Please use the following citation format:


Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit: [http://ocw.mit.edu/terms](http://ocw.mit.edu/terms)
Lecture 15

Cilk
Design and Analysis of Dynamic Multithreaded Algorithms

Bradley C. Kuszmaul

MIT Computer Science and Artificial Intelligence Laboratory

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul
Shared-Memory Multiprocessor

- **Symmetric multiprocessor (SMP)**
- **Cache-coherent nonuniform memory architecture (CC-NUMA)**
Cilk

A C language for dynamic multithreading with a provably good runtime system.

Platforms
- Sun UltraSPARC Enterprise
- SGI Origin 2000
- Compaq/Digital Alphaserver
- Intel Pentium SMP’s

Applications
- virus shell assembly
- graphics rendering
- $n$-body simulation
- ★ Socrates and Cilkchess

Cilk automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling.
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.
Dynamic Multithreading

The \textit{computation dag} unfolds dynamically.

```
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);
  }
}
```

“Processor oblivious.”
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.
Advanced Features

• Returned values can be incorporated into the parent frame using a delayed internal function called an \textit{inlet}:

\begin{verbatim}
int y;
inlet void foo (int x) {
    if (x > y) y = x;
}
...
spawn foo(bar(z));
\end{verbatim}

• Within an inlet, the \texttt{abort} keyword causes all other children of the parent frame to be terminated.

• The \texttt{SYNCHED} pseudovariable tests whether a \texttt{sync} would succeed.

• A Cilk library provides \textit{mutex locks} for atomicity.

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul
Debugging Support

The **Nondeterminator** debugging tool detects and localizes data-race bugs.

“Abelian” Cilk program → Input data set

Information localizing a data race.

FAIL Image removed due to copyright restrictions. Arnold Schwarzenegger.

PASS Every scheduling produces the same result.

A **data race** occurs whenever a thread modifies location and another thread, holding no locks in common, accesses the location simultaneously.
Outline

- Theory and Practice
- A Chess Lesson
- Fun with Algorithms
- Work Stealing
- Opinion & 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

$$T_P = \text{execution time on } P \text{ processors}$$

$$T_1 = \textit{work}$$

$$T_\infty = \textit{critical path}$$
Algorithmic Complexity Measures

$T_P = \text{execution time on } P \text{ processors}$

$T_1 = \text{work}$

$T_\infty = \text{critical path}$

Lower Bounds

• $T_P \geq T_1/P$

• $T_P \geq T_\infty$
Algorithmic Complexity Measures

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

\[ T_1 = \text{work} \]

\[ T_\infty = \text{critical path} \]

Lower Bounds

\[ T_P \geq T_1 / P \]

\[ T_P \geq T_\infty \]

\[ T_1 / T_P = \text{speedup} \]

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

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$. 
**Greedy Scheduling**

**Theorem** [Graham & Brent]:
There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, …
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, …
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq \frac{T_1}{P} + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, execute $P$ of them.
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, execute $P$ of them. If fewer than $P$ tasks are ready, …
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with $T_P \leq T_1/P + T_\infty$.

**Proof.** At each time step, if at least $P$ tasks are ready, execute $P$ of them. If fewer than $P$ tasks are ready, execute all of them.
Greedy Scheduling

**Theorem** [Graham & Brent]: There exists an execution with \( T_P \leq T_1/P + T_\infty \).

**Proof.** At each time step, if at least \( P \) tasks are ready, execute \( P \) of them. If fewer than \( P \) tasks are ready, execute all of them.

**Corollary:** Linear speed-up when \( P \leq T_1/T_\infty \).
Cilk Performance

Cilk’s “work-stealing” scheduler achieves

\[ T_P = \frac{T_1}{P} + O(T_\infty) \] expected time (provably);

\[ T_P \approx \frac{T_1}{P} + T_\infty \] time (empirically).

Near-perfect linear speedup if \( P \leq \frac{T_1}{T_\infty} \).

Instrumentation in Cilk provides accurate measures of \( T_1 \) and \( T_\infty \) to the user.

The average cost of a \textit{spawn} in Cilk-5 is only 2–6 times the cost of an ordinary C function call, depending on the platform.
Outline

• Theory and Practice
• A Chess Lesson
• Fun with Algorithms
• Work Stealing
• Opinion & Conclusion

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

\[ T_P = T_\infty \]

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

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

measured speedup

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul
Socrates Speedup Paradox

**Original program**

\[ T_{32} = 65 \text{ seconds} \]

\[ T_{1} = 2048 \text{ seconds} \]

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

\[ T_{32} = \frac{2048}{32} + 1 = 65 \text{ seconds} \]

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

**Proposed program**

\[ T'_{32} = 40 \text{ seconds} \]

\[ T'_{1} = 1024 \text{ seconds} \]

\[ T'_{\infty} = 8 \text{ seconds} \]

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

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

• Theory and Practice
• A Chess Lesson
• Fun with Algorithms
• Work Stealing
• Opinion & Conclusion
Matrix Multiplication

\[
C = \begin{pmatrix}
  c_{11} & c_{12} & \cdots & c_{1n} \\
  c_{21} & c_{22} & \cdots & c_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  c_{n1} & c_{n2} & \cdots & c_{nn}
\end{pmatrix}
\]

\[
A = \begin{pmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{n1} & a_{n2} & \cdots & a_{nn}
\end{pmatrix}
\]

\[
B = \begin{pmatrix}
  b_{11} & b_{12} & \cdots & b_{1n} \\
  b_{21} & b_{22} & \cdots & b_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  b_{n1} & b_{n2} & \cdots & b_{nn}
\end{pmatrix}
\]

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}
\]
Recursive Matrix Multiplication

Divide and conquer on \( n \times n \) matrices.

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \times \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
A_{11}B_{11} & A_{11}B_{12} \\
A_{21}B_{11} & A_{21}B_{12}
\end{pmatrix} + \begin{pmatrix}
A_{12}B_{21} & A_{12}B_{22} \\
A_{22}B_{21} & A_{22}B_{22}
\end{pmatrix}
\]

8 multiplications of \((n/2) \times (n/2)\) matrices.
1 addition of \(n \times n\) matrices.
Matrix Multiplication in Cilk

\[ C = AB \]

\[ \text{cilk Mult(*C,*A,*B,n)} \]
\{ float T[n][n];
  \text{h base case & partition matrices i}
  \text{spawn Mult(C11,A11,B11,n/2);} \\
  \text{spawn Mult(C12,A11,B12,n/2);} \\
  \text{spawn Mult(C22,A21,B12,n/2);} \\
  \text{spawn Mult(C21,A21,B11,n/2);} \\
  \text{spawn Mult(T11,A12,B21,n/2);} \\
  \text{spawn Mult(T12,A12,B22,n/2);} \\
  \text{spawn Mult(T22,A22,B22,n/2);} \\
  \text{spawn Mult(T21,A22,B21,n/2);} \\
  \text{sync;}
  \text{spawn Add(C,T,n);} \\
  \text{sync;}
  \text{return;}
\}

\[ C = C + T \]

( Coarsen base cases for efficiency.)

\[ \text{cilk Add(*C,*T,n)} \]
\{ \text{h base case & partition matrices i}
  \text{spawn Add(C11,T11,n/2);} \\
  \text{spawn Add(C12,T12,n/2);} \\
  \text{spawn Add(C21,T21,n/2);} \\
  \text{spawn Add(C22,T22,n/2);} \\
  \text{sync;}
  \text{return;}
\}

© 2001-4 by Charles E. Leiserson and Bradley C. Kuszmaul
Analysis of Matrix Addition

cilk Add(*C,*T,n)
{
    if base case & partition matrices
    spawn Add(C11,T11,n/2);
    spawn Add(C12,T12,n/2);
    spawn Add(C21,T21,n/2);
    spawn Add(C22,T22,n/2);
    sync;
    return;
}

Work: \[ A_1(n) = 4 \cdot A_1(n/2) + (1) \]
\[ = (n^2) \]

Critical path: \[ A_\infty(n) = A_\infty(n/2) + (1) \]
\[ = (\lg n) \]

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul
Analysis of Matrix Multiplication

Work: \[ M_1(n) = 8M_1(n/2) + (n^2) \]
\[ = (n^3) \]

Critical path: \[ M_\infty(n) = M_\infty(n/2) + (\lg n) \]
\[ = (\lg^2 n) \]

Parallelism: \[ \frac{M_1(n)}{M_\infty(n)} = (n^3/\lg^2 n) \]

For 1000 \( \£ \) 1000 matrices, parallelism \( \frac{1}{4} \) 10\(^7\).
In modern hierarchical-memory microprocessors, memory accesses are so expensive that minimizing storage often yields higher performance.
No-Temp Matrix Multiplication

cilk Mult2(*C,*A,*B,n
{
  // C = C + A * B
  h base case & partition matrices i
  spawn Mult2(C11,A11,B11,n/2);
  spawn Mult2(C12,A11,B12,n/2);
  spawn Mult2(C22,A21,B12,n/2);
  spawn Mult2(C21,A21,B11,n/2);
  sync;
  spawn Mult2(C21,A22,B21,n/2);
  spawn Mult2(C22,A22,B22,n/2);
  spawn Mult2(C12,A12,B22,n/2);
  spawn Mult2(C11,A12,B21,n/2);
  sync;
  return;
}

Saves space at the expense of critical path.
Analysis of No-Temp Multiply

**Work:** \( M_1(n) = (n^3) \)

**Critical path:** \( M_\infty(n) = 2 M_\infty(n/2) + (1) = (n) \)

**Parallelism:** \( \frac{M_1(n)}{M_\infty(n)} = (n^2) \)

For 1000 £ 1000 matrices, parallelism \( \frac{1}{4} \times 10^6 \). Faster in practice.
Ordinary Matrix Multiplication

\[ c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \]

**IDEA:** Spawn \( n^2 \) inner products in parallel. Compute each inner product in parallel.

**Work:** \( (n^3) \)

**Critical path:** \( (\lg n) \)

**Parallelism:** \( (n^3/\lg n) \)

**BUT,** this algorithm exhibits poor locality and does not exploit the cache hierarchy of modern microprocessors.

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul
Outline

• Theory and Practice
• A Chess Lesson
• Fun with Algorithms
• Work Stealing
• Opinion & 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.

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.

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.

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul 45
Performance of Work-Stealing

**Theorem**: A work-stealing scheduler achieves an expected running time of

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

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 critical-path length by 1. Thus, the expected number of steals is \( O(PT_1) \). Since there are \( P \) processors, the expected time is \( (T_1 + O(PT_1))/P = T_1/P + O(T_1) \).
Outline

• Theory and Practice
• A Chess Lesson
• Fun with Algorithms
• Work Stealing
• Opinion & Conclusion
Data Parallelism

- High level
- Intuitive
- Scales up

- Conversion costs
- Doesn’t scale down
- Antithetical to caches
- Two-source problem
- Performance from tuned libraries

Example:

\[ C = A + B; \]
\[ D = A - B; \]

6 memory references, rather than 4.
Message Passing

- Scales up
- No compiler support needed
- Large inertia
- Runs anywhere

- Coarse grained
- Protocol intensive
- Difficult to debug
- Two-source problem
- Performance from tuned libraries

Shared memory: harder

Distributed memory: easier
distributed memory

In-core: easier

Out-of-core: harder
Conventional (Persistent) Multithreading

- Scales up and down
- No compiler support needed
- Large inertia
- Evolutionary

- Clumsy
- No load balancing
- Coarse-grained control
- Protocol intensive
- Difficult to debug

Parallelism for *programs*, not *procedures*.
Dynamic Multithreading

- High-level linguistic support for fine-grained control and data manipulation.
- Algorithmic programming model based on work and critical path.
- Easy conversion from existing codes.
- Applications that scale up and down.
- Processor-oblivious machine model that can be implemented in an adaptively parallel fashion.
- Doesn’t support a “program model” of parallelism.
Current Research

- We are currently designing jCilk, a Java-based language that fuses dynamic and persistent multithreading in a single linguistic framework.
- A key piece of algorithmic technology is an adaptive task scheduler that guarantees fair and efficient execution.
- Hardware transactional memory appears to simplify thread synchronization and improve performance compared with locking.
- The Nondeterminator 3 will be the first parallel data-race detector to guarantee both efficiency and linear speed-up.
Cilk Contributors

<table>
<thead>
<tr>
<th>Kunal Agarwal</th>
<th>Bradley Kuszmaul</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eitan Ben Amos</td>
<td>Charles E. Leiserson</td>
</tr>
<tr>
<td>Bobby Blumofe</td>
<td>Phil Lisiecki</td>
</tr>
<tr>
<td>Angelina Lee</td>
<td>Rob Miller</td>
</tr>
<tr>
<td>Ien Cheng</td>
<td>Harald Prokop</td>
</tr>
<tr>
<td>Mingdong Feng</td>
<td>Keith Randall</td>
</tr>
<tr>
<td>Jeremy Fineman</td>
<td>Bin Song</td>
</tr>
<tr>
<td>Matteo Frigo</td>
<td>Andy Stark</td>
</tr>
<tr>
<td>Michael Halbherr</td>
<td>Volker Strumpen</td>
</tr>
<tr>
<td>Chris Joerg</td>
<td>Yuli Zhou</td>
</tr>
</tbody>
</table>

...plus many MIT students and SourceForgers.

Image removed due to copyright restrictions. Silkworms.

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul  53
World Wide Web

Cilk source code, programming examples, documentation, technical papers, tutorials, and up-to-date information can be found at:

http://supertech.csail.mit.edu/cilk

Download CILK Today!
Research Collaboration

Cilk is now being used at many universities for teaching and research:

MIT, Carnegie-Mellon, Yale, Texas, Dartmouth Alabama, New Mexico, Tel Aviv, Singapore.

We need help in maintaining, porting, and enhancing Cilk’s infrastructure, libraries, and application code base. If you are interested, send email to:

cilk-support@supertech.lcs.mit.edu

Warning: We are not organized!
Cilk-5 Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>$T_1$</th>
<th>$T_\infty$</th>
<th>$T_1/T_\infty$</th>
<th>$T_1/T_S$</th>
<th>$T_S$</th>
<th>$T_1/T_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockedmul</td>
<td>1024</td>
<td>29.9</td>
<td>.0046</td>
<td>6783</td>
<td>1.05</td>
<td>4.29</td>
<td>7.0</td>
</tr>
<tr>
<td>notempmul</td>
<td>1024</td>
<td>29.7</td>
<td>.0156</td>
<td>1904</td>
<td>1.05</td>
<td>3.9</td>
<td>7.6</td>
</tr>
<tr>
<td>strassen</td>
<td>1024</td>
<td>20.2</td>
<td>.5662</td>
<td>36</td>
<td>1.01</td>
<td>3.54</td>
<td>5.7</td>
</tr>
<tr>
<td>queens</td>
<td>22</td>
<td>150.0</td>
<td>.0015</td>
<td>96898</td>
<td>0.99</td>
<td>18.8</td>
<td>8.0</td>
</tr>
<tr>
<td>cilksort*</td>
<td>4.1M</td>
<td>5.4</td>
<td>.0048</td>
<td>1125</td>
<td>1.21</td>
<td>0.9</td>
<td>6.0</td>
</tr>
<tr>
<td>knapsack</td>
<td>30</td>
<td>75.8</td>
<td>.0014</td>
<td>54143</td>
<td>1.03</td>
<td>9.5</td>
<td>8.0</td>
</tr>
<tr>
<td>lu</td>
<td>2048</td>
<td>155.8</td>
<td>.4161</td>
<td>374</td>
<td>1.02</td>
<td>20.3</td>
<td>7.7</td>
</tr>
<tr>
<td>cholesky*</td>
<td>1.02M</td>
<td>1427.0</td>
<td>3.4</td>
<td>420</td>
<td>1.25</td>
<td>208</td>
<td>6.9</td>
</tr>
<tr>
<td>heat</td>
<td>2M</td>
<td>62.3</td>
<td>.16</td>
<td>384</td>
<td>1.08</td>
<td>9.4</td>
<td>6.6</td>
</tr>
<tr>
<td>fft</td>
<td>1M</td>
<td>4.3</td>
<td>.002</td>
<td>2145</td>
<td>0.93</td>
<td>0.77</td>
<td>5.6</td>
</tr>
<tr>
<td>barnes-hut</td>
<td>65536</td>
<td>124.0</td>
<td>.15</td>
<td>853</td>
<td>1.02</td>
<td>16.5</td>
<td>7.5</td>
</tr>
</tbody>
</table>

All benchmarks were run on a Sun Enterprise 5000 SMP with 8 167-megahertz UltraSPARC processors. All times are in seconds, repeatable to within 10%.
Ease of Programming

<table>
<thead>
<tr>
<th></th>
<th>Original C</th>
<th>Cilk</th>
<th>SPLASH-2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lines</strong></td>
<td>1861</td>
<td>2019</td>
<td>2959</td>
</tr>
<tr>
<td><strong>diff</strong> lines</td>
<td>0</td>
<td>158</td>
<td>1098</td>
</tr>
<tr>
<td><strong>$T_1/T_8$</strong></td>
<td>1</td>
<td>7.5</td>
<td>7.2</td>
</tr>
<tr>
<td><strong>$T_1/T_S$</strong></td>
<td>1</td>
<td>1.024</td>
<td>1.099</td>
</tr>
<tr>
<td><strong>$T_S/T_8$</strong></td>
<td>1</td>
<td>7.3</td>
<td>6.6</td>
</tr>
</tbody>
</table>

Barnes-Hut application for 64K particles running on a 167-MHz Sun Enterprise 5000.
ICFP Programming Contest

• An 8-person Cilk team won **FIRST PRIZE** in the 1998 Programming Contest sponsored by the International Conference on Functional Programming.

• Our Cilk “*Pousse*” program was undefeated among the 49 entries. (Half the entries were coded in C.)

• Parallelizing our program to run on 4 processors took less than 1% of our effort, but it gave us more than a $3.5\times$ performance advantage over our competitors.

• The ICFP Tournament Directors cited Cilk as “*the superior programming tool of choice for discriminating hackers.*”

• For details, see: http://supertech.lcs.mit.edu/~pousse
Whither Functional Programming?

We have had success using functional languages to generate high-performance portable C codes.

- **FFTW**: *The Fastest Fourier Transform in the West* [Frigo-Johnson 1997]: 2–5¢ vendor libraries.
- Divide-and-conquer strategy optimizes cache use.
- A special-purpose compiler written in Objective CAML optimizes FFT dag for each recursive level.
- At runtime, FFTW measures the performance of various execution strategies and then uses dynamic programming to determine a good execution plan.

http://theory.lcs.mit.edu/~fftw
Sacred Cow
Compiling Cilk

Cilk source → cilk2c

source-to-source translator

C post source → gcc

C compiler

gcc → Cilk RTS

Cilk RTS

object code → ld

linking loader

ld → binary

A makefile encapsulates the process.

cilk2c translates straight C code into identical C postsource.
Cilk’s Compiler Strategy

The \texttt{cilk2c} compiler generates two “clones” of each procedure:

- **fast clone** — serial, common-case code.
- **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);
frame->entry = 1;
frame->n = n;
push(frame);
```

**C post-source**

```c
x = fib(n-1);
if (pop() == FAILURE)
{ frame->x = x;
  frame->join--;
  h clean up & return
to scheduler i
}
```

**Frame**

- `entry`
- `join`
- `n`
- `x`
- `y`

**Cilk deque**

- `suspend parent`
- `run child`
- `resume parent remotely`
Compiling **sync** (Fast Clone)

Cilk source

C post-source

**sync**;

cilk2c

<table>
<thead>
<tr>
<th>SLOW</th>
<th>FAST</th>
<th>FAST</th>
<th>FAST</th>
<th>FAST</th>
</tr>
</thead>
</table>

*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--;
    h clean up & return to scheduler
}
if (0) {
    L1:;
n = frame->n;
}
.
.
.
}

Cilk
deque

frame

entry

join

n

x

y

entry

join

restore

program

counter

same

as fast

clone

restore local

variables

if resuming

continue

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul 65
Breakdown of Work Overhead

<table>
<thead>
<tr>
<th>Processor</th>
<th>T₁/Tₛ</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS R10000</td>
<td>115ns</td>
</tr>
<tr>
<td>UltraSPARC I</td>
<td>113ns</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>78ns</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>27ns</td>
</tr>
</tbody>
</table>

Benchmark: fib on one processor.
cilk void Mergesort(int A[], int p, int r)
{
    int q;
    if (p < r)
    {
        q = (p+r)/2;
        spawn Mergesort(A,p,q);
        spawn Mergesort(A,q+1,r);
        sync;
        Merge(A,p,q,r);  // linear time
    }
}

\[ T_1(n) = 2 \ T_1(\frac{n}{2}) + (n) \]
\[ T_\infty(n) \equiv T_\infty(\frac{n}{2^\infty 4}) \]
\[ = (n) \]

**Parallelism:**
\[ \frac{(n \ lg n)}{(n)} = (lg n) \]

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul 67
Parallel Merge

Recursive merge

Binary search

Recursive merge

\[ T_1(n) = T_1(n) + T_1((1-\rho) n) + (\lg n), \text{ where } 1/4 \cdot \rho \cdot 3/4 = (n) \]

\[ T_\infty(n) = T_\infty(3n/4) + (\lg n) = (\lg^2 n) \]
Parallel Mergesort

\[ T_1(n) = 2 \cdot T_1(n/2) + (n) = (n \lg n) \]

\[ T_\infty(n) = T_\infty(n/2) + (\lg^2 n) = (\lg^3 n) \]

Parallelism:
\[ \frac{(n \lg n)}{(\lg^3 n)} = \frac{1}{(n/\lg^2 n)} \]

- Our implementation of this algorithm yields a 21% work overhead and achieves a 6 times speedup on 8 processors (saturating the bus).
- Parallelism of \((n/\lg n)\) can be obtained at the cost of increasing the work by a constant factor.
Student Assignment

Implement the fastest 1000 £ 1000 matrix-multiplication algorithm.

• **Winner:** A variant of *Strassen’s algorithm* which permuted the row-major input matrix into a bit-interleaved order before the calculation.

• **Losers:** Half the groups had *race bugs*, because they didn’t bother to run the Nondeterminator.

• **Learners:** Should have taught *high-performance C* programming first. The students spent most of their time optimizing the serial C code and little of their time Cilkifying it.
Caching Behavior

Cilk’s scheduler guarantees that \[ \frac{Q_P}{P} \cdot \frac{Q_1}{P} + O(\frac{MT_\infty}{B}) \], where \( Q_P \) is the total number of cache faults on \( P \) processors, each with a cache of size \( M \) and cache-line length \( B \).

Divide-and-conquer “cache-oblivious” matrix multiplication has \[ Q_1(n) = O(1 + n^3 / \sqrt{MB}) \], which is asymptotically optimal.

**IDEA:** Once a submatrix fits in cache, no further cache misses on its submatrices.