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

*Langages de programmation — Spécification technique pour les
extensions C++ relatives au parallélisme*

IECNORM.COM : Click to view the full PDF of ISO/IEC TS 19570:2015

IECNORM.COM : Click to view the full PDF of ISO/IEC TS 19570:2015



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2015

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
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

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: [Foreword - Supplementary information](#)

ISO/IEC TS 19570 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Contents

1	General	4
1.1	Scope	4
1.2	Normative references	4
1.3	Namespaces and headers	4
1.4	Terms and definitions	5
1.5	Feature-testing recommendations	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	22
4.4.7	Transform inclusive scan	23

1 General

[parallel.general]

1.1 Scope

[parallel.general.scope]

- ¹ This Technical Specification describes requirements for implementations of an interface that computer programs written in the C++ programming language may use to invoke algorithms with parallel execution. The algorithms described by this Technical Specification are realizable across a broad class of computer architectures.
- ² 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.
- ³ 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

[parallel.general.references]

- ¹ 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:—¹, *Programming Language — C++*
- ² 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*.
- ³ 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

[parallel.general.namespaces]

- ¹ 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*]
- ² 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::`.
- ³ 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

1. To be published. Section references are relative to [N3937](#).

```
#include <meow>
```

1.4 Terms and definitions

[\[parallel.general.defns\]](#)

- 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.
 - Operations on those function objects required by the specification. [*Note:* See clause 25.1 of *C++ Standard Algorithms Library*. — *end note*]

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*]

1.5 Feature-testing recommendations

[\[parallel.general.features\]](#)

- ¹ An implementation that provides support for this Technical Specification shall define the feature test macro(s) in Table 1.

Table 1 — Feature Test Macro(s)

Name	Value	Header
<code>__cpp_lib_experimental_parallel_algorithm</code>	201505	<code><experimental/algorithm></code> <code><experimental/exception_list></code> <code><experimental/execution_policy></code> <code><experimental/numeric></code>

2 Execution policies

[\[parallel.execpol\]](#)

2.1 In general

[\[parallel.execpol.general\]](#)

- ¹ 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*:

```
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 [\[parallel.execpol.synopsis\]](#)

```
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;
```

```

// 2.4, 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

[\[parallel.execpol.type\]](#)

```
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

[\[parallel.execpol.seq\]](#)

```
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

[\[parallel.execpol.par\]](#)

```
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

[\[parallel.execpol.vec\]](#)

```
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

[\[parallel.execpol.dynamic\]](#)

```
class execution_policy
{
public:
    // 2.7.1, execution_policy construct/assign
    template<class T> execution_policy(const T& exec);
    template<class T> execution_policy& operator=(const T& exec);

    // 2.7.2, execution_policy object access
    const type_info& type() const noexcept;
    template<class T> T* get() noexcept;
    template<class T> const T* get() const noexcept;
};
```

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

[*Example*:

```
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));
```

— *end example*]

- 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.

2.7.1 execution_policy construct/assign

[\[parallel.execpol.con\]](#)

- ¹ `template<class T> execution_policy(const T& exec);`

² *Effects*: Constructs an `execution_policy` object with a copy of `exec`'s state.

³ *Remarks*: This constructor shall not participate in overload resolution unless `is_execution_policy<T>::value` is true.

- ⁴ `template<class T> execution_policy& operator=(const T& exec);`

⁵ *Effects*: Assigns a copy of `exec`'s state to `*this`.

⁶ *Returns*: `*this`.

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

[\[parallel.exceptions\]](#)

3.1 Exception reporting behavior

[\[parallel.exceptions.behavior\]](#)

- ¹ 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.
- ² During the execution of a standard parallel algorithm, if the invocation of an element access function exits via 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 exits via an exception. The exception shall be an `exception_list` containing all uncaught exceptions thrown during the invocations of element access functions, or optionally the uncaught exception if there was only one.

[*Note:* For example, when `for_each` is executed sequentially, if an invocation of the user-provided function object throws an exception, `for_each` can exit via the uncaught exception, or throw an `exception_list` containing the original exception. — *end note*]

[*Note:* These guarantees imply that, unless the algorithm has failed to allocate memory and exits via `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 exit via the `std::bad_alloc` exception even if one or more user-provided function objects have exited via 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

3.2 Header `<experimental/exception_list>` synopsis

[\[parallel.exceptions.synopsis\]](#)

```
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;
    };
}
}
}
}

```

- ¹ 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.
- ² The type `exception_list::iterator` shall fulfill the requirements of `ForwardIterator`.
- ³ `size_t size() const noexcept;`
 - ⁴ *Returns:* The number of `exception_ptr` objects contained within the `exception_list`.
 - ⁵ *Complexity:* Constant time.
- ⁶ `iterator begin() const noexcept;`
 - ⁷ *Returns:* An iterator referring to the first `exception_ptr` object contained within the `exception_list`.
- ⁸ `iterator end() const noexcept;`
 - ⁹ *Returns:* An iterator that is past the end of the owned sequence.
- ¹⁰ `const char* what() const noexcept override;`
 - ¹¹ *Returns:* An implementation-defined NTBS.

4 Parallel algorithms

[\[parallel.alg\]](#)

4.1 In general

[\[parallel.alg.general\]](#)

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

[\[parallel.alg.general.user\]](#)

- 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

[\[parallel.alg.general.exec\]](#)

- 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:*

```
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:*

```
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*:

```
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*]

- 4 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*]

- 5 [*Example*:

```
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*]

- ⁶ Algorithms invoked with an execution policy object of type `execution_policy` execute internally as if invoked with the contained execution policy object.
- ⁷ The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

4.1.3 ExecutionPolicy algorithm overloads

[parallel.alg.overloads]

- ¹ 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 2, 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&&`.
- ² Unless otherwise specified, the semantics of `ExecutionPolicy` algorithm overloads are identical to their overloads without.
- ³ Parallel algorithms shall not participate in overload resolution unless `is_execution_policy<decay_t<ExecutionPolicy>>::value` is true.

Table 2 — Table of parallel algorithms

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

[Note: Not all algorithms in the Standard Library have counterparts in Table 2. — end note]

4.2 Definitions

[parallel.alg.defns]

- ¹ 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`.

³ Define *GENERALIZED_NONCOMMUTATIVE_SUM*(op, a1, ..., aN) as follows:

- a1 when N is 1
- op(*GENERALIZED_NONCOMMUTATIVE_SUM*(op, a1, ..., aK), *GENERALIZED_NONCOMMUTATIVE_SUM*(op, aM, ..., aN) where 1 < K+1 = M ≤ N.

⁴

4.3 Non-Numeric Parallel Algorithms

[\[parallel.alg.ops\]](#)

4.3.1 Header <experimental/algorithm> synopsis

[\[parallel.alg.ops.synopsis\]](#)

```
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);
}
}
}
}
```

4.3.2 For each**[parallel.alg.foreach]**

- 1 `template<class ExecutionPolicy,`
`class InputIterator, class Function>`
`void for_each(ExecutionPolicy&& exec,`
`InputIterator first, InputIterator last,`
`Function f);`
- 2 *Effects:* Applies *f* to the result of dereferencing every iterator in the range `[first,last)`.
[*Note:* If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply nonconstant functions through the dereferenced iterator. — end note]
- 3 *Complexity:* Applies *f* exactly `last - first` times.
- 4 *Remarks:* If *f* returns a result, the result is ignored.
- 5 *Notes:* Unlike its sequential form, the parallel overload of `for_each` does not return a copy of its *Function* parameter, since parallelization may not permit efficient state accumulation.
- 6 *Requires:* Unlike its sequential form, the parallel overload of `for_each` requires *Function* to meet the requirements of `CopyConstructible`.
- 7 `template<class InputIterator, class Size, class Function>`
`InputIterator for_each_n(InputIterator first, Size n,`
`Function f);`
- 8 *Requires:* *Function* shall meet the requirements of `MoveConstructible` [*Note:* *Function* need not meet the requirements of `CopyConstructible`. — end note]
- 9 *Effects:* Applies *f* to the result of dereferencing every iterator in the range `[first,first + n)`, starting from *first* and proceeding to `first + n - 1`. [*Note:* If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply nonconstant functions through the dereferenced iterator. — end note]
- 10 *Returns:* `first + n` for non-negative values of *n* and *first* for negative values.
- 11 *Remarks:* If *f* returns a result, the result is ignored.
- 12 `template<class ExecutionPolicy,`
`class InputIterator, class Size, class Function>`
`InputIterator for_each_n(ExecutionPolicy && exec,`
`InputIterator first, Size n,`
`Function f);`
- 13 *Effects:* Applies *f* to the result of dereferencing every iterator in the range `[first,first + n)`, starting from *first* and proceeding to `first + n - 1`. [*Note:* If the type of *first* satisfies the requirements of a mutable iterator, *f* may apply nonconstant functions through the dereferenced iterator. — end note]
- 14 *Returns:* `first + n` for non-negative values of *n* and *first* for negative values.
- 15 *Remarks:* If *f* returns a result, the result is ignored.
- 16 *Notes:* Unlike its sequential form, the parallel overload of `for_each_n` requires *Function* to meet the requirements of `CopyConstructible`.

4.4 Numeric Parallel Algorithms

[\[parallel.alg.numeric\]](#)

4.4.1 Header <experimental/numeric> synopsis

[\[parallel.alg.numeric.synopsis\]](#)

```

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

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(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

```