LECTURE 9
Reducer and Holder Hyperobjects

I–Ting Angelina Lee
October 04, 2012
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.”

2012 — 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 determinacy 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;
}
Summing Example in Cilk

```c++
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;
}
```

Determinacy Race!
Mutual-Exclusion Locking

```cpp
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;
}
```

Problems
Lock overhead & lock contention.
*We still have a determinacy race.*
int compute(const X& v);
int main() {
    const int n = 1000000;
    extern X myArray[n];
    // ...
    cilk::reducer< cilk::opadd<int> > result_r();
    cilk_for (int i = 0; i < n; ++i) {
        *result_r += compute(myArray[i]);
    }
    int x = 0;
    result_r.move_out(x);
    std::cout << "The result is: " << x << std::endl;
    return 0;
}
A **reducer** is designed to replace the use of a nonlocal variable in a parallel computation. The reducer is defined over an **associative** operation, such as addition, maximum, minimum, AND, OR, etc. Strands can update a reducer as if it were an ordinary nonlocal object with the designated operations, but the reducer is, in fact, maintained as a collection of **worker-local views**. The Cilk runtime system coordinates the views and combines them when appropriate. When only one view of the reducer remains, the value of the underlying view reflects all the prior updates to the reducer.

**Example:** summing reducer

\[ r_1 = 42 \]
\[ r_2 = 14 \]
\[ r_3 = 33 \]
\[ r = 89 \]
The Notion of Reducers
### Conceptual Behavior

<table>
<thead>
<tr>
<th>original</th>
<th>equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 1; )</td>
<td>( x1 = 1; )</td>
</tr>
<tr>
<td>( x += 3; )</td>
<td>( x1 += 3; )</td>
</tr>
<tr>
<td>( x++; )</td>
<td>( x1++; )</td>
</tr>
<tr>
<td>( x += 4; )</td>
<td>( x1 += 4; )</td>
</tr>
<tr>
<td>( x++; )</td>
<td>( x1++; )</td>
</tr>
<tr>
<td>( x += 5; )</td>
<td>( x1 += 5; )</td>
</tr>
<tr>
<td>( x += 4; )</td>
<td>( x2 = 0; )</td>
</tr>
<tr>
<td>( x += 5; )</td>
<td>( x2 += 9; )</td>
</tr>
<tr>
<td>( x += 2; )</td>
<td>( x2 += 2; )</td>
</tr>
<tr>
<td>( x += 6; )</td>
<td>( x2 += 6; )</td>
</tr>
<tr>
<td>( x += 5; )</td>
<td>( x2 += 5; )</td>
</tr>
<tr>
<td>( x = x1 + x2; )</td>
<td></td>
</tr>
</tbody>
</table>

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

Can execute in parallel with no races!
If you don’t “look” at the intermediate values, the result is **determinate**, because addition is **associative**.
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.
Programming with Reducers
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:

\[
\begin{align*}
\bullet \ (\mathbb{Z}, +, 0) \\
\bullet \ (\mathbb{R}, \times, 1) \\
\bullet \ (\{\text{TRUE, FALSE}\}, \wedge, \text{TRUE}) \\
\bullet \ (\Sigma^*, \|, \varepsilon) \\
\bullet \ (\mathbb{Z}, \text{MAX, } -\infty)
\end{align*}
\]
Representing Monoids

A Cilk programmer can represent a monoid on type–T objects by creating a C++ class that inherits from cilk::monoid_base<T> and defines

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

**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 \( r \) over \texttt{sum\_monoid} that operates on int–type views can now be defined in terms of \texttt{sum\_monoid\_int}:

\begin{verbatim}
cilk::reducer<sum_monoid_int> r;
\end{verbatim}

- Upon declaration, the default constructor of a reducer initializes \( r \)’s \textit{initial view} with \textit{identity} \( e \).
- In a parallel region, the underlying view can be accessed by “\textit{dereferencing}” the reducer, e.g., \( *r \ += \ 42 \). This operation actually accesses the \textit{local view} of the reducer for the strand that executes it.
- Final value resulted from the updates can be safely retrieved via \( r\text{.move\_out}(...) \) when \textit{no parallel views exist}.
- For a reducer with object–type view, one can use \( r\rightarrow\text{update\_func()} \) to invoke a member function of the underlying view.
Working with Views

Local views can be manipulated directly:

```cpp
cilk::reducer<sum_monoid_int> r;
cilk_for (int i = 0; i < n; ++i) {
    int pre = *r;
    *r = pre + i;  // Same thing as *r += i.
    int post = *r;
    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:

- `reducer< opadd<T> >`: sum elements of type T.*
- `reducer< list_append<T> >`: add to the end of a list whose elements are of type T.
- `reducer< list_prepend<T> >`: add to the beginning of a list whose elements are of type T.
- `reducer< opand<T> >`: bitwise AND of elements of type T.
- `reducer< opor<T> >`: bitwise OR of elements of type T.
- `reducer_max<T>`: maximum of elements of type T.
- `reducer_min<T>`: minimum of elements 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.
Semantics of Reducers (Graph Theory)
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 $s_2 \succ s_1$. 
**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 $s_2 \prec s_1$. 
**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.
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
- $s_1 \succ s_2$.  

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 $s_1 \prec s_3$.  

Definition. The peers of a strand s is the set of strands that logically parallel the strand s, denoted as peers(s) = \{s' \in \text{strands} : s' \parallel s\}.
**Definition.** The peers of a strand $s$ is the set of strands that logically parallel the strand $s$, denoted as $\text{peers}(s) = \{s' \in \text{strands} : s' \parallel s\}$.

**Example.** $\text{peers}(c) = \{a, d, e, f, g, h, l, j, k\}$
Definition. **Serial walk** of computation dag $G$ is the list of instructions encountered in a depth-first execution in which *spawn* edges are followed before *continue* edges.
Stable–View Theorem. Let $r$ be a reducer with an associative operator $\otimes$. Consider a serial walk of $G$, and let $a_1, a_2, \ldots, a_k$ be the update amount to $r$ after strand $x$ and before strand $y$. Denote the view for $r$ in $x$ by $r_x$ and the view in $y$ by $r_y$. If peers($x$) = peers($y$), then we have

$$r_y = r_x \otimes a_1 \otimes a_2 \otimes a_3 \otimes \ldots \otimes a_k.$$
**Stable–View Theorem.** Let $r$ be a reducer with an associative operator $\otimes$. Consider a serial walk of $G$, and let $a_1, a_2, \ldots, a_k$ be the update amount to $r$ after strand $x$ and before strand $y$. Denote the view for $r$ in $x$ by $r_x$ and the view in $y$ by $r_y$. If $\text{peers}(x) = \text{peers}(y)$, then we have

$$r_y = r_x \otimes a_1 \otimes a_2 \otimes a_3 \otimes \ldots \otimes a_k .$$

Stable–view theorem does not apply.
Stable-View Theorem. Let \( r \) be a reducer with an associative operator \( \otimes \). Consider a serial walk of \( G \), and let \( a_1, a_2, \ldots, a_k \) be the update amount to \( r \) after strand \( x \) and before strand \( y \). Denote the view for \( r \) in \( x \) by \( r_x \) and the view in \( y \) by \( r_y \). If \( \text{peers}(x) = \text{peers}(y) \), then we have

\[
r_y = r_x \otimes a_1 \otimes a_2 \otimes a_3 \otimes \ldots \otimes a_k.
\]

Definition. In this case, we say that \( r_y \) is stable with respect to \( r_x \).

Corollary. If \( r \) is only updated through its associative operator \( \otimes \), the result of \( a_1 \otimes \ldots \otimes a_k \) is deterministic.
When Views Are Stable

**Property.** If two strands $x \prec y$ have the same peer sets, then $r_y$ is *stable* with respect to $r_x$.

```c
#include <cilk/cilk.h>
cilk::reducer< opadd<int> > r;
static void foo() {
    cilk_spawn bar();
    cilk_spawn baz();
    // some serial code
    cilk_sync;
    // some other serial code
}
int main(...) {
    cilk_spawn foo();
    // call some other cilk function
    cilk_sync;
    return 0;
}
```
When Views Are Stable

Property. If two strands $x \lessdot y$ have the same peer sets, then $r_y$ is stable with respect to $r_x$.

cilk::reducer< opadd<int> > r;
static void foo() {
  cilk_spawn bar();
  ::
    cilk_spawn baz();
  // some serial code
  ::
    cilk_sync;
  // some other serial code
} int main(...) {
  cilk_spawn foo();
  // call some other cilk function
  cilk_sync;
  return 0;
}

Is $r_4$ stable with respect to $r_2$?

Don’t know. 2 || bar, but 4 does not.
When Views Are Stable

**Property.** If two strands $x \prec y$ have the same peer sets, then $r_y$ is **stable** with respect to $r_x$.

```cpp
cilk::reducer< opadd<int> > r;
static void foo() {
  cilk_spawn bar();
  ::
    cilk_spawn baz();
  // some serial code
  ::
    cilk_sync;
  // some other serial code
} int main(...) {
  cilk_spawn foo();
  // call some other cilk function
  cilk_sync;
  return 0;
}
```

Is $r_4$ stable with respect to $r_1$?

**Yes**
When Views Are Stable

Property. If two strands $x \prec y$ have the same peer sets, then $r_y$ is stable with respect to $r_x$.

cilk::reducer< opadd<int> > r;
static void foo() {
  cilk_spawn bar();
  cilk_spawn baz();
  // some serial code
  cilk_sync;
  // some other serial code
}
int main(...) {
  cilk_spawn foo();
  // call some other cilk function
  cilk_sync;
  return 0;
}

Is $r_5$ stable with respect to $r_2$?

Don’t know. 2 || bar, but 5 does not.
**Property.** If two strands $x \prec y$ have the same peer sets, then $r_y$ is **stable** with respect to $r_x$.

```cpp
cilk::reducer< opadd<int> > r;
static void foo() {
  cilk_spawn bar();
  ...
  cilk_spawn baz();
  // some serial code
  ...
  cilk_sync;
  // some other serial code
}
int main(...) {
  cilk_spawn foo();
  // call some other cilk function
  cilk_sync;
  return 0;
}
```

Is $r_5$ stable with respect to $r_1$? **Yes**
When *reducer is insufficient

Scenario I: The program uses a min reducer to keep track of the min value of some calculation, but the program cares only about the “local” min at each “round” of computation.

- The program does not care about keeping the history from previous rounds.
- Can’t know the local min unless we can reset the view before we start the next round of calculation.
- Ideally, we would like to reset the value of a global min reducer at strand 5, 9, 13, 17 ...
When *reducer is insufficient

**Scenario II:** Would like to repeatedly populate a data structure and then process elements stored in the data structure in parallel.

**Example:** The bag data structure used in parallel breath-first search (PBFS).
- A bag is an unordered set data structure that supports efficient parallel traversal of the set.
- PBFS alternates between two bags to process nodes in one bag and inserts newly found nodes into the second bag.
- Can’t traverse the bag in parallel without emptying out the reducer!
Move Semantics

The `cilk::reducer` library supports `move_in` and `move_out` operations:

- **reducer::move_in:**
  - a way of setting a value at certain point irrespective of what happened before that point.
  - a fast way of populating underlying view.

- **reducer::move_out:**
  - a fast way to empty out the reducer.

Both operations take a single argument: an object of the underlying view type to **swap content with**.

- The move is **destructive** — content of the source view is undefined after the swap.
  - Must do a `move_in` after a `move_out` in order to safely reuse a reducer after.

- When is it safe to use `move_in / move_out`? The **stable-view theorem** applies.
A Real-World Example
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;
    }
}
```
Naive Parallelization

Idea:
Parallelize the search by using a `cilk_for` to search all the children of each internal node in parallel.*

Oops!

```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;
    }
}
```

*Node::const_iterator must be a random-access iterator.
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;
    }
}
```

Determinacy Race!
Refactoring the Code

Idea
Define the \texttt{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 \texttt{walk()} routine must be changed.
The \texttt{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.lock();
        }
        break;
    case Node::INTERNAL:
        cilk_for (Node::const_iterator child = x.begin();
                  child != x.end();
                  ++child) {
            walk(child);
        }
        break;
    }
```
Reducer Solution

Create a list-append reducer, \texttt{list\_r}, whose \texttt{reduce()} function concatenates lists.* The resulted final view from \texttt{list\_r} contains target objects in the same order as in the original serial code.

*List concatenation is associative with the empty list as the identity.

```cpp
Node *target;
std::list<Node *> output_list;
cilk::reducer< cilk::list_append<Node*> > list_r;
...
void walk(Node *x) {
    switch (x->kind) {
    case Node::LEAF:
        if (target->collides_with(x)) {
            list_r->push_back(x);
        }
        break;
    case Node::INTERNAL:
        cilk_for (Node::const_iterator child = x.begin();
                   child != x.end();
                   ++child) {
            walk(child);
        }
        break;
    }
    // do list_r.move_out(output_list) at the end
```
Support for Reducers
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 view of a reducer $r$ through $\ast r$ or $r->update\_func()$ causes the worker to look up the local view of $r$ in the hypermap.
- If a view of $r$ 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 [2] 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 hoist the look-ups using common-subexpression elimination.
- In this common case, the hash-table look-up is performed only once, and subsequent accesses cost equal to one additional level of indirection (typically an L1-cache hit).
A Special Type of Reducers — Holders
Holders 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 (int i = 0; i < N; ++i) {
        x = f(i);
        proc2();
    }
}

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

Which operand to keep depends on the policy — by default, it’s indeterminate (the most efficient one).
Use Case: Scratch Object

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.

```java
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:

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

Parallelizable

**NOT** Parallelizable
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.

cilk::holder<ExpensiveToConstructObject> h;

for (int k=0; k<n; ++k) {
    // ...
    use(*h)
    // ...
}

**Serial execution**

k=0 ⋯ k=1 ⋯ k=2 ⋯ k=3 ⋯ k=4 ⋯ k=5
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.

cilk::holder<ExpensiveToConstructObject> h;

for (int k=0; k<n; ++k) {
    ...
    use(*h)
    ...
}

Parallel execution
k=0  ...  k=1  ...  k=2  ...  k=3  ...  k=4  ...  k=5

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Use Case: Loop–Carried Dependency

```
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`. 
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.
Keep–Last Reducer

A holder whose `reduce()` always returns its right operand gets the best of both worlds.

```cpp
cilk::holder<T, cilk::holder_keep_last> h;
h.move_in(init_val);
cilk_for (int k=0; k<n; ++k) {
  if(*h == e) { *h = kth(init_val, k); }
  foo(*h);
  *h = next(*h);
}
```

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
```
Keep-Last Reducer

A holder whose `reduce()` always returns its right operand gets the best of both worlds.

```cpp
cilk::holder<T, cilk::holder_keep_last> h;
h.move_in(init_val);
cilk_for (int k=0; k<n; ++k) {
    if(*h == e) { *h = kth(init_val, k); }
    foo(*h);
    *h = next(*h);
}
```

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

Parallel execution

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

The right operand is the correct value after the join.