Programming in CILK

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07

But first, some course logistics
Next week’s lectures 9/24 and 9/26

- Homework 2: Research and present (10-15 minutes) an array language or library next week

<table>
<thead>
<tr>
<th>Language / Library</th>
<th>Presentation lead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Arrays</td>
<td>Eric H</td>
</tr>
<tr>
<td>NumPy, DistNumPy</td>
<td>Michael</td>
</tr>
<tr>
<td>HTA</td>
<td>Scott</td>
</tr>
<tr>
<td>HPF</td>
<td>Eric L</td>
</tr>
<tr>
<td>BoxLib, Combo,…</td>
<td>Anshu</td>
</tr>
<tr>
<td>PBLAS</td>
<td>Richard</td>
</tr>
<tr>
<td>Titanium, (UPC)</td>
<td>Mangpo</td>
</tr>
<tr>
<td>Arrays in Electronic Structures (TCE)</td>
<td>Edgar</td>
</tr>
<tr>
<td>RDD</td>
<td>TD</td>
</tr>
<tr>
<td>X10</td>
<td>Philip</td>
</tr>
<tr>
<td>Chapel, Cilk, Sequoia, Legion</td>
<td></td>
</tr>
</tbody>
</table>
Array Library Considerations

• How general is array creation/declarations?
  • 0-based, 1-based, arbitrary, strided, multi-dimensional, etc.
• Which subarray operations are supported?
  • Slice, contiguous, noncontiguous, strided, indexed,…
• What type of layouts are allowed?
  • Blocked, cyclic, arbitrary block-cyclic
  • Multi-dimensional
• What type of overlap is allowed (e.g., ghost regions)
• Is there hierarchical layout control?
• Are unions of rectangles (AMR) supported?
• Are indexed (sparse matrices) supported?
• How is communication done?
  • Fill in ghost region
  • Update tailing submatrix
Programming in CILK

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07

Many slides from Charles Leiserson as noted
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
Fibonacci Example: Creating Parallelism

```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.
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 \textit{Cilk procedure}, capable of being spawned in parallel.

The named \textit{child} Cilk procedure can execute in parallel with the \textit{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)

processors are virtualized

The computation dag unfolds dynamically.
Algorithmic Complexity Measures

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

\[ T_1 = \text{work} \]

\[ T_1 = \text{span}^* \]

*Also called critical-path length or computational depth.

\[ T_P \geq T_1 / P \]

\[ T_P \geq T_1 \]

© 2006 Charles E. Leiserson
Speedup

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

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

\( T_1/T_1 = \text{parallelism} \)

\( = \text{the average amount of work per step along the span.} \)
Example: \texttt{fib(4)}

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

\textbf{Work: } \quad T_1 = 17

\textbf{Span: } \quad T_1 = 8

\textbf{Parallelism: } \quad T_1 / T_1 = 2.125

Using many more than 2 processors makes little sense.
Parallelizing Vector Addition

C

```c
void vadd (real *A, real *B, int n){
    int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```
Parallelizing Vector Addition

```c
void vadd (real *A, real *B, int n){
    if (n<=BASE) {
        int i; for (i=0; i<n; i++) A[i]+=B[i];
    } else {
        void vadd (real *A, real *B, int n){
            vadd (A, B, n/2);
            vadd (A+n/2, B+n/2, n-n/2);
        }
        vadd (A+n/2, B+n/2, n-n/2);
    }
}
```

Parallelization strategy:
1. Convert loops to recursion.
Parallelizing Vector Addition

**Cilk**

```c
void vadd (real *A, real *B, int n){
    if (n<=BASE) {
        int i; for (i=0; i<n; i++) A[i]+=B[i];
    } else {
        vadd (A, B, n/2;
        vadd (A+n/2, B+n/2, n-n/2;
        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 n) {
    if (n <= BASE) {
        int i; for (i = 0; i < n; i++) A[i] += B[i];
    } else {
        spawn vadd (A, B, n/2);
        spawn vadd (A+n/2, B+n/2, n-n/2);
        sync;
    }
}

© 2006 Charles E. Leiserson
Vector Addition Analysis

To add two vectors of length $n$, where $\text{BASE} = \Theta(1)$:

**Work:** $T_1 = \Theta(n)$

**Span:** $T_1 = \Theta(\lg n)$

**Parallelism:** $T_1 / T_1 = \Theta(n/\lg n)$
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.
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 \( 1/P \) chance of reducing the span by 1. Thus, the expected cost of all steals is \( O(PT_1) \). Since there are \( P \) processors, the expected time is

\[ \frac{(T_1 + O(PT_1))}{P} = \frac{T_1}{P} + O(T_1) \]
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 $S_P \cdot PS_1$.

**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.

© 2006 Charles E. Leiserson
Cilk vs. PThreads

How will the following code execute in Pthreads? In Cilk?

```c
for (i=1; i<1000000000; i++) {
    spawn-or-fork foo(i);
}
sync-or-join;
```

What if `foo` contains code that waits (e.g., spins) on a variable being set by another instance of `foo`?

This different is a liveness property:
- Cilk threads are spawned lazily, “may” parallelism
- PThreads are spawned eagerly, “must” parallelism
Definition. A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

Example

```c
int x = 0;
cilk_for(int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```

Slide source: John Gilbert
Avoiding Races

- All the iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.

**Ex.**

```cilk_spawn
qsort(begin, middle);
qsort(max(begin + 1, middle), end);
cilk_sync;
```

**Note:** The arguments to a spawned function are evaluated in the parent before the spawn occurs.
Cilk++ Reducers

- Hyperobjects: reducers, holders, splitters
- Primarily designed as a solution to global variables, but has broader application

```cpp
int result = 0;
cilk_for (size_t i = 0; i < N; ++i) {
    result += MyFunc(i);
}
```

```cpp
#include <reducer_opadd.h>
...
cilk::hyperobject<cilk::reducer_opadd<int> > result;
cilk_for (size_t i = 0; i < N; ++i) {
    result() += MyFunc(i);
}
```

Data race!

Race free!

This uses one of the predefined reducers, but you can also write your own reducer easily

Slide source: John Gilbert
Hyperobjects under the covers

• A reducer `hyperobject<T>` includes an associative binary operator \( \otimes \) and an identity element.

• Cilk++ runtime system gives each thread a private view of the global variable.

• When threads synchronize, their private views are combined with \( \otimes \).
Cilkscreen

- **Cilk**screen runs off the binary executable:
  - Compile your program with `-fcilkscreen`
  - Go to the directory with your executable and say `cilkscreen your_program [options]`
  - **Cilk**screen prints info about any races it detects

- **Cilk**screen **guarantees** to report a race if there exists a parallel execution that could produce results different from the serial execution.

- It runs about 20 times slower than single-threaded real-time.

*Slide source: John Gilbert*
A data race occurs whenever two logically parallel threads, holding no locks in common, access the same location and one of the threads modifies the location.
Cilk’s Memory Model

Programmers may also synchronize through memory using lock-free protocols, although Cilk is agnostic on consistency model.

• If a program contains no data races, Cilk effectively supports sequential consistency.
• If a program contains data races, Cilk’s behavior depends on the consistency model of the underlying hardware.

To aid portability, the Cilk_fence() function implements a memory barrier on machines with weak memory models.
Compiling Cilk

Cilk source \rightarrow \text{cilk2c} \rightarrow \text{C post-source} \rightarrow \text{gcc} \rightarrow \text{object code} \rightarrow \text{ld} \rightarrow \text{binary}

The \text{cilkc} compiler encapsulates the process.

\text{cilk2c} translates straight C code into identical C postsource.

The C compiler encapsulates the process.
Cilk’s Compiler Strategy

The `cilk2c` translator generates two “clones” of each Cilk 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 \texttt{spawn} — Fast Clone

\textit{Cilk source}

\begin{center}
x = \texttt{spawn fib(n-1)};
\end{center}

\texttt{cilk2c}

\begin{center}
\begin{verbatim}
frame-&gt;entry = 1;
frame-&gt;n = n;
push(frame);
\end{verbatim}
\end{center}

\begin{center}
\begin{verbatim}
x = fib(n-1);
\end{verbatim}
\end{center}

\textit{C post-source}

\begin{center}
\begin{verbatim}
if (pop()==FAILURE) {
  frame-&gt;x = x;
  frame-&gt;join--;
  h clean up &
  return to scheduler
}
\end{verbatim}
\end{center}

\textit{Cilk deque}

\begin{itemize}
\item suspend parent
\item run child
\item resume parent remotely
\end{itemize}
Compiling \texttt{sync} — Fast Clone

\texttt{Cilk source} \hspace{1cm} \texttt{sync;} \hspace{1cm} \texttt{cilk2c} \hspace{1cm} \begin{tabular}{|c|c|c|c|}
\hline
SLOW \hline
FAST \hline
FAST \hline
FAST \hline
FAST \hline
\end{tabular}

\textit{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

- **Cilk deque**
  - entry
  - join

© 2006 Charles E. Leiserson
Breakdown of Work Overhead 
(circa 1997)

Benchmark: fib on one processor.
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**

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

\[
T_P = T_\infty
\]

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

measured speedup

\[
\frac{P}{T_1/T_\infty}
\]
Cilk quick history

- 1994 - Cilk 1
- 1998 - Cilk 5
- 2005 - JCilk
- 2006 - Cilk Arts
- 2008 - Cilk++
- 2009 - Intel buys Cilk Arts
- 2010 - Intel released a commercial implementation in its compilers