Intel Cilk™ Plus
-Cilk keywords, HyperObjects
Agenda

• What is Cilk?
• cilk_spawn & cilk_sync
• cilk_for
• Some programming notes
• Reducers
What is Cilk?

• C\C++ Extension to support parallelism

• 3 key words
  – cilk_spawn
  – cilk_sync
  – cilk_for

• Parallel intent not Parallel control
History

- MIT Cilk
  - over 15 years

- Cilk Arts, Inc.
  - Extended Cilk to C++

- Intel Cilk
  - Simplified build \ programming experience.
## A Family of Parallel Programming Models

### Developer Choice

<table>
<thead>
<tr>
<th>Choice of high-performance parallel programming models</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Libraries for pre-optimized and <em>parallelized functionality</em></td>
</tr>
<tr>
<td>• Intel® Cilk™ Plus and Intel® Threading Building Blocks supports composable parallelization of a wide variety of applications.</td>
</tr>
<tr>
<td>• OpenCL* addresses the needs of customers in specific segments, and provides developers an additional choice to maximize their app performance</td>
</tr>
<tr>
<td>• MPI supports distributed computation, combines with other models on nodes</td>
</tr>
</tbody>
</table>

### Intel® Cilk™ Plus
- C/C++ language extensions to simplify parallelism
- Open sourced
- Also an Intel product

### Intel® Threading Building Blocks
- Widely used C++ template library for parallelism
- Open sourced
- Also an Intel product

### Domain-Specific Libraries
- Intel® Integrated Performance Primitives
- Intel® Math Kernel Library

### Established Standards
- Message Passing Interface (MPI)
- OpenMP*
- Coarray Fortran
- OpenCL*

### Research and Development
- Intel® Concurrent Collections
- Offload Extensions
- Intel® Array Building Blocks
- Intel® SPMD Parallel Compiler

---

*Other brands and names are the property of their respective owners.*
Programmers view of Cilk

• Key words
  
  #include <cilk/cilk.h>
  cilk_spawn
  cilk_sync
  cilk_for

• API

  __cilkrts_set_param("nworkers", "4")
  __cilkrts_get_nworkers()
  __cilkrts_get_total_workers()
  __cilkrts_get_worker_number()

• Environment variable

  CILK_NWORKERS
Agenda

• What is Cilk?
• cilk_spawn & cilk_sync
• cilk_for
• Some programming notes
• Reducers
Cilk keywords

• Cilk adds three keywords to C and C++:
  _cilk_spawn
  _cilk_sync
  _cilk_for

• If you #include <cilk/cilk.h>, you can write the keywords as cilk_spawn, cilk_sync, and cilk_for.
Anatomy of a spawn

void f()
{
    cilk_spawn g();
    work
    work
    work
cilk_sync;
    work
}

void g()
{
    work
    work
    work
}

spawned function (child)

spawning function (parent)

continuation

sync
Serial Execution

void f()
{
    g();
    work
    work
    work
    work
}

void g()
{
    work
    work
    work
}

*Other brands and names are the property of their respective owners.
Work Stealing when no other worker is available

```c
void f()
{
    cilk_spawn g();
    work
    work
    work
    work
    cilk_sync;
    work
}
```

```c
void g()
{
}
```

Worker A

Same behavior as serial execution!
Work Stealing when another worker is available

```c
void f()
{
    cilk_spawn g();
    work
    work
    work
    work
    cilk_sync;
}
```

```c
void g()
{
    work
    work
    work
    work
}
```

Worker A

Worker B

Worker ?

steal!
A note on Load Balancing

The Cilk scheduler automatically load balances
Work-stealing Overheads

• **Spawning** is cheap (3-5 times the cost of a function call)

• **Stealing** is much more expensive (requires locks and memory barriers)

• **Most spawns do not result in steals.**

• **Balanced work loads** ➔ less stealing ➔ less overhead.
LAB Activity 1

cilk_spawn and cilk_sync
Agenda

• What is Cilk?
• cilk_spawn & cilk_sync
• cilk_for
• Some programming notes
• Reducers
cilk_for loop

• Looks like a normal for loop.
  ```cilk_for```
  ```
  (int x = 0; x < 1000000; ++x) { ... }
  ```

• Any or all iterations may execute in parallel with one another.

• All iterations complete before program continues.

• Constraints:
  – Limited to a single control variable.
  – Must be able to jump to the start of any iteration at random.
  – Iterations should be independent of one another.
Implementation of cilk_for

cilk_for (int i=0; i < 8; ++i)
f(i);
cilk_for vs. serial for with spawn

• Compare the following loops:

```c
for (int x = 0; x < n; ++x) { cilk_spawn f(x); }
cilk_for (int x = 0; x < n; ++x) { f(x); }
```

• The above two loops have similar semantics, but...

• they have very different performance characteristics.
If work per iteration is small then steal overhead can be significant
cilk_for: Divide and Conquer

Divide and conquer results in few steals and less overhead.
cilk_for examples

cilk_for (int x; x < 1000000; x += 2) { ... }

cilk_for (vector<int>::iterator x = y.begin();
   x != y.end(); ++x) { ... }

cilk_for (list<int>::iterator x = y.begin();
   x != y.end(); ++x) { ... }
Implementation of cilk_for

- The compiler transforms a cilk_for loop body into a function then spawns that function using divide-and-conquer recursion.

```c
void __body(int beg, int n) {
    #pragma intel ivdep
    for (int j=beg; j<beg+n; ++j)
        f(j);
}
__cilkrts_cilk_for(0,n,__body);
```

```c
void __cilkrts_cilk_for(int start, int n, void (*body)(int,int))
{
    if (n > grainsize) {
        int half = n / 2;
        cilk_spawn __cilkrts_cilk_for(start, half, body);
        __cilkrts_cilk_for(start + half, n - half, body);
    }
    else
        body(start, n);
}
```
Implicit syncs

```c
void f() {
    cilk_spawn g();
    cilk_for (int x = 0; x < lots; ++x) {
        ...
    }
    try {
        cilk_spawn h();
    }
    catch (...) {
        ...
    }
}
```

- **At end of a cilk_for body (does not sync g())**
- **At end of a try block containing a sync**
- **At end of a try block containing a spawn**
- **At end of a spawning function**
Agenda

• What is Cilk?
• cilk_spawn & cilk_sync
• cilk_for
• Some programming notes
• Reducers
Grain Size and cilk_for

• spawn overhead can have impact on trivial loops

• reduce the overhead by letting a spawn do more work

    #pragma cilk grainsize = expression
cilk_for (…)

• Default grain size will yield good performance in most cases.

• Pragma is normally used to set grain size to 1 for large, unbalanced loops.
Serialization

• Achieved by removing keywords by
  
  - /Qcilk-serialize or -cilk-serialize

• Same as running with one worker

• Useful for debugging.
LAB Activity 2

Parallelizing using cilk_for
Cilk and the Worker Count

- Default num workers == num hardware threads.
  - Earlier versions used to use num cores

- Environment variable CILK_NWORKERS

- Api __cilkrts_set_param("nworkers","5");
Agenda

• What is Cilk?
• cilk_spawn & cilk_sync
• cilk_for
• Some programming notes
• Reducers
Reducers

• Used to avoid data races

• Can be extended by programmer

• Good for adding to legacy code.
Reducer Views

1. At a spawn, the child receives the parent’s view.

2. After a spawn, the continuation receives either the view from before the spawn or a new view initialized to the identity, *nondeterministically*, depending on whether the continuation was stolen.

3. At (or before) a sync, the views of the child and parent are reduced (merged).
Reducer Views

when there is no steal

```cpp
cilk::reducer_opadd<int> sum(3);

void f()
{
    cilk_spawn g();
    work
    sum += 2;
    work
    cilk_sync;
    work
}
```

```cpp
void g()
{
    work
    sum++;
    work
}
```

Same behavior as serial execution!
Reducer Views
when continuation is stolen

cilk::reducer_opadd<int> sum(3);

void f()
{
    cilk_spawn g();
    work
    sum += 2;
    work
    cilk_sync;
    work
}

void g()
{
    work
    sum++;  
    work
}

34

Worker A
Worker B
Worker ?
A Taste of Reducers in C

```c
#include <cilk/reducer_opadd.h>

long sum_array(long data[], size_t n)
{
    CILK_C_REDUCER_OPADD(r, int, 0);
    CILK_C_REGISTER_REDUCER(r);

cilk_for (int j = 0; j < n; ++j)
    REDUCER_VIEW(*r) += data[j];

    CILK_C_UNREGISTER_REDUCER(r);
return r.value;
}
```

Macros and idioms are needed to make up for C’s lack of templates and overloading.

In serial code, reducer’s final value is accessible, even after reducer has been unregistered.
LAB Activity 3

Experiments with scalability
Reducer Library

Cilk’s hyperobject library contains many commonly used reducers:

- `reducer_list_append`
- `reducer_list_prepend`
- `reducer_max`
- `reducer_max_index`
- `reducer_min`
- `reducer_min_index`
- `reducer_opadd`
- `reducer_ostream`
- `reducer_basic_string`
- ...

You can also write your own using `cilk::monoid_base` and `cilk::reducer`. 
Creating Your Own Reducer

1. Choose a type, a reduction operation, and an identity value.
   - This combination is known as a monoid
   - The reduction operation must be associative, but need not be commutative.
2. Declare a monoid class derived from cilk::monoid_base.
3. Instantiate cilk::reducer with your monoid.
4. Optionally create a wrapper class for type safety.
**Algebraic Monoids**

**Definition.** A *monoid* is a triple \((T, \otimes, e)\), where

- \(T\) is a set,
- \(\otimes\) is an **associative** binary operator,
- \(e\) is an **identity** for \(\otimes\).

**Examples:**

- \((\text{int}, +, 0)\)
- \((\text{double}, *, 1)\)
- \((\text{bool}, &&, \text{true})\)
- \((\text{std::list, concat, empty})\)
- \((\text{int, max, INT_MIN})\)

\[
\begin{align*}
\text{Associative} & \quad a \otimes (b \otimes c) = (a \otimes b) \otimes c \\
\text{Identity} & \quad a \otimes e = e \otimes a = a
\end{align*}
\]
Representing Monoids

In Cilk, we represent a monoid over $T$ by a C++ class $M$ that inherits from $\text{cilk}\::\text{monoid\_base}<T>$ and defines
- a static or const member function $\text{reduce}()$ that implements the binary operator $\otimes$ and
- a static or const member function $\text{identity}()$ that constructs a fresh identity, e.

**Example:**

```cpp
struct sum_monoid : cilk::monoid_base<int> {
    static void reduce(int* left, int* right) {
        *left += *right; // store result into *left
    }
    static void identity(int* p) {
        new (p) int(0); // placement construction
    }
};
```
Defining a Reducer

A reducer over `sum_monoid` may now be defined as follows:

```cpp
cilk::reducer<sum_monoid> x;
```

The local view of `x` can be accessed as `x()`.

```cpp
x() += value;
```

**NOTE** From v13 of the compiler, the `*` operator is preferred:

```cpp
*x += value;
```
Reducer Wrappers

**Issues**

- It is generally inconvenient to replace every access to $x$ in a legacy code base with $x()$.
- Accesses to $x$ are not safe. Nothing prevents a programmer from writing "$x() *= 2$", even though the reducer is defined over $+$.  

> A wrapper class solves these problems.

```cpp
class sum_reducer {
    cilk::reducer<sum_monoid> r;

public:
    sum_reducer(int v = 0) : r(v) {} 
    sum_reducer& operator+=(int v) { r() += v; }
    sum_reducer& operator++() { r()++; }
    int get_value() const { return r(); }
    ... 
};
```

For v13 of compiler, use *r rather than  r()
Implicit syncs

The return value of a function is copied before implicit destructors and before the implicit sync. An explicit sync may be needed to prevent races.

```c
class C {
  ...
};

void f()
{
  C x;
  int r = cilk_spawn g(&x);
  work
  work
cilk_sync;
  return r;

  __retval = r;
  x.~C();
  cilk_sync;
  return __retval
}
```

- **explicit sync** solves the races
- **original statement**
- **compiler – generated implicit code**
- race on `r` between `g()` and return
- race on `x` between `g` and destructor
LAB Activity 4

Using Reducers in C
Cilk Plus Holders

• Similar to reducers, but do not preserve the views beyond the parallel strand.
• One view is maintained on basis of the holder policy (see next slide).
• Holders can be treated as a kind of Thread Local Storage (TLS).
• Using Cilk Plus Holders along with custom wrappers are a great way to reduce the editing effort when using Cilk Plus with legacy code.
Cilk Plus Holders – what view?

• One view is maintained dependent the holder policy
  – holder_keep_indeterminate
  – holder_keep_last
  – holder_keep_last_copy
  – holder_keep_last_swap
  – holder_keep_last_move

NB: Default value already defined in the holder template

```
template < typename Type,
           holder_policy Policy = holder_keep_indeterminate,
           typename Allocator = std::allocator< Type >
> class holder { // etc. };
```
Using Cilk Plus Holders

- **Step 1.** Include the header file `holder.h`

- **Step 2.** Declare your variables to be holders

- **Step 3.** Access holder view using the function operator

```c++
#include <cilk/holder.h>

cilk::holder<int> g;

void test1()
{
    int i;
    g() = 8;
    cilk_spawn[]
    {
        g() = 100;
        i = g();
    }();
    g() = 37;
    cilk_sync;
}
```

For v13 of compiler, use `*g` rather than `g()`.

*Other brands and names are the property of their respective owners.*
Using a custom wrapper
(similar to reducer wrappers in slide 42)

```cpp
// template for wrapper
template < typename T >
class myholder
{
private:
    cilk::holder < T > m_holder;
public:
    myholder < T > & operator = 
        (const T &rhs)
    {
        m_holder() = rhs;
        return   *this;
    }

    operator T &()    
    {
        return m_holder();
    }
};

// code that uses the wrapper
myholder < int > h;
void test2()
{
    int i;
    h = 8;
    cilk_spawn[]
    {
        h = 100;
        i = h;
    }();
h = 37;
cilk_sync;

    For v13 of compiler,
    use "*m_holder"
    rather than
    m_holder()
}
```

Can you see the advantage of this approach?
Summary

• Cilk is a simple extension to C and C++ for shared-memory fork-join parallelism.
• Cilk’s serial semantics and simple syntax does not obscure the program logic.
• Cilk’s work-stealing scheduler automatically load-balances if there sufficient parallelism.
• Cilk is suitable for both loop and recursive (divide an conquer) parallelism.
• Cilk is an excellent choice for parallelizing both new and legacy C and C++ software.
Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools 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 Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110228
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.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.


*Other names and brands may be claimed as the property of others.

Copyright © 2011. Intel Corporation.

http://intel.com/software/products