The Implementation of the Cilk-5 Multithreaded Language
CS 4435 Project Presentation

Peter Goodman

April 18, 2011
Overview

- **Syntax and semantics.**
  - Motivating decisions behind the implementation.
  - Implementation details.
  - Scheduler protocol for conflicts between processors.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Syntax

- *Faithful* extension of GNU C, *not* ANSI C.
  - Every Cilk program has a **serial elision**.
- Introduces five keywords: **cilk**, **spawn**, **sync**, **inlet**, and **abort**.
  - Serial elision:
    ```c
    #define cilk
    #define spawn
    #define sync
    #define inlet
    #define abort
    ```
Syntax

- *Faithful* extension of GNU C, *not* ANSI C.
  - Every Cilk program has a **serial elision**.
- Introduces five keywords: *cilk*, *spawn*, *sync*, *inlet*, and *abort*.
  - Serial elision:
    ```
    #define cilk
    #define spawn
    #define sync
    #define inlet
    #define abort
    ```
Every Cilk procedure must be qualified with the `cilk` keyword:

```cilk
int fib(int n) {
    ...
}
```

`fib` is now seen as parallel C procedure that can spawn threads and do other neat things.
Syntax: spawn and sync

cilk int fib(int n) {
    int n_1, n_2;
    if (2 > n) {
        return 1;
    } else {
        n_1 = spawn fib(n - 1);
        n_2 = spawn fib(n - 2);
        sync;
        return n_1 + n_2;
    }
}
Semantics: spawn

- Idea: each `spawn` statement schedules the parallel execution of the function invocation following the `spawn`.
- Reality: a `spawned` function behaves like a serial function call, but execution can continue past a `spawned` function invocation if the caller is “stolen”.
- Implication: values returned by `spawned` functions are not safe! Solution: synchronization points.
Semantics: spawn

- Idea: each spawn statement schedules the parallel execution of the function invocation following the spawn.
- Reality: a spawned function behaves like a serial function call, but execution can continue past a spawned function invocation if the caller is “stolen”.
- Implication: values returned by spawned functions are not safe!
  Solution: synchronization points.
Semantics: spawn

- Idea: each `spawn` statement schedules the parallel execution of the function invocation following the `spawn`.
- Reality: a `spawned` function behaves like a serial function call, but execution can continue past a `spawned` function invocation if the caller is “stolen”.
- Implication: values returned by `spawned` functions are not safe! Solution: synchronization points.
Semantics: `sync`

- Idea: a synchronization point whereafter all previously spawned threads are known to be completed.
- Represents a local barrier that cannot be passed until all spawned procedure calls have finished.
- Once passed the barrier, return values of spawned function calls are safe to use.
Semantics: \texttt{sync}

- Idea: a synchronization point whereafter all previously \texttt{spawned} threads are known to be completed.
- Represents a local barrier that cannot be passed until all \texttt{spawned} procedure calls have finished.
- Once passed the barrier, return values of \texttt{spawned} function calls are safe to use.
Semantics: sync

- Idea: a synchronization point whereafter all previously spawned threads are known to be completed.
- Represents a local barrier that cannot be passed until all spawned procedure calls have finished.
- Once passed the barrier, return values of spawned function calls are safe to use.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Notation

- Denote by $T_1$ the **work** of a program: the execution time of a program on a single processor.
- Denote by $T_\infty$ the **span** of a program: the execution time of a program on an infinite number of processors.
- Denote by $T_P$ the execution time of a program on $P$ processors.
Notation

- Denote by $T_1$ the work of a program: the execution time of a program on a single processor.
- Denote by $T_\infty$ the span of a program: the execution time of a program on an infinite number of processors.
- Denote by $T_P$ the execution time of a program on $P$ processors.
Notation

- Denote by $T_1$ the **work** of a program: the execution time of a program on a single processor.
- Denote by $T_\infty$ the **span** of a program: the execution time of a program on an infinite number of processors.
- Denote by $T_P$ the execution time of a program on $P$ processors.
Complexity of Scheduling

- Cilk’s scheduler guarantees $T_P \leq T_1/P + O(T_\infty)$ assuming that no locks are used.

- If we could schedule perfectly, then we can immediately fill up all $P$ processors at each step of the span.
  - Filling up $P$ processors running in parallel will complete $P \cdot T_\infty$ units of work in $O(T_\infty)$ time.

- What if $T_1 \gg P \cdot T_\infty$? (i.e. we don’t have enough capacity at each step of the span)
  - Worst-case is that we schedule $o(T_1)$ threads in a single step of the span.
  - Then we can split these up into at most $T_1/P$ units of parallel work!
Complexity of Scheduling

- Cilk’s scheduler guarantees $T_P \leq T_1/P + O(T_\infty)$ assuming that no locks are used.

- If we could schedule perfectly, then we can immediately fill up all $P$ processors at each step of the span.
  - Filling up $P$ processors running in parallel will complete $P \cdot T_\infty$ units of work in $O(T_\infty)$ time.

- What if $T_1 \gg P \cdot T_\infty$? (i.e. we don’t have enough capacity at each step of the span)
  - Worst-case is that we schedule $o(T_1)$ threads in a single step of the span.
  - Then we can split these up into at most $T_1/P$ units of parallel work!
Complexity of Scheduling

- Cilk’s scheduler guarantees $T_P \leq T_1/P + \mathcal{O}(T_\infty)$ assuming that no locks are used.
- If we could schedule perfectly, then we can immediately fill up all $P$ processors at each step of the span.
  - Filling up $P$ processors running in parallel will complete $P \cdot T_\infty$ units of work in $\mathcal{O}(T_\infty)$ time.
- What if $T_1 \gg P \cdot T_\infty$? (i.e. we don’t have enough capacity at each step of the span)
  - Worst-case is that we schedule $o(T_1)$ threads in a single step of the span.
  - Then we can split these up into at most $T_1/P$ units of parallel work!
Complexity of Scheduling

- Cilk’s scheduler guarantees $T_P \leq \frac{T_1}{P} + O(T_\infty)$ assuming that no locks are used.
- If we could schedule perfectly, then we can immediately fill up all $P$ processors at each step of the span.
  - Filling up $P$ processors running in parallel will complete $P \cdot T_\infty$ units of work in $O(T_\infty)$ time.
- What if $T_1 \gg P \cdot T_\infty$? (i.e. we don’t have enough capacity at each step of the span)
  - Worst-case is that we schedule $o(T_1)$ threads in a single step of the span.
  - Then we can split these up into at most $\frac{T_1}{P}$ units of parallel work!
Complexity of Scheduling

- Cilk’s scheduler guarantees $T_P \leq T_1/P + \mathcal{O}(T_\infty)$ assuming that no locks are used.

- If we could schedule perfectly, then we can immediately fill up all $P$ processors at each step of the span.
  - Filling up $P$ processors running in parallel will complete $P \cdot T_\infty$ units of work in $\mathcal{O}(T_\infty)$ time.

- What if $T_1 \gg P \cdot T_\infty$? (i.e. we don’t have enough capacity at each step of the span)
  - Worst-case is that we schedule $o(T_1)$ threads in a single step of the span.
  - Then we can split these up into at most $T_1/P$ units of parallel work!
Scheduling Overheads

- Assume that on average, $P$ processors is not enough to complete a single step of the span in unit time.

- A scheduler adds the following overheads:
  - to the work: by doing more! Complicated scheduling = more work.
    - E.g. figuring out which processor to give a spawned procedure call.
  - to the span: by not using every available processor when it has work that can be done in parallel!
    - E.g. ignoring a spawn and executing a procedure call serially.

- But $P$ processors usually isn’t enough capacity anyway.
  - If we add work overhead, then every cilk procedure is negatively affected.
  - If we add span overhead, then it will only really affects us in the less likely case that not all $P$ processors are busy.
Scheduling Overheads

- Assume that on average, $P$ processors is not enough to complete a single step of the span in unit time.
- A scheduler adds the following overheads:
  - **to the work**: by doing more! Complicated scheduling = more work.
    - E.g. figuring out which processor to give a spawned procedure call.
  - **to the span**: by not using every available processor when it has work that can be done in parallel!
    - E.g. ignoring a spawn and executing a procedure call serially.
- But $P$ processors usually isn’t enough capacity anyway.
  - If we add work overhead, then every cilk procedure is negatively affected.
  - If we add span overhead, then it will only really affects us in the less likely case that not all $P$ processors are busy.
Scheduling Overheads

- Assume that on average, \( P \) processors is not enough to complete a single step of the span in unit time.

- A scheduler adds the following overheads:
  - to the work: by doing more! Complicated scheduling = more work.
    - E.g. figuring out which processor to give a spawned procedure call.
  - to the span: by not using every available processor when it has work that can be done in parallel!
    - E.g. ignoring a spawn and executing a procedure call serially.

- But \( P \) processors usually isn’t enough capacity anyway.
  - If we add work overhead, then every cilk procedure is negatively affected.
  - If we add span overhead, then it will only really affects us in the less likely case that not all \( P \) processors are busy.
Scheduling Overheads

- Assume that on average, $P$ processors is not enough to complete a single step of the span in unit time.
- A scheduler adds the following overheads:
  - **to the work**: by doing more! Complicated scheduling = more work.
    - E.g. figuring out which processor to give a spawned procedure call.
  - **to the span**: by not using every available processor when it has work that *can* be done in parallel!
    - E.g. ignoring a spawn and executing a procedure call serially.
- But $P$ processors usually isn’t enough capacity anyway.
  - If we add work overhead, then every cilk procedure is negatively affected.
  - If we add span overhead, then it will only really affects us in the less likely case that not all $P$ processors are busy.
Scheduling Overheads

- Assume that on average, $P$ processors is not enough to complete a single step of the span in unit time.

- A scheduler adds the following overheads:
  - **to the work**: by doing more! Complicated scheduling = more work.
    - E.g. figuring out which processor to give a spawned procedure call.
  - **to the span**: by not using every available processor when it has work that *can* be done in parallel!
    - E.g. ignoring a spawn and executing a procedure call serially.

- But $P$ processors usually isn’t enough capacity anyway.
  - If we add work overhead, then *every* cilk procedure is negatively affected.
  - If we add span overhead, then it will only really affects us in the less likely case that *not* all $P$ processors are busy.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Implementation Overview

- **Work-first principle**: minimize work overheads at the expense of increasing span overheads.

- Implication: Do more stuff serially.
  - i.e. `spawn` *doesn’t* actually spawn a new thread!
  - Instead, do normal C procedure calls, and let callers be parallelized instead of the callees.
  - Different behavior than `spawn` implies, but with the same semantics.

- Implementation: need two copies of each Cilk procedure:
  - **Fast copy**: A copy that is running serially, i.e. hasn’t been parallelized yet.
  - **Slow copy**: A copy that has been transported onto another processor.
Implementation Overview

- **Work-first principle**: minimize work overheads at the expense of increasing span overheads.

  - Implication: Do more stuff serially.
    - i.e. `spawn` *doesn’t* actually spawn a new thread!
    - Instead, do normal C procedure calls, and let callers be parallelized instead of the callees.
      - Different behavior than `spawn` implies, but with the same semantics.

- Implementation: need two copies of each Cilk procedure:
  - **Fast copy**: A copy that is running serially, i.e. hasn’t been parallelized yet.
  - **Slow copy**: A copy that has been transported onto another processor.
Implementation Overview

- **Work-first principle**: minimize work overheads at the expense of increasing span overheads.
- Implication: Do more stuff serially.
  - i.e. `spawn` *doesn’t* actually spawn a new thread!
  - Instead, do normal C procedure calls, and let callers be parallelized instead of the callees.
    - Different behavior than `spawn` implies, but with the same semantics.
- Implementation: need two copies of each Cilk procedure:
  - **Fast copy**: A copy that is running serially, i.e. hasn’t been parallelized yet.
  - **Slow copy**: A copy that has been transported onto another processor.
Implementation Details: Work-Stealing

- Each processor has a double-ended queue (deque) of in-progress Cilk procedures.
- When a processor is executing code and encounters a spawn, it pushes a frame onto the top of its deque.
- When a processor has nothing to do, it randomly chooses a “victim” processor and tries to steal the activation frame from the bottom of its deque.
  - More on stealing protocol later.
Implementation Details: Work-Stealing

- Each processor has a double-ended queue (deque) of in-progress Cilk procedures.
- When a processor is executing code and encounters a spawn, it pushes a frame onto the top of its deque.
- When a processor has nothing to do, it randomly chooses a “victim” processor and tries to steal the activation frame from the bottom of its deque.

  ▶ More on stealing protocol later.
Implementation Details: Work-Stealing

- Each processor has a double-ended queue (deque) of in-progress Cilk procedures.
- When a processor is executing code and encounters a spawn, it pushes a frame onto the top of its deque.
- When a processor has nothing to do, it randomly chooses a “victim” processor and tries to steal the activation frame from the bottom of its deque.
  - More on stealing protocol later.
Implementation: Fast Copy

- Activation frame is pushed onto the top of the deque before a `spawn`.
  - Frame contains:
    - Local variable state.
    - Information on how to jump past the `spawn`.
- `spawn` itself is a no-op and a normal procedure call is performed.
- After `spawn` procedure call, processor attempts to pop activation frame off of the top of its deque.
  - If frame was stolen then the pop fails and we return a dummy value.
  - If the pop succeeded then we continue on as normal in our serial execution.
- A `sync` is a no-op, because if a `sync` is reached in the fast copy then it has been reached serially.
  - i.e. all prior spawns were executed serially and the current procedure invocation was not stolen.
Implementation: Fast Copy

- Activation frame is pushed onto the top of the deque before a `spawn`.
  - Frame contains:
    - Local variable state.
    - Information on how to jump past the `spawn`.

- `spawn` itself is a no-op and a normal procedure call is performed.

- After `spawn` procedure call, processor attempts to pop activation frame off of the top of its deque.
  - If frame was stolen then the pop fails and we return a dummy value.
  - If the pop succeeded then we continue on as normal in our serial execution.

- A `sync` is a no-op, because if a `sync` is reached in the fast copy then it has been reached serially.
  - i.e. all prior `spawns` were executed serially and the current procedure invocation was not stolen.
Implementation: Fast Copy

- Activation frame is pushed onto the top of the deque before a spawn.
  - Frame contains:
    - Local variable state.
    - Information on how to jump past the spawn.
- spawn itself is a no-op and a normal procedure call is performed.
- After spawn procedure call, processor attempts to pop activation frame off of the top of its deque.
  - If frame was stolen then the pop fails and we return a dummy value.
  - If the pop succeeded then we continue on as normal in our serial execution.
- A sync is a no-op, because if a sync is reached in the fast copy then it has been reached serially.
  - i.e. all prior spawns were executed serially and the current procedure invocation was not stolen.
Implementation: Fast Copy

- Activation frame is pushed onto the top of the deque before a `spawn`.
  - Frame contains:
    - Local variable state.
    - Information on how to jump past the `spawn`.

- `spawn` itself is a no-op and a normal procedure call is performed.

- After `spawn` procedure call, processor attempts to pop activation frame off of the top of its deque.
  - If frame was stolen then the pop fails and we return a dummy value.
  - If the pop succeeded then we continue on as normal in our serial execution.

- A `sync` is a no-op, because if a `sync` is reached in the fast copy then it has been reached serially.
  - i.e. all prior `spawns` were executed serially and the current procedure invocation was not stolen.
Implementation: Slow Copy

- If a processor has nothing to do then it tries to steal an activation frame from a randomly chosen victim processor.
- When it steals, it runs the slow copy of the procedure instead of the fast copy.
  - The slow copy knows how to use the state from the activation frame and is able to jump to the right instruction to resume execution.
- `spawn` is handled in the same way as in the fast copy.
- `sync` checks for “spawned” children that haven’t returned.
Implementation: Slow Copy

- If a processor has nothing to do then it tries to steal an activation frame from a randomly chosen victim processor.
- When it steals, it runs the slow copy of the procedure instead of the fast copy.
  - The slow copy knows how to use the state from the activation frame and is able to jump to the right instruction to resume execution.
- `spawn` is handled in the same way as in the fast copy.
- `sync` checks for “spawned” children that haven’t returned.
Implementation: Slow Copy

- If a processor has nothing to do then it tries to steal an activation frame from a randomly chosen victim processor.
- When it steals, it runs the slow copy of the procedure instead of the fast copy.
  - The slow copy knows how to use the state from the activation frame and is able to jump to the right instruction to resume execution.
- `spawn` is handled in the same way as in the fast copy.
- `sync` checks for “spawned” children that haven’t returned.
Implementation: Slow Copy

- If a processor has nothing to do then it tries to steal an activation frame from a randomly chosen victim processor.
- When it steals, it runs the slow copy of the procedure instead of the fast copy.
  - The slow copy knows how to use the state from the activation frame and is able to jump to the right instruction to resume execution.
- `spawn` is handled in the same way as in the fast copy.
- `sync` checks for “spawned” children that haven’t returned.
Overview

- Syntax and semantics.
- Motivating decisions behind the implementation.
- Implementation details.
- Scheduler protocol for conflicts between processors.
Stealing Protocol

What happens when a “thief” processor tries to steal from a “victim” processor while the victim is popping a spawned procedure?

Two interesting cases:

- Only one activation frame on the deque.
- The deque is empty.
Stealing Protocol

What happens when a “thief” processor tries to steal from a “victim” processor while the victim is popping a spawned procedure?

Two interesting cases:

- Only one activation frame on the deque.
- The deque is empty.
Protocol: Only One Frame

- Both processors race toward acquiring a lock to remove the same frame from the deque.
  - Victim operates on the tail pointer of the deque (top).
  - Thief operates on the head pointer of the deque (bottom).
    - Thief goes straight for the lock, *then* modifies the pointer. That is, only one thief can ever change the pointer.

- Race condition is settled after a lock is acquired on the deque by the winner.
  - Pointers operated on by victim/thief are distinct, so the race condition is easily settled.
  - Thief retreats if it detects—by comparing the pointers—that there is only one frame in the deque.
  - Victim ends up acquiring the lock eventually.
Protocol: Only One Frame

- Both processors race toward acquiring a lock to remove the same frame from the deque.
  - Victim operates on the tail pointer of the deque (top).
  - Thief operates on the head pointer of the deque (bottom).
    - Thief goes straight for the lock, then modifies the pointer. That is, only one thief can ever change the pointer.
- Race condition is settled after a lock is acquired on the deque by the winner.
  - Pointers operated on by victim/thief are distinct, so the race condition is easily settled.
  - Thief retreats if it detects—by comparing the pointers—that there is only one frame in the deque.
  - Victim ends up acquiring the lock eventually.
Protocol: No Frames

- Both thief and victim will fail.
Conclusion

- Cilk-5 obtains provably good scheduling by:
  - following the work-first principle.
  - implementing a work-stealing scheduler.
  - having a simple protocol for handling conflicts.

- Questions?