For_Each and Reduce Algorithms

This lesson goes into the details of reduce algorithm and its working.

WE'LL COVER THE FOLLOWING

- for_each Algorithm
- Understanding Reduce Algorithms
  - Parallel Version - `std::reduce`

for_each Algorithm

In the serial version of `for_each`, the version that was available before C++17 you get a unary function as a return value from the algorithm.

Returning such an object is not possible in a parallel version, as the order of invocations is indeterminate.

Here's a basic example:

```
void ForEachTest() {
	std::vector<int> v(100);
	std::iota(v.begin(), v.end(), 0);

	std::for_each(std::execution::par, v.begin(), v.end(),
		[](int& i) { i += 10; });

	std::for_each_n(std::execution::par, v.begin(), v.size() / 2,
		[](int& i) { i += 10; });
}


int main() {
	ForEachTest();
	return 0;
}
```

The first `for_each` algorithm will update all of the elements of a vector, while the second execution will work only on the first half of the container.

Understanding Reduce Algorithms

Another core algorithm that is available with C++17 is `std::reduce`. This new algorithm provides a parallel version of `std::accumulate`. But it's important to understand the difference.

`std::accumulate` returns the sum of all the elements in a range (or a result of a binary operation that can be different than just a sum).

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

int main(){

  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  auto sum = std::accumulate(v.begin(), v.end(), /*init*/0);
  // sum is 55

  std::cout << sum;

}
```

The algorithm is sequential and performs "left fold", which means it will accumulate elements from the start to the end of a container.

The above example can be expanded into the following code:

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

int main(){

  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  auto sum = 0 +
      v[0] + v[1] + v[2] +
      v[3] + v[4] + v[5] +
      v[6] + v[7] + v[8] + v[9];
  // sum is 55

  std::cout << sum;

}
```

```
}
```

Parallel Version - `std::reduce`

The parallel version - `std::reduce` - computes the final sum using a tree approach (sum sub-ranges, then merge the results, divide and conquer). This method can invoke the binary operation/sum in a nondeterministic order. Thus if `binary_op` is not associative or not commutative, the behavior is also non-deterministic.

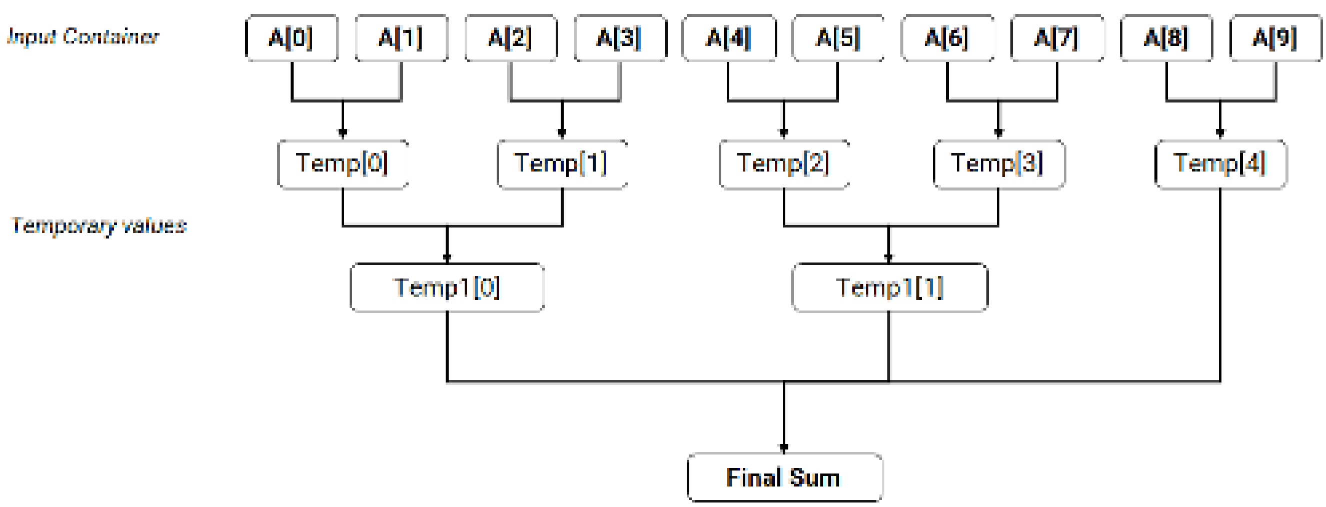
The parallel version code is as follows:

```
void ReduceTest() {
        std::vector<int> v(100);
        std::iota(v.begin(), v.end(), 0);

        auto sum = std::reduce(std::execution::par,
                                            v.begin(), v.end(),
                                            /*init*/0);

        std::cout << sum << '\n';
}
int main() {
  ReduceTest();
  return 0;
}
```

Here’s a simplified picture that illustrates how a sum of 10 elements might work in a parallel way:



The above example with `accumulate` can be rewritten into `reduce`:

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto sum = std::reduce(std::execution::par, v.begin(), v.end(), 0);
```

By default `std::plus<>{}` is used to compute the reduction steps.

A little explanation about associative and commutative operations:

> A binary operation `@` on a set `S` is **associative** if the following equation holds for all `x`, `y`, and `z` in `S`:
>
> `(x @ y) @ z = x @ (y @ z)`
>
> An operation is **commutative** if:
>
> `x @ y = y @ x`

For example, we'll get the same results for accumulate and reduce for a vector of integers (when doing a sum), but we might get a slight difference for a vector of floats or doubles. That's because floating point sum operation is not associative.

An example:

```
#include <iostream>
#include <limits> //for numeric_limits
using namespace std;

int main() {
  std::cout.precision(std::numeric_limits<double>::max_digits10);
  std::cout << (0.1+0.2)+0.3 << " != " << 0.1+(0.2+0.3) << '\n';
}
```

Another example might be the operation type: `plus`, for integer numbers, is associative and commutative, but `minus` is not associative nor commutative:

```
1+(2+3) == (1+2)+3 // sum is associative
1+8     == 8+1     // sum is commutative

1-(5-4) != (1-5)-4 // subtraction is not associative
1-7     != 7-1     // subtraction is not commutative
```

The next lesson will discuss another extension of the `reduce` method, along with scan algorithm. Read on to find out more!