Reducer Hyperobjects
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;
}
Mutual-Exclusion Locking

```
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.*
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.
Global and other nonlocal variables can inhibit parallelism by inducing race bugs.

1973 — Historical perspective

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

2015 — 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.
Reducer Solution

```cpp
int compute(const X& v);
int main() {
    const int n = 1000000;
    extern X myArray[n];
    // ...
    int res = 0;
    cilk::reducer< cilk::opadd<int> > result_r();
    p_result.move_in(res);
    cilk_for (int i = 0; i < n; ++i) {
        *result_r += compute(myArray[i]);
    }
    result_r.move_out(res);
    std::cout << "The result is: " << res << std::endl;
    return 0;
}
```

Declare `result_r` to be a summing reducer operating on int type.

Updates are resolved automatically without races or contention.

At the end, the underlying value of `result_r` reflects all the parallel updates.
Intuition for Reducers

- 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

*\( r \): 42

*\( r \): 14

*\( r \): 33

89
The Notion ofReducers
### Conceptual Behavior

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

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

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
Algebraic Monoids

**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:**

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

**Associative**

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c$$

**Identity**

$$a \otimes e = e \otimes a = a$$
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);
  }
};
```
Defining and Accessing Reducers

A reducer `r` over `sum_monoid` that operates on `int`-type views can now be defined in terms of `sum_monoid_int`:

```cpp
cilk::reducer<sum_monoid_int> r;
```

- Upon declaration, the default constructor of a reducer initializes `r`’s initial view with identity `e`.
- In a parallel region, the underlying view can be accessed by "dereferencing" the reducer, e.g., `*r += 42`. This operation actually accesses the local view of the reducer for the strand that executes it.
- Final value resulted from the updates can be safely retrieved via `r.move_out(...)` when no parallel views exist.
- For a reducer with object-type view, one can use `r->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.

*Behavior is nondeterministic when used with floating-point numbers.

But it’s not hard to “roll your own” using **cilk::monoid_base** and **cilk::reducer**.
Reducer Semantics
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 extract content out of 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`?
**Series Relations**

**Definition.** A strand \(s_1\) (logically) precedes another strand \(s_2\), denoted \(s_1 < s_2\), 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 \(s_2 > s_1\).
**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:** $c \parallel d$. 
Lemmas

**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 $\text{peers}(s) = \{s' \in \text{strands} : s' \parallel s\}$. 
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

**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

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

Other updates are not included.
**Stable-View Theorem**

**Stable-View Theorem.** Let $r$ be a reducer with an associative operator $\otimes$. Consider a serial walk of $G$, and let $a_1, a_2, …, a_k$ 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

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

**Stable-view theorem does not apply.**
Stable-View Theorem

**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**.

One should do `move_in` and `move_out` at strands that have the same peerset.
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();
  ...
  cilkSpawn baz();
  // some serial code
  ...
  cilkSync;
  // some other serial code
}

int main(...) {
  cilkSpawn foo();
  // call some other cilk function
  cilkSync;
  return 0;
}
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 $r_4$ stable with respect to $r_2$?  

*Don’t know.*  
$2 \parallel \text{bar, but } 4 \text{ 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 \( r_x \).

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 \( x < y \) 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();
  cilkSpawn baz();
  // some serial code
  cilk_sync;
  // some other serial code
} int main(...) {
  cilkSpawn foo();
  // call some other cilk function
  cilk_sync;
  return 0;
}
```

Is \( r_5 \) stable with respect to \( r_2 \)?

Don’t know. 2 \( \parallel \) bar, but 5 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 $r_5$ stable with respect to $r_1$? Yes
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 ...
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 breadth-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 extract content out of 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.
Runtime Support forReducers
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 no steals occur, $h'=h \Rightarrow$ in a serial execution, no new views need ever be created.
High-Level Ideas

• Recall: between successful steals, each worker's behavior mirrors the serial execution. 
  *trace:* the execution done by a worker between steals. 
  We just need to create a local view per trace, and accumulate the updates within that trace in the local view.

• A worker maintains a hashtable, called hashmap, per trace that maps from the reducer object to the appropriate local view for the trace.

• An access to a view of a reducer $r$ through $\star r$ 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.

• When a worker finishes its subcomputation, hypermaps are combined using the appropriate reduce() functions.
Recall Runtime DS: Frames

Invocation tree:

When a steal occurs, the frame on top of a victim's deque gets promoted into a full frame.

(Anything not in the deque is suspended.)
Recall Runtime DS: Frames

Invocation tree:

A
  B
    C
      D
        E
          W4
          F
            G
              H
                I
                  W1
                    W2
                      W3

: a full frame
(Anything not in the deque is suspended.)

Say W2 steals D and resumes it. Then, the hypermap in D is transferred to F, and W2 creates a new empty hypermap for the continuation of D.

When the continuation of D access a reducer, it triggers a new views to be created.
Recall Runtime DS: Frames

Invocation tree:

A
  B
   C
     D
      E
       W4
   F
   G
   H
   I
   : : :
   W1 W2 W3

A
  B
   D
      F
       W1
   G
   : : :
   W2

E
 : : :
W3

C
 : : :
W4

: a full frame
(Anything not in the deque is suspended.)

What happens when a spawned child, say E, returns to its stolen parent?
Combining of Hypermaps

Invocation tree:
Combining of Hypermaps

Invocation tree:

Steal tree: pointers to siblings and left-most child that are full frames.

Each full frame has two additional "place holder" for hypermaps.

3 hypermaps per frame: user, left-child (lchild), and right-sibling (rsib).

Whenever a full frame is returning, it first combines its user with lchild and rsib (if there is one), and deposit itself into the appropriate place holder.
Combining of Hypermaps

Invocation tree:

Steal tree: pointers to siblings and left-most child that are full frames.

Each full frame has two additional "place holder" for hypermaps.

3 hypermaps per frame: user, left-child (lchild), and right-sibling (rsib).

When E returns, W3 first combines the hypermaps, and then deposit the resulting hypermap into D's rsib. Then it can unlink E from the steal tree.
Recall Runtime DS: Frames

Invocation tree:

A ➔ B ➔ C
  ➔ D ➔ E ➔ W4
  ➔ F ➔ G ➔ H ➔ I
  ➔ W1 ➔ W2 ➔ W3

A ➔ B ➔ D ➔ F
  ➔ G
  ➔ E
  ➔ C

: a full frame

(Anything not in the deque is suspended.)
Recall Runtime DS: Frames

Invocation tree:

A
  ↗
  ↘
B   C
  ↗
D   E
  ↗
F   G
  ↗
W1  W2  W3

Whenever a steal occurs, the frame on top of a victim's deque gets promoted into a full frame.

The hashmap for a trace is stored with a full frame.