How to Write a Reducer

You can write a custom reducer if none of the supplied reducers satisfies your requirements.

Any of the supplied reducers can be used as models for developing new reducers, although some of these examples are relatively complex. The implementations are found in the `reducer_*.h` header files in the `include/cilk` directory of the installation. Additional examples are provided in the following sections.

The basic reducer infrastructure is in `include/cilk/reducer.h`.

Monoids

In mathematics, a *monoid* comprises a set of values *T*, an associative operation *OP* on those values, and an identity *I* value with respect to the operation. That is (*T, OP, I)* is a monoid if:

- *OP* is associative: for every *x*, *y*, and *z* in *T*, *x OP (y OP z) = (x OP y) OP z*
- *I* is the identity for *OP*: for every x in *T*, x *OP I = I op x = x*

Examples of monoids:

- Values = the set of all integers; Operation = addition; Identity = 0.
- Values = the set of all 32-bit unsigned integers; Operation = addition modulo 2^{32}; Identity = 0.
- Values = the set of all strings; Operation = concatenation; Identity = the empty string.

A reducer is always built around a monoid. That is, it computes a final value by combining a set of values with an operator; it relies on the associativity of the operator to be able to recombine parts of the combination in parallel; and it relies on having an identity value to be able to initialize the accumulator variables for the partial computations.

Components of a Reducer

A reducer can be broken into these logical parts:

- The "value type" is the type of the value computed by the reducer.
- The "view" class represents the accumulator variable within the reducer. In many cases, the view class is the same as the value type. (For example, the view class of a `reducer_opadd` that computes the sum of a set of values is just the type of those values.) However, the view class may also contain additional

information to represent aspects of the computation that are not part of the final computed value. (For example, the view class of a `reducer_string` that computes the concatenation of a set of strings contains a list of substrings that haven't yet been combined.)

The reducer directly contains one instance of the view class (the "leftmost view"); other view instances are created and destroyed as strands start and finish executing in parallel.

- The "monoid" class represents the mathematical monoid that the reducer is built around. It defines the value type and provides the `identity()` and `reduce()` functions used by the reducer when creating and merging views, as well as some functions to handle memory allocation of views. See the discussion below.
- The "reducer" class is the `cilk::reducer` class template, instantiated with the monoid (`cilk::reducer<Monoid>`). A reducer object contains a monoid object and the leftmost view, manages the interaction with the Cilk runtime system, and provides access to the view instance for the current strand during execution.
- The "wrapper" class restricts access to a reducer's views so that only operations that are consistent with the monoid's associative operation are permitted. Wrapper classes typically provide `set_value()` and `get_value()` functions for accessing the computed value.

  Wrapper classes are optional — `cilk::reducer<Monoid>` is a complete reducer which does not restrict access to its views. Wrappers can provide robustness for general-purpose or library reducers, but adding a wrapper may be a waste of effort for a single-purpose / single use reducer or reducers for which the view type already supplies a sufficiently-restrictive interface.

The Monoid Class

A monoid class provides the information needed by the reducer class to create, initialize, merge, and destroy view instances. It must define a typedef, `value_type`, which is the view class for the reducer (or the value type, if there isn't a separate view class), and the following functions, which maybe either `static` or `const`.

`reduce(value_type *left, value_type *right)`	evaluates `*left = *left OP *right`; leaves `*right` in an unspecified, but valid, state
`identity(value_type *p)`	constructs the identity value into the uninitialized

	memory pointed to by `p`
`destroy(value_type *p)`	calls the `value_type` destructor on the object pointed to by `p`
`allocate(size)`	returns a pointer to `size` bytes of raw memory
`deallocate(p)`	deallocates the raw memory pointed to by `p`

Any class which provides these definitions can be used as a monoid class, but in practice, a monoid class is always defined as a subclass of `cilk::monoid_base<T>`, which defines `value_type` to be `T` and defines `allocate()`, `destroy()`, and `deallocate()` functions using operator `new`. A class derived from `monoid_base` needs to declare and implement only the `identity()` and `reduce()` functions.

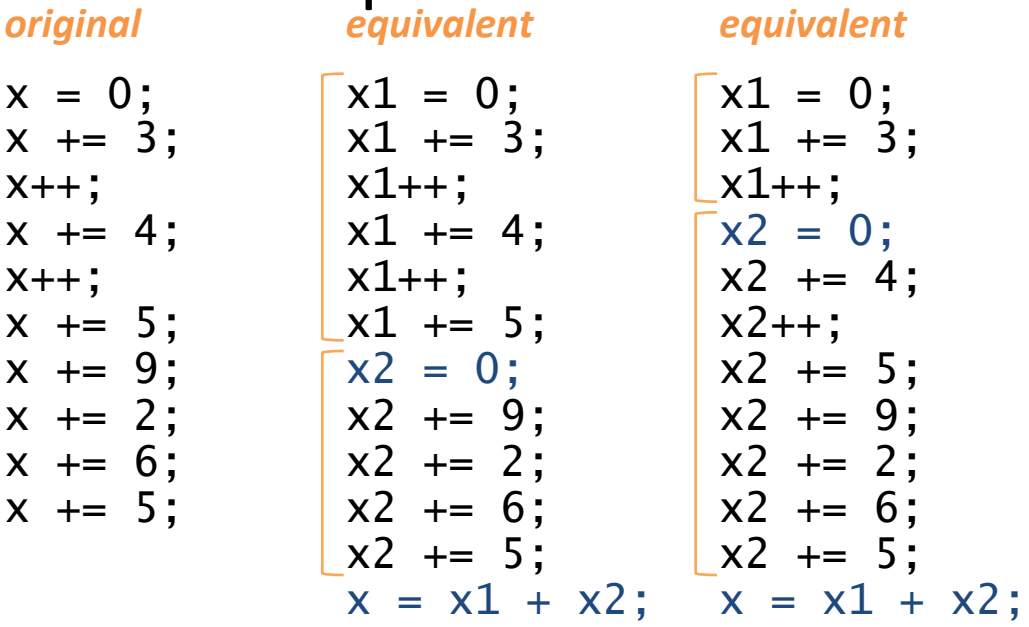
Note that the `reduce()` function is not required to leave any particular value in its "right" argument view, but it must leave it in a valid state so that it can be destructed following the `reduce()`.

Why Associativity Is Necessary

For a reducer to deterministically reproduce the serial semantics, the reduce function and the permitted operations on the view must implement an associative operation. (It does not need to be commutative.)

The serial execution of reduction computes the expression (a_0 *OP* a_1 *OP* a_2 *OP* … *OP* a_N). When the reduction is performed in parallel, that expression is broken up into a set of subexpressions which are then combined — for example, ((a_0 *OP* a_1 *OP* a_2) *OP* ((a_3 *OP* a_4) *OP* (a_5))) *OP* (a_6 *OP* a_7 *OP* … *OP* a_{10}) *OP* …. The partitioning into subexpressions, and the order in which the subexpressions are combined, are unpredictable, and can vary from one run of the program to another. If *OP* is associative, though, it doesn't matter: no matter how the expression is broken up, and no matter what order the subexpressions are associated in, the result will be the same.

Conceptual Behavior



If you don't "look" at the intermediate values, the result is ***determinate***, because addition is ***associative***.

Writing Reducers - A Singly-Linked List

Here is a simple singly-linked list that we created in a loop. We've tried to make the loop parallel by using a `cilk_for`, but it isn't going to work, because there will be data races between the list node updates. If we want to create a value incrementally in parallel, we need a reducer!

```
#include <cilk/cilk.h>
#include <iostream>

//  Singly-linked list.
//
struct IntListNode {
    int data;
    IntListNode* link;
};

// Compute a value. (Probably does something more interesting in
// a real program.)
//
```

```
int compute(int i)
{
    return i;
}

// Create a list.
//
IntListNode* make_list(int n)
{
    IntListNode* list = 0;
    for (int i = 0; i < n; ++i) {
        IntListNode* node = new IntListNode;
        node->data = compute(i);
        node->link = list;
        list = node;
    }
    return list;
}

// Use a list. (Probably does something more interesting in a
// real program.)
//
void print_list(IntListNode* list)
{
    for (IntListNode* node = list; node; node = node->link)
        std::cout << node->data << "\n";
}

int main()
{
    IntListNode* list = make_list(20);
    print_list(list);
}
```

The first step in creating a reducer is to identify the monoid. The data type here is a singly-linked list of integers, represented by `IntListNode` objects. Since parallelizing the computation will result in creating and combining multiple sublists, the associative operation must be list concatenation. The identity value for list concatenation is the empty list. Adding an object at the beginning of a list is equivalent to concatenating a new singleton list containing the object to be added at the front of the list (`list = {x} || list`), so the reduce operation must be "`left = right || left`".

How is the list to be represented? Is our view type different from our value type? A singly-linked list can be uniquely represented by a pointer to its first node (or a null pointer for an empty list), and this is what is returned from the `make_list` function and passed to the `print_list` function. However, there is no efficient way to concatenate two lists

represented just by pointers to their heads, so our view type needs both a head and a tail pointer.

So now we have the structure of our monoid. The value type is a pointer to the first node in the list, or `null` for an empty list. The view is a (head, tail) pointer pair, and provides an operation to add a new value to the list. The monoid's identity function initializes an empty-list view, and its reduce function concatenates the lists represented by two views.

This is a single-use reducer, so we won't bother with a wrapper. Here is the resulting parallel code using our new reducer:

```
#include <cilk/cilk.h>
#include <cilk/reducer.h>
#include <iostream>

//  Singly-linked list.
//
struct IntListNode {
    int data;
    IntListNode* link;
};


class IntListMonoid;            // forward declaration

//  View class.
//
class IntListView {
    friend class IntListMonoid; // for the identity and reduce functions
    IntListNode* head;
    IntListNode* tail;
public:
    void add_value(int x) {
        IntListNode* node = new IntListNode;
        node->data = x;
        node->link = head;
        head = node;
        if (!tail) tail = node;
    }
    IntListNode* get_value() const { return head; }
};

// Monoid class.
//
struct IntListMonoid : public cilk::monoid_base<IntListView> {

    // The monoid value_type is the view type.
    typedef IntListView value_type;
```

```
    // Set *view to the empty list.
    //
    static void identity(IntListView* view) {
        view->head = view->tail = 0;
    }

    // Move the right list to the beginning of the left list.
    // Leave the right list empty.
    //
    static void reduce(IntListView* left, IntListView* right) {
        if (right->head) {
            right->tail->link = left->head;
            left->head = right->head;
            right->head = right->tail = 0;
        }
    }
};


// Compute a value. (Probably does something more interesting in
// a real program.)
//
int compute(int i)
{
    return i;
}

// Create a list.
//
IntListNode* make_list(int n)
{
    cilk::reducer<IntListMonoid> list;
    cilk_for (int i = 0; i < n; ++i) {
        list->add_value(i);     // "list->" accesses the view
    }
    return list->get_value();   // "list->" accesses the view
}

// Use a list. (Probably does something more interesting in a
// real program.)
//
void print_list(IntListNode* list)
{
    for (IntListNode* node = list; node; node = node->link)
        std::cout << node->data << "\n";
}

int main()
{
    IntListNode* list = make_list(20);
    print_list(list);
}
```