Numerical algorithms

Matt Warner

Contained the the header file: `numeric`, are a couple of benefical functions. Here are the four major ones:

1 std::accumlate

Computes the sum (by default) of the given value `init` and the elements in the range [first, last), and returns the computed sum. `std::accumulate` has one definition that simply takes iterators `first` and `last`, and `init`, which is used to initialize the accumulator `acc`. The other definition of `std::accumulate` takes the same parameters as the previous definition, but adds an additional fourth parameter, `op`. Which is a *binary function object* that is applied. This binary function object can be used to perform other operators on the container like multiply, etc.

The below snippet are the two declarations.

```
template< typename InputIt, typename T >
T accumulate( InputIt first, InputIt last, T init );   (1)

template< typename InputIt, typename T, typename BinaryOp >
T accumulate( InputIt first, InputIt last, T init, BinaryOp op );  (2)
```

Note that if any of the following conditions is satisfied, the behavior is undefined.

- `T` is not *CopyConstructable*
- `T` is not *CopyAssignable*
- `op` modifies any elements of [first, last)
- `op` invalidates any iterator or subrange in [first, last).

Also note that the signature of `op` should be equivalent to the following (doesn't strictly need to be const &).

```
Ret fun(const Type1&, const Type2 &b);
```

Further note that the type `Type1` must be such that an object of type `T` can be implicitly converted to `Type1`. The type `Type2` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type2`. Lastly, the type `Ret` must be such that an object of type `T` can be assigned a value of type `Ret`.

The behavior of this function template is equivalent to:

```
template <typename InputIterator, typename T>
T accumulate (InputIterator first, InputIterator last, T init) {
    while (first != last) {
        init = init + *first;
        ++first;
    }
    return init;
}
```

Or if you provide a function object:

```
template <typename InputIterator, typename T>
T accumulate (InputIterator first, InputIterator last, T init) {
    while (first != last) {
        init = binary_op(init, *first);
        ++first;
    }
    return init;
}
```

Also note that the header file `functional` provides a couple of function objects that are handy for something like `std::accumulate`. These include:

```
std::multiplies<T>();
std::minus<T>();
std::plus<T>();
std::divides<T>();
std::modulus<T>();
```

Heres an example program that uses `std::accumulate`

```
#include <functional>
#include <iostream>
#include <numeric>
#include <string>
#include <vector>

int main() {
    std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
    int sum = std::accumulate(v.begin(), v.end(), 0);
    int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());

    auto dash_fold = [](std::string a, int b) {
        return std::move(a) + " - " + std::to_string(b);
    }
    std::string s = std::accumulate(std::next(v.begin()), v.end(), )
}
```

2 reduce (c++17)

The `reduce()` method in C++ is used for applying an algorithm to a range of elements in an array. By default, it returns the sum of values of elements in the applied range. It behaves similarly to `std::accumulate in STL`.

There are a variety of overloads for the reduce function. We have:

```
template<typename Inputit>
typename std::iterator_traits<InputIt>::value_type
reduce(InputIt first, InputIt last);

template<typename ExecutionPolicy, typename ForwardIt>
typename std::iterator_traits<ForwardIt>::value_type
reduce(ExecutionPolicy&& policy, ForwardIt first, FowardIt last);

template<typename InputIt, typename T>
T reduce(InputIt first, InputIt last, T init);

template<typename ExecutionPolicy, typename ForwardIt, typename T>
T reduce(Execution Policy, ForwardIt first, Forward It last, T init);

template< typename InputIt, typename T, typename BinaryOp >
T reduce( InputIt first, InputIt last, T init, BinaryOp op );

template<typename ExecutionPolicy, typename T, typename BinaryOp>
typename std::iterator_traits<ForwardIt>::value_type
T reduce(ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, T init, BinaryOp op);
```

Parameters

first, last - the range of elements to apply the algorithm to

init - the initial value of the generalized sum

policy - the execution policy to use.

op - binary *function object* that will be applied in unspecified order to the result of dereferencing the input iterators, the result of other `op` and `init`

Return value

Returns the value of the result of the reduction operation.

Execution Policies

Execution policies are a C++17 feature which allows programmers to ask for algorithms to be parallelised. These are three execution policies in C++17:

- **`std::execution::seq`** - do not parallelise
- **`std::execution::par`** - parallelise
- **`std::exection::par_unseq`** - parallelise and vectorise (requires that the operation can be interleaved, so no acquiring mutexes and such)

Example:

```

int main() {
    std::vector<int> v = {1,2,3,4,5};

    // reduce returns sum of the elements in the given range
    int sum = reduce(v.begin(), v.end(), 0);
    std::cout << "Default execution of the reduce function: " <<  sum << std::endl;

    // here it returns the sum without needing to pass the initial value
    sum = reduce(v.begin(), v.end());
    cout << "Execution with default initial value: " << sum << std::endl;
}
```

The difference between **`std::accumulate`** and **`std::reduce`** is the execution policy. Lets say we use **`std::minus`** instead of **`std::plus`** as our reduction operation.

`./figures/fig3.png`

`std::accumulate`

`./figures/fig4.png`

`std::reduce`

With **`std::reduce`** we got the wrong answer because of the mathematical properties of subtraction. You can't abitrarily reorder the operands, or compute the operations out of order when doing subtraction. This is formalised in the properties of *commutativity* and *associativity*.

With **`std::reduce`** the elements of the range may be grouped and rearranged in arbitrary order, which might break your code if the binary operation is not commutative and associative.

2.1 inner_product

```
template< typename InputIt1, typename InputIt2, typename T >
T inner_product( InputIt1 first1, InputIt1 last1,                    (1)
                 InputIt2 first2, T init );

template< typename InputIt1, typename InputIt2, typename T,
          typename BinaryOp1, typename BinaryOp2 >
T inner_product( InputIt1 first1, InputIt1 last1,                    (2)
                 InputIt2 first2, T init,
                 BinaryOp1 op1, BinaryOp2 op2 );
```

Computes inner product (i.e sum of products) or performs ordered map/reduce operation on the range [first1, last1] and the range of `std::distance(first1,last1)` elements beginning at first2.

1. Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with the expression `acc = std::move(acc) + (*i1) * (*i2)` for every iterator `i1` in the range [first1, last1] in order and its correspoinding iterator `i2` in the range beginning at `first2`. For built-in meaning of + and *, this computes inner product of the two ranges.

2. Initializes the accumulator `acc` (of type T) with the initial value `init` and then modifies it with the expression `acc = op1(std::move(acc), op2(*i1,*i2))` for every iterator `i1` in the range [first1, last1] in order and its corresponding iterator `i2` in the range beginning at `first2`.

The implementation might look something like this for the overload that accepts **no** binary function objects.

```
template<typename InputIt1, typename InputIt2, typename T>
constexpr
T inner_product(InputIt1 first1, InputIt1 last1, InputIt2 first2, T init)
{
    while (first1 != last1)
    {
        init = std::move(init) + (*first1) * (*first2);
        ++first1;
        ++first2;
    }
    return init;
}
```

For the version of `std::inner_product` that takes two binary function objects. The implementation might look like this.

```
template<typename InputIt1, typename InputIt2, typename T
         typename BinaryOp1, typename BinaryOp2>
constexpr
T inner_product(InputIt1 first1, InputIt1 last1, InputIt2 first2, T init, BinaryOp1 op1,
↪  BinaryOp2, op2)
{
    while (first1 != last1)
    {
        init = op1(std::move(init), op2(*first1,*first2));
        ++first1;
        ++first2;
    }
    return init;
}
```

Heres an example of `std::inner_product in practice`

```
int main()
{
    std::vector<int> a{0,1,2,3,4};
    std::vector<int> b{5,4,3,2,1};

    int r1 = std::inner_product(a.begin(), a.end() b.begin(), 0);
    std::cout << "Inner product of a and b: " << r1 << "\n";

    int r2 = std::inner_product(a.begin(), a.end() b.begin(), 0,
                                std::plus<>(), std::equal_to<>());
    std::cout << "Number of pairwise matches between a and b" << r2 << "\n";
}
```

Output:

```
Inner product of a and b: 21
Number of pairwise matches between a and b: 2
```

Interestingly enough, you dont actually need 2 containers for `inner_product`, if you pass the `first1` again as `first2`, you can apply a transformation on the same element twice. For example, say you wanted to calculate the sum of squares in a container, if your transfor

3 transform_reduce

Just like how `std::reduce` is the parallelized version of `std::accumulute`, `transform_reduce` is the parallelized version of `std::inner_product`. This function has a wide range of parameter options.

```
// Default transform and reduce operations (multiple and add)
template <typename InputIt1, typename InputIt2, typename T>
T transform_reduce (InputIt1 first1, InputIt1 last1, InputIt2 first2, T init);


// Takes two binary function objects.
template< typename InputIt1, typename InputIt2, typename T,
typename BinaryOp1, typename BinaryOp2 >
T transform_reduce( InputIt1 first1, InputIt1 last1,
                    InputIt2 first2, T init,
                    BinaryOp1 reduce, BinaryOp2 transform );

// Takes an Execution policy
template <typename ExecutionPolicy, typename ForwardIt1, typename ForwardIt2, typename T>
T transform_reduce(ExecutionPolicy&& policy,
                   FowardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, T init);

// Takes an Execution policy and two binary function objects.
template< typename ExecutionPolicy,
          typename ForwardIt1, typename ForwardIt2, typename T,
          typename BinaryOp1, typename BinaryOp2 >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt1 first1, ForwardIt1 last1,
                    ForwardIt2 first2, T init,
                    BinaryOp1 reduce, BinaryOp2 transform )

// Only requires one container
template< typename InputIt, typename T,
```

```cpp
            typename BinaryOp, typename UnaryOp >
T transform_reduce( InputIt first, InputIt last, T init,
                    BinaryOp reduce, UnaryOp transform );

// One container, takes execution policy
template< typename ExecutionPolicy,
          typename ForwardIt, typename T,
          typename BinaryOp, typename UnaryOp >
T transform_reduce( ExecutionPolicy&& policy,
                    ForwardIt first, ForwardIt last, T init,
                    BinaryOp reduce, UnaryOp transform );
```

Much like `std::inner_product`, this algorithm will apply a transform operation on `first1`, `*first2`, and reduces the results (possibly permuted and aggregated in unspecified manner).

Note:-

The result is non-deterministic if the reduce is not associative or not commutative (like with `std::reduce`)

4 partial_sum

5 adjacent_difference