Multithreaded Programming in Cilk

Charles E. Leiserson
MIT
Cilk

A C language for programming dynamic multithreaded applications on shared-memory multiprocessors.

- virus shell assembly
- graphics rendering
- $n$-body simulation
- heuristic search
- dense and sparse matrix computations
- friction-stir welding simulation
- artificial evolution

Cilk’s provably good runtime system automatically manages low-level aspects of parallel execution, including protocols, load balancing, and scheduling.
Fibonacci

Cilk code

```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 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);
    }
}
```

- **cilk** 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.
Cilk’s Thread Scheduler

- Cilk’s randomized “work-stealing” scheduler load-balances the computation at run-time.
- A mathematical proof guarantees near-perfect linear speed-up on applications with sufficient parallelism, as long as the architecture has sufficient memory bandwidth.
- A Cilk program running on 1 processor typically exhibits less than 2% slowdown compared with its C version.
- A spawn/return in Cilk is over 450 times faster than a Pthread create/exit and less than 3 times slower than an ordinary C function call on a modern Intel processor.
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.
Debugging

The *Nondeterminator* debugging tool provably guarantees to detect and localize data-race bugs.

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

- The **inlet** keyword specifies an internal function that can be called to incorporate a returned result into the parent frame in a nonstandard way when a spawned child returns.
- The **abort** keyword forces all spawned children to terminate abruptly.
- The **SYNCHED** pseudovariable tests whether a **sync** would succeed.
- A **Cilk** library provides **mutex locks** for atomicity. (Alas, no **transactional memory**!)
OUTLINE

• Overview
• Cilk Performance
• A Chess Lesson
• Alpha-Beta Search
• 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

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

\[ T_1 = \text{work} \]

\[ T_\infty = \text{span} \]
Algorithmic Complexity Measures

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

\[ T_1 = \text{work} \]

\[ T_\infty = \text{span} \]

Lower Bounds

\[ T_P \geq T_1/P \]

\[ T_P \geq T_\infty \]
Algorithmic Complexity Measures

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

\[ T_1 = \text{work} \]

\[ T_\infty = \text{span} \]

Lower Bounds

\[ T_P \geq T_1 / P \]

\[ T_P \geq T_\infty \]

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

\[ T_1 / T_\infty = \text{parallelism} \]
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 \( T_P \leq T_1/P + T_\infty \).

**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 $P \ll T_1/T_\infty$. 
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).

- Linear speedup if \( P \ll T_1/T_{\infty} \) (parallelism).

- Instrumentation in *Cilk* allows the user to determine accurate measures of \( T_1 \) and \( T_{\infty} \).

- The average cost of a spawn in *Cilk* is only 2–6 times the cost of an ordinary C function call, depending on the platform.
### Cilk Benchmarks (c. 1997)

<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>0.0046</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>0.0156</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>0.5662</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>0.0015</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>0.0048</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>0.0014</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>0.4161</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>0.16</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>0.002</td>
<td>2145</td>
<td>0.93</td>
<td>0.77</td>
<td>5.6</td>
</tr>
</tbody>
</table>

**barnes-hut** | 65536 | 124.0 | 0.15       | 853            | 1.02      | 16.5  | 7.5       |

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%. 

© 2007 by Charles E. Leiserson
HPC Challenge Benchmarks

Performance of the Cilk code rivals or exceeds that of the hand-coded MPI.

Cilk won the 2006 HPC Challenge Class 2 Award for Best Overall Productivity. Cilkifying all six benchmarks required 137 keywords.
## Productivity

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MPI SLoC</th>
<th>Cilk SLoC</th>
<th>Cilk CMulticore</th>
<th>Dist. to</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM</td>
<td>658</td>
<td>58</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>PTRANS</td>
<td>2261</td>
<td>81</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>RandomAccess</td>
<td>1883</td>
<td>123</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>HPL</td>
<td>15608</td>
<td>348</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>DGEMM</td>
<td>184 †</td>
<td>97</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>FFTE</td>
<td>230</td>
<td>1747</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

† MPI DGEMM uses the HPL parallel matrix multiplication.
Fast Fourier Transform

1024x1024 Complex FFT (Double Precision)

GFLOPS/s

Number of Cores

Performance of *Cilk*’s FFTW competes with Intel’s hand-tuned MKL FFT.
OUTLINE

• Overview
• Cilk Performance
• A Chess Lesson
• Alpha-Beta Search
• 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

\[
T_P = T_\infty
\]

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

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

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

measured speedup
**Socrates Speedup Paradox**

**Original program**

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

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

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

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

\[ = 65 \text{ seconds} \]

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

\[ = 5 \text{ seconds} \]

**Proposed program**

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

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

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

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

\[ = 40 \text{ seconds} \]

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

\[ = 10 \text{ seconds} \]

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

• Overview
• Cilk Performance
• A Chess Lesson
• Alpha-Beta Search
• Cilk’s Scheduler
• Conclusion
Cilk’s Inlet Mechanism

Returned values can be incorporated into the parent frame using a delayed internal function called an *inlet*.

```
int max = -INF;
int max_index = 0;

inlet void update_max ( int x, int index )
{
    if (x > max)
    {
        max = x;
        max_index = index;
    }
}

for (i=0; i<1000000; ++i)
{
    update_max ( spawn bar(i), i );
}

sync;
```

*Cilk* provides *implicit atomicity* among the threads belonging to the same frame.
Min-Max Search

- Two players: MAX □ and MIN ●.
- The game tree represents all moves from the current position within a given search depth.
- At leaves, apply a static evaluation function.
- MAX chooses the maximum score among its children.
- MIN chooses the minimum score among its children.
Alpha-Beta Pruning

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
Idea: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*. 

**Alpha-Beta Pruning**
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Alpha-Beta Pruning**

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff*. 
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Alpha-Beta Pruning**

**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Alpha-Beta Pruning**

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
Alpha-Beta Pruning

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
Alpha-Beta Pruning

IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
Alpha-Beta Pruning

**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
Alpha-Beta Pruning

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Alpha-Beta Pruning**

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Alpha-Beta Pruning**

**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
Alpha-Beta Pruning

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — *beta cutoff.*
**Alpha-Beta Pruning**

**Idea:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — **beta cutoff**.

*Unfortunately, this heuristic is inherently serial.*
Observation: In a best-ordered tree, the degree of every internal node is either 1 or maximal.

Idea: [Feldman-Mysliwietz-Monien 91] If the first child fails to generate a cutoff, speculate that the remaining children can be searched in parallel without wasting any work: “young brothers wait.”
cilk int search( position *prev, int move, int depth )
{
    position cur;      /* current position */
    int bestscore = -INF; /* best score so far */
    int num_moves;     /* number of children */
    int mv;            /* index of child */
    int sc;            /* child’s score */
    int cutoff = FALSE; /* have we seen a cutoff? */

    ● View from MAX’s standpoint; MIN’s viewpoint can be obtained by negating scores — negamax.
    ● The current position is generated by the child, not by the parent.
    ● The alpha and beta limits and the move list are fields of the position structure.
Parallel Alpha-Beta (II)

```c
inlet void get_score(int child_sc) {
    child_sc = -child_sc;  /* negamax */

    if (child_sc > bestscore) {
        bestscore = child_sc;
        if (child_sc > cur.alpha) {
            cur.alpha = child_sc;
            if (child_sc >= cur.beta) {
                cutoff = TRUE;  /* no need to search further */
                abort;  /* terminate other children */
            }
        }
    }
}
```

The `abort` keyword causes all spawned children of the frame to terminate abruptly.
/* create current position and set up for search */

make_move(prev, move, &cur);

sc = eval(&cur);    /* static evaluation */

if ( abs(sc) >= MATE || depth <= 0 )    /* leaf node */
{
  return (sc);
}

cur.alpha = -prev->beta;    /* negamax */
cur.beta = -prev->alpha;

/* generate moves, hopefully in best-first order*/

num_moves = gen_moves(&cur);
Parallel Alpha-Beta (IV)

```c
/* search the moves */

for (mv=0; !cutoff && mv<num_moves; mv++)
{
    get_score( spawn search(&cur, mv, depth-1) );
    if ((mv==0) sync; /* young brothers wait */
}

sync; /* this sync is outside the loop so that the searches after the first execute in parallel */

return (bestscore);
}
```
“Cilkifying” Alpha-Beta

Only 6 Cilk keywords need be embedded in the C program to parallelize it.

In fact, the program can be parallelized using only 5 keywords at the expense of minimal obfuscation.

Typically, the major programming challenge for “Cilkifying” an application is to express the program recursively, as opposed to actually parallelizing it.
OUTLINE

- Overview
- Cilk Performance
- A Chess Lesson
- Alpha-Beta Search
- 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.

![Diagram showing work deque and spawning]

*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.

```plaintext
P
|
| Spawn!
| P
|
| P
|
| P
|
| Spawn!
| P
|
```

© 2007 by Charles E. Leiserson

Multithreaded Programming in Cilk

April 2, 2007
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.

![Diagram showing work deques and a steal operation]

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

\[ T_P \leq \frac{T_1}{P} + O(T_1) \]

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 \( \frac{1}{P} \) chance of reducing the span by 1. Thus, the expected number of steals is \( O(PT_1) \). Since there are \( P \) processors, the expected time is

\[ \frac{(T_1 + O(PT_1))}{P} = \frac{T_1}{P} + O(T_1) \]
OUTLINE

• Overview
• Cilk Performance
• A Chess Lesson
• Alpha-Beta Search
• Cilk’s Scheduler
• Conclusion
Summary

• Cilk provides a simple parallel programming model that extends C in a faithful manner.

• Cilk scales down, as well as up.

• Cilk offers a release strategy for parallel software that supports traditional regression testing.
  □ Debug and test the serial elision.
  □ Certify absence of races using the Nondeterminator.

• Cilk encourages divide-and-conquer recursion, which is provably cache-friendly.

• Cilk is a processor-oblivious language that allows adaptive scheduling in multiprogramming environments.
Cilk Arts, Inc.

- Headquartered in Lexington, Massachusetts.
- Focused on the multicore opportunity.
- Currently prefunded.
- Launch product: *Cilk++* (~12 months).
- Business model: license the runtime platform.
- URL: [www.cilk.com](http://www.cilk.com).
Cilk++ Features

- C++.
- Windows and Linux (and other Unixes).
- Remove the limitation that only parallel functions can spawn.
- Allow spawning of arbitrary statement blocks.
- Loop syntax to avoid recoding loops as divide-and-conquer.
- Support for exceptions (à la JCilk).
Cilk Contributors

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