A Quick Introduction To The Intel Cilk Plus Runtime

6.S898: Advanced Performance Engineering for Multicore Applications
March 8, 2017

Adapted from slides by Charles E. Leiserson, Saman P. Amarasinghe, and Jim Sukha
Cilk Language Constructs

Cilk extends C and C++ with three keywords to expose task parallelism: cilk_spawn, cilk_sync, and cilk_for.

Cilk Fibonacci code

```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 child function is **spawned**: It is allowed (but not required) to execute in parallel with the parent caller.

Control cannot pass this point until the function is **synched**: all spawned children have returned.
Simple Cilk Example: Fib

How is a Cilk program compiled and executed in parallel?

1. The **compiler** takes program and generates assembly with calls to the Cilk Plus runtime library, libcilkrts.so.

2. When executing a program, the **runtime library** is dynamically loaded and handles scheduling of the program on multiple worker threads.

Cilk Fibonacci code

```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;
}
```
What Do the Compiler and Runtime Do?

Cilk Fibonacci code

```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;
}
```

Compiler

```c
int fib(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    if (n < 2) return n;
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_fib(&x, n);
    y = fib(n-2);
    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;
}
```

Today: Dive into some of this code.

Cilk runtime scheduler

```c
// static cilk_fiber* worker_scheduling_loop_body(cilk_fiber* current_fiber, void* wptr)
{
    __cilkrts_worker *w = (__cilkrts_worker*) wptr;
    CILK_ASSERT(current_fiber == w->l->scheduling_fiber);
    // Stage 1: Transition from executing user code to the runtime code.
    // We don't need to do this call here any more, because
    // every switch to the scheduling fiber should make this call
    // using a post_switch_proc on the fiber.
    // enter_runtime_transition_proc(w->l->scheduling_fiber, wptr);
    // After Stage 1 is complete, w should no longer have
    // an associated full frame.
    CILK_ASSERT(NULL == w->l->frame_ff);
    // Stage 2.  First do a quick check of our 1-element queue.
    full_frame *ff = pop_next_frame(w);
    if (!ff) {
        // Stage 3.  We didn't find anything from our 1-element
        // queue.  Now go through the steal loop to find work.
        ff = search_until_work_found_or_done(w);
    }
    if (ff) {
        // Stage 3.  We didn't find anything from our 1-element
        // queue.  Now go through the steal loop to find work.
        full_frame *ff = pop_next_frame(w);
        if (!ff) {
            // Stage 3.  We didn't find anything from our 1-element
            // queue.  Now go through the steal loop to find work.
            ff = search_until_work_found_or_done(w);
        }
        __cilkrts_pop_frame(&sf);
        if (sf.flags)
            __cilkrts_leave_frame(&sf);
    }
}
```

```c
void spawn_fib(int *x, int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame_fast(&sf);
    if (n < 2) return n;
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_fib(&x, n);
    y = fib(n-2);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    *x = fib(n-1);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```
Organization of Runtime Source Code

The Intel Cilk Plus runtime source code is available online: https://bitbucket.org/intelcilkruntime/intel-cilk-runtime

- Basic data structures: include/internal/abi.h
- Compiler-inserted runtime calls: runtime/cilk-abi.c
- Runtime data structures: runtime/full_frame.h, runtime/full_frame.c, runtime/local_state.h
- Heart of the Cilk Plus scheduler: runtime/scheduler.c
Outline

- Review of randomized work stealing
- Compiler and runtime internals
  - Fast path: executing with no steals
  - Data structures for steals
- Steals: the ugly details
Randomized Work Stealing: Working

Each worker maintains a work deque of ready strands, and it manipulates the bottom of the deque like a stack \[\text{MKH90, BL94, FLR98}\].
Randomized Work Stealing: Working

Each worker 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
spawn
call
```

```
spawn
call
```

```
spawn
```

Spawn!
Randomized Work Stealing: Working

Each worker maintains a **work deque** of ready strands, and it manipulates the bottom of the deque like a stack [MKH90, BL94, FLR98].
Randomized Work Stealing: Working

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

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

When a worker runs out of work, it becomes a thief and steals from the top of a random victim’s deque.
Randomized Work Stealing: Stealing

When a worker runs out of work, it becomes a thief and steals from the top of a random victim’s deque.
Randomized Work Stealing: Stealing

When a worker runs out of work, it becomes a thief and steals from the top of a random victim’s deque.
Work-Stealing Bounds

**Theorem** [BL94]: The Cilk work-stealing scheduler achieves expected running time

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

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 \( O(PT_\infty) \). Because there are \( P \) processors, the expected running time is

\[ (T_1 + O(PT_\infty)) / P = T_1 / P + O(T_\infty) \].
Work-Stealing Bounds

**Theorem** [BL94]: The Cilk work-stealing scheduler achieves expected running time

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

on \( P \) processors.
The Work-First Principle

Corollary [BL94]: A program with sufficient parallelism satisfies $\frac{T_1}{P} \gg O(T_\infty)$, meaning that workers steal infrequently and the program exhibits linear speedup.

To optimize the execution of programs with sufficient parallelism, the implementation of the Cilk runtime system abides by work-first principle: Optimize for ordinary serial execution, at the expense of some additional computation in steals.
The work-first principle guides the division of the runtime-system implementation between the compiler and the runtime library.

The 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.

The runtime library:

- Handles slow paths of execution, i.e., when a steal occurs.
- Uses data structures that are generally larger.
Cactus Stack

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.
Outline

- Review of randomized work stealing
- Compiler and runtime internals
  - Fast path: executing with no steals
  - Data structures for steals
- Steals: the ugly details
Our Running Example

Example Cilk code

```c
int f(int n) {
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```

- **Function f** is a **spawning function**.
- **Function g** is a **spawned** by **f**.
- The call to **h** occurs in the **continuation** of **cilk_spawn g()**.
Example Cilk code

```c
int f(int n) {
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```

Source: Tapir/LLVM compiler source code, lib/Transforms/Tapir/CilkABI.cpp.
Basic Data Structures

The Cilk Plus runtime maintains three basic data structures as workers execute work.

- Cilk Plus maintains a **worker** structure for every worker used to execute a program.
- Cilk Plus creates a **Cilk stack frame** to represent each **spawning function** — each function that contains a `cilk_spawn`.
- Cilk Plus creates a **spawn-helper (stack) frame** for each instance of a `cilk_spawn` that executes.
Our Running Example

```c
int f(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_g(&x, n);
    y = h(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_g(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = g(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```
Cilk Stack Frame

Each Cilk stack frame stores:

- A **context buffer**, which contains enough information to resume a function at a continuation, i.e., after a spawn or sync.
- A pointer to its **parent** Cilk stack frame.

Source: runtime library source code, `include/internal/abi.h`
Basic Worker Data Structure

For each worker $w$, the Cilk runtime system maintains:

- A chain of **Cilk stack frames**. The end of the chain is $w->current_stack_frame$.
- A **deque** of pointers to Cilk stack frames, with $w->head$ and $w->tail$ pointers.

Each worker also operates on its own ordinary **execution stack**, which stores normal frame data, e.g., local variables of the function.

Source: runtime library source code, include/internal/abi.h.
Calling a Function That Spawns

A call to $f$ does the following.

1. Update the execution stack as normal.
2. Creates a Cilk stack frame, $f_{sf}$, on the execution stack.
3. Pushes $f_{sf}$ onto the chain of Cilk stack frames.
Spawning a Function g from f

Spawning g from f involves 5 steps:

1. Save the continuation of f in the Cilk stack frame.
2. Call the spawn-helper function and initialize its Cilk stack frame, g_hf.
3. Evaluate the arguments of g, calling any necessary C++ constructors.
4. Mark g_hf as detached and push its parent f_sf onto the deque.
5. Call function g.

```c
int f(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_g(&x, n);
    y = h(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_g(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = g(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```
The setjmp and longjmp Instructions

The Cilk runtime uses setjmp and longjmp to suspend and resume the execution of functions.

- **setjmp**: Save the current execution context in a specified buffer.
- **longjmp**: Restore the current execution context from the specified buffer.

The setjmp instruction returns 0 or 1, depending on whether it’s reached by normal execution or by a longjmp.

```c
int f(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_g(&x, n);
    y = h(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_g(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = g(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```
The content of the jump buffer that setjmp and longjmp use depends on the architecture and operating system.

- On Linux and x86_64, this buffer just stores a few registers: the program counter, the stack pointer, the base pointer, and callee-saved registers.
- The Cilk runtime library ensures that other state (e.g., the execution stack) is maintained.
Example: `cilk_spawn` of `g`

```c
int f(int n)
    __cilkrts_stack_frame sf;
    __cilkrts_detach();
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_g(&x, n);
    y = h(n);
    if (sf.flags & CILK_FRAME_UNSYNCHED)
        if (!setjmp(sf.ctx))
            __cilkrts_sync(&sf);
    int result = x + y;
    __cilkrts_pop_frame(&sf);
    return result;
}

void spawn_g(int *x, int n)
    __cilkrts_stack_frame sf;
    __cilkrts_detach();
    *x = g(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```

**Save state of f into f_sf and call the spawn helper.**

**Create spawn-helper Cilk stack frame, g_hf.**

**Mark g_hf as detached, and push f_sf onto deque.**

**Call g.**
Example: Return From cilk_spawn of g

Returning from a spawned function g involves 5 steps:

1. Return from g.
2. Copy the return value of g.
3. Call C++ destructors for any computed temporaries.
4. Undo the detach of the Cilk stack frame.
5. Leave the spawn-helper function.

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

void spawn_g(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = g(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```
Example: Return From a `cilk_spawn`

```c
int f(int n) {
    __cilkrts_stack_frame_t sf;
    __cilkrts_enter_frame(&sf);
    int x, y;
    if (!setjmp(sf.ctx))
        spawn_g(&x, n);
    y = h(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_g(int *x, int n) {
    __cilkrts_stack_frame sf;
    __cilkrts_enter_frame_fast(&sf);
    __cilkrts_detach();
    *x = g(n);
    __cilkrts_pop_frame(&sf);
    if (sf.flags)
        __cilkrts_leave_frame(&sf);
}
```

**Return from g.**

**Pop g_hf from the chain of Cilk stack frames.**

**Attempt to remove f_sf from the deque.**

**But we need to check if the parent was stolen!**
The Cilk runtime system implements the **THE protocol** to synchronize updates to the deque. (See `runtime/scheduler.c`.)

**Pseudocode for the simplified THE protocol:**

- Speculatively decrement `tail` for the common case.
- If the deque looks empty, lock the deque and try again.
- The deque really is empty, meaning the parent continuation was stolen.

**Worker/Victim**

```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;
}
```

**Thief**

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

The thief always locks the deque.
Result of `__cilkrts_leave_frame()`

There are two possible outcomes from calling `__cilkrts_leave_frame`:

a) **Fast path:** If the continuation in `f` was not stolen then `__cilkrts_leave_frame` returns normally.

b) **Slow path:** Otherwise, control jumps into the runtime library.
The execution of a `cilk_sync` branches based on whether the function has synched.

- If so, then execution continues normally.
- Otherwise, the continuation of the `cilk_sync` is saved, and `__cilkrts_sync()` is called to transfer control into the runtime.
Returning From a Function That Spawns

When the spawning function returns, \_\_cilkrts\_leave\_frame is called to remove its Cilk stack frame.

No need to update the deque if the function did not detach.
Where are these routines implemented?

- The compiler implements and inlines `enter_frame`, `enter_frame_fast`, `detach`, and `pop_frame`.
- The runtime library implements `__cilkrts_sync` and `__cilkrts_leave_frame`. (See runtime/cilk-abi.c.)
Outline

- Review of randomized work stealing
- Compiler and runtime internals
  - Fast path: executing with no steals
  - Data structures for steals
  - Steals: the ugly details
Parallel Execution Stacks

Two workers executing a spawned routine and its continuation in parallel use distinct execution stacks.

Example Cilk code

```c
int f(int n) {
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```

Execution stack for worker w0

- main
- f
- f_sf
- spawn_g
- g_hf
- g

Execution stack for worker w1

- h
Accessing The Parent Stack Frame

After stealing, a worker can access state in its parent’s stack via a separate pointer.

Example Cilk code

```c
int f(int n) {
    int x, y;
    x = cilk_spawn g(n);
    y = h(n);
    cilk_sync;
    return x + y;
}
```

Execution stack for worker w0

- `main`
- `f`
- `f_sf`
- `spawn_g`
- `g_hf`
- `g`

Execution stack for worker w1

- `h`

w1->rsp
w1->rbp
Stalling

Execution on a stack **stalls** if the worker discovers its deque to be empty.

**Example:** Worker \( w_0 \) returns from `cilk_spawn` of `g`:

Worker \( w_0 \) can’t pop its deque!

Worker \( w_0 \) wants to start stealing, but worker \( w_1 \) needs the state on \( w_0 \)’s stack!
Full Frames

The Cilk Plus runtime system maintains full frames to keep track of executing and stalled function frames.

- A full frame has an associated Cilk stack frame, as well as a lock, a join counter, and other fields.
- Every worker that is executing user code has an active full frame.
- Other full frames are suspended.
The Full Frame Tree

Full frames are connected together in a full frame tree.

- Each full frame maintains parent, right-sibling, and left-child pointers.
- The tree structure reflects the relationship between stack frames.
- Busy leaves property: All leaves of the full frame tree are active full frames.
Workers and Full Frames

Each worker executing user code tracks its full frame in the field `l->frame_ff`.

- That full frame points to the oldest Cilk stack frame associated with this worker.
- The worker’s pointer to its full frame is local state associated with the worker that the compiler doesn’t care about.
Source Code for Full Frames

- The full frame data structure is defined in the runtime library, in runtime/full_frame.h and runtime/full_frame.c.
- The local state associated with a worker is defined in the runtime library, in runtime/local_state.h and runtime/local_state.c.
Outline

- Review of randomized work stealing
- Compiler and runtime internals
  - Fast path: executing with no steals
  - Data structures for steals
- Steals: the ugly details
The stealing algorithm is implemented in the runtime library, in runtime/scheduler.c.

- The method `random_steal()` implements random selection of a victim and the THE protocol for the thief.
- Management of full frames to execute a steal (i.e., “the ugly details”) is implemented in `detach_for_steal()`.
Target of a Steal

When a thief worker \( w_1 \) steals from a victim worker \( w_0 \), it steals a **chain** of stack frames.
Steps to Perform a Steal

A thief steals a continuation from a victim in 5 steps.

1. Pop the victim’s deque.
2. Call `unroll_call_stack()` to update the full frame tree.
3. Make the loot the thief’s active frame.
4. Create a new child full frame for the victim.
5. Execute the stolen computation.
Example of a Steal: Step 1

Pop the deque of the victim, worker w0.
Example of a Steal: Step 2

Call `unroll_call_stack()` on the target of the steal.

a) **Reverse** the chain.

b) Promote each Cilk stack frame to a full frame.
Example of a Steal: Step 2

Call `unroll_call_stack()` on the target of the steal.

a) Reverse the chain.

b) **Promote** each Cilk stack frame to a full frame.
Example of a Steal: Step 3

Make the loot the active frame of the thief, worker \textit{w1}.

Worker \textit{w1} struct

Deque

\begin{align*}
\text{current_stack_frame} & \quad l->frame_ff \\
\text{head} & \quad / \\
\text{tail} & \quad /
\end{align*}

Worker \textit{w0} struct

Deque

\begin{align*}
\text{current_stack_frame} & \quad l->frame_ff \\
\text{head} & \quad /
\end{align*}
Example of a Steal: Step 4

Create a new child full frame for w0.
Example of a Steal: Step 5

Begin executing the stolen continuation.

Worker w0 struct
Deque
head
tail
current_stack_frame
l->frame_ff

Worker w1 struct
Deque
head
tail
current_stack_frame
l->frame_ff
More Cilk Features

The Cilk Plus runtime also contains support for other features.

- **Reducers** [FHLL09]
- **Pedigrees** [LSS12]
- Exception-handling
- Support for multiple user threads

For the most part, these features can be safely ignored during initial experimentation with the runtime.
Log in to the cloud machine:
$ ssh 6898tapir.csail.mit.edu

Get the runtime source code:
$ git clone https://bitbucket.org/intelcilkruntime/intel-cilk-runtime

Build the runtime from source:
$ libtoolize
$ aclocal
$ automake --add-missing
$ autoconf
$ LIBS=-ldl ./configure
$ make

Compile some Cilk code to use your custom-built runtime:
$ clang my_cilk_prog.c -fcilkplus -L /path/to/intel-cilk-runtime/.libs \
> -o my_cilk_prog
$ LD_LIBRARY_PATH=/path/to/intel-cilk-runtime/.libs ldd ./my_cilk_prog
Hands-On: Generating Stats

- At the top of runtime/stats.h, add
  
  ```
  #define CILK_PROFILE 1
  ```

- Recompile the runtime system:
  
  ```
  $ make clean && make
  ```

- Add `__cilkrts_dump_stats()` to the end of your Cilk program.

- Recompile and rerun your Cilk program, and see the runtime statistics!

- **Challenge:** Implement your own statistic.
References

Return From `cilk_spawn`: Slow Path

The slow path from returning from a `cilk_spawn` changes stacks to enter the runtime.

- Executed on user-code stack
  - `__cilkrts_leave_frame()`
  - Reducer elimination
    - `do_return_from_spawn()`
      - If `f` is not synched, enter steal loop
      - If `f` is synched, enter runtime scheduling loop

- Executed on runtime stack
  - Enter steal loop
  - Enter runtime scheduling loop

Continue executing after `cilk_sync`
The slow path for a cilk_sync follows a similar path.

1. Executed on user-code stack:
   - `__cilkrts_sync()`
   - Reducer elimination
   - `do_sync()`

2. Executed on runtime stack:
   - Enter runtime scheduling loop
   - Enter steal loop
   - `provably_good_steal()`
     - `f` is not synched
     - `f` is synched

Continue executing after `cilk_sync`