Lecture 4
The Cilk Runtime System

Alexandros–Stavros Iliopoulos
February 16, 2023
Cilk Programming

Cilk allows programmers to make software run faster using parallel processors.

**Serial fib**

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

Running time $T_S$.

**Parallelized fib using Cilk**

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

Running time $T_P$ on $P$ processors.
Scheduling in Cilk

- The Cilk concurrency platform allows the programmer to express **logical parallelism** in an application.

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
```
Scheduling in Cilk

- 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.

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
```
Scheduling in Cilk

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

source code

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

The compiler and runtime library together implement the scheduler.
WORK STEALING AND THE WORK-FIRST PRINCIPLE
Serial Execution & Stack Frames

Example:

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;

    x = fib(n-1);
    y = fib(n-2);

    return (x + y);
}
```

```
fib(4)
```
Serial Execution & Stack Frames

Example:
fib(4)

int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x + y);
}
Serial Execution & Stack Frames

Example:
fib(4)

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;

    x = fib(n-1);
    y = fib(n-2);

    return (x + y);
}
```
Serial Execution & Stack Frames

Example:

fib(4)

```c
int fib(int n) {
  if (n < 2) return n;
  int x, y;

  x = fib(n-1);
  y = fib(n-2);

  return (x + y);
}
```
Serial Execution & Stack Frames

Example:
fib(4)

```c
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  x = fib(n-1);
  y = fib(n-2);
  return (x + y);
}
```

Call stack

- 4
- 3
- 2
- 1

Execution trace

1 → 2 → 3 → 4
Serial Execution & Stack Frames

Example:

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x + y);
}
```

Call stack:
- 4
- 3
- 2

Execution trace:
1. fib(4)
2. fib(3)
3. fib(2)
4. fib(1)

© 2018–2023 MIT Algorithm Engineering Instructors
Serial Execution & Stack Frames

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x + y);
}
```

Example:
```
fib(4)
```
int fib(int n) {
    if (n < 2) return n;
    int x, y;

    x = fib(n-1);
    y = fib(n-2);

    return (x + y);
}

Example:

fib(4)
Serial Execution & Stack Frames

Example:

```java
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    x = fib(n-1);
    y = fib(n-2);
    return (x + y);
}
```

Call stack

The trace unfolds dynamically. The call stack keeps track of outstanding functions.
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}

Example:
fib(4)

The trace unfolds dynamically and expresses the logical parallelism in the program.
Work Stealing

Each worker (processor) maintains a work deque of ready strands, and it manipulates the bottom of the deque like a call stack.
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a call stack.
Each worker (processor) maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a call stack.
Work Stealing

Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack.
Work Stealing

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

- Each worker maintains a work deque of ready strands.
- The bottom of the deque is manipulated like a stack.

**Diagram:**
- The deque contains states: spawned, called, and a last state.
- The workers (processors) interact with the deque, spawning new strands and manipulating the bottom of the deque.

© 2018–2023 MIT Algorithm Engineering Instructors
Work Stealing

Each worker (processor) maintains a *work deque* of ready strands, and it manipulates the bottom of the deque like a stack.

When a worker runs out of work, it *steals* from the top of a *random* victim’s deque.
Work Stealing

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

When a worker runs out of work, it steals from the top of a random victim’s deque.
Parallel Speedup

\( T_S \) — work of a serial program
Suppose the serial program is parallelized.
\( T_1 \) — work of the parallel program
\( T_\infty \) — span of the parallel program
\( T_P \) — running time of the parallel program on \( P \) cores
Parallel scalability = \( T_1 / T_P \)
Parallel speedup = \( T_S / T_P \)
Work–Stealing Bounds

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

\[ T_P \approx \frac{T_1}{P} + O(T_\infty) \]

on \( P \) processors.
Work–Stealing Bounds

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

$$T_P \approx \frac{T_1}{P} + O(T_\infty)$$

on $P$ processors.

Time workers spend **working**.
Work–Stealing Bounds

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

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

on \( P \) processors.

- Time workers spend **working**.
- Time workers spend **stealing**.
Theorem. The Cilk work-stealing scheduler achieves expected running time

\[ T_P \approx \frac{T_1}{P} + O(T_\infty) \]

on \( P \) processors.

If the program has ample parallelism, i.e., \( \frac{T_1}{T_\infty} \gg P \), then the first term dominates, and \( T_P \approx \frac{T_1}{P} \).
Parallel Speedup

\( T_S \) — work of a serial program
Suppose the serial program is parallelized.
\( T_1 \) — work of the parallel program
\( T_\infty \) — span of the parallel program
\( T_P \) — running time of the parallel program on \( P \) cores
Parallel scalability = \( \frac{T_1}{T_P} \)
Parallel speedup = \( \frac{T_S}{T_P} \)

To achieve linear speedup on \( P \) processors over the serial program, i.e., \( T_P \approx \frac{T_S}{P} \), we need:
1. Ample parallelism: \( \frac{T_1}{T_\infty} \gg P \).
2. High work efficiency: \( \frac{T_S}{T_1} \approx 1 \).
The Work–First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk scheduler aims to maintain high work efficiency by abiding by the work–first principle:

Optimize for ordinary serial execution, at the expense of some additional overhead in steals.
Core Functionalities for Work Stealing
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}

Example:
fib(4)

The trace unfolds dynamically and expresses the **logical parallelism** in the program.
Workers Mirror Serial Execution

Example:

fib(4)
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk Spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {  
  if (n < 2) return n; 
  int x, y; 
  cilk_scope { 
    x = cilk_spawn fib(n-1); 
    y = fib(n-2); 
  } 
  return (x + y); 
}

Example:
fib(4)
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}

Example:
fib(4)
Successful Steals Create Parallelism

Example:

fib(4)

P2 resumes fib(4) mid-execution.
int fib(int n) {
  if (n < 2) return n;
  int x, y;
  cilk_scope {
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
  }
  return (x + y);
}

Example:
fib(4)
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
Successful Steals Create Parallelism

Example:

fib(4)
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}

Successful Steals Create Parallelism

Example: fib(4)

P3 resumes fib(3) mid-execution.
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.
```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    } // sync
    return (x + y);
}
```

Example:

fib(4)
```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    } // sync
    return (x + y);
}
```

Example:

fib(4)

Syncs (cilk_scope)

P1 %rip

P2 %rip

P3 %rip

Sync?

© 2018–2023 MIT Algorithm Engineering Instructors
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    } // sync
    return (x + y);
}
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    } // sync
    return (x + y);
}

Example:
fib(4)
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    } // sync
    return (x + y);
}
Putting Everything Together

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

Workers

P1

P2

P3

Cactus stack

© 2018–2023 MIT Algorithm Engineering Instructors
Putting Everything Together

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
```
Putting Everything Together

```c
int fib(int n) {
    if (n < 2) return n;
    int x, y;
    cilk_scope {
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
    }
    return (x + y);
}
```
Putting Everything Together

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

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

Workers

Deque

Processor state

Cactus stack

%rbx, %r10, ...

%rsp

%rip

%rbx, %r10, ...

%rsp

%rip

%rbx, %r10, ...

%rsp

%rip

%rbx, %r10, ...

%rsp

%rip

%rbx, %r10, ...

%rsp

%rip
Required Functionalities

- Each worker needs to keep track of its own execution context, including work that it is responsible for / available to be stolen.
- After a successful steal, a worker can resume the stolen function mid-execution.
- Upon a sync, a worker needs to know whether there is any spawned subroutine still executing on another worker.
Cilk Runtime Data Structures

The Cilk runtime utilizes three basic data structures as workers execute work:

- **Worker deques** to keep track of subroutines which are being executed or available to steal.
- A *Cilk stack frame structure* to represent each spawning function (*Cilk* function) and store its execution context.
- A *full-frame tree* to represent function instances that have ever been stolen (to support true parallel execution).

*henceforth simply referred to as the frame*
Division of Labor

The work-first principle guides the division of the Cilk runtime between the compiler and the runtime library.

Compiler

- Manages a handful of light-weight data structures (e.g., Cilk stack frames and deques).
- Implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).

Runtime library

- Manages the more heavy-weight data structures (e.g., the full-frame tree).
- Handles slow paths of execution (e.g., when a steal occurs).
SPAWNS AND STEALS: DEQUES & CILK STACK FRAMES
Deque of Frames

Each Cilk worker maintains a deque of references to Cilk Stack frames* containing work available to be stolen.

*We’ll discuss what these references are in a few slides.
Spawn

When spawning, the current frame is pushed onto the bottom of the deque.
When spawning, the current frame is pushed onto the bottom of the deque.
When spawning, the current frame is pushed onto the bottom of the deque.
When spawning, the current frame is pushed onto the bottom of the deque.
When spawning, the current frame is pushed onto the bottom of the deque.
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.

[Diagram of a deque and a cactus stack showing frame pop from bottom]
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.
Return from Spawn

When returning from a spawn, the current frame is popped from the bottom of the deque.
When returning from a spawn, the current frame is popped from the bottom of the deque.
Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.
Stealing Frames

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.
Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.
Stealing Frames

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.

Some coordination is required.
Synchronizing Thieves and Workers

Cilk uses a **mutex** associated with each deque to perform synchronization.
Cilk uses a **mutex** associated with each deque to perform synchronization.
Cilk uses a mutex associated with each deque to perform synchronization.

Question: Is it more important to optimize the operations of workers or those of thieves?
Cilk uses a mutex associated with each deque to perform synchronization.

Question: Is it more important to optimize the operations of workers or those of thieves?

Answer: Operations of workers.
When a worker is about to return from a spawned function, it tries to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.
When a worker is about to return from a spawned function, it tries to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.

**Question:** Which case is more important to optimize?
When a worker is about to return from a spawned function, it tries to pop the stack frame from the tail of the deque. There are two possible outcomes:

1. If the pop succeeds, then the execution continues as normal.
2. If the pop fails, then the worker is out of work to do, and it becomes a thief and tries to steal.

**Question:** Which case is more important to optimize?

**Answer:** Case 1, successful pop.
The THE Protocol

Worker protocol

```c
void push() { tail++; }

bool pop() {
    tail--;
    if (head > tail) {
        tail++;
        lock(L);
        tail--;
        if (head > tail) {
            tail++;
            unlock(L);
        }
    }
    unlock(L);
}
return SUCCESS;
}
```

The worker and the thief coordinate using the THE protocol

Thief protocol

```c
bool steal() {
    lock(L);
    head++;
    tail--; 
    if (head > tail) {
        head--;
        unlock(L);
    }
    unlock(L);
    return SUCCESS;
}
```
The THE Protocol

Worker protocol

```c
void push() { tail++; }

bool pop() {
    tail--;  
    if (head > tail) {
        tail++; 
        lock(L); 
        tail--;  
        if (head > tail) {
            tail++; 
        lock(L); 
        unlock(L); 
        } 
        unlock(L); 
    }
    return SUCCESS;
}
```

Observation I: Synchronization is only necessary when the deque is almost empty.

Thief protocol

```c
bool steal() {
    lock(L); 
    head++; 
    tail--;  
    if (head > tail) {
        tail++; 
        unlock(L); 
        return FAILURE; 
    }
    unlock(L);  
    return SUCCESS;
}
```
The THE Protocol

Worker protocol

```c
void push() { tail++; }

bool pop() {
    tail--;  
    if (head > tail) {
        tail++;
        lock(L);
        tail--;  
        if (head > tail) {
            tail++;
            unlock(L);
            return FAILURE;
        }
        unlock(L);
    }
    return SUCCESS;
}
```

Observation II: The pop operation is more likely to succeed than fail.

Thief protocol

```c
bool steal() {
    lock(L);
    head++;
    tail--;  
    if (head > tail) {
        tail++;
        unlock(L);
        return FAILURE;
    }
    unlock(L);
    return SUCCESS;
}
```

© 2018–2023 MIT Algorithm Engineering Instructors
The THE Protocol

Worker protocol

```c
void push() { tail++; }

bool pop() {
    tail--;  
    if (head > tail) {
        tail++;  
        lock(L);  
        tail--;  
        if (head > tail) {
            tail++;  
            unlock(L);  
            return FAILURE;  
        }
        unlock(L);  
    }
    return SUCCESS;  
}
```

The Work–First Principle: Optimize the operations of workers.

Thief protocol

```c
bool steal() {
    lock(L);  
    head++;  
    if (head > tail) {
        head--;  
        unlock(L);  
        return FAILURE;  
    }
    unlock(L);  
    return SUCCESS;  
}
```
### The THE Protocol

#### Worker protocol

```c
void push() {
    lock(L);
    head++;
    unlock(L);
    return SUCCESS;
}

bool pop() {
    tail--;  
    if (head > tail) {
        tail++;  
        lock(L);
        tail--;  
        if (head > tail) {
            tail++;  
            unlock(L);
            return FAILURE;
        }
        unlock(L);
    }
    unlock(L);
    return SUCCESS;
}
```

#### Thief protocol

```c
bool steal() {
    lock(L);
    head++;  
    if (head > tail) {
        head--;  
        unlock(L);
        return FAILURE;
    }
    unlock(L);
    return SUCCESS;
}
```

**The Work–First Principle:** Optimize the operations of workers. Workers pop the deque *optimistically*…
The THE Protocol

**Worker protocol**

```c
void push() {
    tail--;  
    if (head > tail) {
        tail++;  
        lock(L);  
        tail--;  
        if (head > tail) {  
            tail++;  
            unlock(L);  
            return FAILURE;  
        }  
        unlock(L);  
    }
    return SUCCESS;
}
```

**Thief protocol**

```c
bool steal() {
    lock(L);  
    head++;  
    tail--;  
    if (head > tail) {  
        tail++;  
        unlock(L);  
        return FAILURE;  
    }  
    unlock(L);  
}
```

The Work–First Principle:
Optimize the operations of workers.

- Workers pop the deque **optimistically**...
- ...and only grab the deque’s lock if the deque appears to be empty.
Worker protocol

```c
void push() {
    tail--;  
    if (head > tail) {
        tail++; 
        lock(L); 
        tail--; 
        if (head > tail) {
            tail++; 
            unlock(L);  
        }  
        unlock(L); 
    }  
    return SUCCESS; 
}
```

```
bool pop() {
    tail--; 
    if (head > tail) {
        tail++;  
        lock(L);  
        if (head > tail) {
            tail++;  
            unlock(L);  
        }  
        unlock(L); 
    }  
    return SUCCESS; 
}
```

The Work–First Principle: Optimize the operations of workers. Workers pop the deque optimistically... and only grab the deque's lock if the deque appears to be empty.

Thief protocol

```c
bool steal() {
    lock(L); 
    head++;  
    if (head > tail) {
        head--; 
        unlock(L);  
    }  
    unlock(L);  
    return FAILURE; 
}
```

Thieves always grab the lock.
Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.
Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.
Successful Steal

Workers operate on the bottom of the deque, while thieves try to steal work from the top of the deque.
Successful Steal

Workers operate on the **bottom** of the deque, while thieves try to steal work from the **top** of the deque.

Need to set up the thief’s stack and processor state after a successful steal.
To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.

**Cilk code**

```c
x = cilk_spawn fib(n-1);
```

**Compiled pseudocode**

```c
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
// (continuation)
```
To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.

Cilk code

```c
x = cilk_spawn fib(n-1);
```

Compiled pseudocode

```c
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
// (continuation)
```
To save and restore processor state, the Cilk compiler allocates a local buffer in each frame that spawns.

Cilk code

\[ x = \text{cilk}_\text{spawn} \ fib(n-1); \]

Compiled pseudocode

```
BUFFER ctx;
SAVE_STATE(&ctx);
if (!setjmp(&ctx))
  x = fib(n-1);
// (continuation)
```

Buffer to store processor state.

Save processor state into \texttt{ctx}, and allow a worker to resume the continuation.
Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.
Deque References to Frames

Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.
Worker deques store references to the buffers in each frame, from which thieves can retrieve processor state.
SYNCS: THE FULL-FRAME TREE
A `cilk_scope` waits on child frames, not on workers.

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

Example:
```c
fib(4)
```

Can’t sync yet!
Cilk supports nested synchronization, where a frame waits only on its child subcomputations.

Waiting on 3 children.

Waiting on 2 children.

Waiting on 2 children.

Waiting on 3 children.
Cilk supports **nested synchronization**, where a frame waits only on its child subcomputations.

How does Cilk keep track of who’s waiting on whom?
The Cilk runtime maintains a tree of **full frames** to keep track of synchronization information.

Processors work on active frames. Other frames are **suspended**.

Each full frame corresponds with at least one function frame.
Full-Frame Data

To maintain the state of the running program, each full frame maintains:

- A **join counter** of the number of (unsynched) child frames.
- References to **parent** and **child** full frames.
- References into the corresponding **Cilk stack frames** on the cactus stack.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

![Diagram showing the tree structure with nodes P1, P2, and P3, and labeled with 'spawned', 'called', 'spawned', and 'called'].

Steal!
Let’s see how the tree structure is maintained.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

The thief steals the full frame and creates a new full frame for the victim.

The victim’s new full frame is a child of the stolen full frame.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.
Let’s see how the tree structure is maintained.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

A full frame suspends at a sync if it has outstanding child frames.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

A full frame suspends at a sync if it has outstanding child frames.
Maintaining the Full-Frame Tree

Let’s see how the tree structure is maintained.

A full frame suspends at a sync if it has outstanding child frames.
Let’s see how the tree structure is maintained.

A full frame suspends at a sync if it has outstanding child frames.
Let’s see how the tree structure is maintained.

A full frame suspends at a sync if it has outstanding child frames.
Question: If the program has ample parallelism, what do we expect typically happens when the program execution reaches the end of a `cilk_scope`?
**Question:** If the program has ample parallelism, what do we expect typically happens when the program execution reaches the end of a `cilk_scope`?

**Answer:** The executing function contains no outstanding spawned children.
**Question:** If the program has ample parallelism, what do we expect typically happens when the program execution reaches the end of a `cilk_scope`?

**Answer:** The executing function contains no outstanding spawned children.

How does the runtime optimize for this case?
Managing the Full-Frame Tree: Sync

Diagram showing the synchronization process among multiple processes (P) with spawned and called states.
A flags field in each Cilk stack frame maintains the frame’s status, which is set when stolen. Only stolen spawning frames need nontrivial sync.
Like `cilk_spawn`, a `cilk_scope` is compiled using `setjmp`, in order to save the processor’s state when the frame is suspended.

**Cilk code**
```
cilk_scope { ... };
```

**C pseudocode**
```
BUFFER ctx;
...
if (WAS_STOLEN)
  if (!setjmp(&ctx))
    __cilkrts_sync(&ctx);
```
Compiled Code for Sync

Like `cilk_spawn`, a `cilk_scope` is compiled using `setjmp`, in order to save the processor’s state when the frame is suspended.

Cilk code

```c
#pragma omp cilk_scope
{
  ... 
}
```

C pseudocode

```c
BUFFER ctx;
...
if (WAS_STOLEN)
  if (!setjmp(&ctx))
    __cilkrts_sync(&ctx);
```
Like `cilk_spawn`, a `cilk_scope` is compiled using `setjmp`, in order to save the processor’s state when the frame is suspended.

**Cilk code**

```c
cilk_scope { ... };
```

**C pseudocode**

```c
BUFFER ctx;
...
if (WAS_STOLEN)
  if (!setjmp(&ctx))
    __cilkrts_sync(&ctx);
```

- Same buffer as used for spawns.
- Call into the runtime to suspend the frame.
DESIGN CHOICES
The Work–First Principle

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk runtime system works to maintain high work–efficiency by abiding by the work–first principle:

Optimize for the *ordinary serial execution*, at the expense of some additional overhead in steals.
Division of Labor

The work–first principle guides the division of the Cilk runtime system between the compiler and the runtime library.

• The compiler implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).
• The runtime library handles slow paths of execution, e.g., when a steal occurs.
Division of Labor

The work–first principle guides the division of the Cilk runtime system between the compiler and the runtime library.

• The compiler implements optimized fast paths for execution of functions when no steals have occurred (i.e., no actual parallelism has been realized).

• The runtime library handles slow paths of execution, e.g., when a steal occurs.

Examples:
• The THE protocol
• The implementation of cilk_spawn and cilk_sync
• The organization of full frames vs Cilk stack frames
Classic randomized work–stealing: Steal from a randomly chosen victim and steal from the top of its deque.
Choice of Whom / What to Steal

Classic randomized work-stealing:
Steal from a randomly chosen victim and steal from the top of its deque.

• The random choice and stealing from top allow us to amortize the cost of steals against the span term.
Choice of Whom / What to Steal

Classic randomized work-stealing: Steal from a randomly chosen victim and steal from the top of its deque.

- The random choice and stealing from top allow us to amortize the cost of steals against the span term.
- Randomness also avoids contention.
Choice of Whom / What to Steal

Classic randomized work-stealing: Steal from a randomly chosen victim and steal from the top of its deque.

- The random choice and stealing from top allow us to amortize the cost of steals against the span term.
- Randomness also avoids contention.
- An old performance bug in the runtime: every worker had a random number generator initialized with the same seed, which leads to high contention because everyone chose the same sequence of victims.
Spawn Semantics

*Continuation-stealing (work-first):* execute the spawned child and prepare the continuation to be stolen.

```c
int foo(int n) {
    int x, y;
    cilk_scope {
        x = cilk_spawn bar(n);
        y = baz(n);
    }
    return x + y;
}
```
Spawn Semantics

**Continuation-stealing (work-first):** execute the spawned child and prepare the continuation to be stolen.

**Child-stealing (help-first):** push the spawned child onto the deque so it can be stolen and continue executing the spawning function. Pop off spawned children to execute when encountering a sync.

```c
int foo(int n) {
    int x, y;
    cilk_scope {
        x = cilk_spawn bar(n);
        y = baz(n);
    }
    return x + y;
}
```
Issues with Child–Stealing: Space

```cilk
    cilk_scope {
      for(int i=0; i<1000; i++) {
        cilk_spawn foo(i);
      }
    }
```

Child–stealing: create 1000 work items and push them onto the deque before start doing any work!

Continuation–stealing: work on the spawned iteration and let the rest of the loops to be stolen potentially.
Continuation–stealing:

- Bounded space utilization.
- Better work–efficiency.
- One–worker execution follows that of serial projection.
- For private caches, one can bound the cache misses during parallel executions.

Child–stealing:

- Potentially unbounded space utilization.
- Worse work–efficiency.
- One–worker execution does NOT follow that of serial projection.
- No proven bound on cache misses during parallel executions.
Continuation–stealing:

- Bounded space utilization.
- Better work–efficiency.
- One worker execution follows that of serial projection.
- For private caches, one can bound the cache misses during parallel executions.

Child–stealing:

- Potentially unbounded space utilization.
- Worse work–efficiency.
- One worker execution does NOT follow that of serial projection.
- No proven bound on cache misses during parallel executions.

*Only monsters steal children!*