Programming in CILK

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07
Administrivia

• Homework 3 due tomorrow
• Homework 4 (last one) coming out soon
• Project proposal due Fri, Oct 19: Describe problem, machine, predict bottlenecks and likely parallelism (~1-page)
• Discussion section Wednesday
• Lecture on Cell on Friday
• Next Wednesday, Oct 17: Quiz
Cilk

A C language for programming dynamic multithreaded applications on shared-memory multiprocessors.

Example applications:

- virus shell assembly
- graphics rendering
- \( n \)-body simulation
- heuristic search
- dense and sparse matrix computations
- friction-stir welding simulation
- artificial evolution

© 2006 Charles E. Leiserson
Apply Parallel Programming Model Questions

- **Programming model** is made up of the languages and libraries that create an abstract view of the machine
- **Control**
  - How is parallelism created?
  - How are dependencies (orderings) enforced?
- **Data**
  - Can data be shared or is it all private?
  - How is shared data accessed or private data communicated?
- **Synchronization**
  - What operations can be used to coordinate parallelism
  - What are the atomic (indivisible) operations?
Fibonacci Example: Creating Parallelism

```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);
}
}
```

Cilk code

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

Cilk is a **faithful** extension of C. A Cilk program’s **serial elision** is always a legal implementation of Cilk semantics. Cilk provides **no** new data types.
Basic Cilk Keywords

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

Identifies a function as a Cilk procedure, capable of being spawned in parallel.

The named child Cilk procedure can execute in parallel with the parent caller.

Control cannot pass this point until all spawned children have returned.
Dynamic Multithreading

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

Example: fib(4)

processors are virtualized

The computation dag unfolds dynamically.
Algorithmic Complexity Measures

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

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

LOWER BOUNDS
- \[ T_P \geq T_1 / P \]
- \[ T_P \geq T_\infty \]

*Also called critical-path length or computational depth.*
**Definition:** \( T_1/T_P = \text{speedup} \) on \( P \) processors.

If \( T_1/T_P = \Theta(P) \leq P \), we have **linear speedup**; if \( = P \), we have **perfect linear speedup**; if \( > P \), we have **superlinear speedup**, which is not possible in our model, because of the lower bound \( T_P \geq T_1/P \).

\( T_1/T_\infty = \text{parallelism} \)

= the average amount of work per step along the span.
Example: \texttt{fib(4)}

Assume for simplicity that each Cilk thread in \texttt{fib()} takes unit time to execute.

\textbf{Work:} \( T_1 = 17 \)

\textbf{Span:} \( T_\infty = 8 \)

\textbf{Parallelism:} \( T_1/T_\infty = 2.125 \)

\textit{Using many more than 2 processors makes little sense.}
Parallelizing Vector Addition

```c
void vadd (real *A, real *B, int n) { 
    int i; for (i=0; i<n; i++) A[i] += B[i];
}
```
Parallelizing Vector Addition

Parallelization strategy:
1. Convert loops to recursion.
Parallelizing Vector Addition

C

```c
void vadd (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk

```c
void vadd (real *A, real *B, int n){
    if (n<=BASE) {
        int i; for (i=0; i<n; i++) A[i]+=B[i];
    } else {
        vadd (A, B, n/2;
        vadd (A+n/2, B+n/2, n-n/2;
        sync;
    }
}
```

Parallelization strategy:
1. Convert loops to recursion.
2. Insert Cilk keywords.

Side benefit: D&C is generally good for caches!
vector addition

cilk void vadd (real *A, real *B, int n) {
  if (n <= BASE) {
    int i; for (i = 0; i < n; i++) A[i] += B[i];
  } else {
    spawn vadd (A, B, n/2);
    spawn vadd (A+n/2, B+n/2, n-n/2);
    sync;
  }
}

© 2006 Charles E. Leiserson
Vector Addition Analysis

To add two vectors of length $n$, where $\text{BASE} = \Theta(1)$:

**Work:** $T_1 = \Theta(n)$

**Span:** $T_\infty = \Theta(\lg n)$

**Parallelism:** $T_1/T_\infty = \Theta(n/\lg n)$
Another Parallelization

C

```c
void vadd1 (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
void vadd (real *A, real *B, int n){
    int j; for (j=0; j<n; j+=BASE) {
        vadd (A+j, B+j, min(BASE, n-j));
    }
}
```

Cilk

```cilk
void vadd1 (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
cilk void vadd (real *A, real *B, int n){
    int j; for (j=0; j<n; j+=BASE) {
        spawn vadd (A+j, B+j, min(BASE, n-j));
    }
sync;
}
```
To add two vectors of length $n$, where $\text{BASE} = \Theta(1)$:

**Work:** $T_1 = \Theta(n)$

**Span:** $T_\infty = \Theta(n)$

**Parallelism:** $T_1/T_\infty = \Theta(1)$
Optimal Choice of BASE

To add two vectors of length $n$ using an optimal choice of BASE to maximize parallelism:

**Work:** $T_1 = \Theta(n)$

**Span:** $T_\infty = \Theta(BASE + n/\text{BASE})$

Choosing $BASE = \sqrt{n} \Rightarrow T_\infty = \Theta(\sqrt{n})$

**Parallelism:** $T_1/T_\infty = \Theta(\sqrt{n})$
Greedy Scheduling

**IDEA:** Do as much as possible on every step.

**Definition:** A thread is *ready* if all its predecessors have executed.

**Complete step**
- $\geq P$ threads ready.
- Run any $P$.

**Incomplete step**
- $< P$ threads ready.
- Run all of them.
Cilk’s Work-Stealing Scheduler

Each processor maintains a work deque of ready threads, and it manipulates the bottom of the deque like a stack.

When a processor runs out of work, it steals a thread from the top of a random victim’s deque.
Performance of Work-Stealing

**Theorem**: Cilk’s work-stealing scheduler achieves an expected running time of

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

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

\[ (T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty) \].
**Space Bounds**

**Theorem.** Let $S_1$ be the stack space required by a serial execution of a Cilk program. Then, the space required by a $P$-processor execution is at most $S_P \leq PS_1$.

**Proof** (by induction). The work-stealing algorithm maintains the *busy-leaves property*: every extant procedure frame with no extant descendents has a processor working on it.
Cilk vs. PThreads

How will the following code execute in Pthreads? In Cilk?

```c
for (i=1; i<1000000000; i++) {
    spawn-or-fork foo(i);
}
sync-or-join;
```

What if foo contains code that waits (e.g., spins) on a variable being set by another instance of foo?

This different is a liveness property:
- Cilk threads are spawned lazily, “may” parallelism
- PThreads are spawned eagerly, “must” parallelism
Operating on Returned Values

Programmers may sometimes wish to incorporate a value returned from a spawned child into the parent frame by means other than a simple variable assignment.

Example:

\[
x += \text{spawn foo}(a,b,c);
\]

Cilk achieves this functionality using an internal function, called an \textit{inlet}, which is executed as a secondary thread on the parent frame when the child returns.
int max, ix = -1;

inlet void update ( int val, int index ) {
    if (idx == -1 || val > max) {
        ix = index; max = val;
    }
}

for (i=0; i<1000000; i++) {
    update ( spawn foo(i), i );
}

sync; /* ix now indexes the largest foo(i) */

• The inlet keyword defines a void internal function to be an inlet.
• In the current implementation of Cilk, the inlet definition may not contain a spawn, and only the first argument of the inlet may be spawned at the call site.
### Semantics of Inlets

```c
int max, ix = -1;
inlet void update ( int val, int index ) {
    if (idx == -1 || val > max) {
        ix = index; max = val;
    }
}

for (i=0; i<1000000; i++) {
    update ( spawn foo(i), i );
}
sync; /* ix now indexes the largest foo(i) */
```

1. The non-`spawn` args to `update()` are evaluated.
2. The Cilk procedure `foo(i)` is spawned.
3. Control passes to the next statement.
4. When `foo(i)` returns, `update()` is invoked.
Abort Cilk Threads: Computing a Product

\[ p = \prod_{i=0}^{n} A_i \]

```c
int product(int *A, int n) {
    int i, p=1;
    for (i=0; i<n; i++) {
        p *= A[i];
    }
    return p;
}
```

**Optimization:** Quit early if the partial product ever becomes 0.
Computing a Product in Parallel

\[ p = \prod_{i=0}^{n} A_i \]

cilk int prod(int *A, int n) {
    int p = 1;
    if (n == 1) {
        return A[0];
    } else {
        p *= spawn product(A, n/2);
        p *= spawn product(A+n/2, n-n/2);
        sync;
        return p;
    }
}

How do we quit early if we discover a zero?
Cilk’s Abort Feature

```cilk
int product(int *A, int n) {
    int p = 1;
    inlet void mult(int x) {
        p *= x;
        return;
    }

    if (n == 1) {
        return A[0];
    } else {
        mult( spawn product(A, n/2) );
        mult( spawn product(A+n/2, n-n/2) );
        sync;
        return p;
    }
}
```

1. Recode the implicit inlet to make it explicit.
Cilk’s Abort Feature

```cilk
int product(int *A, int n) {
    int p = 1;
    inlet void mult(int x) {
        p *= x;
        return;
    }

    if (n == 1) {
        return A[0];
    } else {
        mult( spawn product(A, n/2) );
        mult( spawn product(A+n/2, n-n/2) );
        sync;
        return p;
    }
}
```

2. Check for 0 within the inlet.
Cilk’s Abort Feature

2. Check for 0 within the inlet.
Cilk’s Abort Feature

cilk int product(int *A, int n) {
    int p = 1;
    inlet void mult(int x) {
        p *= x;
        if (p == 0) {
            abort; /* Aborts existing children, */
            /* but not future ones. */
            return;
        }
    }
    if (n == 1) {
        return A[0];
    } else {
        mult( spawn product(A, n/2) );
        mult( spawn product(A+n/2, n-n/2) );
        sync;
        return p;
    }
}
Cilk’s Abort Feature

cilk int product(int *A, int n) {
    int p = 1;
itlet void mult(int x) {
        p *= x;
        if (p == 0) {
            abort; /* Aborts existing children, */
            /* but not future ones. */
            return;
        }
    }
    if (n == 1) {
        return A[0];
    } else {
        mult( spawn product(A, n/2) );
        if (p == 0) {
            /* Don’t spawn if we’ve */
            return 0; /* already aborted! */
        }
        mult( spawn product(A+n/2, n-n/2) );
        sync;
        return p;
    }
}
Mutual Exclusion

Cilk’s solution to mutual exclusion is no better than anybody else’s.

Cilk provides a library of spin locks declared with \texttt{Cilk\_lockvar}.

- To avoid deadlock with the Cilk scheduler, a lock should only be held within a Cilk thread.
- \textit{i.e.}, \texttt{spawn} and \texttt{sync} should not be executed while a lock is held.

Fortunately, Cilk’s control parallelism often mitigates the need for extensive locking.
The \texttt{cilkc} compiler encapsulates the process.

\texttt{cilkc2c} translates straight C code into identical C postsource.

\texttt{cilkc2c} translates straight C code into identical C postsource.
The \texttt{cilk2c} translator generates two “clones” of each Cilk procedure:

- \textit{fast clone}—serial, common-case code.
- \textit{slow clone}—code with parallel bookkeeping.

- The \textit{fast clone} is always spawned, saving live variables on Cilk’s work deque (shadow stack).
- The \textit{slow clone} is resumed if a thread is stolen, restoring variables from the shadow stack.
- A check is made whenever a procedure returns to see if the resuming parent has been stolen.
Compiling `spawn` — Fast Clone

### Cilk Source

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

### Cilk Deque

```c
if (pop()==FAILURE) {
    frame->x = x;
    frame->join--;
    \{ clean up & return to scheduler \}
}
```

### Cilk2c

```c
frame
entry
join
n
x
y
```

### C Post-source

```c
frame->entry = 1;
frame->n = n;
push(frame);
```

```c
x = fib(n-1);
```
Compiling \texttt{sync} — Fast Clone

No synchronization overhead in the fast clone!
void fib_slow(fib_frame *frame) {
    int n,x,y;
    switch (frame->entry) {
        case 1: goto L1;
        case 2: goto L2;
        case 3: goto L3;
    }
    frame->entry = 1;
    frame->n = n;
    push(frame);
    x = fib(n-1);
    if (pop()==FAILURE) {
        frame->x = x;
        frame->join--;
        // clean up & return to scheduler
    }
}

if (0) {
    L1:;
    n = frame->n;
}

Breakdown of Work Overhead
(circa 1997)

Benchmark: `fib` on one processor.
Cilk Chess Programs


- Socrates 2.0 took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs’ 1824-node Intel Paragon.


Socrates Normalized Speedup

\[ T_P = T_1 / P + T_\infty \]

\[ TP = T_\infty \]

\[ \frac{T_1 / T_P}{T_1 / T_\infty} \]

measured speedup

\[ \frac{P}{T_1 / T_\infty} \]