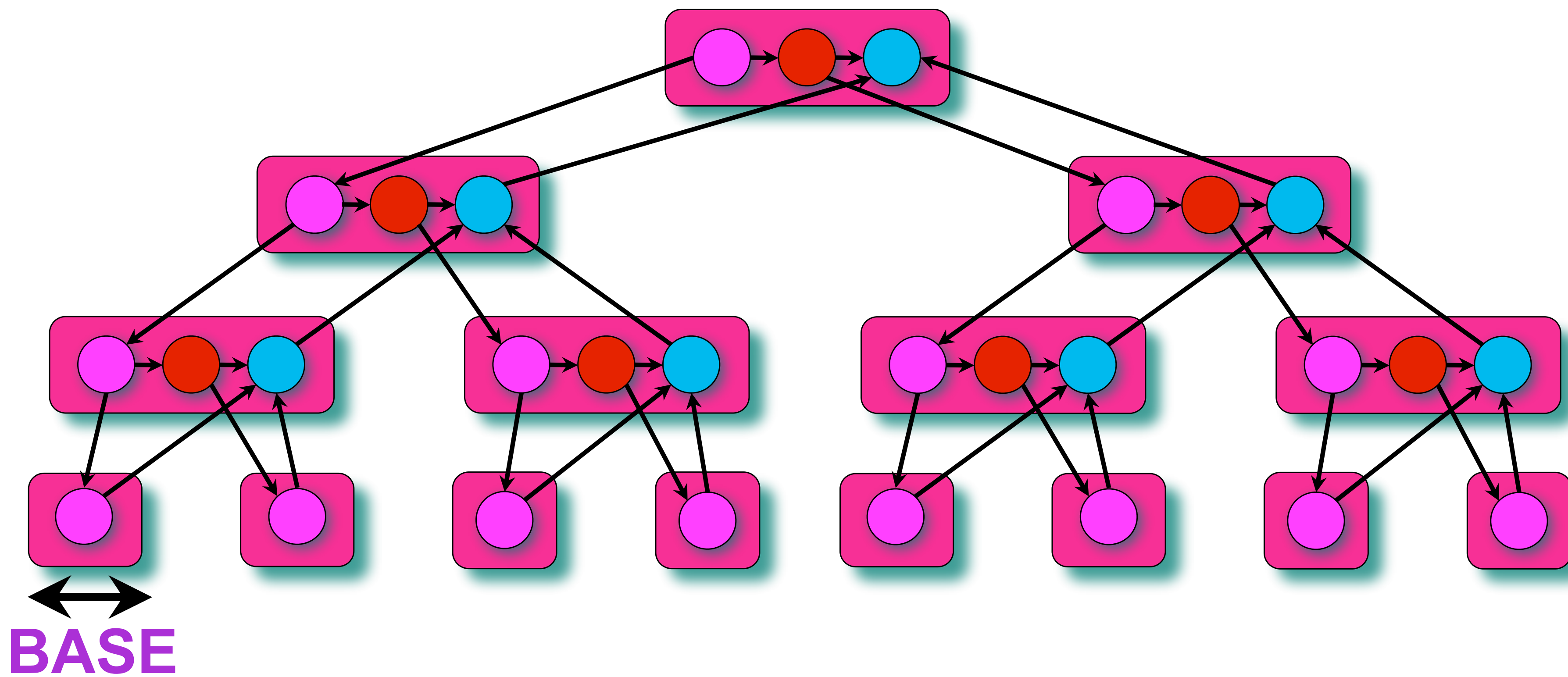
Vector Addition Analysis

To add two vectors of length n, where BASE = Θ(1):

Work: $T_1 = \Theta(n)$

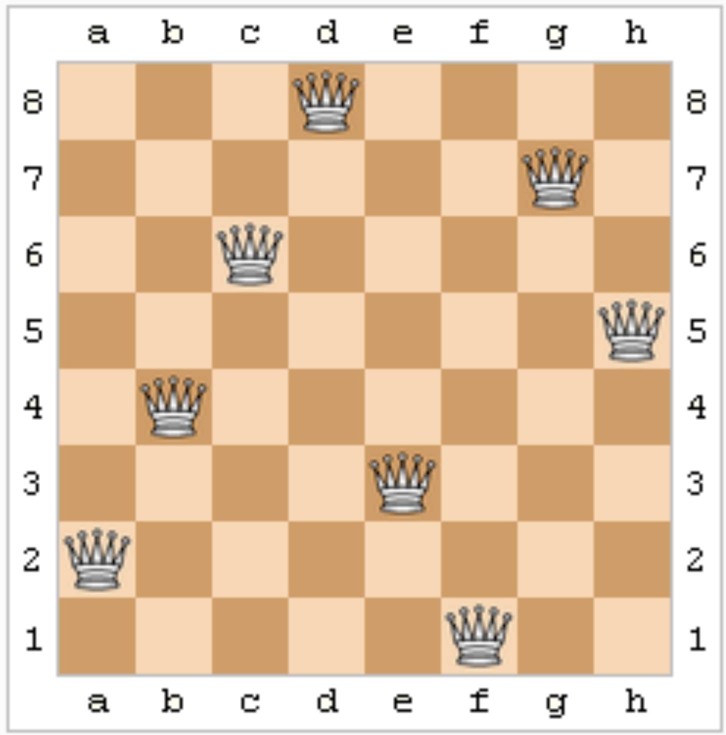
Span: $T_\infty = \Theta(\lg n)$

Parallelism: $T_1/T_\infty = \Theta(n/\lg n)$



Example: N Queens

- **Problem**
  - **—place N queens on an N x N chess board**
  - **—no 2 queens in same row, column, or diagonal**
- **Example: a solution to 8 queens problem**



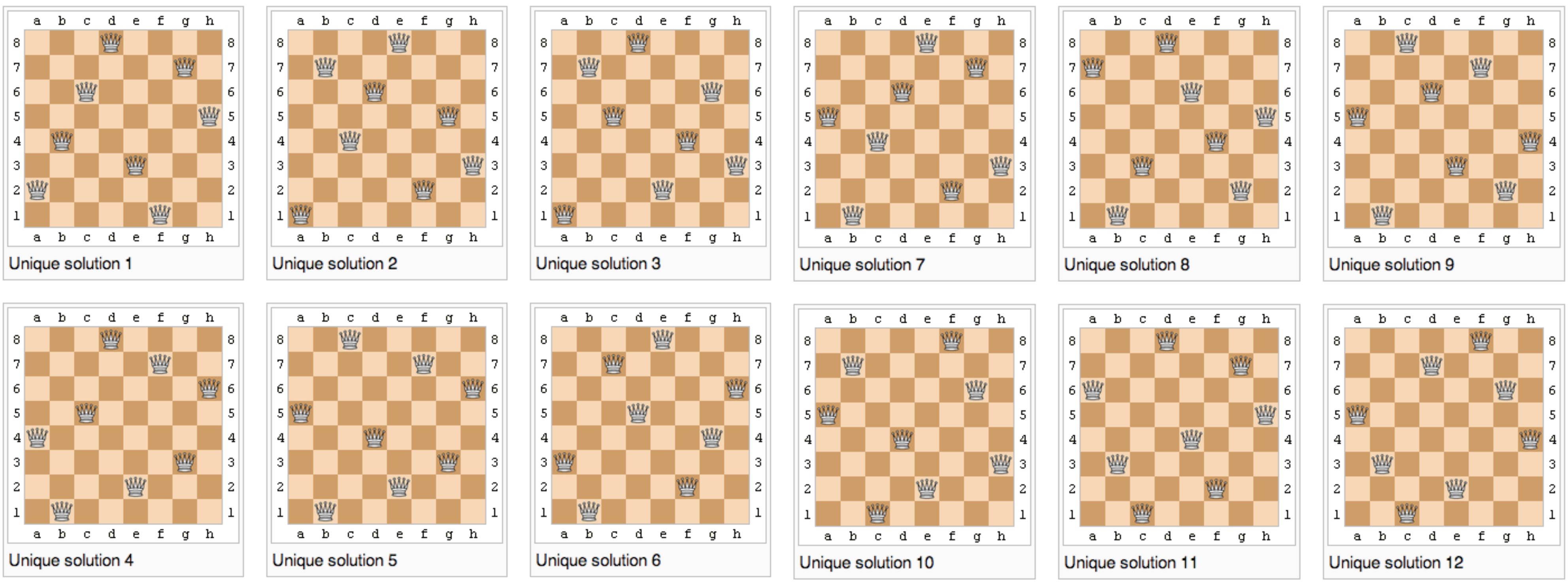
One possible solution

Image credit: http://en.wikipedia.org/wiki/Eight_queens_puzzle

N Queens: Many Solutions Possible

Example: 8 queens

- 92 distinct solutions
- 12 unique solutions; others are rotations & reflections



Unique solution 1, Unique solution 2, Unique solution 3, Unique solution 7, Unique solution 8, Unique solution 9

Unique solution 4, Unique solution 5, Unique solution 6, Unique solution 10, Unique solution 11, Unique solution 12

Image credit: http://en.wikipedia.org/wiki/Eight_queens_puzzle

N Queens Solution Sketch

Sequential Recursive Enumeration of All Solutions

```
int nqueens(n, j, placement) {
   // precondition: placed j queens so far
   if (j == n)  { print placement; return; }
  for (k = 0; k < n; k++)
    if putting j+1 queen in kth position in row j+1 is legal
        add queen j+1 to placement
        nqueens(n, j+1, placement)
        remove queen j+1 from placement
}
```

- Where's the potential for parallelism?
- What issues must we consider?

Parallel N Queens Solution Sketch

```
void nqueens(n, j, placement) {
   // precondition: placed j queens so far
   if (j == n) {  /* found a placement  */ process placement; return; }
   for (k = 1; k <= n; k++)
      if putting j+1 queen in kth position in row j+1 is legal
         copy placement into newplacement and add extra queen
         cilk_spawn nqueens(n,j+1,newplacement)
   cilk_sync
   discard placement
}
```

Issues regarding placements

- **how can we report placements?**
- **what if a single placement suffices?**
  - **no need to compute all legal placements**
  - **so far, no way to terminate children exploring alternate placement**

Approaches to Managing Placements

- Choices for reporting multiple legal placements
  - count them
  - print them on the fly
  - collect them on the fly; print them at the end
- If only one placement desired, can skip remaining search

References

- "Introduction to Parallel Computing" by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
- Charles E. Leiserson. Cilk LECTURE 1. Supercomputing Technologies Research Group. Computer Science and Artificial Intelligence Laboratory. http://bit.ly/mit-cilk-lec1
- Charles Leiserson, Bradley Kuzmaul, Michael Bender, and Hua-wen Jing. MIT 6.895 lecture notes - Theory of Parallel Systems. http://bit.ly/mit-6895-fall03
- Intel Cilk++ Programmer's Guide. Document # 322581-001US.