6.172
Performance Engineering of Software Systems

LECTURE 8
Reducer and Holder Hyperobjects

Charles E. Leiserson

October 4, 2011
Global Variable Considered Harmful

1973 — *Historical perspective*
*Wulf & Shaw*: “We claim that the non-local variable is a major contributing factor in programs which are difficult to understand.”

2010 — *Today’s reality*
Nonlocal variables are used extensively, in part because they avoid *parameter proliferation* — long argument lists to functions for passing numerous, frequently used variables.

Global and other nonlocal variables can inhibit parallelism by inducing *race bugs.*
Coping with Race Bugs

- Although *locking* can “solve” some race bugs, *lock contention* can destroy all parallelism.
- Manually, making *local copies* of the nonlocal variables can remove contention, but at the cost of restructuring program logic.
- Cilk provides *hyperobjects*, such as *reducers* and *holders*, to mitigate data races on nonlocal variables without the need for locks or code restructuring.

**IDEA:** Different strands may see different *views* of the hyperobject.
int compute(const X& v);
int main()
{
    const int n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    for (int i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
               << result
               << std::endl;
    return 0;
}
int compute(const X& v);
int main()
{
    const int n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    cilk_for (int i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
int compute(const X& v);
int main()
{
    const int n = 1000000;
    extern X myArray[n];
    // ...

    mutex L;
    int result = 0;
    cilk_for (int i = 0; i < n; ++i)
    {
        int temp = compute(myArray[i]);
        L.lock();
        result += temp;
        L.unlock();
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
int compute(const X& v);
int main()
{
    const std::int ARRAY_SIZE = 1000000;
    extern X myArray[ARRAY_SIZE];
    // ...
    int result = 0;
    cilk::reducer_ptr<add_monoid<int>> p(&result);
    
    cilk_for (int i = 0; i < ARRAY_SIZE; ++i)
    {
        *p += compute(myArray[i]);
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}

Declare `result` to be a summing reducer accessed via the hyperpointer `p`.

Updates are resolved automatically without races or contention.

At the end, the underlying value of `result` value reflects all the parallel updates.
• The Notion of Reducers
• Programming with Reducers
• A Real–World Example
• Graph Theory
• Semantics of Reducers
• Holders
Intuition for Reducers

- Defining a \textit{reducer pointer} $p$ that “points to” an object $x$ hyperizes $x$ and makes it a \textit{reducer}.
- The reducer is defined over an \textit{associative} operation, such as addition, maximum, minimum, AND, OR, etc.
- Strands can update $*p$ as if it were an ordinary nonlocal object, but the reducer is, in fact, maintained as a collection of worker-local \textit{views}.
- The Cilk runtime system coordinates the views and combines them when appropriate.
- When only one view of the reducer remains, the hyperized object $x$ reflects all the prior updates to $*p$.

**Example:** summing reducer

- $*p$: 42
- $*p$: 14
- $*p$: 33
- $x$: 89
If you don’t “look” at the intermediate values, the result is *determinate*, because addition is *associative*.
### Conceptual Behavior

<table>
<thead>
<tr>
<th>original</th>
<th>equivalent</th>
<th>equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>x = 1;</td>
<td>x = 1;</td>
</tr>
<tr>
<td>x += 3;</td>
<td>x += 3;</td>
<td>x += 3;</td>
</tr>
<tr>
<td>x++;</td>
<td>x++;</td>
<td>x++;</td>
</tr>
<tr>
<td>x += 4;</td>
<td>x += 4;</td>
<td>x += 4;</td>
</tr>
<tr>
<td>x++;</td>
<td>x++;</td>
<td>x1 = 0;</td>
</tr>
<tr>
<td>x += 5;</td>
<td>x += 5;</td>
<td>x1 += 5;</td>
</tr>
<tr>
<td>x += 9;</td>
<td></td>
<td>x1 += 9;</td>
</tr>
<tr>
<td>x -= 2;</td>
<td></td>
<td>x1 -= 2;</td>
</tr>
<tr>
<td>x += 6;</td>
<td></td>
<td>x1 += 6;</td>
</tr>
<tr>
<td>x += 5;</td>
<td></td>
<td>x1 += 5;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x += x1;</td>
</tr>
</tbody>
</table>

If you don’t “look” at the intermediate values, the result is **determinate**, because addition is **associative**.
Related Work

- **OpenMP’s reduction construct**
  - Tied to parallel for loop.
- **TBB’s parallel reduce template**
  - Tied to loop construct.
- **Data–parallel (APL, NESL, ZPL, etc.) reduction**
  - Tied to the vector operation.
- **Google’s MapReduce**
  - Tied to the map function.

In contrast, Cilk reducers are not tied to any control or data structure. They can be named anywhere (globally, passed as parameters, stored in data structures, etc.). Wherever and whenever they are dereferenced, they produce the local view.
• The Notion of Reducers
• Programming with Reducers
• A Real–World Example
• Graph Theory
• Semantics of Reducers
• Holders
Definition. A monoid is a triple \((T, \otimes, e)\), where
- \(T\) is a set,
- \(\otimes\) is an associative binary operator on elements of \(T\),
- \(e \in T\) is an identity element for \(\otimes\).

Examples:

- \((\mathbb{Z}, +, 0)\)
- \((\mathbb{R}, \times, 1)\)
- \((\{\text{TRUE, FALSE}\}, \land, \text{TRUE})\)
- \((\Sigma^*, \|, \epsilon)\)
- \((\mathbb{Z}, \text{MAX}, -\infty)\)
Representing Monoids

A Cilk programmer can represent a monoid on type-\(T\) objects by creating a C++ class that inherits from \texttt{cilk::monoid\_base<\texttt{T}>} and defines

- a member function \texttt{reduce()} that implements the binary operator \(\otimes\),
- a member function \texttt{identity()} that constructs a fresh identity \(e\), and
- other updating operations.

\textbf{Example}

```cpp
class sum_monoid_int : cilk::monoid_base<int> {
    static void reduce(int* left, int* right) {
        *left += *right; // order is important!
    }
    static void identity(int* p) {
        new (p) int(0);
    }
};
```
A reducer pointer `px` over `sum_monoid` that designates an object `x` as a reducer can now be defined in terms of `sum_monoid_int`:

```
int x = 0;
cilk::reducer_ptr<sum_monoid_int> px(&x);
```

In a parallel region, the *hyperized* object should normally only be accessed through a reducer pointer, e.g., `*px += 42`. This operation actually accesses the local view of the reducer for the strand that executes it.

The hyperized object can be safely accessed (e.g., `std::cout << x;`) when no parallel views exist.
Local views can be manipulated directly:

```cpp
int x = 0;
cilk::reducer_ptr<sum_monoid_int> px(&x);
cilk_for (int i = 0; i < n; ++i)
{
    int pre = *px;
    *px = pre + i;        // Same thing as *px += i.
    int post = *px;
    assert((post - pre) == i);  // This works...
}
```

Careful: One can accidentally manipulate a reducer in a way that is inconsistent with the associative operation for the monoid, e.g., writing `*px *= 2` even though the reducer is defined over `+`.

A wrapper class can solve this problem.
Reducer Library

Cilk’s hyperobject library defines reducer templates for many commonly used monoids:

- `add_reducer_ptr<T>`: sum elements of type `T`.*
- `and_reducer_ptr<T>`: bitwise AND of elements of type `T`.
- `or_reducer_ptr<T>`: bitwise OR of elements of type `T`.
- `max_reducer_ptr<T>`: maximum of elements of type `T`.
- `min_reducer_ptr<T>`: minimum of elements of type `T`.
- `list_append_reducer_ptr<T>`: add to the end of a list whose elements are of type `T`.

But it’s not hard to “roll your own” using `cilk::monoid_base` and `cilk::reducer`.

*Behavior is nondeterministic when used with floating-point numbers.
• The Notion of Reducers
• Programming with Reducers
• A Real–World Example
• Graph Theory
• Semantics of Reducers
• Holders
A mechanical assembly is represented as a tree of subassemblies down to individual parts.

Collision-detection problem: Find all “collisions” between an assembly and a target object.
**Goal:**
Create a list of all the parts in a mechanical assembly that collide with a given target object.

```cpp
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)
{
    switch (x->kind) {
        case Node::LEAF:
            if (target->collides_with(x))
            {
                output_list.push_back(x);
            }
            break;
        case Node::INTERNAL:
            for (Node::const_iterator child = x.begin(); child != x.end(); ++child)
            {
                walk(child);
            }
            break;
    }
}
```
Idea:
Parallelize the search by using a `cilk_for` to search all the children of each internal node in parallel.*

Oops!

Node *target;
std::list<Node *> output_list;
...
void walk(Node *x) {
    switch (x->kind) {
        case Node::LEAF:
            if (target->collides_with(x)) {
                output_list.push_back(x);
            }
            break;
        case Node::INTERNAL:
            cilk_for (Node::const_iterator child = x.begin();
                child != x.end();
                ++child)
            {
                walk(child);
            }
            break;
    }
}

____________
*Node::const_iterator must be a random-access iterator.

© 2011 Charles E. Leiserson
Problem:
The global variable `output_list` is updated in parallel, causing a race bug.

```cpp
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)
{
    switch (x->kind) {
    case Node::LEAF:
        if (target->collides_with(x))
        {
            output_list.push_back(x);
        }
        break;
    case Node::INTERNAL:
        cilk_for (Node::const_iterator child = x.begin();
                  child != x.end(); ++child)
        {
            walk(child);
        }
        break;
    }
}
```
**Idea**

Define the `walk()` routine to return a list of nodes that collide with the target `x`. After walking a node’s children recursively in parallel, return the concatenation of their lists.

**Problems**

- The signature of the `walk()` routine must be changed.
- The `cilk_for` loop must be rewritten by hand as parallel divide-and-conquer.
**A Mutex Solution**

**Locking:**
Each leaf locks `output_list` to ensure that updates occur atomically. Unfortunately, lock contention inhibits speed-up. Also, the list is produced in a jumbled order.

```cpp
Node *target;
std::list<Node *> output_list;
mutex output_list_mutex;
...
void walk(Node *x)
{
    switch (x->kind) {
    case Node::LEAF:
        if (target->collides_with(x))
            {
                output_list_mutex.lock();
                output_list.push_back(x);
                output_list_mutex.unlock();
            }
        break;
    case Node::INTERNAL:
        cilk_for (Node::const_iterator child = x.begin();
            child != x.end();
            ++child)
        {
            walk(child);
        }
        break;
    }
}
```
Reducer Solution

Hyperize `output_list` with a reducer pointer `p_list` whose `reduce()` function concatenates lists.* The `output_list` is produced in the same order as in the original C++.

```cpp
Node *target;
std::list<Node *> output_list;
cilk::list_append_reducer_ptr<Node*> p_list(&output_list);
...
void walk(Node *x)
{
    switch (x->kind) {
    case Node::LEAF:
        if (target->collides_with(x))
        {
            p_list->push_back(x);
        }
        break;
    case Node::INTERNAL:
        cilk_for (Node::const_iterator child = x.begin();
        child != x.end();
        ++child)
        {
            walk(child);
        }
        break;
    }
}
```

*List concatenation is associative with the empty list as the identity.*
• The Notion of Reducers
• Programming with Reducers
• A Real–World Example
• Graph Theory
• Semantics of Reducers
• Holders
Series Relations

Definition. A strand \( s_1 \) **(logically) precedes** another strand \( s_2 \), denoted \( s_1 \prec s_2 \), if there exists a path from \( s_1 \) to \( s_2 \) in the computation dag. We also say that \( s_2 \) **follows** (or **succeeds**) \( s_1 \), written \( s_2 \succ s_1 \).

Example: \( a \prec b \).

Definition. Two strands \( s_1 \) and \( s_2 \) are in **series** if either \( s_1 \prec s_2 \) or \( s_2 \prec s_1 \).
**Parallel Relation**

**Definition.** A strand $s_1$ *logically parallels* another strand $s_2$, denoted $s_1 \parallel s_2$, if no path exists from $s_1$ to $s_2$ nor from $s_2$ to $s_1$ in the computation dag.

**Example:** $c \parallel d$. 
**Tetrachotomy Lemma.** For any two strands $s_1$ and $s_2$, exactly one of the following holds:

- $s_1 = s_2$,
- $s_1 \parallel s_2$,
- $s_1 \prec s_2$, or
- $s_1 \succ s_2$.

\[\square\]

**Transitivity Lemma.** For any three strands $s_1, s_2,$ and $s_3$, we have $s_1 \prec s_2$ and $s_2 \prec s_3$ implies $s_1 \prec s_3$.

\[\square\]
**Definition.** Two strands $s_1$ and $s_2$ are *peers* with respect to a third strand $s$ if $s_1 \parallel s_2$, $s \parallel s_1$, and $s \parallel s_2$. Strands $s_1$ and $s_2$ are also called $s$–*peers*. (Note that $s_1$ and $s_2$ must both precede or both follow $s$.)

**Example.** The strands $b$ and $c$ are both $a$–peers and $d$–peers.
**Definition.** If a strand $s_1$ has no $s$–peers, then $s_1$ is *peerless* with respect to $s$, or $s$–peerless.

**Example.** The strand $d$ is $a$–peerless.
Definition. Two strands $s_1$ and $s_2$ are *mutually peerless* if $s_1$ is $s_2$–peerless and $s_2$ is $s_1$–peerless.

Example. The strands $a$ and $d$ are mutually peerless.

Lemma. If two strands $s_1$ and $s_2$ are mutually peerless, then $s \parallel s_1$ implies that $s \parallel s_2$. ■
• The Notion of Reducers
• Programming with Reducers
• A Real–World Example
• Graph Theory
• Semantics of Reducers
• Holders
When Views Are Hyperized Objects

**Property.** Let \( s \) be a strand that hyperizes an object \( x \), and let \( px \) be a hyperpointer to \( x \) in a strand \( s' \triangleright s \), where \( s' \) is \( s \)-peerless. Then \( \&*px = \&x \) in strand \( s' \).

Since strand \( s_4 \) is \( s_2 \)-peerless, we have \( \&*px = \&x \) in strand \( s_4 \).
When Views Are the Same

**Property.** Mutually peerless strands see the same views of every hyperobject.

\[
\begin{align*}
1 & \text{cilk_spawn foo();} \\
2 & \text{cilk::reducer_ptr<some_monoid> px(&x);} \\
3 & \text{// &px is not necessarily &x} \\
4 & \text{cilk_sync;} \\
5 & \text{assert(&*px == &x);} \\
\end{align*}
\]

Strands \(s_0\) and \(s_5\) see the same views, strands \(s_1\) and \(s_4\) see the same views, and strands \(s_2\) and \(s_3\) see the same views.
How Cilk Maintains Views

Upon a `cilk_spawn`:
- the child owns the view $h$ owned by the parent before the `cilk_spawn`;
- the parent owns a new view $h'$, initialized to the identity $e$.

After a spawned child returns:
- the parent owns the child’s view $h$, which is reduced with the parent’s view $h'$ sometime before the `cilk_sync`, and $h'$ is destroyed.

**Key optimization:** If $h' = e$, the implementation can avoid the reduce operation ⇒ in a serial execution, no new views need ever be created.
Lazy Implementation (Simplified)

- Each worker maintains a hypermap as a hash table, which maps hyperobjects into views.*
- An access to a reducer $x$ through a reducer pointer $px$ causes the worker to look up the local view of $x$ in the hypermap.
- If a view of $x$ does not exist in the hypermap, the worker creates a new view with value $e$.
- During load-balancing, when a worker “steals” a sub-computation, it creates an empty hypermap.
- When a worker finishes its subcomputation, hypermaps are combined using the appropriate $reduce()$ functions.
- The actual distributed protocol becomes rather tricky to avoid deadlock and ensure rapid completion — a SPAA 2009 paper provides details.

*In fact, each worker maintains 2 additional auxiliary hypermaps to assist in bookkeeping.
Overheads

- For programs with sufficient parallelism, the total cost of performing $O(1)$-time `reduce()` functions is provably small.

- The cost of an access to a reducer view is never worse than a hash-table look-up.

- If the reducer is accessed several times within a region of code, however, the compiler can optimize look-ups using common-subexpression elimination.

- In this common case, the hash-table look-up is performed only once, resulting in an access cost equal to one additional level of indirection (typically an L1-cache hit).
• The Notion of Reducers
• Programming with Reducers
• A Real–World Example
• Graph Theory
• Semantics of Reducers
• Holders
**Hold**ers provide “composable” thread-local storage.

**Example:** Pass a value from `proc1` to `proc4` without passing spurious parameters to `proc2` and `proc3`.

```c
T x;

void proc1() {
    for (i = 0; i < N; ++i) {
        x = f(i);
        proc2();
    }
}

void proc2() { proc3(); }
void proc3() { proc4(); }
void proc4() { use(x); }
```
**Holdes**

**Holdes** provide “composable” thread–local storage.

**Example:** Pass a value from proc1 to proc4 without passing spurious parameters to proc2 and proc3.

A holder is a reducer whose binary operator $\otimes$ simply returns one of its operands.

If two strands are mutually peerless, their views of the holder are identical.

The view returned by $*px$ in proc4() is the same as that in the corresponding iteration of the cilk_for loop, irrespective of whether bar() contains parallelism.
Suppose that an expensive-to-construct object $x$ is declared within a loop, but $x$ need not be freshly constructed each time through the loop.

```plaintext
for( int k=0; k<n; ++k ) {
    ExpensiveToConstructObject x;
    ...
    use(x)
    ...
}
```

The programmer can **hoist** the construction of $x$ out of the loop for efficiency in serial code:

```plaintext
ExpensiveToConstructObject x;
for( int k=0; k<n; ++k ) {
    ...
    use(x)
    ...
}
```

Parallelizable

NOT Parallelizable
Case: Scratch Object

Using a holder allows the common case to avoid the construction, while providing a freshly constructed object exactly when it is needed to avoid a race.

```cpp
holder_ptr<ExpensiveToConstructObject> p();
for( int k=0; k<n; ++k ) {
  ...
  use(&p)
  ...
}
```

Serial execution

```
k=0 ... k=1 ... k=2 ... k=3 ... k=4 ... k=5
```

© 2011 Charles E. Leiserson
Holder to the Rescue!

Using a holder allows the common case to avoid the construction, while providing a freshly constructed object exactly when it is needed to avoid a race.

```
holder_ptr<ExpensiveToConstructObject> p();
for( int k=0; k<n; ++k ) {
  ...
  use(&p)
  ...
}
```

Parallel execution

\[ k=0 \rightarrow k=1 \rightarrow k=2 \rightarrow k=3 \rightarrow k=4 \rightarrow k=5 \]

steal
Case: Loop–Carried Dependency

```c
x = init_val;
for( int k=0; k<n; ++k ) {
    foo(x);
    x = next(x);
}
```

- Suppose that it is safe to run multiple invocations of `foo(x)` in parallel. The only thing stopping us from executing iterations in parallel is the assignment `x = next(x)`, which is called a **loop–carried dependency**.

- Let us assume that the kth iteration of `x` can be computed directly in an efficient manner by a function `kth(init_val, k)`.
  - For example, if `next(x)` is “`x + c`”, the value of `x` for the kth iteration is `init_val + k*c`.  

© 2011 Charles E. Leiserson
Loop–Carried Dependency

The loop–carried dependency can then be removed and the loop parallelized as follows:

```c
x = init_val;
for( int k=0; k<n; ++k ) {
    foo(x);
    x = next(x);
}
```

Problem. The call to `kth()` may be expensive compared with `next()`, increasing the work overhead.
A holder that always returns its second operand gets the best of both worlds.

```cilk
// cilk::holder_ptr<T, cilk::holder_keep_last> px(&x);
*px = init_val;
for (int k=0; k<n; ++k) {
    if (*px == e) *px = kth(init_val, k);
    foo(*px);
    *px = next(*px);
}
```

We assume here that `next()` never returns the default constructed value `e` of a type-`T` object.

**Serial execution**

```
k=0  ⋯  k=1  ⋯  k=2  ⋯  k=3  ⋯  k=4  ⋯  k=5
```
A holder whose `reduce()` always returns its second operand gets the best of both worlds.

```cpp
cilk::holder_ptr<T, cilk::holder_keep_last> px(&x);
*px = init_val;
cilk_for( int k=0; k<n; ++k ) {
  if( *px == e ) *px = kth(init_val, k);
  foo(*px);
  *px = next(*px);
}
```

In the common case, the then-clause is never executed.

Parallel execution

```
k=0  ⋯  k=1  ⋯  k=2  ⋯  k=3  ⋯  k=4  ⋯  k=5
```

The second operand is the correct value after the join.