```
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct Sum {
    float value;
    Sum() : value(0) {}
    Sum( Sum& s, split ) {value = 0;}
    void operator()( const blocked_range<float*>& range ) {
        float temp = value;
        for( float* a=range.begin(); a!=range.end(); ++a ) {
            temp += *a;
        }
        value = temp;
    }
    void join( Sum& rhs ) {value += rhs.value;}
};

float ParallelSum( float array[], size_t n ) {
    Sum total;
    parallel_reduce( blocked_range<float*>( array, array+n, 1000 ),
                     total );
    return total.value;
}
```

The example generalizes to reduction for any associative operation *op* as follows:

- Replace occurrences of 0 with the identity element for *op*
- Replace occurrences of `+=` with *op*= or its logical equivalent.
- Change the name `Sum` to something more appropriate for *op*.

The operation may be noncommutative. For example, *op* could be matrix multiplication.

The block size of 1000 can be omitted if you use an `auto_partitioner` or `affinity_paritioner`. With an `auto_partitioner`, the invocation of `parallel_reduce` can be changed as shown below:

```
    parallel_reduce( blocked_range<float*>( array, array+n, 1000 ),
                     total, auto_partitioner() );
```

3.7 parallel_scan<Range,Body> Template Function

Summary

Template function that computes parallel prefix.

Syntax

```
template<typename Range, typename Body>
void parallel_scan( const Range& range, Body& body );
```

Header

```
#include "tbb/parallel_scan.h"
```

Description

A `parallel_scan(`*range*`,`*body*`)` computes a parallel prefix, also known as parallel scan. This computation is an advanced concept in parallel computing that is sometimes useful in scenarios that appear to have inherently serial dependences.

A mathematical definition of the parallel prefix is as follows. Let $\oplus$ be an associative operation $\oplus$ with left-identity element $id_{\oplus}$. The parallel prefix of $\oplus$ over a sequence x_0, x_1, ...x_{n-1} is a sequence y_0, y_1, y_2, ...y_{n-1} where:

- $y_0 = id_{\oplus} \oplus x_0$
- $y_i = y_{i-1} \oplus x_i$

For example, if $\oplus$ is addition, the parallel prefix corresponds a running sum. A serial implementation of parallel prefix is:

```
T temp = id⊕;
for( int i=1; i<=n; ++i ) {
    temp = temp ⊕ x[i];
    y[i] = temp;
}
```

Parallel prefix performs this in parallel by reassociating the application of $\oplus$ and using two passes. It may invoke $\oplus$ up to twice as many times as the serial prefix algorithm. Given the right grain size and sufficient hardware threads, it can out perform the serial prefix because even though it does more work, it can distribute the work across more than one hardware thread.

TIP: Because `parallel_scan` needs two passes, systems with only two hardware threads tend to exhibit small speedup. `parallel_scan` is best considered a glimpse of a technique for future systems with more than two cores. It is nonetheless of interest because it shows how a problem that appears inherently sequential can be parallelized.

The template `parallel_scan<Range,Body>` implements parallel prefix generically. It requires the signatures described in Table 9.

Table 9: parallel_scan Requirements

Pseudo-Signature	Semantics
`void Body::operator()( const Range& r, pre_scan_tag )`	Accumulate summary for range `r`.
`void Body::operator()( const Range& r, final_scan_tag )`	Compute scan result and summary for range `r`.

`Body::Body( Body& b, split )`	Split `b` so that `this` and `b` can accumulate summaries separately. Body `*this` is object `a` in the table row below.
`void Body::reverse_join( Body& a )`	Merge summary accumulated by `a` into summary accumulated by `this`, where `this` was created earlier from `a` by `a`'s splitting constructor. Body `*this` is object `b` in the table row above.
`void Body::assign( Body& b )`	Assign summary of `b` to `this`.

A summary contains enough information such that for two consecutive subranges *r* and *s*:

- If *r* has no preceding subrange, the scan result for *s* can be computed from knowing s and the summary for *r*.
- A summary of r concatenated with s can be computed from the summaries of r and s.

For example, if computing a running sum of an array, the summary for a range *r* is the sum of the array elements corresponding to *r*.

Figure 2 shows one way that `parallel_scan` might compute the running sum of an array containing the integers 1-16. Time flows downwards in the diagram. Each color denotes a separate Body object. Summaries are shown in brackets.

1. The first two steps split the original blue body into the pink and yellow bodies. Each body operates on a quarter of the input array in parallel. The last quarter is processed later in step 5.
2. The blue body computes the final scan and summary for 1-4. The pink and yellow bodies compute their summaries by prescanning 5-8 and 9-12 respectively.
3. The pink body computes its summary for 1-8 by performing a reverse_join with the blue body.
4. The yellow body computes its summary for 1-12 by performing a reverse_join with the pink body.
5. The blue, pink, and yellow bodies compute final scans and summaries for portions of the array.
6. The yellow summary is assigned to the blue body. The pink and yellow bodies are destroyed.

Note that two quarters of the array were not prescanned. The `parallel_scan` template makes an effort to avoid prescanning where possible, to improve performance when there are only a few or no extra worker threads. If no other workers are available, parallel_scan processes the subranges without any pre_scans, by processing the subranges from left to right using final scans. That's why final scans must compute a summary as well as the final scan result. The summary might be needed to process the next subrange if no worker thread has prescanned it yet.

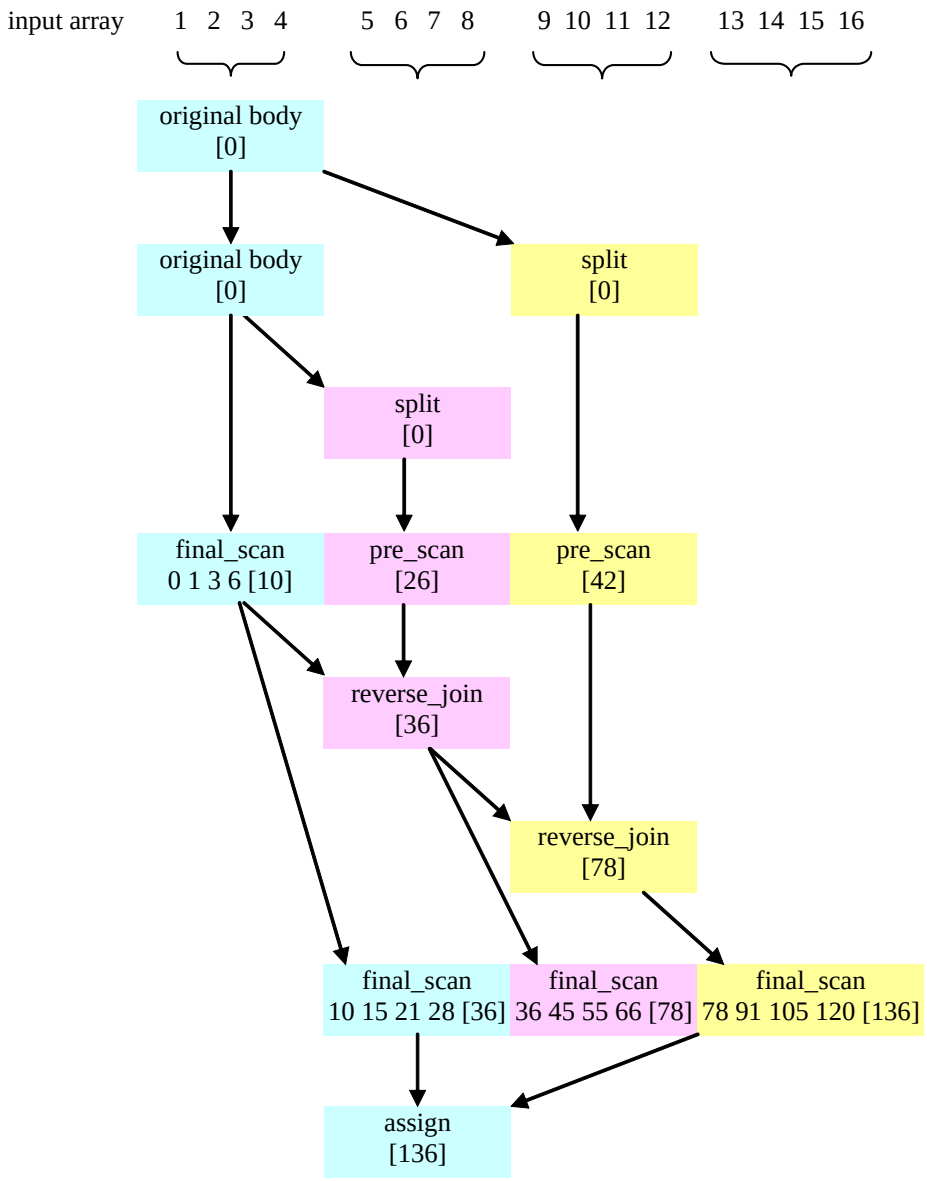

Figure 2: Example execution of parallel_scan

The following code demonstrates how the signatures could be implemented to use `parallel_scan` to compute the same result as the earlier sequential example involving ⊕.

```
using namespace tbb;

class Body {
    T sum;
    T* const y;
    const T* const x;
```

```
public:
    Body( T y_[], const T x_[] ) : sum(id⊕), x(x_), y(y_) {}
    T get_sum() const {return sum;}

    template<typename Tag>
    void operator()( const blocked_range<int>& r, Tag ) {
        T temp = sum;
        for( int i=r.begin(); i<r.end(); ++i ) {
            temp = temp ⊕ x[i];
            if( Tag::is_final_scan() )
                y[i] = temp;
        }
        sum = temp;
    }
    Body( Body& b, split ) : x(b.x), y(b.y), sum(id⊕) {}
    void reverse_join( Body& a ) { sum = a.sum ⊕ sum;}
    void assign( Body& b ) {sum = b.sum;}
};

float DoParallelScan( T y[], const T x[], int n) {
    Body body(y,x);
    parallel_scan( blocked_range<int>(0,n,1000), body );
    return body.get_sum();
}
```

The definition of `operator()` demonstrates typical patterns when using `parallel_scan`.

- A single template defines both versions. Doing so is not required, but usually saves coding effort, because the two versions are usually similar. The library defines static method `is_final_scan()` to enable differentiation between the versions.
- The prescan variant computes the ⊕ reduction, but does not update `y`. The prescan is used by `parallel_scan` to generate look-ahead partial reductions.
- The final scan variant computes the ⊕ reduction and updates y.

The operation `reverse_join` is similar to the operation `join` used by `parallel_reduce`, except that the arguments are reversed. That is, `this` is the *right* argument of ⊕. Template function `parallel_scan` decides if and when to generate parallel work. It is thus crucial that ⊕ is associative and that the methods of `Body` faithfully represent it. Operations such as floating-point addition that are somewhat associative can be used, with the understanding that the results may be rounded differently depending upon the association used by `parallel_scan`. The reassociation may differ between runs even on the same machine. However, if there are no worker threads available, execution associates identically to the serial form shown at the beginning of this section.

3.7.1 pre_scan_tag and final_scan_tag Classes

Summary

Types that distinguish the phases of `parallel_scan`..

Syntax

```
struct pre_scan_tag;
struct final_scan_tag;
```

Header

```
#include "tbb/parallel_scan.h"
```

Description

Types `pre_scan_tag` and `final_scan_tag` are dummy types used in conjunction with `parallel_scan`. See the example in Section 3.7 for how they are used in the signature of `operator()`.

Members

```
namespace tbb {

    struct pre_scan_tag {
        static bool is_final_scan();
    };

    struct final_scan_tag {
        static bool is_final_scan();
    };

}
```

3.7.1.1 bool is_final_scan()

Returns

True for a `final_scan_tag`, otherwise false.

3.7.2 Using the Partitioner Preview Feature

Summary

Template function that computes parallel prefix, with the splitting of the range guided by the Partitioner parameter.

Syntax

```
template<typename Range, typename Body, typename Partitioner>
        void parallel_scan( const Range& range, Body& body,
                            Partitioner& partitioner );
```

Header

```
#include "tbb/parallel_scan.h"
```