Programming Languages — Technical Specification for C++ Extensions for Parallelism

Warning
This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.
Copyright notice

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

ISO copyright officer
Case postale 56, CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Reproduction may be subject to royalty payments or a licensing agreement.
Violators may be prosecuted.
# Contents

1 **General** ................................................................. 4  
   1.1 Scope ............................................................... 4  
   1.2 Normative references ........................................... 4  
   1.3 Namespaces and headers ....................................... 4  
       1.3.1 Terms and definitions ................................... 5  
2 **Execution policies** .................................................. 6  
   2.1 In general .......................................................... 6  
   2.2 Header <experimental/execution_policy> synopsis ............... 6  
   2.3 Execution policy type trait ..................................... 7  
   2.4 Sequential execution policy .................................... 7  
   2.5 Parallel execution policy ........................................ 7  
   2.6 Parallel+Vector execution policy ............................... 7  
   2.7 Dynamic execution policy ....................................... 8  
       2.7.1 execution_policy construct/assign ......................... 8  
       2.7.2 execution_policy object access ........................... 9  
   2.8 Execution policy objects ........................................ 9  
3 **Parallel exceptions** ................................................ 10  
   3.1 Exception reporting behavior .................................. 10  
   3.2 Header <experimental/exception_list> synopsis .............. 10  
4 **Parallel algorithms** .................................................. 12  
   4.1 In general .......................................................... 12  
       4.1.1 Requirements on user-provided function objects ....... 12  
       4.1.2 Effect of execution policies on algorithm execution . 12  
       4.1.3 ExecutionPolicy algorithm overloads .................... 14  
   4.2 Definitions ........................................................ 14  
   4.3 Non-Numeric Parallel Algorithms ............................... 15  
       4.3.1 Header <experimental/algorithm> synopsis .............. 15  
       4.3.2 For each ..................................................... 16  
   4.4 Numeric Parallel Algorithms .................................... 17  
       4.4.1 Header <experimental/numeric> synopsis ................. 17  
       4.4.2 Reduce ...................................................... 20  
       4.4.3 Exclusive scan ............................................. 20  
       4.4.4 Inclusive scan .............................................. 21  
       4.4.5 Transform reduce .......................................... 22  
       4.4.6 Transform exclusive scan .................................. 23  
       4.4.7 Transform inclusive scan .................................. 24
1 General

1.1 Scope

This Technical Specification describes requirements for implementations of an interface that
can be used by computer programs written in the C++ programming language to invoke
algorithms with parallel execution. The algorithms described by this Technical Specification
are realizable across a broad class of computer architectures.

This Technical Specification is non-normative. Some of the functionality described by this
Technical Specification may be considered for standardization in a future version of C++,
but it is not currently part of any C++ standard. Some of the functionality in this Technical
Specification may never be standardized, and other functionality may be standardized in a
substantially changed form.

The goal of this Technical Specification is to build widespread existing practice for parallelism
in the C++ standard algorithms library. It gives advice on extensions to those vendors who wish
to provide them.

1.2 Normative references

The following referenced document is indispensable for the application of this document. For
dated references, only the edition cited applies. For undated references, the latest edition of
the referenced document (including any amendments) applies.

— ISO/IEC 14882:—, Programming Languages — C++

ISO/IEC 14882:— is herein called the C++ Standard. The library described in ISO/IEC 14882:—
clauses 17-30 is herein called the C++ Standard Library. The C++ Standard Library
components described in ISO/IEC 14882:— clauses 25, 26.7 and 20.7.2 are herein called the
C++ Standard Algorithms Library.

Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++14 §17)
is included into this Technical Specification by reference.

1.3 Namespaces and headers

Since the extensions described in this Technical Specification are experimental and not part of
the C++ Standard Library, they should not be declared directly within namespace std. Unless
otherwise specified, all components described in this Technical Specification are declared in
namespace std::experimental::parallel::v1.

[ Note: Once standardized, the components described by this Technical Specification are
expected to be promoted to namespace std. — end note ]

Unless otherwise specified, references to such entities described in this Technical Specification
are assumed to be qualified with std::experimental::parallel::v1, and references to entities
described in the C++ Standard Library are assumed to be qualified with std:::

Extensions that are expected to eventually be added to an existing header <meow> are provided
inside the <experimental/meow> header, which shall include the standard contents of <meow> as if by
#include <meow>

1. To be published. Section references are relative to N3937.
1.3.1 Terms and definitions

1 For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

2 A parallel algorithm is a function template described by this Technical Specification declared in namespace std::experimental::parallel::v1 with a formal template parameter named ExecutionPolicy.

3 Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

   - All operations of the categories of the iterators that the algorithm is instantiated with.
   - Functions on those sequence elements that are required by its specification.
   - User-provided function objects to be applied during the execution of the algorithm, if required by the specification.

These functions are herein called element access functions.

[ Example: The sort function may invoke the following element access functions:

   - Methods of the random-access iterator of the actual template argument, as per 24.2.7, as implied by the name of the template parameters RandomAccessIterator.
   - The swap function on the elements of the sequence (as per 25.4.1.1 [sort]/2).
   - The user-provided Compare function object.

   — end example ]
2 Execution policies

2.1 In general

This clause describes classes that are execution policy types. An object of an execution policy type indicates the kinds of parallelism allowed in the execution of an algorithm and expresses the consequent requirements on the element access functions.

[Example:

```c++
std::vector<int> v = ...

// standard sequential sort
std::sort(v.begin(), v.end());

using namespace std::experimental::parallel;

// explicitly sequential sort
sort(seq, v.begin(), v.end());

// permitting parallel execution
sort(par, v.begin(), v.end());

// permitting vectorization as well
sort(par_vec, v.begin(), v.end());

// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if (v.size() > threshold)
{
    exec = par;
}

sort(exec, v.begin(), v.end());
```
— end example ]

[Note: Because different parallel architectures may require idiosyncratic parameters for efficient execution, implementations of the Standard Library may provide additional execution policies to those described in this Technical Specification as extensions. — end note ]

2.2 Header <experimental/execution_policy> synopsis

```c++
namespace std {
    namespace experimental {
        namespace parallel {
            inline namespace v1 {
                // 2.3, Execution policy type trait
                template<class T> struct is_execution_policy;
                template<class T> constexpr bool is_execution_policy_v = is_execution_policy<T>::value;
            }
        }
    }
}
```

§ 2.2
// 2.4, Sequential execution policy
class sequential_execution_policy;

// 2.5, Parallel execution policy
class parallel_execution_policy;

// 2.6, Parallel+Vector execution policy
class parallel_vector_execution_policy;

// 2.7, Dynamic execution policy
class execution_policy;

2.3 Execution policy type trait

template<class T> struct is_execution_policy { see below };

1 is_execution_policy can be used to detect parallel execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

2 is_execution_policy<T> shall be a UnaryTypeTrait with a BaseCharacteristic of true_type if T is the type of a standard or implementation-defined execution policy, otherwise false_type.

[ Note: This provision reserves the privilege of creating non-standard execution policies to the library implementation. — end note ]

3 The behavior of a program that adds specializations for is_execution_policy is undefined.

2.4 Sequential execution policy

class sequential_execution_policy{ unspecified };

1 The class sequential_execution_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution may not be parallelized.

2.5 Parallel execution policy

class parallel_execution_policy{ unspecified };

1 The class parallel_execution_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

2.6 Parallel+Vector execution policy

class parallel_vector_execution_policy{ unspecified };

1 The class parallel_vector_execution_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be vectorized and parallelized.
2.7 Dynamic execution policy

The class `execution_policy` is a container for execution policy objects. `execution_policy` allows dynamic control over standard algorithm execution.

**Example:**

```cpp
std::vector<float> sort_me = ...

using namespace std::experimental::parallel;
execution_policy exec = seq;

if(sort_me.size() > threshold)
    { exec = std::par; }

std::sort(exec, std::begin(sort_me), std::end(sort_me));
```

Objects of type `execution_policy` shall be constructible and assignable from objects of type `T` for which `is_execution_policy<T>::value` is true.

2.7.1 `execution_policy` construct/assign

```cpp
template<class T> execution_policy(const T& exec);
```

**Effects:** Constructs an `execution_policy` object with a copy of `exec`'s state.

**Remarks:** This constructor shall not participate in overload resolution unless `is_execution_policy<T>::value` is true.

```cpp
template<class T> execution_policy& operator=(const T& exec);
```

**Effects:** Assigns a copy of `exec`'s state to `*this`.

**Returns:** `*this`. 

1 The class `execution_policy` is a container for execution policy objects. `execution_policy` allows dynamic control over standard algorithm execution.

2 Objects of type `execution_policy` shall be constructible and assignable from objects of type `T` for which `is_execution_policy<T>::value` is true.
2.7.2 execution_policy object access [parallel.execpol.access]

1    const type_info& type() const noexcept;

2   Returns: typeid(T), such that T is the type of the execution policy object contained by *this.

3    template<class T> T* get() noexcept;
    template<class T> const T* get() const noexcept;

4   Returns: If target_type() == typeid(T), a pointer to the stored execution policy object;
            otherwise a null pointer.

5   Requires: is_execution_policy<T>::value is true.

2.8 Execution policy objects [parallel.execpol.objects]

constexpr sequential_execution_policy seq{};
constexpr parallel_execution_policy par{};
constexpr parallel_vector_execution_policy par_vec{};

1 The header <experimental/execution_policy> declares a global object associated with each type of
execution policy defined by this Technical Specification.
3 Parallel exceptions

3.1 Exception reporting behavior

During the execution of a standard parallel algorithm, if temporary memory resources are required and none are available, the algorithm throws a `std::bad_alloc` exception.

During the execution of a standard parallel algorithm, if the invocation of an element access function terminates with an uncaught exception, the behavior of the program is determined by the type of execution policy used to invoke the algorithm:

- If the execution policy object is of type `class parallel_vector_execution_policy`, `std::terminate` shall be called.
- If the execution policy object is of type `sequential_execution_policy` or `parallel_execution_policy`, the execution of the algorithm terminates with an `exception_list` exception. All uncaught exceptions thrown during the invocations of element access functions shall be contained in the `exception_list`.

[Note: For example, the number of invocations of the user-provided function object in `for_each` is unspecified. When `for_each` is executed sequentially, only one exception will be contained in the `exception_list` object. — end note]

[Note: These guarantees imply that, unless the algorithm has failed to allocate memory and terminated with `std::bad_alloc`, all exceptions thrown during the execution of the algorithm are communicated to the caller. It is unspecified whether an algorithm implementation will “forge ahead” after encountering and capturing a user exception. — end note]

[Note: The algorithm may terminate with the `std::bad_alloc` exception even if one or more user-provided function objects have terminated with an exception. For example, this can happen when an algorithm fails to allocate memory while creating or adding elements to the `exception_list` object. — end note]

- If the execution policy object is of any other type, the behavior is implementation-defined.

3.2 Header `<experimental/exception_list>` synopsis

```cpp
namespace std {
    namespace experimental {
        namespace parallel {
            inline namespace v1 {

                class exception_list : public exception {
                
                    public:
                        typedef unspecified iterator;

                    size_t size() const noexcept;
                    iterator begin() const noexcept;
                    iterator end() const noexcept;
                }
            }
        }
    }
}
```
const char* what() const noexcept override;
};

The class `exception_list` owns a sequence of `exception_ptr` objects. The parallel algorithms may use the `exception_list` to communicate uncaught exceptions encountered during parallel execution to the caller of the algorithm.

The type `exception_list::iterator` shall fulfill the requirements of `ForwardIterator`.

size_t size() const noexcept;

Returns: The number of `exception_ptr` objects contained within the `exception_list`.

Complexity: Constant time.

iterator begin() const noexcept;

Returns: An iterator referring to the first `exception_ptr` object contained within the `exception_list`.

iterator end() const noexcept;

Returns: An iterator that is past the end of the owned sequence.

const char* what() const noexcept override;

Returns: An implementation-defined NTBS.
4 Parallel algorithms

4.1 In general

This clause describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

4.1.1 Requirements on user-provided function objects

1 Function objects passed into parallel algorithms as objects of type `BinaryPredicate`, `Compare`, and `BinaryOperation` shall not directly or indirectly modify objects via their arguments.

4.1.2 Effect of execution policies on algorithm execution

1 Parallel algorithms have template parameters named `ExecutionPolicy` which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

2 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `sequential_execution_policy` execute in sequential order in the calling thread.

3 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `parallel_execution_policy` are permitted to execute in an unordered fashion in either the invoking thread or in a thread implicitly created by the library to support parallel algorithm execution. Any such invocations executing in the same thread are indeterminately sequenced with respect to each other. [ Note: It is the caller's responsibility to ensure correctness, for example that the invocation does not introduce data races or deadlocks. — end note ]

[ Example:

```cpp
using namespace std::experimental::parallel;
int a[] = {0,1};
std::vector<int> v;
for_each(par, std::begin(a), std::end(a), [&](int i) {
  v.push_back(i*2+1);
});
```

The program above has a data race because of the unsynchronized access to the container `v`. — end example ]

[ Example:

```cpp
using namespace std::experimental::parallel;
std::atomic<int> x = 0;
int a[] = {1,2};
for_each(par, std::begin(a), std::end(a), [&](int n) {
  x.fetch_add(1, std::memory_order_relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order_relaxed) == 1) { }
});
```

The above example depends on the order of execution of the iterations, and is therefore undefined (may deadlock). — end example ]
Example:

```cpp
using namespace std::experimental::parallel;
int x=0;
std::mutex m;
int a[] = {1, 2};
for_each(par, std::begin(a), std::end(a), [&](int) {
   m.lock();
   ++x;
   m.unlock();
});
```

The above example synchronizes access to object `x` ensuring that it is incremented correctly.
— end example }

The invocations of element access functions in parallel algorithms invoked with an execution policy of type `parallel_vector_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, and unsequenced with respect to one another within each thread. 
[ Note: This means that multiple function object invocations may be interleaved on a single thread. — end note ]

[ Note: This overrides the usual guarantee from the C++ standard, Section 1.9 [intro.execution] that function executions do not interleave with one another. — end note ]

Since `parallel_vector_execution_policy` allows the execution of element access functions to be interleaved on a single thread, synchronization, including the use of mutexes, risks deadlock. Thus the synchronization with `parallel_vector_execution_policy` is restricted as follows:

A standard library function is `vectorization-unsafe` if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, and if it is not a memory allocation or deallocation function. Vectorization-unsafe standard library functions may not be invoked by user code called from `parallel_vector_execution_policy` algorithms.

[ Note: Implementations must ensure that internal synchronization inside standard library routines does not induce deadlock. — end note ]

Example:

```cpp
using namespace std::experimental::parallel;
int x=0;
std::mutex m;
int a[] = {1, 2};
for_each(par_vec, std::begin(a), std::end(a), [&](int) {
   m.lock();
   ++x;
   m.unlock();
});
```

The above program is invalid because the applications of the function object are not guaranteed to run on different threads. — end example }

[ Note: The application of the function object may result in two consecutive calls to `m.lock` on the same thread, which may deadlock. — end note ]

[ Note: The semantics of the `parallel_execution_policy` or the `parallel_vector_execution_policy` invocation allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation due to lack of resources. — end note ]
Algorithms invoked with an execution policy object of type `execution_policy` execute internally as if invoked with the contained execution policy object.

The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

### 4.1.3 ExecutionPolicy algorithm overloads

The Parallel Algorithms Library provides overloads for each of the algorithms named in Table 1, corresponding to the algorithms with the same name in the C++ Standard Algorithms Library. For each algorithm in Table 1, if there are overloads for corresponding algorithms with the same name in the C++ Standard Algorithms Library, the overloads shall have an additional template type parameter named `ExecutionPolicy`, which shall be the first template parameter. In addition, each such overload shall have the new function parameter as the first function parameter of type `ExecutionPolicy&&`.

Unless otherwise specified, the semantics of `ExecutionPolicy` algorithm overloads are identical to their overloads without.

Parallel algorithms shall not participate in overload resolution unless `is_execution_policy<decay_t<ExecutionPolicy>>::value` is true.

#### Table 1 — Table of parallel algorithms

<table>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>copy_n</td>
<td>count</td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>transform_exclusive_scan</td>
<td>transform_inclusive_scan</td>
<td>transform_reduce</td>
<td>uninitialized_copy</td>
</tr>
<tr>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
<td>unique</td>
</tr>
<tr>
<td>unique_copy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Note: Not all algorithms in the Standard Library have counterparts in Table 1. — end note]

### 4.2 Definitions

Define `GENERALIZED_SUM(op, a1, ..., aN)` as follows:

- `a1` when `N` is `1`
- `op(GENERALIZED_SUM(op, b1, ..., bK), GENERALIZED_SUM(op, bM, ..., bN))` where
  - `b1`, ..., `bN` may be any permutation of `a1`, ..., `aN` and
  - `1 < K+1 = M ≤ N`. 

§ 4.2
3 Define \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\texttt{op}, a_1, \ldots, a_N) as follows:

- \(a_1\) when \(N = 1\)
- \(\texttt{op}(\texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\texttt{op}, a_1, \ldots, a_K), \texttt{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\texttt{op}, a_M, \ldots, a_N))\) where \(1 < K+1 = M \leq N\).

4

4.3 Non-Numeric Parallel Algorithms

4.3.1 Header <experimental/algorithm> synopsis

```cpp
namespace std {
    namespace experimental {
        namespace parallel {
            inline namespace v1 {
                template<class ExecutionPolicy, class InputIterator, class Function>
                    void for_each(ExecutionPolicy&& exec,
                                    InputIterator first, InputIterator last,
                                    Function f);
                template<class InputIterator, class Size, class Function>
                    InputIterator for_each_n(InputIterator first, Size n,
                                               Function f);
                template<class ExecutionPolicy, class InputIterator, class Size, class Function>
                    InputIterator for_each_n(ExecutionPolicy&& exec,
                                               InputIterator first, Size n,
                                               Function f);
            }
        }
    }
}
```

§ 4.3.1
4.3.2 For each

```
1 template<class ExecutionPolicy,
    class InputIterator, class Function>
    void for_each(ExecutionPolicy&& exec,
                  InputIterator first, InputIterator last,
                  Function f);
```

Effects: Applies \( f \) to the result of dereferencing every iterator in the range \([\text{first}, \text{last})\). [Note: If the type of \text{first} satisfies the requirements of a mutable iterator, \( f \) may apply nonconstant functions through the dereferenced iterator. — end note]

Complexity: Applies \( f \) exactly \( \text{last} - \text{first} \) times.

Remarks: If \( f \) returns a result, the result is ignored.

Notes: Unlike its sequential form, the parallel overload of for_each does not return a copy of its \text{Function} parameter, since parallelization may not permit efficient state accumulation.

Requires: Unlike its sequential form, the parallel overload of for_each requires \text{Function} to meet the requirements of \text{CopyConstructible}.

```
2 template<class InputIterator, class Size, class Function>
    InputIterator for_each_n(InputIterator first, Size n,
                              Function f);
```

Requires: \text{Function} shall meet the requirements of \text{MoveConstructible} [Note: Function need not meet the requirements of \text{CopyConstructible}. — end note]

Effects: Applies \( f \) to the result of dereferencing every iterator in the range \([\text{first}, \text{first} + \text{n})\), starting from \text{first} and proceeding to \( \text{first} + \text{n} - 1 \). [Note: If the type of \text{first} satisfies the requirements of a mutable iterator, \( f \) may apply nonconstant functions through the dereferenced iterator. — end note]

Returns: \( \text{first} + \text{n} \) for non-negative values of \text{n} and \text{first} for negative values.

Remarks: If \( f \) returns a result, the result is ignored.

```
3 template<class ExecutionPolicy,
    class InputIterator, class Size, class Function>
    InputIterator for_each_n(ExecutionPolicy && exec,
                             InputIterator first, Size n,
                             Function f);
```

Effects: Applies \( f \) to the result of dereferencing every iterator in the range \([\text{first}, \text{first} + \text{n})\), starting from \text{first} and proceeding to \( \text{first} + \text{n} - 1 \). [Note: If the type of \text{first} satisfies the requirements of a mutable iterator, \( f \) may apply nonconstant functions through the dereferenced iterator. — end note]

Returns: \( \text{first} + \text{n} \) for non-negative values of \text{n} and \text{first} for negative values.

Remarks: If \( f \) returns a result, the result is ignored.

Notes: Unlike its sequential form, the parallel overload of for_each_n requires \text{Function} to meet the requirements of \text{CopyConstructible}. 

§ 4.3.2
4.4 Numeric Parallel Algorithms

4.4.1 Header <experimental/numeric> synopsis

namespace std {
namespace experimental {
namespace parallel {
inline namespace v1 {
  template<class InputIterator>
  typename iterator_traits<InputIterator>::value_type
  reduce(InputIterator first, InputIterator last);
  template<class ExecutionPolicy,
           class InputIterator>
  typename iterator_traits<InputIterator>::value_type
  reduce(ExecutionPolicy&& exec,
         InputIterator first, InputIterator last);
  template<class InputIterator, class T>
  T reduce(InputIterator first, InputIterator last, T init);
  template<class ExecutionPolicy,
           class InputIterator, class T>
  T reduce(ExecutionPolicy&& exec,
           InputIterator first, InputIterator last, T init);
  template<class InputIterator, class T, class BinaryOperation>
  T reduce(InputIterator first, InputIterator last, T init,
           BinaryOperation binary_op);
  template<class ExecutionPolicy, class InputIterator, class T, class BinaryOperation>
  T reduce(ExecutionPolicy&& exec,
           InputIterator first, InputIterator last, T init,
           BinaryOperation binary_op);
  template<class InputIterator, class OutputIterator, class T>
  OutputIterator
  exclusive_scan(InputIterator first, InputIterator last,
                 OutputIterator result, T init);
  template<class ExecutionPolicy,
           class InputIterator, class OutputIterator, class T>
  OutputIterator
  exclusive_scan(ExecutionPolicy&& exec,
                 InputIterator first, InputIterator last,
                 OutputIterator result, T init);
  template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
  OutputIterator
  exclusive_scan(InputIterator first, InputIterator last,
                 OutputIterator result, T init, BinaryOperation binary_op);
  template<class ExecutionPolicy,
           class InputIterator, class OutputIterator, class T>
  OutputIterator
  exclusive_scan(ExecutionPolicy&& exec,
                 InputIterator first, InputIterator last,
                 OutputIterator result, T init, BinaryOperation binary_op);

class T, class BinaryOperation>
OutputIterator
exclusive_scan(ExecutionPolicy&& exec,
InputIterator first, InputIterator last,
OutputIterator result,
T init, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result);

template<class ExecutionPolicy,
class InputIterator, class OutputIterator>
OutputIterator
inclusive_scan(ExecutionPolicy&& exec,
InputIterator first, InputIterator last,
OutputIterator result);

template<class InputIterator, class OutputIterator,
class BinaryOperation>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result,
BinaryOperation binary_op);

template<class ExecutionPolicy,
class InputIterator, class OutputIterator,
class BinaryOperation>
OutputIterator
inclusive_scan(ExecutionPolicy&& exec,
InputIterator first, InputIterator last,
OutputIterator result,
BinaryOperation binary_op);

template<class InputIterator, class UnaryOperation,
class T, class BinaryOperation>
T transform_reduce(InputIterator first, InputIterator last,
UnaryOperation unary_op,
T init, BinaryOperation binary_op);

template<class ExecutionPolicy,
class InputIterator, class UnaryOperation,
class T, class BinaryOperation>
T transform_reduce(InputIterator first, InputIterator last,
UnaryOperation unary_op,
T init, BinaryOperation binary_op);
class InputIterator, class UnaryOperation,
class T, class BinaryOperation>
T transform_reduce(ExecutionPolicy&& exec,
    InputIterator first, InputIterator last,
    UnaryOperation unary_op,
    T init, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator,
class UnaryOperation, class T, class BinaryOperation>
OutputIterator
    transform_exclusive_scan(InputIterator first, InputIterator last,
        OutputIterator result,
        UnaryOperation unary_op,
        T init, BinaryOperation binary_op);

template<class ExecutionPolicy,
class InputIterator, class OutputIterator,
class UnaryOperation, class T, class BinaryOperation>
OutputIterator
    transform_exclusive_scan(ExecutionPolicy&& exec,
        InputIterator first, InputIterator last,
        OutputIterator result,
        UnaryOperation unary_op,
        T init, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator,
class UnaryOperation, class BinaryOperation>
OutputIterator
    transform_inclusive_scan(InputIterator first, InputIterator last,
        OutputIterator result,
        UnaryOperation unary_op,
        BinaryOperation binary_op);

template<class ExecutionPolicy,
class InputIterator, class OutputIterator,
class UnaryOperation, class BinaryOperation>
OutputIterator
    transform_inclusive_scan(ExecutionPolicy&& exec,
        InputIterator first, InputIterator last,
        OutputIterator result,
        UnaryOperation unary_op,
        BinaryOperation binary_op);

template<class InputIterator, class OutputIterator,
class UnaryOperation, class BinaryOperation, class T>
OutputIterator
    transform_inclusive_scan(InputIterator first, InputIterator last,
        OutputIterator result,
        UnaryOperation unary_op,
        BinaryOperation binary_op, T init);

template<class ExecutionPolicy,
class InputIterator, class OutputIterator,
class UnaryOperation, class BinaryOperation, class T>
OutputIterator
    transform_inclusive_scan(ExecutionPolicy&& exec,
        InputIterator first, InputIterator last,
        OutputIterator result,
        UnaryOperation unary_op,
        BinaryOperation binary_op, T init);
transform_inclusive_scan(ExecutionPolicy&& exec,  
    InputIterator first, InputIterator last,  
    OutputIterator result,  
    UnaryOperation unary_op,  
    BinaryOperation binary_op, T init);
}
}
}

4.4.2 Reduce

1 template<class InputIterator>
    typename iterator_traits<InputIterator>::value_type
    reduce(InputIterator first, InputIterator last);

2 Effects: Same as reduce(first, last, typename iterator_traits<InputIterator>::value_type{}).

3 template<class InputIterator, class T>
    T reduce(InputIterator first, InputIterator last, T init);

4 Effects: Same as reduce(first, last, init, plus<>()).

5 template<class InputIterator, class T, class BinaryOperation>
    T reduce(InputIterator first, InputIterator last, T init,  
             BinaryOperation binary_op);

6 Returns: GENERALIZED_SUM(binary_op, init, *first, ..., *(first + (last - first) - 1)).

7 Requires: binary_op shall not invalidate iterators or subranges, nor modify elements in the  
            range [first,last).

8 Complexity: O(last - first) applications of binary_op.

9 Notes: The primary difference between reduce and accumulate is that the behavior of reduce  
        may be non-deterministic for non-associative or non-commutative binary_op.

4.4.3 Exclusive scan

1 template<class InputIterator, class OutputIterator,  
    class T>
    OutputIterator  
    exclusive_scan(InputIterator first, InputIterator last,  
                   OutputIterator result,  
                   T init);

2 Effects: Same as exclusive_scan(first, last, result, init, plus<>()).
template<class InputIterator, class OutputIterator, 
class T, class BinaryOperation>
OutputIterator
exclusive_scan(InputIterator first, InputIterator last, 
OutputIterator result, 
T init, BinaryOperation binary_op);

Effects: Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of 
\text{GENERALIZED_NONCOMMUTATIVE_SUM}(\text{binary_op}, \text{init}, \ast\text{first}, \ldots, \ast(\text{first} + (i - \text{result}) - 1)).

Returns: The end of the resulting range beginning at \text{result}.

Requires: \text{binary_op} shall not invalidate iterators or subranges, nor modify elements in the 
ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

Complexity: \(O(\text{last} - \text{first})\) applications of \text{binary_op}.

Notes: The difference between \text{exclusive_scan} and \text{inclusive_scan} is that \text{exclusive_scan} 
excludes the \(i\)th input element from the \(i\)th sum. If \text{binary_op} is not mathematically 
associative, the behavior of \text{exclusive_scan} may be non-deterministic.

4.4.4 Inclusive scan

\[\text{inclusive_scan}(\text{first, last, result, plus}())\]
template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator
inclusive_scan(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary_op, T init);

**Effects:** Assigns through each iterator \(i\) in \([result, result + (last - first))\) the value of \(\text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(binary\_op, *first, ..., *(first + (i - result)))\) or \(\text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(binary\_op, init, *first, ..., *(first + (i - result)))\) if \(init\) is provided.

**Returns:** The end of the resulting range beginning at \(result\).

**Requires:** \(binary\_op\) shall not invalidate iterators or subranges, nor modify elements in the ranges \([first, last)\) or \([result, result + (last - first))\).

**Complexity:** \(O(last - first)\) applications of \(binary\_op\).

**Notes:** The difference between \texttt{exclusive\_scan} and \texttt{inclusive\_scan} is that \texttt{inclusive\_scan} includes the \(i\)th input element in the \(i\)th sum. If \(binary\_op\) is not mathematically associative, the behavior of \texttt{inclusive\_scan} may be non-deterministic.

### 4.4.5 Transform reduce

**template<class InputIterator, class UnaryFunction, class T, class BinaryOperation>**

\[
T \text{ transform\_reduce}(InputIterator first, InputIterator last, UnaryOperation unary\_op, T init, BinaryOperation binary\_op);
\]

**Returns:** \(\text{GENERALIZED\_SUM}(binary\_op, init, unary\_op(*first), ..., unary\_op(*(first + (last - first) - 1)))\).

**Requires:** Neither \(unary\_op\) nor \(binary\_op\) shall invalidate subranges, or modify elements in the range \([first, last)\).

**Complexity:** \(O(last - first)\) applications each of \(unary\_op\) and \(binary\_op\).

**Notes:** \texttt{transform\_reduce} does not apply \(unary\_op\) to \(init\).
4.4.6 Transform exclusive scan

```
template<class InputIterator, class OutputIterator,
        class UnaryOperation,
        class T, class BinaryOperation>
OutputIterator
transform_exclusive_scan(InputIterator first, InputIterator last,
                           OutputIterator result,
                           UnaryOperation unary_op,
                           T init, BinaryOperation binary_op);
```

1 Effects: Assigns through each iterator \(i\) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of 
\[
\text{GENERALIZED_NONCOMMUTATIVE_SUM}(\text{binary_op, init, unary_op(*first), \ldots, unary_op(*\text{(first + (i - result) - 1))}).
\]

2 Returns: The end of the resulting range beginning at result.

3 Requires: Neither unary_op nor binary_op shall invalidate iterators or subranges, or modify 
elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

4 Complexity: \(O(\text{last} - \text{first})\) applications each of unary_op and binary_op.

5 Notes: The difference between transform_exclusive_scan and transform_inclusive_scan is that 
transform_exclusive_scan excludes the ith input element from the ith sum. If binary_op is not 
mathematically associative, the behavior of transform_exclusive_scan may be non-
deterministic. transform_exclusive_scan does not apply unary_op to init.
4.4.7 Transform inclusive scan

```
1 template<class InputIterator, class OutputIterator, 
    class UnaryOperation, 
    class BinaryOperation>
    OutputIterator 
    transform_inclusive_scan(InputIterator first, InputIterator last, 
          OutputIterator result, 
          UnaryOperation unary_op, 
          BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, 
    class UnaryOperation, 
    class BinaryOperation, class T>
    OutputIterator 
    transform_inclusive_scan(InputIterator first, InputIterator last, 
          OutputIterator result, 
          UnaryOperation unary_op, 
          BinaryOperation binary_op, T init);
```

2 Effects: Assigns through each iterator \( i \) in \([\text{result}, \text{result} + (\text{last} - \text{first}))\) the value of 
\( \text{GENERALIZED NONCOMMUTATIVE SUM}(\text{binary_op}, \text{unary_op}(\text{*first}), ..., \text{unary_op}(\text{*(first + (i - result)))}) \) or 
\( \text{GENERALIZED NONCOMMUTATIVE SUM}(\text{binary_op}, \text{init}, \text{unary_op}(\text{*first}), ..., \text{unary_op}(\text{*(first + (i - result)))}) \) if init is provided.

3 Returns: The end of the resulting range beginning at result.

4 Requires: Neither unary_op nor binary_op shall invalidate iterators or subranges, or modify 
elements in the ranges \([\text{first}, \text{last})\) or \([\text{result}, \text{result} + (\text{last} - \text{first}))\).

5 Complexity: \( \text{O}(\text{last} - \text{first}) \) applications each of unary_op and binary_op.

6 Notes: The difference between transform_exclusive_scan and transform_inclusive_scan is that 
transform_inclusive_scan includes the ith input element from the ith sum. If binary_op is not 
mathematically associative, the behavior of transform_inclusive_scan may be non-
deterministic. transform_inclusive_scan does not apply unary_op to init.