Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

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

Charles E. Leiserson
Supercomputing Technologies Research Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

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.

Shared-Memory Multiprocessor

In particular, over the next decade,
chip multiprocessors (CMP’s) will be
an increasingly important platform!

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: speculative computing, mutual exclusion, race detection.

Lecture 1

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

Copyright © 2006 Charles E. Leiserson
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

Fibonacci

```cilk
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 is a faithful extension of C. A Cilk program’s serial edition is always a legal implementation of Cilk semantics. Cilk provides no new data types.

Basic Cilk Keywords

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

```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);
    }
}
```

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

Dynamic Multithreading

Example: fib(4)

“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 (\textit{spawn}, \textit{sync}, \textit{return}).
- Every edge \( e \in E \) is either a \textit{spawn} edge, a \textit{return} edge, or a \textit{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 \textit{malloc}.)

View of stack

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

Copyright © 2006 Charles E. Leiserson
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

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.*

---

Speedup

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

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

Parallelism

Because we have the lower bound \( T_P \geq T_\infty \), the maximum possible speedup given \( T_1 \) and \( T_\infty \) is

\[ T_1/T_\infty = \text{parallelism} \]

= 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.

Work: $T_w = 17$
Span: $T_s = 8$

Example: \texttt{fib(4)}

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

Work: $T_w = 17$
Span: $T_s = 8$
Parallelism: $T_w/T_s = 2.125$

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

\texttt{C}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

Parallelization strategy:
1. Convert loops to recursion.

\texttt{C}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

\texttt{C}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

\texttt{Cilk}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

\texttt{Cilk}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

\texttt{Cilk}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

\texttt{Cilk}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

\texttt{Cilk}

\begin{verbatim}
void vadd (real *A, real *B, int L, int H){
    int i; for (i=L; i<H; i++) A[i] += B[i];
}
\end{verbatim}

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

\textbf{Side benefit:} D&C is generally good for caches!
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

Vector Addition

```c
void vadd (real *A, real *B, int L, int M)
{
    int i; for (i=L; i<2i; i++) A[i] += B[i];
}
```

Analysis

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 \( \text{BASE} \) to maximize parallelism:

- **Work:** \( T_1 = \Theta(n) \)
- **Span:** \( T_{\infty} = \Theta(\text{BASE} + n/\text{BASE}) \)

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

Another Parallelization

```c
void vadd (real *A, real *B, int L, int M)
{
    int i; for (i=L; i<2i; i++) A[i] += B[i];
}
```

Cilk

```c
void vadd (real *A, real *B, int L, int M)
{
    int i; for (i=L; i<2i; i++) A[i] += B[i];
}
```

Lecture 1

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

Copyright © 2006 Charles E. Leiserson
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

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.

Complete step
• ≥ 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

\[
P \leq \frac{T_i}{P} + T_m.
\]

Proof:
• # complete steps ≤ \( \frac{T_i}{P} \), since each complete step performs P work.
• # incomplete steps ≤ \( T_m \), 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 \( T_P^* \geq \max\{T/P, T_m\} \) (lower bounds), we have

\[
T_P \leq \frac{T_i}{P} + T_m \leq 2\max\{T/P, T_m\} \leq 2T_P^*.
\]
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

Linear Speedup

Corollary. Any greedy scheduler achieves near-perfect linear speedup whenever \( P \ll T_1/T_\infty \).

Proof. Since \( P \ll T_1/T_\infty \) is equivalent to \( T_\infty \ll T_1/P \), the Greedy Scheduling Theorem gives us

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

Thus, the speedup is \( T_1/T_P \approx P \).

Cilk Performance

- Cilk’s “work-stealing” scheduler achieves
  - \( T_P = T_1/P + O(T_\infty) \) expected time (provably);
  - \( T_P \approx T_1/P + T_\infty \) time (empirically).
- Near-perfect linear speedup if \( P \ll T_1/T_\infty \).
- Instrumentation in Cilk allows the user to determine accurate measures of \( T_1 \) and \( T_\infty \).
- 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’ 1,824-node Intel Paragon.

Socrates Normalized 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!
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

Socrates Speedup Paradox

Original program
\[ T_{32} = 65 \text{ seconds} \]
\[ T_1 = 2048 \text{ seconds} \]
\[ T_{32} = 2048/32 + 1 = 65 \text{ seconds} \]
\[ T_{512} = 2048/512 + 1 = 5 \text{ seconds} \]

Proposed program
\[ T'_3 = 40 \text{ seconds} \]
\[ T'_1 = 1024 \text{ seconds} \]
\[ T'_32 = 1024/32 + 8 = 40 \text{ seconds} \]
\[ T'_512 = 1024/512 + 8 = 10 \text{ seconds} \]

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

Lesson

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.

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.

Copyright © 2006 Charles E. Leiserson
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

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

\[ T_p \leq T_i + P + O(T_\infty) \]

on \( P \) processors.

Pseudocode. A processor is either working or stealing. The total time all processors spend working is \( T_i \). 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

\[ (T_i + O(PT_\infty)) \approx T_i + P + O(T_\infty) \].
Multithreaded Programming in Cilk — LECTURE 1
July 13, 2006

Space Bounds

**Theorem.** Let $S_f$ 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 $S_f \leq PS_1$.

**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=0; i<=1000000000; i++) {
    spawn Foo(i);
}
```

**MORAL**

Better to steal parents than children!

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

Minicourse Outline

- **LECTURE 1**
  - *Basic Cilk Programming*
  - *Performance Measures*
  - *Parallelizing Vector Addition*
  - *Scheduling Theory*
  - *A Chess Lesson*
  - *Cilk’s Scheduler*
  - *Conclusion*

- **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*
  - *Speculative computing, mutual exclusion, race detection*