Multithreaded Programming in Cilk

LECTURE 1

Charles E. Leiserson

Supercomputing Technologies Research Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
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
In particular, over the next decade, chip multiprocessors (CMP’s) will be an increasingly important platform!
Cilk Is Simple

- Cilk extends the C language with just a *handful* of keywords.
- Every Cilk program has a *serial semantics*.
- Not only is Cilk fast, it provides *performance guarantees* based on performance abstractions.
- Cilk is *processor-oblivious*.
- Cilk’s *provably good* runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling.
- Cilk supports *speculative* parallelism.
Minicourse Outline

● **LECTURE 1**  
*Basic Cilk programming:* Cilk keywords, performance measures, scheduling.

● **LECTURE 2**  
*Analysis of Cilk algorithms:* matrix multiplication, sorting, tableau construction.

● **LABORATORY**  
*Programming matrix multiplication in Cilk*  
— *Dr. Bradley C. Kuszmaul*

● **LECTURE 3**  
*Advanced Cilk programming:* inlets, abort, speculation, data synchronization, & more.
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
• Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
Fibonacci

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)

“Processor oblivious”

The computation dag unfolds dynamically.
Multithreaded Computation

- The dag $G = (V, E)$ represents a parallel instruction stream.
- Each vertex $v \in V$ represents a *Cilk* thread: a maximal sequence of instructions not containing parallel control ($\text{spawn}$, $\text{sync}$, $\text{return}$).
- Every edge $e \in E$ is either a $\text{spawn}$ edge, a $\text{return}$ edge, or a $\text{continue}$ edge.
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 also supports `malloc`.)

Cilk’s *cactus stack* supports several views in parallel.
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
• Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
Algorithmic Complexity Measures

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

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

\[ T_1 = \text{work} \]
Algorithmic Complexity Measures

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

\[ T_1 = \text{work} \]

\[ T_2 = \text{span}^* \]

* Also called *critical-path length* or *computational depth*. 
Algorithmic Complexity Measures

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

\[ T_1 = \text{work} \]

\[ T_2 = \text{span}^* \]

**LOWER BOUNDS**

- \( T_P \geq T_1 / P \)
- \( T_P \geq T_2 \)

*Also called critical-path length or computational depth.*
Speedup

**Definition:** $T_1/T_P = \text{speedup}$ on $P$ processors.

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

Because we have the lower bound $T_P \geq T_2$, the maximum possible speedup given $T_1$ and $T_2$ is $T_1/T_2 = \text{parallelism} = \text{the average amount of work per step along the span.}$
Example: $\text{fib}(4)$

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

Work: $T_1 = 17$

Span: $T_2 = 8$
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:} \( T_2 = 8 \)

\textbf{Parallelism:} \( T_1 / T_2 = 2.125 \)

Using many more than 2 processors makes little sense.
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
• Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
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

```c
void vadd (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){
    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);
    }
}
```

Parallelization strategy:

1. Convert loops to recursion.
Parallelizing Vector Addition

#include <stdio.h>

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

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;
    }
}
Vector Addition Analysis

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

- **Work:** $T_1 = \Theta(n)$
- **Span:** $T_2 = \Theta(\lg n)$
- **Parallelism:** $T_1/T_2 = \Theta(n/\lg n)$

Vector Addition Analysis

Base: $\Theta(1)$

Work: $T_1 = \Theta(n)$

Span: $T_2 = \Theta(\lg n)$

Parallelism: $T_1/T_2 = \Theta(n/\lg n)$
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];
}
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;
}```
Analysis

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

- **Work:** $T_1 = \Theta(n)$
- **Span:** $T_2 = \Theta(n)$
- **Parallelism:** $T_1/T_2 = \Theta(1)$
To add two vectors of length $n$ using an optimal choice of BASE to maximize parallelism:

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

**Span:** $T_2 = \Theta(\text{BASE} + n/\text{BASE})$

Choosing BASE = $\sqrt{n}$ * $T_2 = \Theta(\sqrt{n})$

**Parallelism:** $T_1/T_2 = \Theta(\sqrt{n})$
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
  • Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
Scheduling

• Cilk allows the programmer to express potential parallelism in an application.

• The Cilk scheduler maps Cilk threads onto processors dynamically at runtime.

• Since on-line schedulers are complicated, we’ll illustrate the ideas with an off-line scheduler.
Greedy Scheduling

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

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

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

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

**Complete step**
- \( \mu \) \( P \) threads ready.
- Run any \( P \).
Greedy Scheduling

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

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

**Complete step**
- \( \mu \) \( P \) threads ready.
- Run any \( P \).

**Incomplete step**
- \(< P\) threads ready.
- Run all of them.
**Greedy-Scheduling Theorem**

**Theorem** [Graham ’68 & Brent ’75]. Any greedy scheduler achieves

\[ T_P \leq T_1/P + T_\infty. \]

**Proof.**

- # complete steps \( \leq T_1/P \), since each complete step performs \( P \) work.
- # incomplete steps \( \leq T_2 \), since each incomplete step reduces the span of the unexecuted dag by 1.
Optimality of Greedy

**Corollary.** Any greedy scheduler achieves within a factor of 2 of optimal.

**Proof.** Let $T_P^*$ be the execution time produced by the optimal scheduler. Since $T_P^* \geq \max\{T_1/P, T_2\}$ (lower bounds), we have

$$T_P \leq T_1/P + T_2 \leq 2 \max\{T_1/P, T_2\} \leq 2T_P^*.$$
Linear Speedup

Corollary. Any greedy scheduler achieves near-perfect linear speedup whenever $P / T_1/T_2$.

Proof. Since $P / T_1/T_2$ is equivalent to $T_2 / T_1/P$, the Greedy Scheduling Theorem gives us

$$T_P \leq T_1/P + T_2$$

and $T_1/P$.

Thus, the speedup is $T_1/T_P \leq P$.

Definition. The quantity $(T_1/T_2)/P$ is called the parallel slackness.
Cilk Performance

● Cilk’s “work-stealing” scheduler achieves
  ■ $T_P = T_1/P + O(T_2)$ expected time (provably);
  ■ $T_P \approx T_1/P + T_2$ time (empirically).

● Near-perfect linear speedup if $P / T_1/T_2$.

● Instrumentation in Cilk allows the user to determine accurate measures of $T_1$ and $T_2$.

● The average cost of a `spawn` in Cilk-5 is only 2–6 times the cost of an ordinary C function call, depending on the platform.
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
• Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
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.


- **Cilkchess** tied for 3rd in the 1999 WCCC running on NASA’s 256-node SGI Origin 2000.
Socrates Normalized Speedup

\[ T_P = T_{\infty} \]

\[ T_P = \frac{T_1}{P} + T_{\infty} \]

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

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

measured speedup
Developing ★ Socrates

- For the competition, ★ Socrates was to run on a 512-processor Connection Machine Model CM5 supercomputer at the University of Illinois.
- The developers had easy access to a similar 32-processor CM5 at MIT.
- One of the developers proposed a change to the program that produced a speedup of over 20% on the MIT machine.
- After a back-of-the-envelope calculation, the proposed “improvement” was rejected!
\[ T_{32} = \frac{2048}{32} + 1 = 65 \text{ seconds} \]

\[ T_{512} = \frac{2048}{512} + 1 = 5 \text{ seconds} \]

\[ T_1 = 2048 \text{ seconds} \]

\[ T_\infty = 1 \text{ second} \]

\[ T'_{32} = \frac{1024}{32} + 8 = 40 \text{ seconds} \]

\[ T'_{512} = \frac{1024}{512} + 8 = 10 \text{ seconds} \]

\[ T_p \approx \frac{T_1}{P} + T_\infty \]
Lesson

Work and span can predict performance on large machines better than running times on small machines can.
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
• Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
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.

Spawn!
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.
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.
Cilk’s Work-Stealing Scheduler

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

Return!
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.
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.
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.
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

$$T_P \leq T_1/P + O(T_2)$$

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 $O(PT_2)$. Since there are $P$ processors, the expected time is

$$(T_1 + O(PT_2))/P = T_1/P + O(T_2).$$
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.
Linguistic Implications

Code like the following executes properly without any risk of blowing out memory:

```
for (i=1; i<1000000000; i++) {
    spawn foo(i);
}
sync;
```

**MORAL**

Better to steal parents than children!
LECTURE 1

• Basic Cilk Programming
• Performance Measures
• Parallelizing Vector Addition
• Scheduling Theory
• A Chess Lesson
• Cilk’s Scheduler
• Conclusion
Key Ideas

• Cilk is simple: *cilk*, *spawn*, *sync*

• Recursion, recursion, recursion, …

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span

• Work & span
Minicourse Outline

- Lecture 1
  *Basic Cilk programming*: Cilk keywords, performance measures, scheduling.

- Lecture 2
  *Analysis of Cilk algorithms*: matrix multiplication, sorting, tableau construction.

- Laboratory
  *Programming matrix multiplication in Cilk* — Dr. Bradley C. Kuszmaul

- Lecture 3
  *Advanced Cilk programming*: inlets, abort, speculation, data synchronization, & more.