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 can be run in parallel and the rest must run serially, the speedup is at most $1/(1-\alpha)$.

But whose application can be decomposed into just a serial part and a parallel part? For my application, what speedup should I expect?
OUTLINE

• What Is Parallelism?
• Scheduling Theory
• Cilk++ Runtime System
• A Chess Lesson
• What Is Parallelism?
• Scheduling Theory
• Cilk++ Runtime System
• A Chess Lesson
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.
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);
    }
}
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.
$T_P = \text{execution time on } P \text{ processors}$
\[ T_P = \text{execution time on } P \text{ processors} \]

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

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

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

\[ = 18 \quad = 9 \]

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

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

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

\textbf{WORK LAW}
\[ T_P \geq T_1 / P \]

\textbf{SPAN LAW}
\[ T_P \geq T_\infty \]

*Also called \textit{critical-path length} or \textit{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

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

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

If \( \frac{T_1}{T_P} = P \), we have *(perfect)* linear speedup.

If \( \frac{T_1}{T_P} > P \), we have *superlinear speedup*, which is not possible in this performance model, because of the 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

$$\frac{T_1}{T_\infty} = \textit{parallelism}$$

$= \text{the average amount of work per step along the span.}$

$= 18/9$

$= 2$. 
Example: $\text{fib}(4)$

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

**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.
• The Cilk++ tool suite provides a *scalability analyzer* called *Cilkview*.
• Like the Cilkscreen race detector, Cilkview uses *dynamic instrumentation*.
• 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
Cilkview Output

Work Law (linear speedup)
Burdened parallelism — estimates scheduling overheads
Example: Parallel quicksort

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>Θ(n lg n)</td>
<td>Θ(lg³n)</td>
<td>Θ(n/lg²n)</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>Θ(n³)</td>
<td>Θ(lg n)</td>
<td>Θ(n³/lg n)</td>
</tr>
<tr>
<td>Strassen</td>
<td>Θ(n^{lg7})</td>
<td>Θ(lg²n)</td>
<td>Θ(n^{lg7}/lg²n)</td>
</tr>
<tr>
<td>LU-decomposition</td>
<td>Θ(n³)</td>
<td>Θ(n lg n)</td>
<td>Θ(n²/lg n)</td>
</tr>
<tr>
<td>Tableau construction</td>
<td>Θ(n²)</td>
<td>Θ(n^{lg3})</td>
<td>Θ(n²-lg³)</td>
</tr>
<tr>
<td>FFT</td>
<td>Θ(n lg n)</td>
<td>Θ(lg²n)</td>
<td>Θ(n/lg n)</td>
</tr>
<tr>
<td>Breadth-first search</td>
<td>Θ(E)</td>
<td>Θ(Δ lg V)</td>
<td>Θ(E/Δ lg V)</td>
</tr>
</tbody>
</table>

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

© 2010 Charles E. Leiserson
• What Is Parallelism?
• 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

$$T_P \leq T_1/P + T_\infty \leq 2 \cdot \max\{T_1/P, T_\infty\} \leq 2T_P^*.$$
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 the programmer to measure
  $T_1$ and $T_\infty$. 

© 2010 Charles E. Leiserson
• What Is Parallelism?
• 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].
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
spawn

spawn
call
call
spawn

spawn
call

spawn
call

spawn
call

P
P
P
P
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].

![Diagram](image-url)
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].

Return!
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.
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].

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.
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].

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

**Theorem.** 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!
OUTLINE

• What Is Parallelism?
• Scheduling Theory
• Cilk++ Runtime System
• A Chess Lesson
Cilk Chess Programs

- **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

\[
\frac{T_1/T_P}{T_1/T_\infty} \quad 0.1
\]

\[
T_P = T_\infty
\]

\[
T_P = \frac{T_1}{P}
\]

\[
T_P = \frac{T_1}{P} + T_\infty
\]

measured speedup

Normalize by parallelism

\[
\frac{P}{T_1/T_\infty}
\]
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>$T_{32} = 2048/32 + 1$</td>
<td>$T'_{32} = 1024/32 + 8$</td>
</tr>
<tr>
<td></td>
<td>$= 65$ seconds</td>
</tr>
<tr>
<td></td>
<td>$= 40$ seconds</td>
</tr>
<tr>
<td>$T_{512} = 2048/512 + 1$</td>
<td>$T'_{512} = 1024/512 + 8$</td>
</tr>
<tr>
<td></td>
<td>$= 5$ seconds</td>
</tr>
<tr>
<td></td>
<td>$= 10$ seconds</td>
</tr>
</tbody>
</table>
Moral of the Story

Work and span beat running times for predicting scalability of performance.
6.172 Performance Engineering of Software Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.