LEcTure 13
The Cilk Runtime System
Tao B. Schardl
Recall: Cilk Programming

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

Serial matrix multiply

```
for (int i = 0; i < n; ++i)
    for (int k = 0; k < n; ++k)
        for (int j = 0; j < n; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Running time $T_S$.

Cilk matrix multiply

```
cilk_for (int i = 0; i < n; ++i)
    for (int k = 0; k < n; ++k)
        for (int j = 0; j < n; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Running time $T_P$ on $P$ processors.
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 scheduling algorithm is provably efficient.

```c
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```
Recall: Cilk Platform

This lecture: How does Cilk work?

Cilk source:

```c
int64_t fib(int64_t n) {
    if (n < 2) { return n; }
    else {
        int64_t x, y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x + y);
    }
}
```
The compiler and runtime library together implement the runtime system.

```c
int64_t fib(int64_t n) {
    if (n < 2) { return n; }
    else {
        int64_t x, y;
        x = cilk_spawn fib(n-1);
        y = fib(n-2);
        cilk_sync;
        return (x + y);
    }
}
```
What the Compiler Generates

**Cilk code**

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

**Cilk compiler**

```c
int foo(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    y = baz(n);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    int result = x + y;
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
    return result;
}
```

**C pseudocode of compiled result**

```c
void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    __cilkrts_pop_frame(&sf);
    __cilkrts_leave_frame(&sf);
}
```
Outline

• Required Functionality
• Performance Considerations
• Implementing a Worker Deque
• Spawning Computation
• Stealing Computation
• Synchronizing Computation
REQUIRED FUNCTIONALITY
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)

The computation dag unfolds dynamically.
P1
\texttt{\%rip} \hspace{1cm} \texttt{int fib (int n) \{\}
    \texttt{if (n < 2) return n;\texttt{\}}
    \texttt{else \{\texttt{\}}}
        \texttt{int x, y;\texttt{\}}
        \texttt{x = cilk\_spawn fib(n-1); \texttt{\}}
        \texttt{y = fib(n-2); \texttt{\}}
        \texttt{cilk\_sync; \texttt{\}}
        \texttt{return x + y; \texttt{\}}
\texttt{\}\texttt{\}}
\texttt{\}}
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;
    }
}
Parallel Execution: Syncs

Example:

```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;
    }
}
```

How does a `cilk_sync` wait only on nested subcomputations?
**Required Functionality**

- A single worker must be able to execute the computation on its own similarly to an ordinary **serial computation**.
- A thief must be able to jump into the middle of an executing function to **steal a continuation**.
- A sync must stall a function’s execution until **child subcomputations** complete.

**What other functionality is needed?**
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.

**Views of stack**

Cilk’s **cactus stack** supports multiple views in parallel.
Recall: Work Stealing

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 deque contains a mixture of spawned frames and called frames.
Recall: Work Stealing

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

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!
Recall: Work Stealing

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.
Recall: Work Stealing

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

A steal takes all frames up to the next spawned frame.

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

© 2008–2018 by the MIT 6.172 Lecturers
Recall: Work Stealing

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.

What is involved in stealing frames?

- What synchronization is needed?
- What happens to the stack?
- How efficient can this be?
Required Functionality

- A single worker must be able to execute the computation on its own similarly to an ordinary *serial computation*.
- A thief must be able to jump into the middle of an executing function to *steal a continuation*.
- A `sync` must stall a function’s execution until `child subcomputations` complete.
- The runtime must implement a *caactus stack* for its parallel workers.
- Thieves must be able to handle *mixtures of called spawned functions*.
Performance Considerations
Theorem [BL94]. 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 achieves linear speedup, then workers spend most of their time working.
Ideally, parallelizing a serial code makes it run \( P \) times faster on \( P \) processors.

**Serial matrix multiply**

```
for (int i = 0; i < n; ++i)
    for (int k = 0; k < n; ++k)
        for (int j = 0; j < n; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Running time \( T_S \).

**Cilk matrix multiply**

```
cilk_for (int i = 0; i < n; ++i)
    for (int k = 0; k < n; ++k)
        for (int j = 0; j < n; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

With sufficient parallelism, running time is \( T_P \approx T_1/P \).

Goal: \( T_P \approx T_S/P \), meaning that \( T_S \approx T_1 \).
Let $T_S$ denote the work of a serial program. Suppose the serial program is parallelized. Let $T_1$ denote the work of the parallel program, and let $T_\infty$ denote the span of the parallel program.

To achieve linear speedup on $P$ processors over the serial program, i.e., $T_P \approx T_S/P$, the parallel program must exhibit:

- Ample parallelism: $T_1/T_\infty \gg P$.
- High work efficiency: $T_S/T_1 \approx 1$. 

© 2008–2018 by the MIT 6.172 Lecturers
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 computation in steals.
Division of Labor

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

Compiler
- Uses a handful of small data structures, e.g., workers and stack frames.
- Implements optimized fast paths for execution of functions when no steals have occurred.

Runtime library
- Uses larger data structures.
- Handles slow paths of execution, e.g., when a steal occurs.
IMPLEMENTING A WORKER DEQUE
Function `foo` is a **spawning function**, meaning that `foo` contains a `cilk_spawn`.  
Function `bar` is **spawned** by `foo`.  
The call to `baz` occurs in the **continuation** of the spawn.
**Problem:** How do we implement a worker’s deque?

- The worker should operate its own deque like a stack.
- A steal needs to transfer ownership of several consecutive frames to a thief.
- A thief needs to be able to resume a continuation.
IDEA: The worker deque is an external structure with pointers to stack frames.

- A Cilk worker maintains head and tail pointers to its deque.
- Stealable frames maintain a local structure to store information necessary for stealing the frame.
The Intel Cilk Plus runtime elaborates on this idea as follows:

- Every spawned subcomputation runs in its own `spawn-helper` function.
- The runtime maintains three basic data structures as workers execute work:
  - A `worker structure` for every worker used to execute the program.
  - A Cilk stack-frame structure for each instantiation of a spawning function.
  - A `spawn-helper stack frame` for each instantiation of a `cilk_spawn`.
Spawn–Helper Functions

Cilk code

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

Cilk compiler

C Cilk pseudocode of compiled result

```c
int foo(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    y = baz(n);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    int result = x + y;
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
    return result;
}

void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    __cilkrts_pop_frame(&sf);
    __cilkrts_leave_frame(&sf);
}
```
Cilk Stack–Frame Structures

Cilk stack–frame structures

Cilk pseudocode of compiled result

C Cilk stack–frame structures

C pseudocode of compiled result

```c
int foo(int n) {
  __cilkrts_stack_frame_t sf;
  __cilkrts_enter_frame(&sf);
  int x, y;
  if (!setjmp(sf.ctx))
    spawn_bar(&x, n);
  y = baz(n);
  if (sf.flags & CILK_FRAME_UNSYNCHED)
      if (!setjmp(sf.ctx))
        __cilkrts_sync(&sf);
  int result = x + y;
  __cilkrts_pop_frame(&sf);
  return result;
}
```

```c
void spawn_bar(int *x, int n) {
  __cilkrts_stack_frame sf;
  __cilkrts_enter_frame_fast(&sf);
  __cilkrts_detach();
  *x = bar(n);
  __cilkrts_pop_frame(&sf);
  __cilkrts_leave_frame(&sf);
}
```
Each Cilk stack frame stores:

- A context buffer, `ctx`, which contains enough information to resume a function at a continuation, i.e., after a `cilk_spawn` or `cilk_sync`.
- An integer, `flags`, that summarizes the state of the Cilk stack frame.
- A pointer, `parent`, to its parent Cilk stack frame.
The Cilk Worker Structure (Simplified)

Each Cilk worker maintains:

- A **deque** of stack frames that can be stolen.
- A pointer to the **current stack frame**.

Example:
Function `foo` spawned `bar`, which called `quux`, which spawned `fred`.

© 2008–2018 by the MIT 6.172 Lecturers
SPAWNING COMPUTATION
C pseudocode of a spawning function

```c
int foo(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    y = baz(n);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    int result = x + y;
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
    return result;
}
```

- Create and initialize a Cilk stack-frame structure.
- Prepare to spawn.
- Invoke the spawn helper.
- Perform a sync.
- Clean up the Cilk stack-frame structure.
- Clean up the deque.
C pseudocode of a spawn helper

```c
void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    __cilkrts_pop_frame(&sf);
    __cilkrts_leave_frame(&sf);
}
```

Create and initialize a Cilk stack-frame structure.

Update the deque to allow the parent to be stolen.

Invoke the spawned subroutine.

Clean up the Cilk stack-frame structure.

Clean up the deque and attempt to return.

© 2008–2018 by the MIT 6.172 Lecturers
When execution enters a spawning function, the Cilk worker’s current stack-frame structure is updated.
Preparing to Spawn

Cilk code

```c
int foo(int n) {
    ...
    x = cilk_spawn bar(n);
    ...
}
```

Cilk uses the `setjmp` function to allow thieves to steal the continuation.

The `setjmp` function stores information necessary for resuming the function at the `setjmp` into the given buffer.

**QUESTION:** What information needs to be saved?

**ANSWER:** Registers `%rip`, `%rbp`, `%rsp`, and callee-saved registers.
Spawning a Function

C pseudocode

```c
int foo(int n) {
    ...
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    ...
}
void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    ...
}
```
Spawning a Function

C pseudocode

```c
int foo(int n) {
    ...
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    ...
}
void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame_sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    ...
}
```

Deque

Call stack

© 2008–2018 by the MIT 6.172 Lecturers
Spawning a Function

C pseudocode

```c
int foo(int n) {
    ...
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    ...
}

void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    ...
}
```

© 2008–2018 by the MIT 6.172 Lecturers
Spawning a Function

C pseudocode

```c
int foo(int n) {
    ...
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    ...
}

void spawn_bar(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = bar(n);
    ...
}
```

Cilk worker

Deque

Call stack

© 2008–2018 by the MIT 6.172 Lecturers
Returning from a Spawn

Cilk worker
- head
- tail
- current_sf

Deque

Call stack
- main
- foo
- foo_sf
- spawn_bar
- spawn_bar_sf
- bar

C pseudocode

```c
void spawn_bar(int *x, int n) {
    ...
    *x = bar(n);
    __cilkrts_pop_frame(&sf);
    __cilkrts_leave_frame(&sf);
}
```
Returning from a Spawn

Cilk worker
- head
- tail
- current_sf

Deque

Call stack
- main
- foo
- foo_sf
- spawn_bar
- spawn_bar_sf

C pseudocode

```c
void spawn_bar(int *x, int n) {
    ...
    *x = bar(n);
    __cilkrts_pop_frame(&sf);
    __cilkrts_leave_frame(&sf);
}
```

© 2008–2018 by the MIT 6.172 Lecturers
Returning from a Spawn

Cilk worker
- head
- tail
- current_sf

Deque

Call stack
- main
- foo
- foo_sf
- spawn_bar
- spawn_bar_sf

C pseudocode

```c
void spawn_bar(int *x, int n) {
    ...
    *x = bar(n);
    __cilkrts_pop_frame(&sf);
    __cilkrts_leave_frame(&sf);
}
```

May or may not return, depending on what’s in the worker’s deque.
In `__cilkrts_leave_frame`, the worker 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. It thus becomes a thief and tries to steal work from the top of a random victim’s deque.

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

**Answer:** Case 1.
Stealing Computation
Recall: Stealing Work

Conceptually, a thief takes frames off of the top of a victim worker’s deque.
A thief steals from the head of the victim worker’s deque. Need to handle concurrent accesses to the deque!
Synchronizing Deque Accesses

Worker protocol

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

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

The worker only grabs a lock if the deque appears to be empty.

The worker pops the optimistically.

Thief protocol

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

The thief always grabs a lock before operating on the deque.
Resuming a Continuation

Cilk uses the `longjmp` function to resume a stolen continuation.

C pseudocode

```c
int foo(int n) {
    ...
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    ...
}
```

Previously, the victim performed a `setjmp` to store register state in `foo_sf.ctx`.

Executing `longjmp(current_sf->ctx, 1)` sets the thief’s registers to start executing at the location of the `setjmp`.

Victim’s Call stack

- main
- foo
- foo_sf
Resuming a Continuation

The contract between `setjmp` and `longjmp` ensures the thief resumes the continuation.

- On its direct invocation, `setjmp(buffer)` returns 0.
- Invoking `longjmp(buffer, x)` causes the `setjmp` to effectively return again with the integer value x.

```
C pseudocode
int foo(int n) {
    ...
    if (!setjmp(sf.ctx))
        spawn_bar(&x, n);
    ...
}
```

Because a thief reaches this point by calling `longjmp(current_sf->ctx,1)`, the condition fails, and the thief jumps to the continuation.
Implementing the Cactus Stack

Thieves maintain their own call stacks and use pointer tricks to implement the cactus stack.

Example: A thief steals the continuation of `foo`, and then calls `baz`.

Thief’s `%rbp`  

Victim’s call stack

```
main
foo
foo_sf
spawn_bar
spawn_bar_sf
bar
```
Synchronizing Computation
Recall: Nested Synchronization

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

**Example:**
```
fib(4)
```

Synchronization happens in a **nested** fashion.
If a worker reaches a `cilk_sync` before all spawned subcomputations are complete, the worker should become a **thief**, but the worker’s current function frame should not disappear!

- The existing subcomputations might access **state** in that frame, which is their parent frame.
- In the future, another worker must **resume** that frame and execute the `cilk_sync`.
- The `cilk_sync` only applies to nested subcomputations of the frame, **not** to all subcomputations or workers.
The Cilk runtime maintains a tree of full frames, which stores state for parallel subcomputations.

A full frame keeps track of its parent and child frames.

Other full frames are suspended.

Processors work on active full frames.
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.

The thief steals the full frame and creates a new full frame for the victim.
Let’s see how steals can produce a tree of full frames.

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

© 2008-2018 by the MIT 6.172 Lecturers
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.

*Full-frame illustrations resized for cleanliness.

© 2008-2018 by the MIT 6.172 Lecturers
Let’s see how steals can produce a tree of full frames.
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.

By stealing the full frame, existing pointers to parent full frame are preserved.

*Full-frame illustrations resized for cleanliness.

© 2008–2018 by the MIT 6.172 Lecturers
Let's see how steals can produce a tree of full frames.
Let’s see how steals can produce a tree of full frames.

Maintaining Full Frames

Steal!
Maintaining Full Frames

Let’s see how steals can produce a tree of full frames.

*Full-frame illustrations resized for cleanliness.

© 2008–2018 by the MIT 6.172 Lecturers
Suspending Full Frames

The full frame cannot sync because of the running child subcomputation.
Suspending Full Frames

Suspending!
Suspending Full Frames

Steal!
**QUESTION:** If the program has ample parallelism, what do we expect will typically happen when the program execution reaches a `cilk_sync`?

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

How does the runtime optimize for this case?
Every full frame is associated with a Cilk stack frame.

The flags field in the Cilk stack frame maintains the full frame’s status.
The code compiled to implement a `cilk_sync` checks the flags field before performing an expensive call to `__cilkrts_sync` in the Cilk runtime library.
More Cilk Runtime Features

The Cilk runtime system implements many other features and optimizations:
- Schemes for making the full-frame tree simpler and easier to maintain.
- Data structure and protocol enhancements to support C++ exceptions.
- Sibling pointers between full frames to support reducer hyperobjects.
- Pedigrees to assign a unique, deterministic ID to each strand efficiently in parallel.
THE TAPIR COMPILER
[SML17]
Question: When do `cilk_spawn`, `cilk_sync`, and `cilk_for` get compiled?
Example: Normalize

```c
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y, 
    const double *restrict X, 
    int n) {
    for (int i = 0; i < n; ++i)
        Y[i] = X[i] / norm(X, n);
}
```

**Test:** Random vector of \( n = 64M \) elements

**Machine:** AWS c4.8xlarge  
**Compiler:** GCC 6.2

**Running time:** \( T_s = 0.312 \) s
Optimizing the Serial Code

__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict X, const double *restrict Y, int n) {
    for (int i = 0; i < n; ++i)
        Y[i] = X[i] / norm(X, n);
}

The `norm` function performs $\Theta(n)$ work.

GCC can move the call to `norm` out of the serial loop.

Work before hoisting: $T(n) = \Theta(n^2)$

Work after hoisting: $T(n) = \Theta(n)$
GCC Compiling Cilk Code

Cilk code

```c
void normalize(double *restrict Y,
    const double *restrict X, int n) {
    cilk_for (int i = 0; i < n; ++i)
        Y[i] = X[i] / norm(X, n);
}
```

Helper function encodes the loop body.

Call into Cilk runtime library to execute a `cilk_for` loop.

The compiler can’t move `norm` out of the loop.

Cilk

pseudo-code

```c
void normalize(double *restrict Y,
    const double *restrict X, int n) {
    struct args_t args = { Y, X, n };
    _cilkrts_cilk_for(normalize_helper, args, 0, n);
}
```

```c
void normalize_helper(struct args_t *args, int i) {
    double *Y = args.Y;
    double *X = args.X;
    int n = args.n;
    Y[i] = X[i] / norm(X, n);
}
```
**Performance of Parallel Normalize**

```c
__attribute__((const))
double norm(const double *x) {
    double sum = 0;
    for (int i = 0; i < n; i++)
        sum += x[i] * x[i];
    return sqrt(sum);
}

void normalize(double *restrict Y, const double *restrict X, int n) {
    cilk_for (int i = 0; i < n; i++)
        Y[i] = X[i] / norm;
}
```

The norm function was also parallelized.

**Test:** Random vector of \( n=64M \) elements

**Machine:** AWS c4.8xlarge  
**Compiler:** GCC 6.2

**Running time of serial code:** \( T_S = 0.312 \) s

**18-core running time:** \( T_{18} = 180.657 \) s

**1-core running time:** \( T_1 = 2600.287 \) s

Terrible work efficiency! \( \frac{T_S}{T_1} = \frac{0.312}{2600} \approx \frac{1}{8600} \)
Tapir embeds fork-join parallelism into LLVM’s IR.
Unsafe Optimizations

Problem: There are many examples of optimizations on serial code that cannot be safely applied to parallel code [MP90].

Cilk code

```c
void foo(int n) {
    cilk_for (int i = 0; i < n; ++i)
        bar(5*i);
}
```

Incorrectly optimized Cilk code

```c
void foo(int n) {
    int tmp = 0;
    cilk_for (int i = 0; i < n; ++i) {
        bar(tmp);
        tmp += 5;
    }
}
```

Unleashing LLVM on parallel programs requires some care.
Tapir uses an asymmetric representation of the parallel tasks in the CFG.

Tapir adds three constructs to LLVM’s IR: detach, reattach, and sync.

```c
int foo(int n) {
    int x, y;
    x = cilk_spawn bar(n);
    y = baz(n);
    cilk_sync;
    return x + y;
}
```
If the program contains no determinacy races, then it is semantically equivalent to its serial projection.
In Tapir, parallel loops look similar to serial loops, with some differences due to parallelism.

**Serial normalize CFG**

- `br (0 < n), loop, exit`

  1. `i0 = \( \phi([0, \text{entry}],[i1, \text{loop}]) \)`
  2. `\text{norm0} = \text{norm}(X, n)`
  3. `Y[i0] = X[i0] / \text{norm0}`
  4. `i1 = i0 + 1`
  5. `br (i1 < n), loop, exit`

**Tapir normalize CFG**

- `br (0 < n), header, exit`

  1. `i0 = \( \phi([0, \text{entry}],[i1, \text{latch}]) \)`
  2. `\text{detach} \text{ body, latch}`
  3. `\text{norm0} = \text{norm}(X, n)`
  4. `Y[i0] = X[i0] / \text{norm0}`
  5. `\text{reattach} \text{ latch}`
  6. `i1 = i0 + 1`
  7. `br (i1 < n), header, sync`

- `\text{sync} \text{ exit}`

  - `return`
Impact on LLVM

LLVM can reason about a Tapir CFG as a relatively minor change to the CFG of the serial elision.

• Many standard compiler analyses required no changes.
• Memory analysis required a minor change to handle Tapir’s constructs (~450 lines of code).
• Some optimizations, e.g., code hoisting and tail-recursion elimination, required some changes to work on Tapir CFG’s.

In total, implementing Tapir involved adding or modifying ~6000 lines of LLVM’s 4–million–line codebase.
Parallelize Normalize with Tapir

```c
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y,
               const double *restrict X,
               int n) {
    cilk_for (int i = 0; i < n;
        Y[i] = X[i] / norm(X);
}
```

Good work efficiency: \( \frac{T_S}{T_1} = 97\% \)

**Test:** Random vector of \( n=64 \)M elements

**Machine:** AWS c4.8xlarge  
**Compiler:** Tapir/LLVM

**Running time of serial code:** \( T_S = 0.312 \) s

1-core running time: \( T_1 = 0.321 \) s

18-core running time: \( T_{18} = 0.081 \) s
Same as Tapir/LLVM except that Cilk constructs are compiled early.

Tapir/LLVM doesn’t fix everything, but it helps parallel programs achieve good work efficiency.
Case Study: OpenMP
Example: OpenMP Normalize

Test: Random vector of n=64M elements
Machine: AWS c4.8xlarge    Compiler: GCC 6.2
Running time of serial code: \( T_S = 0.312 \text{ s} \)
1-core running time: \( T_1 = 0.329 \text{ s} \)
18-core running time: \( T_{18} = 0.205 \text{ s} \)

Why do we get this performance?

Parallel speedup is not great.

Good work efficiency without Tapir?

The \texttt{norm} function was also parallelized.

```
__attribute__((const))
double norm(const double *x)
{
    double sum = 0;
    for (int i = 0; i < n; ++i)
        sum += x[i] * x[i];
    return sqrt(sum);
}

void normalize(double *restrict Y,
                const double *restrict X,
                int n) {
    double Xnorm = norm(X, n);
    for (int i = 0; i < n; ++i)
        Y[i] = X[i] / Xnorm;
}
```
Each processor invokes this helper method on \( n/P \) iterations.

The helper function's loop on \( n/P \) iterations can be optimized.
The norm function performs $\Theta(n)$ work.

Each processor invokes `ompOutlined` on $n/P$ iterations.

The variable `local_n` is approximately $n/P$.

Work of `ompOutlined`:

Total work on $P$ processors:
__attribute__((const))
double norm(const double *X, int n);

void normalize(double *restrict Y, 
    const double *restrict X, 
    int n) {
#pragma omp parallel for
    for (int i = 0; i < n; ++i)
        Y[i] = X[i] / norm(X, n);
}

- This code is only work efficient on 1 processor.
- This code can never achieve more than minimal parallel speedup.

Work on P processors: 
T(n) = Θ(Pn)
Takeaways

The work-first principle

Optimize for ordinary serial execution, at the expense of some additional computation in steals.

Two more takeaways:

- Think about the performance model for your program.
- Know what your parallel runtime system is doing.
6.172 Performance Engineering of Software Systems
Fall 2018

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