Document Number: N3850
Date: 2014-02-XX
Reply to: Jared Hoberock
NVIDIA Corporation
jhoberock@nvidia.com

Working Draft, Technical Specification on C++ Extensions for Parallelism

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 General [general]

1.1 Scope [general.scope]

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.

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 [general.references]

The following reference document is indepensible 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 14882:2011, Programming Languages – C++

ISO/IEC 14882:2011 is herein called the C++ Standard. The library described in ISO/IEC 14882:2011 clauses 17-30 is herein called the C++ Standard Library. The C++ Standard Library components described in ISO/IEC 14882:2011 clauses 25 and 26.7 are herein called the C++ Standard Algorithms Library.

Unless otherwise specified, the whole of the C++ Standard Library introduction [lib.library] is included into this Technical Specification by reference.

1.3 Namespaces and headers [general.namespaces]

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`.

[*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 other entities described in this Technical Specification are assumed to be qualified with `std::experimental::parallel`, 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>
```

1.4 Terms and definitions [general.defns]

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

XXX define *user-provided function objects* ?

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

2 Execution policies [execpol]

2.1 In general [execpol.general]

This subclause describes classes that represent *execution policies*. An *execution policy* is an object that expresses the requirements on the ordering of functions invoked as a consequence of the invocation of a standard algorithm. Execution policies afford standard algorithms the discretion to execute in parallel.

[*Example:*

```
std::vector<int> vec = ...

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

using namespace std::experimental::parallel;

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

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

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

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

sort(exec, vec.begin(), vec.end());
```

– *end example*]

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

2.2 Header <experimental/execution_policy> synopsis [execpol.synop]

```
#include <type_traits>

namespace std {
namespace experimental {
namespace parallel {
  // 2.3, execution policy type trait
  template<class T> struct is_execution_policy;

  // 2.4, sequential execution policy
  class sequential_execution_policy;

  // 2.5, parallel execution policy
  class parallel_execution_policy;

  // 2.6, vector execution policy
  class vector_execution_policy

  // 2.7, dynamic execution policy
  class execution_policy;

  // 2.8, specialized algorithms
  void swap(sequential_execution_policy &a, sequential_execution_policy &b);
  void swap(parallel_execution_policy &a, parallel_execution_policy &b);
  void swap(vector_execution_policy &a, vector_execution_policy &b);
  void swap(execution_policy &a, execution_policy &b);

  // 2.9, standard execution policy objects
  extern const sequential_execution_policy seq;
  extern const parallel_execution_policy   par;
  extern const vector_execution_policy     vec;
}
}
}
```

1. An implementation may provide additional execution policy types besides `parallel_execution_policy`, `sequential_execution_policy`, `vector_execution_policy`, or `execution_policy`.

2.3 Execution policy type trait [execpol.type]

```
namespace std {
namespace experimental {
namespace parallel {
  template<class T> struct is_execution_policy
    : integral_constant<bool, 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. If `T` is the type of a standard or implementation-defined non-standard execution policy, `is_execution_policy<T>` shall be publicly derived from `integral_constant<bool,true>`, otherwise from `integral_constant<bool,false>`.

3. The effect of specializing `is_execution_policy` for a type which is not defined by the library is unspecified.

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

2.4 Sequential execution policy [execpol.seq]

```
namespace std {
namespace experimental {
namespace parallel {

  class sequential_execution_policy
  {
    void swap(sequential_execution_policy &other);
  };
}
}
}
```

1. The class `sequential_execution_policy` provides a mechanism to require a standard algorithm invocation to execute in a sequential order.

2. Implementations of `sequential_execution_policy` are permitted to provide additional non-standard data and function members.

   [*Note:* This provision permits `sequential_execution_policy` objects to be stateful. – *end note.*]

```
void swap(sequential_execution_policy &other);
```

2. *Effects:* Swaps the state of `*this` and `other`.

2.5 Parallel execution policy [execpol.par]

```
namespace std {
namespace experimental {
namespace parallel {

  class parallel_execution_policy
  {
    void swap(parallel_execution_policy &other);
  };
}
}
}
```

1. The class `parallel_execution_policy` provides a mechanism to allow a standard algorithm invocation to execute in an unordered fashion when executed on separate threads, and indeterminately sequenced when executed on a single thread.

2. Implementations of `parallel_execution_policy` are permitted to provide additional non-standard data and function members.

   [*Note:* This provision permits `parallel_execution_policy` objects to be stateful. – *end note.*]

```
void swap(parallel_execution_policy &other);
```

2. *Effects:* Swaps the state of `*this` and `other`.

2.6 Vector execution policy [execpol.vec]

```
namespace std {
namespace experimental {
namespace parallel {

  class vector_execution_policy
  {
    void swap(vector_execution_policy &other);
  };
}
}
}
```

1. The class `vector_execution_policy` provides a mechanism to allow a standard algorithm invocation to execute in an unordered fashion when executed on separate threads, and unordered when executed on a single thread.

2. Implementations of `vector_execution_policy` are permitted to provide additional non-standard data and function members.

   [*Note:* This provision permits `vector_execution_policy` objects to be stateful. – *end note.*]

```
void swap(vector_execution_policy &other);
```

2. *Effects:* Swaps the state of `*this` and `other`.

2.7 Dynamic execution policy [execpol.dynamic]

```
namespace std {
namespace experimental {
namespace parallel {

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

```
      // 2.7.2, object access
      const type_info& target_type() const;
      template<class T> T *target();
      template<class T> const T *target() const;
  };
}
}
}
```

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

[*Example:*

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

std::execution_policy exec = std::seq;

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

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

– *end example*]

3. The stored dynamic value of an `execution_policy` object may be retrieved.
   [*Example:*

```
void some_api(std::execution_policy exec, int arg1, double arg2)
{
  if(exec.target_type() == typeid(std::seq))
  {
    std::cout << "Received a sequential policy" << std::endl;
    auto *exec_ptr = exec.target<std::sequential_execution_policy>();
  }
  else if(exec.target_type() == typeid(std::par))
  {
    std::cout << "Received a parallel policy" << std::endl;
    auto *exec_ptr = exec.target<std::parallel_execution_policy>();
  }
  else if(exec.target_type() == typeid(std::vec))
  {
    std::cout << "Received a vector policy" << std::endl;
    auto *exec_ptr = exec.target<std::vector_execution_policy>();
  }
  else
  {
    std::cout << "Received some other kind of policy" << std::endl;
  }
}
```

– *end example*]

4. Objects of type `execution_policy` shall be constructible and assignable from any additional non-standard execution policy provided by the implementation.

2.7.1 `execution_policy` construct/assign/swap

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

1. *Effects:* Constructs an `execution_policy` object with a copy of `exec`'s state.
2. *Remarks:* This signature does not participate in overload resolution if `is_execution_policy<T>::value` is `false`.

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

3. *Effects:* Assigns a copy of `exec`'s state to `*this`.
4. *Returns:* `*this`.
5. *Remarks:* This signature does not participate in overload resolution if `is_execution_policy<T>::value` is `false`.

```
void swap(execution_policy &other);
```

1. *Effects:* Swaps the stored object of `*this` with that of `other`.

2.7.2 `execution_policy` object access

```
const type_info &target_type() const;
```

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

```
template<class T> T *target();
template<class T> const T *target() const;
```

2. *Returns:* If `target_type() == typeid(T)`, a pointer to the stored execution policy object; otherwise a null pointer.
3. *Remarks:* This signature does not participate in overload resolution if `is_execution_policy<T>::value` is `false`.

2.8 Execution policy specialized algorithms [execpol.algorithms]

```
void swap(sequential_execution_policy &a, sequential_execution_policy &b);
void swap(parallel_execution_policy &a, parallel_execution_policy &b);
void swap(vector_execution_policy &a, vector_execution_policy &b);
void swap(execution_policy &a, execution_policy &b);
```

1. *Effects:* `a.swap(b)`.

2.9 Standard execution policy objects [execpol.objects]

```
namespace std {
namespace experimental {
namespace parallel {
  extern const sequential_execution_policy seq;
  extern const parallel_execution_policy   par;
  extern const vector_execution_policy     vec;
}
}
}
```

1. The header `<execution_policy>` declares a global object associated with each standard execution policy.
2. An implementation may provide additional execution policy objects besides `seq`, `par`, or `vec`.
3. Concurrent access to these objects shall not result in a data race.

```
const sequential_execution_policy seq;
```

4. The object `seq` requires a standard algorithm to execute sequentially.

```
const parallel_execution_policy par;
```

5. The object `par` allows a standard algorithm to execute in an unordered fashion when executed on separate threads, and indeterminately sequenced when executed on a single thread.

```
const vector_execution_policy vec;
```

6. The object `vec` allows a standard algorithm to execute in an unordered fashion when executed on separate threads, and unordered when executed on a single thread.

3 Parallel exceptions [exceptions]

3.1 Exception reporting behavior [exceptions.behavior]

1. During the execution of a standard parallel algorithm, if the application of a function object 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 `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 application of user-provided function objects shall be contained in the `exception_list`, however the number of such exceptions is unspecified.

  [*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*]

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

2. If temporary memory resources are required by the algorithm and none are available, the algorithm may terminate with an `std::bad_alloc` exception.

   [*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*]

3.2 Header `<experimental/exception>` synopsis

```
namespace std {
namespace experimental {
namespace parallel {
  class exception_list : public exception
  {
    public:
      typedef exception_ptr     value_type;
      typedef const value_type& reference;
      typedef const value_type& const_reference;
      typedef size_t            size_type;
      typedef unspecified       iterator;
      typedef unspecified       const_iterator;

      size_t size() const;
      iterator begin() const;
      iterator end() const;

    private:
      std::list<exception_ptr> exceptions_; // exposition only
  };
}
}
}
```

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

```
size_t size() const;
```

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

```
exception_list::iterator begin() const;
```

3. *Returns:* An iterator pointing to the first `exception_ptr` object contained within the `exception_list`.

```
exception_list::iterator end() const;
```

4. *Returns:* An iterator pointing to one position past the last `exception_ptr` object contained within the `exception_list`.

4 Parallel algorithms [alg]

4.1 In general [alg.general]

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

4.1.1 Effect of execution policies on parallel algorithm execution [alg.general.exec]

1. Parallel algorithms accept execution policy parameters which describe the manner in which they apply user-provided function objects.

2. The applications of function objects in parallel algorithms invoked with an execution policy object of type `sequential_execution_policy` execute in sequential order in the calling thread.

3. The applications of function objects in parallel algorithms invoked with an execution policy object of type `parallel_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, or indeterminately sequenced if executed on one thread. [*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;
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 applications of function objects in parallel algorithms invoked with an execution policy of type `vector_execution_policy` are permitted to execute in an unordered fashion in unspecified threads, or unsequenced if executed on one thread. [*Note:* as a consequence, function objects governed by the `vector_execution_policy` policy must not synchronize with each other. Specifically, they must not acquire locks. — *end note*]

   [*Example:*

```
using namespace std::experimental::parallel;
int x;
std::mutex m;
int a[] = {1,2};
for_each(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.

   [*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*]

   – *end example*]

   [*Note:* The semantics of the `parallel_execution_policy` or the `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.*]

5. If they exist, a parallel algorithm invoked with an execution policy object of type `parallel_execution_policy` or `vector_execution_policy` may apply iterator member functions of a stronger category than its specification requires. In this case, the application of these member functions are subject to provisions 3. and 4. above, respectively.

   [*Note:* For example, an algorithm whose specification requires `InputIterator` but receives a concrete iterator of the category `RandomAccessIterator` may use `operator[]`. In this case, it is the algorithm caller's responsibility to ensure `operator[]` is race-free. – *end note.*]

6. Algorithms invoked with an execution policy object of type `execution_policy` execute internally as if invoked with instances of type `sequential_execution_policy`, `parallel_execution_policy`, or a non-standard implementation-defined execution policy depending on the dynamic value of the `execution_policy` object.

7. The semantics of parallel algorithms invoked with an execution policy object of type other than those described by this Technical Specification are unspecified.

4.2 Specialized memory algorithms [alg.memory]

This subclause defines function templates for constructing multiple objects in uninitialized memory buffers in parallel.

4.2.1 Header <experimental/memory> synopsis [alg.memory.synop]

```
namespace std {
namespace experimental {
namespace parallel {
  template<class ExecutionPolicy,
           class InputIterator, class ForwardIterator>
    ForwardIterator uninitialized_copy(ExecutionPolicy &&exec,
                                       InputIterator first, InputIterator last,
                                       ForwardIterator result);
  template<class ExecutionPolicy,
           class ForwardIterator, class T>
    void uninitialized_fill(ExecutionPolicy &&exec,
                            ForwardIterator first, ForwardIterator last
                            const T& x);
  template<class ExecutionPolicy,
           class ForwardIterator, class Size>
    ForwardIterator uninitialized_fill_n(ExecutionPolicy &&exec,
                                         ForwardIterator first, Size n,
                                         const T& x);
}
}
}
```

4.2.2 uninitialized_copy [alg.memory.uninitialized.copy]

```
template<class ExecutionPolicy,
         class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(ExecutionPolicy &&exec,
                                     InputIterator first, InputIterator last,
                                     ForwardIterator result);
template<class ExecutionPolicy,
         class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(ExecutionPolicy &&exec,
                                       InputIterator first, Size n,
                                       ForwardIterator result);
```

1. *Effects:* Copy constructs the element referenced by every iterator `i` in the range `[result,result + (last - first))` as if by the expression

   ```
   ::new (static_cast<void*>(&*i))
       typename iterator_traits<ForwardIterator>::value_type(*(first + (i - result)))
   ```

   The execution of the algorithm is parallelized as determined by `exec`.

2. *Returns:* `result + (last - first)`.

3. *Complexity:* `O(last - first)`.

4. *Remarks:* Neither signature shall participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(ExecutionPolicy &&exec,
                                       InputIterator first, Size n,
                                       ForwardIterator result);
```

1. *Effects:* Copy constructs the element referenced by every iterator `i` in the range `[result,result + n)` as if by the expression

   ```
   ::new (static_cast<void*>(&*i))
       typename iterator_traits<ForwardIterator>::value_type(*(first + (i - result)))
   ```

The execution of the algorithm is parallelized as determined by `exec`.

2. *Returns:* `result + n`.
3. *Complexity:* `O(n)`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.2.3 `uninitialized_fill` [alg.memory.uninitialized.fill]

```
template<class ExecutionPolicy,
         class ForwardIterator, class T>
  void uninitialized_fill(ExecutionPolicy &&exec,
                          ForwardIterator first, ForwardIterator last
                          const T& x);
```

1. *Effects:* Copy constructs the element referenced by every iterator `i` in the range `[first,last)` as if by the expression

   ```
   ::new (static_cast<void*>(&*i))
       typename iterator_traits<ForwardIterator>::value_type(x)
   ```

The execution of the algorithm is parallelized as determined by `exec`.

2. *Complexity:* `O(last - first)`.
3. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.2.4 `uninitialized_fill_n` [alg.memory.uninitialized.fill.n]

```
template<class ExecutionPolicy,
         class ForwardIterator, class Size>
  ForwardIterator uninitialized_fill_n(ExecutionPolicy &&exec,
                                       ForwardIterator first, Size n,
                                       const T& x);
```

1. *Effects:* Copy constructs the element referenced by every iterator `i` in the range `[first,first + n)` as if by the expression

```
::new (static_cast<void*>(&*i))
    typename iterator_traits<ForwardIterator>::value_type(x)
```

The execution of the algorithm is parallelized as determined by `exec`.

2. *Returns:* `first + n`.
3. *Complexity:* `O(n)`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3 Generic algorithms library [alg.gen]

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

4.3.1 Header `<experimental/algorithm>` synopsis [alg.gen.synop]

```
namespace std {
namespace experimental {
namespace parallel {
  // non-modifying sequence operations:
  template<class ExecutionPolicy,
           class InputIterator, class Predicate>
    bool all_of(ExecutionPolicy &&exec,
                InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy,
           class InputIterator, class Predicate>
    bool any_of(ExecutionPolicy &&exec,
                InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy,
           class InputIterator, class Predicate>
    bool none_of(ExecutionPolicy &&exec,
                 InputIterator first, InputIterator last, Predicate pred);

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

  template<class ExecutionPolicy,
           class InputIterator, class T>
    InputIterator find(ExecutionPolicy &&exec,
```

```
                    InputIterator first, InputIterator last,
                    const T& value);
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  InputIterator find_if(ExecutionPolicy &&exec,
                        InputIterator first, InputIterator last,
                        Predicate pred);
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  InputIterator find_if_not(ExecutionPolicy &&exec,
                            InputIterator first, InputIterator last,
                            Predicate pred);

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy &exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy &&exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);

template<class ExecutionPolicy,
         class InputIterator, class ForwardIterator>
  InputIterator
    find_first_of(ExecutionPolicy &&exec,
                  InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class ExecutionPolicy,
         class InputIterator, class ForwardIterator,
         class BinaryPredicate>
  InputIterator
    find_first_of(ExecutionPolicy &&exec,
                  InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);

template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator adjacent_find(ExecutionPolicy &&exec, ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find(ExecutionPolicy &&exec, ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);

template<class ExecutionPolicy,
         class InputIterator, class EqualityComparable>
  typename iterator_traits<InputIterator>::difference_type
```

```
    count(ExecutionPolicy &&exec,
          InputIterator first, InputIterator last, const EqualityComparable &value);
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  typename iterator_traits<InputIterator>::difference_type
    count_if(ExecutionPolicy &&exec,
             InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  pair<InputIterator1,InputIterator2>
    mismatch(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1,InputIterator2>
    mismatch(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  bool equal(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class BinaryPredicate>
  bool equal(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search(ExecutionPolicy &&exec,
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1 search(ExecutionPolicy &&exec,
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate pred);
template<class ExecutionPolicy,
         class ForwardIterator, class Size, class T>
  ForwardIterator search_n(ExecutionPolicy &&exec,
                           ForwardIterator first, ForwardIterator last, Size count,
                           const T& value);
template<class ExecutionPolicy,
         class ForwardIterator, class Size, class T, class BinaryPredicate>
  ForwardIterator search_n(ExecutionPolicy &&exec,
                           ForwardIterator first, ForwardIterator last, Size count,
                           const T& value, BinaryPredicate pred);
```

```
// modifying sequence operations:
// copy:
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator>
  OutputIterator copy(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last,
                      OutputIterator result);

template<class ExecutionPolicy,
         class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(ExecutionPolicy &&exec,
                        InputIterator first, Size n,
                        OutputIterator result);

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

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

// swap:
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ExecutionPolicy &&exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator1 first2);

template<class ExecutionPolicy,
         class InputIterator, class OutputIterator,
         class UnaryOperation>
  OutputIterator transform(ExecutionPolicy &&exec,
                           InputIterator first, InputIterator last,
                           OutputIterator result, UnaryOperation op);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
  OutputIterator
    transform(ExecutionPolicy &&exec,
              InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);

template<class ExecutionPolicy,
```

```
        class ForwardIterator, class T>
  void replace(ExecutionPolicy &&exec,
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
template<class ExecutionPolicy,
        class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy &&exec,
                  ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
template<class ExecutionPolicy,
        class InputIterator, class OutputIterator, class T>
  OutputIterator
    replace_copy(ExecutionPolicy &&exec,
                 ForwardIterator first, ForwardIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);
template<class ExecutionPolicy,
        class InputIterator, class OutputIterator, class Predicate, class T>
  OutputIterator
    replace_copy_if(ExecutionPolicy &&exec,
                    InputIterator first, InputIterator last,
                    OutputIterator result,

template<class ExecutionPolicy,
        class ForwardIterator, class T>
  void fill(ExecutionPolicy &&exec,
            ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy,
        class OutputIterator, class Size, class T>
  void fill_n(ExecutionPolicy &&exec,
              OutputIterator first, Size n, const T& value);

template<class ExecutionPolicy,
        class ForwardIterator, class Generator>
  void generate(ExecutionPolicy &&exec,
                ForwardIterator first, ForwardIterator last, Generator gen);
template<class ExecutionPolicy,
        class OutputIterator, class Size, class Generator>
  OutputIterator generate_n(ExecutionPolicy &&exec,
                            OutputIterator first, Size n, Generator gen);

template<class ExecutionPolicy,
        class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy,
        class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy &&exec,
                            ForwardIterator first, ForwardIterator last, Predicate pred);
template<class ExecutionPolicy,
        class InputIterator, class OutputIterator, class T>
  OutputIterator
    remove_copy(ExecutionPolicy &&exec,
                InputIterator first, InputIterator last,
```

```
                OutputIterator result, const T& value);
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    remove_copy_if(ExecutionPolicy &&exec,
                   InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);

template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, typename BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator last
                         BinaryPredicate pred);

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

template<class ExecutionPolicy,
         class BidirectionalIterator>
  void reverse(ExecutionPolicy &&exec,
               BidirectionalIterator first, BidirectionalIterator last);

template<class ExecutionPolicy,
         class BidirectionalIterator, class OutputIterator>
  OutputIterator
    reverse_copy(ExecutionPolicy &&exec,
                 BidirectionalIterator first,
                 BidirectionalIterator last, OutputIterator result);

template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator rotate(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator middle,
                         ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, class OutputIterator>
  OutputIterator
    rotate_copy(ExecutionPolicy &&exec,
                ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result);
```

```
// partitions:
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy,
         class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ExecutionPolicy &&exec,
              ForwardIterator first,
              ForwardIterator last, Predicate pred);
template<class ExecutionPolicy,
         class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(ExecutionPolicy &&exec,
                     BidirectionalIterator first,
                     BidirectionalIterator last, Predicate pred);
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2>
partition_copy(ExecutionPolicy &&exec,
               InputIterator first, InputIterator last,
               OutputIterator1 out_true, OutputIterator2 out_false,
               Predicate pred);
template<class ExecutionPolicy,
         class ForwardIterator, class Predicate>
  ForwardIterator partition_point(ExecutionPolicy &&exec,
                                  ForwardIterator first,
                                  ForwardIterator last,
                                  Predicate pred);

// sorting and related operations:
// sorting:
template<class ExecutionPolicy,
         class RandomAccessIterator>
  void sort(ExecutionPolicy &&exec,
            RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy &&exec,
            RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<class ExecutionPolicy,
         class RandomAccessIterator>
  void stable_sort(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

```
template<class ExecutionPolicy,
         class RandomAccessIterator>
  void partial_sort(ExecutionPolicy &&exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);
template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy &&exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
template<class ExecutionPolicy,
         class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy,
         class InputIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);

template<class ExecutionPolicy,
         class ForwardIterator>
  bool is_sorted(ExecutionPolicy &&exec,
                 ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy &&exec,
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator is_sorted_until(ExecutionPolicy &&exec,
                                  ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  ForwardIterator is_sorted_until(ExecutionPolicy &&exec,
                                  ForwardIterator first, ForwardIterator last,
                                  Compare comp);

template<class ExecutionPolicy,
         class RandomAccessIterator>
  void nth_element(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);
```

```
template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);

// merge:
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    merge(ExecutionPolicy &&exec,
          InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    merge(ExecutionPolicy &&exec,
          InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);

template<class ExecutionPolicy,
         class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy &&exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);
template<class ExecutionPolicy,
         class BidirectionalIterator,
         class Compare>
  void inplace_merge(ExecutionPolicy &&exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);

// set operations:
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  bool includes(ExecutionPolicy &&exec,
                InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class Compare>
  bool includes(ExecutionPolicy &&exec,
                InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                Compare comp);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
```

```
  OutputIterator
    set_union(ExecutionPolicy &&exec,
              InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_union(ExecutionPolicy &&exec,
              InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_intersection(ExecutionPolicy &&exec,
                     InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_intersection(ExecutionPolicy &&exec,
                     InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_difference(ExecutionPolicy &&exec,
                   InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_difference(ExecutionPolicy &&exec,
                   InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_symmetric_difference(ExecutionPolicy &&exec,
                             InputIterator1 first1, InputIterator1 last1,
```

```
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result);
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_symmetric_difference(ExecutionPolicy &&exec,
                            InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            OutputIterator result, Compare comp);

// minimum and maximum:
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last);
                              Compare comp);
template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last,
template<class ExecutionPolicy,
         class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy &&exec,
                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy &&exec,
                   ForwardIterator first, ForwardIterator last, Compare comp);
                               Compare comp);


template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  bool
    lexicographical_compare(ExecutionPolicy &&exec,
                            InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class Compare>
  bool
    lexicographical_compare(ExecutionPolicy &&exec,
```

```
                              InputIterator1 first1, InputIterator1 last1,
                              InputIterator2 first2, InputIterator2 last2,
                              Compare comp);
}
}
}
```

4.3.2 All of [alg.gen.all_of]

```
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  bool all_of(ExecutionPolicy &&exec,
              InputIterator first, InputIterator last, Predicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `true` if `[first,last)` is empty or `pred(*i)` is true for every iterator `i` in the range `[first,last)` and `false` otherwise.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.3 Any of [alg.gen.any_of]

```
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  bool any_of(ExecutionPolicy &&exec,
              InputIterator first, InputIterator last, Predicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `false` if `[first,last)` is empty or if there is no iterator `i` in the range `[first,last)` such that `pred(*i)` is `true`, and `true` otherwise.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.4 None of [alg.gen.none_of]

```
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  bool none_of(ExecutionPolicy &&exec,
               InputIterator first, InputIterator last, Predicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `true` if `[first,last)`is empty or if `pred(*i)` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.5 For each [alg.gen.foreach]

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

1. *Requires:* `Function` shall meet the requirements of `MoveConstructible` [*Note:* `Function need not meet the requirements of`CopyConstructible‘. – *end note*]
2. *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 throughthe dereferenced iterator. – *end note*]
3. *Returns:* `std::move(f)`.
4. *Complexity:* Applies `f` exactly `n` times.
5. *Remarks:* If `f` returns a result, the result is ignored.

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

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

1. *Effects:* The first algorithm applies `f` to the result of dereferencing every iterator in the range `[first,last)`. The second algorithm applies `f` to the result of dereferencing every iterator in the range `[first,first+n)`. The execution of the algorithm is parallelized as determined by `exec`. [*Note:* If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply nonconstant functions through the dereferenced iterator. – *end note*]
2. *Returns:* `for_each` returns `last` and `for_each_n` returns `first + n` for non-negative values of `n` and `first` for negative values.
3. *Complexity:* `O(last - first)` or `O(n)`.
4. *Remarks:* If `f` returns a result, the result is ignored.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.6 Find [alg.gen.find]

```
template<class ExecutionPolicy,
         class InputIterator, class T>
  InputIterator find(ExecutionPolicy &&exec,
                     InputIterator first, InputIterator last,
                     const T& value);

template<class ExecutionPolicy,
```

```
        class InputIterator, class Predicate>
  InputIterator find_if(ExecutionPolicy &&exec,
                        InputIterator first, InputIterator last,
                        Predicate pred);

template<class ExecutionPolicy,
        class InputIterator, class Predicate>
  InputIterator find_if_not(ExecutionPolicy &&exec,
                            InputIterator first, InputIterator last,
                            Predicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding expression holds: `*i == value`, `pred(*i) != false`, `pred(*i) == false`. Returns `last` if no such iterator is found.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.7 Find end [alg.gen.find.end]

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1
    find_end(ExecutionPolicy &exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1
    find_end(ExecutionPolicy &&exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
```

1. *Effects: The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for any non-negative integer `n < (last2 - first2)`, the following corresponding conditions hold: `*(i + n) == *(first2 + n)`, `pred(*(i + n), *(first2 + n)) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.
3. *Requires:* Neither `operator==` nor `pred` shall invalidate iterators or subranges, nor modify elements in the ranges `[first1,last1)1 or`[first2,last2)‘.
4. *Complexity:* `O(m * n)`, where `m == last2 - first1` and `n = last1 - first1 - (last2 - first2)`.
5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.8 Find first [alg.gen.find.first.of]

```
template<class ExecutionPolicy,
         class InputIterator, class ForwardIterator>
  InputIterator
    find_first_of(ExecutionPolicy &&exec,
                  InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);

template<class ExecutionPolicy,
         class InputIterator, class ForwardIterator,
         class BinaryPredicate>
  InputIterator
    find_first_of(ExecutionPolicy &&exec,
                  InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2,
                  BinaryPredicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following conditions hold: `*i == *j`, `pred(*i,*j) != false`. Returns `last1` if `[first2,last2)` is empty or if no such iterator is found.
3. *Requires:* Neither `operator==` nor `pred` shall invalidate iterators or subranges, nor modify elements in the ranges `[first1,last1)` or `[first2,last2)`.
4. *Complexity*: `O(m * n)`, where `m == last1 - first1` and `n == last2 - first2`.
5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.9 Adjacent find [alg.gen.adjacent.find]

```
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator adjacent_find(ExecutionPolicy &&exec, ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy,
         class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find(ExecutionPolicy &&exec, ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i + 1)`, `pred(*i, *(i + 1)) != false`. Returns `last` if no such iterator is found.
3. *Requires:* Neither `operator==` nor `pred` shall invalidate iterators or subranges, nor modify elements in the range `[first,last)`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.10 Count [alg.gen.count]

```
template<class ExecutionPolicy,
         class InputIterator, class EqualityComparable>
  typename iterator_traits<InputIterator>::difference_type
    count(ExecutionPolicy &&exec,
          InputIterator first, InputIterator last, const EqualityComparable &value);

template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  typename iterator_traits<InputIterator>::difference_type
    count_if(ExecutionPolicy &&exec,
             InputIterator first, InputIterator last, Predicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.11 Mismatch [alg.gen.mismatch]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  pair<InputIterator1,InputIterator2>
    mismatch(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class BinaryPredicate>
  pair<InputIterator1,InputIterator2>
    mismatch(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* A pair of iterators `i` and `j` such that `j == first2 + (i - first)` and `i` is the first iterator in the range `[first1,last1)` for which the following corresponding conditions hold:

```
!(i == *(first2 + (i - first1)))
pred(*i, *(first2 + (i - first1))) == false
```

Returns the pair `last1` and `first2 + (last1 - first1)` if such an iterator `i` is not found.

3. *Complexity:* `O(last1 - first1)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.12 Equal [alg.gen.equal]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  bool equal(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class BinaryPredicate>
  bool equal(ExecutionPolicy &&exec,
             InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
```

1. *Effects*: The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `true` if for every iterator `i` in the range `[first1,last1)` the following corresponding conditions hold: `*i == *(first2 + (i - first1))`, `pred(*i, *(first2 + (i - first1))) != false`. Otherwise, returns `false`.
3. *Complexity:* `O(last1 - first1)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.13 Search [alg.gen.search]

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search(ExecutionPolicy &&exec,
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  ForwardIterator1 search(ExecutionPolicy &&exec,
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          BinaryPredicate pred);
```

1. *Effects*: Finds a subsequence of equal values in a sequence.

   The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2)` such that for any non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n)`, `pred(*(i + n), *(first2 + n)) != false`. Returns `first1` if `[first2,last2)` is empty, otherwise returns `last1` if no such iterator is found.
3. *Complexity:* `O((last1 - first1) * (last2 - first2))`.

```
template<class ExecutionPolicy,
         class ForwardIterator, class Size, class T>
  ForwardIterator search_n(ExecutionPolicy &&exec,
                           ForwardIterator first, ForwardIterator last, Size count,
```

```
                          const T& value);

template<class ExecutionPolicy,
         class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  ForwardIterator search_n(ExecutionPolicy &&exec,
                          ForwardIterator first, ForwardIterator last, Size count,
                          const T& value, BinaryPredicate pred);
```

1. *Requires:* The type `Size` shall be convertible to integral type.
2. *Effects:* Finds a subsequence of equal values in a sequence.

   The algorithm's execution is parallelized as determined by `exec`.
3. *Returns:* The first iterator `i` in the range `[first,last-count)` such that for any non-negative integer `n` less than `count` the following corresponding conditions hold: `*(i + n) == value, pred(*(i + n),value) != false`. Returns `last` if no such iterator is found.
4. *Complexity:* `O(last - first)`.

4.3.14 Copy [alg.gen.copy]

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

1. *Effects:* For each iterator `i` in the range `[first,last)`, performs `*(result + (i - first) = *i`. The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `result + (last - first)`.
3. *Requires:* `result` shall not be in the range `[first,last)`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class InputIterator, class Size, class OutputIterator>
  OutputIterator copy_n(ExecutionPolicy &&exec,
                        InputIterator first, Size n,
                        OutputIterator result);
```

1. *Effects:* For each non-negative integer `i < n`, performs `*(result + i) = *(first + i)`. The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `result + n`.
3. *Complexity:* `O(n)`.
4. *Remarks:* The signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    copy_if(ExecutionPolicy &&exec,
            InputIterator first, InputIterator last,
            OutputIterator result, Predicate pred);
```

1. *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.
2. *Effects:* Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `pred(*i)` is `true`. The algorithm's execution is parallelized as determined by `exec`.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* Stable.

   The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.15 Move [alg.gen.move]

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

1. *Effects:* For each iterator `i` in the range `[first,last)`, performs `*(result + (i - first) = std::move(*i)`. The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `result - (last - first)`.
3. *Requires:* `result` shall not be in the range `[first,last)`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.16 Swap [alg.gen.swap]

```
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2
    swap_ranges(ExecutionPolicy &&exec,
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator1 first2);
```

1. *Effects:* For each non-negative integer `n < (last1 - first1)` performs: `swap(*(first1 + n), *(first2 + n))`. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* The two ranges `[first1,last1)` and `[first2,first2 + (last1 - first1))` shall not overlap. `*(first1 + n)` shall be swappable with `*(first2 + n)`.

3. *Returns:* `first2 + (last1 - first1)`.

4. *Complexity:* `O(last1 - first1)`.

5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.17 Transform [alg.gen.transform]

```
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator,
         class UnaryOperation>
  OutputIterator transform(ExecutionPolicy &&exec,
                           InputIterator first, InputIterator last,
                           OutputIterator result, UnaryOperation op);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
  OutputIterator
    transform(ExecutionPolicy &&exec,
              InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
```

1. *Effects:* Assigns through every iterator `i` in the range `[result,result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result))` or `binary_op(*(first1 + (i - result)), *(first2 + (i - result))`. The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,last1]`, `[first2,first2 + (last1 - first1)]`, and `[result,result + (last1 - first1)]`.

3. *Returns:* `result + (last1 - first1)`.

4. *Complexity:* `O(last - first)` or `O(last1 - first1)`.

5. *Remarks:* `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.18 Replace [alg.gen.replace]

```
template<class ExecutionPolicy,
         class ForwardIterator, class T>
  void replace(ExecutionPolicy &&exec,
               ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);

template<class ExecutionPolicy,
         class ForwardIterator, class Predicate, class T>
  void replace_if(ExecutionPolicy &&exec,
```

```
                ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

1. *Requires:* The expression `*first = new_value` shall be valid.
2. *Effects:* Substitutes elements referred by the iterator `i` in the range `[first,last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`. The algorithm's execution is parallelized as determined by `exec`.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class T>
  OutputIterator
    replace_copy(ExecutionPolicy &&exec,
                 ForwardIterator first, ForwardIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);

template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class Predicate, class T>
  OutputIterator
    replace_copy_if(ExecutionPolicy &&exec,
                    InputIterator first, InputIterator last,
                    OutputIterator result,
                    Predicate pred, const T& new_value);
```

1. *Requires:* The results of the expressions `*first` and `new_value` shall be writable to the `result` output iterator. The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.
2. *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

   ```
   *(first + (i - result)) == old_value
   pred(*(first + (i - result))) != false
   ```

   The algorithm's execution is parallelized as determined by `exec`.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.19 Fill [alg.gen.fill]

```
template<class ExecutionPolicy,
         class ForwardIterator, class T>
  void fill(ExecutionPolicy &&exec,
            ForwardIterator first, ForwardIterator last, const T& value);

template<class ExecutionPolicy,
```

```
        class OutputIterator, class Size, class T>
  void fill_n(ExecutionPolicy &&exec,
              OutputIterator first, Size n, const T& value);
```

1. *Requires:* The expression `value` shall be writable to the output iterator. The type `Size` shall be convertible to an integral type.
2. *Effects:* The first algorithm assigns `value` through all the iterators in the range `[first,last)`. The second value assigns `value` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing. The algorithm is parallelized as determined by `exec`.
3. *Returns:* `fill_n` returns `first + n` for non-negative values of `n` and `first` for negative values.
4. *Complexity:* `O(last - first)` or `O(n)`.
5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.20 Generate [alg.gen.generate]

```
template<class ExecutionPolicy,
         class ForwardIterator, class Generator>
  void generate(ExecutionPolicy &&exec,
                ForwardIterator first, ForwardIterator last, Generator gen);

template<class ExecutionPolicy,
         class OutputIterator, class Size, class Generator>
  OutputIterator generate_n(ExecutionPolicy &&exec,
                            OutputIterator first, Size n, Generator gen);
```

1. *Effects:* The first algorithm invokes the function object `gen` and assigns the value of `gen` through all the iterators in the range `[first,last)`. The second algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing. The algorithms execution is parallelized as determined by `exec`.
2. *Requires:* `gen` takes no arguments, `Size` shall be convertible to an integral type.
3. *Returns:* `generate_n` returns `first + n` for non-negative values of `n`and `first` for negative values.
4. *Complexity:* `O(last - first)` or `O(n)`.
5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.21 Remove [alg.gen.remove]

```
template<class ExecutionPolicy,
         class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator last, const T& value);

template<class ExecutionPolicy,
         class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy &&exec,
                            ForwardIterator first, ForwardIterator last, Predicate pred);
```

1. *Requires:* The type of `*first` shall satisfy the `MoveAssignable` requirements.
2. *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`. The algorithm's execution is parallelized as determined by `exec`.
3. *Returns:* The end of the resulting range.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* Stable.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.
6. *Note:* Each element in the range `[ret,last)`, where `ret` is the returned value, has a valid but unspecified state, because the algorithms can eliminate elements by swapping with or moving from elements that were originally in that range.

```
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class T>
  OutputIterator
    remove_copy(ExecutionPolicy &&exec,
                InputIterator first, InputIterator last,
                OutputIterator result, const T& value);

template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class Predicate>
  OutputIterator
    remove_copy_if(ExecutionPolicy &&exec,
                   InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
```

1. *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap. The expression `*result = *first` shall be valid.
2. *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `*i == value`, `pred(*i) != false`. The algorithm's execution is parallelized as determined by `exec`.
3. *Returns:* The end of the resulting range.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* Stable.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.22 Unique [alg.gen.unique]

```
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator unique(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy,
```

```
        class ForwardIterator, typename BinaryPredicate>
  ForwardIterator unique(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator last
                         BinaryPredicate pred);
```

1. *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `*(i - 1) == *i` or `pred(*(i - 1), *i) != false`. The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* The comparison function shall be an equivalence relation. The type of `*first` shall satisfy the `MoveAssignable` requirements.

3. *Returns:* The end of the resulting range.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

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

template<class ExecutionPolicy,
         class InputIterator, class OutputIterator, class BinaryPredicate>
  OutputIterator
    unique_copy(ExecutionPolicy &&exec,
                InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
```

1. *Requires:* The comparison function shall be an equivalence relation. The ranges `[first,last)` and `[result,result + (last-first))` shall not overlap. The expression `*result = *first` shall be valid. If neither `InputIterator` nor `OutputIterator` meets the requirements of forward iterator then the value type of `InputIterator` shall be `CopyConstructible` and `CopyAssignable`. Otherwise `CopyConstructible` is not required.

2. *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`. The algorithm's execution is parallelized as determined by `exec`.

3. *Returns:* The end of the resulting range.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.23 Reverse [alg.gen.reverse]

```
template<class ExecutionPolicy,
         class BidirectionalIterator>
  void reverse(ExecutionPolicy &&exec,
               BidirectionalIterator first, BidirectionalIterator last);
```

1. *Effects:* For each non-negative integer `i <= (last - first)/2`, applies `iter_swap` to all pairs of iterator `first + i`, `(last - i) - 1`. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* `*first` shall be swappable.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class BidirectionalIterator, class OutputIterator>
  OutputIterator
    reverse_copy(ExecutionPolicy &&exec,
                 BidirectionalIterator first,
                 BidirectionalIterator last, OutputIterator result);
```

1. *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for any non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - i) = *(first + i)`. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.
3. *Returns*: `result + (last - first)`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.24 Rotate [alg.gen.rotate]

```
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator rotate(ExecutionPolicy &&exec,
                         ForwardIterator first, ForwardIterator middle,
                         ForwardIterator last);
```

1. *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`. The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* `first + (last - middle)`.
3. *Remarks:* This is a left rotate.

   The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4. *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges. `ForwardIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirements of `MoveConstructible` and the requirements of `MoveAssignable`.

5. *Complexity:* `O(last - first)`.

```
template<class ExecutionPolicy,
         class ForwardIterator, class OutputIterator>
  OutputIterator
    rotate_copy(ExecutionPolicy &&exec,
                ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result);
```

1. *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`. `The algorithm's execution is parallelized as determined by`exec‘.

2. *Returns:* `result + (last - first)`.

3. *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.25 Partitions [alg.gen.partitions]

```
template<class ExecutionPolicy,
         class InputIterator, class Predicate>
  bool is_partitioned(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last, Predicate pred);
```

1. *Requires:* `InputIterator`'s value type shall be convertible to `Predicate`'s argument type.

2. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

3. *Returns:* `true` if `[first,last)` is empty or if `[first,last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that do not.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class ForwardIterator, class Predicate>
  ForwardIterator
    partition(ExecutionPolicy &&exec,
              ForwardIterator first,
              ForwardIterator last, Predicate pred);
```

1. *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it. The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* An iterator `i` such that for any iterator `j` in the range `[first,i)`, `pred(*j) != false`, and for any iterator `k` in the range `[i,last)`, `pred(*k) == false`.

3. *Requires:* `ForwardIterator` shall satisfy the requirements of `ValueSwappable`.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class BidirectionalIterator, class Predicate>
  BidirectionalIterator
    stable_partition(ExecutionPolicy &&exec,
                     BidirectionalIterator first,
                     BidirectionalIterator last, Predicate pred);
```

1. *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it. The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* An iterator `i` such that for any iterator `j` in the range `[first,i)`, `pred(*j) != false`, and for any iterator `k` in the range `[i,last)`, `pred(*k) == false`. The relative order of the elements in both groups is preserved.

3. *Requires:* `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirements of `MoveConstructible` and of `MoveAssignable`.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class InputIterator, class OutputIterator1,
         class OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2>
partition_copy(ExecutionPolicy &&exec,
               InputIterator first, InputIterator last,
               OutputIterator1 out_true, OutputIterator2 out_false,
               Predicate pred);
```

1. *Requires:* `InputIterator`'s value type shall be `Assignable`, and shall be writable to the `out_true` and `out_false` `OutputIterator`s, and shall be convertible to `Predicate`'s argument type. The input range shall not overlap with either of the output ranges.

2. *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `pred(*i)` is `true`, or to the output range beginning with `out_false` otherwise. The algorithm's execution is parallelized as determined by `exec`.

3. *Returns:* A pair `p` such that `p.first` is the end of the output range beginning at `out_true` and `p.second` is the end of the output range beginning at `out_false`.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class ForwardIterator, class Predicate>
  ForwardIterator partition_point(ExecutionPolicy &&exec,
                                  ForwardIterator first,
                                  ForwardIterator last,
                                  Predicate pred);
```

1. *Requires:* `ForwardIterator`'s value type shall be convertible to `Predicate`'s argument type. `[first,last)` shall be partitioned by `pred`, i.e. all elements that satisfy `pred` shall appear before those that do not.
2. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
3. *Returns:* An iterator `mid` such that `all_of(first, mid, pred)` and `none_of(mid, last, pred)` are both `true`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.26 `sort` [alg.gen.sort]

```
template<class ExecutionPolicy,
         class RandomAccessIterator>
  void sort(ExecutionPolicy &&exec,
            RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void sort(ExecutionPolicy &&exec,
            RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

1. *Effects:* Sorts the elements in the range `[first,last)`. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirements of `MoveConstructible` and of `MoveAssignable`.
3. *Complexity:* `O(n lg n)`, where `n = last - first`.
4. *Remarks:* The signature shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.27 `stable_sort` [alg.gen.stable.sort]

```
template<class ExecutionPolicy,
         class RandomAccessIterator>
  void stable_sort(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator last,
                   Compare comp);
```

1. *Effects:* Sorts the elements in the range `[first,last)`. The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirements of `MoveConstructible` and of `MoveAssignable`.

3. *Complexity:* `O(n lg n)`, where `n = last - first`.

4. *Remarks:* Stable.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.28 `partial_sort` [alg.gen.partial.sort]

```
template<class ExecutionPolicy,
         class RandomAccessIterator>
  void partial_sort(ExecutionPolicy &&exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void partial_sort(ExecutionPolicy &&exec,
                    RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                    Compare comp);
```

1. *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

   The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirements of `MoveConstructible` and of `MoveAssignable`.

   `middle` shall be in the range `[first,last)`.

3. *Complexity:* `O(m lg n)`, where `m = last - first` and `n = middle - first`.

4. *Remarks:* The signatures shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.29 `partial_sort_copy` [alg.gen.partial.sort.copy]

```
template<class ExecutionPolicy,
         class InputIterator, class RandomAccessIterator>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
```

```
template<class ExecutionPolicy,
         class InputIterator, class RandomAccessIterator,
         class Compare>
  RandomAccessIterator
    partial_sort_copy(ExecutionPolicy &&exec,
                      InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
```

1. *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

   The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

3. *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*result_first` shall satisfy the requirements of `MoveConstructible` and of `MoveAssignable`.

4. *Complexity:* `O(m lg n)`, where `m = last - first` and `n = min(last - first, result_last - result_first)`.

5. *Remarks:* The signatures shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.30 `is_sorted` [alg.gen.is.sorted]

```
template<class ExecutionPolicy,
         class ForwardIterator>
  bool is_sorted(ExecutionPolicy &&exec,
                 ForwardIterator first, ForwardIterator last);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* `is_sorted_until(forward<ExecutionPolicy>(exec), first, last) == last`

3. *Complexity:* `O(last - first)`.

4. *Remarks:* The signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  bool is_sorted(ExecutionPolicy &&exec,
                 ForwardIterator first, ForwardIterator last,
                 Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* `is_sorted_until(forward<ExecutionPolicy>(exec), first, last, comp) == last`

3. *Complexity:* `O(last - first)`.

4. *Remarks:* The signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator is_sorted_until(ExecutionPolicy &&exec,
                                  ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  ForwardIterator is_sorted_until(ExecutionPolicy &&exec,
                                  ForwardIterator first, ForwardIterator last,
                                  Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* If `distance(first, last) < 2)`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last)` for which the range `[first,i)` is sorted.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.31 Nth element [alg.gen.nth.element]

```
template<class ExecutionPolicy,
         class RandomAccessIterator>
  void nth_element(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last);

template<class ExecutionPolicy,
         class RandomAccessIterator, class Compare>
  void nth_element(ExecutionPolicy &&exec,
                   RandomAccessIterator first, RandomAccessIterator nth,
                   RandomAccessIterator last, Compare comp);
```

1. *Effects:* Reorders the range `[first,last)` such that the element referenced by `nth` is the element that would be in that position if the whole range were sorted. Also for any iterator `i` in the range `[first,nth)` and any iterator `j` in the range `[nth,last)` the following corresponding condition holds: `!(*j < *i)` or `comp(*j, *i) == false`.

   The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirements of `MoveConstructible` and of `MoveAssignable`.

   `nth` shall be in the range `[first,last)`.
3. *Complexity:* `O(last - first)`.
4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.32 Merge [alg.gen.merge]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    merge(ExecutionPolicy &&exec,
          InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    merge(ExecutionPolicy &&exec,
          InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
```

1. *Effects:* Copies all the elements of the two ranges `[first1,last1)` and `[first2,last2)` into the range `[result,result_last)`, where `result_last` is `result + (last1 - first1) + (last2 - first2)`, such that the resulting range satisfies `is_sorted(result, result_last)` or `is_sorted(result, result_last, comp)`, respectively. The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `operator<` or `comp`. The resulting range shall not overlap with either of the input ranges.

3. *Returns:* `result + (last1 - first1) + (last2 - first2)`.

4. *Complexity:* `O(m + n)`, where `m = last1 - first1` and `n = last2 - first2`.

5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class BidirectionalIterator>
  void inplace_merge(ExecutionPolicy &&exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template<class ExecutionPolicy,
         class BidirectionalIterator,
         class Compare>
  void inplace_merge(ExecutionPolicy &&exec,
                     BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last, Compare comp);
```

1. *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))` will be `false`. The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* The ranges `[first,middle)` and `[middle,last)` shall be sorted with respect to `operator<` or `comp`. `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable`. The type of `*first` shall satisfy the requirementsof `MoveConstructible`and of `MoveAssignable`.

3. *Remarks:* Stable.

4. *Complexity:* `O(m + n)`, where `m = middle - first` and `n = last - middle`.

5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.33 Includes [alg.gen.includes]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  bool includes(ExecutionPolicy &&exec,
                InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class Compare>
  bool includes(ExecutionPolicy &&exec,
                InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
                Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* The ranges `[first1,last1)` and `[first2,last2)` shall be sorted with respect to `operator<` or `comp`.

3. *Returns:* `true` if `[first2,last2)` is empty or if every element in the range `[first2,last2)` is contained in the range `[first1,last1)`. Returns `false` otherwise.

4. *Complexity:* `O(m + n)`, where `m = last1 - first1` and `n = last2 - first2`.

5. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.34 `set_union` [alg.gen.set.union]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_union(ExecutionPolicy &&exec,
              InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
```

```
set_union(ExecutionPolicy &&exec,
          InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
```

1. *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* The resulting range shall not overlap with either of the original ranges.
3. *Returns:* The end of the constructed range.
4. *Complexity:* `O(m + n)`, where `m = last1 - first1` and `n = last2 - first2`.
5. *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2,last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.35 `set_intersection` [alg.gen.set.intersection]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_intersection(ExecutionPolicy &&exec,
                     InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_intersection(ExecutionPolicy &&exec,
                     InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
```

1. *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* The resulting range shall not overlap with either of the original ranges.
3. *Returns:* The end of the constructed range.
4. *Complexity:* `O(m + n)`, where `m = last1 - first1` and `n = last2 - first2`.
5. *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2,last2)` contains n elements that are equivalent to them, the first $min(m, n)$ elements shall be copied from the first range to the output range, in order.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.36 `set_difference` [alg.gen.set.difference]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_difference(ExecutionPolicy &&exec,
                   InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_difference(ExecutionPolicy &&exec,
                   InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
```

1. *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2,last2)` to the range beginning at `result`. The elements in the constructed range are sorted. The algorithm's execution is parallelized as determined by `exec`.
2. *Requires:* The resulting range shall not overlap with either of the original ranges.
3. *Returns:* The end of the constructed range.
4. *Complexity:* `O(m + n)`, where `m = last1 - first1` and `n = last2 - first2`.
5. *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2,last2)` contains n elements that are equivalent to them, the last $max(m-n,0)$ elements from `[first1,last1)` shall be copied to the output range.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.37 `set_symmetric_difference` [alg.gen.set.symmetric.difference]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator>
  OutputIterator
    set_symmetric_difference(ExecutionPolicy &&exec,
                             InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator
    set_symmetric_difference(ExecutionPolicy &&exec,
                             InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
```

1. *Effects:* Copies the elements of the range `[first1,last1)` that are not present in the range `[first2,last2)`, and the elements of the range `[first2,last2)` that are not present in the range `[first1,last1)` to the range beginning at `result`. The elements in the constructed range are sorted. The algorithm's execution is parallelized as determined by `exec`.

2. *Requires:* The resulting range shall not overlap with either of the original ranges.

3. *Returns:* The end of the constructed range.

4. *Complexity:* `O(m + n)`, where `m = last1 - first1` and `n = last2 - first2`.

5. *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2,last2)` contains n elements that are equivalent to them, then $|m - n|$ of those elements shall be copied to the output range: the last $m - n$ of these elements from `[first1,last1)` if $m > n$, and the last $n - m$ of these elements from `[first2,last2)` if $m < n$.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.38 Minimum and maximum [alg.gen.min.max]

```
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator min_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  ForwardIterator min_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* The first iterator `i` in the range `[first,last)` such that for any iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`. Returns `last` if `first == last`.

3. *Complexity:* `O(last - first)`.

4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class ForwardIterator>
  ForwardIterator max_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  ForwardIterator max_element(ExecutionPolicy &&exec,
                              ForwardIterator first, ForwardIterator last,
                              Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* The first iterator `i` in the range `[first,last)` such that for any iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`. Returns `last` if `first == last`.

3. *Complexity:* `O(last - first)`.

4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class ForwardIterator>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy &&exec,
                   ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy,
         class ForwardIterator, class Compare>
  pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy &&exec,
                   ForwardIterator first, ForwardIterator last, Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* `make_pair(first, first)` if `[first,last)` is empty, otherwise `make_pair(m, M)`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and where `M` is the last iterator in `[first,last)` such that no iterator in the range refers to a larger element.

3. *Complexity:* `O(last - first)`.

4. *Remarks:* The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.3.39 Lexicographical comparison [alg.gen.lex.comparison]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2>
  bool
    lexicographical_compare(ExecutionPolicy &&exec,
                            InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class Compare>
  bool
    lexicographical_compare(ExecutionPolicy &&exec,
                            InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)` and `false` otherwise.

3. *Complexity:* `O(min(m,n))`, where `m = last1 - first1` and `n = last2 - first2`.

4. *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

   An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.4 Generalized numeric operations [alg.numerics]

This subclause describes components that C++ programs may use to perform seminumerical operations in parallel.

4.4.1 Header <experimental/numeric> synopsis [alg.numerics.synop]

```
namespace std {
namespace experimental {
namespace parallel {
  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 ExecutionPolicy,
           class InputIterator1, class InputIterator2, class T>
    T inner_product(ExecutionPolicy &&exec,
                    InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, T init);
  template<class ExecutionPolicy,
           class InputIterator1, class InputIterator2, class T,
```

```
          class BinaryOperation1, class BinaryOperation2>
  T inner_product(ExecutionPolicy &&exec,
                  InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);

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

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

4.4.2 Reduce [alg.numerics.reduce]

```
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);
```

1. *Effects:* The second algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The result of the sum of `T(0)` and the elements in the range `[first,last)`.

   The order of operands of the sum is unspecified.
3. *Requires:* Let `T` be the type of `InputIterator`'s value type, then `T(0)` shall be a valid expression. The `operator+` function associated with `T` shall have associativity and commutativity.

   `operator+` shall not invalidate iterators or subranges, nor modify elements in the range `[first,last)`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The second signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
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);
```

1. *Effects:* The execution of the second and fourth algorithms is parallelized as determined by `exec`.
2. *Returns:* The result of the generalized sum of `init` and the elements in the range `[first,last)`.

   Sums of elements are evaluated with `operator+` or `binary_op`. The order of operands of the sum is unspecified.
3. *Requires:* The `operator+` function associated with `InputIterator`'s value type or `binary_op` shall have associativity and commutativity.

   Neither `operator+` nor `binary_op` shall invalidate iterators or subranges, nor modify elements in the range `[first,last)`.
4. *Complexity:* `O(last - first)`.
5. *Remarks:* The second and fourth signatures shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.4.3 Inner product [alg.numerics.inner.product]

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class T>
  T inner_product(ExecutionPolicy &&exec,
                  InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* The result of the sum `init + (*(first1 + i) * *(first2 + i) + ...` for every integer `i` in the range `[0, (last1 - first1))`.

   The order of operands of the sum is unspecified.

3. *Requires:* `operator+` shall have associativity and commutativity.

   `operator+` shall not invalidate iterators or subranges, nor modify elements in the ranges `[first1,last1)` or `[first2,first2 + (last1 - first1))`.

4. *Complexity:* `O(last1 - first1)`.

5. *Remarks:* The signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

```
template<class ExecutionPolicy,
         class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  T inner_product(ExecutionPolicy &&exec,
                  InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* The result of the generalized sum whose operands are `init` and the result of the pairwise binary operation `binary_op2(*i,*(first2 + (i - first1)))` for all iterators `i` in the range `[first1,last1)`.

   The generalized sum's operands are combined via application of the pairwise binary operation `binary_op1`.

   The order of operands of the sum is unspecified.

3. *Requires:* `binary_op1` shall have associativity and commutativity.

   `binary_op1` and `binary_op2` shall neither invalidate iterators or subranges, nor modify elements in the ranges `[first1,last1)` or `[first2,first2 + (last1 - first1))`.

4. *Complexity:* `O(last1 - first1)`.

5. *Remarks:* The signature shall not participate in overload resolution if

   `is_execution_policy<ExecutionPolicy>::value` is `false`.

4.4.4 Exclusive scan [alg.numerics.exclusive.scan]

```
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);
```

1. *Effects:* For each iterator `i` in `[result,result + (last - first))`, produces a result such that upon completion of the algorithm, `*i` yields the generalized sum of `init` and the elements in the range `[first, first + (i - result))`.

   During execution of the algorithm, every evaluation of the above sum is either of the corresponding form

   `(init + A) + B)` or `A + B` or

   `binary_op(binary_op(init,A), B)` or `binary_op(A, B)`

   where there exists some iterator `j` in `[first, last)` such that:

   1. `A` is the partial sum of elements in the range `[j, j + n)`.
   2. `B` is the partial sum of elements in the range `[j + n, j + m)`.
   3. `n` and `m` are positive integers and `j + m < last`.

   The execution of the second and fourth algorithms is parallelized as determined by `exec`.

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

3. *Requires:* The `operator+` function associated with `InputIterator`'s value type or `binary_op` shall have associativity.

Neither `operator+` nor `binary_op` shall invalidate iterators or subranges, nor modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The second and fourth signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

6. *Notes:* The primary difference between `exclusive_scan` and `inclusive_scan` is that `exclusive_scan` excludes the `i`th input element from the `i`th sum.

4.4.5 Inclusive scan [alg.numerics.inclusive.scan]

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

template<class ExecutionPolicy,
         class InputIterator, class OutputIterator,
         class T, class BinaryOperation>
  OutputIterator
    inclusive_scan(ExecutionPolicy &&exec,
                   InputIterator first, InputIterator last,
```

```
                    OutputIterator result,
                    T init, BinaryOperation binary_op);
```

1. *Effects:* For each iterator `i` in `[result,result + (last - first))`, produces a result such that upon completion of the algorithm, `*i` yields the generalized sum of `init` and the elements in the range `[first, first + (i - result)]`.

   During execution of the algorithm, every evaluation of the above sum is either of the corresponding form

   `(init + A) + B)` or `A + B` or

   `binary_op(binary_op(init,A), B)` or `binary_op(A, B)`

   where there exists some iterator `j` in `[first, last)` such that:

   1. `A` is the partial sum of elements in the range `[j, j + n)`.
   2. `B` is the partial sum of elements in the range `[j + n, j + m)`.
   3. `n` and `m` are positive integers and `j + m < last`.

   The execution of the second and fourth algorithms is parallelized as determined by `exec`.

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

3. *Requires:* The `operator+` function associated with `InputIterator`'s value type or `binary_op` shall have associativity.

   Neither `operator+` nor `binary_op` shall invalidate iterators or subranges, nor modify elements in the ranges `[first,last)` or `[result,result + (last - first))`.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* The second, fourth, and sixth signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.

6. *Notes:* The primary difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the `i`th input element in the `i`th sum.

4.4.6 Adjacent difference [alg.numerics.adjacent.difference]

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

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

1. *Effects:* Performs `*result = *first` and for each iterator `i` in the range `[first + 1, last)`, performs `*result = *i - *(i - 1)`, or `*result = binary_op(*i, *(i - 1))`, respectively.

   The algorithm's execution is parallelized as determined by `exec`.

2. *Returns:* `result + (last - first)`.

3. *Requires:* The result of the expression `*i - *(i - 1)` or `binary_op(*i, *(i - 1))` shall be writable to the `result` output iterator.

   Neither `operator-` nor `binary_op` shall invalidate iterators or subranges, nor modify elements in the range `[first,last)` or `[result,result + (last - first))`.

4. *Complexity:* `O(last - first)`.

5. *Remarks:* `result` may be equal to `first`.

   The signatures shall not participate in overload resolution if `is_execution_policy<ExecutionPolicy>::value` is `false`.