Software Performance Engineering

LECTURE 8
Races and Parallelism

Xuhao Chen

October 9, 2025

Nested Parallelism in Cilk

```
int64_t fib(int64_t n) {
  if (n < 2)
    return n;
  int64_t x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

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

Control cannot exit this context until all spawned children have returned

- Cilk keywords grant permission for parallel execution
- They do not command parallel execution

Loop Parallelism in Cilk

Example:
In-place **matrix transpose**

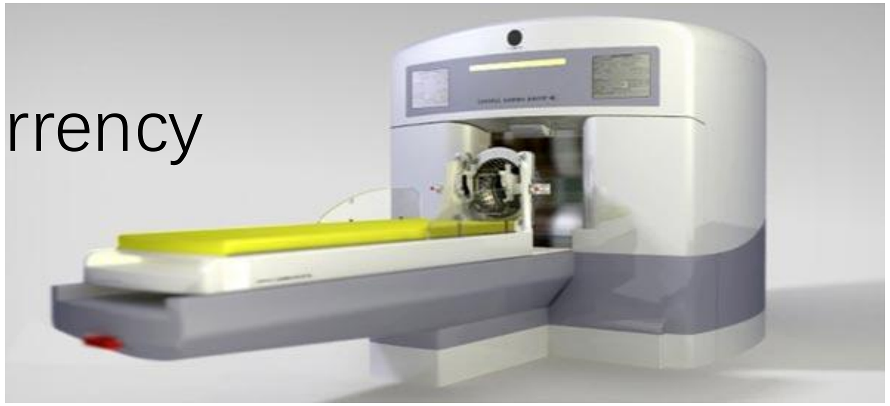
$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \Rightarrow A^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots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

The iterations of a `cilk_for` loop execute in parallel

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

DETERMINACY RACES

Race Conditions



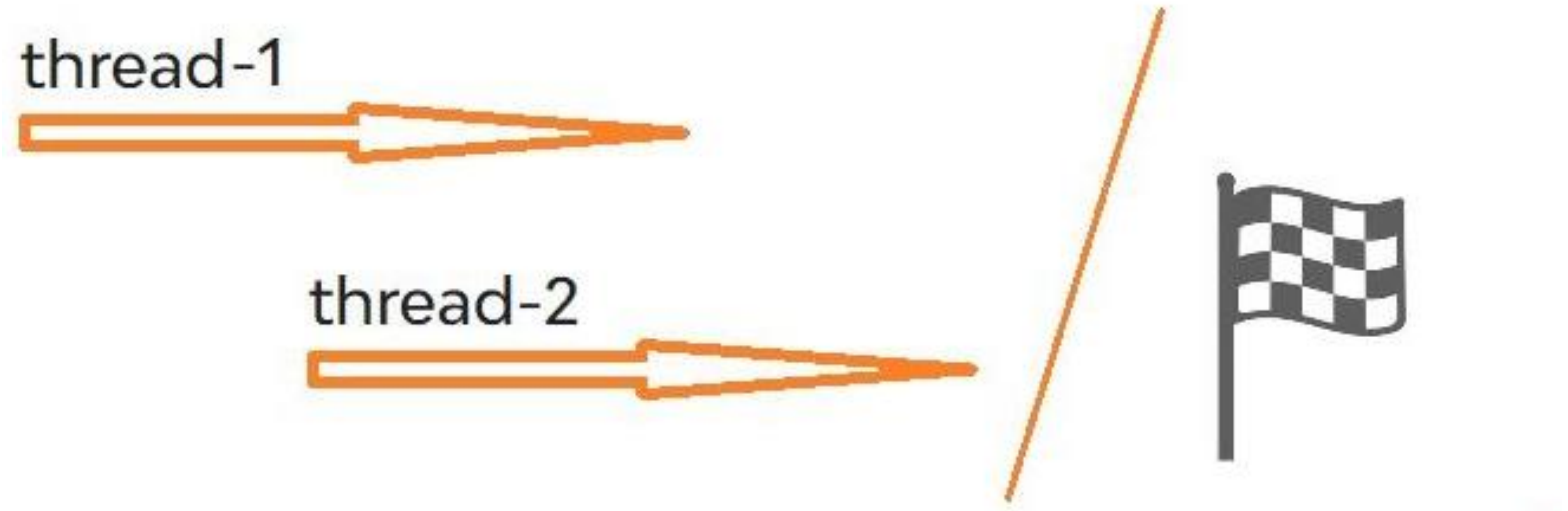
- **Race conditions** are the bane of concurrency
- Famous race bugs include
  - Therac-25 radiation therapy machine — killed 3 people and seriously injured many
  - Northeast Blackout of 2003 — left 50 million people without power

Race bugs are notoriously difficult to discover by conventional testing!

Race Condition

- Arise when multiple code paths executing at the same time

- Multiple code paths take a different amount of time than expected
  - they can finish in a different order than expected



Race Condition

- Arise when multiple code paths executing at the same time
- Multiple code paths take a different amount of time than expected
  - they can finish in a different order than expected
- A **data race** is a type of race condition
  - A C/C++ program containing a data race has undefined behavior
- Hard to reproduce and debug as the result is **nondeterministic**

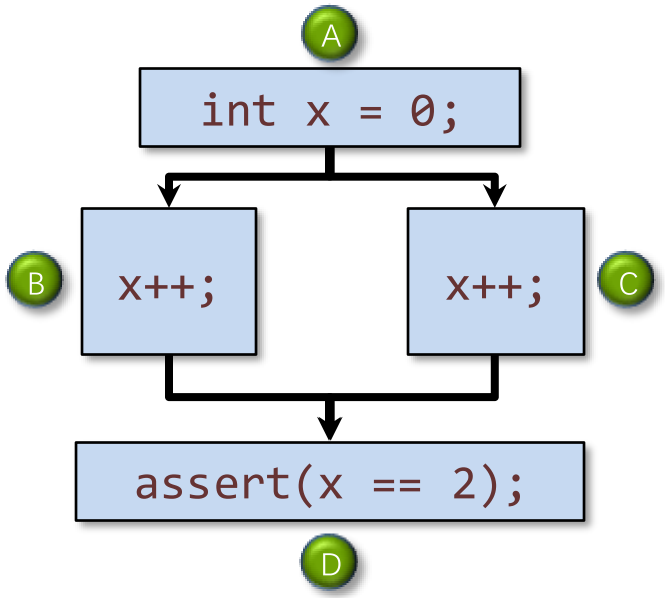
Determinacy Races

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.

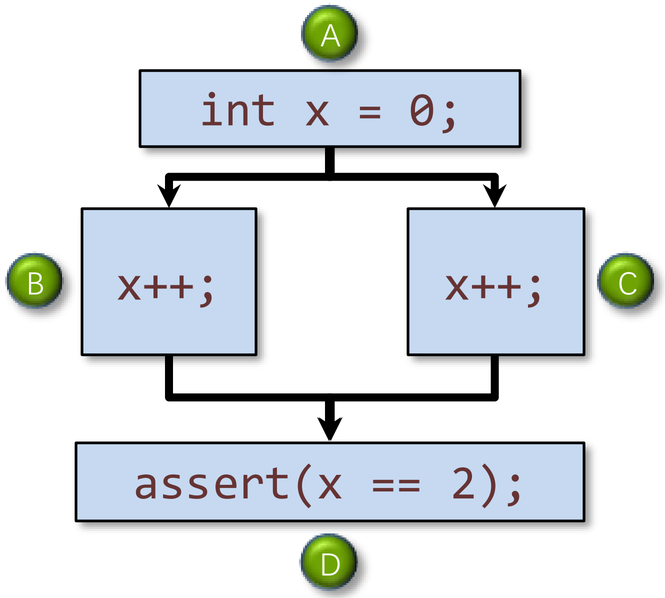
Example

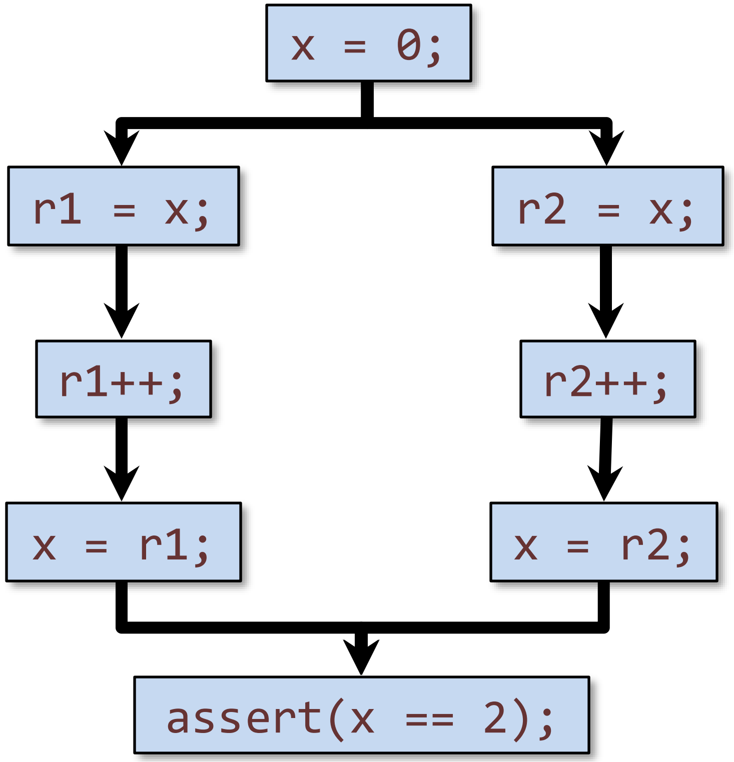
```
int x = 0;
cilk_for (int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

Trace



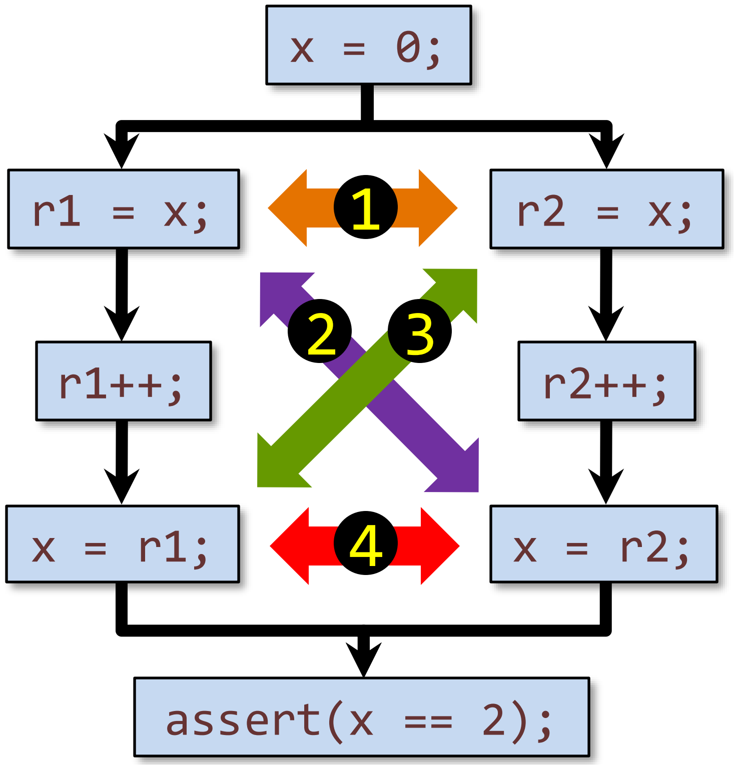
A Closer Look





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.

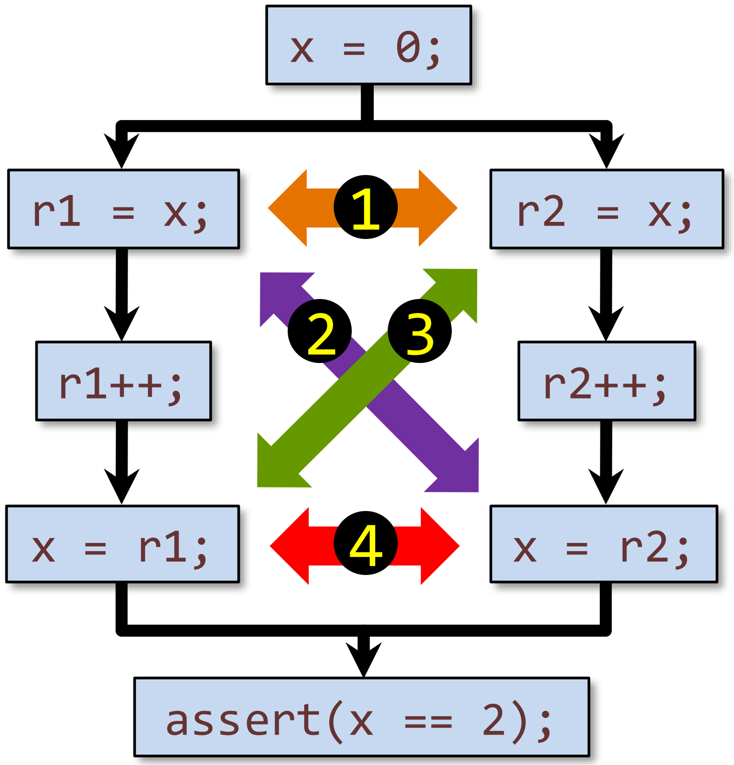


Which of these four pairs of instructions race?

a. 1
b. 2
c. 3
d. 4
e. 1 2
f. 2 3
g. 3 4
h. 1 2 3
i. 2 3 4
j. None of the above.

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.

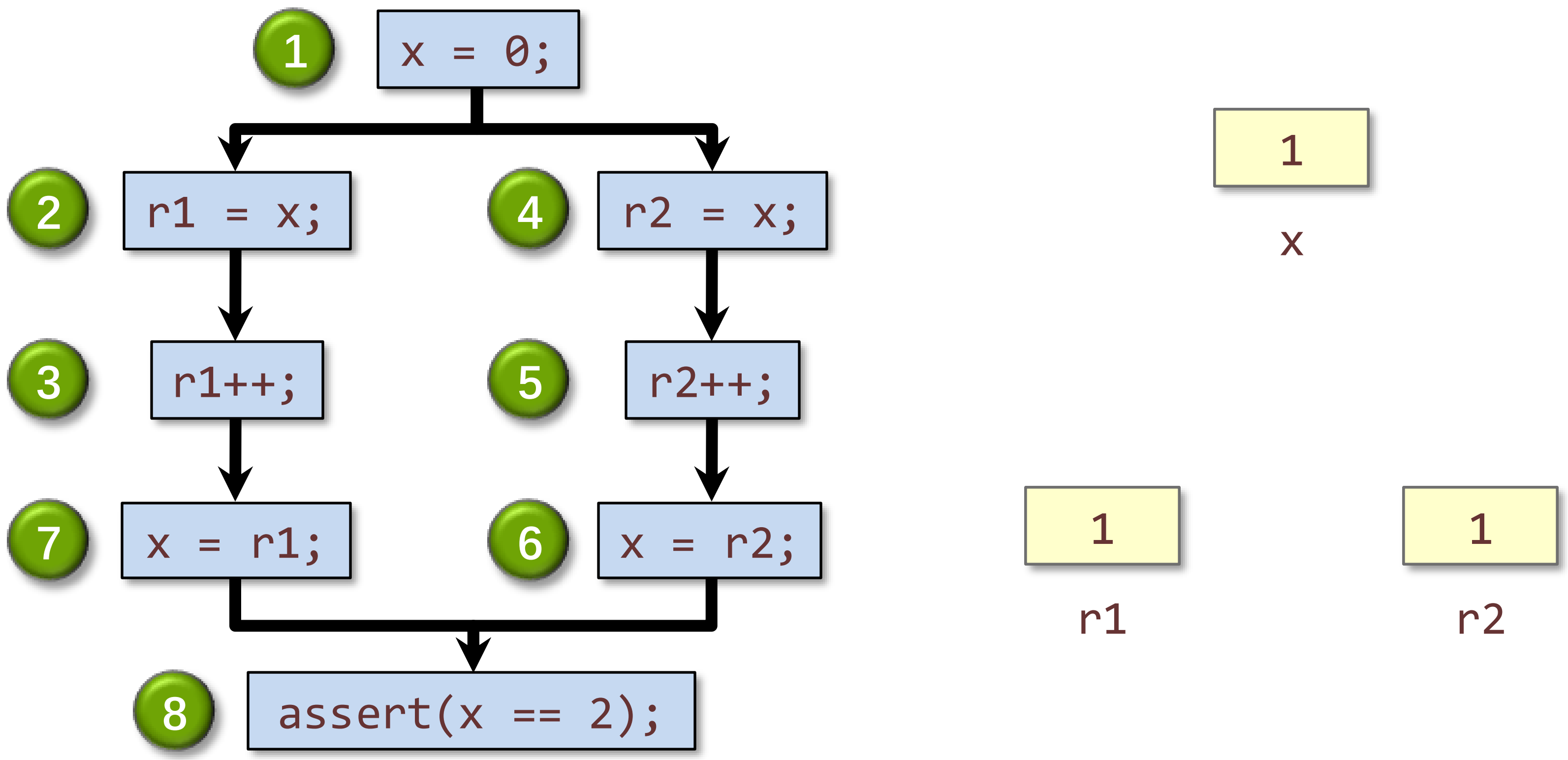


Which of these four pairs of instructions race?

a. 1
b. 2
c. 3
d. 4
e. 1 2
f. 2 3
g. 3 4
h. 1 2 3
i. 2 3 4
j. None of the above.

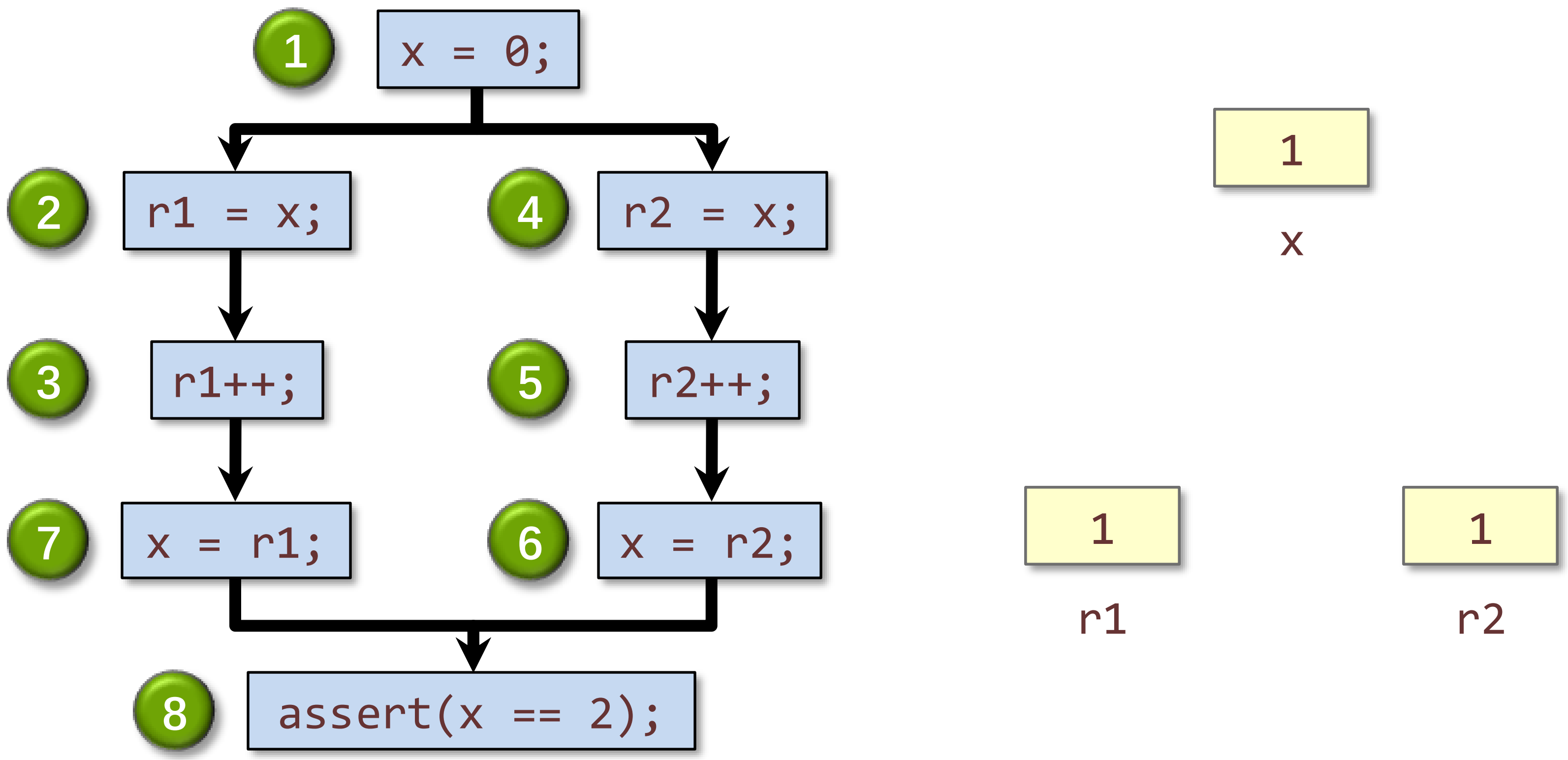
Race Bugs

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.



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 $A \parallel B$ (A is parallel to B).

A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

Two sections of code are independent if they have no determinacy races between them.

Avoiding Races

- Iterations of a `cilk_for` should be independent.

- After a `cilk_spawn`, the code executed by the spawned task should be independent of the subsequent code executed by the parent and any tasks that the parent spawns or calls, until the `cilk_scope` block is exited
  - arguments to a spawned function are evaluated in the parent before the spawn occurs

- Machine word size matters. Watch out for races in packed data structures:

```
struct {
  char a;
  char b;
} x;
```

Ex. Updating `x.a` and `x.b` in parallel may cause a race! Nasty, because it may depend on the compiler optimization level. (Safe on x86-64)

Cilksan Race Detector

- Compile with `-fsanitize=cilk` to produce a Cilksan-instrumented program
- If there is a determinacy race on a given input, Cilksan guarantees to find it
- 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 instrumented, or you'll miss some bugs
- Cilksan is your best friend.

Race Example: Queens

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

OpenCilk Cilksan Execution

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*      Read 43ef18 nqueens ./nqueens.c:87:3
|         `-to variable a (declared at nqueens.c:50)
+      Call 43f73b nqueens ./nqueens.c:91:29
+     Spawn 43efd7 nqueens ./nqueens.c:91:29
|*    Write 43efa9 nqueens ./nqueens.c:89:10
||        `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+     Call 43f73b nqueens ./nqueens.c:91:29
+   Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+     Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

runtime overhead is nearly constant compared with a serial execution

- ~7× slower for this example

Cilksan Report

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

ASCII art on the left edge depicts the race context.

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*      Read 43ef18 nqueens ./nqueens.c:87:3
|         `-to variable a (declared at nqueens.c:50)
+      Call 43f73b nqueens ./nqueens.c:91:29
+     Spawn 43efd7 nqueens ./nqueens.c:91:29
|*    Write 43efa9 nqueens ./nqueens.c:89:10
||        `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+     Call 43f73b nqueens ./nqueens.c:91:29
+    Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+     Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

Cilksan Report

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

ASCII art on the left edge depicts the race context.

* = racing instructions

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*     Read 43ef18 nqueens ./nqueens.c:87:3
|         `-to variable a (declared at nqueens.c:50)
+     Call 43f73b nqueens ./nqueens.c:91:29
+    Spawn 43efd7 nqueens ./nqueens.c:91:29
|*   Write 43efa9 nqueens ./nqueens.c:89:10
||        `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+    Call 43f73b nqueens ./nqueens.c:91:29
+   Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+    Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

Cilksan Report

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*      Read 43ef18 nqueens ./nqueens.c:87:3
|         `-to variable a (declared at nqueens.c:50)
+      Call 43f73b nqueens ./nqueens.c:91:29
+     Spawn 43efd7 nqueens ./nqueens.c:91:29
|*    Write 43efa9 nqueens ./nqueens.c:89:10
||        `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+     Call 43f73b nqueens ./nqueens.c:91:29
+    Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+     Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

ASCII art on the left edge depicts the race context.

- * = racing instructions
- \+ = stack frames (call/spawn)

Cilksan Report

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*      Read 43ef18 nqueens ./nqueens.c:87:3
|         `-to variable a (declared at nqueens.c:50)
+      Call 43f73b nqueens ./nqueens.c:91:29
+     Spawn 43efd7 nqueens ./nqueens.c:91:29
|*    Write 43efa9 nqueens ./nqueens.c:89:10
||        `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+     Call 43f73b nqueens ./nqueens.c:91:29
+   Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+     Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

ASCII art on the left edge depicts the race context.

- `*` = racing instructions
- `+` = stack frames (call/spawn)
- `|/` = common calling context

Cilksan Report

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

ASCII art on the left edge depicts the race context.

- `*` = racing instructions
- `+` = stack frames (call/spawn)
- `|/` = common calling context

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*      Read 43ef18 nqueens ./nqueens.c:87:3
|         `-to variable a (declared at nqueens.c:50)
+      Call 43f73b nqueens ./nqueens.c:91:29
+     Spawn 43efd7 nqueens ./nqueens.c:91:29
|*    Write 43efa9 nqueens ./nqueens.c:89:10
||        `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+     Call 43f73b nqueens ./nqueens.c:91:29
+    Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+     Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

Cilksan Report

nqueens.c

```
[...]
b = (char*) alloca((j+1) * sizeof(char));
memcpy(b, a, j * sizeof(char));
for (int i = 0; i < n; i++) {
  b[j] = i;  /* <-- racy write! */
  if (ok(j+1,b))
    cnt[i] = cilk_spawn nqueens(n,j+1,b);
}
[...]
```

terminal

```
$ ./nqueens 12
Running Cilksan race detector
Running ./nqueens with n = 12.
Race detected at address 7f7db6c0f2e6
*     Read 43ef18 nqueens ./nqueens.c:87:3
|        `-to variable a (declared at nqueens.c:50)
+     Call 43f73b nqueens ./nqueens.c:91:29
+    Spawn 43efd7 nqueens ./nqueens.c:91:29
|*   Write 43efa9 nqueens ./nqueens.c:89:10
||       `-to variable b (declared at ./nqueens.c:53)
|/ Common calling context
+    Call 43f73b nqueens ./nqueens.c:91:29
+   Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
+    Call 43f42b main ./nqueens.c:125:9
   Allocation context
    Stack object b (declared at ./nqueens.c:53)
     Alloc 43eef8 in nqueens ./nqueens.c:86:16
      Call 43f73b nqueens ./nqueens.c:91:29
     Spawn 43efd7 nqueens ./nqueens.c:91:29
[...]
      Call 43f42b main ./nqueens.c:125:9

2.544000
Total number of solutions : 14200

Race detector detected total of 1 races.
Race detector suppressed 3479367 duplicate error
messages
$
```

ASCII art on the left edge depicts the race context.

- `*` = racing instructions
- `+` = stack frames (call/spawn)
- `|/` = common calling context
- `␣` = allocation context

Tips for Effective Performance Engineering

- Maintain the invariant that your code is correct.
- Regression test heavily and automatically to ensure correctness.
- Don’t be a slob: Treat your source code with respect.

Good code hygiene enables fast code

WHAT IS PARALLELISM?

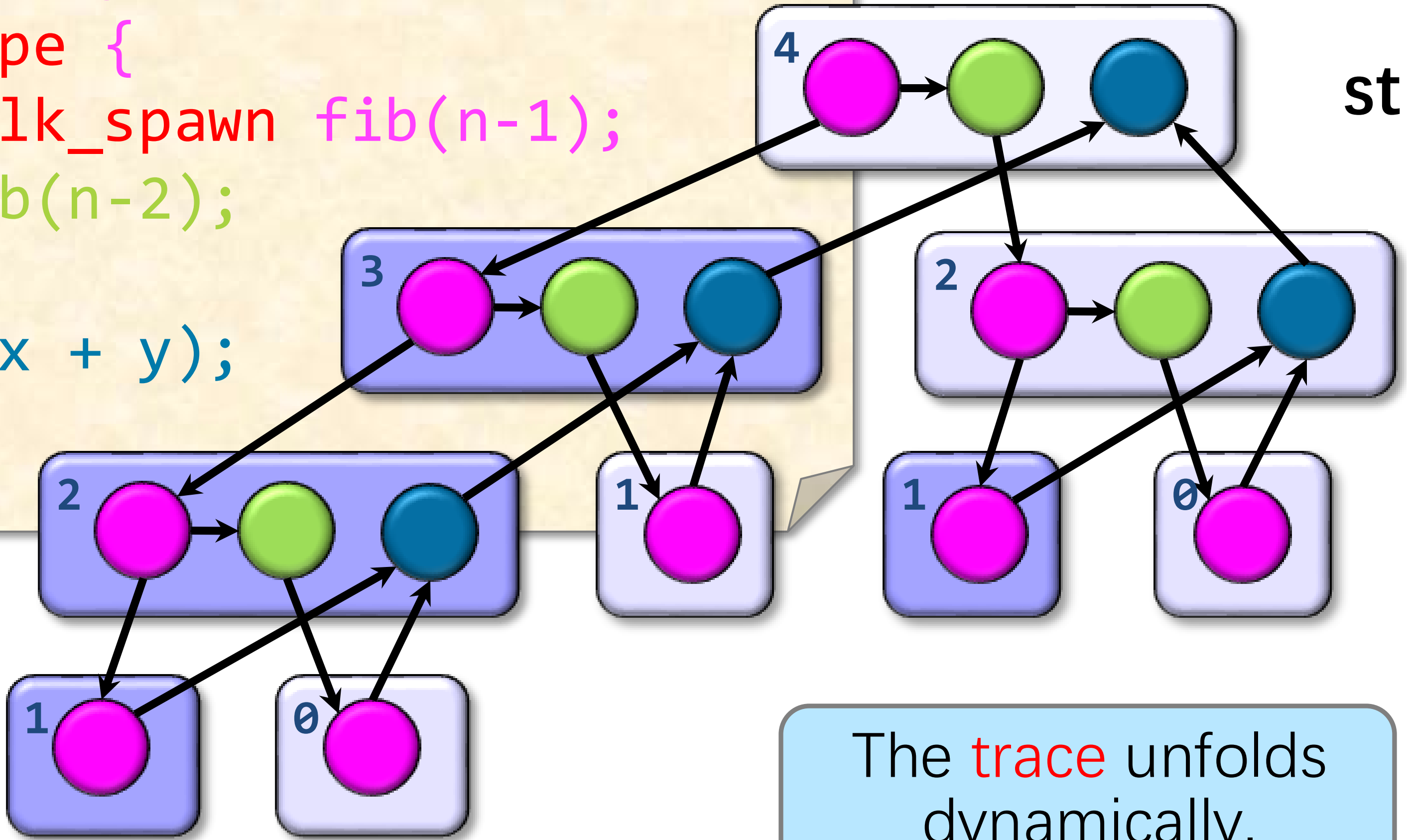
Execution Model

```
int64_t fib(int64_t n) {
  if (n < 2)
    return n;
  int64_t x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Execution Model

```
int64_t fib(int64_t n) {
  if (n < 2)
    return n;
  int64_t x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
```

Example:
`fib(4)`

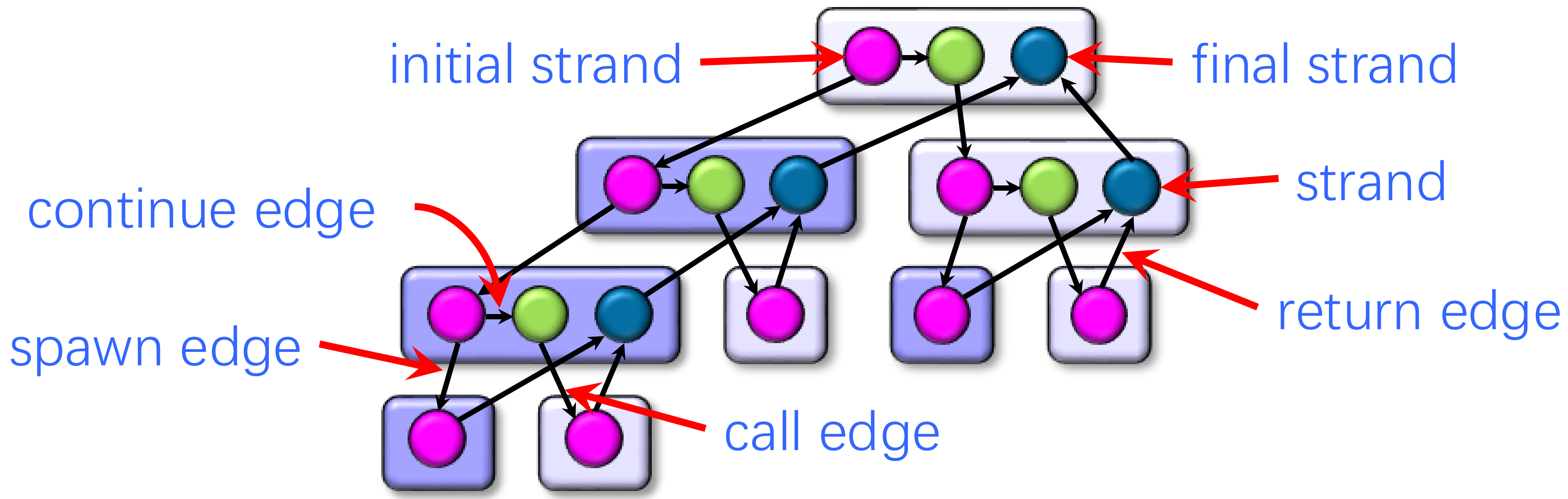


strand

"Processor oblivious"

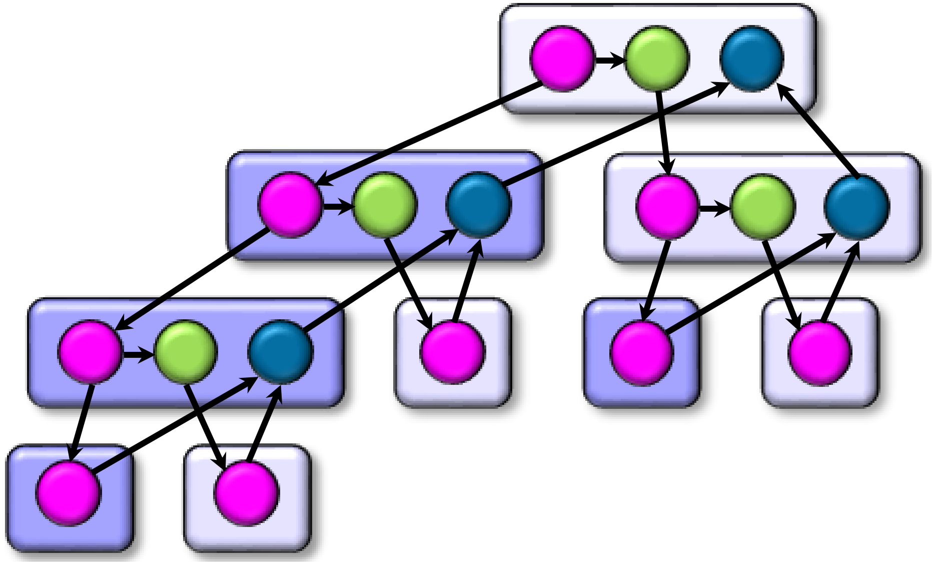
The trace unfolds dynamically.

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 edge.
- The compiler converts loop parallelism (`cilk_for`) to spawns and syncs using recursive divide-and-conquer.

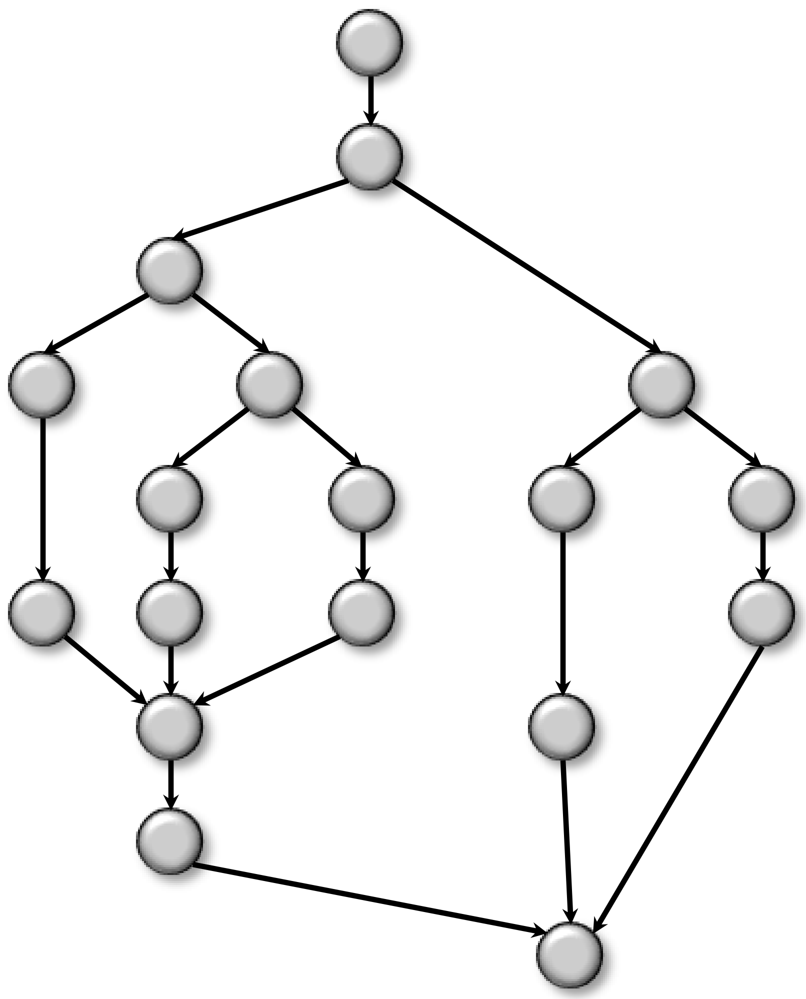
How Much Parallelism?



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?

Example Trace Dag



Q. What is the parallelism (maximum possible speedup) of this computation, assuming that each strand executes in unit time? Pick the closest number.

a. 1
b. 2
c. 3
d. 4
e. 5
f. 6

Amdahl's "Law"

Gene M. Amdahl

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on.*

*Paraphrased.

Amdahl's "Law"

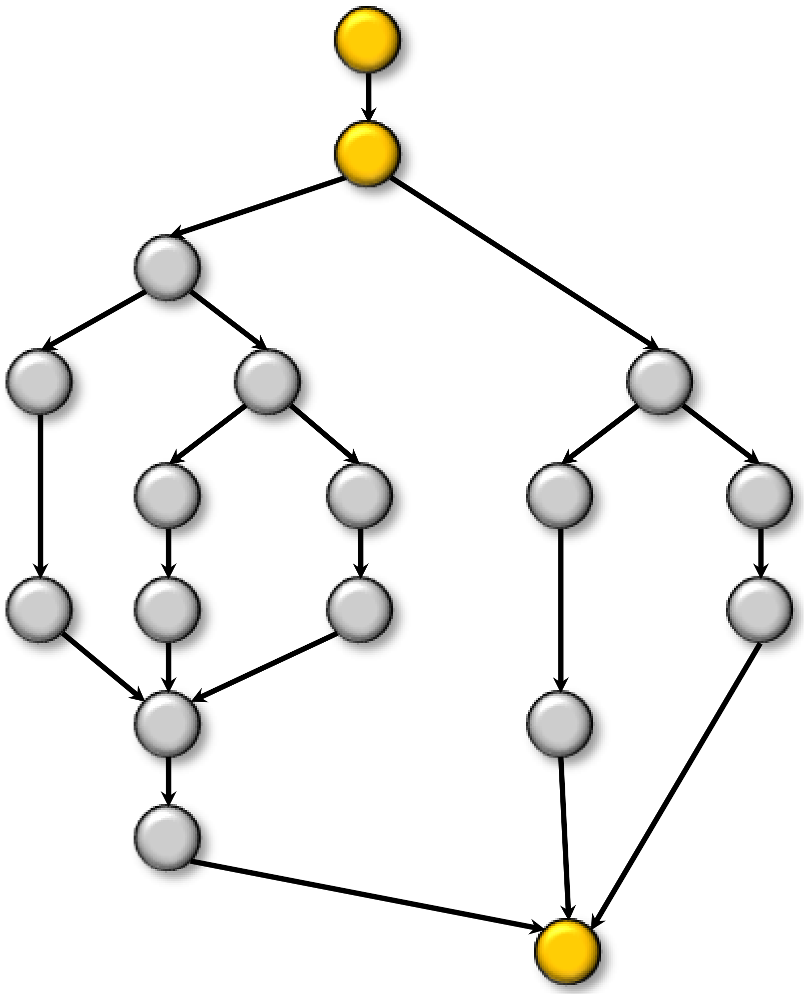
- In general, if a fraction **α** of an application must be run serially, the speedup can be at most **1/α**.

$$\text{Speedup} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

α is the a fraction of the application must be run serially
$\boldsymbol{P}$ is the number of processors

Quantifying Parallelism

What is the **parallelism** of this computation?

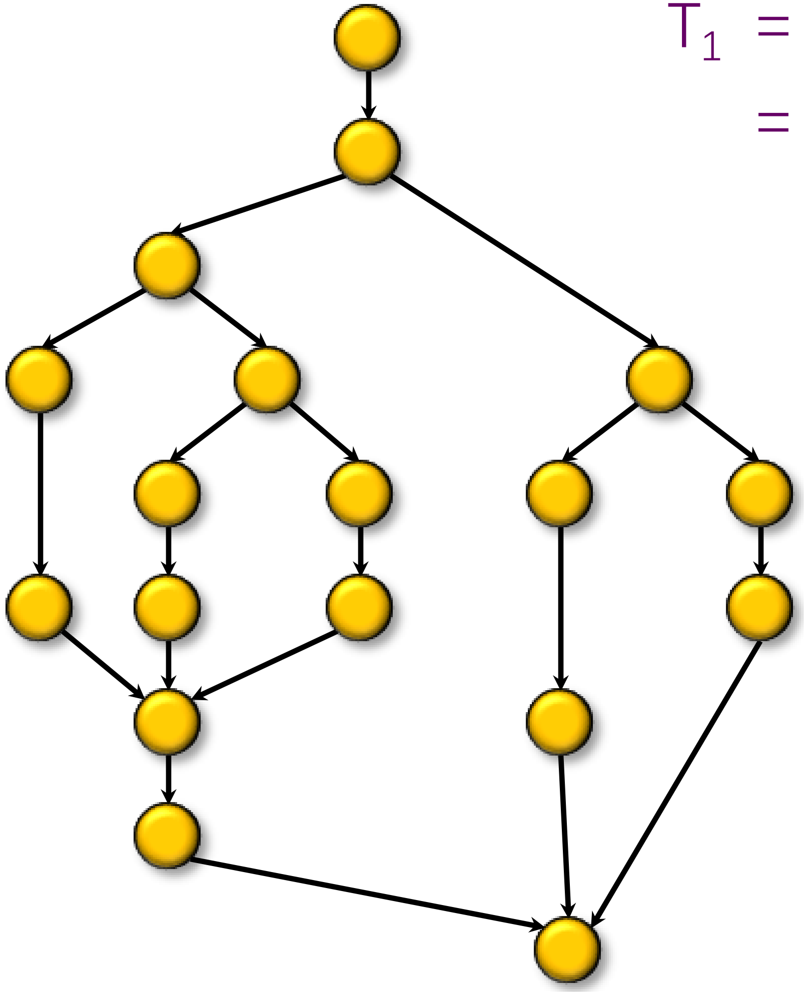


Amdahl's Law says that since the serial fraction is **3/18 = 1/6**, the speedup is upper-bounded by **6**.

But this bound is **weak**.

Performance Measures

T_P = execution time on P processors

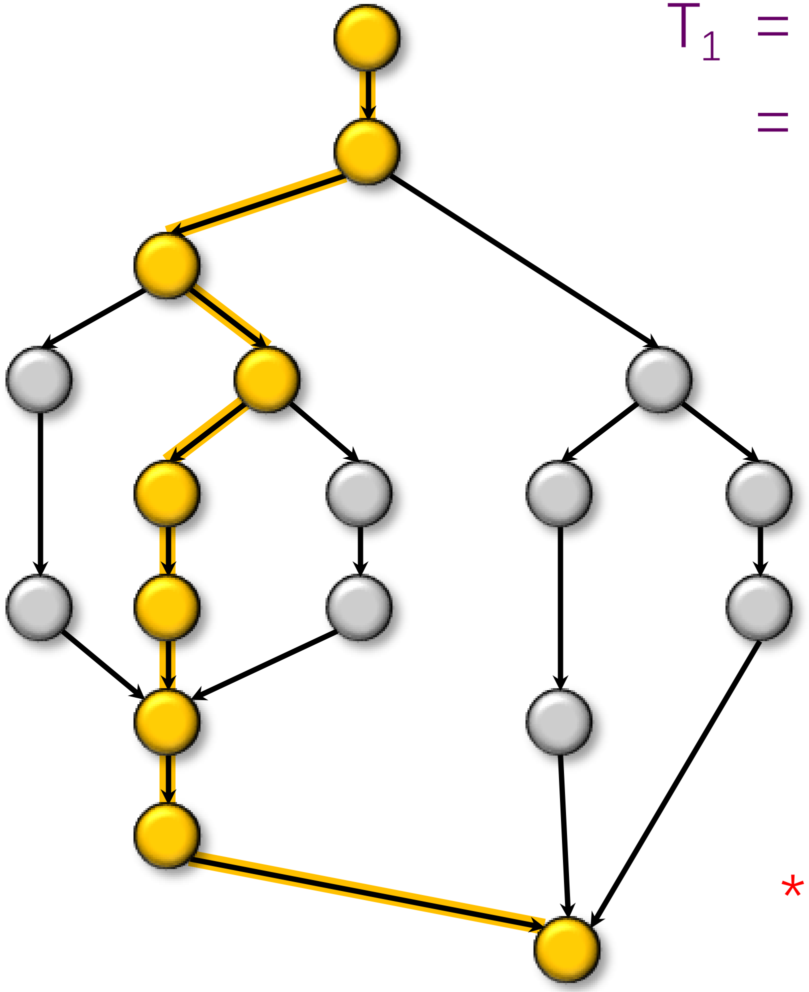


T_1 = work

= 18

Performance Measures

T_P = execution time on $\boldsymbol{P}$ processors



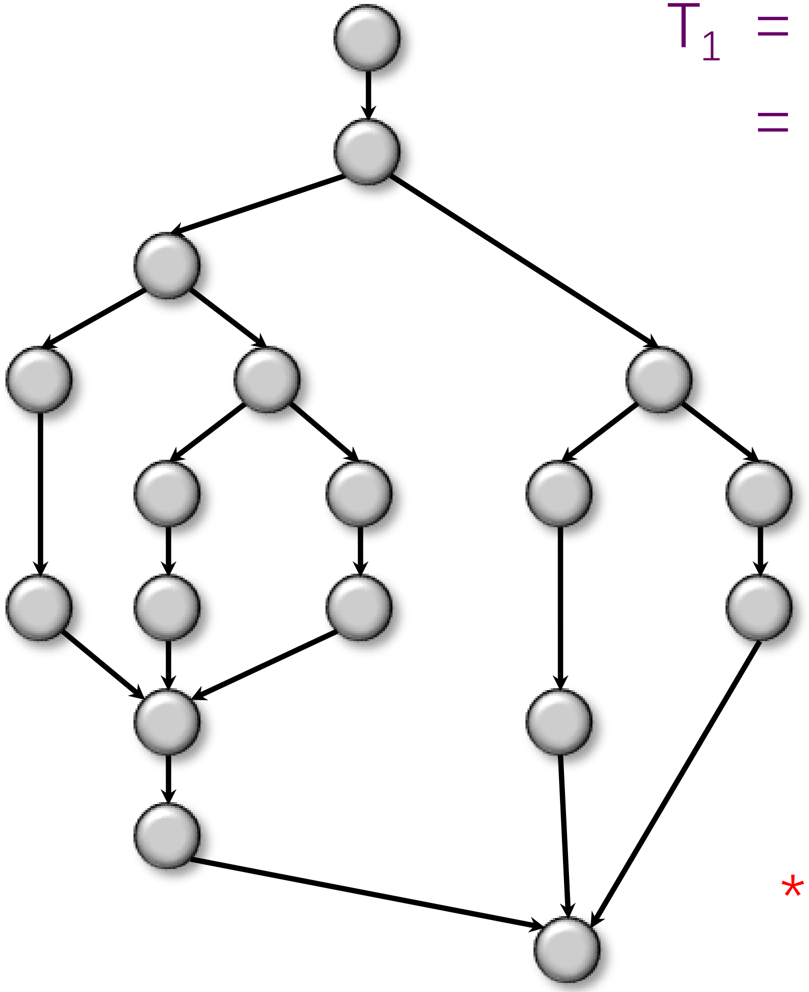
T_1 = work
= 18

T_∞ = span*
= 9

* Also called critical-path length or computational depth.

Performance Measures

T_P = execution time on ***P*** processors



T_1 = work = 18

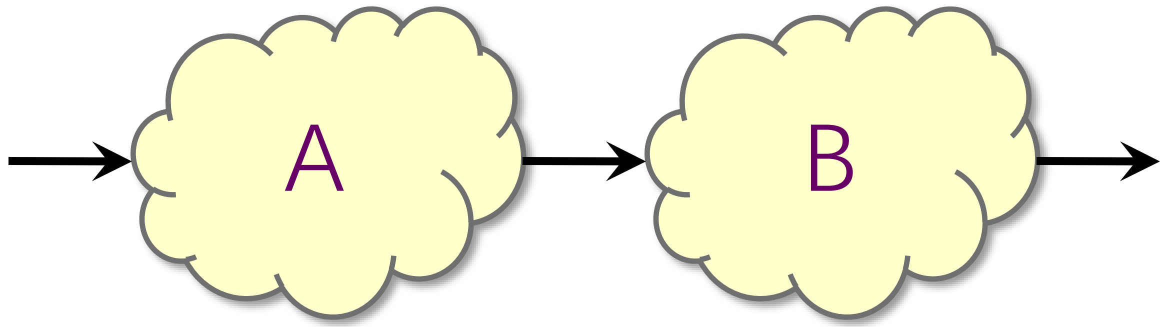
T_∞ = span* = 9

WORK LAW
$T_P \geqslant T_1/P$

SPAN LAW
$T_P \geqslant T_\infty$

* Also called critical-path length or computational depth.

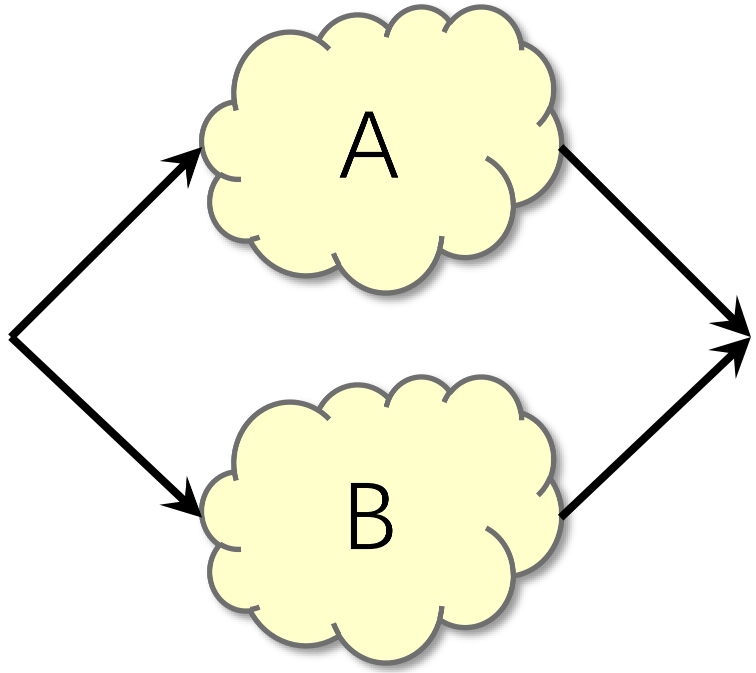
Series Composition



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

Parallel Composition



Work: $T_1(A \cup B) = T_1(A) + T_1(B)$

Span: $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$

Speedup

Definition. T_1/T_P = speedup on P processors.

- If $T_1/T_P = P$, we have (perfect) linear speedup.
- If $T_1/T_P < P$, we have sublinear 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 $T_P \geqslant T_1/P$.

Parallelism

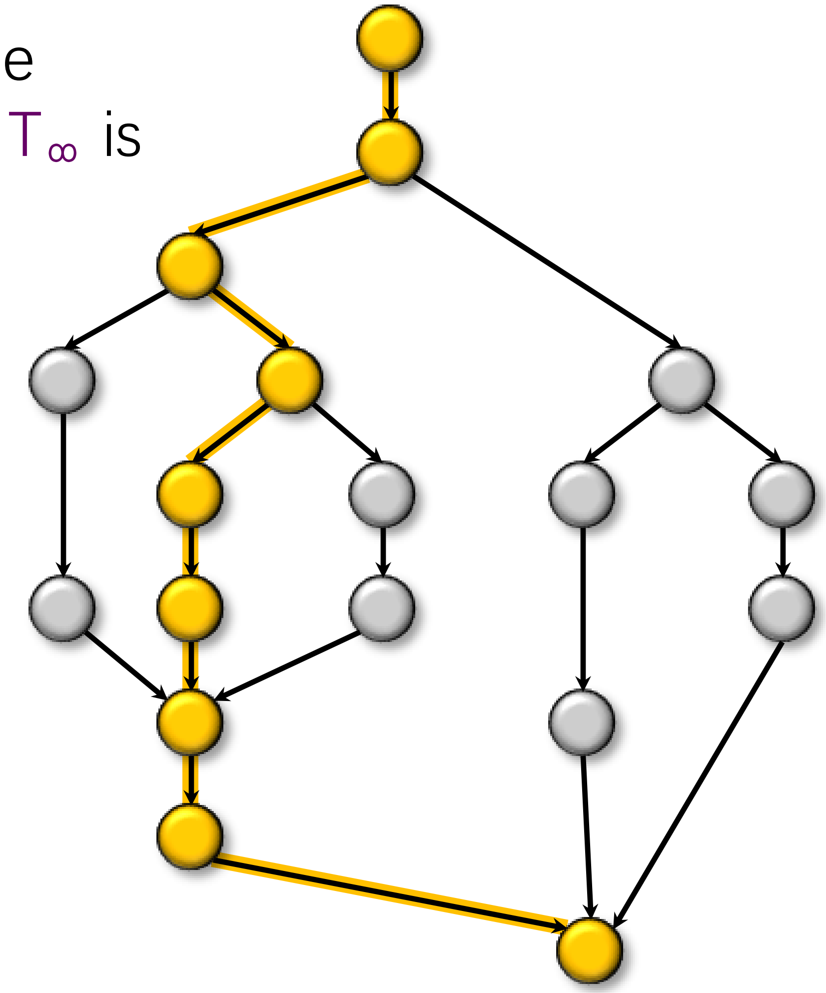
As the SPAN LAW dictates that $T_P \geqslant T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = parallelism

= the average amount of work per step along the span

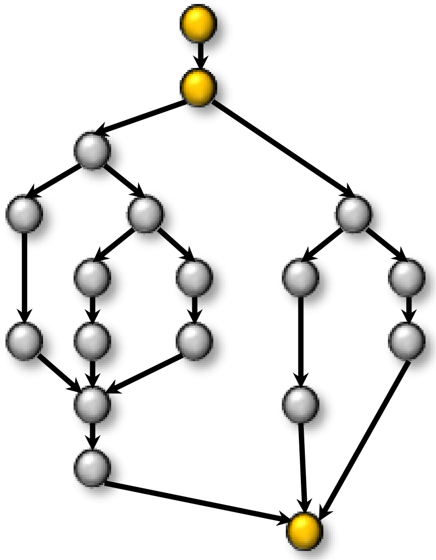
= $18/9$

= 2 .



Amdahl's Law vs. the Span Law

Amdahl's Law for a program where α of T_1 must run serially:



$T_1/T_\infty = 18/9 = 2$

$$\text{Speedup} = T_1/T_P = \frac{T_1}{\alpha T_1 + (1-\alpha)T_1/P} = \frac{1}{\alpha + \frac{1-\alpha}{P}}$$

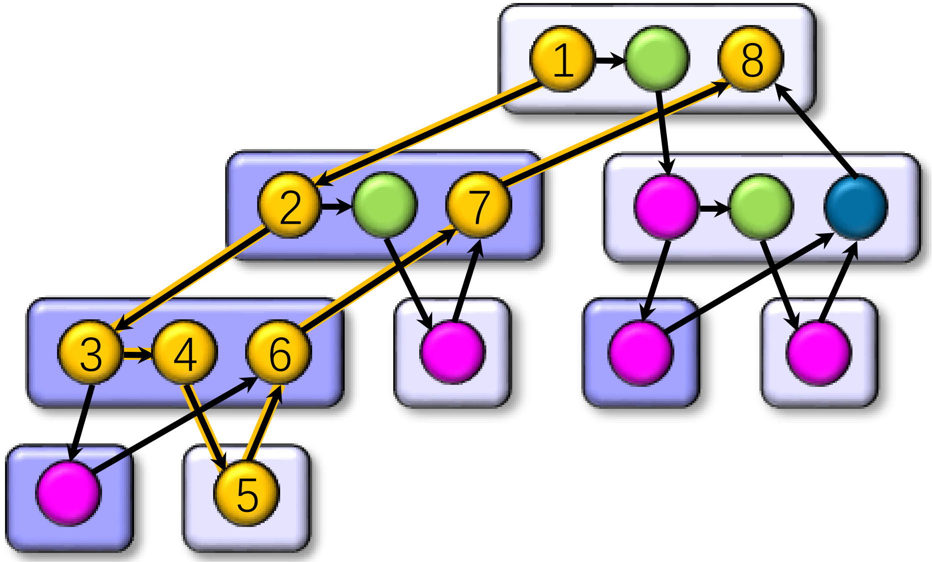
ignores how much of parallelism can actually be attained due to dependencies

$$P \Rightarrow \infty \quad = \frac{1}{\alpha} = 6$$

$$P = 2 \quad = \frac{18}{3 + 15/2} = 1.7$$

$$P = 10 \quad = \frac{18}{3 + 15/10} = 4.0 \ ?$$

Example: `fib(4)`



Assume for simplicity that each strand in `fib(4)` takes unit time to execute

Work: $T_1 = \mathbf{17}$

Span: $T_\infty = \mathbf{8}$

Parallelism: $T_1/T_\infty = \mathbf{2.125}$

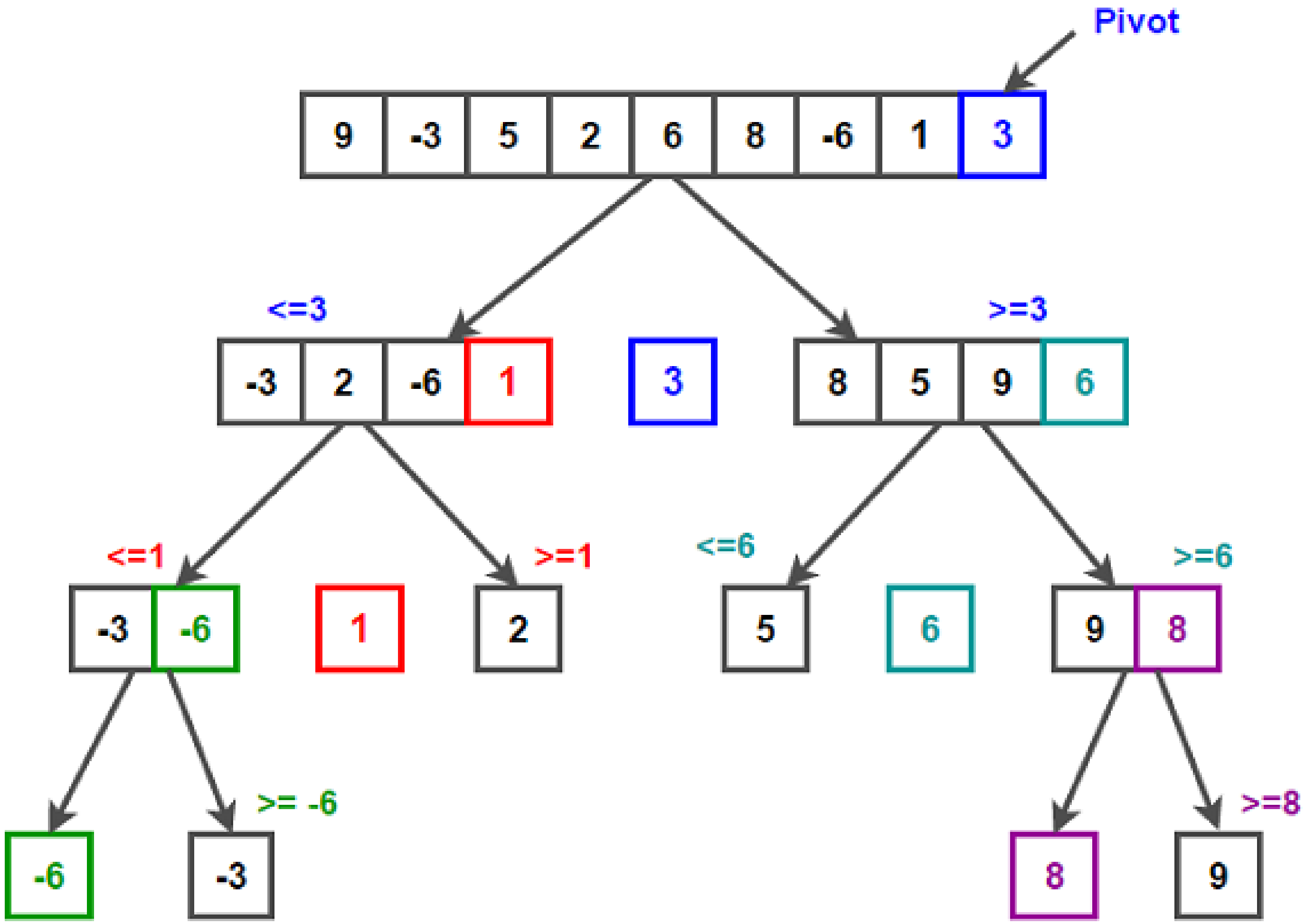
Using many more than **2** processors can yield only marginal performance gains.

The Cilkscale Scalability Analyzer

Cilkscale Scalability Analyzer

- The OpenCilk compiler provides a scalability analyzer called Cilkscale
- 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.

Quicksort



Parallelizing Quicksort

Example: quicksort

```
static void qsort(int * begin, int * end)
{
  if (begin < end) {
     int last = *(end - 1);
     // linear-time partition
     int * middle = partition(begin, end - 1, last);
     // move pivot to middle
     swap(end - 1, middle);
     // recurse
     qsort(begin, middle);
     qsort(middle + 1, end);
}
```

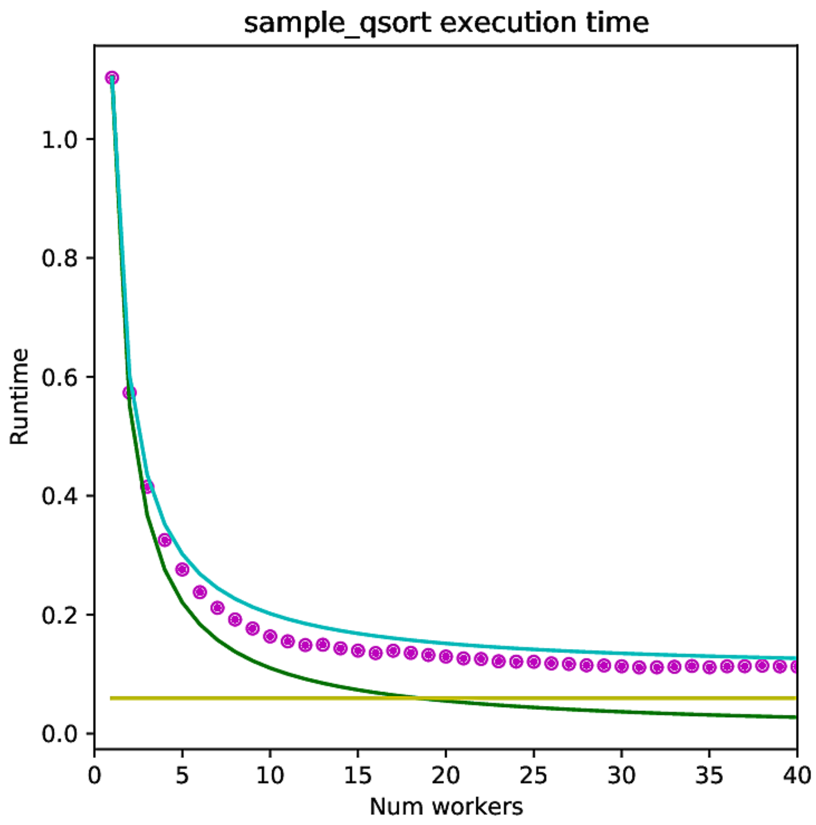
Parallelizing Quicksort

Example: Parallel quicksort

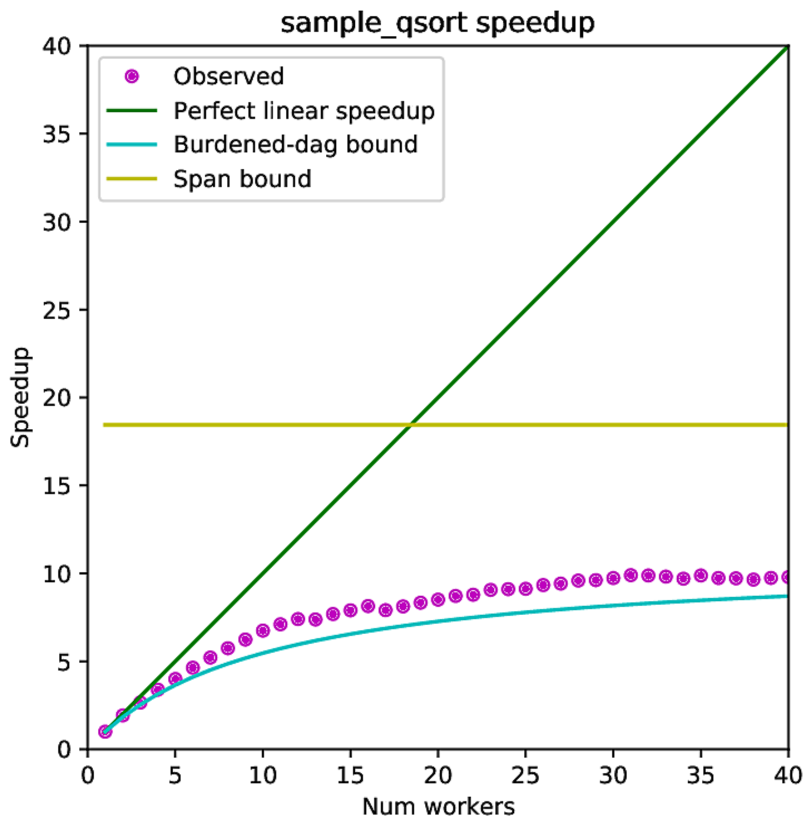
```
static void p_qsort(int* begin, int* end)
{
  if (begin < end) {
     int last = *(end - 1);
     // linear-time partition
     int * middle = partition(begin, end - 1, last);
     // move pivot to middle
     swap(end - 1, middle);
     // recurse
     cilk_scope {
       cilk_spawn p_qsort(begin, middle);
       p_qsort(middle + 1, end);
     }
}
```

Analyze the sorting of 10,000,000 numbers. ★★★ ***Guess the parallelism!*** ★★★

Cilkscale: Scalability Visualizer

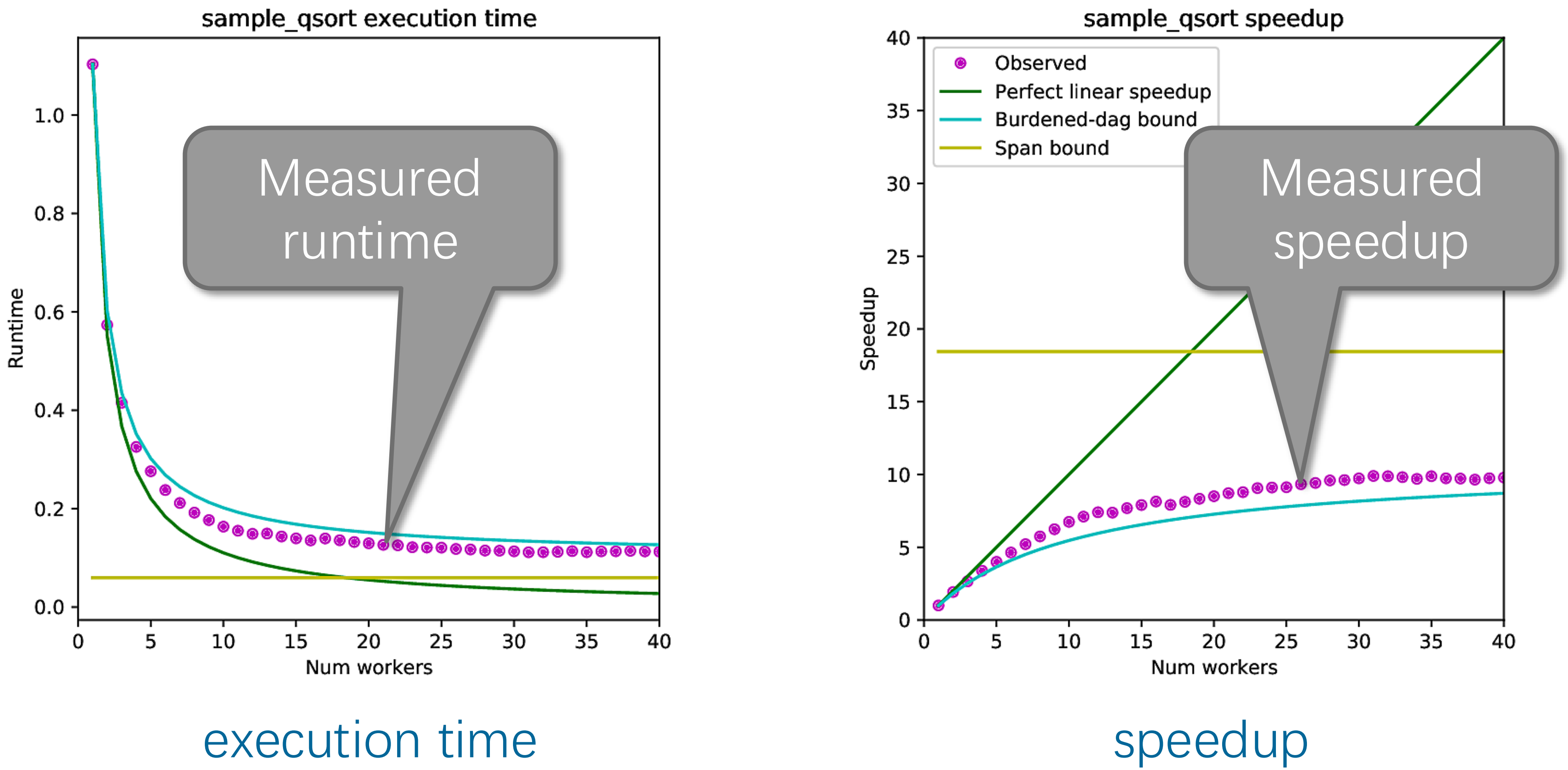


execution time



speedup

Cilkscale: Scalability Visualizer



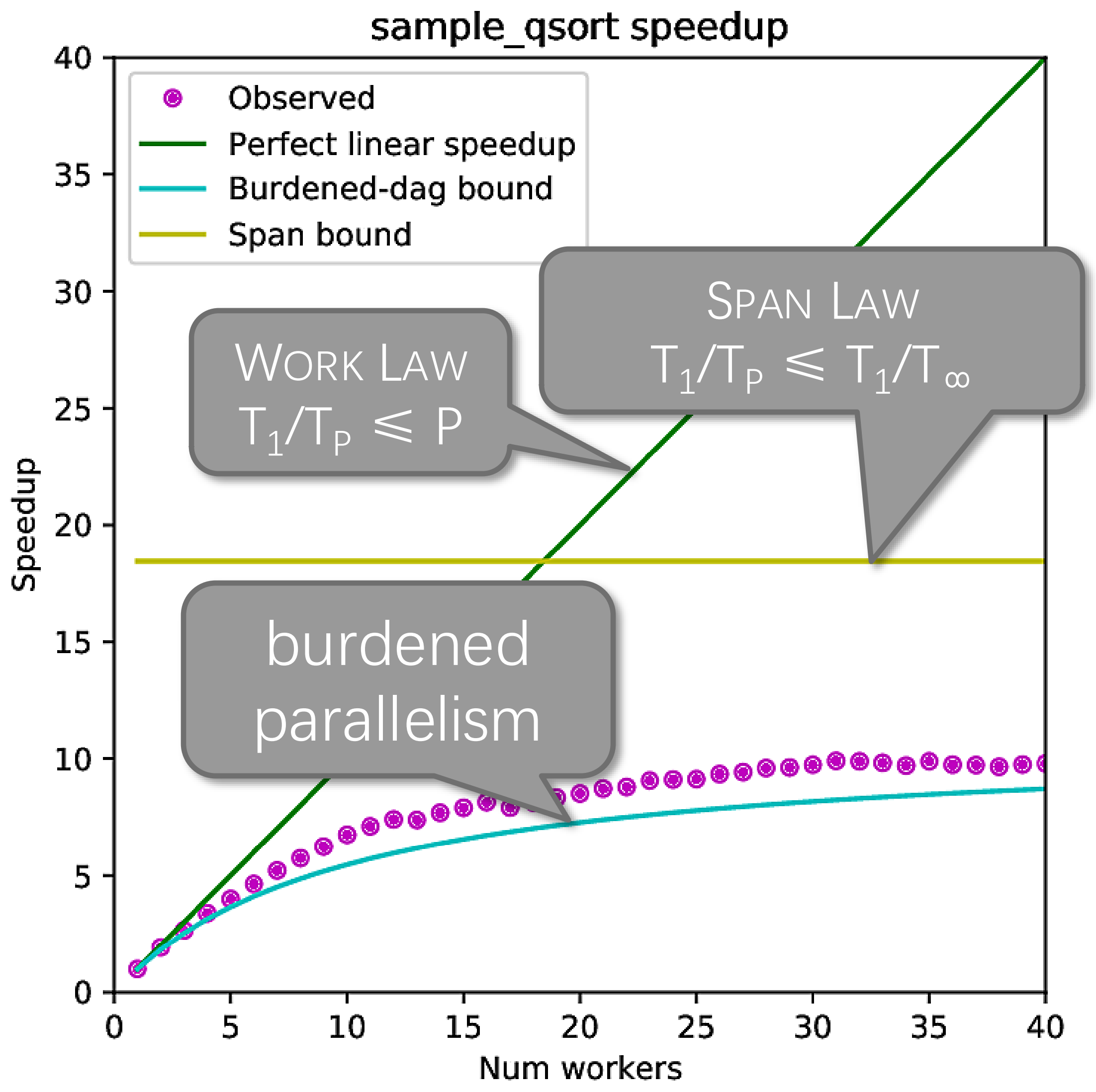
Cilksan autobenchmarks the code, running it on 1, 2, 3, ⋯ processors, and the visualizer displays the results.

Cilkscale: Speedup Analysis

Cilkscale's analyzer determines the work and span.

The visualizer plots the WORK and SPAN LAWS.

The visualizer also plots burdened parallelism, which indicates whether the program may incur scheduling overhead



speedup

Theoretical Analysis

Example: Parallel quicksort

```
static void p_qsort(int* begin, int* end) {
  if (begin < end) {
     int last = *(end - 1);
     // linear-time partition
     int * middle = partition(begin, end - 1, last);
     // move pivot to middle
     swap(end - 1, middle);
     // recurse
     cilk_scope {
       cilk_spawn p_qsort(begin, middle);
       p_qsort(middle + 1, end);
     }
}
```

Expected work = **$\Theta(n \lg n)$**

Expected span = **$\Theta(n)$**

Parallelism = **$\Theta(\lg n)$**

puny

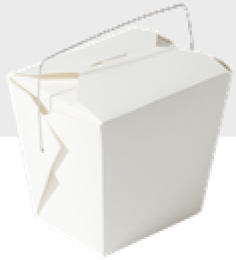
Interesting Practical* Algorithms

Algorithm	Work	Span	Parallelism
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$	$\Theta(n/\lg^2 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$	$\Theta(n^3/\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$	$\Theta(n^{\lg 7}/\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$	$\Theta(n^2/\lg n)$
Tableau construction	$\Theta(n^2)$	$\Theta(n^{\lg 3})$	$\Theta(n^{2-\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n/\lg n)$
Breadth-first search	$\Theta(E)$	$\Theta(\Delta \lg V)$	$\Theta(E/\Delta \lg V)$

---

*Cilk on 1 processor competitive with the best C.

Take-Aways



- **Determinacy races** are usually bugs.
- Determinacy races can be detected and localized using **Cilksan**
  - and a good regression-testing methodology
- The **Work & Span Laws** provide lower bounds on the parallelism
  - maximum possible speedup
- **Cilkscale** can analyze the work, span, and parallelism of a computation
- Many highly parallel and work-efficient algorithms can be programmed in Cilk