The “Cilk” part is a small set of linguistic extensions to C/C++ to support fork-join parallelism. (The “Plus” part supports vector parallelism.)

Developed originally by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009.

Based on the award-winning Cilk multithreaded language developed at MIT.

Features a provably efficient work-stealing scheduler.

Provides a hyperobject library for parallelizing code with non-local variables.

Includes the Cilkscreen race detector and Cilkview scalability analyzer.
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;
    }
}
```
The serialization of a Cilk program is always a legal interpretation of the program’s semantics.

Remember, Cilk keywords grant permission for parallel execution. They do not command parallel execution.

To obtain the serialization:

```c
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```
The Cilk concurrency platform 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.

```c
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 Platform

1. **Cilk++ source**

```cpp
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);
    }
}
```

2. **Compiler**

3. **Hyperobject Library**

4. **Runtime System**

5. **Cilkscreen Race Detector**

6. **Cilkview Scalability Analyzer**

**Serialization**

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

The computation dag unfolds dynamically.

Example:
fib(4)

“Processor oblivious”
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?
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
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

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

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

\[ = 18 \quad = 9 \]

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

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

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

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

**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

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

**Span:** \( T_\infty(A \cup B) = T_\infty(A) + T_\infty(B) \)
**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 \text{sublinear speedup}.
- If \( T_1/T_P = P \), we have \text{(perfect) linear speedup}.
- If \( T_1/T_P > P \), we have \text{superlinear speedup}, which is not possible in this simple performance model, because of the \text{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

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

= the average amount of work per step along the span

= $\frac{18}{9}$

= 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.
**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 $P$. 

$P = 3$
**Greedy Scheduling**

**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’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 \( T_\infty \).