Lecture 6
Races and Parallelism

Charles E. Leiserson
September 23, 2014
Recall: Basics of Cilk

```c
int fib(int n) {
    if (n < 2) return n;
    int 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.
Loop Parallelism in Cilk

Example: In-place matrix transpose

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

```c
// 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;
  }
}
```
DETETERMINANCY RACES
Race Conditions

Race conditions are the bane of concurrency. Famous race bugs include the following:

- Therac–25 radiation therapy machine — killed 3 people and seriously injured many more.
- North American Blackout of 2003 — left 50 million people without power.

Race bugs are notoriously difficult to discover by conventional testing!
**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**

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

**dependency graph**
A Closer Look

```c
int x = 0;
x++;
assert(x == 2);
```

```c
x = 0;
r1 = x;
r1++;  
x = r1;
assert(x == 2);
```

```c
x = 0;
r2 = x;
r2++;  
x = r2;
```
**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.

```
1. x = 0;
2. r1 = x;
3. r1++;  // r1 = 1
4. r2 = x;
5. r2++;  // r2 = 1
6. x = r2;  // x = 1
7. x = r1;  // x = 1
8. assert(x == 2);  // assert(1 == 2)
```
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).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Race Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>none</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>read race</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>read race</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>write race</td>
</tr>
</tbody>
</table>

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

- Iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.
  - **Note:** The 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 Intel x86–64.)
Cilkscreen Race Detector

• If an ostensibly deterministic Cilk program run on a given input could possibly behave any differently than its serial elision, Cilkscreen guarantees to report and localize the offending race.
• Employs a regression-test methodology, where the programmer provides test inputs.
• Identifies filenames, lines, and variables involved in races, including stack traces.
• Runs off the binary executable using dynamic instrumentation.
• Runs about 20–50 times slower than real-time.
• Your best friend.
WHAT IS PARALLELISM?
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)
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)

"Processor oblivious"

The computation dag unfolds dynamically.
A parallel instruction stream 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. 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?
Amdahl’s “Law”

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.

In general, if a fraction $\alpha$ of an application must be run serially, the speedup can be at most $1/\alpha$. 
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.
Performance Measures

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} = 18 \]
Performance Measures

\[ T_p = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} \quad = 18 \]

\[ T_\infty = \text{span}^* \quad = 9 \]

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

\[ T_P = \text{execution time on } P \text{ processors} \]

\[ T_1 = \text{work} = 18 \]

\[ T_\infty = \text{span}^* = 9 \]

**Work Law**
- \( T_P \geq T_1 / P \)

**Span Law**
- \( T_P \geq T_\infty \)

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

\[ T_1(A \cup B) = T_1(A) + T_1(B) \]

\[ T_\infty(A \cup B) = T_\infty(A) + T_\infty(B) \]
Parallel Composition

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

**Span:** \( T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\} \)
Definition. $T_1/T_p = \text{speedup}$ 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 \textit{Work Law} $T_p \geq T_1/P$. 
Because the SPAN LAW dictates that \( T_p \geq T_\infty \), the maximum possible speedup given \( T_1 \) and \( T_\infty \) is
\[
T_1/T_\infty = \text{parallelism}
\]
\[
= \text{the average amount of work per step along the span}
\]
\[
= 18/9
\]
\[
= 2.
\]
Example: $\text{fib}(4)$

Assume for simplicity that each strand in $\text{fib}(4)$ takes unit time to execute.

\[
\begin{align*}
\text{Work: } & T_1 = 17 \\
\text{Span: } & T_\infty = 8 \\
\text{Parallelism: } & T_1 / T_\infty = 2.125
\end{align*}
\]

Using many more than 2 processors can yield only marginal performance gains.
THE CILKVIEW
SCALABILITY ANALYZER
The Cilk Plus tool suite provides a scalability analyzer called Cilkview.

Like the Cilkscreen race detector, Cilkview uses dynamic instrumentation to analyze a serial execution of a program.

Cilkview computes work and span to derive upper bounds on parallel performance.

Cilkview also estimates scheduling overhead to compute a burdened span for lower bounds.
Quicksort Analysis

**Example:** Parallel quicksort

```c
static void quicksort(uint64_t *left, uint64_t *right)
{
    uint64_t *p;
    if (left == right) return;
    p = partition(left, right);
    cilk_spawn quicksort(left, p);
    quicksort(p + 1, right);
    cilk_sync;
}
```

Analyze the sorting of 100,000,000 numbers.

★★★ **Guess the parallelism!** ★★★
Cilkview Output

Measured speedup
Cilkview Output

Parallelism

11.21
Cilkview Output

SPAN LAW

© 2013–2014 Charles E. Leiserson and I–Ting Angelina Lee

32
Cilkview Output

WORK LAW (linear speedup)
Burdened parallelism: estimates scheduling overheads
Theoretical Analysis

Example: Parallel quicksort

```c
static void quicksort(uint64_t *left, uint64_t *right)
{
    uint64_t *p;
    if (left == right) return;
    p = partition(left, right);
    cilk_spawn quicksort(left, p);
    quicksort(p + 1, right);
    cilk_sync;
}
```

Expected work = $O(n \log n)$
Expected span = $\Omega(n)$
Parallelism = $O(\log n)$
### Interesting Practical* Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Span</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge sort</td>
<td>$\Theta(n \lg n)$</td>
<td>$\Theta(\lg^3 n)$</td>
<td>$\Theta(n/\lg^2 n)$</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>$\Theta(n^3)$</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(n^3/\lg n)$</td>
</tr>
<tr>
<td>Strassen</td>
<td>$\Theta(n^{\lg 7})$</td>
<td>$\Theta(\lg^2 n)$</td>
<td>$\Theta(n^{\lg 7}/\lg^2 n)$</td>
</tr>
<tr>
<td>LU-decomposition</td>
<td>$\Theta(n^3)$</td>
<td>$\Theta(n \lg n)$</td>
<td>$\Theta(n^2/\lg n)$</td>
</tr>
<tr>
<td>Tableau construction</td>
<td>$\Theta(n^2)$</td>
<td>$\Theta(n^{\lg 3})$</td>
<td>$\Theta(n^{2-\lg 3})$</td>
</tr>
<tr>
<td>FFT</td>
<td>$\Theta(n \lg n)$</td>
<td>$\Theta(\lg^2 n)$</td>
<td>$\Theta(n/\lg n)$</td>
</tr>
<tr>
<td>Breadth–first search</td>
<td>$\Theta(E)$</td>
<td>$\Theta(\Delta \lg V)$</td>
<td>$\Theta(E/\Delta \lg V)$</td>
</tr>
</tbody>
</table>

* *Cilk on 1 processor competitive with the best C.*
Scheduling

- Cilk allows the programmer to express potential parallelism in an application.
- The Cilk scheduler maps strands onto processors dynamically at runtime.
- Since the theory of distributed schedulers is complicated, we’ll explore the ideas with a centralized scheduler.
**IDEA:** Do as much as possible on every step.

**Definition.** A strand is **ready** if all its predecessors have executed.
**IDEA:** Do as much as possible on every step.

**Definition.** A strand is **ready** if all its predecessors have executed.

**Complete step**
- $\geq P$ strands ready.
- Run any $P$.
**IDEA:** Do as much as possible on every step.

**Definition.** A strand is ready if all its predecessors have executed.

**Complete step**
- $\geq P$ strands ready.
- Run any $P$.

**Incomplete step**
- $< P$ strands ready.
- Run all of them.
Theorem [G68, B75, EZL89]. Any greedy scheduler achieves

\[ T_P \leq T_1/P + T_\infty. \]

Proof.

- # complete steps ≤ \( T_1/P \), since each complete step performs \( P \) work.
- # incomplete steps ≤ \( T_\infty \), since each incomplete step reduces the span of the unexecuted dag by 1. ■
Corollary. Any greedy scheduler achieves within a factor of 2 of optimal.

Proof. Let $T_p^*$ be the execution time produced by the optimal scheduler. Since $T_p^* \geq \max\{T_1/P, T_\infty\}$ by the WORK and SPAN LAWS, we have

\[
T_p \leq T_1/P + T_\infty \\
\leq 2 \cdot \max\{T_1/P, T_\infty\} \\
\leq 2T_p^*. \quad \blacksquare
\]
**Linear Speedup**

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

**Proof.** Since $T_1 / T_\infty \gg P$ is equivalent to $T_\infty \ll T_1 / P$, the Greedy Scheduling Theorem gives us

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

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

**Definition.** The quantity $T_1 / PT_\infty$ is called the parallel slackness.
Cilk Performance

- Cilk’s work-stealing scheduler achieves:
  - $T_P = T_1/P + O(T_\infty)$ expected time (provably);
  - $T_P \approx T_1/P + T_\infty$ time (empirically).

- Near-perfect linear speedup as long as $P \ll T_1/T_\infty$.

- Instrumentation in Cilkview allows you to measure $T_1$ and $T_\infty$. 
THE CILK RUNTIME SYSTEM
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

Spawn!
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

When a worker runs out of work, it steals from the top of a random victim’s deque.
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

When a worker runs out of work, it steals from the top of a random victim’s deque.
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

```
spawn
  call
  call
  call
spawn
spawn

P
P
P
P
```

When a worker runs out of work, it steals from the top of a random victim’s deque.
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

When a worker runs out of work, it steals from the top of a random victim’s deque.
Cilk Runtime System

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

When a worker runs out of work, it steals from the top of a random victim’s deque.
Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].

**Theorem** [BL94]: With sufficient parallelism, workers steal infrequently ⇒ linear speed–up.
**Work–Stealing Bounds**

**Theorem** [BL94]. The Cilk work–stealing scheduler achieves expected running time

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

on $P$ processors.

**Pseudoproof.** A processor is either working or stealing. The total time all processors spend working is $T_1$. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(PT_\infty)$. Since there are $P$ processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty).$$
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.
**Theorem.** Let $S_1$ be the stack space required by a serial execution of a Cilk program. Then the stack space required by a $P$-processor execution is at most $S_P \leq PS_1$.

**Proof** (by induction). The work-stealing algorithm maintains the busy-leaves property: Every extant leaf activation frame has a worker executing it. ■
Linguistic Implications

Code like the following executes properly without any risk of blowing out memory:

```c
for (int i=1; i<1000000000; ++i) {
    cilk_spawn foo(i);
}
cilk_sync;
```

**MORAL**

*Only monsters steal children!* ★★★★★