Design and Analysis of Dynamic Multithreaded Algorithms

Bradley C. Kuszmaul

MIT Computer Science and Artificial Intelligence Laboratory
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.
Fibonacci

```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

```c

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.
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);
    }
}

“The computation dag unfolds dynamically.”

“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`.)

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}:

```c
int y;
\textbf{inlet} void foo (int x) {
    if (x > y) y = x;
}
...
\textbf{spawn} foo(bar(z));
```

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

- The \textbf{SYNCHED} pseudovariable tests whether a \textit{sync} would succeed.

- A Cilk library provides \textit{mutex locks} for atomicity.
A **data race** occurs whenever a thread modifies a 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

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

$T_1 = \text{work}$
Algorithmic Complexity Measures

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

$$T_1 = \text{work}$$

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

- $T_P = T_1/P + O(T_\infty)$ expected time (provably);
- $T_P \approx T_1/P + T_\infty$ time (empirically).

Near-perfect linear speedup if $P \leq T_1/T_\infty$.

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

The average cost of a \texttt{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
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

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

\[
T_P = T_\infty
\]

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

-measured speedup-
Socrates Speedup Paradox

Original program

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

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

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

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

Proposed program

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

\[ T'_1 = 1024 \text{ seconds} \]

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

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

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

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

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

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

(Coarsen base cases for efficiency.)

\[ C = C + T \]

\[ C = AB \]
Analysis of Matrix Addition

```
cilk Add(*C, *T, n)
{
    h base case & partition matrices i
    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:** \[ A_1(n) = 4 A_1(n/2) + 1 \]  
(1)

**Critical path:** \[ A_\infty(n) \equiv A_\infty(n/2) + (\lg n) \]  
(1)
Analysis of Matrix Multiplication

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

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 \( \frac{1}{4} \times 10^7 \).
Stack Temporaries

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

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
    base case & partition matrices
    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;
}
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 \( \frac{1}{4} 10^6 \). 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:** \(\frac{n^3}{\lg n}\)

**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.

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.

![Diagram showing work deques and Spawn events](image-url)
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.

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**: 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] In-core

[harder] [easier] [harder] Out-of-core
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 \textit{jCilk}, a Java-based language that fuses dynamic and persistent multithreading in a single linguistic framework.

• A key piece of algorithmic technology is an \textit{adaptive task scheduler} that guarantees fair and efficient execution.

• \textit{Hardware transactional memory} appears to simplify thread synchronization and improve performance compared with locking.

• The \textit{Nondeterminator 3} will be the first parallel data-race detector to guarantee both efficiency and linear speed-up.
Cilk Contributors

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

...plus many MIT students and SourceForgers.
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

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>$T_8$</th>
<th>$T_1/T_8$</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>lines</td>
<td>1861</td>
<td>2019</td>
<td>2959</td>
</tr>
<tr>
<td>lines diff</td>
<td>0</td>
<td>158</td>
<td>1098</td>
</tr>
<tr>
<td>$T_{1}/T_{8}$</td>
<td>1</td>
<td>7.5</td>
<td>7.2</td>
</tr>
<tr>
<td>$T_{1}/T_{S}$</td>
<td>1</td>
<td>1.024</td>
<td>1.099</td>
</tr>
<tr>
<td>$T_{S}/T_{8}$</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× 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

cilk2c translates straight C code into identical C postsource.

A makefile encapsulates the process.

A makefile encapsulates the process.

Cilk source -> cilk2c -> Cilk postsource

C compiler: gcc

Cilk RTS

object code -> Cilk RTS

ld

binary

Cilk source

C post source

gcc

ld

binary

source-to-source translator

 linking loader

© 2001–4 by Charles E. Leiserson and Bradley C. Kuszmaul
Cilk’s Compiler Strategy

The *cilk2c* compiler generates two “clones” of each procedure:

- **fast clone** — serial, common-case code.
- **slow clone** — code with parallel bookkeeping.

- The *fast clone* is always spawned, saving live variables on Cilk’s work deque (shadow stack).
- The *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**

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

**Cilk deque**

```
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
}
```

```
C post-source
```

```
entry
join
n
x
y
entry
join
```
Compiling **sync** (Fast Clone)

- Cilk source
- sync
- ;
- cilk2c

| SLOW | FAST | FAST | FAST | FAST |

No synchronization overhead in the fast clone!
Compiling the Slow Clone

```c
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;
    }
    ..
}
```

- `frame`: entry & join
- `n`, `x`, `y`: variables
- `Cilk deque`: same as fast clone
- `restore`: clean up variables if resuming
- `continue`: continue as fast clone
- `restore local variables`: clean up & return to scheduler
- `entry join`: same as fast clone
Breakdown of Work Overhead

Benchmark: fib on one processor.
Mergesorting

```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
    }
}
```

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

\[
T_\infty(n) \equiv T_\infty(n/\log n)
\]

\[
= (n)\]

Parallelism:

\[
\frac{(n \log n)}{(n)} = (\log n)
\]
Parallel Merge

Recursive merge

Binary search

Recursive merge

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

\[ = (n) \]

\[ T_\infty(n) = T_\infty(\frac{3n}{4}) + (\lg n) \]

\[ = (\lg^2 n) \]
Parallel Mergesort

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

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

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

- 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.
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
\[ \frac{Q_p}{P} \cdot \frac{Q_1}{P} + O(MT_\infty /B) \],
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.