Multithreaded Programming in Cilk — LECTURE 1

July 13, 2006

Multithreaded Programming in *Cilk*

LECTURE 1

Charles E. Leiserson

Supercomputing Technologies Research Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

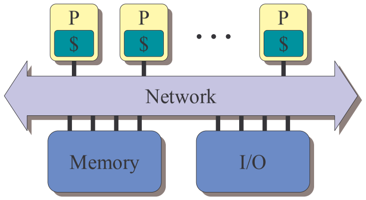
Cilk

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

Shared-Memory Multiprocessor



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

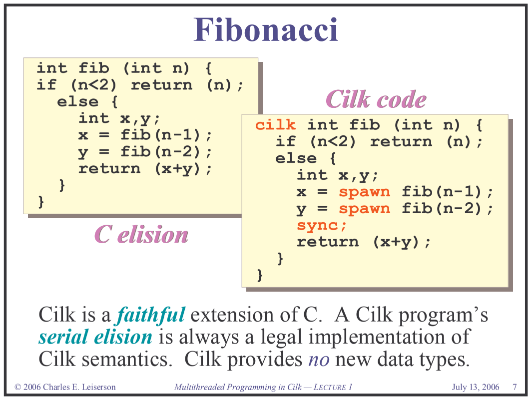
- LECTURE 1
  Basic Cilk programming: Cilk keywords, performance measures, scheduling.
- LECTURE 2
  Analysis of Cilk algorithms: matrix multiplication, sorting, tableau construction.
- LABORATORY
  Programming matrix multiplication in Cilk — Dr. Bradley C. Kuszmaul
- LECTURE 3
  Advanced Cilk programming: 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**

Multithreaded Programming in Cilk — LECTURE 1

July 13, 2006



Fibonacci

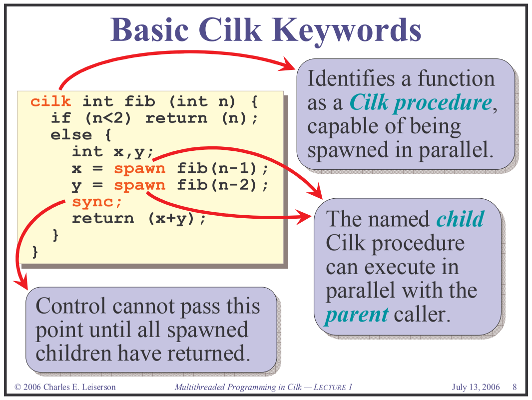
```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

C elision

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.



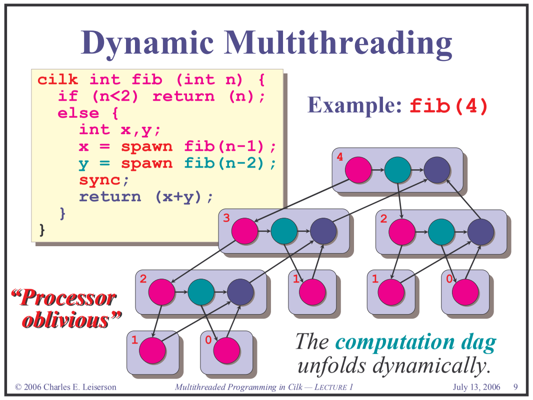
Basic Cilk Keywords

```
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.



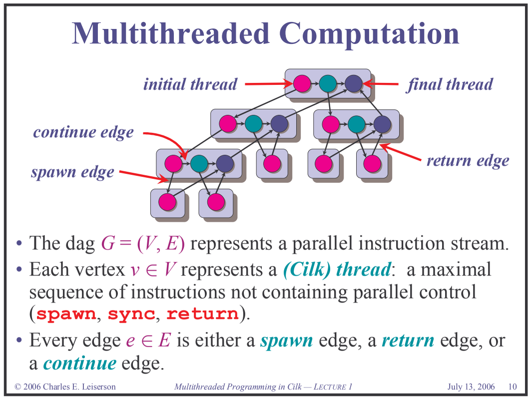
Dynamic Multithreading

```
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)`

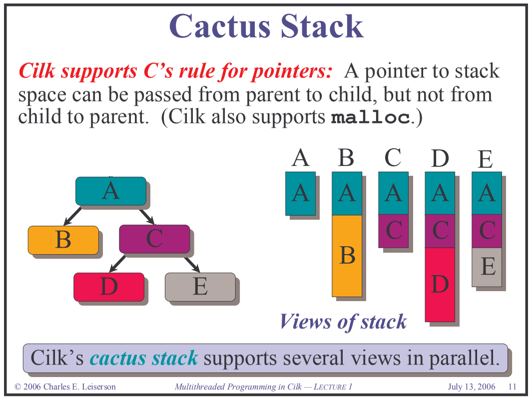
"Processor oblivious"

*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 (`spawn`, `sync`, `return`).
- Every edge $e \in E$ is either a ***spawn*** edge, a ***return*** edge, or a ***continue*** edge.

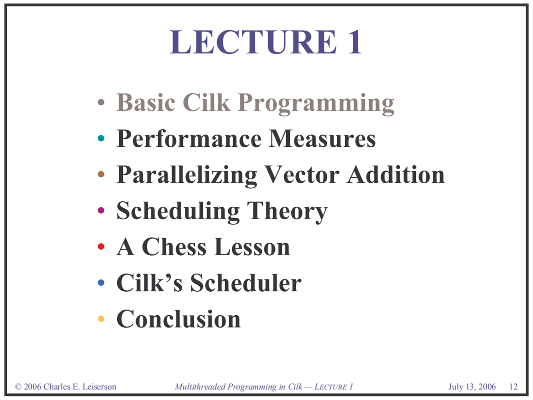


Cactus Stack

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`.)

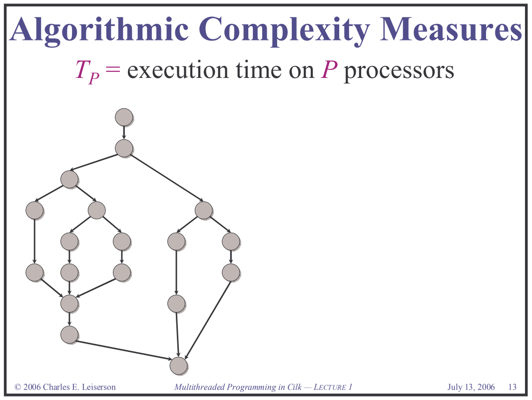
Views of stack

Cilk's ***cactus stack*** supports several views in parallel.



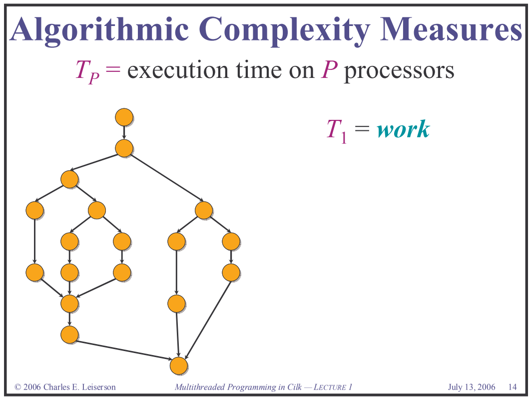
LECTURE 1

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



Algorithmic Complexity Measures

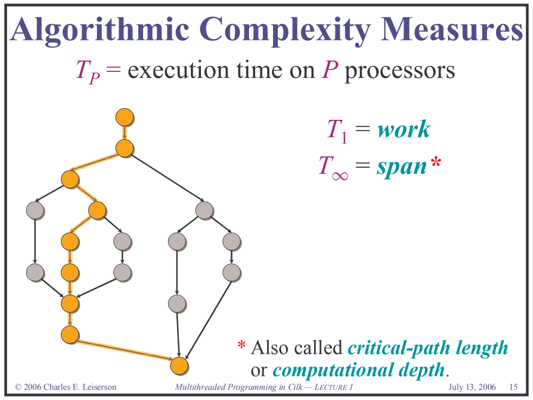
T_P = execution time on P processors



Algorithmic Complexity Measures

T_P = execution time on P processors

T_1 = ***work***



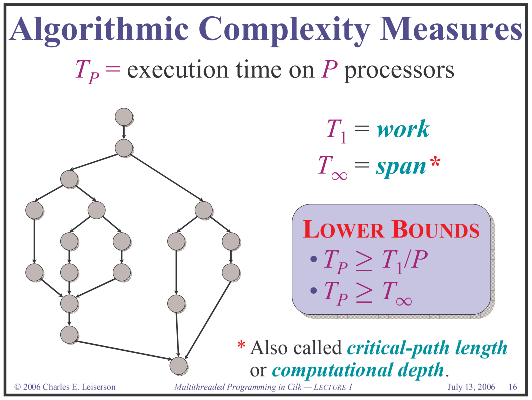
Algorithmic Complexity Measures

T_P = execution time on P processors

T_1 = ***work***

T_∞ = ***span****

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



Algorithmic Complexity Measures

T_P = execution time on P processors

T_1 = ***work***

T_∞ = ***span****

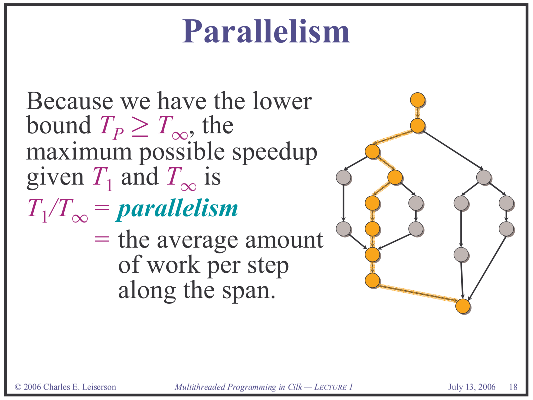
LOWER BOUNDS
- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

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

Speedup

Definition: T_1/T_P = ***speedup*** on P processors.

If $T_1/T_P = \Theta(P) \leq P$, 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 $T_P \geq T_1/P$.

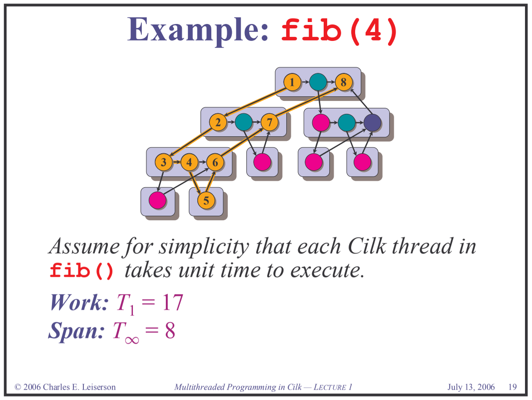


Parallelism

Because we have the lower bound $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is
T_1/T_∞ = ***parallelism***
= the average amount of work per step along the span.

Multithreaded Programming in Cilk — LECTURE 1

July 13, 2006

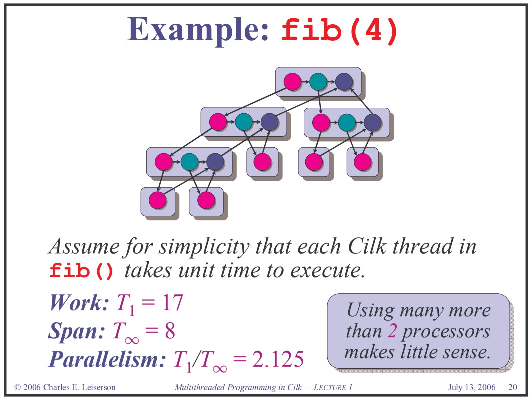


Example: `fib(4)`

Assume for simplicity that each Cilk thread in **`fib()`** *takes unit time to execute.*

Work: $T_1 = 17$

Span: $T_\infty = 8$



Example: `fib(4)`

Assume for simplicity that each Cilk thread in **`fib()`** *takes unit time to execute.*

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 2.125$

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 L, int H){
  int i; for (i=L; i<H; i++) A[i]+=B[i];
}
```

Parallelizing Vector Addition

C

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

C

```
void vadd (real *A, real *B, int L, int H){
  if (L+BASE>H) {
    int i; for (i=L; i<H; i++) A[i]+=B[i];
  } else {
    vadd (A, B, L, (L+H)/2);
    vadd (A, B, (L+H)/2, H);
  }
}
```

Parallelization strategy:

1. Convert loops to recursion.

Parallelizing Vector Addition

C

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

Cilk

```
cilk vadd (real *A, real *B, int L, int H){
  if (L+BASE>H) {
    int i; for (i=L; i<H; i++) A[i]+=B[i];
  } else {
    spawn(A, B, L, (L+H)/2);
    spawn(A, B, (L+H)/2, H);
  } sync;
}
```

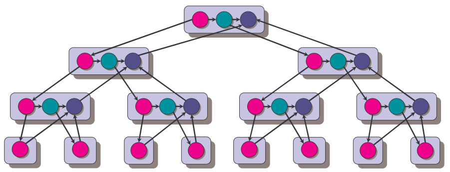
Parallelization strategy:

1. Convert loops to recursion.
2. Insert Cilk keywords.

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

Vector Addition

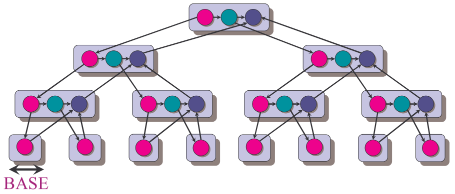
```
cilk void vadd (real *A, real *B, int L, int H){
  if (L+BASE>H) {
    int i; for (i=L; i<H; i++) A[i]+=B[i];
  } else {
    spawn vadd (A, B, L, (L+H)/2);
    spawn vadd (A, B, (L+H)/2, H);
    sync;
  }
}
```



Vector Addition Analysis

To add two vectors of length *n*, where BASE = Θ(1):

Work: $T_1 = \Theta(n)$
Span: $T_\infty = \Theta(\lg n)$
Parallelism: $T_1/T_\infty = \Theta(n/\lg n)$



Another Parallelization

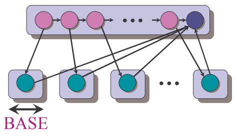
C

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

Cilk

```
cilk void vadd (real *A, real *B, int L, int H){
  int i; for (i=L; i<H; i++) A[i]+=B[i];
}
cilk void vadd1 (real *A, real *B, int L, int H){
  int j; for (j=L; j<H; j+=BASE) {
    spawn vadd (A, B, j, min(H,j+BASE));
  }
sync;
}
```

Analysis

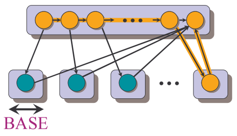


To add two vectors of length *n*, where BASE = Θ(1):

Work: $T_1 = \Theta(n)$
Span: $T_\infty = \Theta(n)$
Parallelism: $T_1/T_\infty = \Theta(1)$

PUNY!

Optimal Choice of BASE



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

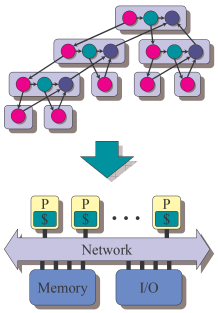
Work: $T_1 = \Theta(n)$
Span: $T_\infty = \Theta(\text{BASE} + n/\text{BASE})$
Choosing $\text{BASE} = \sqrt{n} \Rightarrow T_\infty = \Theta(\sqrt{n})$
Parallelism: $T_1/T_\infty = \Theta(\sqrt{n}\,)$

LECTURE 1

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

Scheduling

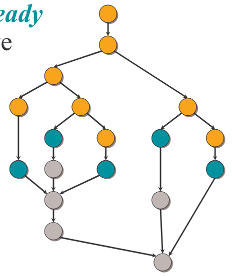
- Cilk allows the programmer to express *potential* parallelism in an application.
- The Cilk ***scheduler*** maps Cilk threads onto processors dynamically at runtime.
- Since ***on-line*** schedulers are complicated, we'll illustrate the ideas with an ***off-line*** scheduler.



Greedy Scheduling

IDEA: Do as much as possible on every step.

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



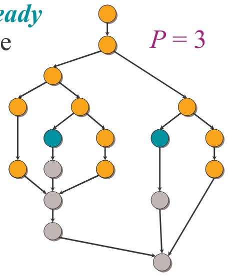
Greedy Scheduling

IDEA: Do as much as possible on every step.

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

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

$P = 3$



Greedy Scheduling

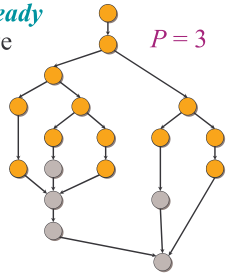
IDEA: Do as much as possible on every step.

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.

$P = 3$



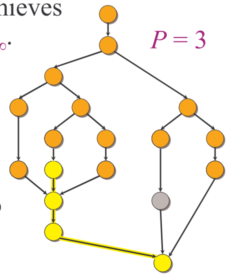
Greedy-Scheduling Theorem

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 $\leq T_\infty$, since each incomplete step reduces the span of the unexecuted dag by 1. ■

$P = 3$



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 $T_P{}^* \geq \max\{T_1/P, T_\infty\}$ (lower bounds), we have

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2{\cdot}\max\{T_1/P, T_\infty\} \\ &\leq 2T_P{}^* . \end{aligned}$$ ■

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

$$\begin{aligned} T_P &\le T_1/P + T_\infty \\ &\approx T_1/P\,. \end{aligned}$$

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_∞ .
- 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.

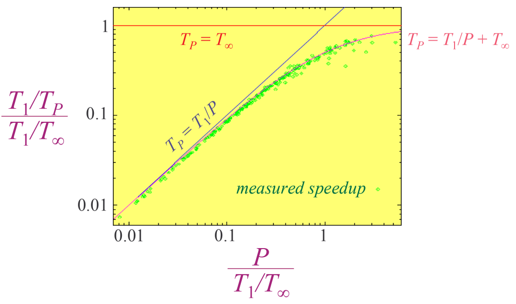
LECTURE 1

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

Cilk Chess Programs

- ***★Socrates*** placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node Connection Machine CM5.
- ***★Socrates 2.0*** took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs' 1824-node Intel Paragon.
- ***Cilkchess*** placed 1st in the 1996 Dutch Open running on a 12-processor Sun Enterprise 5000. It placed 2nd in 1997 and 1998 running on Boston University's 64-processor SGI Origin 2000.
- ***Cilkchess*** tied for 3rd in the 1999 WCCC running on NASA's 256-node SGI Origin 2000.

★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	***Proposed program***
$T_{32} = 65$ seconds	$T'_{32} = 40$ seconds

$$T_P \approx T_1/P + T_\infty$$

$T_1 = 2048$ seconds $T_\infty = 1$ second	$T'_1 = 1024$ seconds $T'_\infty = 8$ seconds
$T_{32} = 2048/32 + 1$ $= 65$ seconds	$T'_{32} = 1024/32 + 8$ $= 40$ seconds
$T_{512} = 2048/512 + 1$ $= 5$ seconds	$T'_{512} = 1024/512 + 8$ $= 10$ seconds

Lesson

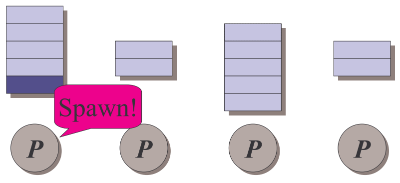
Work and span can predict performance on large machines better than running times on small machines can.

LECTURE 1

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

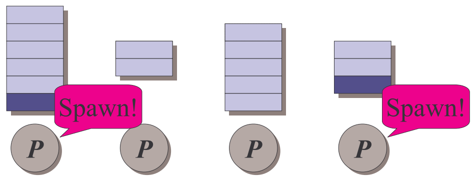
Cilk's Work-Stealing Scheduler

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



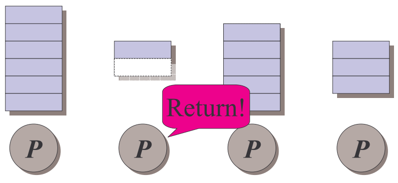
Cilk's Work-Stealing Scheduler

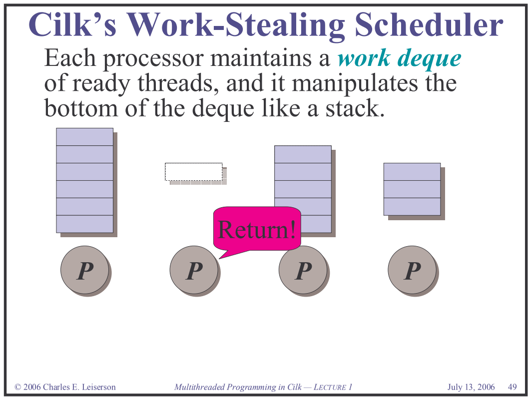
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



Cilk's Work-Stealing Scheduler

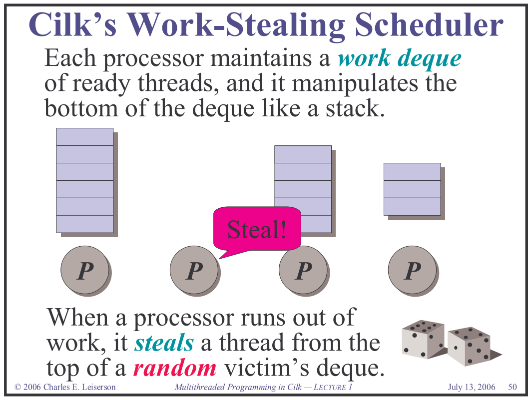
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.





Cilk's Work-Stealing Scheduler

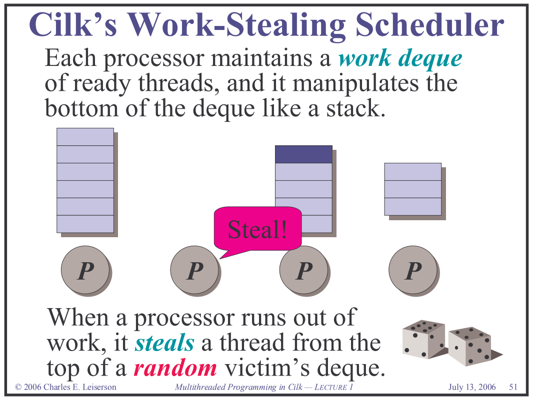
Each processor maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



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.

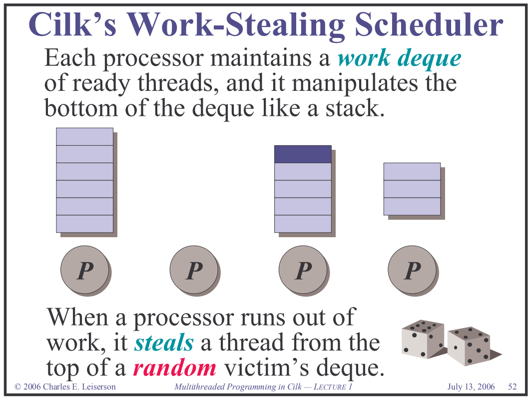
When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



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.

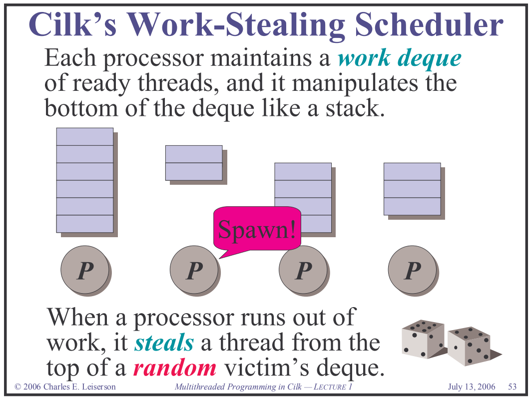
When a processor runs out of work, it ***steals*** a thread from the top of a ***random*** victim's deque.



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

$$T_P \le T_1/P + O(T_\infty)$$

on P processors.

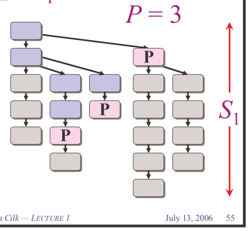
Pseudoproof. A processor is either ***working*** or ***stealing***. The total time all processors spend working is T_1. Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(PT_\infty)$. Since there are P processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty)\ .\ \blacksquare$$

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 $S_P \le PS_1$.

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

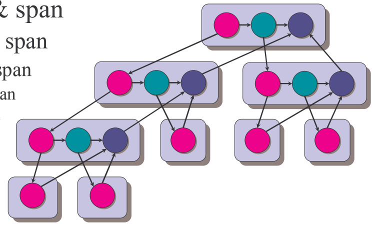
Better to steal parents than children!

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
- 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: speculative computing, mutual exclusion, race detection.