Design philosophy

• Integrate with C and C++, uses language extensions
• Target shared memory machines
• User identifies and specifies parallelism, Cilk manages it
  • User identifies function invocations that can execute independently - spawn
• Cilk generates the code to support the parallelism
• Synchronization is available to control parallel execution
• Cilk maintains a work queue to efficiently exploit parallelism
frames or stack frames

Stack frames are essential to modern (i.e. since the early 1960s) function invocation

Allow storage to be created that is
• local to an invocation in sequential programs
• automatically removed when the invocation leaves

Allows separate invocations of a function to have a unique identity

Supports recursion and clean returns from deep chains of function calls
Let’s look at a simple Cilk function

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 keyword identifies this as a Cilk function operating under Cilk rules

But before talking about how Cilk would execute this, let’s review how this would be executed sequentially given the call fib(2);

Each invocation of fib has its own stack frame, and so there is a frame created for fib(2)

Because of line 06, space is created on the frame for x and y

At line 08 a new frame is created and fib(1) is called

Execution begins for fib(1). After 03 is executed, the value of n (1) is placed into fib(2)’s x variable

Execution continues to 09, the value 0 is placed into y, the values are added and returned to the return variable at the call to fib(2)
Let’s look at a simple Cilk function

```cilk
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06        int x, y;
07        x = spawn fib (n-1);
08        y = spawn fib (n-2);
09        sync;
10        return (x+y);
11     }
12 }
```
What synchs are for

```c
01 cilk int fib (int n)  
02 {  
03     if (n < 2) return n;  
04     else  
05     {  
06         int x, y;  
07         x = spawn fib (n-1);  
08         y = spawn fib (n-2);  
09         sync;  
10         return (x+y);  
11     }  
12 }  
```

After spawning `fib(1)` and `fib(0)` execution proceeds to the `sync` statement at line 11.

The `sync` statement stops processing until all function invocations spawned by this function (fib(2)) with this frame have reached it. It is a form of barrier.

The `sync` ensures that both spawns have returned before the `return` in statement 13 is executed. Not doing this would create a race and an incorrect program.
How synchs work

```cilk
01 cilk int fib (int n)  
02 {  
03     if (n < 2) return n;  
04     else  
05     {  
06         int x, y;  
07  
08         x = spawn fib (n-1);  
09         y = spawn fib (n-2);  
10  
11         sync;  
12  
13         return (x+y);  
14     }  
15 }
```

After spawning `fib(1)` and `fib(0)` execution proceeds to the `sync` statement at line 11.

The `sync` statement stops processing until all function invocations spawned by this function (fib(2)) with this frame have reached it. It is a form of barrier.
How synchs work

```cilk
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else {
05         int x, y;
06
07         x = spawn fib (n-1);
08         y = spawn fib (n-2);
09
10         sync;
11     }
12     return (x+y);
13 }
14 }
```

After spawning `fib(1)` and `fib(0)` execution proceeds to the `sync` statement at line 11.

The `sync` statement stops processing until all function invocations spawned by this function (fib(2)) with this frame have reached it. It is a form of barrier.

The synch in `fib(3)`
How synchs work

```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);
    }
}
```

After spawning `fib(1)` and `fib(0)` execution proceeds to the `sync` statement at line 11.

The `sync` statement stops processing until all function invocations spawned by this function (`fib(2)`) with this frame have reached it. It is a form of barrier.
How synchs work

01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06         int x, y;
07
08         x = spawn fib (n-1);
09         y = spawn fib (n-2);
10
11         sync;
12
13         return (x+y);
14     }
15 }

After spawning \texttt{fib(1)} and \texttt{fib(0)} execution proceeds to the \texttt{sync} statement at line 11.

The \texttt{sync} statement stops processing until \textit{all} function invocations spawned by \texttt{this} function (\texttt{fib(2)}) \textit{with this frame} have reached it. It is a form of barrier.
Inlets are Cilk constructs that process return values before they are returned.

Inlets always execute atomically.

Cilk normally requires a procedure to be spawned as a separate statement and continues with its execution, this rule is relaxed for inlets.

fib(n-1) is invoked;
the parent continues executing after the inlet.

When fib(n-1) returns its thread passes control to summer

When summer is finished, the thread that executed it waits at the sync.
Implicit Inlets

Give a way of expressing reductions, etc. succinctly

Cannot be mixed with explicit inlets, i.e.

```
x += summer(spawn(fib(n-1))
```

would not be legal

**aborts**

Cilk allows an **abort** statement to appear in an inlet -- it kills all spawned threads of the parent procedure

- They do not die instantly
- They may terminate normally, and return a value

It is up to the user to handle these situations

```c
short int fib (short n) {
    short x = 0;
    if (n<2) return n;
    else {
        x+= spawn fib (n-1));
        x += spawn fib (n-2));
        sync;
        return (x);
    }
```
Scheduling

- In a sequential execution, when executing a spawn, Cilk will
  - push the frame and program counter of the parent onto a stack
  - execute the spawned procedure
  - dequeue the parent frame and continue its execution
Scheduling - parallel execution

- Each processor maintains a deque or double ended queue
- When a function is spawned any frames that need to be suspended are placed on the deque
- The processor owning the deque can only remove frames from the end it inserted them
- Other processors may remove from the other end
Scheduling - parallel execution

• When an function is spawned
  • place the parent onto the bottom of the deque/“stack”
  • execute the spawned function, which may place itself onto the bottom of the stack if it spawns functions
• When the function returns, pop work off of the bottom (the frame of the parent of the spawned function)
• If a thread is idle, take work off of the top of the deque
Sequential stack example

```
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);
    }
}
```
Sequential stack example

```
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 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 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);
    }
}
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else {
05         int x, y;
06         x = spawn fib (n-1);
07         y = spawn fib (n-2);
08         sync;
09         return (x+y);
10     }
11 }
12
13
14
15
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06         int x, y;
07         x = spawn fib (n-1);
08         y = spawn fib (n-2);
09         sync;
10         return (x+y);
11     }
12 }
13 }
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 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);
    }
}
Sequential stack example

```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);
    }
}
```

fib(4)

fib(3)

fib(2)

fib(1)

fib(0)

fib(1)

fib(0)

Monday, April 24, 17
Sequential stack example

```
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 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);
  }
}
Work stealing example

If there is an idle thread $T_i$, the Cilk scheduler will take work off of the top of the queue and give it to that thread.
Work stealing example

T₀ queue

T₁ queue

active T₀

fib(1)

fib(2)

fib(3)

fib(4)

fib(0)

fib(1)

fib(2)

fib(0)

fib(1)

fib(2)

fib(0)

fib(1)

fib(2)

fib(0)
Work stealing example

T₀ queue

T₁ queue

fib(4)
fib(3)
fib(2)
fib(2)
fib(0)
fib(1)
fib(1)

fib(4)
fib(3)
fib(2)
fib(1)

active T₀

active T₁

Monday, April 24, 17
Work stealing example

```plaintext
T0 queue

|   |   |   |   | 2 | 3 |

T1 queue

|   |   |   |   |   | 1 |
```

```
1
2
3
4
5
6
7
8
9
```

```
active T0

fib(1)  fib(0)  fib(2)  fib(3)  fib(4)
```

```
active T1

fib(1)  fib(0)  fib(2)
```

```
top bottom
```

```
Monday, April 24, 17
```
Parallel Programming with Cilk Plus

Arch D. Robison
Load Balancing and Locality
Race-Free \neq \text{Deterministic}

Parallel programs

Deterministic

Race free

\textbf{Thread 1} \hspace{1cm} \textbf{Thread 2}

\begin{align*}
\text{x} &= 1; \\
m.\text{lock}(); \\
\text{x} &= 1; \\
m.\text{unlock}(); \\
\end{align*}

\begin{align*}
\text{x} &= 1; \\
m.\text{lock}(); \\
\text{x} &= 2; \\
m.\text{unlock}(); \\
\end{align*}
Thread 1
a.lock();
b.lock();
++A;
--B;
b.unlock();
a.unlock();

Thread 2
b.lock();
a.lock();
--B;
++A;
a.unlock();
b.unlock();
## Philosophy of Cilk Plus

<table>
<thead>
<tr>
<th>Division of Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programmer</strong></td>
</tr>
<tr>
<td>Specify what <em>can</em> run in parallel.</td>
</tr>
<tr>
<td>Make parallelism easy to express.</td>
</tr>
<tr>
<td>Provide much more <em>potential</em> parallelism than system can use.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Express SIMD opportunities.</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Avoid races.</td>
</tr>
<tr>
<td>Minimize use of locks.</td>
</tr>
<tr>
<td>Promote locality via cache-oblivious style.</td>
</tr>
</tbody>
</table>
Style Issue

// Bad Style

cilk_spawn f();
cilk_spawn g();
// nop
cilk_sync;

// Preferred style

cilk_spawn f();
g();
// nop
cilk_sync;

Wasted fork
Serial Elision

Cilk keywords can be trivially eliminated:

```c
#define cilk_spawn
#define cilk_sync
#define cilk_for for
```

Resulting program is called the **serial elision**

- It is a valid serial C/C++ program!

Likewise, the serial elision is always a valid implementation of a Cilk program:

- Means a Cilk program can always run on a single thread.
- Fundamental requirement for avoiding oversubscription.
Races

Race
• Two unordered memory references and at least one is a write.

Cilk program is deterministic if:
• It has no races
• It uses no locks
• Reducer operations are associative

Deterministic Cilk program has same effect as its serial elision.

Will talk about automatic race detection later.

Floating-point + and * are almost associative.
Effective Cilk Plus: Writing Scalable Programs

Work-span model of complexity
Load balancing
Amortizing scheduling overhead
Hazards of locks
Hyperobjects revisited
Correctness tools survey
DAG Model of Computation

Program is a directed acyclic graph (DAG) of tasks

The hardware consists of workers

Scheduling is greedy

• No worker idles while there is a task available.
Work-Span Model

$T_p = \text{time to run with } P \text{ workers}$

$T_1 = work$
- time for serial execution
- sum of all work

$T_\infty = span$
- time for critical path
Work-Span Example

\[ T_1 = \text{work} = 7 \]
\[ T_\infty = \text{span} = 5 \]
Burdened Span

Includes extra cost for synchronization

Often dominated by cache line transfers.
Lower Time Bound on Greedy Scheduling

(Implies upper bound on speedup)

Work-Span Limit

$$\max(\frac{T_1}{P}, T_\infty) \leq T_P$$
Upper Time Bound on Greedy Scheduling

(Implies lower bound on speedup)

Brent’s Lemma

\[ T_P \leq (T_1 - T_\infty)/P + T_\infty \]
Load Balancing by Work-stealing

Each processor has a deque of spawned tasks.

When each processor has work to do, a spawn is roughly the cost of about 25 function calls.
Load Balancing by Work-stealing

![Diagram showing load balancing with work-stealing]

- Initial state: Process (P) with spawned tasks.
- Process returns a spawn message.
- Process gets new spawn task.
- Process (P) continues to spawn tasks.

*Other brands and names are the property of their respective owners.*
Load Balancing by Work-stealing

Return!
Work-stealing task scheduler

P

P

P

P

spawn
spawn
spawn
spawn
spawn
spawn
spawn

Steal!

spawn
spawn
spawn
spawn
spawn
spawn
spawn

spawn
spawn
spawn
spawn
spawn

Copyright© 2012, Intel Corporation. All rights reserved.
*Other brands and names are the property of their respective owners.
Work-stealing task scheduler

With sufficient parallelism, the steals are rare, and we get **linear speedup** (ignoring memory effects).
OpenMP Tactics to Unlearn
(Thanks to James Cownie for List)

1. Creating one work item per thread.
2. Anything involving `omp_get_thread_num()`.
3. Fear of nested parallelism.
Problem with One Work Item Per Thread

.destroys composability
- No way to know if running as child or sibling of other parallel work.

Hurts load balancing.
- Gives scheduler no parallel slack.

Advice: Choose grain size based on amortizing scheduling overhead, not balancing load.
Problem with Using Thread Ids

Thus thread id can change in surprising ways.

- Id after spawn can be different than before spawn.
- Id after sync can be different than before spawns.

Race, because i==j!

Advice: Use hyperobjects (reducers and holders).
Embrace Nested Parallelism

Cilk was designed for nested parallelism.

Unused nested parallelism is inexpensive.

• Execution is serial when all threads are busy.
Performance Tools

Intel® Cilk™ View
• Automatic work-span analysis for Cilk™ Plus

Intel® Amplifier
• General threading analysis
• Good for spotting hardware-related bottlenecks
Sample Cilk View Output

Uses *burdened span* that estimates scheduling costs.
Two Race Detectors for Cilk Plus

Intel® Cilk Screen
- “Happens before” on strands + “Lock set”
- Theoretically efficient implementation that strict fork-join nature of Cilk

Intel® Parallel Inspector
- “Happens before” on threads + “Lock set”
- Also detects potential deadlock
- Also has memory checker
- GUI integrates into Visual Studio

Both based on “Pin” dynamic instrumentation technology.
http://www.pintool.org/
Cilk Screen Example

```c
void f() {
    int x[10];
    cilk_for( int i=0; i<10; ++i )
        x[i] = pseudo_random();
}
```

```c
int pseudo_random() {
    static int state = 1;
    return state = a*state+b;
}
```

$ icc -g randomfill.cpp$ cilkscreen a.out

Cilkscreen Race Detector V2.0.0, Build 2516

Race condition on location 0x600b84
write access at 0x40062b: (/tmp/randomfill.cpp:7, pseudo_random+0x19)
read access at 0x40061a: (/tmp/randomfill.cpp:7, pseudo_random+0x8)
called by 0x2b2156f08b07: (__$U0+0xc7)
called by 0x2b2156f08848: (cilk_for_recursive<unsigned int, void *>(void*, unsigned int, unsigned int)>+0x128)
called by 0x2b2156f086b8: (__$U1+0xb8)
called by 0x2b2156f082c5: (cilk_for_root<unsigned int, void *>(void*, unsigned int, unsigned int)>+0x135)
called by 0x2b2156f0818a: (__cilkrts_cilk_for_32+0xa)
### Philosophy of Cilk Plus

#### Division of Responsibilities

<table>
<thead>
<tr>
<th>Programmer</th>
<th>Cilk Plus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify what <em>can</em> run in parallel.</td>
<td>Make parallelism easy to express. Enable clean composition.</td>
</tr>
<tr>
<td>Provide much more <em>potential</em> parallelism than system can use.</td>
<td>Throttle <em>actual</em> parallelism.</td>
</tr>
<tr>
<td>Express SIMD opportunities.</td>
<td>Make SIMD easy to express. Generate SIMD code.</td>
</tr>
<tr>
<td>Avoid races.</td>
<td>Synchronize strands of execution.</td>
</tr>
<tr>
<td>Minimize use of locks.</td>
<td>Provide hyperobjects.</td>
</tr>
<tr>
<td>Promote locality via cache-oblivious style.</td>
<td>Depth-first serial execution.</td>
</tr>
</tbody>
</table>
URLs

Cilk Plus home page
• http://cilkplus.org

Cilk Plus Forum

Cilk Plus Specifications

Intel® Cilk™ Plus Software Development Kit
  - Cilk Screen Race Detector
  - Cilk View Scalability Analyzer

GCC 4.7 Branch
• http://gcc.gnu.org/svn/gcc/branches/cilkplus/

Intel ® Parallel Inspector
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR
IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on
Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using
specific computer systems, components, software, operations and functions. Any change to any of
those factors may cause the results to vary. You should consult other information and performance
tests to assist you in fully evaluating your contemplated purchases, including the performance of that
product when combined with other products.

Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and Cilk
are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that
are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and
other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on
microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended
for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for
Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information
regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Where to get Cilk

- Cilk Arts was Charles Leiserson’s company to commercialize Cilk
- Acquired by Intel in 2009
- In September 2010 released by Intel as Intel Cilk Plus
  - adds support for reductions
  - simplifies the language
  - debugger integration
- Spec published, and Intel is encouraging other vendors to support the language