LECTURE 6
Parallelism and Scalability

Charles E. Leiserson

September 27, 2011
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

\[
\begin{pmatrix}
a_{11} & a_{12} & \ldots & a_{1n} \\
a_{21} & a_{22} & \ldots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n1} & a_{n2} & \ldots & a_{nn}
\end{pmatrix}
\rightarrow
\begin{pmatrix}
a_{11} & a_{21} & \ldots & a_{n1} \\
a_{12} & a_{22} & \ldots & a_{n2} \\
\vdots & \vdots & \ddots & \vdots \\
a_{1n} & a_{2n} & \ldots & a_{nn}
\end{pmatrix}
\]

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;
    }
}
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 call, spawn, sync, or return (or thrown exception).

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?
• What Is Parallelism?
• The Cilkview Scalability Analyzer
• Scheduling Theory
• Cilk Runtime System
• A Chess Lesson
OUTLINE

• What Is Parallelism?
• The Cilkview Scalability Analyzer
• Scheduling Theory
• Cilk Runtime System
• A Chess Lesson
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 $1/\alpha$. 

Gene M. Amdahl
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.
$T_p = \text{execution time on } P \text{ processors}$
$T_P = \text{execution time on } P \text{ processors}$

$T_1 = work$

$= 18$
**Performance Measures**

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

\[ T_1 = \text{work} \quad T_\infty = \text{span}^* \]

\[ = 18 \quad = 9 \]

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

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

\[ T_1 = \text{work} \quad T_\infty = \text{span}^* \]

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

**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:** \( T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\} \)
Def. \[ \frac{T_1}{T_P} = \text{speedup} \] on \( P \) processors.

If \( \frac{T_1}{T_P} < P \), we have \textit{sublinear speedup}.
If \( \frac{T_1}{T_P} = P \), we have \textit{(perfect) linear speedup}.
If \( \frac{T_1}{T_P} > P \), we have \textit{superlinear speedup},
which is not possible in this performance model, because of the \textit{Work Law} \( T_P \geq \frac{T_1}{P} \).
Because the Span Law dictates that $T_p \geq T_\infty$, the maximum possible speedup given $T_1$ and $T_\infty$ is $T_1/T_\infty = \text{parallelism}$

= 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.

$\textbf{Work: } T_1 = 17$

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

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

Using many more than 2 processors can yield only marginal performance gains.
• What Is Parallelism?
• The Cilkview Scalability Analyzer
• Scheduling Theory
• Cilk Runtime System
• A Chess Lesson
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.
Example: Parallel quicksort

```cpp
template<typename T>
void qsort(T begin, T end) {
    if (begin != end) {
        T middle = partition(
            begin,
            end,
            bind2nd( less<typename iterator_traits<T>::value_type>(),
                     *begin )
        );
        cilk_spawn qsort(begin, middle);
        qsort(max(begin + 1, middle), end);
        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

- Measured Speedup
- Lower Performance Bound
- Upper Performance Bound
- Application Parallelism = 11.21
- Ideal Speedup

© 2010 Charles E. Leiserson
Cilkview Output

Work Law (linear speedup)
Cilkview Output

Burdened parallelism — estimates scheduling overheads
Example: Parallel quicksort

```c
template <typename T>
void qsort(T begin, T end) {
    if (begin != end) {
        T middle = partition(
            begin,
            end,
            bind2nd( less<typename iterator_traits<T>::value_type>(),
                *begin )
        );
        cilk_spawn qsort(begin, middle);
        qsort(max(begin + 1, middle), end);
        cilk_sync;
    }
}
```

Expected work = $O(n \lg n)$
Expected span = $\Omega(n)$
Parallelism = $O(\lg n)$
## 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++. 

© 2010 Charles E. Leiserson
OUTLINE

• What Is Parallelism?
• The Cilkview Scalability Analyzer
• Scheduling Theory
• Cilk Runtime System
• A Chess Lesson
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 \( \leq T_1/P \), since each complete step performs \( P \) work.
- # incomplete steps \( \leq T_\infty \), 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

\[
T_p \leq T_1/P + T_\infty \\
\leq 2 \cdot \max\{T_1/P, T_\infty\} \\
\leq 2 T_p^* .
\]
Corollary. Any greedy scheduler achieves near-perfect linear speedup whenever \( \frac{T_1}{T_\infty} \gg P \).

Proof. Since \( \frac{T_1}{T_\infty} \gg P \) is equivalent to \( T_\infty \ll \frac{T_1}{P} \), the Greedy Scheduling Theorem gives us

\[
T_P \leq \frac{T_1}{P} + T_\infty \approx \frac{T_1}{P}.
\]

Thus, the speedup is \( \frac{T_1}{T_P} \approx P \). ■

Definition. The quantity \( \frac{T_1}{P T_\infty} \) is called the parallel slackness.
Cilk Performance

- Cilk’s work-stealing scheduler achieves
  - \( T_p = \frac{T_1}{P} + O(T_\infty) \) expected time (provably);
  - \( T_p \approx \frac{T_1}{P} + T_\infty \) time (empirically).
- Near-perfect linear speedup as long as \( P \ll \frac{T_1}{T_\infty} \).
- Instrumentation in Cilkview allows the programmer to measure \( T_1 \) and \( T_\infty \).
OUTLINE

• What Is Parallelism?
• The Cilkview Scalability Analyzer
• Scheduling Theory
• Cilk Runtime System
• A Chess Lesson
Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
Each worker (processor) maintains a \textit{work deque} of ready strands, and it manipulates the bottom of the deque like a stack \cite{MKH90, BL94, FLR98}.
Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
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].
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.
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.
Theorem [BL94]: With sufficient parallelism, workers steal infrequently ⇒ \textit{linear speed-up}.
Theorem [BL94]. The Cilk work–stealing scheduler achieves expected running time
\[ T_P \leq T_1/P + O(T_\infty) \]
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:** Better to steal parents from their children than children from their parents!
• What Is Parallelism?
• The Cilkview Scalability Analyzer
• Scheduling Theory
• Cilk Runtime System
• A Chess Lesson

● **Socrates 2.0** took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs’ 1824–node Intel Paragon.


● **Cilkchess** tied for 3rd in the 1999 WCCC running on NASA’s 256–node SGI Origin 2000.
Socrates Speedup

\[ T_P = T_\infty \]

\[ T_P = T_1/P \]

\[ T_P = T_1/P + T_\infty \]

Normalized by parallelism

\[ \frac{T_1/T_P}{T_1/T_\infty} \]

measured speedup
For the competition, Socrates was to run on a 512-processor Connection Machine Model CM5 supercomputer at the University of Illinois.

The developers had easy access to a similar 32-processor CM5 at MIT.

One of the developers proposed a change to the program that produced a speedup of over 20% on the MIT machine.

After a back-of-the-envelope calculation, the proposed “improvement” was rejected!
### Socrates Paradox

<table>
<thead>
<tr>
<th>Original program</th>
<th>Proposed program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{32} = 65$ seconds</td>
<td>$T'_{32} = 40$ seconds</td>
</tr>
<tr>
<td>$T_P \approx T_1/P + T_\infty$</td>
<td></td>
</tr>
<tr>
<td>$T_1 = 2048$ seconds</td>
<td>$T'_1 = 1024$ seconds</td>
</tr>
<tr>
<td>$T_\infty = 1$ second</td>
<td>$T'_\infty = 8$ seconds</td>
</tr>
<tr>
<td>$T_{32} = 2048/32 + 1$ = 65 seconds</td>
<td>$T'_{32} = 1024/32 + 8$ = 40 seconds</td>
</tr>
<tr>
<td>$T_{512} = 2048/512 + 1$ = 5 seconds</td>
<td>$T'_{512} = 1024/512 + 8$ = 10 seconds</td>
</tr>
</tbody>
</table>
Moral of the Story

Work and span beat running times for predicting scalability of performance.