This Presentation Will Be Recorded

- By joining this Zoom web tutorial session, you automatically consent to the recording of all video, audio, and chat-room content.
- Furthermore, you grant permission to the OpenCilk organization to share the recordings, in full or in part, internally and with third parties.
- Please join without video and stay muted if you do not wish to be recorded.

Slides downloadable at
http://opencilk.org/pact21/opencilk-pact-2021.pdf

How to Parallelize Your Own Language Using OpenCilk Components

I-Ting Angelina Lee
Washington University in St. Louis

Tao B. Schardl
MIT CSAIL

And many helpers

www.opencilk.org

contact@opencilk.org

International Conference on Parallel Architectures and Compilation Techniques
September 27, 2021

Getting Started

- Join the Slack channel: https://tinyurl.com/OpenCilkSlack, channel `#pact2021`.
- You will need Docker set up to do the hands-on exercises.
- Download the Docker image: https://tinyurl.com/OpenCilkDocker
- We provide a script, `docker.sh`, to help you use the Docker image: https://tinyurl.com/OpenCilkDockerSh

Using the Docker Image

- To setup the Docker image initially:

```
$ ./docker.sh init
```

- To run code in the Docker container:

```
$ ./docker.sh run
```

- In the Docker container, verify the version of `clang`:

```
$ clang --version
clang version 12.0.0
```

What Is OpenCilk?

- OpenCilk is an open-source implementation of the Cilk concurrency platform.
- Cilk extends C/C++ with a small set of linguistic control constructs to support fork-join parallelism.

- Cilk focuses on:
  - Shared-memory multiprocessing
  - Client-side multiprogrammed environments
  - Regular and irregular parallel computations
  - Predictable and composable performance

Features of Cilk

- A **processor-oblivious** programming model with simple, effective, and composable language constructs for expressing parallelism.
- A provably and practically efficient **work-stealing** scheduler.
- A suite of **productivity tools**:
  - Cilksan: Determinacy race detector
  - Cilkscale: Scalability analyzer

OpenCilk System Architecture

- Compatibility — Provide backward compatibility with Cilk Plus minus vector ops (i.e., Cilk++).
- Open source — Distribute under liberal open-source licenses.
- Componentization — Divide system into distinct software components with well-defined interfaces.
- Integration — As individual components are enhanced, ensure that they continue to interoperate with the entire platform.
- Reliability — Provide a suite of extensive tests and benchmarks to ensure that releases are stable, perform well, and are free of serious bugs.

BASICS OF RECURSIVE FORK–JOIN PARALLEL PROGRAMMING

Nested Parallelism in Cilk

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t 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 (*processor oblivious*).

Loop Parallelism in Cilk

Example:
In-place matrix transpose

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad A^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}$$

The iterations of a `cilk_for` loop may execute in parallel.

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

Cilk keywords grant permission for parallel execution. They do not command parallel execution (*processor oblivious*).

Serial Projection

Cilk source

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

serial projection

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = fib(n-1);
    y = fib(n-2);

    return (x + y);
  }
}
```

The *serial projection* of a Cilk program is always a legal interpretation of the program's semantics.

Moreover, a Cilk program executing on one core behaves exactly the same as the execution of its serialization.

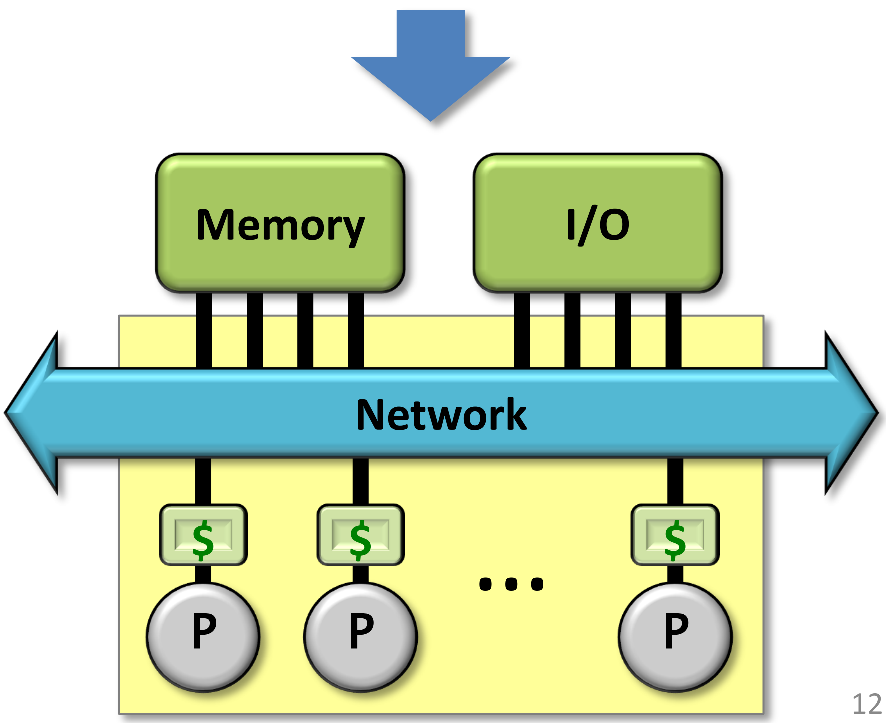
To obtain the serial projection:

```
#define cilk_spawn
#define cilk_sync
#define cilk_for for
```

Scheduling in Cilk

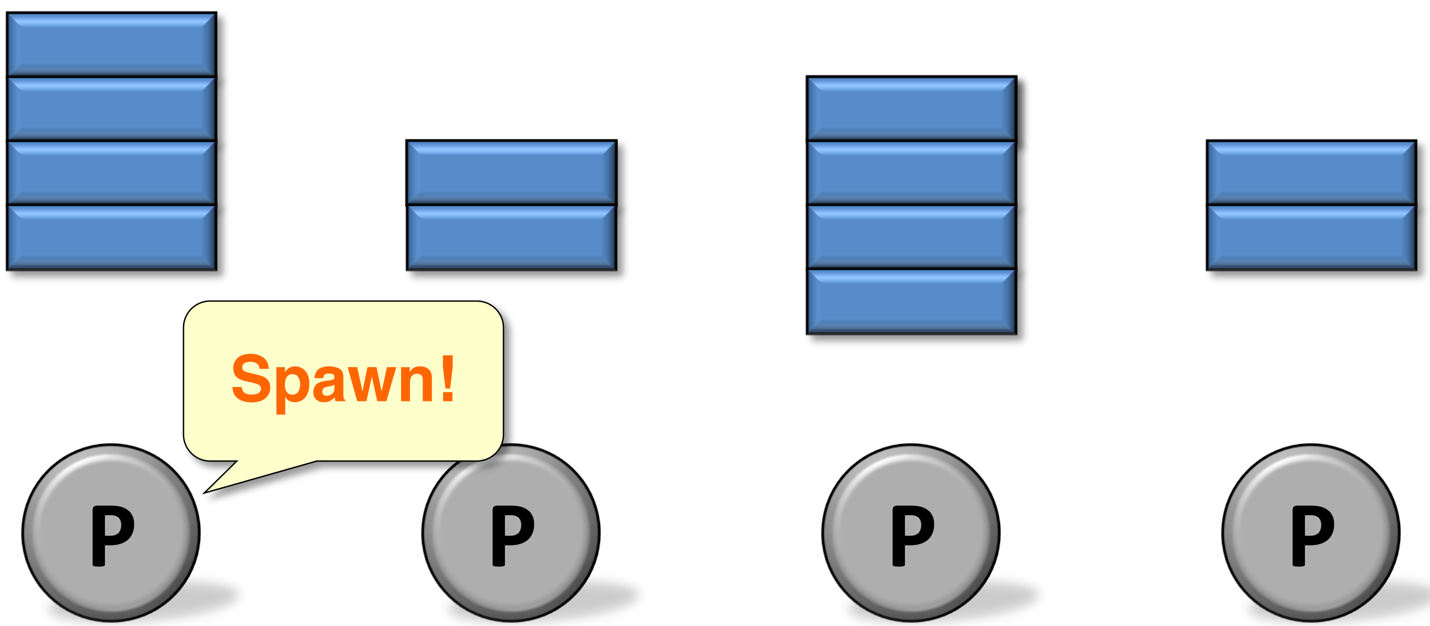
- Cilk allows the programmer to express logical parallelism in an application.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.
- Cilk's *work-stealing scheduler* is provably efficient.

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```



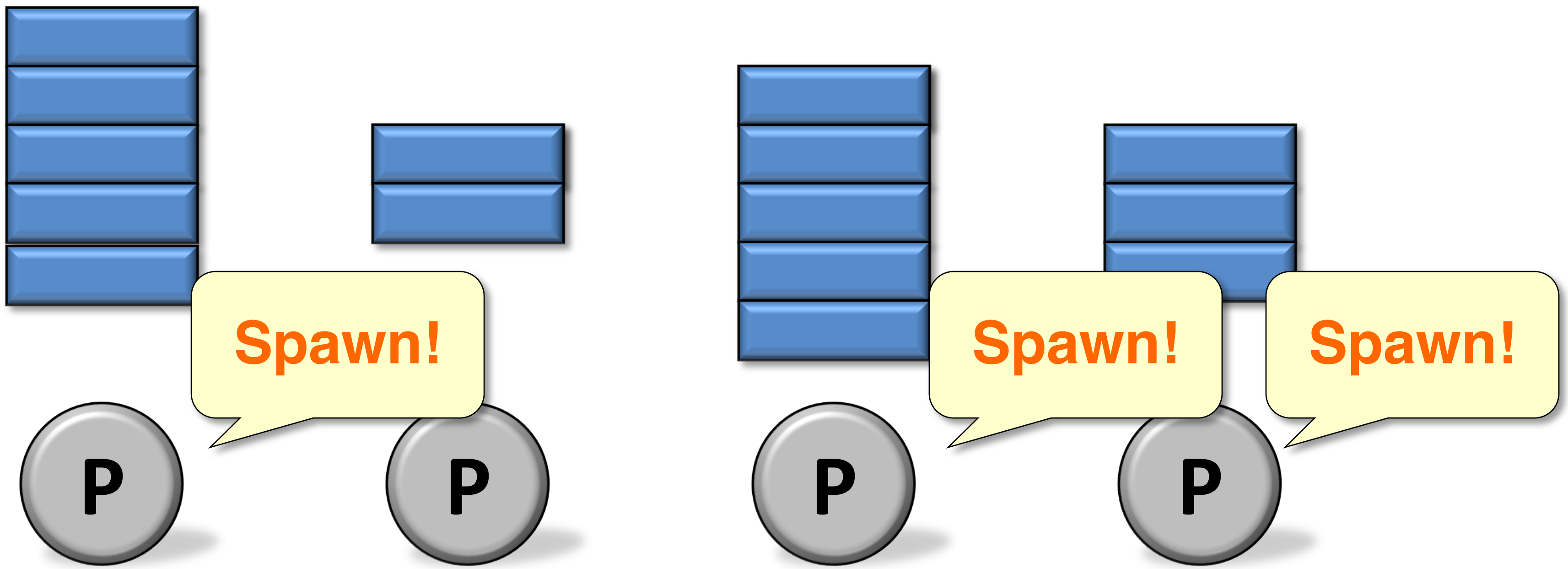
Cilk's Work-Stealing Scheduler

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



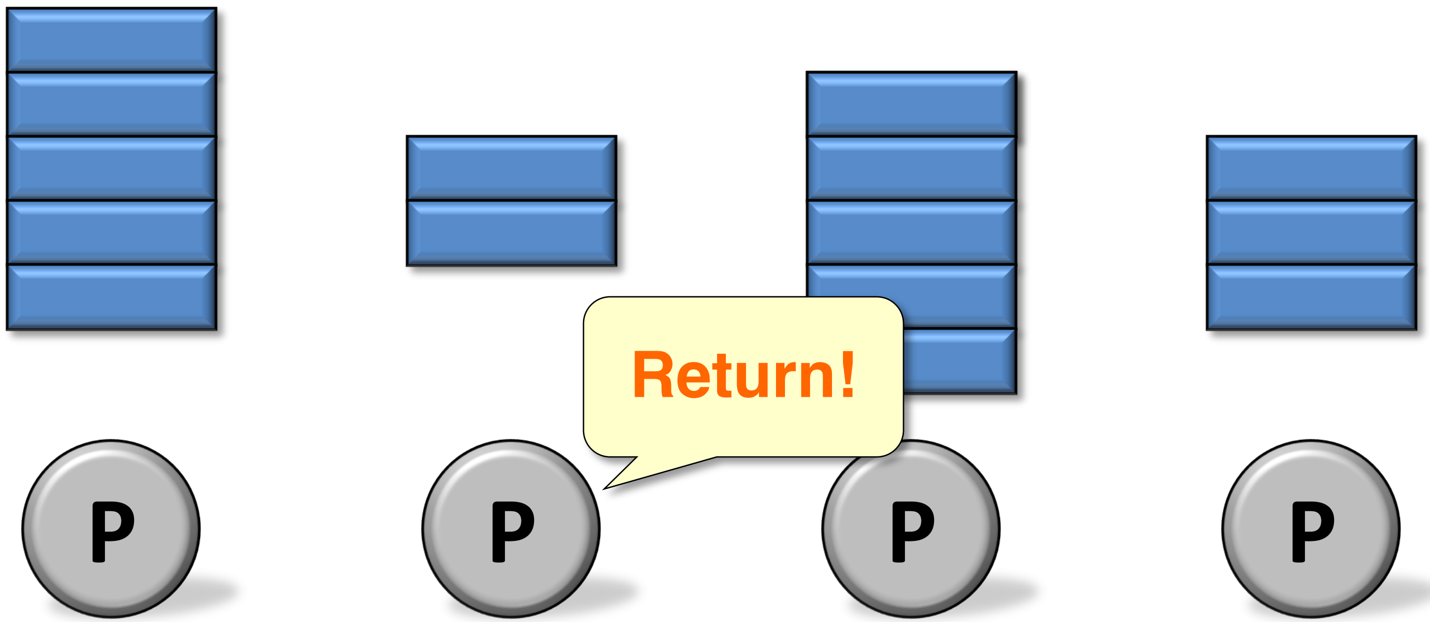
Cilk's Work–Stealing Scheduler

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



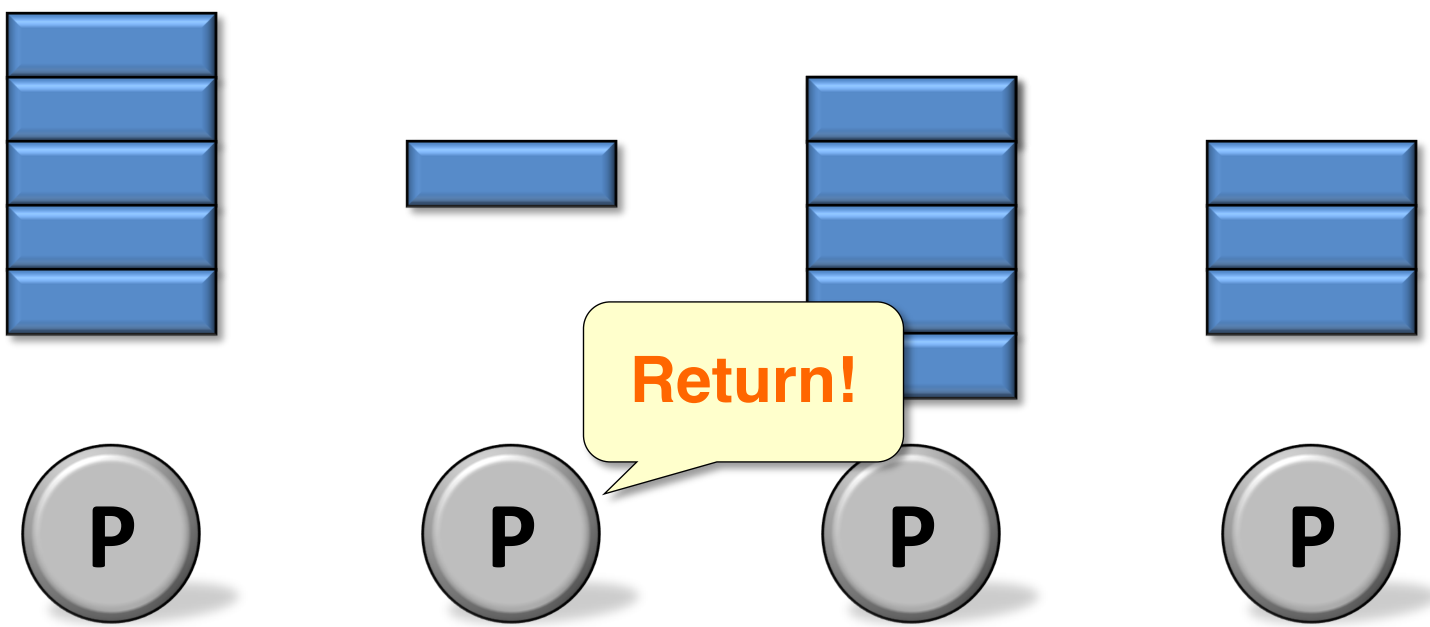
Cilk's Work-Stealing Scheduler

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



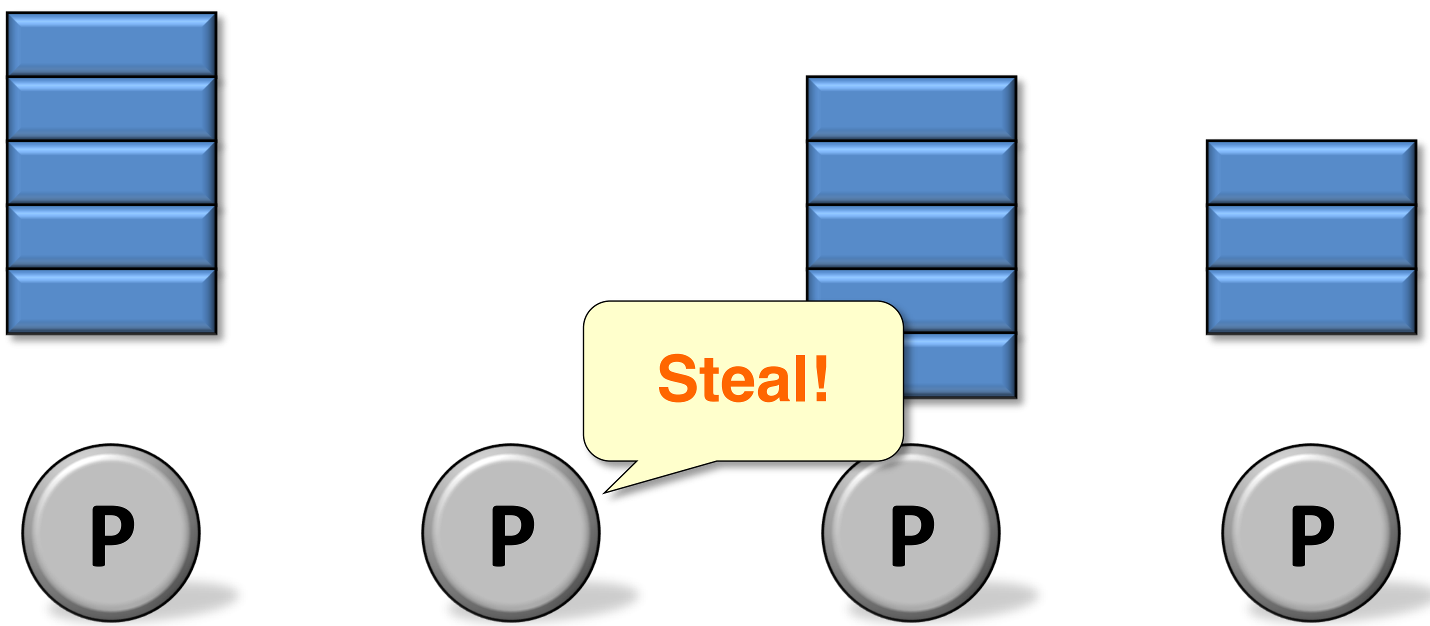
Cilk's Work-Stealing Scheduler

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



Cilk's Work-Stealing Scheduler

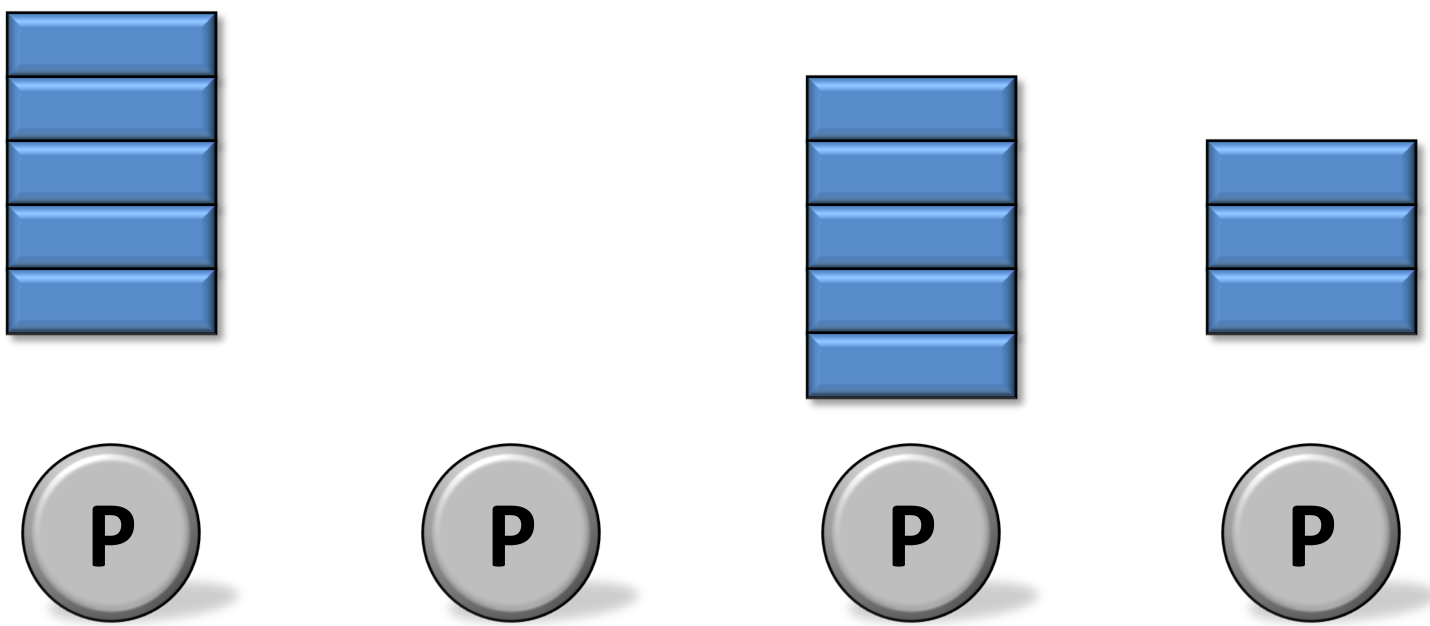
Each worker (processor) maintains a deque of ready work, and it manipulates the bottom of the deque like a stack.



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

Cilk's Work-Stealing Scheduler

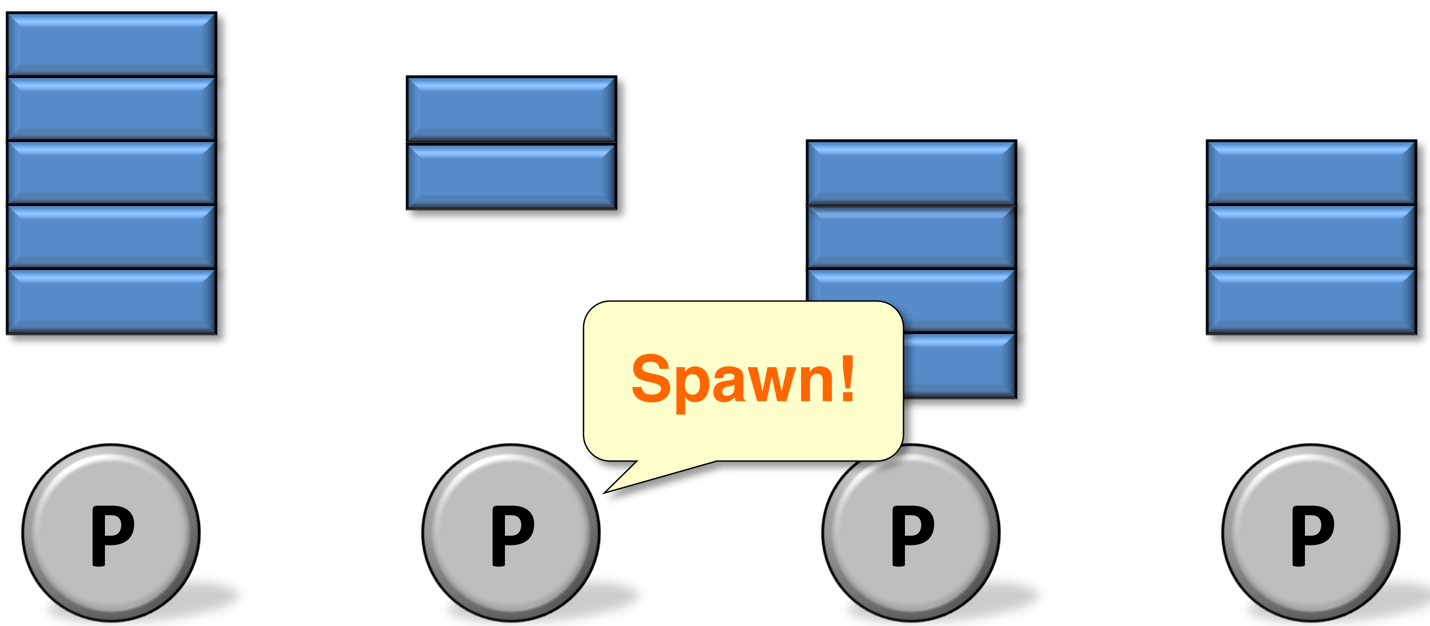
Each worker (processor) maintains a deque of ready work, and it manipulates the bottom of the deque like a stack.



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

Cilk's Work-Stealing Scheduler

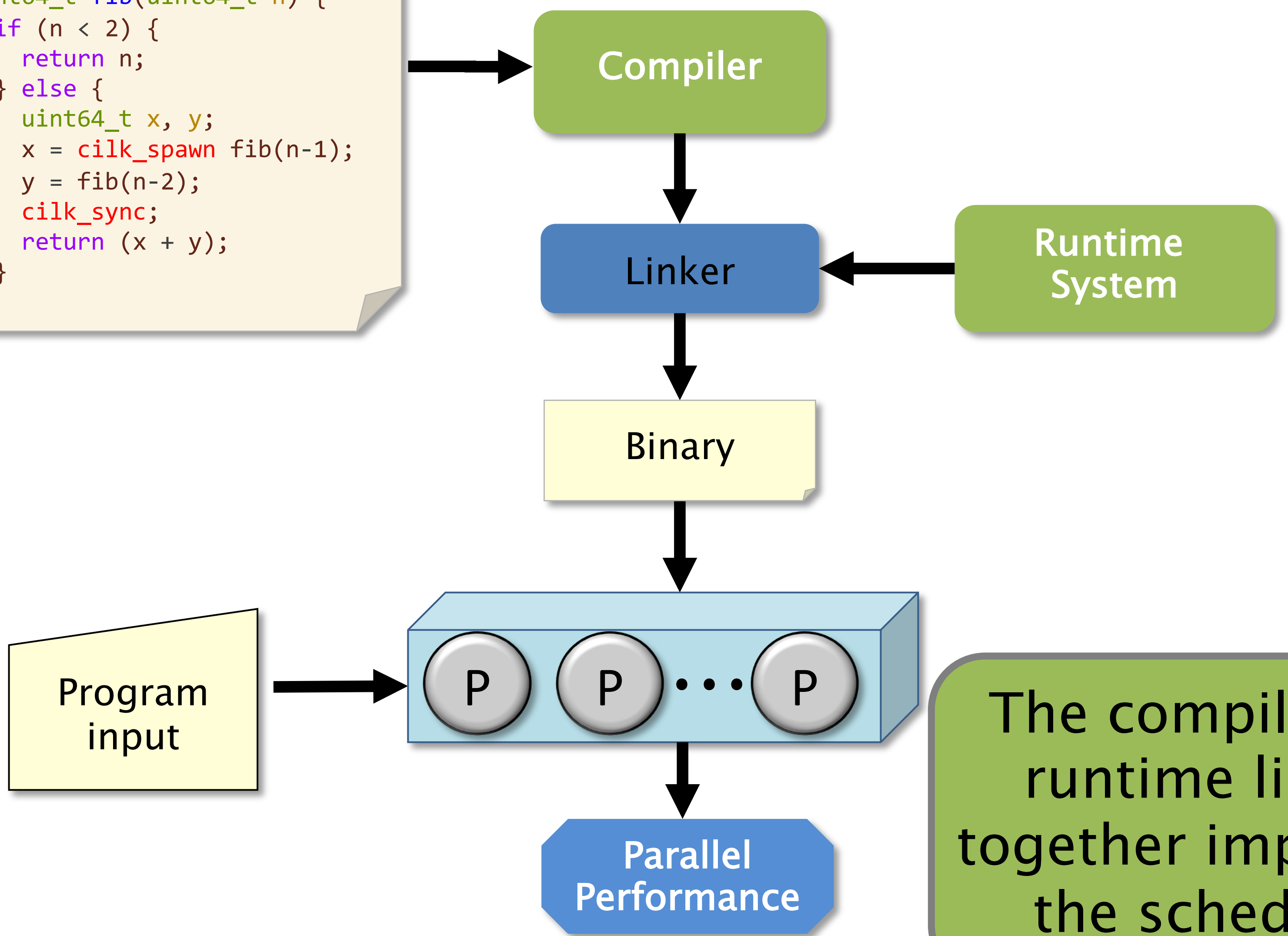
Each worker (processor) maintains a deque of ready work, and it manipulates the bottom of the deque like a stack.



Resume execution upon a successful steal.

OpenCilk Platform

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```



The compiler and runtime library together implement the scheduler.

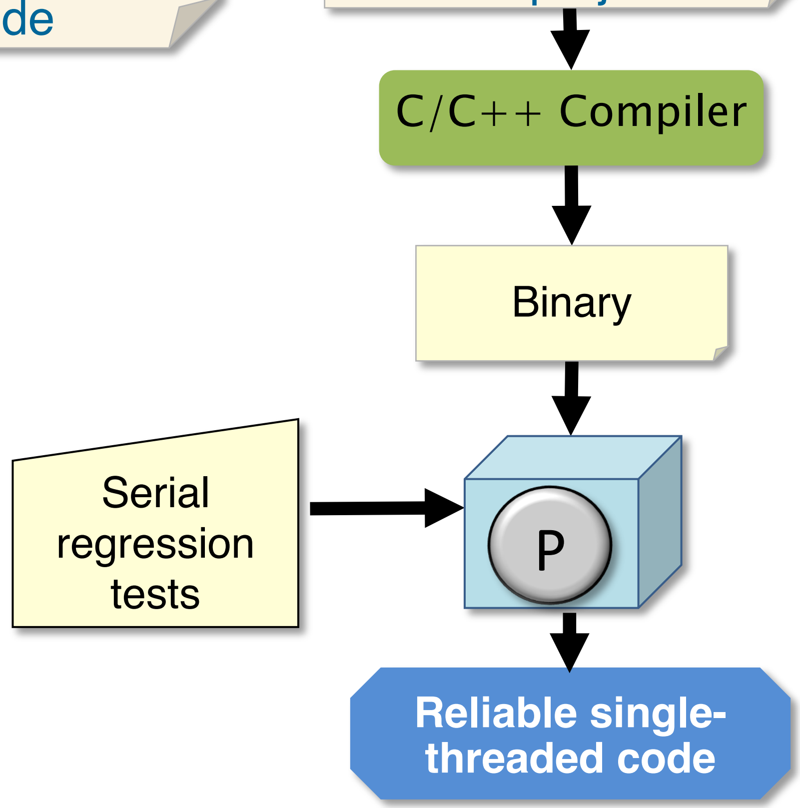
Dev Flow: Serial Testing First

```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

Cilk code

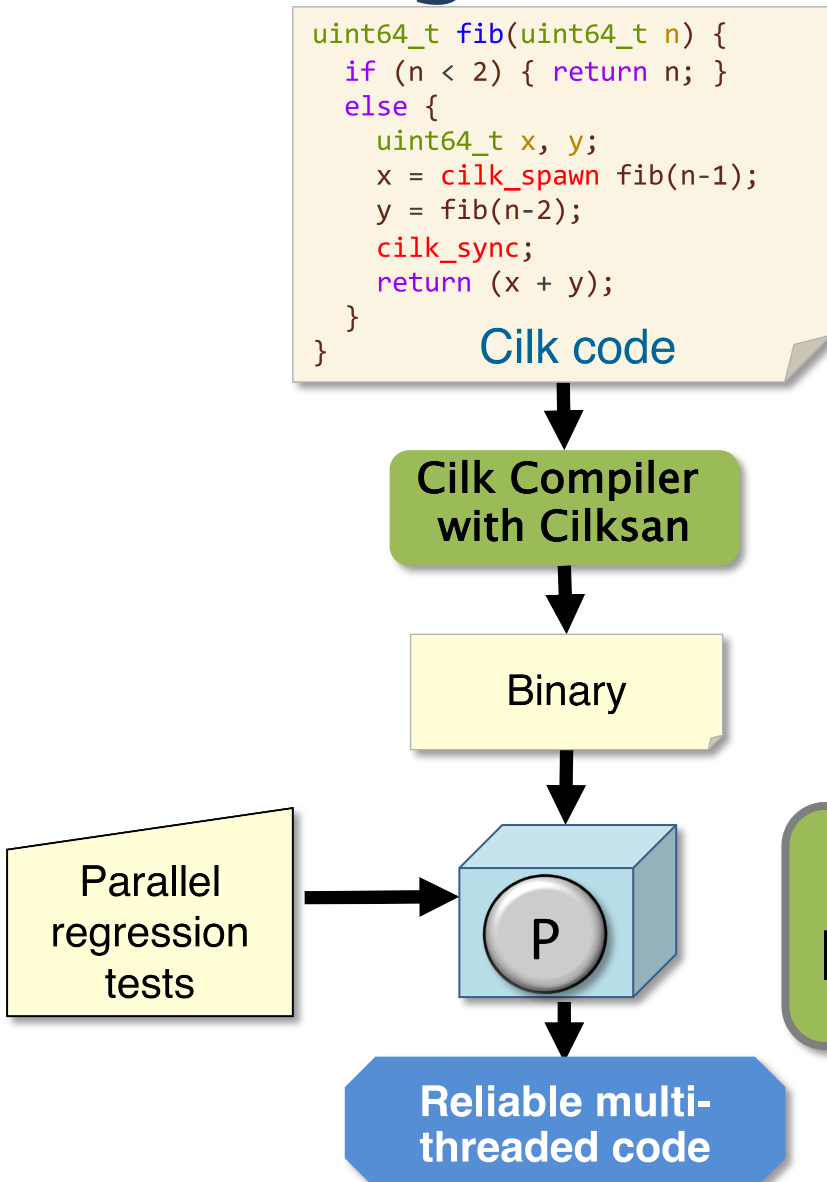
```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x + y);
  }
}
```

serial projection



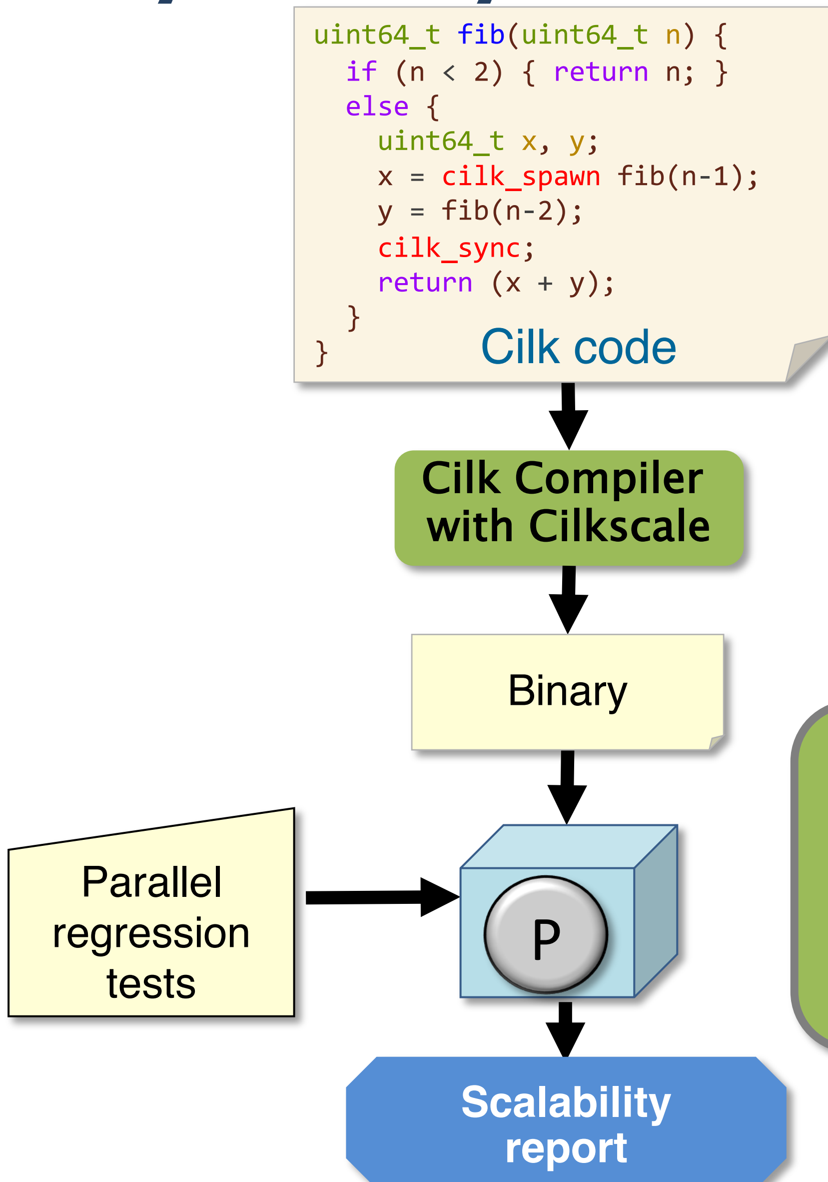
Cilk's serial projection enables simple serial testing.

Parallel Testing



Cilksan finds and localizes race bugs.

Scalability Analysis



Cilkscale analyzes how well your program will scale to larger machines.

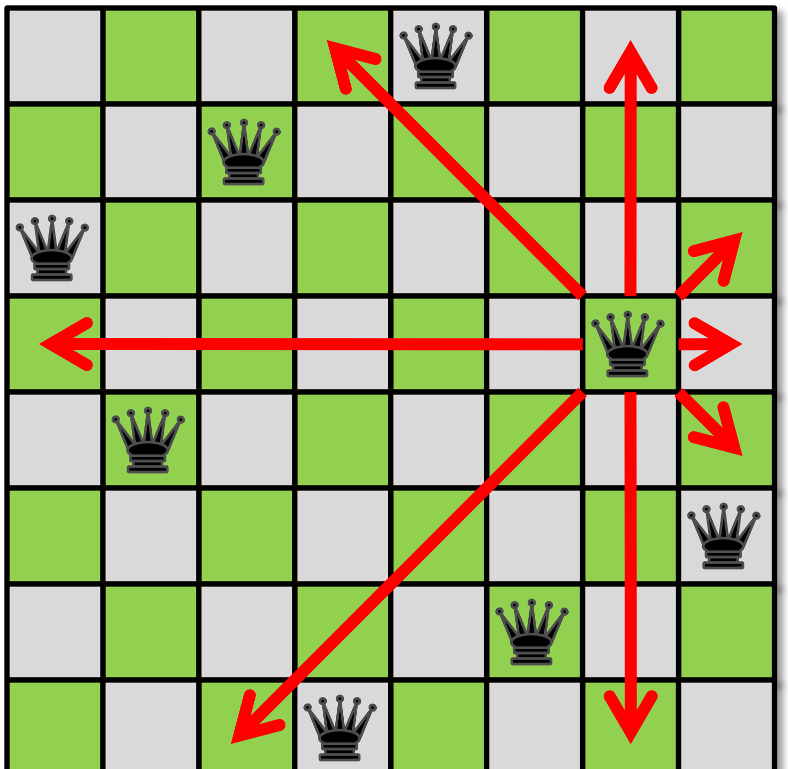
Hands-On with Cilk Programming

- Take a look at `nqueens.c`.
- How do you parallelize this code?

The N–Queen Problem

Problem

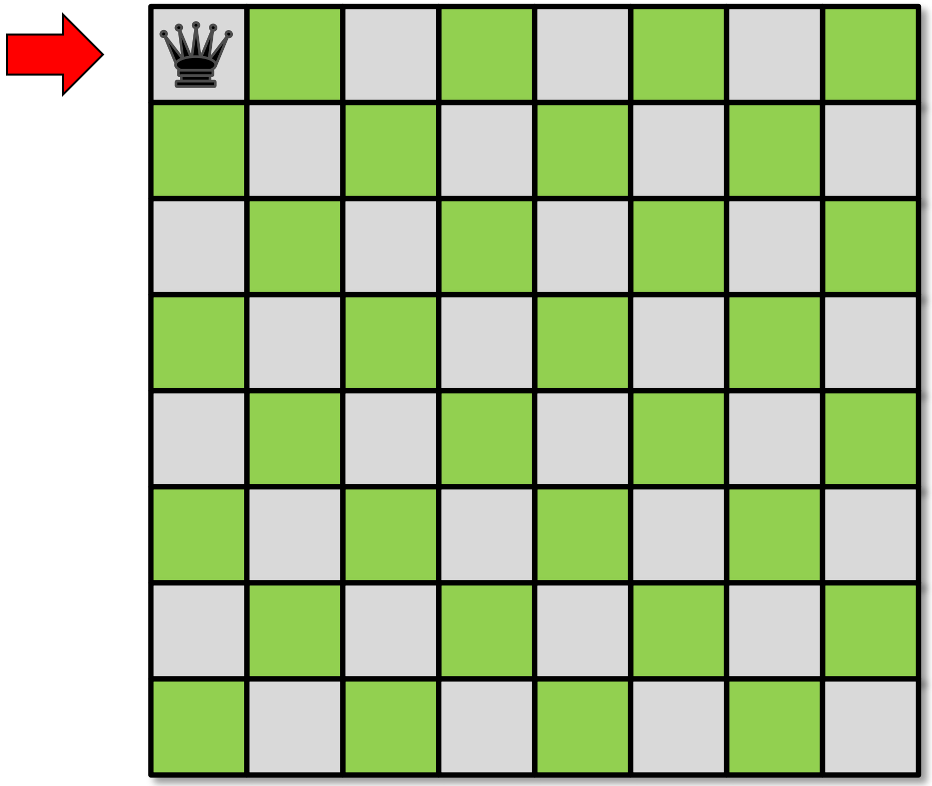
Place n queens on an $n \times n$ chessboard so that no queen attacks another, i.e., no two queens in any row, column, or diagonal. Count the number of possible solutions.



Backtracking Search

Strategy

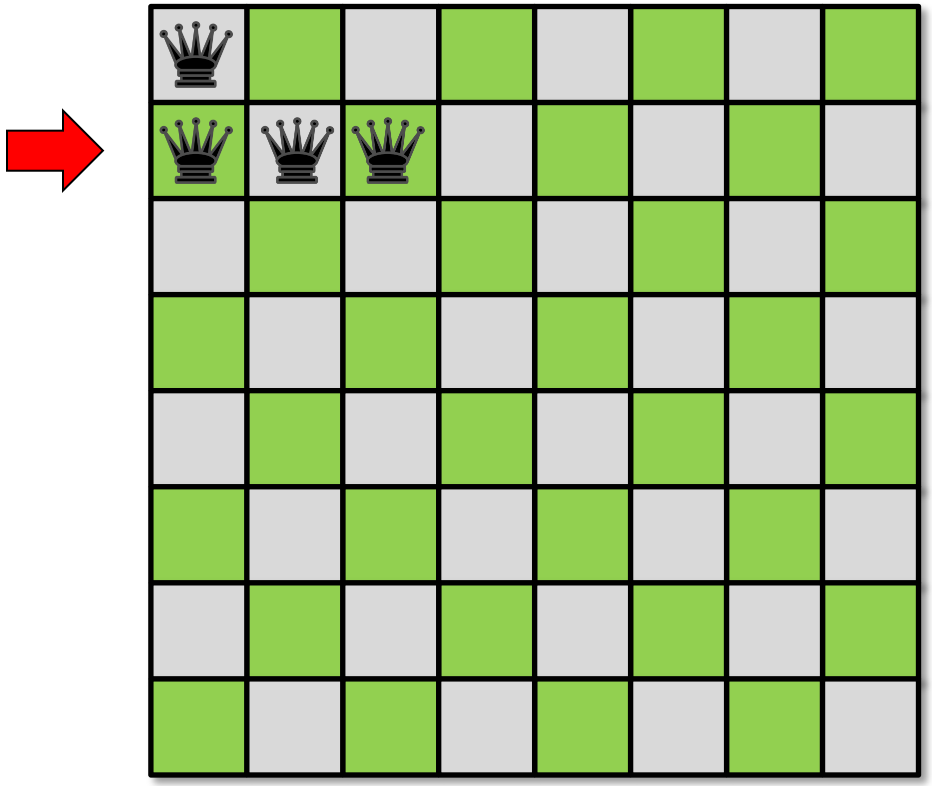
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

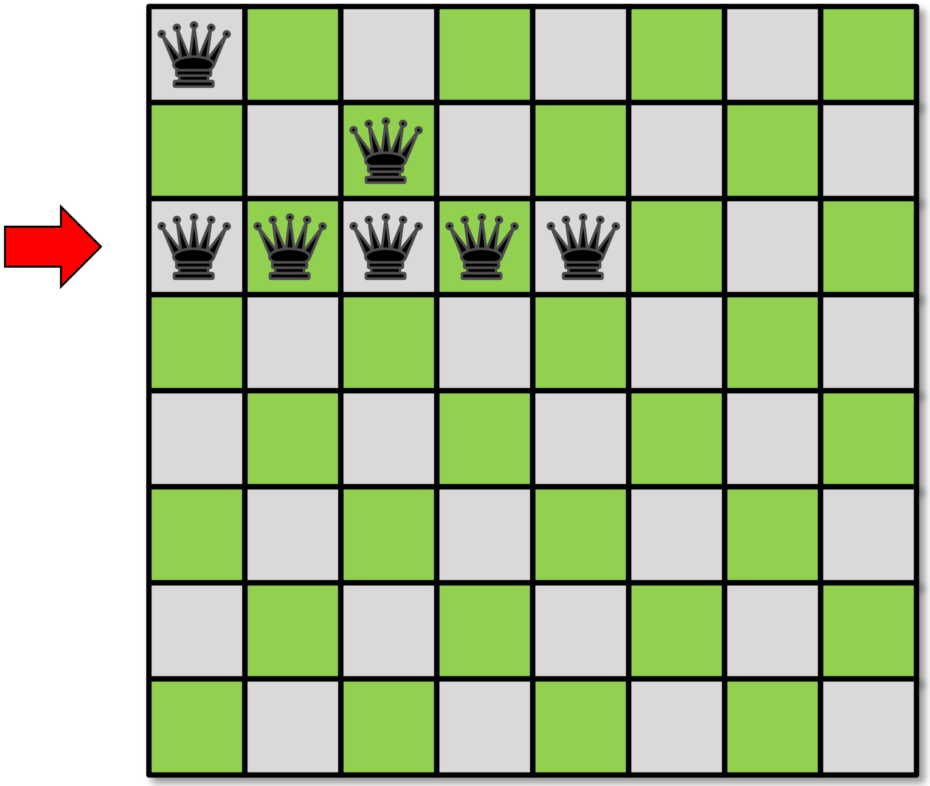
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

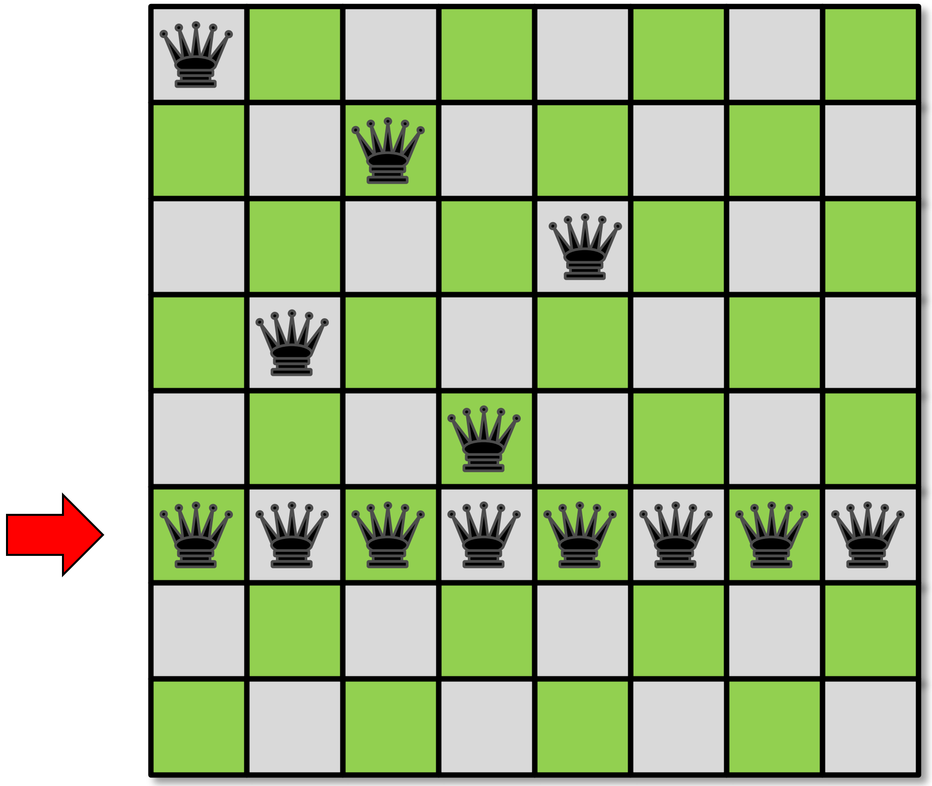
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

Try placing queens row by row. If you can't place a queen in a row, backtrack.

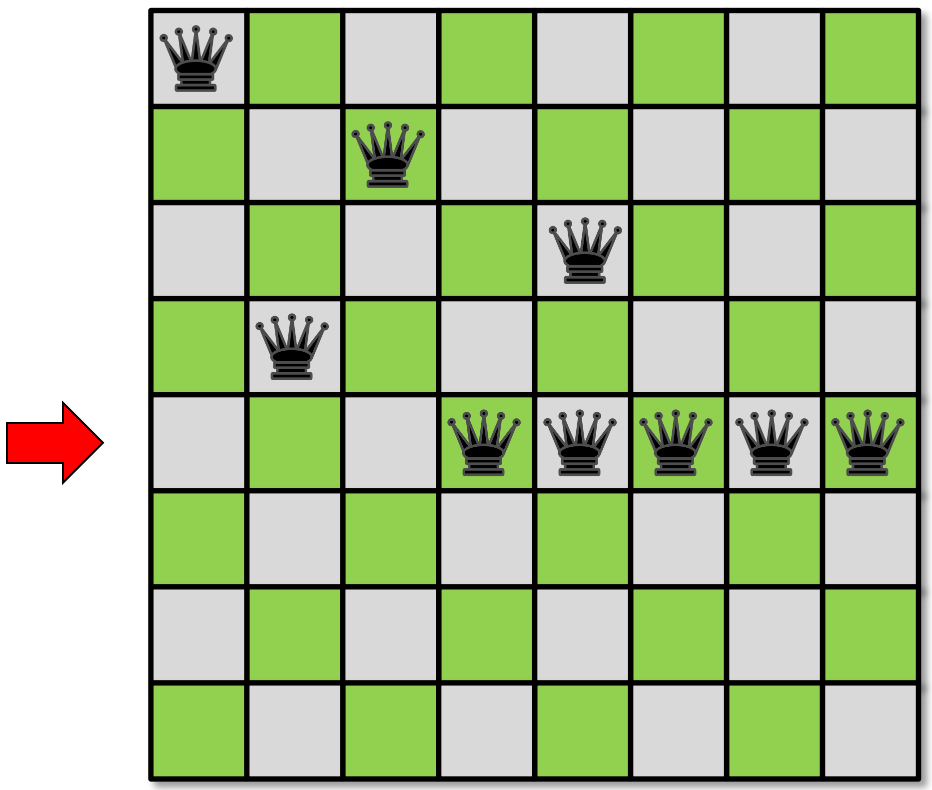


Backtrack!

Backtracking Search

Strategy

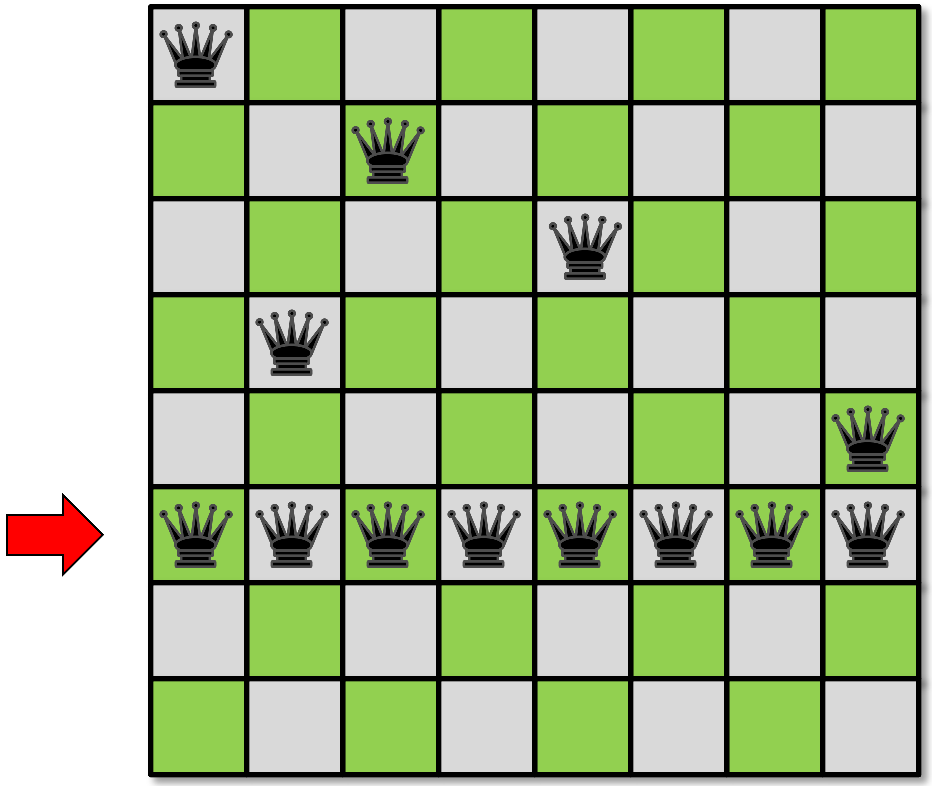
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtracking Search

Strategy

Try placing queens row by row. If you can't place a queen in a row, backtrack.

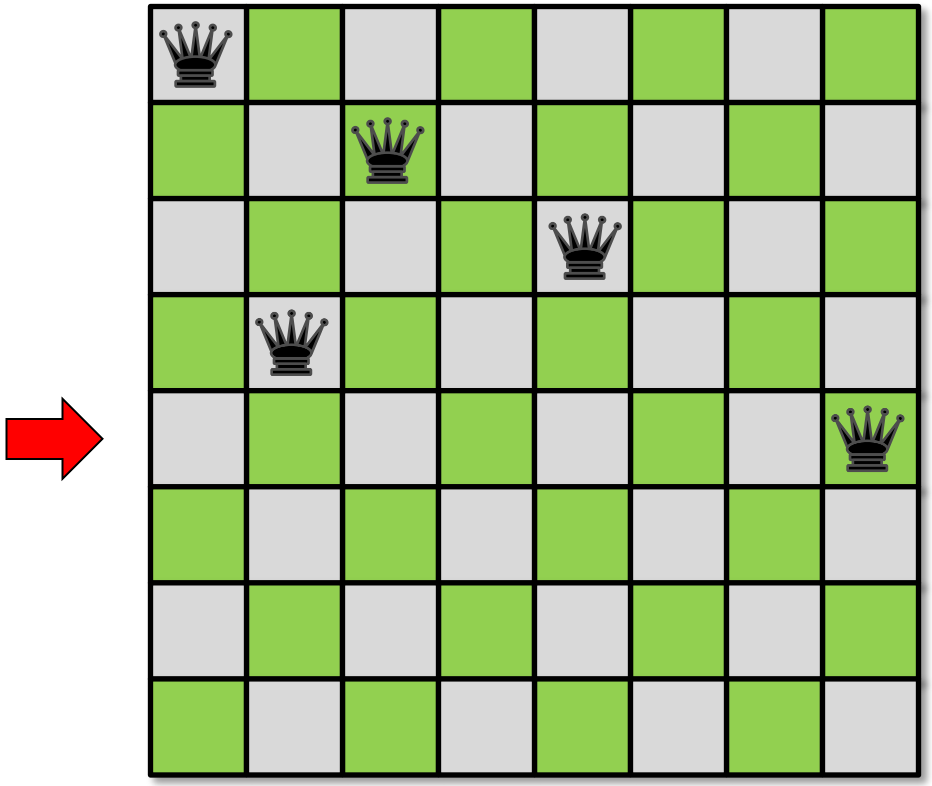


Backtrack!

Backtracking Search

Strategy

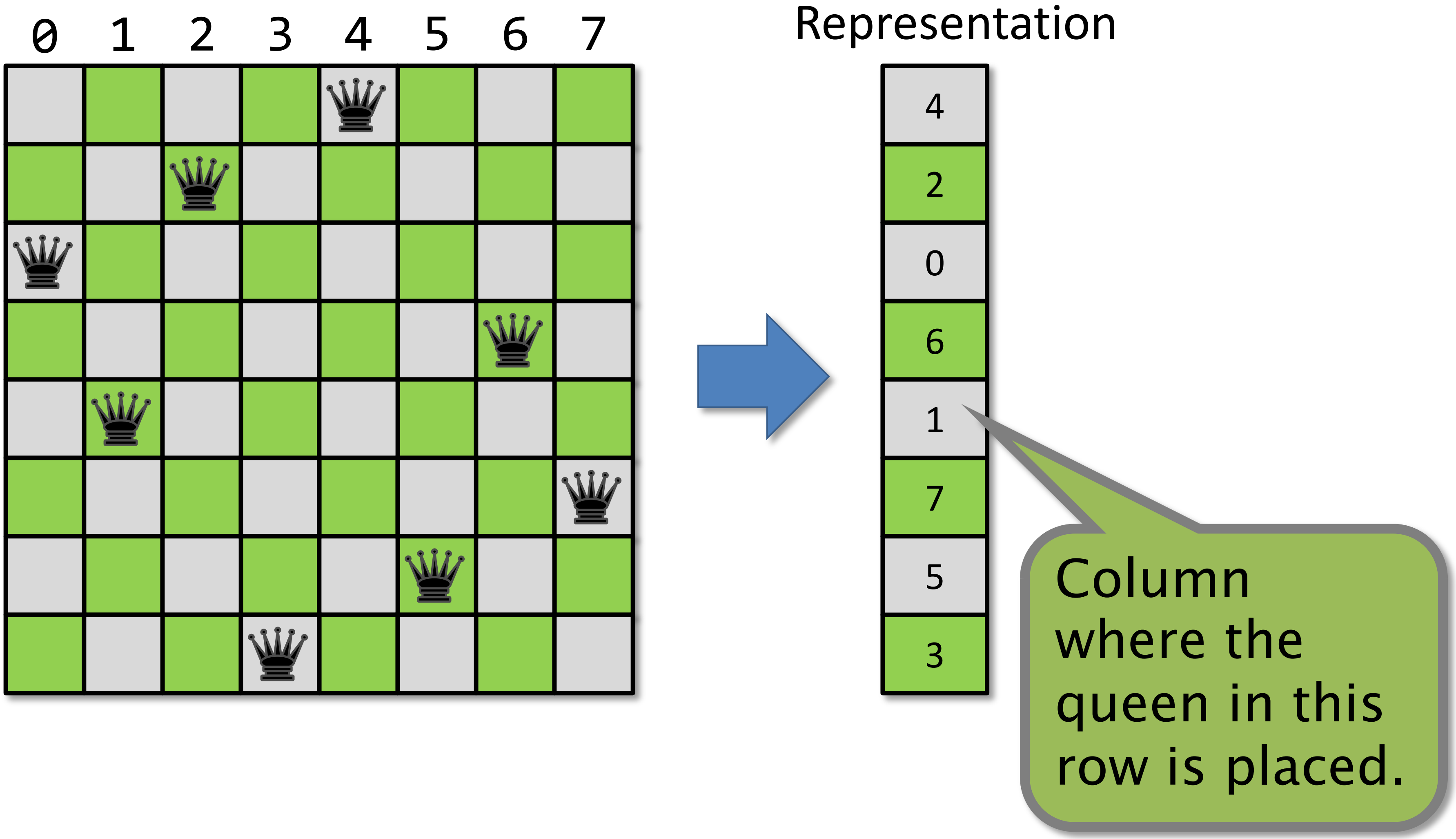
Try placing queens row by row. If you can't place a queen in a row, backtrack.



Backtrack!

Board Representation

The board can be represented as an array of integers.



Hands-On with Cilk Programming

- Take a look at `nqueens.c`.
- How do you parallelize this code?

- In the Docker container, compile and run the code once parallelized:

```
$ cd /tutorial
$ make nqueens
$ ./nqueens 13
```

Racy NQueens Code (racy-nqueens.c)

```
int nqueens(int n, int row, char *board) {
  int *count;
  char *new_board;
  int solNum = 0;
  if (n == row) { return 1; } // end of the board; found a solution

  count = (int *) alloca(n * sizeof(int));
  (void) memset(count, 0, n * sizeof (int));

  new_board = (char *) alloca((row + 1) * sizeof (char));
  memcpy(new_board, board, row * sizeof (char));

  for (int col = 0; col < n; col++) {
    new_board[row] = col;
    if (no_conflict(row + 1, new_board))
      count[col] = cilk_spawn nqueens(n, row + 1, new_board);
  }
  cilk_sync;

  for (int i = 0; i < n; i++) { solNum += count[i]; }

  return solNum;
}
```

Where’s the race?

Debugging Race Conditions

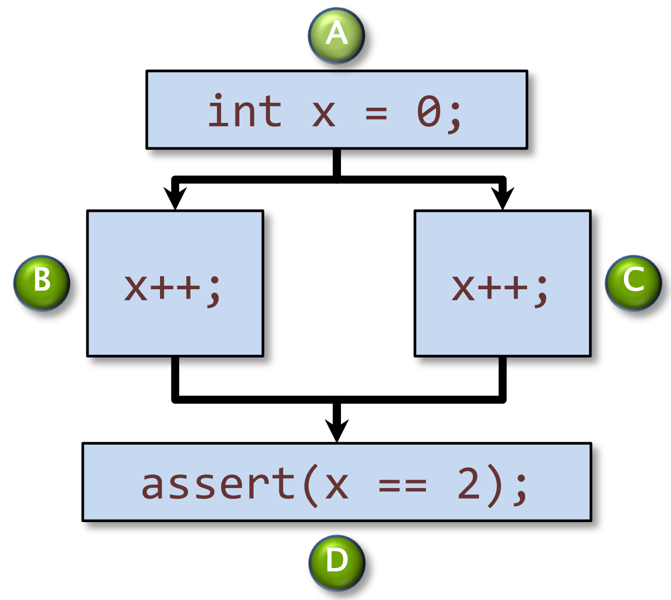
Determinacy Races

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.

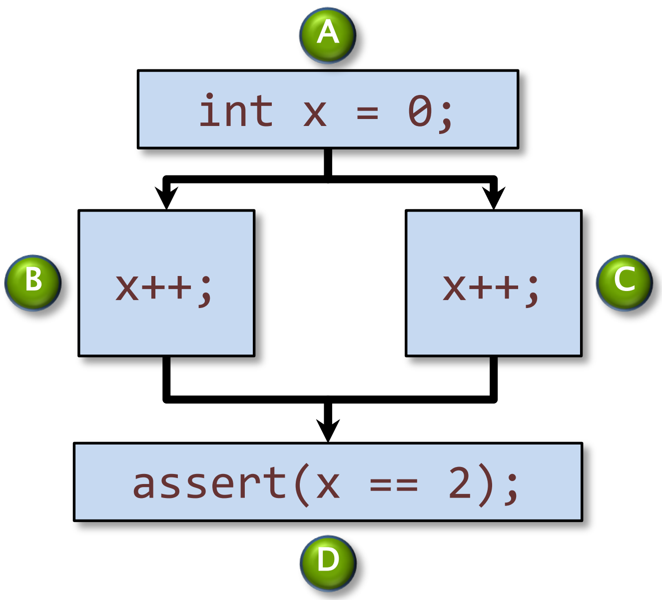
Example

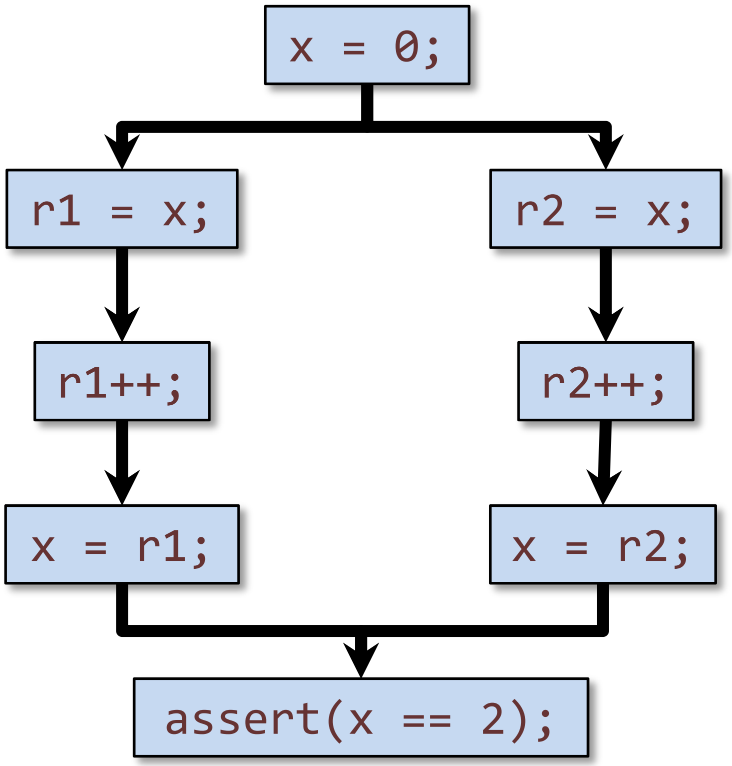
```
int x = 0;
cilk_for (int i=0, i<2, ++i) {
  x++;
}
assert(x == 2);
```

Trace



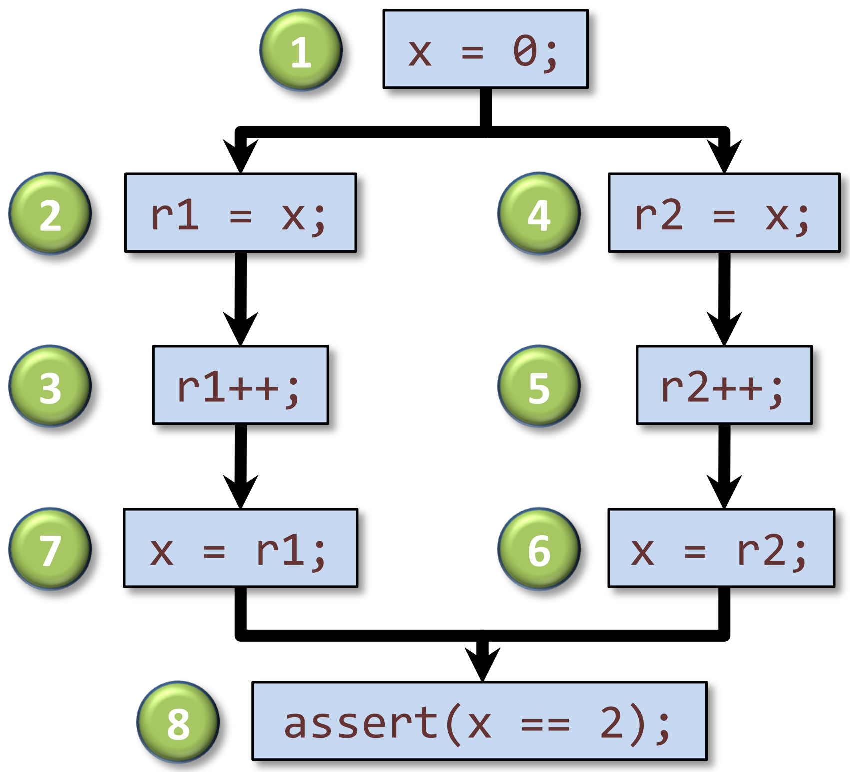
A Closer Look

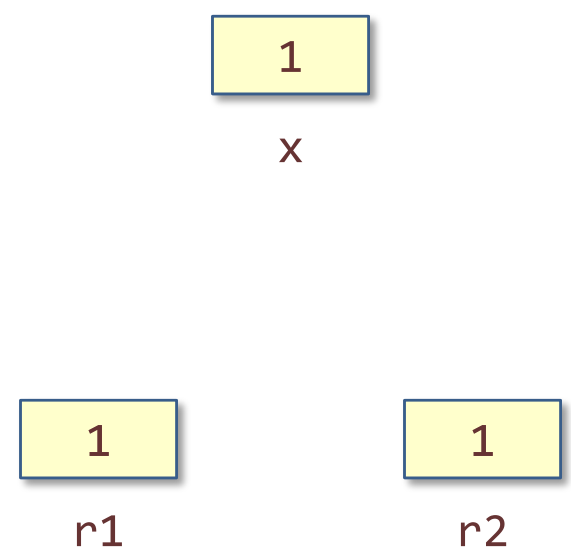




Race Bugs

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.

In Contrast, Data Races

DEFINITION: A *data race* occurs when two logically parallel instructions *holding no locks in common* access the same memory location and at least one of the instructions performs a write.

Although data-race-free programs obey atomicity constraints, they can still be nondeterministic, because acquiring a lock can cause a determinacy race with another lock acquisition.

WARNING: Codes that use locks are nondeterministic by intention.

Determinism

Cilk supports writing *deterministic* parallel programs, in which every memory location is updated with the same sequence of values in every execution.

- The program always behaves the same way on a given input, regardless of scheduling.
- For many interesting and practical programs, there is no need to use locks or other concurrency mechanisms.

Advantage: DEBUGGING!

Cilksan Race Detector

- The Cilksan-instrumented program is produced by compiling with the `-fsanitize=cilk` command-line compiler switch.
- If an ostensibly deterministic Cilk program run on a given input could possibly behave any differently than its serial projection, Cilksan guarantees to report and localize the offending race.
- Cilksan employs a regression-test methodology, where the programmer provides test inputs.
- Cilksan identifies filenames, lines, and variables involved in races, including stack traces.
- Ensure that all program files are instru-mented, or you'll miss some bugs.
- Cilksan is your best friend.

Hands-On with Cilksan (~15 min)

- In the Docker container, compile `racy-nqueens.c` with Cilksan enabled:

```
$ cd /tutorial
$ make -B racy-nqueens CILKSAN=1
$ ./racy-nqueens 9
```

- You should see a race report. Where is the race?
- How do you fix the race?

Hands-On with Cilksan

```
Race detected at address 7f460b325874
*     Read 43ef18 nqueens ./racy-nqueens.c:73:3
|        `-to variable board (declared at ./racy-nqueens.c:58)
+      Call 43f73b nqueens ./racy-nqueens.c:78:29
+     Spawn 43efd7 nqueens ./racy-nqueens.c:78:29
|*    Write 43efa9 nqueens ./racy-nqueens.c:76:18
||       `-to variable new_board (declared at ./racy-nqueens.c:60)
\| Common calling context
 +     Call 43f73b nqueens ./racy-nqueens.c:78:29
 +    Spawn 43efd7 nqueens ./racy-nqueens.c:78:29
[...]
 +     Cal
   Alloca
    Stack
     Allo
      Cal
     Spaw
[...]
      Cal
```

racy-nqueens.c

```
[...]
72 new_board = (char *) alloca((row + 1) * sizeof (char));
73 memcpy(new_board, board, row * sizeof (char));
74
75 for (int col = 0; col < n; col++) {
76   new_board[row] = col;
77   if (no_conflict(row + 1, new_board))
78     count[col] = cilk_spawn nqueens(n,row+1,new_board);
79 }
[...]
```

What is Parallelism?

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;
  }
}
```

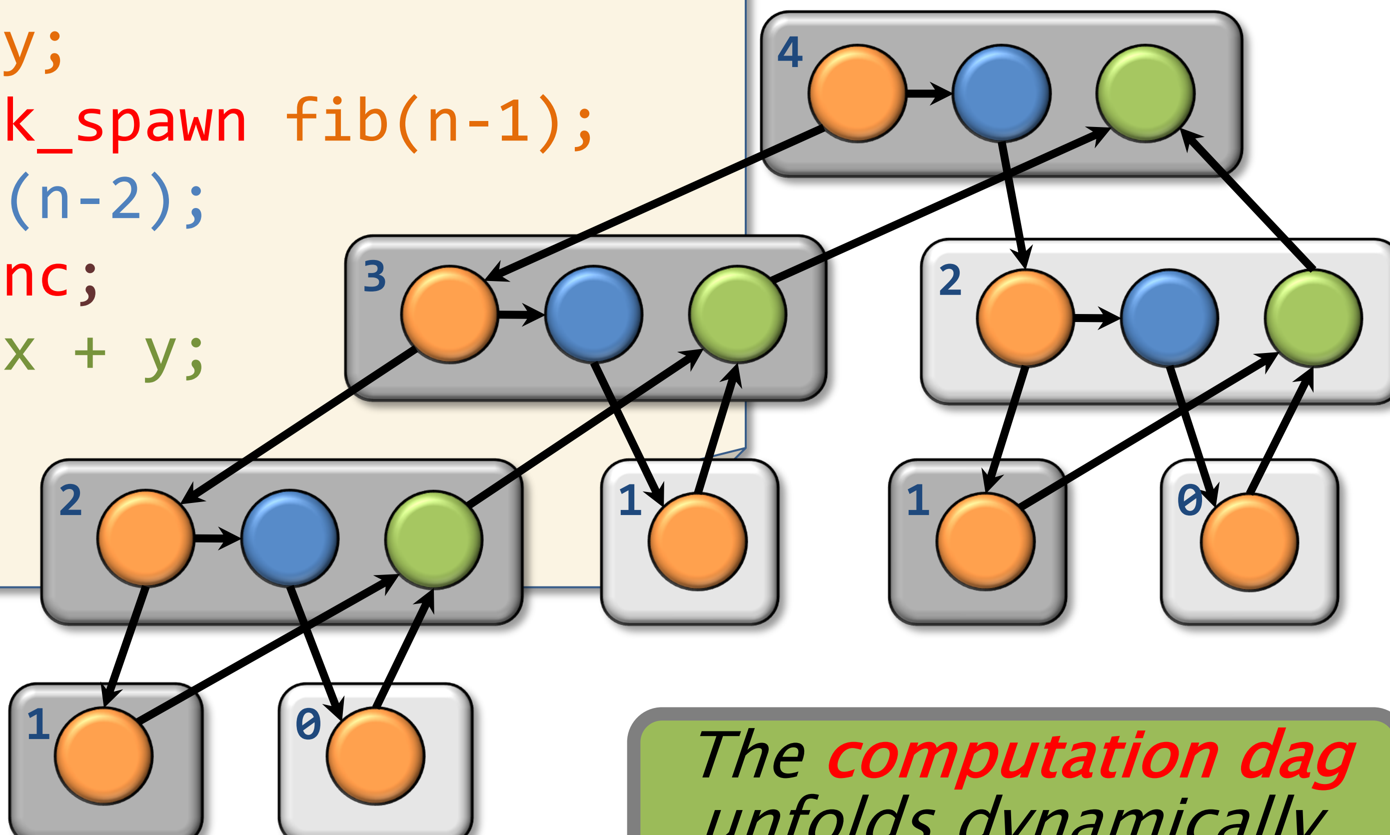
Example:

`fib(4)`

Execution Model

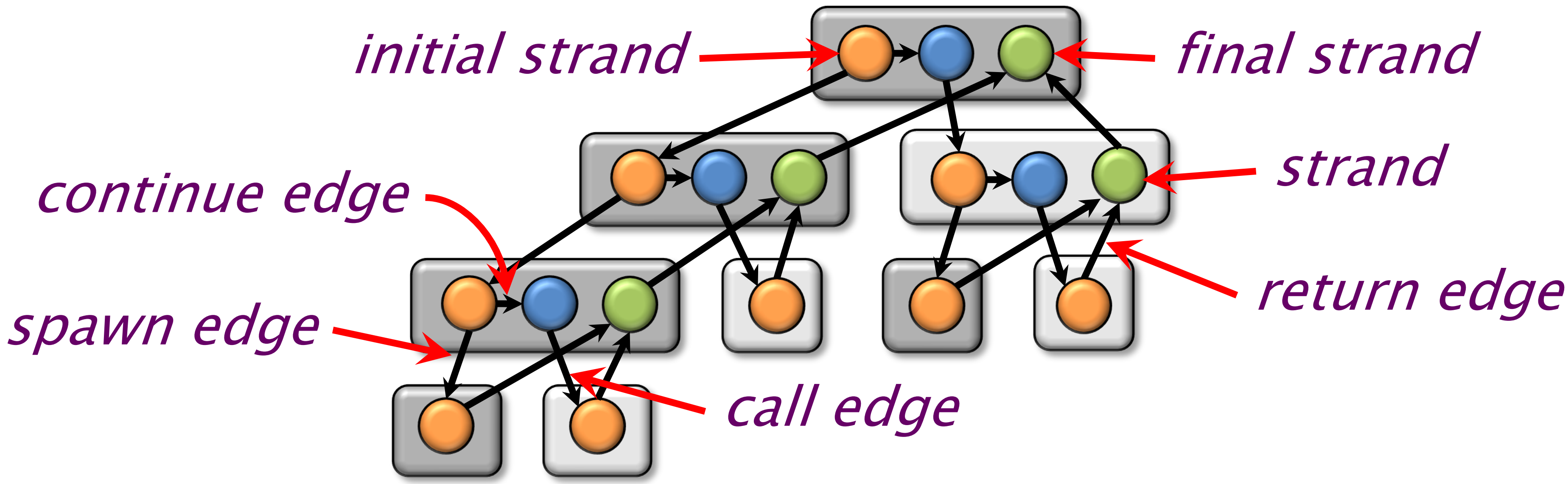
```
int fib(int n) {
  if (n < 2) return n;
  else {
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x + y;
  }
}
```

Example:
`fib(4)`



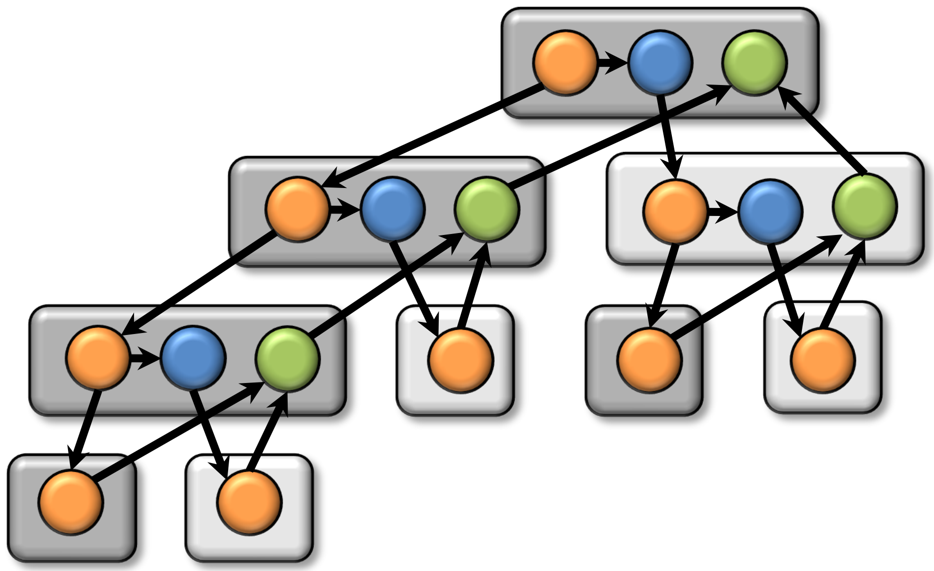
*The **computation dag** unfolds dynamically.*

Trace Dag



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

How Much Parallelism?

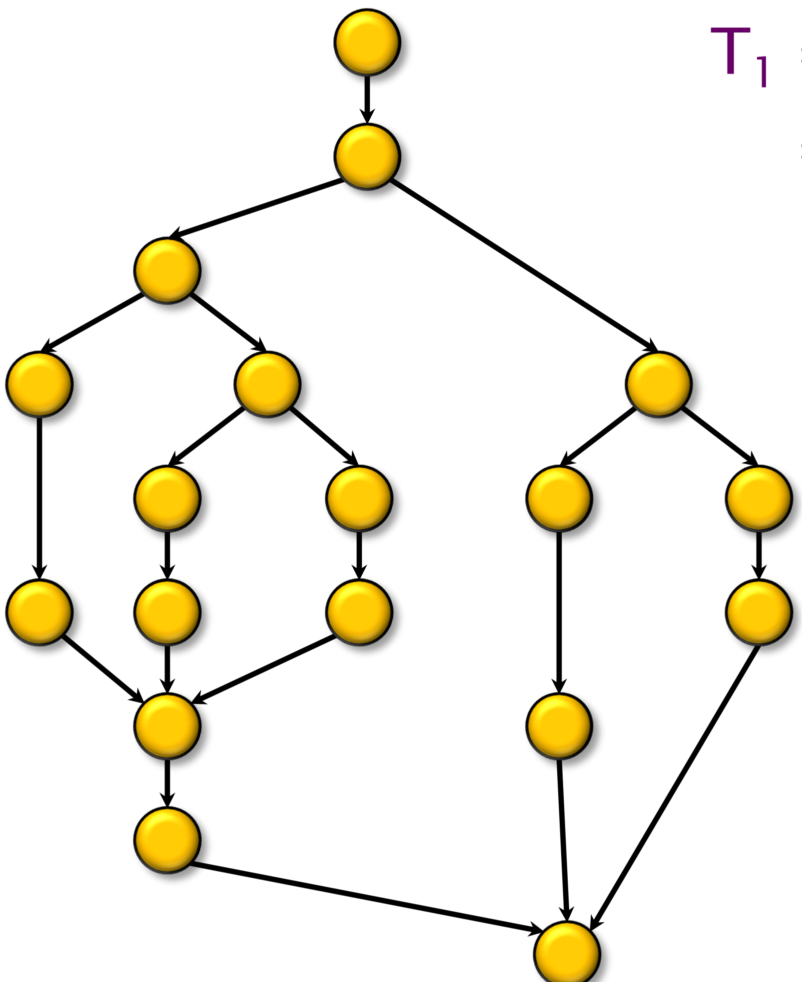


Assuming that each strand executes in unit time, what is the *parallelism* of this computation?

In other words, what is the maximum possible speedup of this computation, where *speedup* is how much faster the parallel code runs compared to the serial code?

Performance Measures

T_P = execution time on P processors

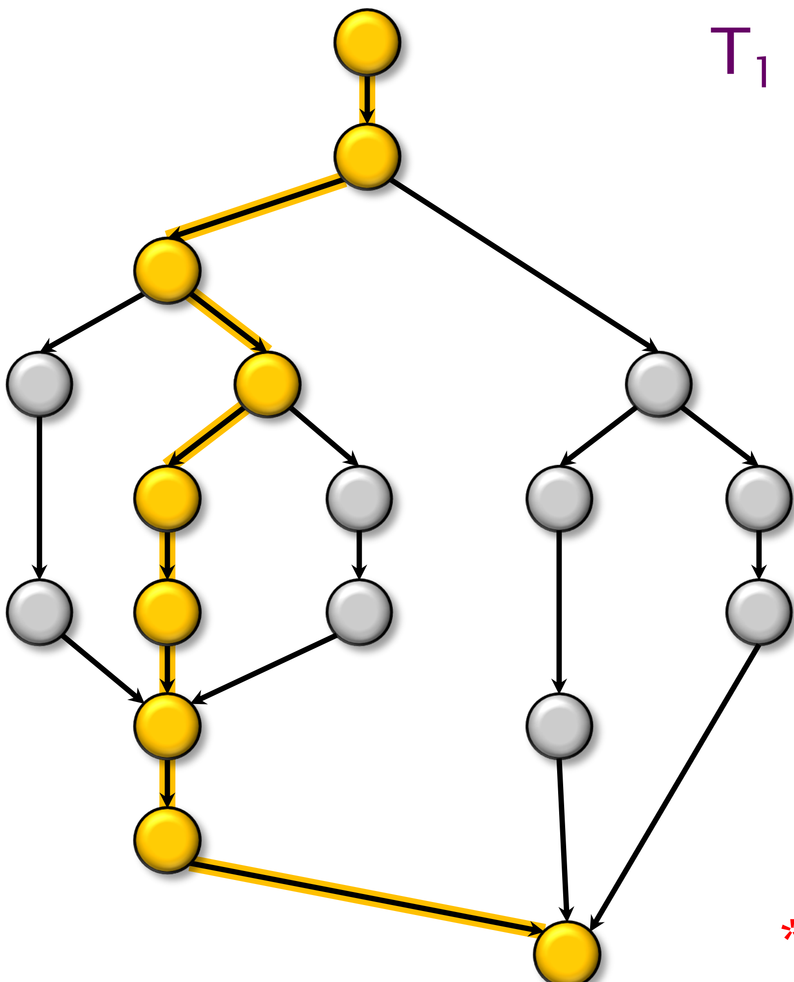


T_1 = *work*

= 18

Performance Measures

T_P = execution time on P processors



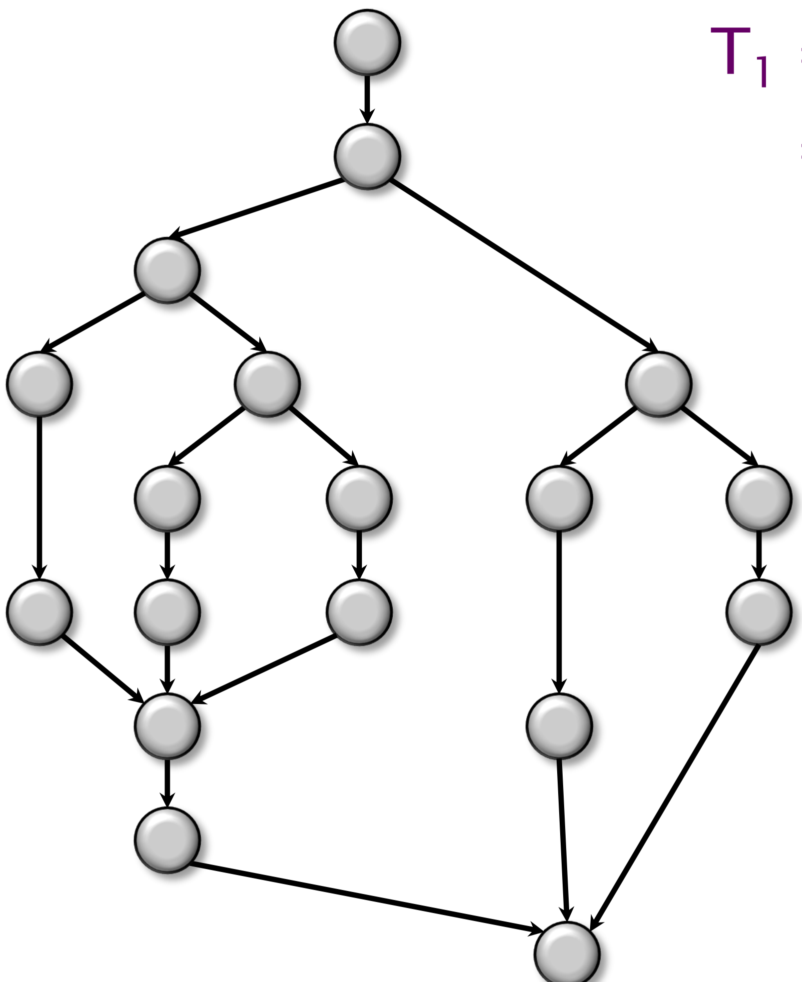
T_1 = *work* = 18

T_∞ = *span** = 9

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

Performance Measures

T_P = execution time on P processors



T_1 = *work*
= 18

T_∞ = *span*
= 9

Work Law
$T_P \geq T_1/P$

Span Law
$T_P \geq T_\infty$

Speedup

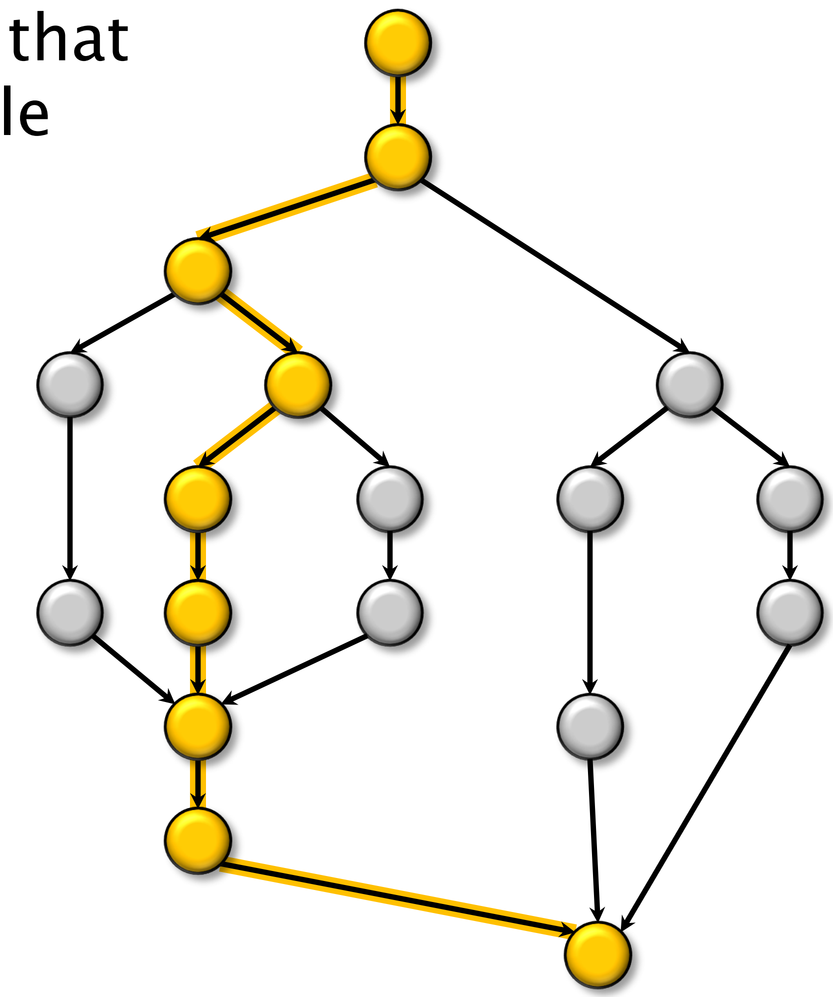
DEFINITION: T_1/T_P = ***speedup*** on P processors.

- If $T_1/T_P < P$, we have ***sublinear speedup***.
- 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 simple performance model, because of the ***WORK LAW*** $T_P \geq T_1/P$.

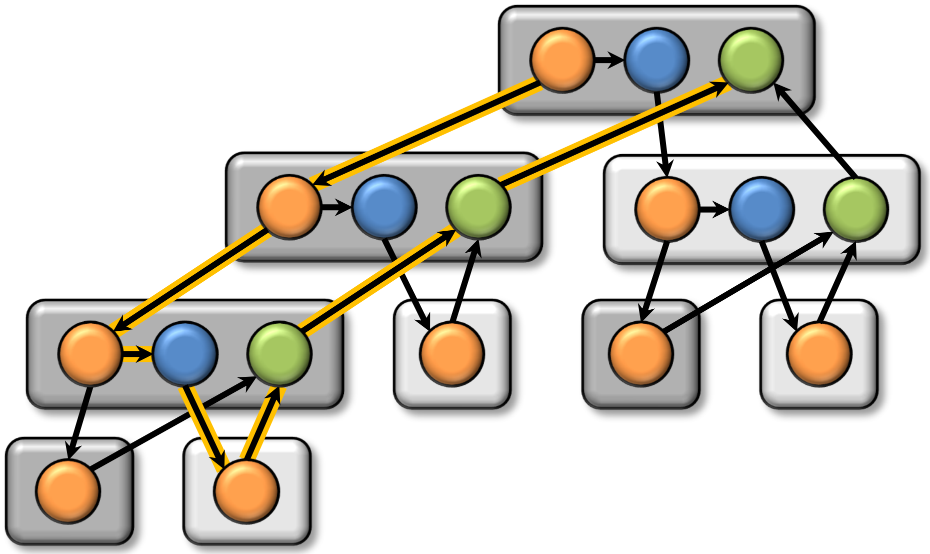
Parallelism

Because the *SPAN LAW* dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

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



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.

Cilk Performance Bound

Definition. T_P — execution time on P processors

T_1 — **work** T_∞ — **span**

T_1 / T_∞ — **parallelism**

Theorem [BL94]. A work-stealing scheduler can achieve expected running time

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

on P processors.

In Practice. Cilk's scheduler achieves execution time

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

on P processors.

Linear Speedup

Corollary. Cilk scheduler achieves near-perfect linear speedup whenever $T_1/T_\infty \gg P$.

Proof. Since $T_1/T_\infty \gg P$ is equivalent to $T_\infty \ll T_1/P$, Cilk's performance bound gives us

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\approx T_1/P \,. \quad \text{(first term dominates)} \end{aligned}$$

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

Cilkscale Scalability Analyzer

- The OpenCilk compiler provides a scalability analyzer called *Cilkscale*, which is similar in some ways to Intel's Cilkview tool.
- Like the Cilksan race detector, Cilkscale uses compiler instrumentation to analyze a serial execution of a program.
- Cilkscale computes work and span to derive upper bounds on parallel performance of all or just part of your program.
- Cilkscale is really three tools in one:
  - an analyzer,
  - an autobenchmarker,
  - a visualizer.

BREAK

Cheetah Runtime System

Cilk Performance Bound

Theorem [BL94]. A work-stealing scheduler can achieve expected running time

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

on P processors.

Time workers spend working.

Time workers spend stealing.

If the program has ample parallelism, i.e., $T_1/T_\infty \gg P$, then the first term dominates, and $T_P \approx T_1/P$.

Parallel Speedup

Let T_S denote the work of a serial program. Suppose the serial program is parallelized. Let T_1 denote the work of the parallel program and let T_∞ denote the span of the parallel program.

Parallel speedup is measured by T_S/T_P .

To achieve linear speedup on P processors over the serial program, i.e., $T_P \approx T_S/P$, the parallel program must exhibit:

- Ample parallelism: $T_1/T_\infty \gg P$.
- High work efficiency: $T_S/T_1 \approx 1$.

The Work-First Principle

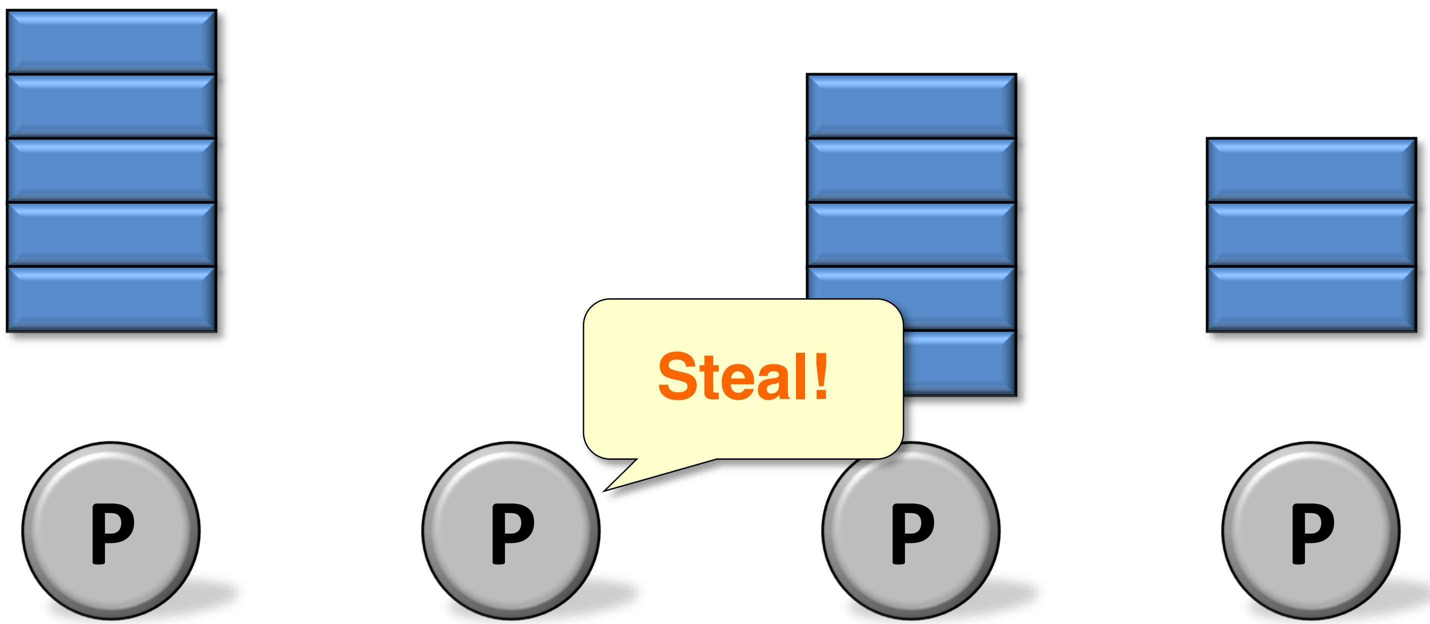
To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk scheduler works to maintain high work-efficiency by abiding by the *work-first principle*:

Optimize for the ordinary serial execution, at the expense of some additional overhead in steals.

Cheetah Runtime System: Required Functionalities

Cilk's Work-Stealing Scheduler

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

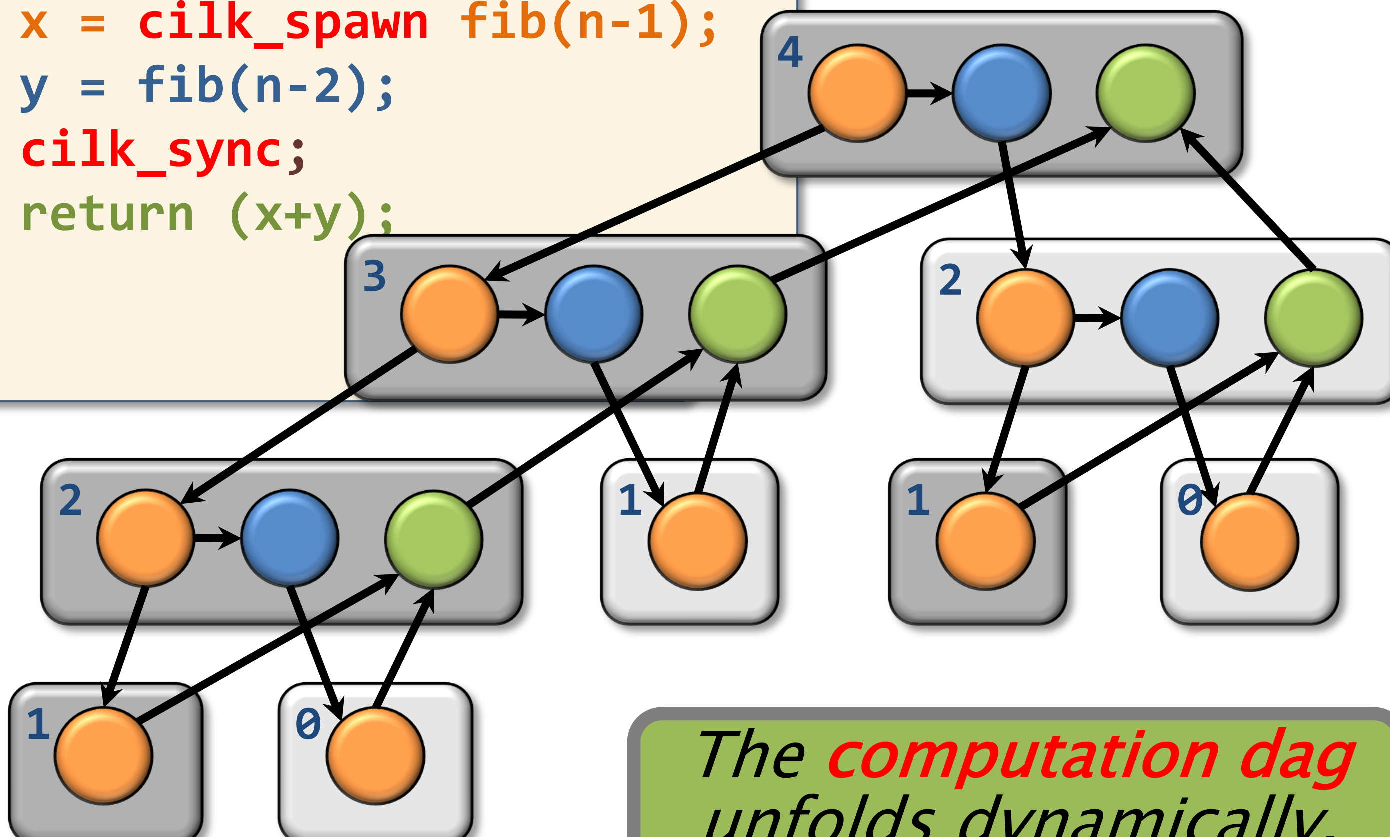


- Single-worker execution mirrors that of its serial projection.
- When a worker runs out of work, it ***steals*** from the top of a random victim's deque.

Cilk's 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)`

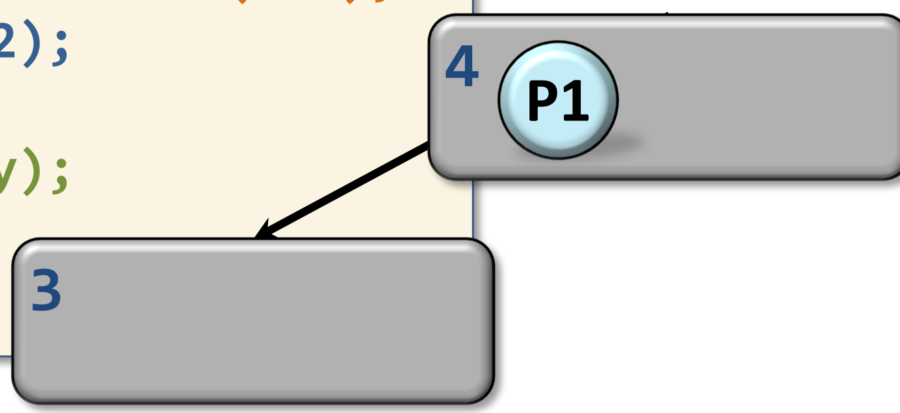


The computation dag unfolds dynamically.

A Worker's Behavior Mirrors Serial Execution

P1 %rip →

```
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:

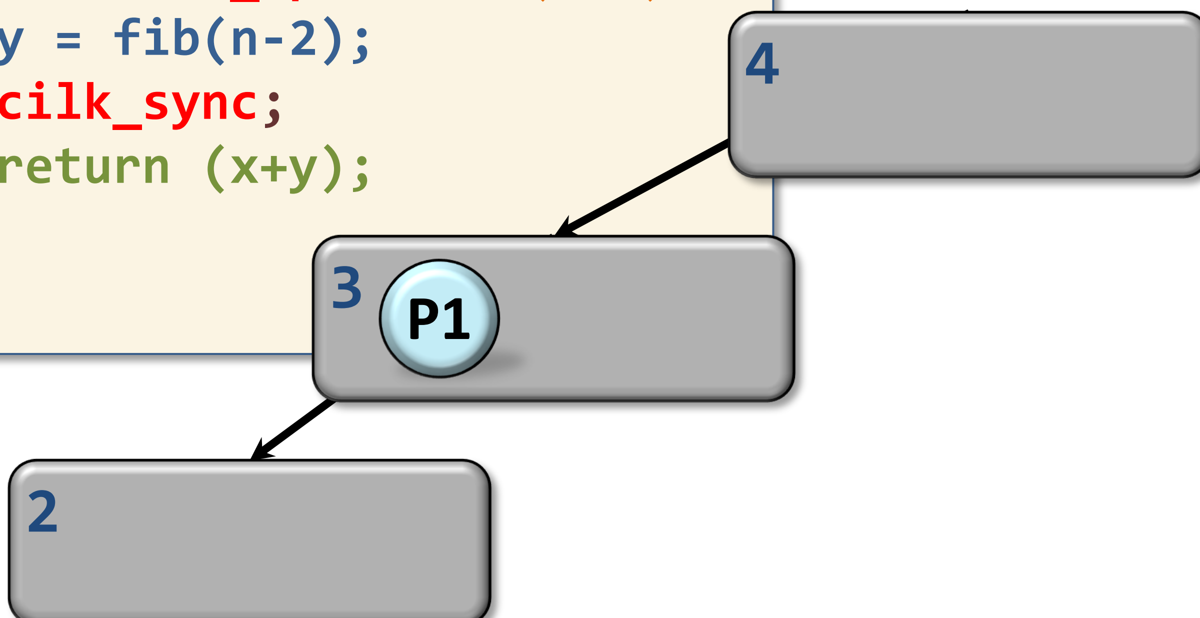
fib(4)

A Worker's Behavior Mirrors Serial Execution

P1 %rip →

```
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)`

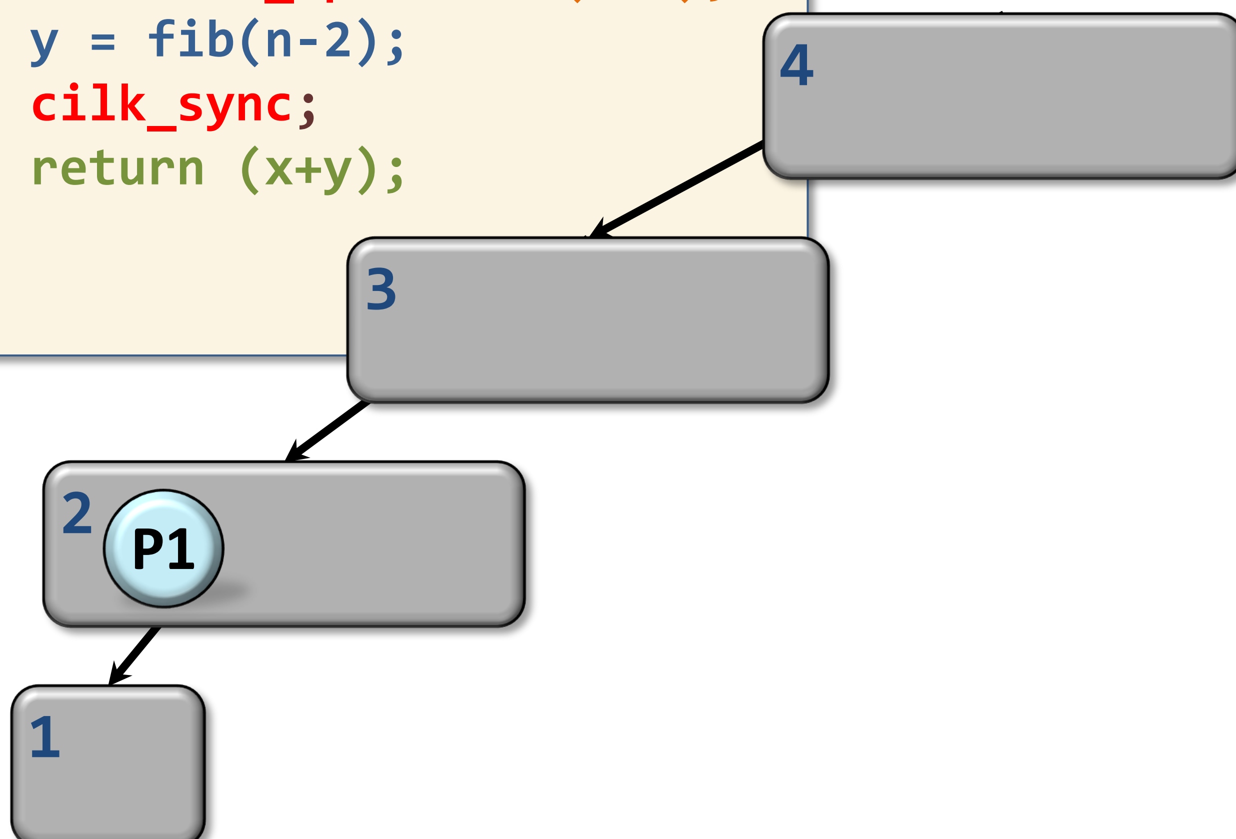


A Worker's Behavior Mirrors Serial Execution

P1 %rip →

```
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)`



Successful Steals Create Parallelism

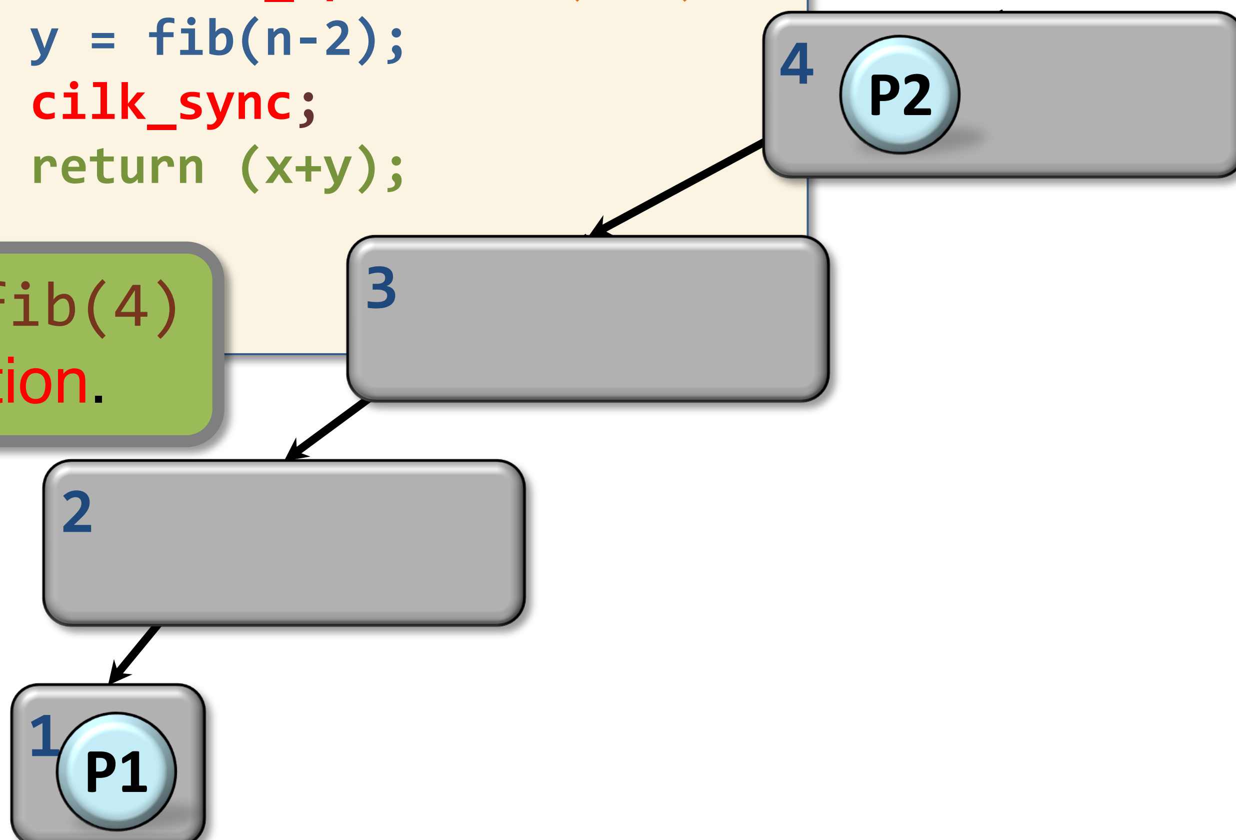
P1 %rip →

P2 %rip →

```
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)

P2 resumes fib(4) mid-execution.



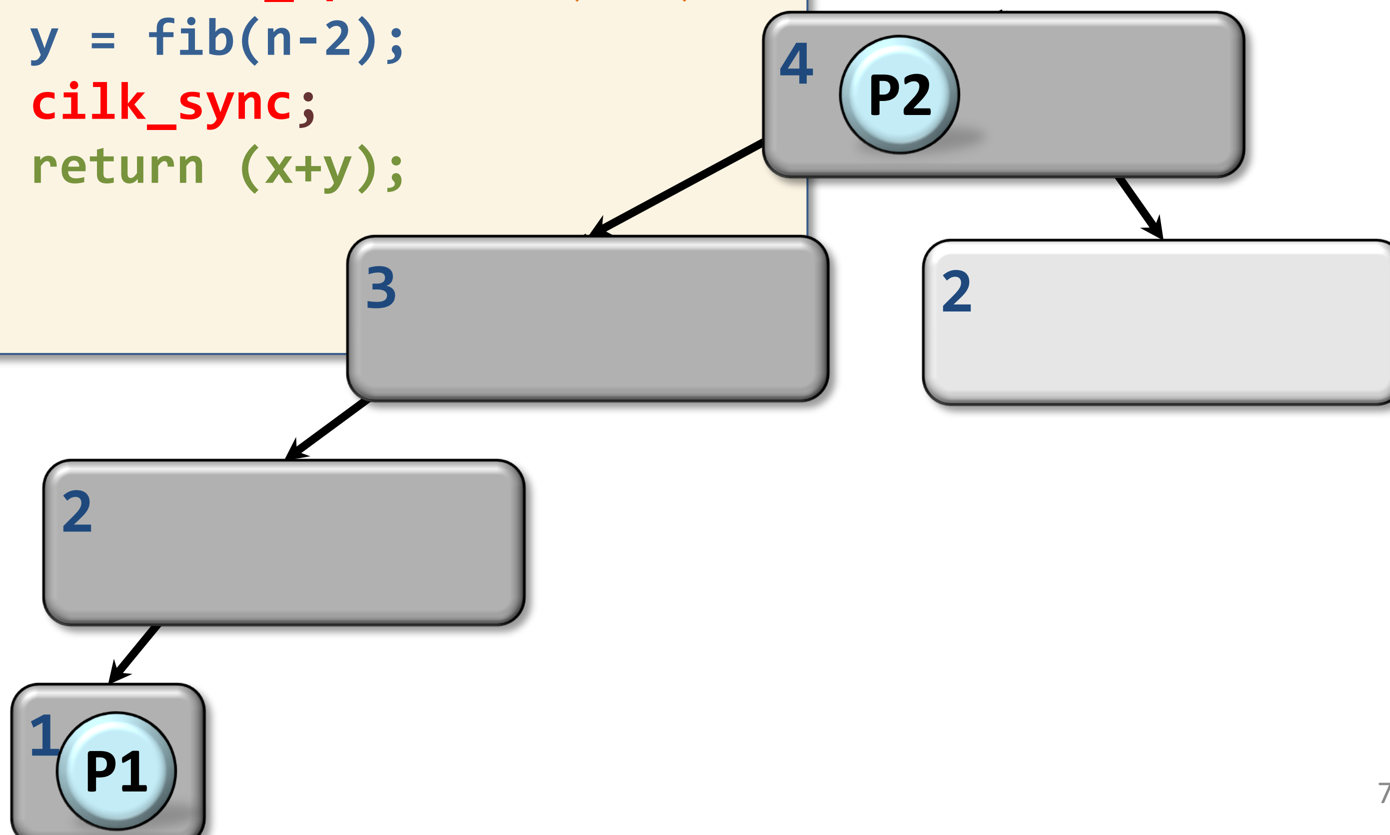
Successful Steals Create Parallelism

P1 %rip →

P2 %rip →

```
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)



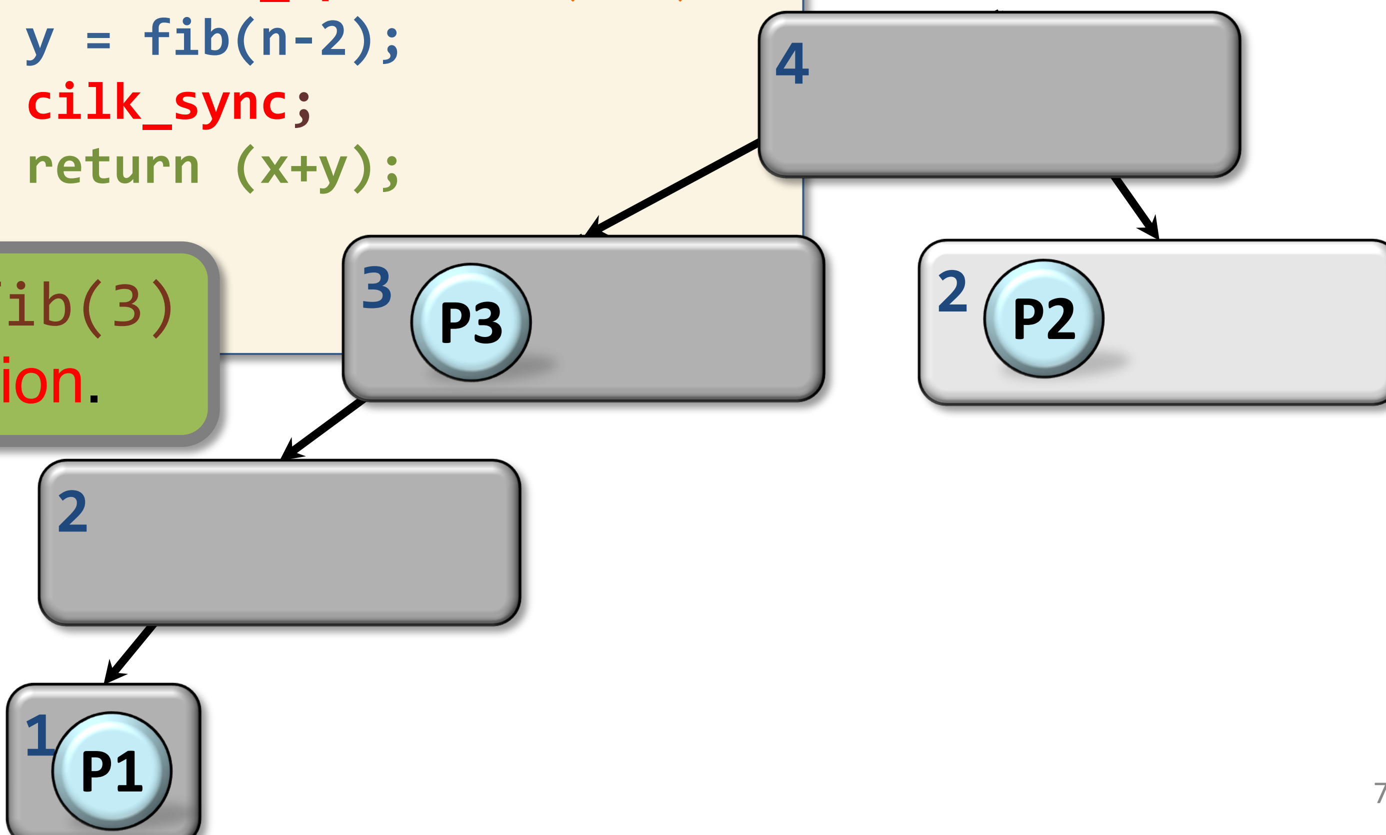
Successful Steals Create Parallelism

P2 %rip →
P1 %rip →

P3 %rip →

```
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);
  }
```

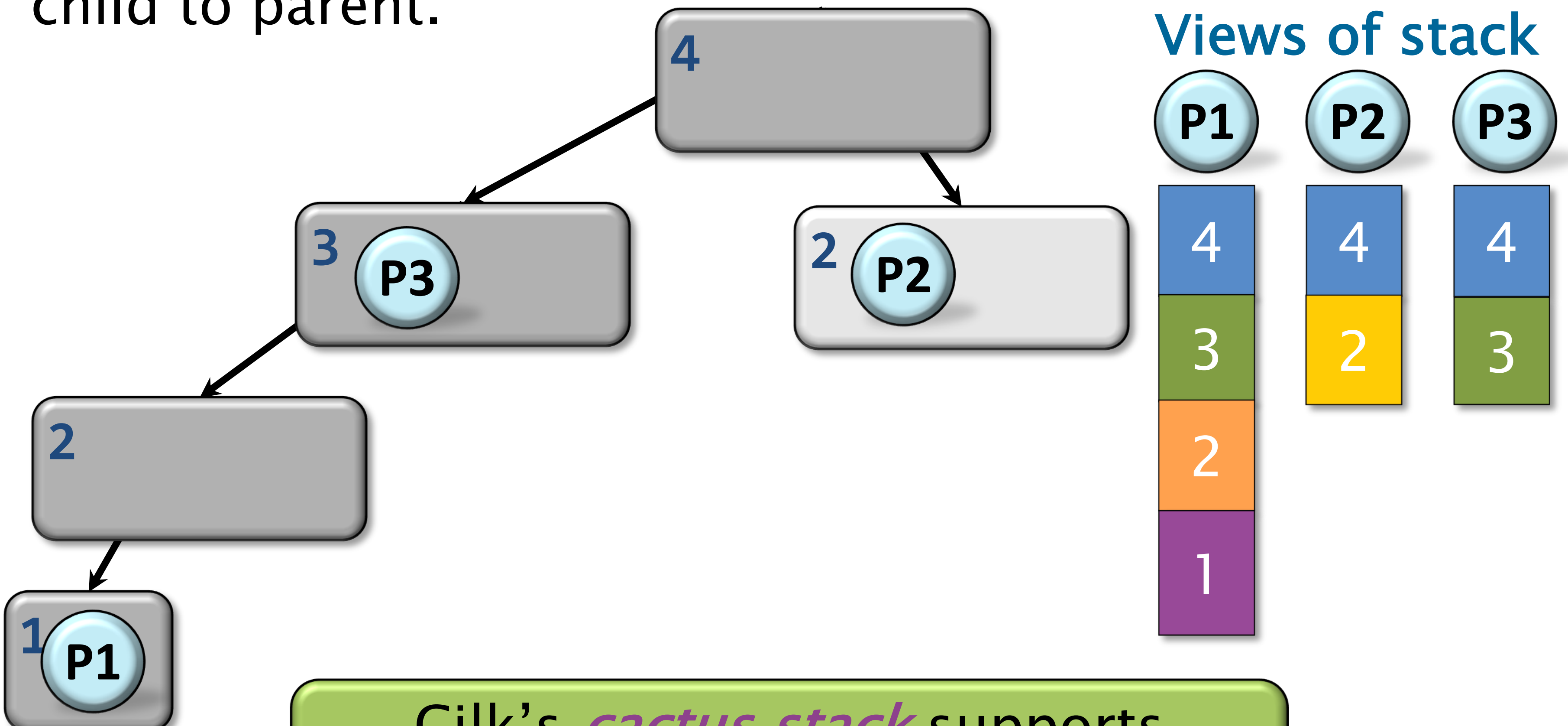
Example:
`fib(4)`



P3 resumes `fib(3)` mid-execution.

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.

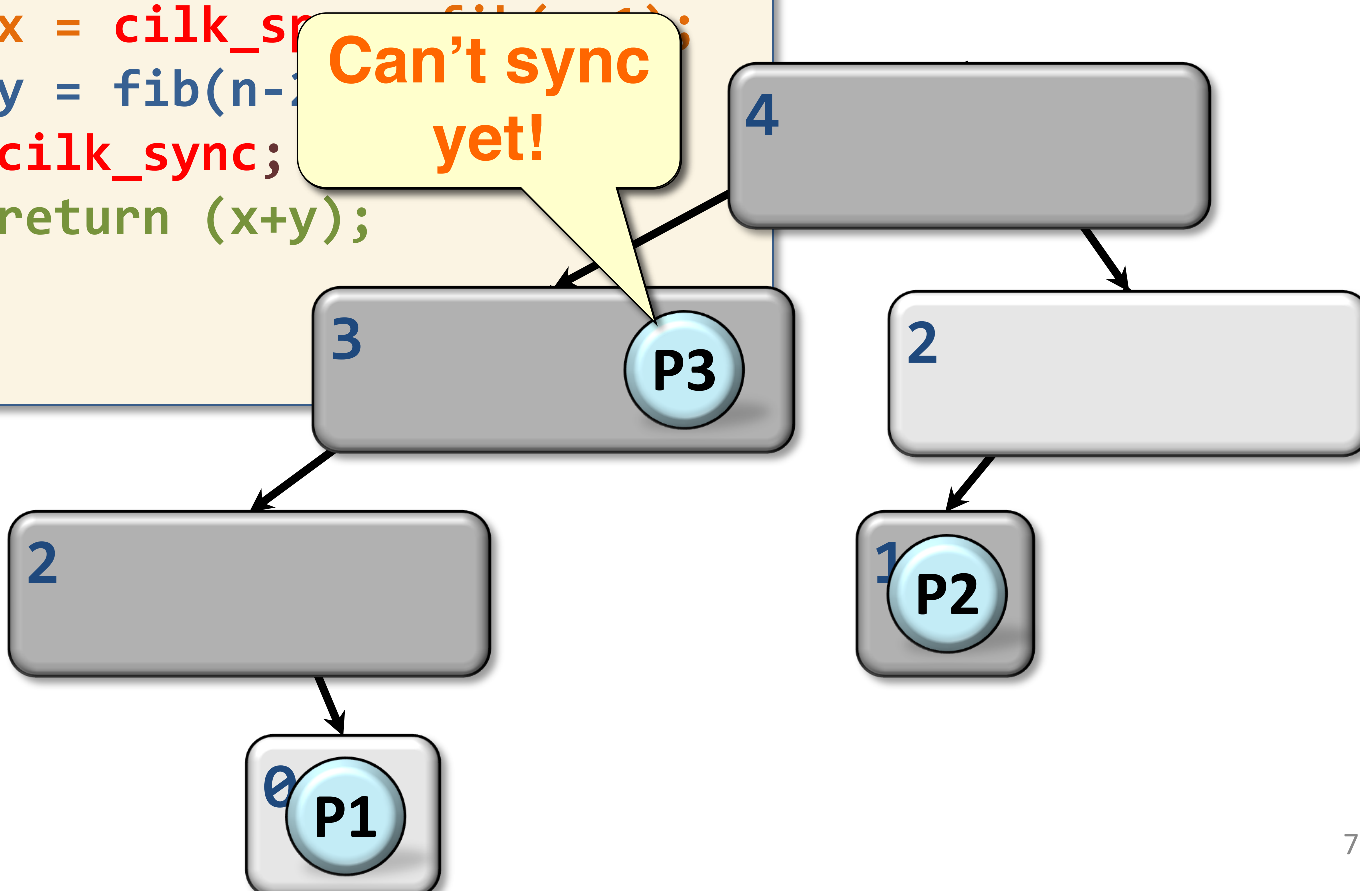
Syncs

P2 %rip →
P1 %rip →

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

P3 %rip →

Example:
`fib(4)`



Required Functionalities

- Each worker needs to keep track of its own execution context, including work that it is responsible for / available to be stolen.
- A worker after a successful steal, can resume the stolen function mid-execution.
- Upon a sync, a worker needs to know whether there is any spawned subroutine still executing on another worker.
- The runtime must maintain the cactus stack abstraction as the parallel execution unfolds.

Cheetah Runtime Data Structures

The Cheetah runtime utilizes three basic data structures as workers execute work:

- A *work deque* storing the execution context of ready work.
- A *Cilk stack frame structure* to represent each spawning function (*Cilk* function).
- A *closure tree* to represent function instances that has every been stolen to support true parallel execution.

Division of Labor

The work-first principle guides the division of the Cilk scheduler between the compiler and the runtime library.

Compiler

- Manages a handful of small data structures (e.g., initialization / operations on Cilk stack frames and deques).
- Implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).

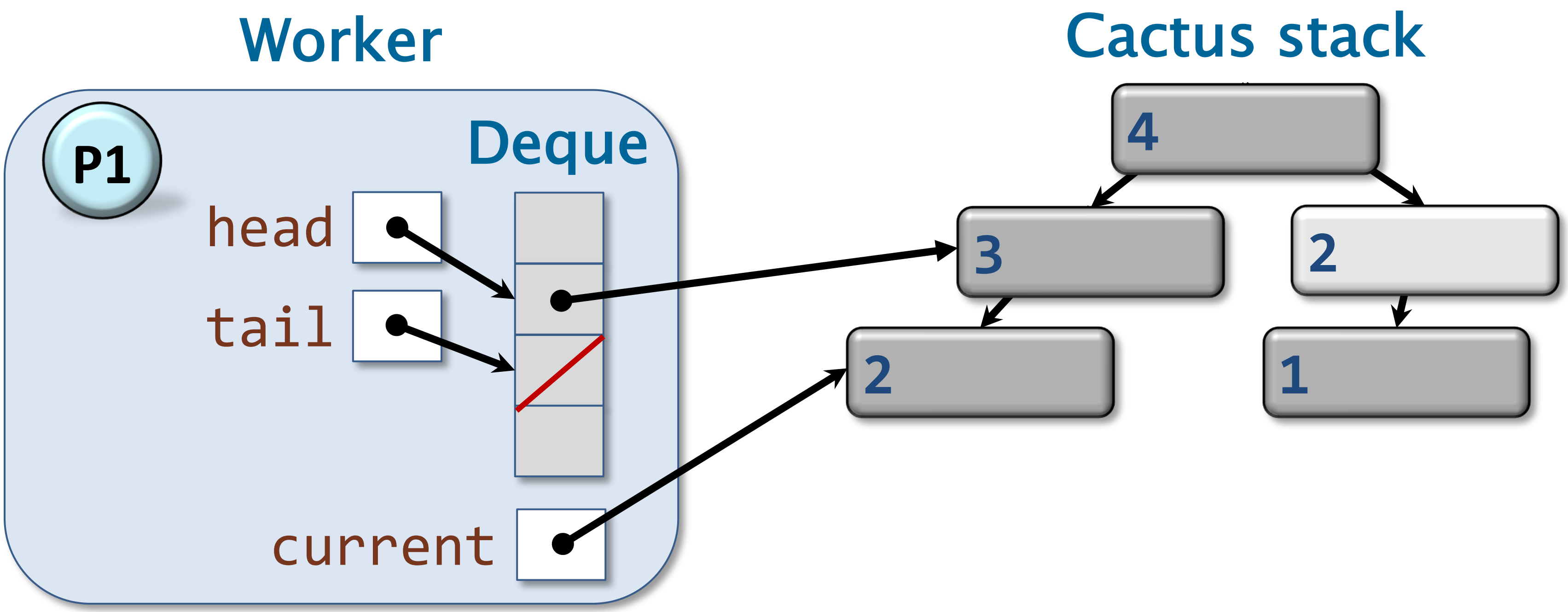
Runtime library

- Manages the more heavy-weight data structures (e.g., the closure tree).
- Handles slow paths of execution, e.g., when a steal occurs.

Cheetah Runtime System: Organization of the Runtime Data Structure

Deque of Frames

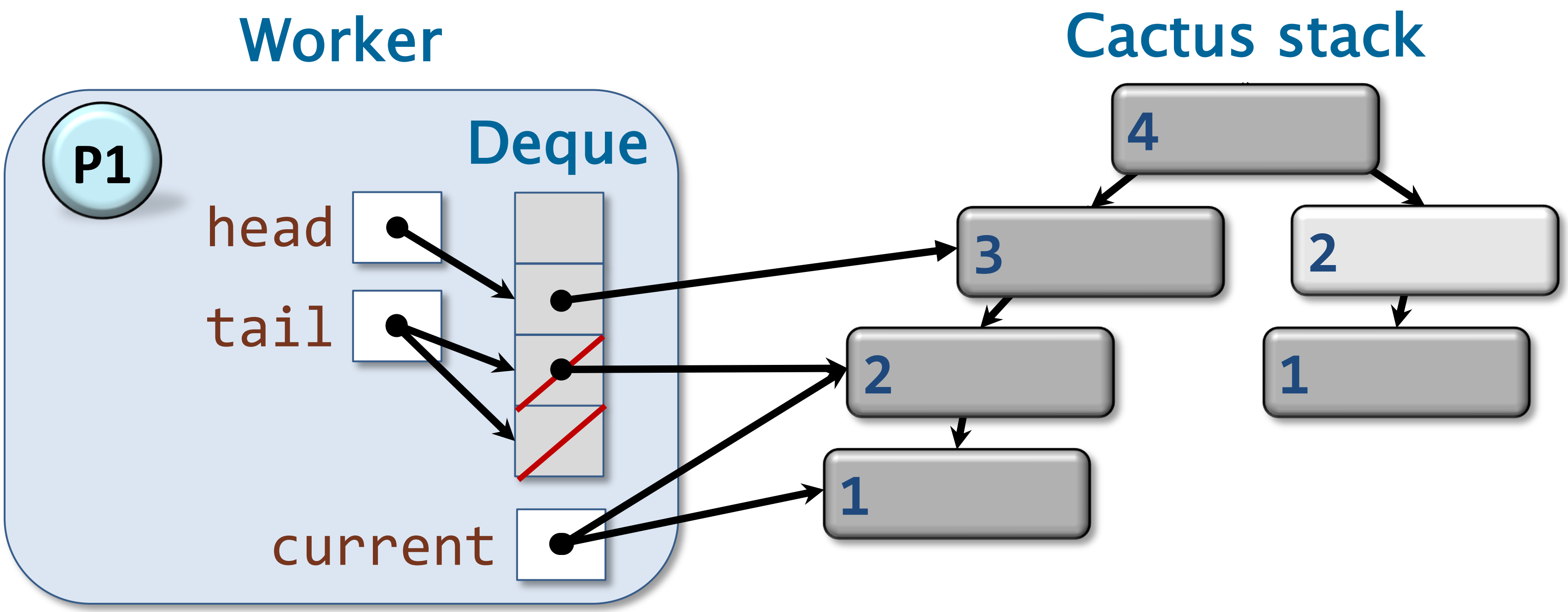
Each Cilk worker maintains a deque of references to Cilk stack frames* containing work available to be stolen.



*We'll discuss what a Cilk stack frame contains later.

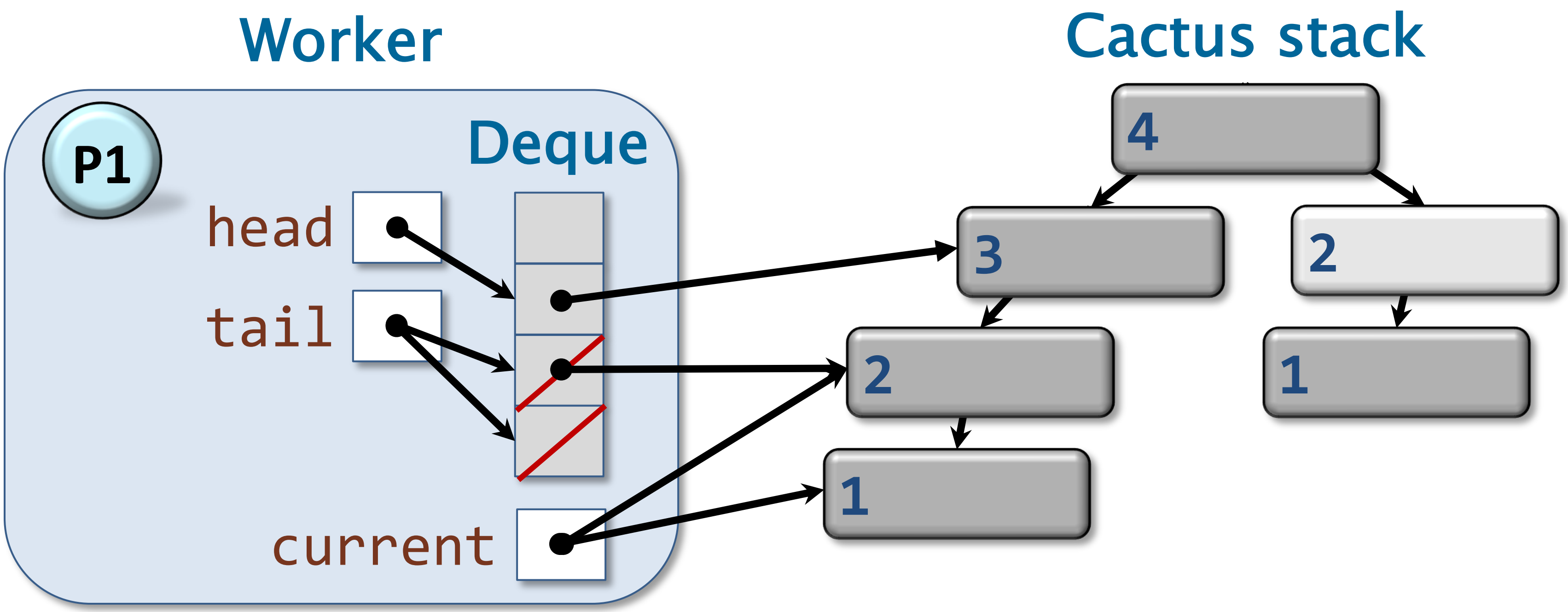
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.



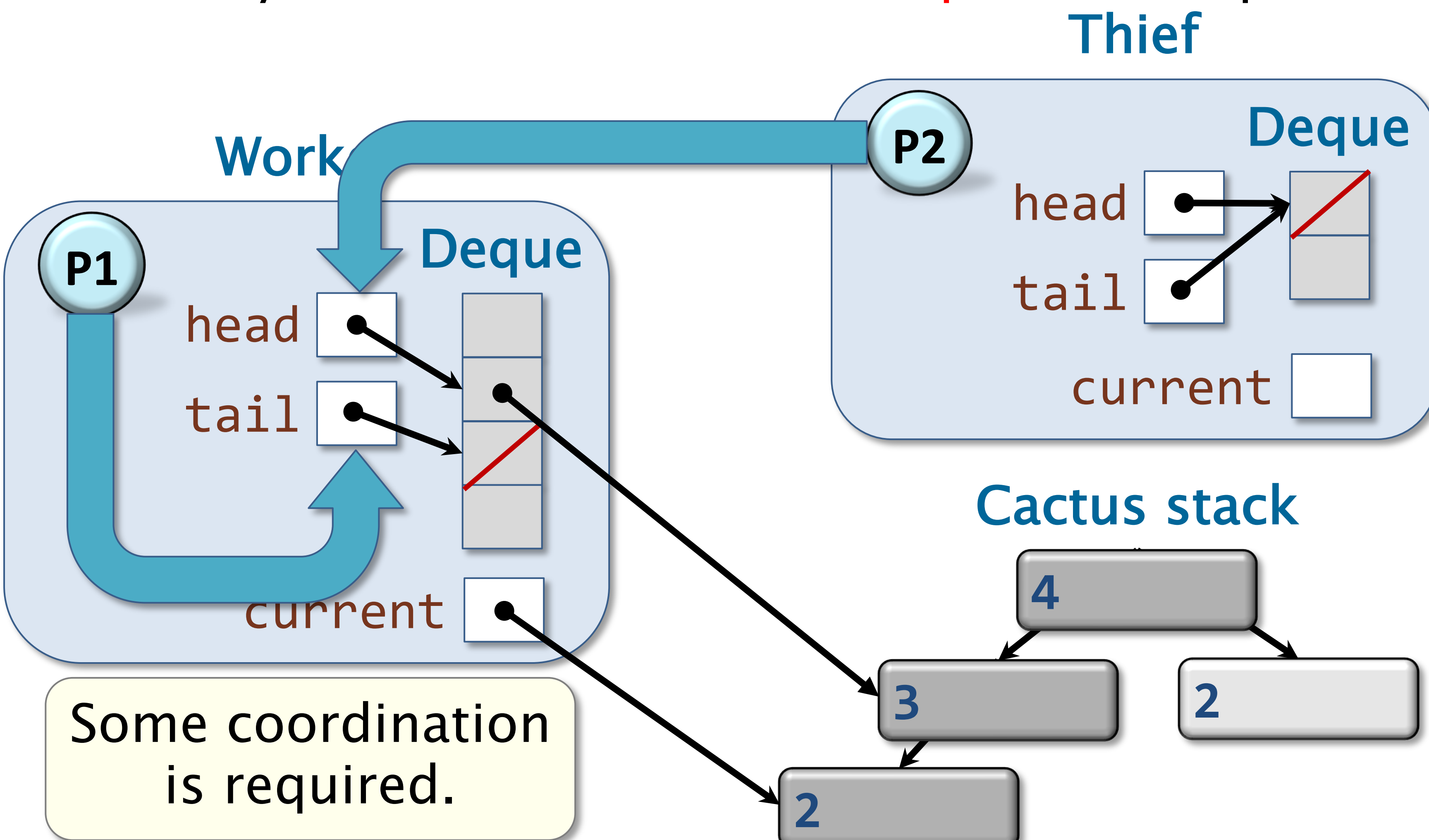
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.



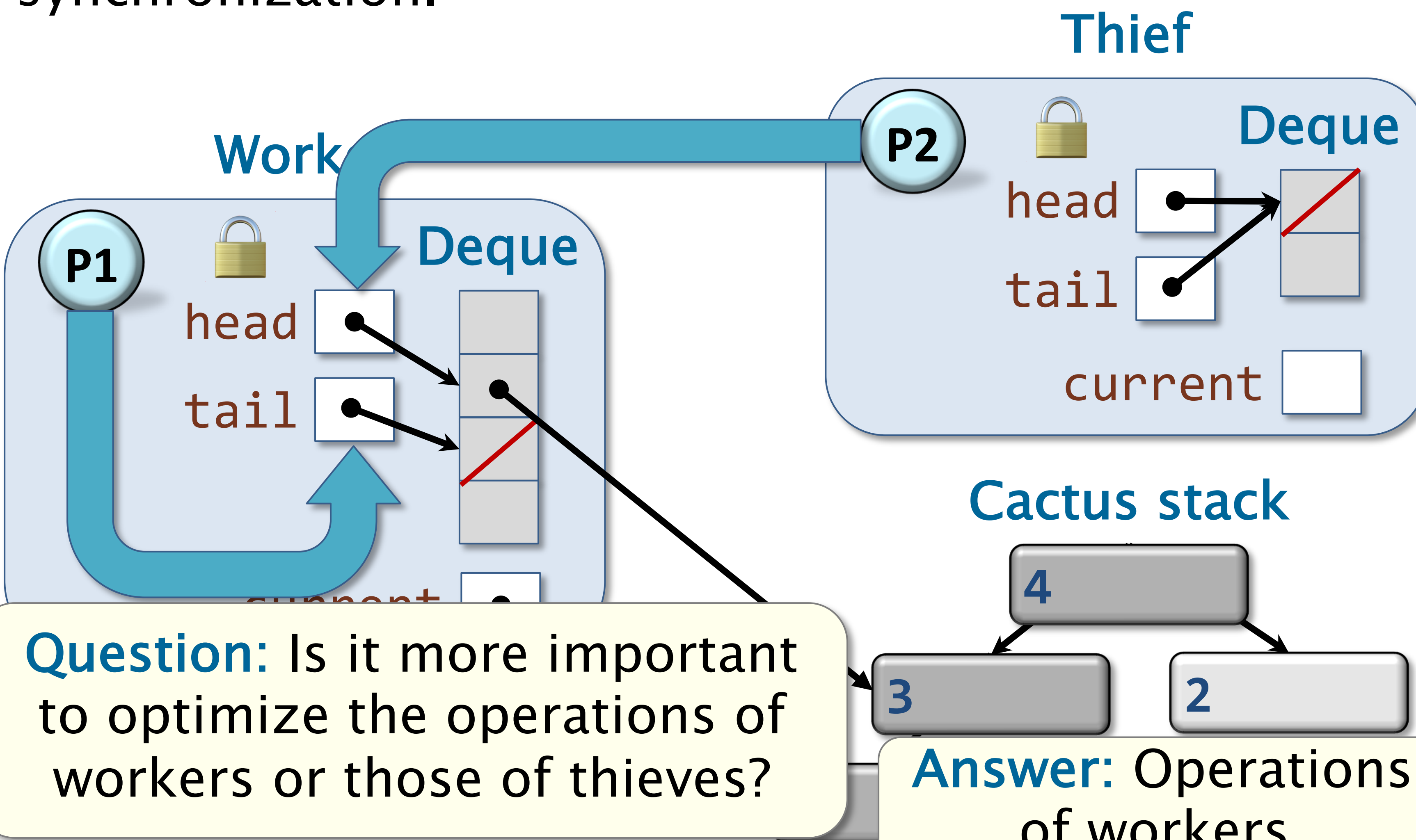
Stealing Frames

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



Synchronizing Thieves and Workers

Cilk uses a lock associated with each deque to perform synchronization.



Question: Is it more important to optimize the operations of workers or those of thieves?

Answer: Operations of workers.

Popping the Deque

When a worker is about to return from a spawned function, it needs to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.

Question: Which case is more important to optimize?

Answer: Case 1.

The THE Protocol

Worker protocol

```
void push() { tail++; }
bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

The worker and the thief coordinate using *the THE protocol*

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() { tail++; }
bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Observation I:
Synchronization is only necessary when the deque is almost empty.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() { tail++; }
bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Observation II: The pop operation is more likely to succeed than fail.

Thief protocol

```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
  unlock(L);
  return SUCCESS;
}
```

The THE Protocol

Worker protocol

```
void push() {
bool pop() {
  tail--;
  if (head > tail) {
    tail++;
    lock(L);
    tail--;
    if (head > tail) {
      tail++;
      unlock(L);
      return FAILURE;
    }
    unlock(L);
  }
  return SUCCESS;
}
```

Workers pop the deque optimistically…

ork-First
Optimize the operations of workers.

Thief protocol

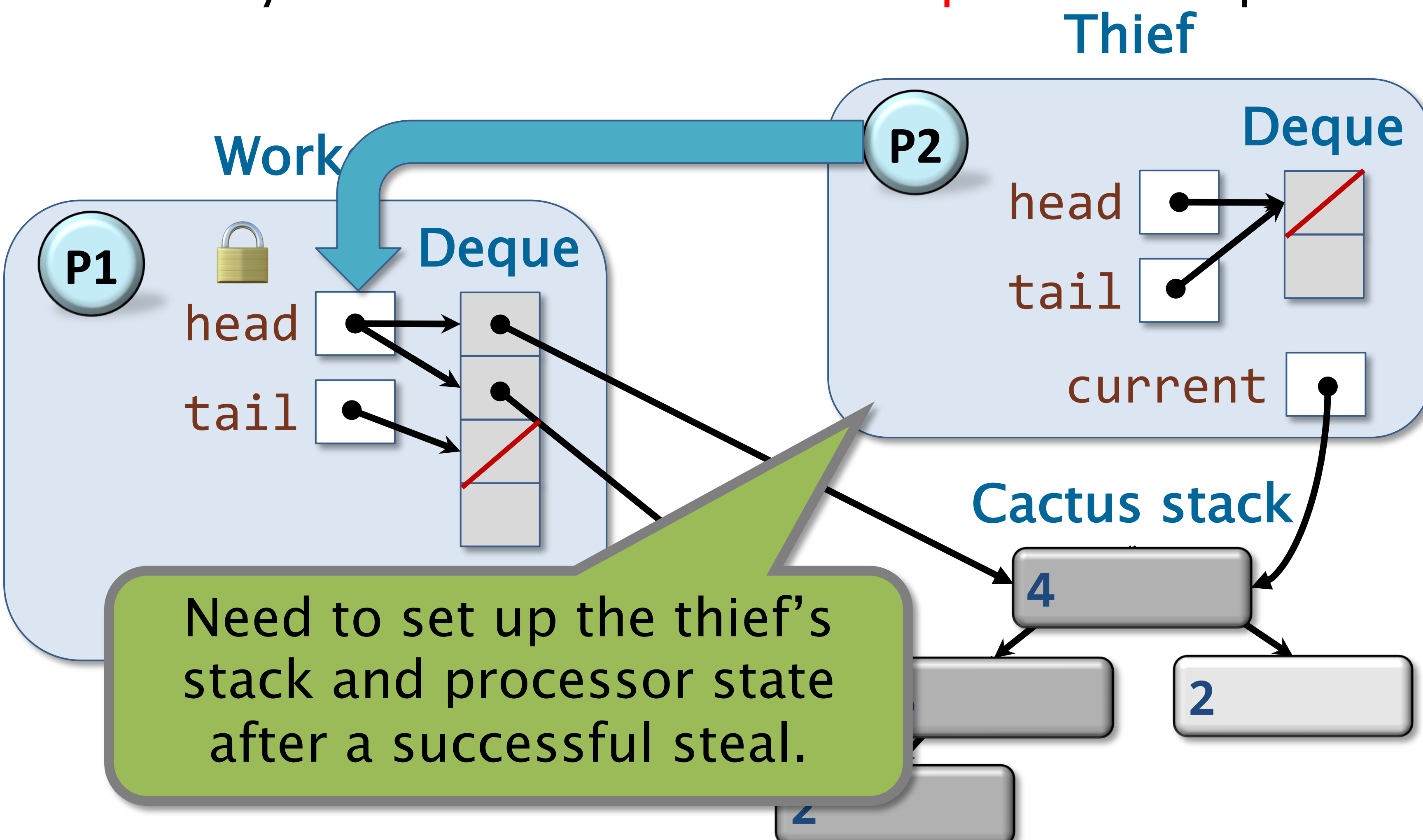
```
bool steal() {
  lock(L);
  head++;
  if (head > tail) {
    head--;
    unlock(L);
    return FAILURE;
  }
```

Thieves always grab the lock.

…and only grab the deque's lock if the deque appears to be empty.

Successful Steal

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.



Saving and Restoring Processor State

To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.

Cilk code

```
x = cilk_spawn fib(n-1);
```

Compiled pseudocode

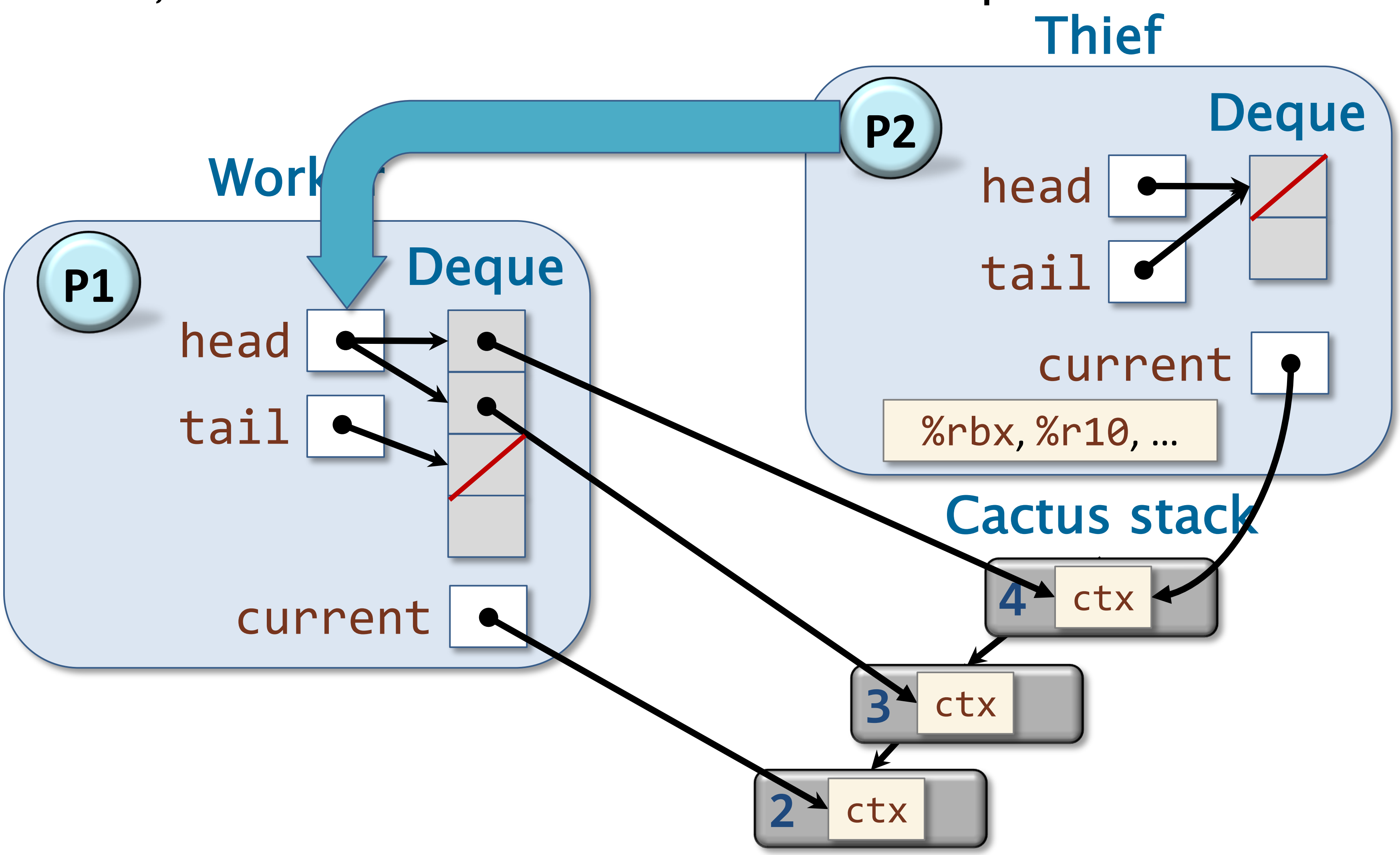
```
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
```

Buffer to store processor state.

Save processor state into ctx and allow a worker to resume the continuation.

Deque References to Frames

Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.

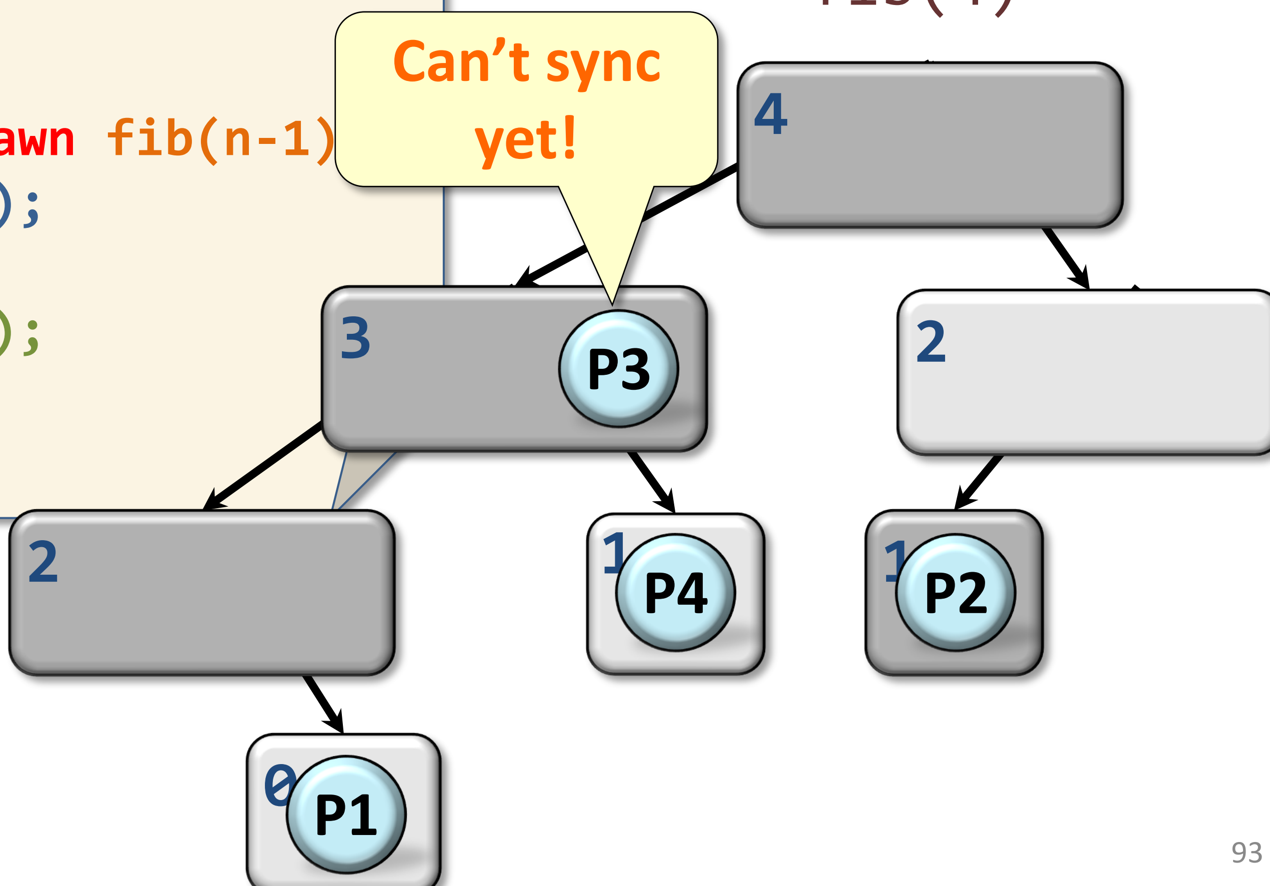


Semantics of Sync

A **cilk_sync** waits on child frames, not just on workers.

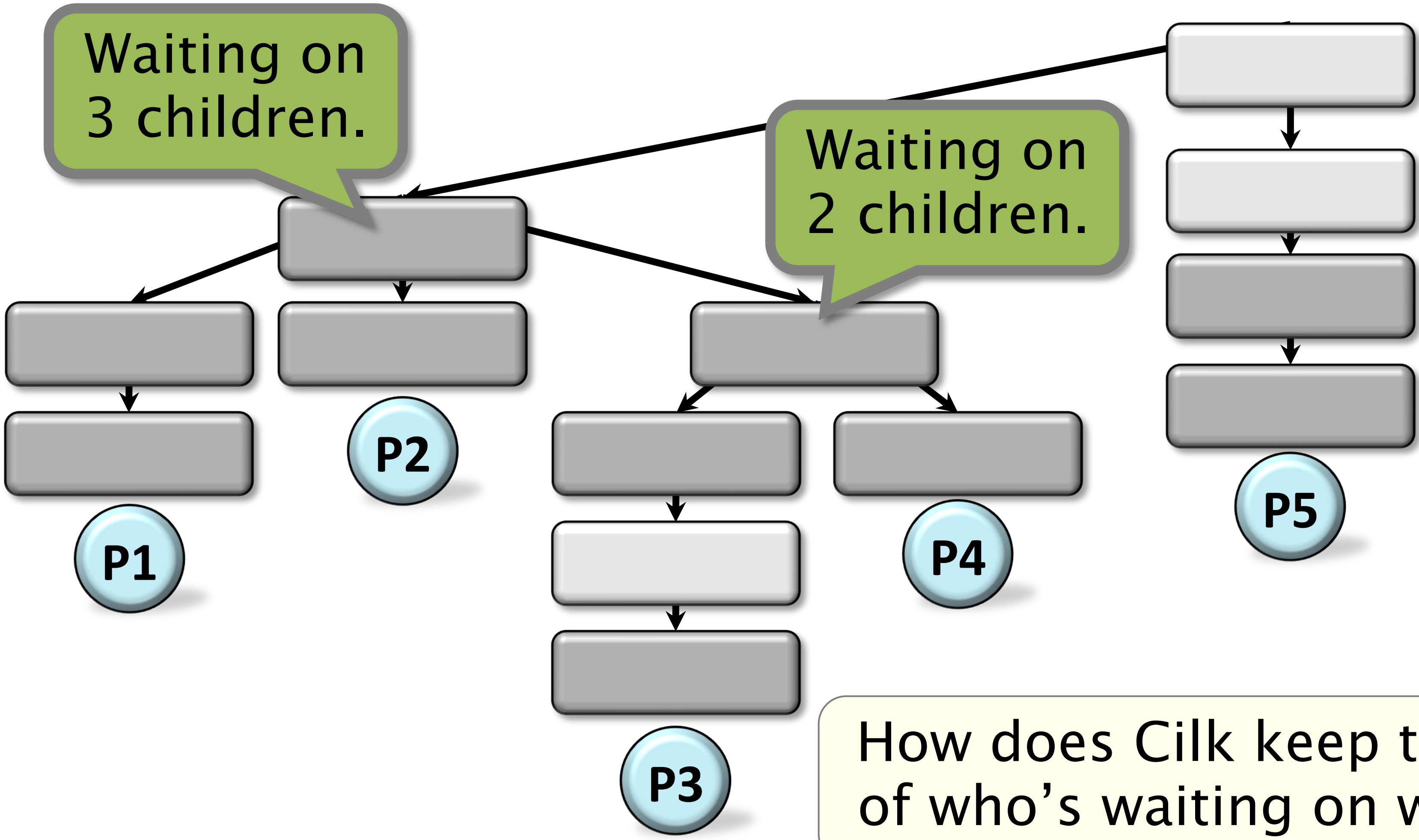
```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1)
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

Example:
fib(4)



Nested Synchronization

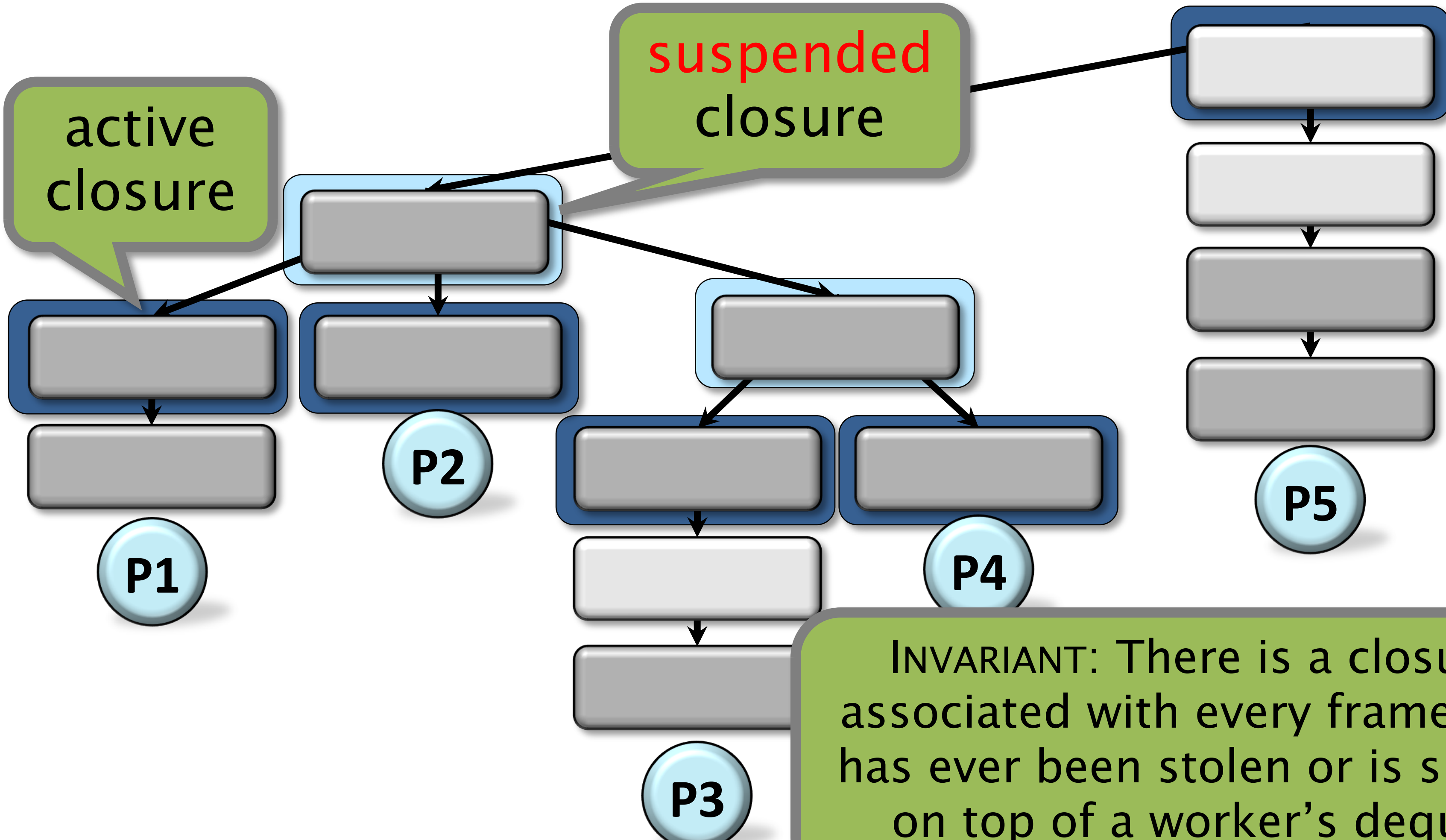
Cilk supports *nested synchronization*, where a frame waits only on its child subcomputations.



How does Cilk keep track of who's waiting on who?

Closure Tree

The Cilk runtime maintains a tree of *closures* to keep track of synchronization information.

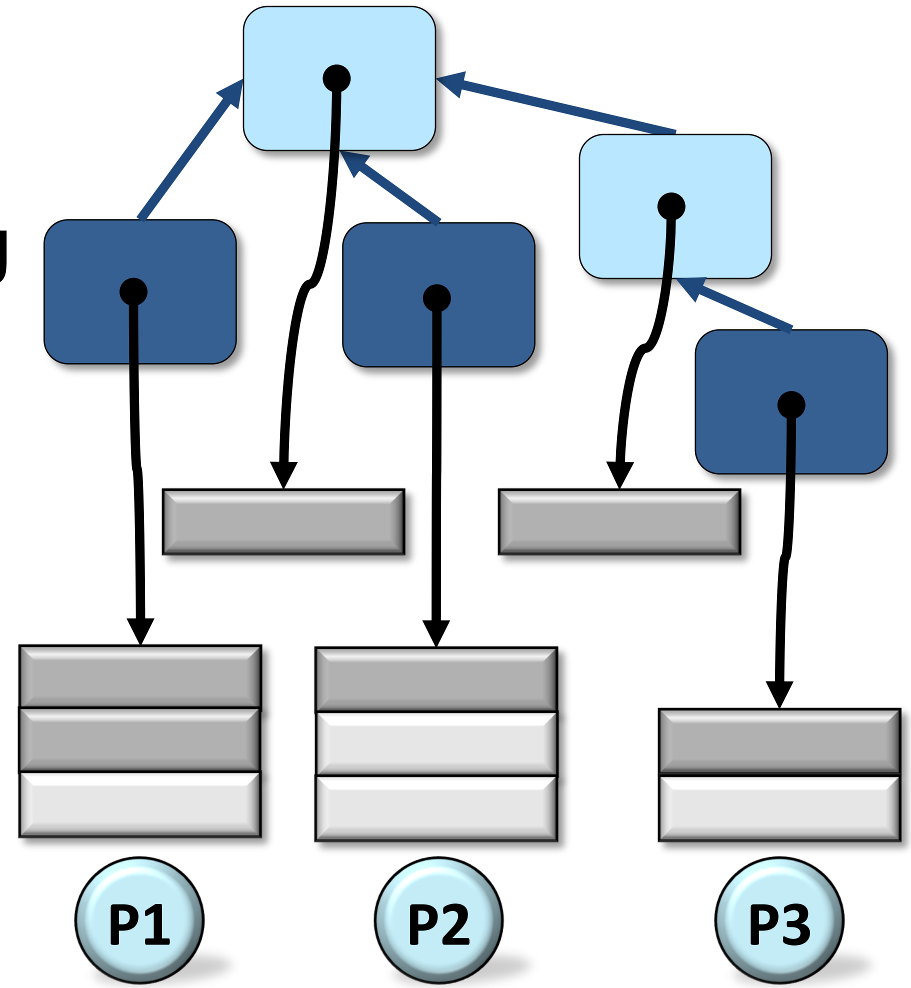


INVARIANT: There is a closure associated with every frame that has ever been stolen or is sitting on top of a worker's deque.

Closure Data

To maintain the state of the running program, each closure maintains:

- A join counter of the number of outstanding spawned children.
- References to parent and child closures.
- References into the corresponding Cilk stack frames on the cactus stack.



Common Case for Sync

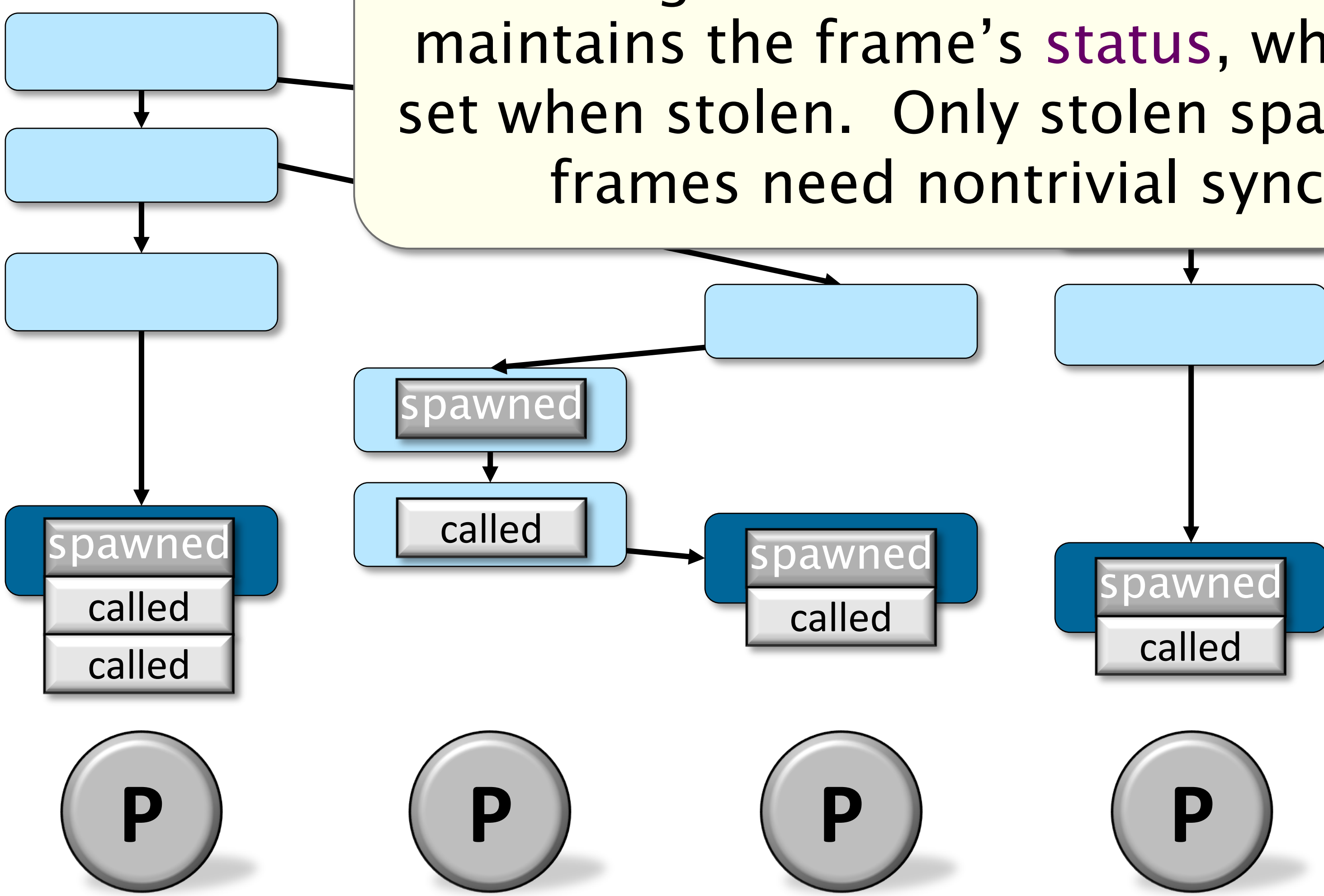
Question: If the program has ample parallelism, what do we expect typically happens when the program execution reaches a `cilk_sync`?

Answer: The executing function contains no outstanding spawned children.

How does the scheduler optimize for this case?

Managing the Full-Frame Tree: Sync

The flag field in a Cilk stack frame maintains the frame's status, which is set when stolen. Only stolen spawning frames need nontrivial sync.



Compiled Code for Sync

Like `cilk_spawn`, a `cilk_sync` is compiled using `setjmp`, in order to save the processor's state when the frame is suspended.

Cilk code

```
cilk_sync;
```

C pseudocode

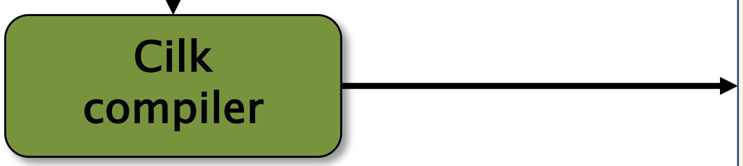
```
BUFFER ctx;
…
if (!setjmp(&ctx))
  __cilkrts_sync(&ctx);
```

Same buffer as used for spawns.

Call into the runtime to suspend the frame.

Full Compiler/Runtime ABI

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```




```
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  /* s = cilk_spawn bar(n); */
  if (!setjmp(sf.ctx))
    bar_spawn_helper(&x, n);
  y = baz(n);
  /* cilk_sync */
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_leave_frame(&sf);
  return result;
}

void bar_spawn_helper(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_helper(&sf);
  __cilkrts_detach();
  *x = bar(n);
  __cilkrts_leave_frame_helper(&sf);
}
```

C pseudocode of compiled result

Full Compiler/Runtime ABI

Cilk stack frame contains the buffer for saving execution context

save execution context to prepare for spawn

save execution context in the event of a non-trivial sync

push the Cilk stack frames corresponding to the spawning of bar onto the deque

clean up and try to pop the deque

```
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  /* s = cilk_spawn bar(n); */
  if (!setjmp(sf.ctx))
    bar_spawn_helper(&x, n);
  y = baz(n);
  /* cilk_sync */
  if (sf.flags & CILK_FRAME_UNSYNCHED)
    if (!setjmp(sf.ctx))
      __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_leave_frame(&sf);
  return result;
}

void bar_spawn_helper(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_helper(&sf);
  __cilkrts_detach();
  *x = bar(n);
  __cilkrts_leave_frame_helper(&sf);
}
```

Hands-On with Cheetah

- First: cheetah runtime overview
- Compile and run `nqueens`:

```
$ cd /tutorial
$ make cheetah nqueens
$ ./nqueens 13
```

- Enable stats in `cheetah`: set `CILK_STATS` to `1` in `cheetah/runtime/rts-config.h`.
- Compile `cheetah` and run `nqueens` again.
- Add instrumentation to correct stats output. (Be sure to recompile `nqueens` if you modify `cilk2c_inline.c`)

Cheetah Runtime System: Design Choices

The Work-First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk runtime system works to maintain high work-efficiency by abiding by the ***work-first principle***:

Optimize for the ordinary serial execution, at the expense of some additional overhead in steals.

Division of Labor

The work-first principle guides the division of the Cilk runtime system between the compiler and the runtime library.

- The compiler implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).
- The runtime library handles slow paths of execution, e.g., when a steal occurs.

Examples:

- The THE protocol
- The implementation of **`cilk_sync`**
- The use of Cilk stack frames versus closures

Implementation of Spawn

Classic randomized work-stealing:
Continuation-stealing / work-first: go execute the spawned child and package up the continuation to be stolen.

Alternative: *child-stealing / help-first*: push the spawned child onto the deque so it can be stolen and continue execute the spawning function. Pop off spawned children to execute when encounter a sync.

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

Issues with Child-Stealing: Space

```
for (int i = 0; i < 1000; ++i) {
  cilk_spawn foo(i);
}
cilk_sync;
```

Child-stealing: will create 1000 work items and push them onto the deque before start doing any work!

Continuation-stealing: work on the spawned iteration and let the rest of the loops to be stolen potentially.

Continuation-Stealing vs Child-Stealing

Continuation-stealing:

- Potentially better space utilization.
- Better work-efficiency.
- One-worker execution follows that of serial projection.
- For private caches, one can bound the cache misses due to parallel executions.

Child-stealing:

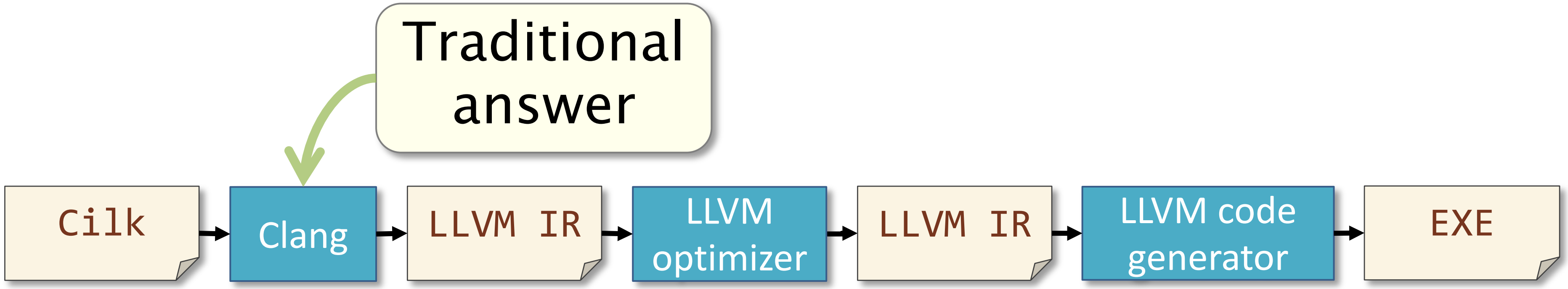
- Potentially worse space utilization.
- Worse work-efficiency.
- One-worker execution does NOT follow that of serial elision.
- No proven bound on cache misses due to parallel executions.

Reads:
"Only Monsters
Steal Children."

OpenCilk Compiler Middle-End

Compilation Pipeline

QUESTION: Where does the compiler deal with `cilk_spawn`, `cilk_sync`, and `cilk_for`?



Example: Normalize

```
__attribute__((const)) double norm(const double *X, int n);

void normalize(double *restrict Y, const double *restrict X,
               int n) {
  for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Test: Random vector, $n=64M$

Machine: Amazon AWS c4.8xlarge

Running time: $T_S = 0.312$ s

Performance of Parallel Normalize

```
__attribute__((const)) double norm(const double *X, int n);

void normalize(double *restrict Y, const double *restrict X,
               int n) {
  cilk_for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Terrible work efficiency!
$T_S/T_1 = 0.312/2600$
$\sim 1/8600$

Test: Random vector, $n = 64M$
Machine: Amazon AWS c4.8xlarge
Running time of serial code: $T_S = 0.312$ s
18-core running time: $T_{18} = 180.657$ s
1-core running time: $T_1 = 2600.287$ s

Effect of Compiling Cilk Code

Cilk code

```
void normalize(double *restrict Y,
               const double *restrict X, int n) {
  cilk_for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Cilk compiler

Helper function encodes the loop body.

Call into Cilk runtime library to execute a cilk_for loop.

C pseudo-code

```
            uble *restrict
               const double *restrict X, int n) {
  struct args_t args = { Y, X, n };
  __cilkrts_cilk_for(normalize_helper, args, 0, n);
}
void normalize_helper(struct args_t args, int i) {
  double *Y = args.Y;
  double *X = args.X;
  int n = args.n;
  Y[i] = X[i] / norm(X, n);
}
```

The compiler can't move norm out of the loop.

Tapir: Task-Parallel IR

Tapir embeds recursive fork-join parallelism into LLVM's IR.

Traditional Cilk compiler pipeline

`Cilk` → Clang → `LLVM IR` → LLVM optimizer → `LLVM IR` → LLVM code generator → `EXE`

OpenCilk compiler pipeline

`Cilk` → Clang → `Tapir` → LLVM optimizer → `Tapir` → LLVM code generator → `EXE`

Tapir adds three instructions to LLVM IR that encode recursive fork-join parallelism.

With few changes, LLVM existing optimizations work on parallel code.

Impact on LLVM

Compiler component	LLVM 6.0 (lines)	Tapir/LLVM (lines)
Core middle-end functionality	500,283	2,989
Base classes	62,488	0
Instructions	141,321	1,013
Memory behavior	18,907	536
Other analyses	84,348	17
Optimizations	193,219	1,423
Regression tests	3,482,802	5,745
Parallelism lowering	0	5,780
Parallel-tool support	0	3,341
Other	1,856,877	285
Total	5,839,962	18,140

Parallelize Normalize with Tapir

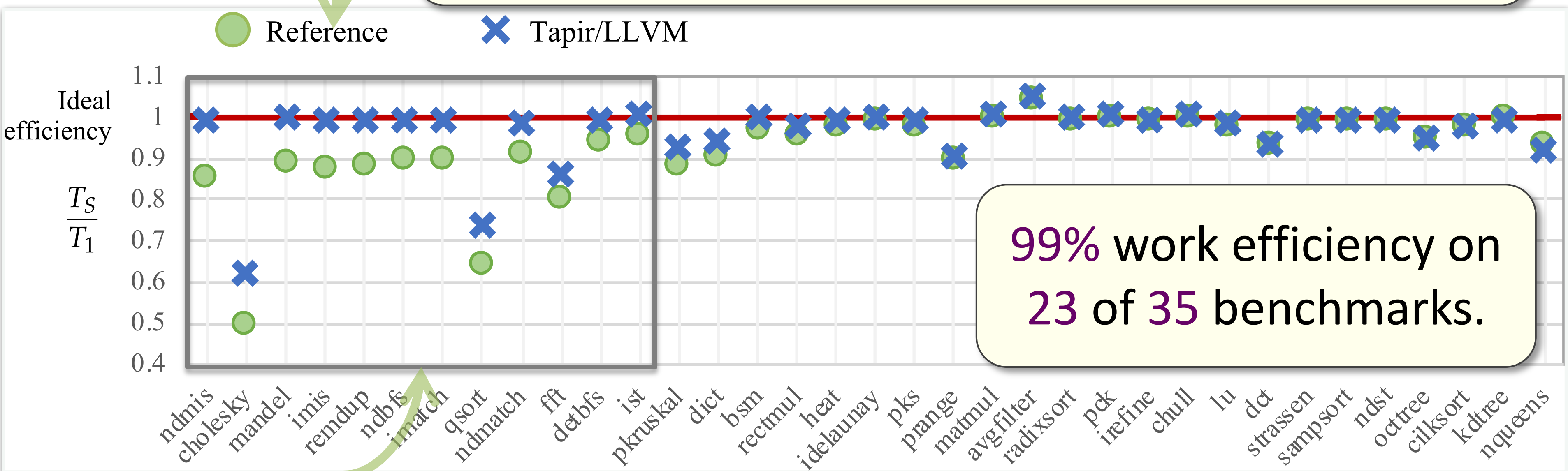
```
__attribute__((const)) double norm(const double *X, int n);

void normalize(double *restrict Y, const double *restrict X,
               int n) {
  cilk_for (int i = 0; i < n; ++i)
    Y[i] = X[i] / norm(X, n);
}
```

Great work efficiency:
$T_S/T_1 = 97\%$

Test: Random vector, $n=64M$
Machine: Amazon AWS c4.8xlarge
Running time of serial code: $T_S = 0.312$ s
1-core running time: $T_1 = 0.321$ s
18-core running time: $T_{18} = 0.081$ s

Work–Efficiency Improvement

Same as Tapir/LLVM, but the front-end handles parallel language constructs the traditional way.



Decreasing difference between Tapir/LLVM and Reference

Improved work efficiency by ≥ 5%.

OpenCilk Compiler Middle-End: Tapir

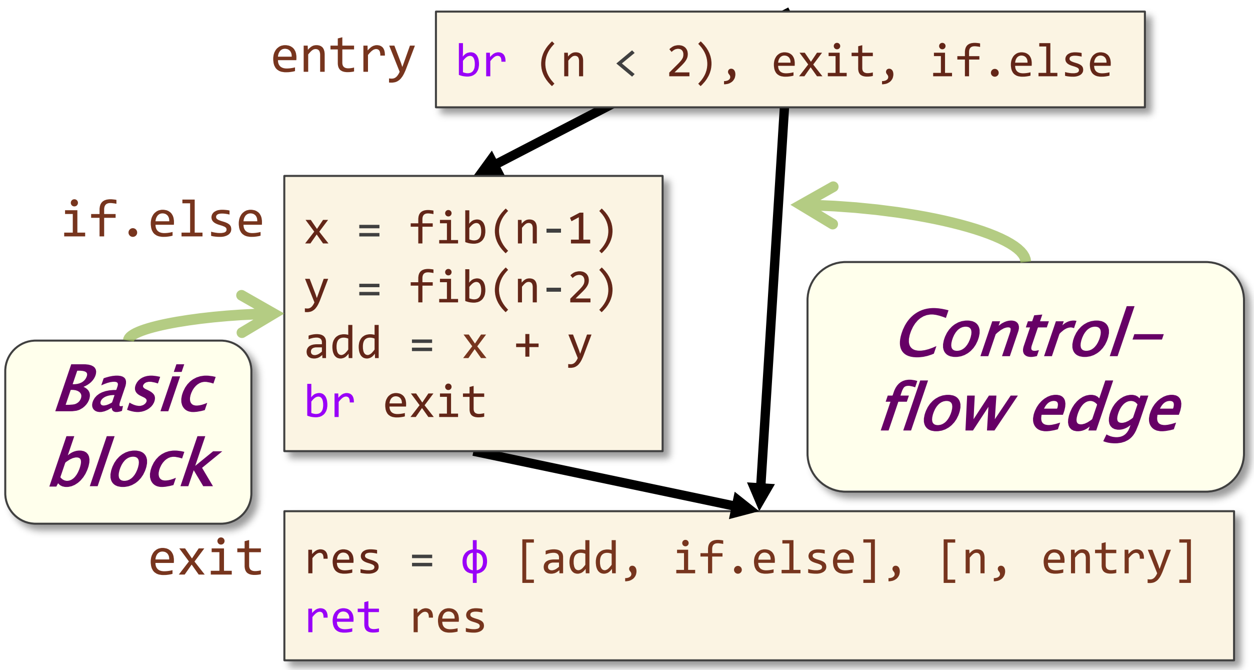
Background: LLVM IR

LLVM represents each function as a *control–flow graph (CFG)*.

C code

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

Control–flow graph (CFG)

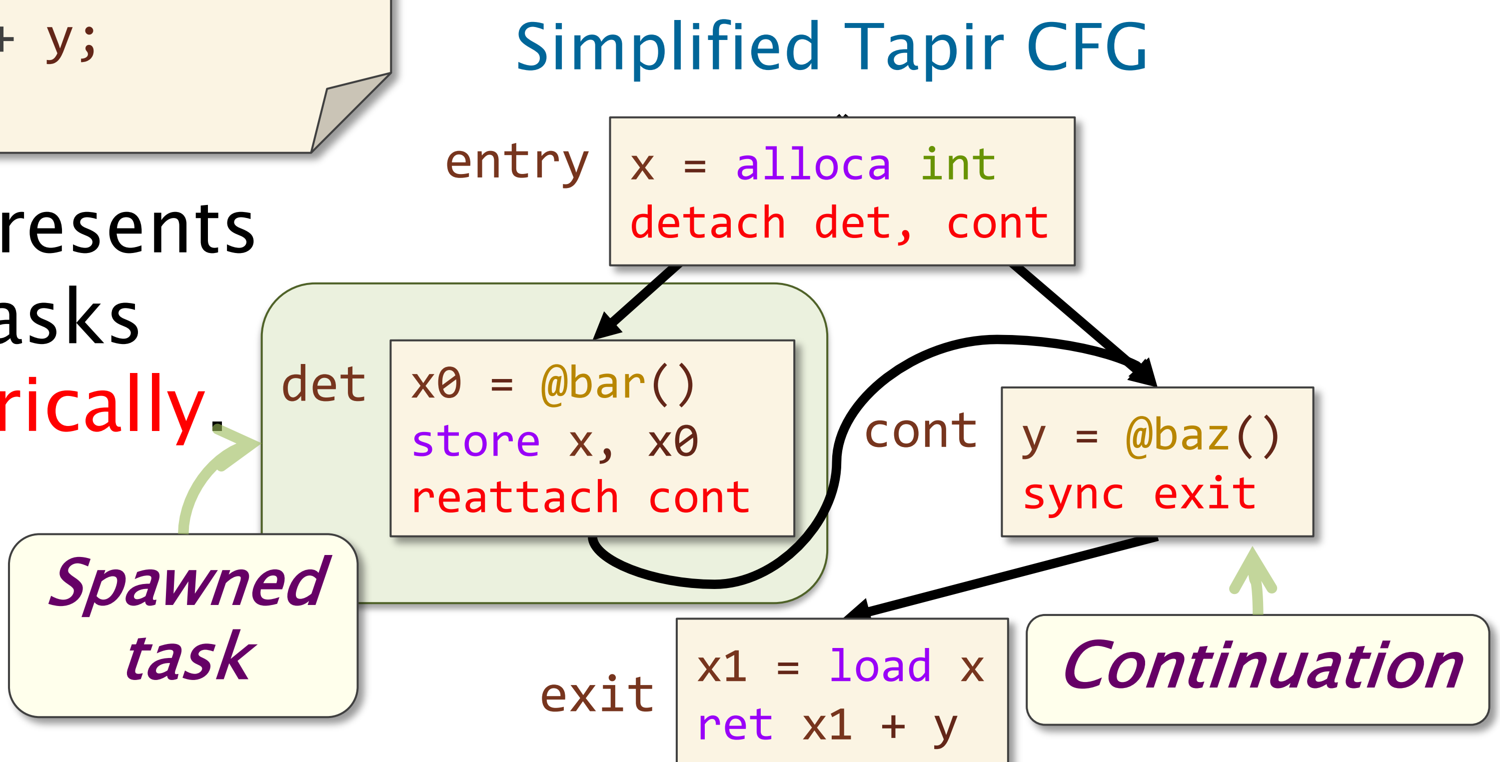


A Simplified Tapir CFG

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

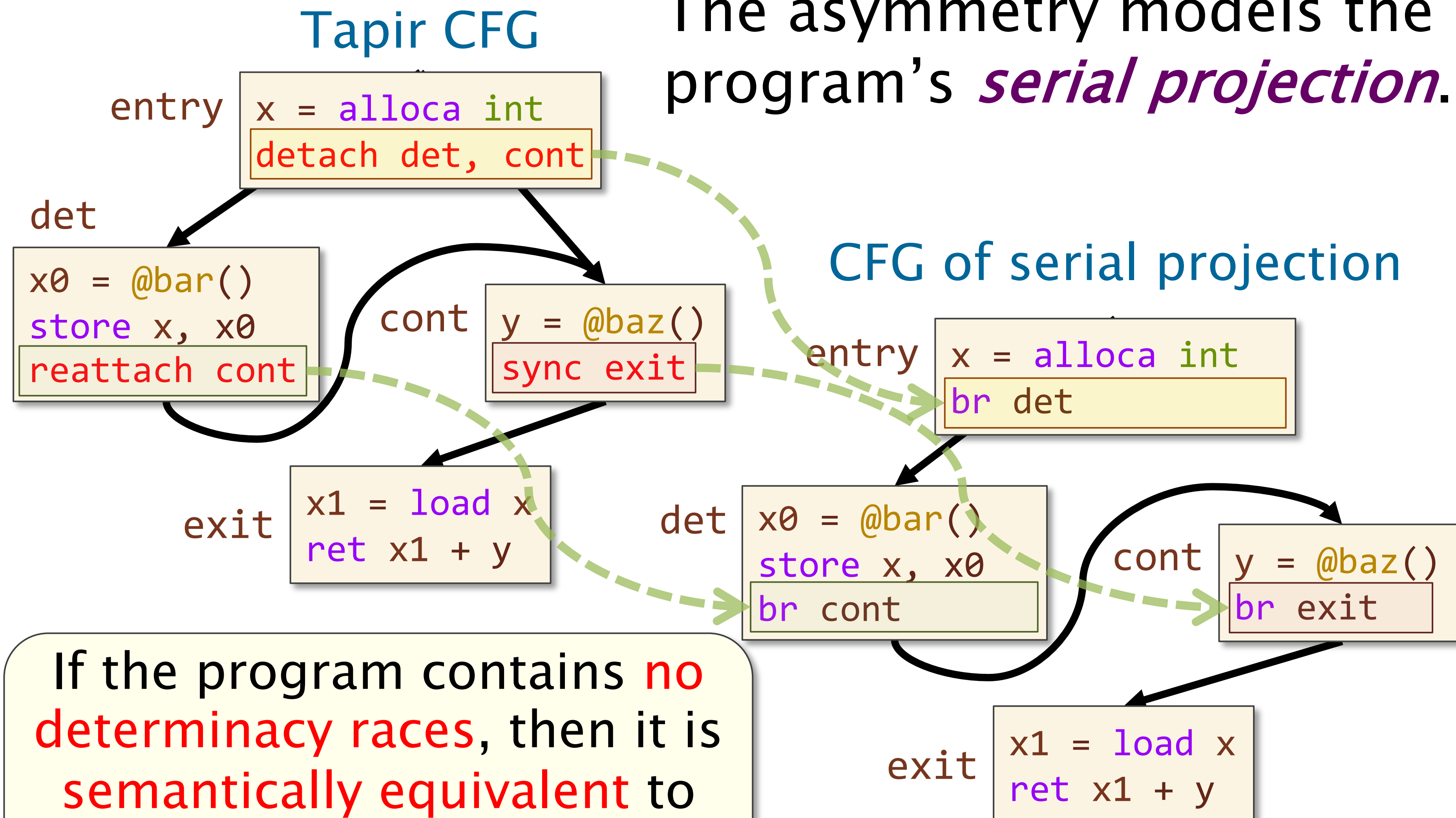
Tapir adds three constructs to LLVM's IR: **detach**, **reattach**, and **sync**.

Tapir represents parallel tasks **asymmetrically**.



Serial Projection

The asymmetry models the program's ***serial projection***.



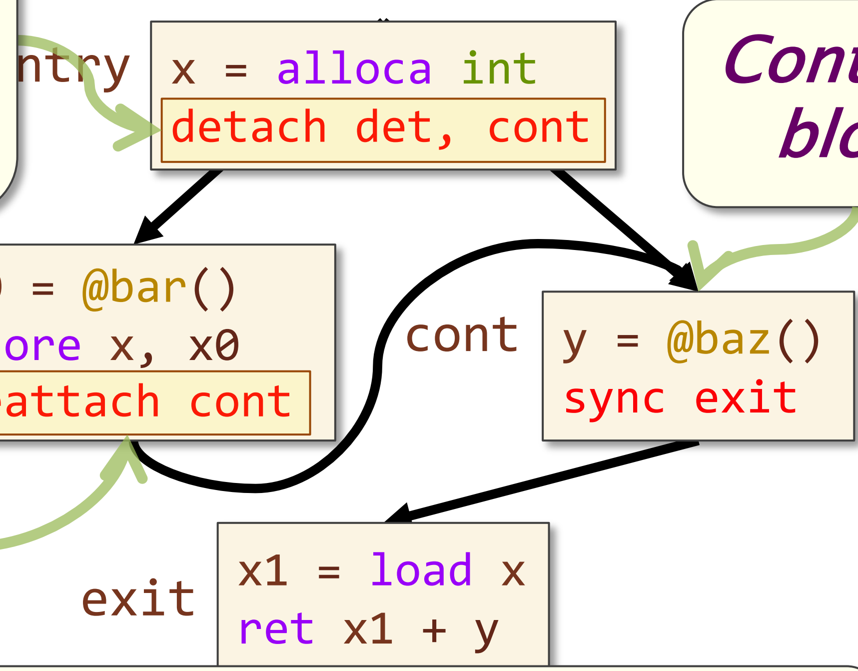
If the program contains no determinacy races, then it is semantically equivalent to its serial projection.

Detach and Reattach

The detach and reattach instructions denote the start and end of a spawned task.

A detach *spawns* a task starting at the detached block to run in parallel with the continue block.

Tapir CFG



Continue block

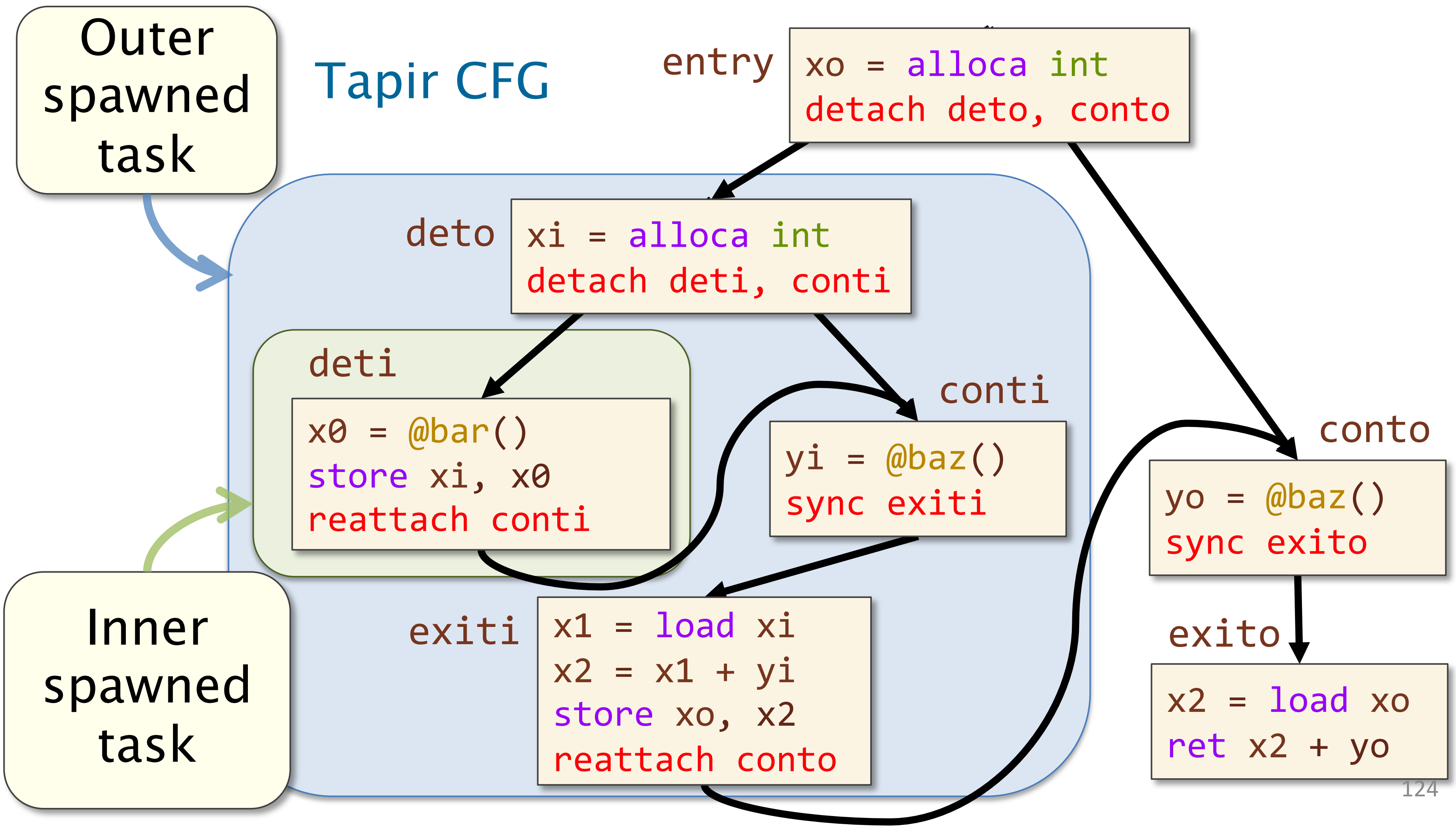
Detached block

A reattach terminates a spawned task.

INVARIANT: A reattach must identify the same continue block as its corresponding detach.

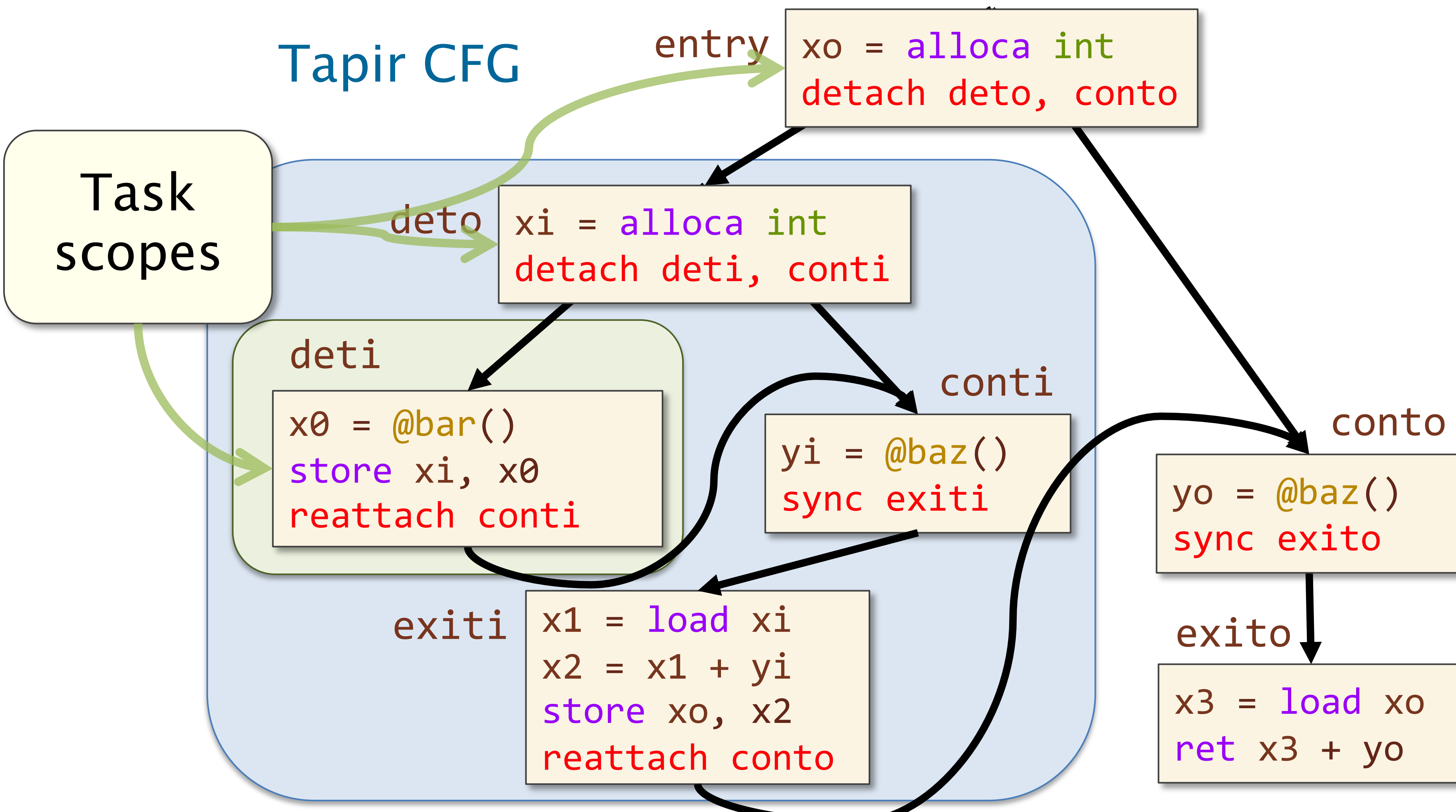
Nested Spawning in Tapir

Tapir supports nested spawning of tasks.



Task Scopes

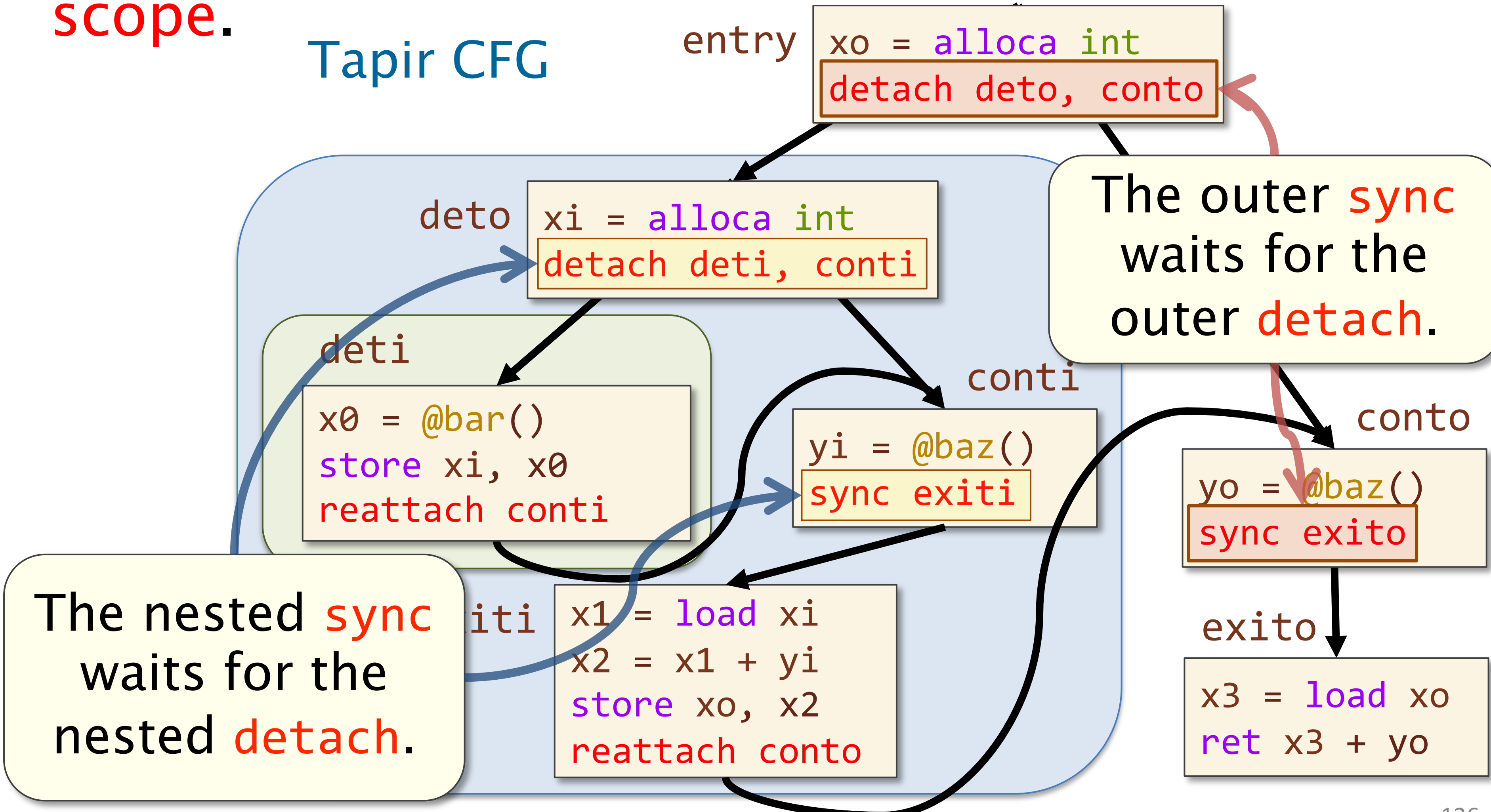
A ***task scope*** corresponds with a function or a spawned task therein.



Sync

The sync instruction syncs tasks within its task scope.

Tapir CFG



Problem: Selective Syncs

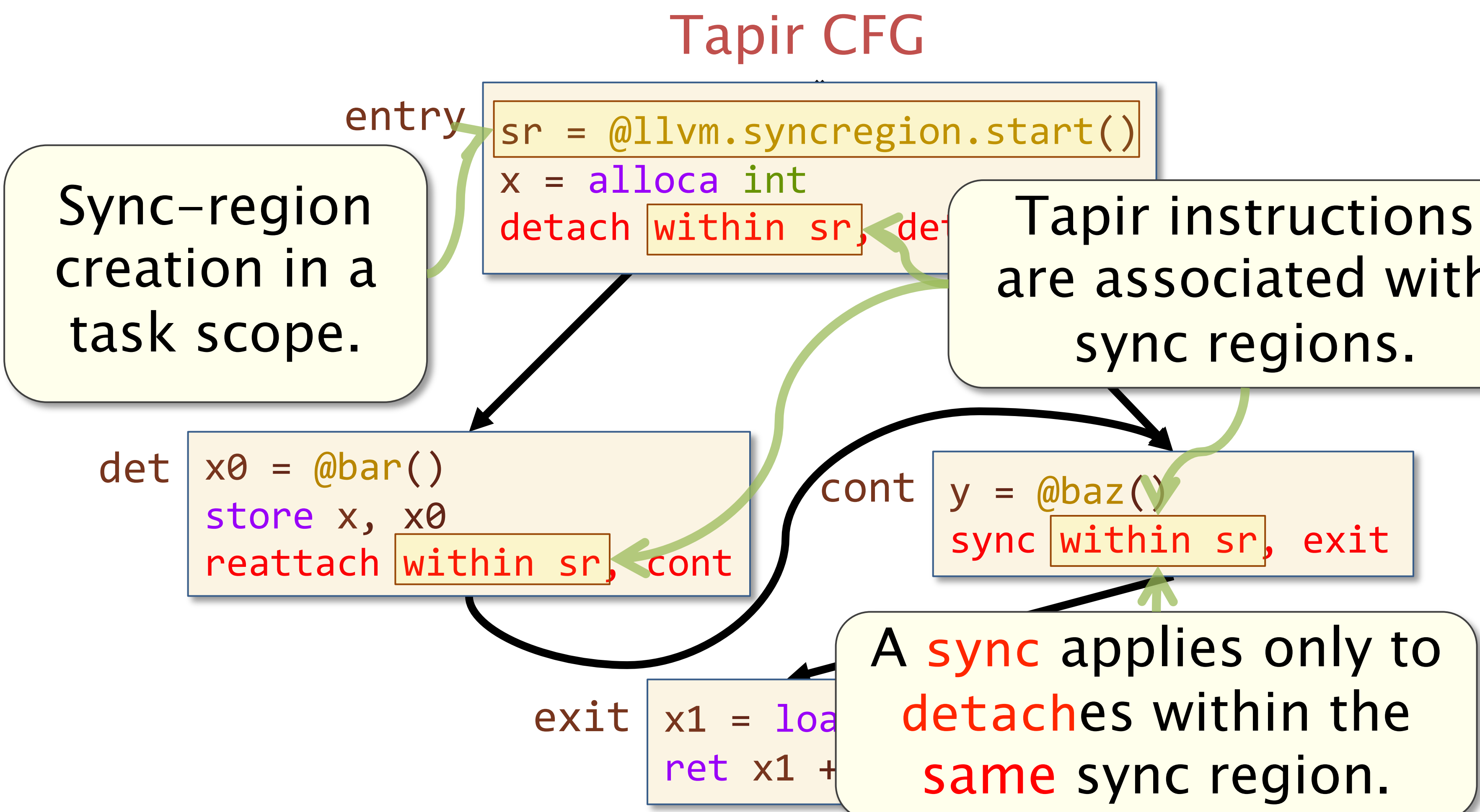
What if a sync instruction shouldn't apply to all spawned tasks within the task scope?

```
int foo(int n) {
  int x, y;
  x = cilk_spawn bar(n);
  cilk_for (int i = 0; i < n; ++i)
    loop_body(i);
  y = baz(n);
  cilk_sync;
  return x + y;
}
```

The implicit `cilk_sync` at the end of this loop should not synchronize the spawn of `bar()`.

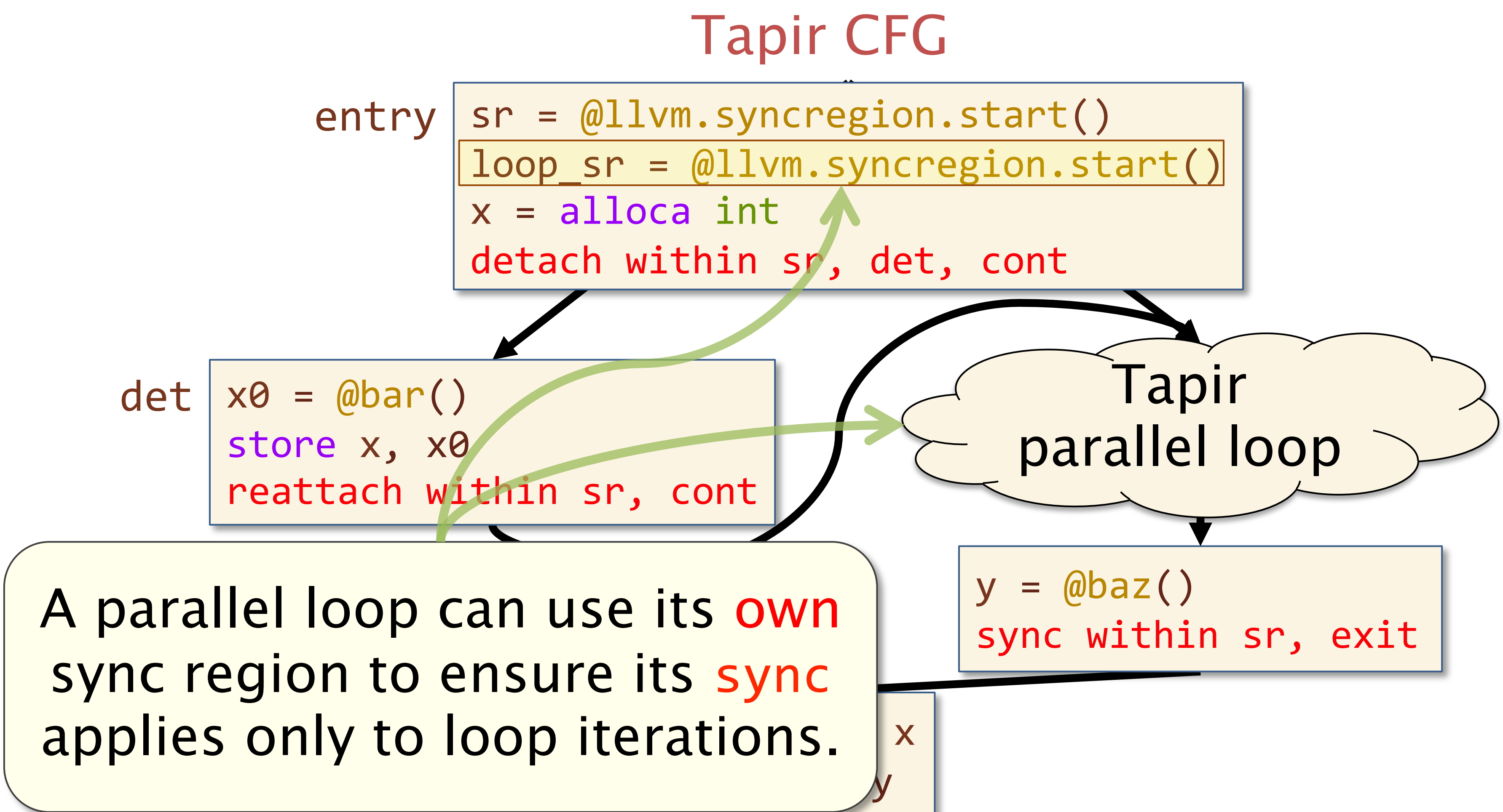
Sync Regions

Tapir's constructs also use a *sync region* to identify what spawned tasks a sync affects.



Differentiating Syncs

Different parallel language constructs can use different sync regions.



Hands-On: Kaleidoscope

In this hands-on, you will use OpenCilk to add `spawn` and `sync` expressions to a toy programming language, Kaleidoscope[1].

Kaleidoscope code

```
def binary : 1 (x y) y;
def fib(n)
  if (n < 2) then n
  else
    var x, y in
      x = fib(n-1) :
      y = fib(n-2) :
      (x + y);
```

Parallel Kaleidoscope code in `fib.k`

```
def binary : 1 (x y) y;
def fib(n)
  if (n < 2) then n
  else
    var x, y in
      (spawn x = fib(n-1)) :
      y = fib(n-2) :
      sync
      (x + y);
```

[1] https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html

Hands-On: Kaleidoscope

The code in `toy-spawn-sync.cpp` uses OpenCilk to implement a simple Parallel Kaleidoscope compiler, with the following components:

- A lexer and parser translate Kaleidoscope code into an ***abstract syntax tree (AST)***.
- Code-generator routines generate Tapir and LLVM IR from the AST. Current focus
- The driver uses LLVM's JIT interface to optimize the Tapir intermediate representation, generate machine code, and run the executable.

Hands-On: Kaleidoscope (~20 min)

HANDS-ON: Complete the code-generator routines to produce Tapir for `spawn` and `sync`.

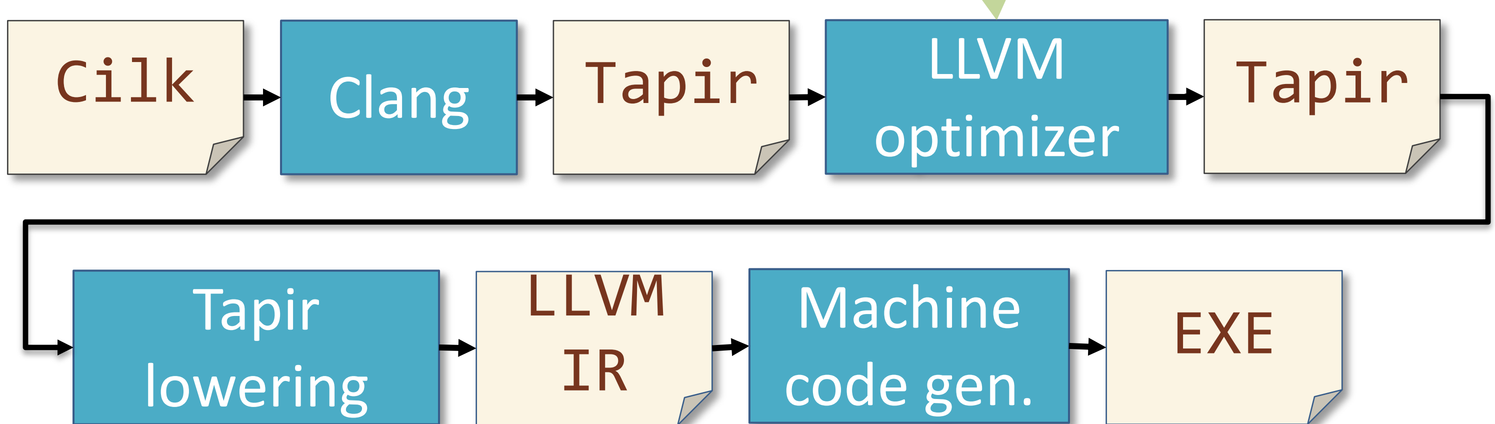
- Follow the instructions, marked `HANDS-ON`, in `toy-spawn-sync.cpp` to finish implementing `SpawnExprAST::codegen()` and `SyncExprAST::codegen()`.
- In the Docker container, test your code on different worker counts:

```
$ cd /tutorial
$ make toy-spawn-sync
$ ./toy-spawn-sync < fib.k
$ CILK_NWORKERS=1 ./toy-spawn-sync < fib.k
```

May take a couple of seconds.

Compiler Pipeline with Tapir

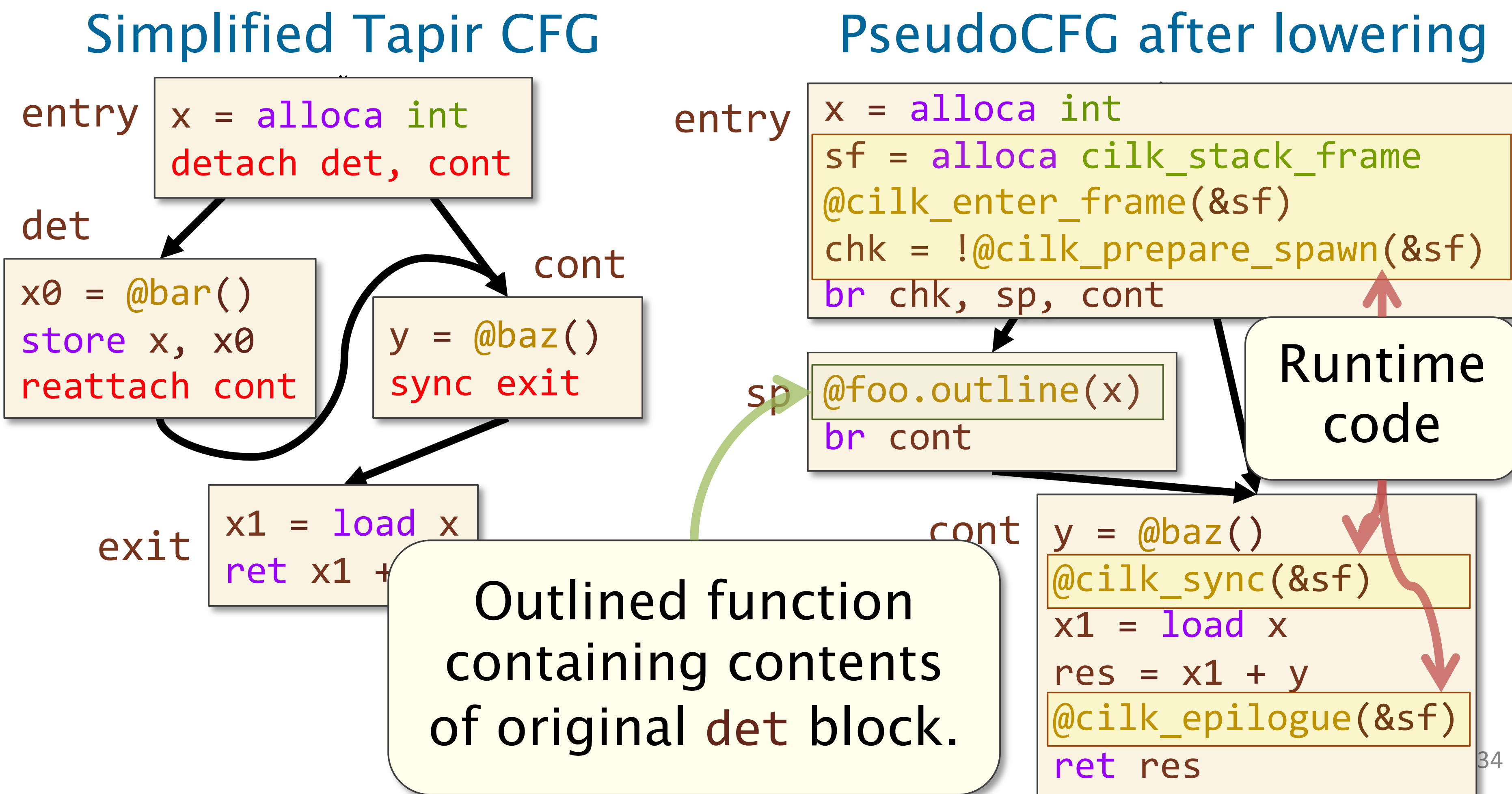
Includes traditional LLVM optimizations and new Tapir-specific optimizations, such as parallel-loop stripmining.



Transforms Tapir instructions into ordinary LLVM IR, based on a *Tapir target*.

Tapir Lowering

Tapir lowering **outlines** spawned tasks into separate functions and **inserts** runtime code.



BREAK

Productivity Tools: Cilksan and Cilkscale

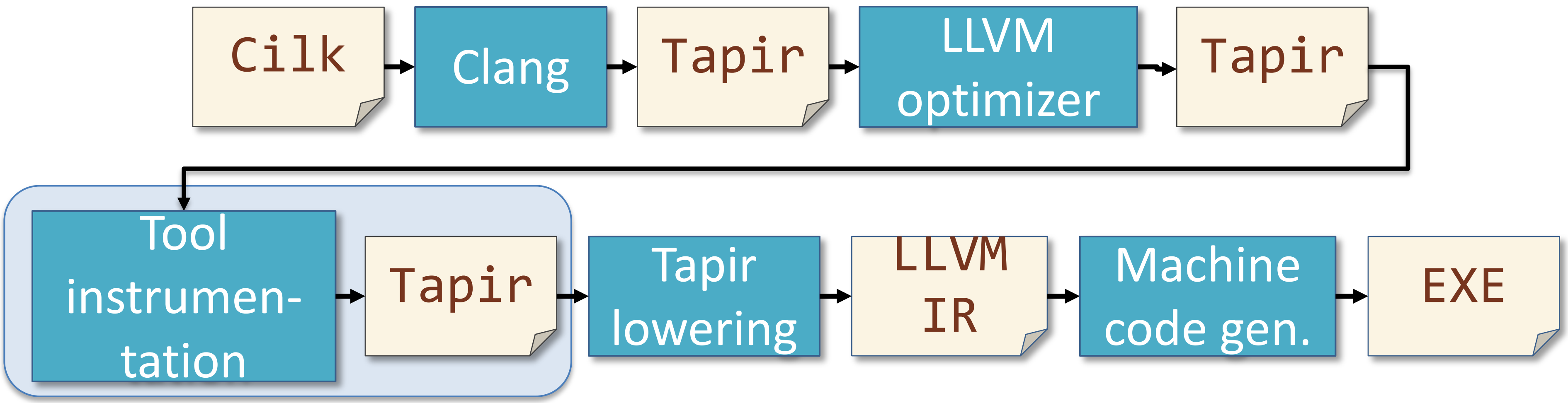
Cilksan and Cilkscale

OpenCilk's productivity tools, Cilksan and Cilkscale, use *compiler instrumentation*.

- Each tool is implemented as a *library*, which is linked to the executable.
- Each tool has a corresponding *compiler pass* in the OpenCilk compiler that inserts instrumentation in the form of calls into the tool's library.

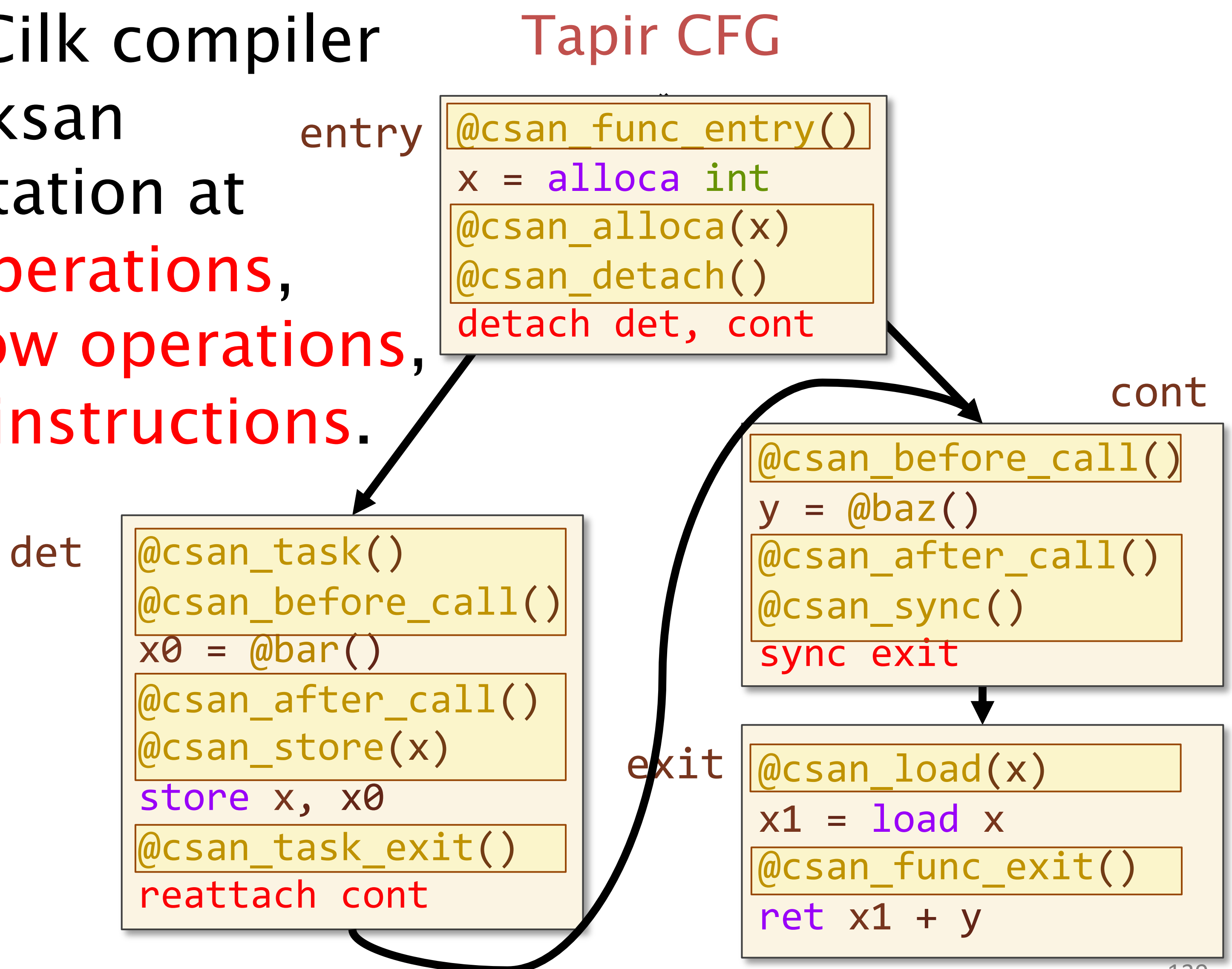
Tool Compiler Pass

The OpenCilk compiler inserts instrumentation just before Tapir lowering.



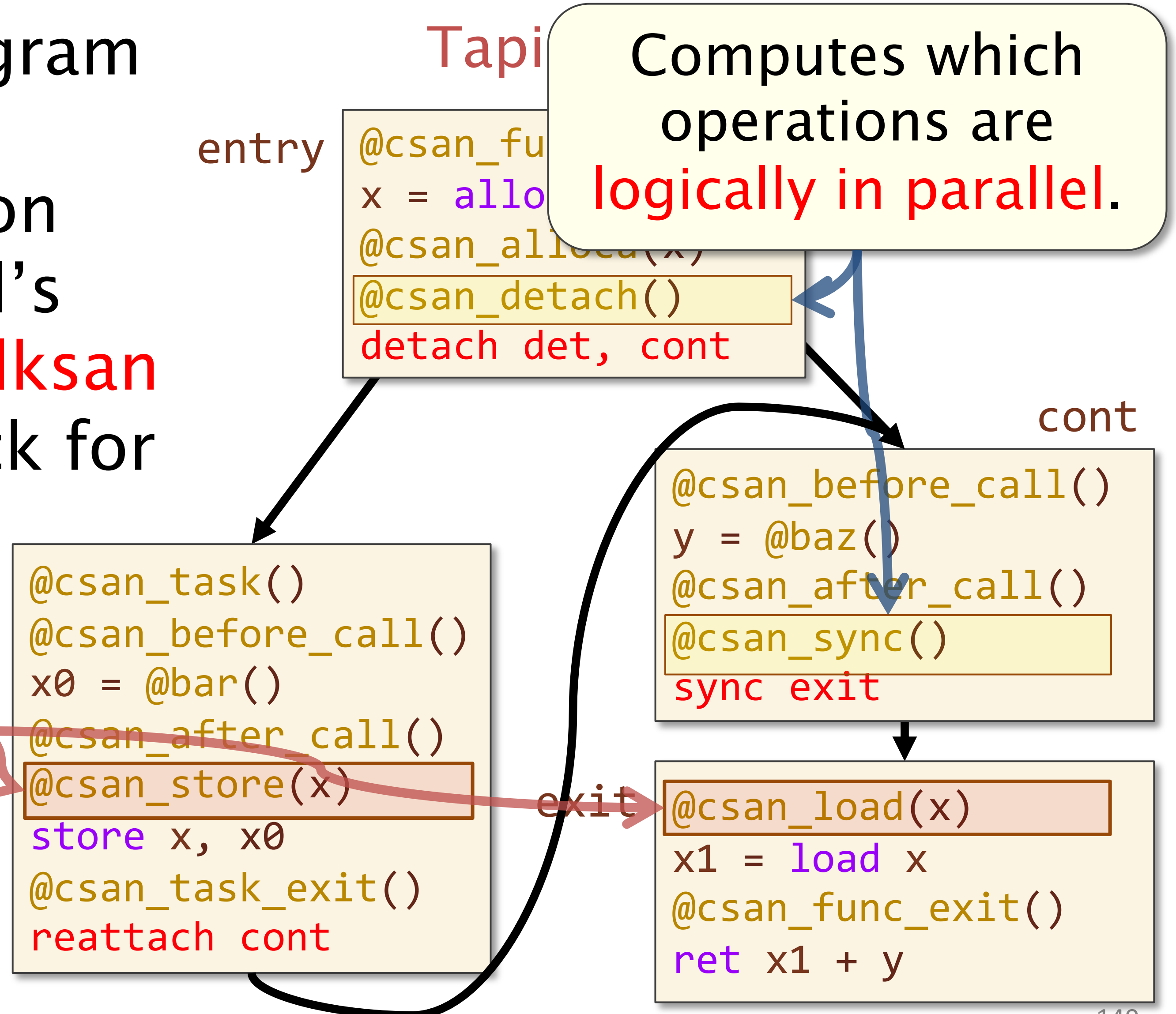
Example: Cilksan Instrumentation

The OpenCilk compiler inserts Cilksan instrumentation at memory operations, control-flow operations, and Tapir instructions.



Driving the Cilksan Library

When the program is run, the instrumentation drives the tool's logic in the Cilksan library to check for races.

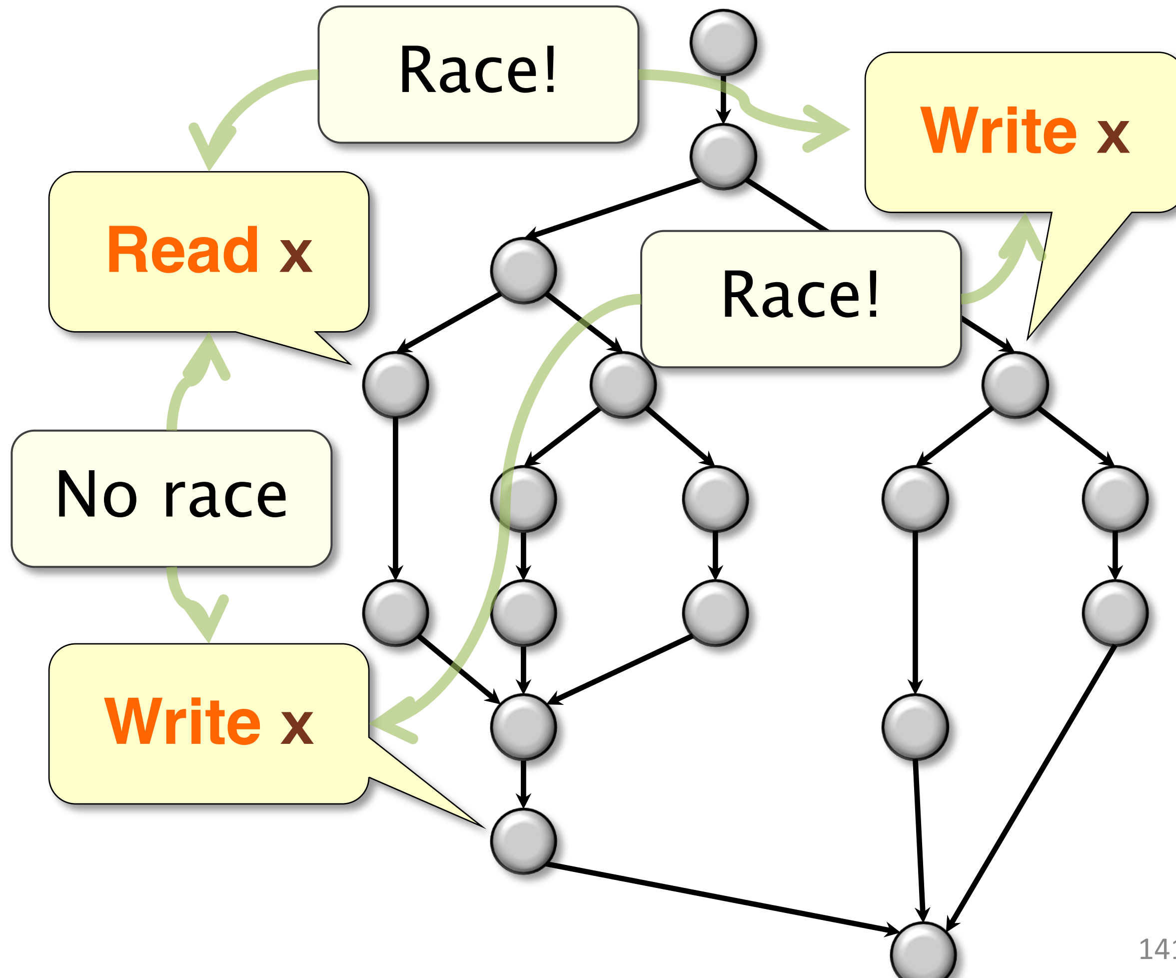


Records memory reads and writes.

How Cilksan Works (Intuition)

Intuitively, Cilksan maintains the computation's trace dag to find parallel memory accesses.

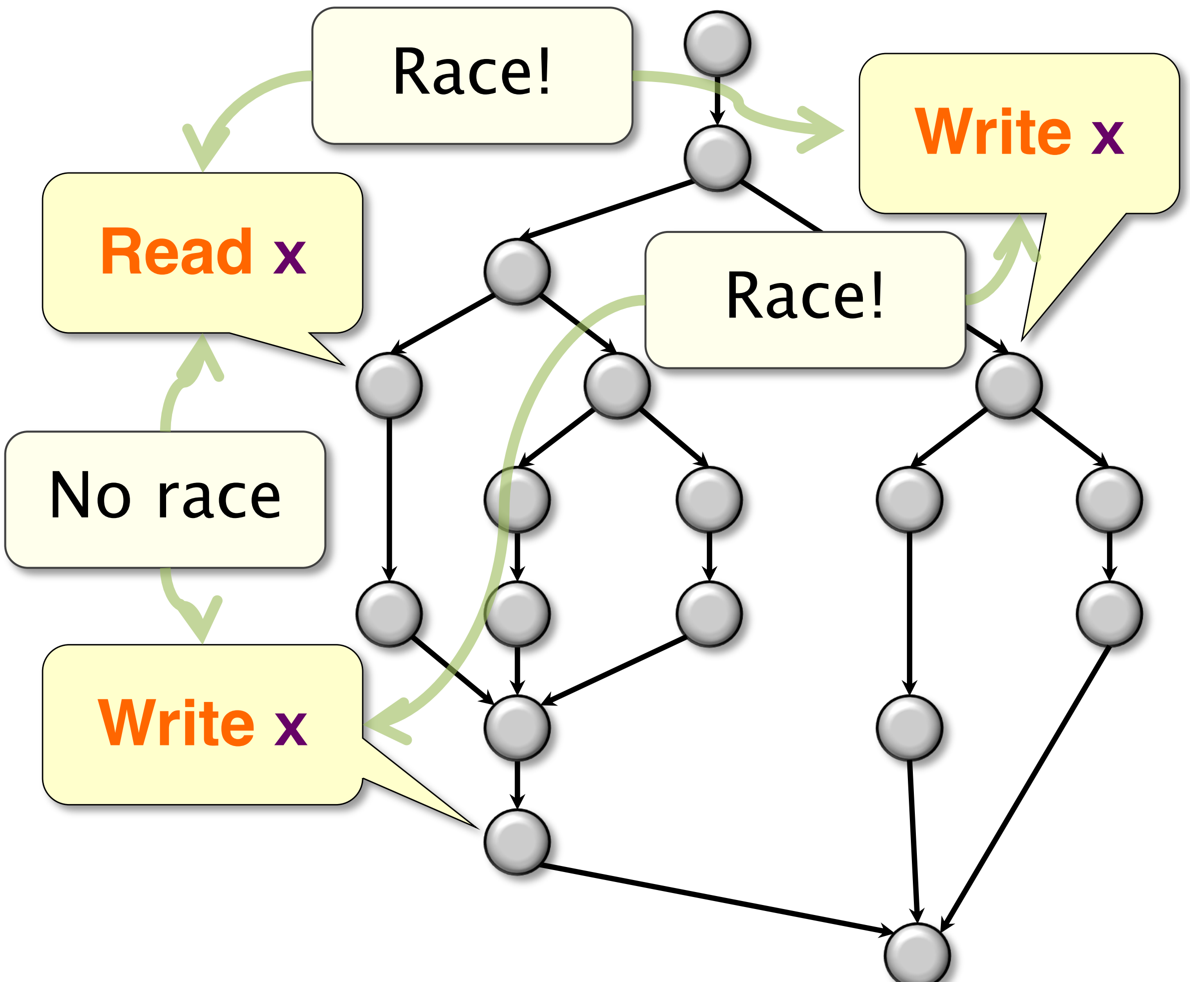
Because race-detection is based on the dag, Cilksan's race-detection is guaranteed, regardless of scheduling.



How Cilksan Works

Storing the trace dag is inefficient in practice.

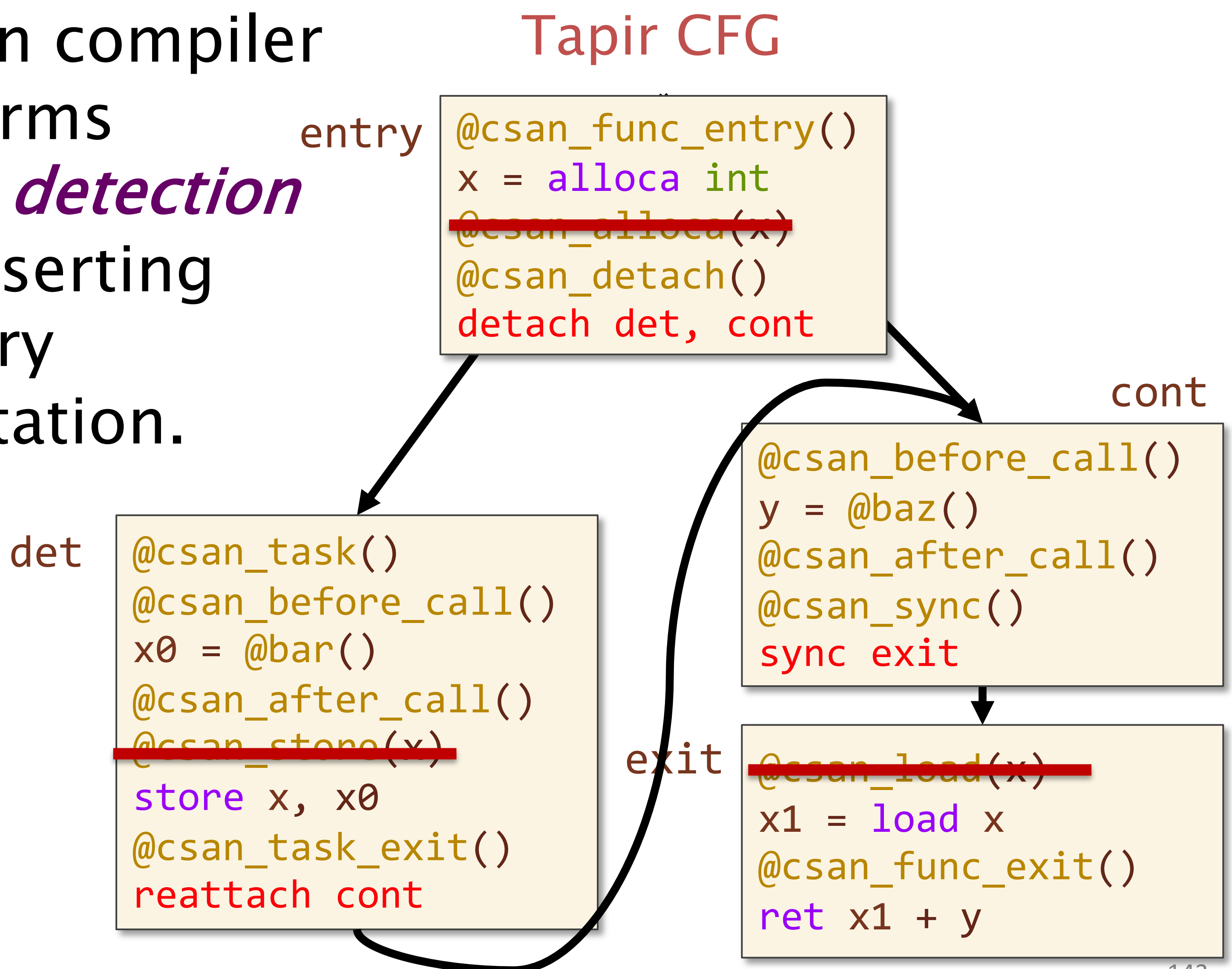
Instead, Cilksan implements the *SP-bags algorithm*[FL99] to achieve the same effect.



[FL99] Feng, Leiserson. Efficient Detection of Determinacy Races in Cilk Programs. Theory of Computing Systems, 1999.

Optimizing Cilksan Instrumentation

The Cilksan compiler pass performs ***static race detection*** to avoid inserting unnecessary instrumentation.



Hands-On: Kaleidoscope parfor

The toy-parfor.cpp code adds parfor, a parallel-for construct, to Kaleidoscope.

Kaleidoscope parallel loop in fib-loop.k

```
def fibloop(n)
  parfor i = 0, i < n in
    fib(i);
```

But the construct has a bug in it that results in a determinacy race!

Hands-On: Kaleidoscope parfor

Just like `toy-spawn-sync.cpp`, the code in `toy-parfor.cpp` uses OpenCilk to implement a simple Parallel Kaleidoscope compiler:

- A lexer and parser translate Kaleidoscope code into an ***abstract syntax tree (AST)***.
- Code-generator routines generate Tapir and LLVM IR from the AST.
- The driver uses LLVM's JIT interface to optimize the Tapir intermediate representation, generate machine code, and run the executable.

Current focus

Hands-On: Kaleidoscope `parfor` (~20 min)

HANDS-ON: Use Cilksan to identify the race in the `parfor` implementation.

- Follow the instructions, marked `HANDS-ON`, in `toy-parfor.cpp` (in `FunctionAST::codegen()` and `InitializeModuleAndPassManager()`) to enable the use of Cilksan.
- In the Docker container, run the following to observe the race in `parfor`:

```
$ cd /tutorial
$ make toy-parfor
$ ./toy-parfor -O0 --run-cilksan < fib-loop.k
```

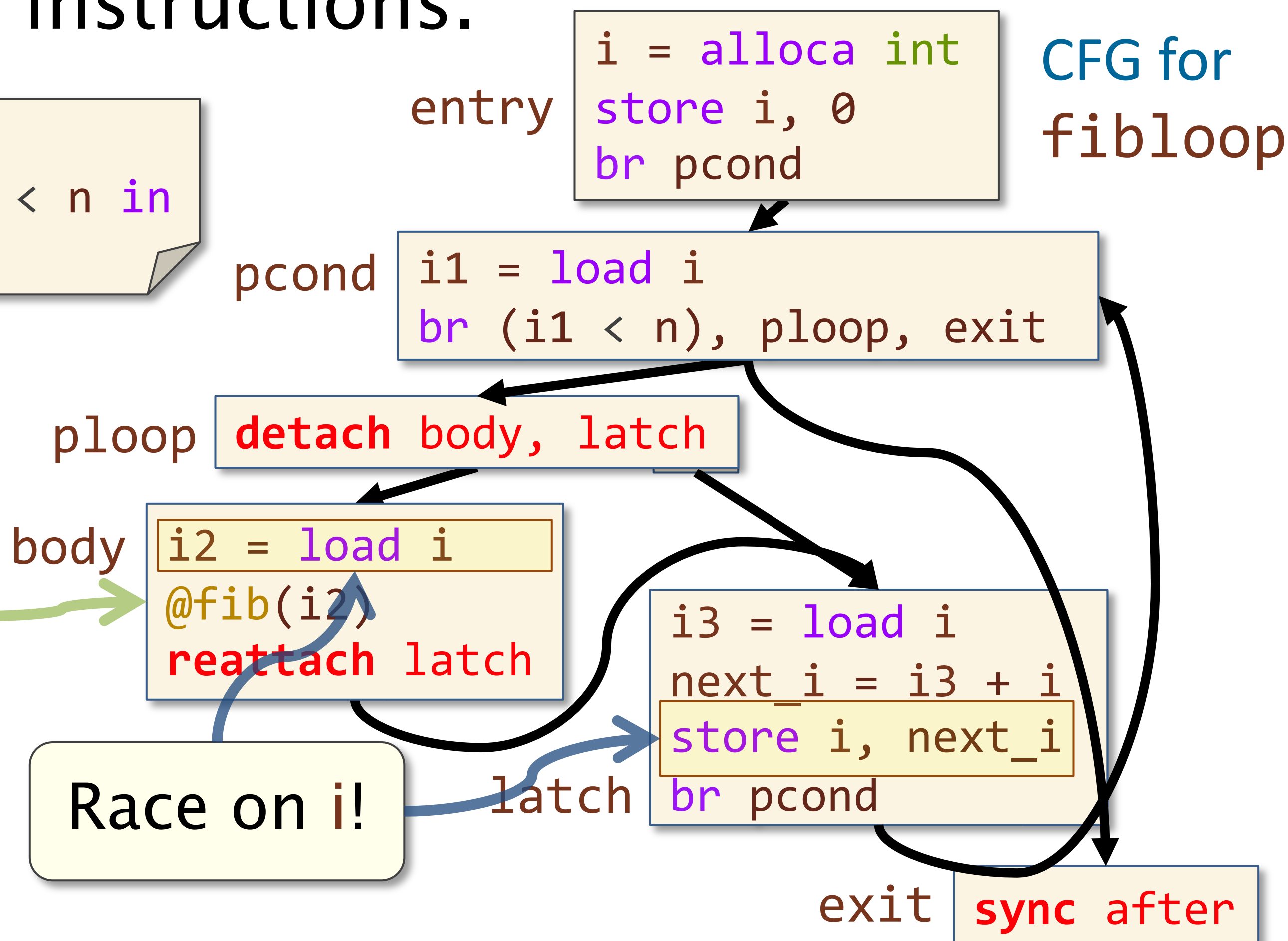
- OPTIONAL, HARD: Fix the race.

Parallel Loops in Tapir

The parfor implementation was made by copying the implementation of for and then adding Tapir instructions.

```
def fibloop(n)
  parfor i = 0, i < n in
    fib(i);
```

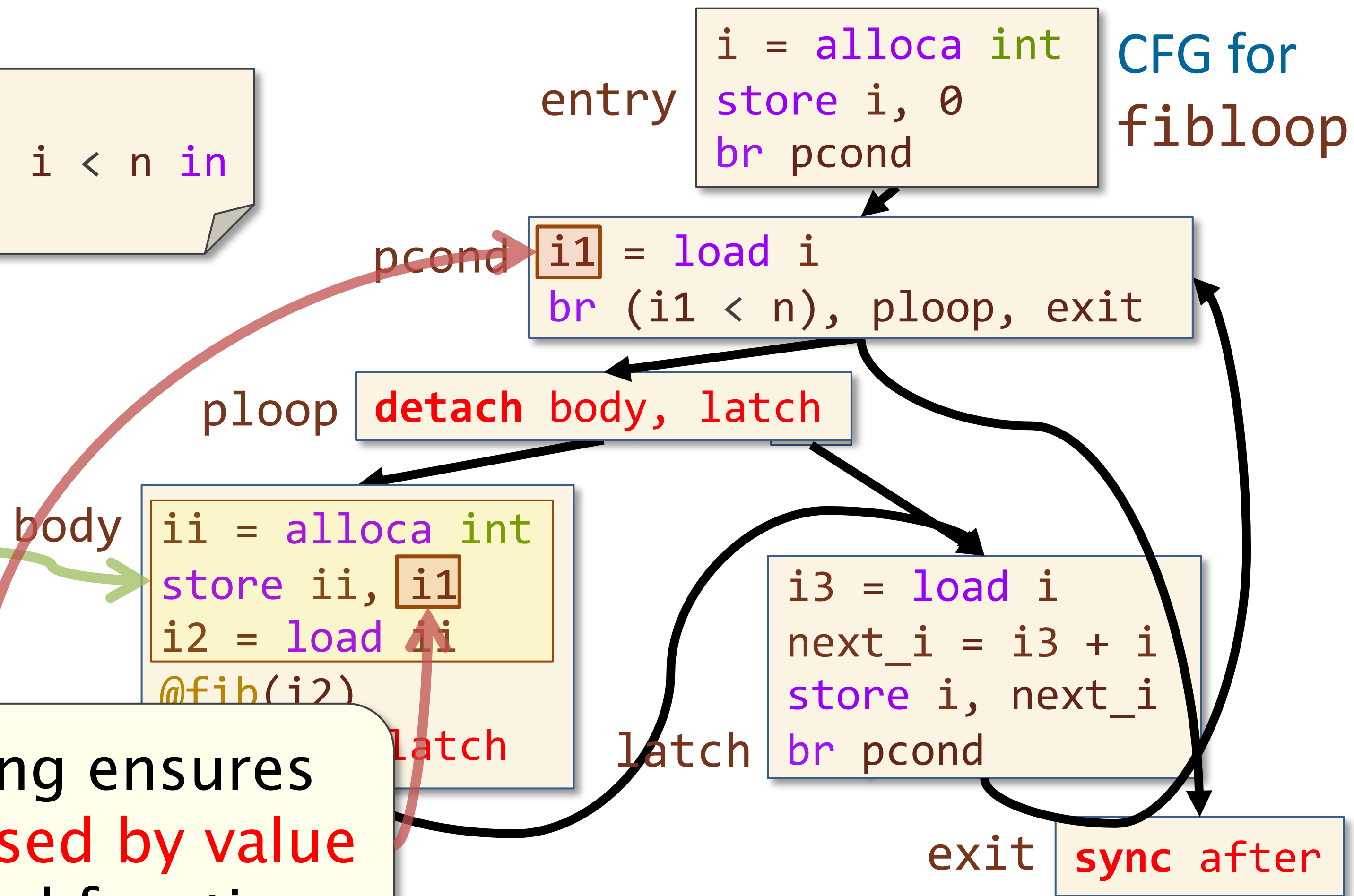
NOTE: Loop body is spawned.



Fixing the Race

Here is one way to fix the race:

```
def fibloop(n)
  parfor i = 0, i < n in
    fib(i);
```



Modified code

Tapir lowering ensures that i1 is passed by value to the outlined function.

Lowering Parallel Loops in Tapir

During Tapir lowering, Tapir's *LoopSpawning pass* converts parallel loops to spawns and syncs using recursive divide-and-conquer.

- Tapir loops are first canonicalized using standard LLVM loop transformations.
- The LoopSpawning pass outlines each* parallel loop into a separate function that implements the parallel divide-and-conquer recursion using Tapir.
- Those generated Tapir instructions are later lowered to runtime calls.

*To prevent compiler misoptimization, only marked loops are transformed.

Thank you

www.opencilk.org
contact@opencilk.org

Support Acknowledgments

- **National Science Foundation:** OpenCilk development is supported in part by the National Science Foundation under Grant No. CNS-1925609. Any opinions, findings, and conclusions or recommendations expressed in this tutorial are those of the presenters and do not necessarily reflect the views of the National Science Foundation.
- **United States Air Force Research Laboratory:** OpenCilk development is supported in part by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this tutorial are those of the presenters and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute content for Government purposes notwithstanding any copyright notation herein.