A Parallel Algorithms Library | N3554

Jared Hoberock Jaydeep Marathe Michael Garland Olivier Giroux
Vinod Grover {jhoberock, jmarathe, mgarland, ogiroux, vgrover}@nvidia.com
Artur Laksberg Herb Sutter {arturl, hsutter}@microsoft.com Arch Robison
arch.robison@intel.com

2013-03-15

1 Executive Summary

We introduce a library of algorithms with parallel execution semantics. Some of these algorithms have semantics similar to the existing standard library of sequential algorithms. Some of the algorithms we propose are novel.

We introduce three parallel execution policies for parallel algorithm execution: `std::seq`, `std::par`, and `std::vec`, as well as a facility for vendors to provide non-standard execution policies as extensions. These policy objects may be used to specify how a parallel algorithm should be executed:

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

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

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

// parallel sort
std::sort(std::par, vec.begin(), vec.end());

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

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

std::sort(exec, vec.begin(), vec.end());

// parallel sort with non-standard implementation-provided execution policies:
std::sort(vectorize_in_this_thread, vec.begin(), vec.end());
std::sort(submit_to_my_thread_pool, vec.begin(), vec.end());
```

```
std::sort(execute_on_that_gpu, vec.begin(), vec.end());
std::sort(offload_to_my_fpga, vec.begin(), vec.end());
std::sort(send_this_computation_to_the_cloud, vec.begin(), vec.end());
```

Algorithms invoked with `std::seq` execute internally in sequential order in the calling thread.

Algorithms invoked with `std::par` are permitted to execute internally in an unordered fashion in unspecified threads. It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks.

Algorithms invoked with `std::vec` are permitted to execute internally in an unordered fashion in unspecified threads. In addition to the restrictions implied by `std::par`, it is the caller's responsibility to ensure that a `std::vec` invocation does not throw exceptions or attempt to perform synchronization operations.

Algorithms invoked without an execution policy execute as if they were invoked with `std::seq`.

An implementation may provide additional execution policies besides `std::seq`, `std::par`, or `std::vec`.

This proposal is a pure addition to the existing C++ standard library; we do not believe it alters the semantics of any existing functionality.

---

2 Design Notes and Outstanding Questions

Before introducing the detailed specification of our proposal, we begin by outlining our rationale and by noting that we have identified a number of outstanding questions which require further work to resolve.

2.1 General Considerations

This proposal is motivated by a strong desire to provide a standard model of parallelism enabling performance portability across the broadest possible range of parallel architectures. For this reason, we have exposed parallelism in the most abstract manner possible in order to avoid presuming the existence of a particular parallel machine model. In particular, we have intentionally avoided a specification which would be required to introduce concurrency by creating threads. Additionally, our proposal's semantics ensure that a sequential implementation is always a valid realization of an algorithm invocation.

One limitation of STL-like algorithms is that they encourage the programmer to engage in a style of programming which may be an obstacle to achieving large absolute performance. For example, in situations where a sequential programmer might implement a program using a single `for` loop, a parallel programmer might express the same program as a sequence of separate `gather`, `for_each`, and `scatter` phases. This is troublesome because in many cases the performance of most STL algorithms is bounded by the speed of memory bandwidth, and the rate of memory bandwidth scaling on parallel architectures is slowing.

One way to ameliorate such problems is to combine the use of parallel algorithms with "fancy" iterators in the style of the Boost Iterator Library. Iterators such as `transform_iterator` can fuse the effect of `std::transform` into another algorithm call, while a `permutation_iterator` can fuse a scatter or gather. By fusing together several "elemental" operations into a single function consumed by a parallel algorithm, memory bandwidth requirements can be reduced significantly. However, because this idea is orthogonal to the idea of parallel algorithms, this proposal does not include a novel iterator library.

While we are enthusiastic that our library solution to parallelism can offer a portable way to author programs for parallel architectures, we acknowledge that it is only a partial solution for parallelism. We expect our library to coexist in an ecosystem of standard language and library constructs which target parallelism at varying levels of abstraction.

2.2 Naming

Integrating and exposing a new library of parallel algorithms into the existing C++ standard library is an interesting challenge. Because this proposal introduces a large set of novel parallel algorithms with semantics subtly different from their existing sequential counterparts, it is crucial to provide the programmer with the means to safely and unambiguously express her parallelization intent. By the same token, it is important for the library interface to distinguish between the different concurrency guarantees provided by parallel and sequential algorithms. Yet, the interface must be flexible enough such that parallelization does not become burdensome.

The primary means by which the user will interact with the standard parallel algorithms library will be by invoking parallel algorithms by name. Because many parallel algorithm names are already taken by their sequential counterparts (e.g., `sort`), we require a way to disambiguate these invocations.

After considering a variety of alternative designs, we propose to integrate parallelism into the existing standard algorithm names and distinguish parallelism via parallel execution policies with distinct types. As the code sample in the executive summary demonstrates, we feel that this scheme provides deep integration with the existing standard library.

2.2.1 Rejected Alternative Naming Schemes

Here, we discuss the rationale for rejecting alternative schemes we considered.

A `parallel_` Name Prefix A simple way to disambiguate parallel algorithms from their sequential versions would be simply to give them new, unique names. Indeed, this is the approach suggested by Intel & Microsoft's earlier paper N3429 and is the one taken in their libraries (i.e. Threading Building Blocks & Parallel Patterns Library, respectively). It is impossible for a human reader or implementation to confuse a call to `std::for_each` for `std::parallel_for_each` and vice versa.

While such an approach could be standardized safely, it is unclear that this scheme is scalable from the six or so algorithms provided by TBB & PPL to the large set of algorithms we propose to parallelize. The following two code examples demonstrate issues we anticipate.

By requiring the programmer to choose between two static names, it seems impossible to allow the dynamic selection between the sequential or parallel version of an algorithm without imposing an unnecessary burden on the programmer:

```
std::vector sort_me = ...
size_t parallelization_threshold = ...

if(sort_me.size() > parallelization_threshold)
{
  std::parallel_sort(sort_me.begin(), sort_me.end());
}
else
{
  std::sort(sort_me.begin(), sort_me.end());
}
```

It is likely that this idiom would become a repetitive burden prone to mistakes.

A similar problem exists for static decisions. Consider a function template which wishes to make a static decision regarding parallelism:

```
template<bool parallelize>
void func(std::vector &vec)
{
  if(parallelize)
  {
    std::parallel_transform(vec.begin(), vec.end(), vec.begin(), f);
    std::parallel_sort(vec.begin(), vec.end());
    std::parallel_unique(vec.begin(), vec.end());
  }
  else
  {
    std::transform(vec.begin(), vec.end(), vec.begin(), f);
    std::sort(vec.begin(), vec.end());
    std::unique(vec.begin(), vec.end());
  }
}
```

This idiom requires the programmer to repeat the function body twice even though the semantics of both implementations are largely identical.

Finally, such a scheme also seems unnecessarily verbose: a sophisticated program composed of repeated calls to a large variety of parallel algorithms would become a noisy repetition of `parallel_`.

We require a scheme which preserves the safety of unique names but which can also be terse, flexible, and resilient to programmer error. Distinguishing parallel algorithms by execution policy parameters ensures safe disambiguation while also enabling the same terse style shared by the existing algorithms library. The execution policy parameter also provides flexibility and solves the problems of the previous two code examples.

With execution policies, the first dynamic parallelization example becomes:

```
std::vector sort_me = ...
size_t threshold = ...

std::execution_policy exec = std::seq;

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

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

The second static parallelization example becomes:

```
template<ExecutionPolicy>
void func(ExecutionPolicy &policy, std::vector &vec)
{
  std::transform(policy, vec.begin(), vec.end(), vec.begin(), f);
  std::sort(policy, vec.begin(), vec.end());
  std::unique(policy, vec.begin(), vec.end());
}
```

A Nested `std::parallel` Namespace Another naming scheme would be to provide overloads of the existing standard algorithms in a nested `std::parallel` namespace. This scheme would avoid many of the problems we identified with distinguishing parallel algorithms by a name prefix. However, we observed that a namespace would introduce ambiguities when algorithms are invoked via argument dependent lookup:

```
void func(std::vector &vec)
{
  transform(policy, vec.begin(), vec.end(), vec.begin(), f);
  sort(policy, vec.begin(), vec.end());
  unique(policy, vec.begin(), vec.end());
}
```

Are the algorithms invoked by `func` parallel or not? A reader must search for `using` to be sure.

Finally, we note that nested namespaces inside `std::` are unconventional and generally frowned upon.

2.3 Vectorization

In addition to sequential and parallel execution policies, we consider vectorization as a third possibility. In many architectures, an increasing amount of parallelism is exposed via SIMD units. Naturally, affording parallel programmers a simple, standard means of accessing these resources is desirable. Like `std::par`, the `std::vec` execution policy would permit an algorithm invocation to distribute execution across a collection of threads. However, stronger requirements on the behavior of iterator and functor arguments than those permitted by both `std::seq` and `std::par` would allow the implementation greater opportunity to exploit vectorization. These requirements would preclude the use of locks and exceptions in functors, for example.

While preparing this proposal, the existence and precise semantics of `std::vec` were controversial. In particular, it was unclear whether algorithms invoked with a `std::vec` policy were should be permitted to generate concurrency by introducing additional threads. Our current proposal suggests that the restrictions enforced on user-defined code and permissions granted to algorithm invocations by `std::vec` should be a superset of those provided by `std::par`.

Whether or not we decide to propose to include a `std::vec` policy, we expect our library solution to parallelism to complement other library and language forms of vectorization and parallelism in an eventual standard. Just as the existing sequential `std::for_each` library function coexists with the sequential `for` loop language construct, we expect a future C++ standard to offer the programmer a variety of complementary choices along a spectrum of abstraction.

2.4 Execution Policies as Stateful Types

We propose that parallel execution policies have distinct, stateful types:

```
namespace std
{
  class sequential_execution_policy { ... };

  extern const sequential_execution_policy seq;

  class parallel_execution_policy { ... };

  extern const parallel_execution_policy par;

  class vector_execution_policy { .. };

  extern const vector_execution_policy vec;
```

```
  // a dynamic execution policy container
  class execution_policy { ... };
}
```

and that parallel algorithms receive these objects as their first, templatized parameter:

```
template<typename ExecutionPolicy,
         typename Iterator>
void algo(ExecutionPolicy &&exec, Iterator first, Iterator last);
```

Owing to the variety of parallel architectures we propose that implementations be permitted to define non-standard implementation-defined execution policies as extensions. We expect that users with special knowledge about their standard library implementation and underlying parallel architecture will exploit these policies to achieve higher performance.

We believe this design represents existing practice and have tabulated a list of some examples found in parallel algorithm libraries in production:

Library	Execution Policy Type	Notes
Thrust	`thrust::execution_policy`	Controls algorithm dispatch to several different parallel backend targets
TBB	`tbb::auto_partitioner`	Selects an automatic partitioning strategy
PPL	`concurrency::affinity_partitioner`	Improves algorithm cache affinity
Boost.MPI	`boost::mpi::communicator`	Coordinates MPI ranks such that they can cooperate in collective algorithms
Parallel `libstdc++`	`__gnu_parallel::_Parallelism`	Selects from among several parallel execution strategies
C++ AMP	`concurrency::accelerator_view`	Controls algorithm execution locality
Bolt	`bolt::cl::control`	Controls algorithm command queue, debug information, load balancing, etc.

Table 1: Examples of execution policies found in existing libraries

We propose that parallel algorithms receive execution policy objects as their first, instead of last, parameter primarily for two reasons:

1. It mirrors the form of `std::async`'s interface.
2. The first argument allows the reader to easily note the invocation's parallelization at a glance.
3. It preserves the desirable property that algorithms invoked with a lambda look similar to a `for` loop:
   `std::for_each(std::par, vec.begin(), vec.end(), [](int &x){ x += 13; });`

An alternative design would place the execution policy last and provide a default value:

```
template<typename Iterator,
         typename ExecutionPolicy>
void algo(Iterator first, Iterator last, ExecutionPolicy &&exec = std::par);
```

This design would collapse the "surface area" of the algorithms API considerably and provide deeper integration into the existing standard algorithms as execution policies become just a final, optional parameter.

Of the libraries we surveyed, Thrust, Boost.MPI, C++ AMP, and Bolt consistently placed execution policy parameters first. PPL tended to place execution policies last, but occasionally accepted execution policy-like hints such as allocators first. TBB and GNU parallel `libstdc++` consistently placed execution policies last.

2.4.1 Rejected Execution Policy Designs

However, other designs are possible. An alternative design might require all parallel execution policies to be derived from a common root type, say, `std::execution_policy`:

```
namespace std
{

class execution_policy { ... };

class sequential_execution_policy : public execution_policy { ... };

extern const sequential_execution_policy seq;

class parallel_execution_policy : public execution_policy { ... };

extern const parallel_execution_policy par;

class vector_execution_policy : public execution_policy { ... };

extern const vector_execution_policy vec;

}
```

Instead of a template parameter, algorithm interfaces would receive references to `std::execution_policy`:

```
template<typename Iterator>
void algo(std::execution_policy &exec, Iterator first, Iterator last);
```

We rejected this design for a number of reasons:

- Erasing the concrete type of the execution policy may make dispatching the algorithm's implementation more expensive than necessary. We worry that for `std::seq` invocations across small sequences, the cost of type erasure and algorithm dispatch could dominate the cost of the algorithm.
- Requiring an execution policy's type to derive from a particular root type may make it impractical for implementations to define non-standard policies.
- Requiring an execution policy's type to derive from a root would preclude treating policies as simple types with value semantics. Inheritance from a common root would require APIs to receive policies by reference or pointer.

- By making the execution policy parameter a concrete type, we would have to commit to either lvalue or rvalue reference semantics for the parameter. With a template parameter, we may support both.
- Erasing the concrete type of the execution policy would require the implementation to instantiate code for all possible policies for each algorithm invocation. Because parallel algorithm implementations are often significantly more complex than their sequential counterparts, this may result in substantial code generation at each call site.

In our survey of existing library designs, we observed that libraries tended not to adopt a common root type for execution policies.

The exception is Thrust, which exposes a common execution policy root type which allows users of the library to create novel execution policies. However, this proposal's design reserves that privilege for the library implementation.

Some libraries accept a variety of execution policy types to allow for algorithm customization, while others require a single concrete type.

For example, both TBB and PPL allow for customization and receive their partitioner arguments as template parameters. Similarly, GNU parallel `libstdc++` exposes policies as a forest of inheritance trees. The roots of individual trees are unrelated.

Other libraries do not appear to allow for a variety of policies and instead provide a single concrete policy type. These types do not appear to allow customization through inheritance. Boost.MPI, C++ AMP, and Bolt are examples.

Another alternative design might require all parallel execution policies to have the same type:

```
namespace std
{

class execution_policy { ... };

extern const execution_policy seq;

extern const execution_policy par;

extern const execution_policy vec;

}
```

in this alternative design, algorithms would receive such policies by value:

```
template<typename Iterator>
void algo(execution_policy exec, Iterator first, Iterator last);
```

This interface shares most of the same drawbacks as the previous, but allows trafficking execution polices by value.

On the other hand, our proposed algorithm parallel execution policy parameters are similiar in form and spirit to `std::async`'s launch policy parameter, which is a dynamic bitfield. There could be some value in mirroring the convention of `std::async`'s interface in the parallel algorithms library.

2.5 `for_each` Interface Consistency

Because a parallel version of `for_each` cannot accumulate state in its function object argument, the interface we propose for `for_each` returns a copy of its `last` iterator parameter instead of a copy of its function object:

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

The rationale for this choice is to avoid discarding iterator information originating in higher-level algorithms implemented through lowerings to `for_each`.

For example, because `for_each` returns an iterator, `copy` may be implemented through a lowering to `for_each`:

```
template<class ExecutionPolicy, class InputIterator, class OutputIterator>
OutputIterator copy(ExecutionPolicy &&exec,
                    InputIterator first, InputIterator last, OutputIterator result)
{
  return std::get<1>(std::for_each(exec,
                                   __make_zip_iterator(first,result),
                                   __make_zip_iterator(last,result),
                                   __copy_function).get_iterator_tuple());
}
```

Without `for_each`'s result, `copy` must be implemented through some other non-standard means, which may be burdensome. While implementations of the standard library could work around this limitation, it would be regrettable to impose this burden on programmers who wish to implement algorithms with a similar iterator interface.

This is also the motivation behind the addition of our proposed `for_each_n` algorithm, which may implement algorithms such as `copy_n`, `fill_n`, etc.

On the other hand, it may be safer to require our parallel `for_each` to simply return a copy of its function object for consistency's sake.

2.6 Iterator Traversal Requirements

Even though random access to data is a prerequisite for parallel execution, we propose that the interface to parallel algorithms should not impose additional requirements over the existing standard library on the traversal categories of their iterator parameters. In the absense of random access iterators, an implementation may elect to fall back to sequential execution. Alternatively, an implementation may elect to introduce temporary copies of input and output ranges.

2.7 Associativity/Commutativity of Binary Operators

Some parallel algorithms such as `reduce` place stricter requirements on the binary operations they receive than do analogous sequential algorithms such as `accumulate`.

In particular, `reduce` requires its binary operation parameter to be both mathematically associative and commutative in order to accomodate a parallel sum.

To our knowledge, what it means for a binary function object to be associative or commutative is not well-defined by the C++ standard. However, the standard does make such an effort for other mathematical operations, such as strict weak comparison.

For algorithms which require associative binary operators like `reduce`, should we define concepts such as `AssociativeOperation` similarly to `Compare` instead of using `BinaryOperation`?

Because floating point operations are non-associative, a useful definition of this concept would need to be flexible.

2.8 Machine Width and Space Complexity

Our proposal provides asymptotic upper bounds on work complexity for each parallel algorithm in terms of the input size. Asymptotic guarantees on space complexity would be useful as well, particularly because unlike the typical sequential algorithm, many parallel algorithms require non-trivial temporary storage. The size of such temporary storage requirements often depends on the size of the parallel machine.

Unfortunately, C++ does not currently support a notion of parallel machine size. The closest analogue seems to be the value returned by the function `std::thread::hardware_concurrency`.

At first glance, relating work complexity to the result of `std::thread::hardware_concurrency` might seem like a reasonable thing to do. However, we note that the value of this function is merely advisory; it is not guaranteed to correspond to an actual physical machine width. The second more significant problem with interpreting `std::thread::hardware_concurrency` as machine width is that it presumes a particular machine model of parallelism, i.e., one in which the basic primitive of parallelism is a single thread. While this is a good model for some parallel architectures, it is a poor fit for others. For example, the width of a parallel machine with a significant investment in SIMD vector units would be ill-described in terms of threads.

2.9 Container Support for Parallelism

A parallel algorithms library is a fine starting point for exposing parallelism to programmers in an accessible manner. However, algorithms are only a part of a complete solution for authoring parallel C++ programs. In addition to algorithms, the standard library also provides containers for manipulating data in a safe and convenient manner. While this proposal is focused exclusively on standard algorithms, many of the operations on standard containers such as `std::vector` also offer rich opportunities for parallelism. As in sequential programs, without support for parallelism in containers, authoring sophisticated parallel programs will become burdensome as programmers will be forced to manage data in an ad hoc fashion.

Should containers such as `std::vector` be enhanced analogously to the standard algorithms to support parallel execution? We plan to explore the design of such a library in a future paper.

---

3 Algorithm execution policies

Header <execution_policy> synopsis

```
namespace std
{

template<class T> struct is_execution_policy;

class parallel_execution_policy
{
  public:
    void swap(parallel_execution_policy &other);
```

```
    // implementation-defined public members follow
    ...

  private:
    // implementation-defined state follows
    ...
};

void swap(parallel_execution_policy &a, parallel_execution_policy &b);

template<> struct is_execution_policy<parallel_execution_policy> : true_type {};

extern const parallel_execution_policy par;

class sequential_execution_policy
{
  public:
    void swap(sequential_execution_policy &other);

    // implementation-defined public members follow
    ...

  private:
    // implementation-defined state follows
    ...
};

void swap(sequential_execution_policy &a, sequential_execution_policy &b);

template<> struct is_execution_policy<sequential_execution_policy> : true_type {};

extern const sequential_execution_policy seq;

class vector_execution_policy
{
  public:
    void swap(vector_execution_policy &other);

    // implementation-defined public members follow
    ...

  private:
    // implementation-defined state follows
    ...
};

void swap(vector_execution_policy &a, vector_execution_policy &b);

template<> struct is_execution_policy<vector_execution_policy> : true_type {};

extern const vector_execution_policy vec;

// implementation-defined execution policy extensions follow
```

```
...

class execution_policy
{
  public:
    template<class ExecutionPolicy>
    execution_policy(const ExecutionPolicy &exec,
                     typename enable_if<
                       is_execution_policy<ExecutionPolicy>::value
                     >::type * = 0);

    template<class ExecutionPolicy>
    typename enable_if<
      is_execution_policy<ExecutionPolicy>::value,
      execution_policy &
    >::type
    operator=(const ExecutionPolicy &exec);

    void swap(execution_policy &other);

    // obtains the typeid of the stored target
    const type_info& target_type() const;

    // obtains a pointer to the stored target
    template<class ExecutionPolicy>
    typename enable_if<
      is_execution_policy<ExecutionPolicy>::value,
      ExecutionPolicy *
    >::type
    target();

    template<class ExecutionPolicy>
    typename enable_if<
      is_execution_policy<ExecutionPolicy>::value,
      const ExecutionPolicy *
    >::type
    target() const;

  private:
    // exposition only
    int which;

    union
    {
      parallel_execution_policy   parallel;
      sequential_execution_policy sequential;
      vector_execution_policy     vector;

      // any other possibilities follow
      ...
    };
};

void swap(execution_policy &a, execution_policy &b);
```

```
}
```

4 Class template `is_execution_policy`

```
namespace std {
  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 parallel algorithm 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 library is unspecified.

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

5 Class `execution_policy`

1. Objects of type `execution_policy` may be used to dynamically control the invocation of parallel algorithms.

6 Execution Policies and Their Effect on Algorithm Execution

1. Algorithms invoked with an execution policy argument of type `sequential_execution_policy` execute internally in sequential order in the calling thread.

2. Algorithms invoked with an execution policy argument of type `parallel_execution_policy` or `vector_execution_policy` are permitted to execute internally in an unordered fashion in unspecified threads.

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

3. An implementation may provide additional execution policy types besides `parallel_execution_policy`, `sequential_execution_policy`, `vector_execution_policy`, or `execution_policy`. Objects of type `execution_policy` must be constructible and assignable from any additional non-standard execution policy provided by the implementation.

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

5. Algorithms invoked without an execution policy argument execute as if they were invoked with an execution policy argument of type `sequential_execution_policy`.

6. Implementations of `parallel_execution_policy`, `sequential_execution_policy`, and `vector_execution_policy` are permitted to provide additional non-standard data and function members.

   [*Note:* This provision permits objects of these types to be stateful. – *end note.*]

7 Example Usage of `execution_policy`:

`std::execution_policy` allows us to dynamically control algorithm execution:

```
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());
```

`std::execution_policy` allows us to pass execution policies through a binary interface:

```
void some_api(std::execution_policy exec, int arg1, double arg2);

void foo()
{
  // call some_api with std::par
  some_api(std::par, 7, 3.14);
}
```

Retrieving the dynamic value from an `std::execution_policy` an API similar to `std::function`:

```
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;
    std::sequential_execution_policy *exec_ptr = exec.target<std::sequential_execution_policy>();
  }
  else if(exec.target_type() == typeid(std::par))
  {
    std::cout << "Received a parallel policy" << std::endl;
    std::parallel_execution_policy *exec_ptr = exec.target<std::parallel_execution_policy>();
  }
  else if(exec.target_type() == typeid(std::vec))
  {
    std::cout << "Received a vector policy" << std::endl;
    std::vector_execution_policy *exec_ptr = exec.target<std::vector_execution_policy>();
  }
  else
  {
    std::cout << "Received some other kind of policy" << std::eedl;
  }
}
```

In the current design, `std::execution_policy::target` returns a pointer similar to `std::function::target`. However, `std::execution_policy`'s current design precludes an "empty" or invalid state. An alternative design might require `std::execution_policy::target` to return a reference and throw an exception in the case of type mismatch.

8 Reporting Exceptions

8.1 Exception behavior of functions launched through a `std::sequential_execution_policy` argument

An algorithm invoked with a sequential execution policy reports exceptional behavior in the same manner as the legacy algorithms.

8.2 Exception behavior of functions launched through a `std::parallel_execution_policy` argument

An algorithm invoked with a parallel execution policy may report exceptional behavior by throwing an exception.

A parallel algorithm may report exceptional behavior to the caller by throwing one of three exception types:

- If temporary memory resources are required by the algorithm and none are available, the algorithm may throw `std::bad_alloc`.
- If parallel resources are required by the algorithm and none are available, the algorithm may throw `std::system_error`.
- If one or more uncaught exceptions are thrown for any other reason during the execution of the algorithm:
  - The exception is collected in an `exception_list` associated with the algorithm's invocation.
  - If the `exception_list` associated with the algorithm's invocation is non-empty, it is thrown once all tasks have terminated.

Note that these guarantees imply that 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.

8.3 Exception behavior of functions launched through a `std::vector_execution_policy` argument

If program-defined code invoked by an algorithm invoked with a vector execution policy throws an exception, the behavior is undefined.

Header `<exception>` synopsis

```
namespace std {
  class exception_list
  {
    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;

    const char *what() const;

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

9 Interaction with Threads

For algorithms invoked with a `sequential_execution_policy` argument:

1. Program-defined functions arising from the manipulation of algorithm parameters (e.g., functors, iterators, etc.) are invoked in the calling thread of execution.
2. Program-defined functions arising from the manipulation of algorithm parameters are invoked in sequential order.
3. Algorithms synchronize with the caller.

For algorithms invoked with a `parallel_execution_policy` argument:

1. It is the responsibility of the algorithm implementation to allocate temporary storage in a thread-safe manner.

   [*Note:* One way to guarantee this is to allocate all dynamic storage in the calling thread before concurrent computation begins. – *end note.*]
2. It is unspecified in which thread(s) program-defined functions arising from the manipulation of algorithm parameters are invoked.
3. It is unspecified in which order program-defined functions arising from the manipulation of algorithm parameters are invoked.
4. Algorithm invocations synchronize with the caller.
5. Algorithms make no guarantee of forward progress.

   [*Note:* Program-defined code executed by an algorithm's invocation of a functor or iterator dereference cannot wait on the result of another invocation. – *end note.*]
6. An algorithm's behavior is undefined if program-defined code executed through algorithm parameter manipulation may introduce a data race.

For algorithms invoked with a `vector_execution_policy` argument:

1. A vector algorithm invocation inherits all the previous restrictions of `parallel_execution_policy`.
2. An algorithm's behavior is undefined if program-defined code executed through algorithm parameter manipulation throws an exception.
3. An algorithm's behavior is undefined if program-defined code executed through algorithm parameter mainpulation attempts to synchronize.

10 Proposed Additions to Algorithms and Algorithm-like functions

Here we enumerate the additions we propose to the standard algorithms library.

We propose to overload an existing algorithm name when the existing C++11 specification allows sufficient discretion for a parallel implementation (e.g., `transform`) or when we feel no other name would be appropriate (e.g., `for_each`, `inner_product`).

We propose to introduce a new algorithm name when the existing analogous algorithm name implies a sequential implementation (e.g., `accumulate` versus `reduce`).

11 Overloads of Existing Algorithms Introduced by this Proposal

11.1 Existing specialized algorithms from `<memory>`

11.1.1 Header `<memory>` synopsis

```
namespace std {
  // specialized algorithms
  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);
}
```

11.1.2 `uninitialized_copy [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`.

11.1.3 `uninitialized_fill [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`.

11.1.4 `uninitialized_fill_n [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`.

11.2 Existing function Templates from `<algorithm>`

11.2.1 Header `<algorithm>` synopsis

```
namespace std {
  // 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 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);

// 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);

// binary search
template<class ExecutionPolicy,
         class ForwardIterator, class T>
  ForwardIterator
    lower_bound(ExecutionPolicy &&exec,
                ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ExecutionPolicy,
         class ForwardIterator, class T, class Compare>
  ForwardIterator
    lower_bound(ExecutionPolicy &&exec,
                ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

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

template<class ExecutionPolicy,
         class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ExecutionPolicy &&exec,
                ForwardIterator first,
                ForwardIterator last, const T& value);
template<class ExecutionPolicy,
         class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ExecutionPolicy &&exec,
                ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);

template<class ExecutionPolicy,
         class ForwardIterator, class T>
  bool binary_search(ExecutionPolicy &&exec,
                     ForwardIterator first, ForwardIterator last,
                     const T& value);
template<class ExecutionPolicy,
         class ForwardIterator, class T, class Compare>
  bool binary_search(ExecutionPolicy &&exec,
                     ForwardIterator first, ForwardIterator last,
                     const T& value, 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);
} // namespace parallel
}
```

11.2.2 All of [alg.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`.

11.2.3 Any of [alg.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`.

11.2.4 None of [alg.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`.

11.2.5 Find [alg.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`.

11.2.6 Find end [alg.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`.

11.2.7 Find first [alg.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`.

11.2.8 Adjacent find [alg.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`.

11.2.9 Count [alg.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`.

11.2.10 Mismatch [alg.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`.

11.2.11 Equal [alg.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`.

11.2.12 Copy [alg.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`.

11.2.13 Move [alg.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`.

11.2.14 Swap [alg.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`.

11.2.15 Transform [alg.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`.

11.2.16 Replace [alg.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`.

11.2.17 Fill [alg.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`.

11.2.18 Generate [alg.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`.

11.2.19 Remove [alg.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`.

11.2.20 Unique [alg.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`.

11.2.21 Reverse [alg.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`.

11.2.22 Rotate [alg.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`.

11.2.23 Partitions [alg.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`.

11.2.24 `sort [alg.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`.

11.2.25 `stable_sort [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`.

11.2.26 `partial_sort [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`.

11.2.27 partial_sort_copy [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`.

11.2.28 is_sorted [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`.

11.2.29 Nth element [alg.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`.

11.2.30 `lower_bound [lower.bound]`

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

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

1. *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `e < value` or `comp(e, value)`.
2. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
3. *Returns:* The furthermost iterator `i` in the range `[first,last)` such that for any iterator `j` in the range `[first,i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) != false`.
4. *Complexity:* `O(lg (last - first))`.
5. *Remarks:* The signatures shall not participate in overload resolution if

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

11.2.31 `upper_bound [upper.bound]`

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

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

1. *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expression `!(value < e)` or `!comp(value, e)`.

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

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

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

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

11.2.32 `equal_range [equal.range]`

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

template<class ExecutionPolicy,
         class ForwardIterator, class T, class Compare>
  pair<ForwardIterator, ForwardIterator>
    equal_range(ExecutionPolicy &&exec,
                ForwardIterator first,
                ForwardIterator last, const T& value,
                Compare comp);
```

1. *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first,last)`, `e < value` shall imply `!(value < e)` or `comp(e, value)` shall imply `!comp(value, e)`.

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

3. *Returns:*

```
make_pair(lower_bound(forward<ExecutionPolicy>(exec), first, last, value),
          upper_bound(forward<ExecutionPolicy>(exec), first, last, value))
```

   or

```
make_pair(lower_bound(forward<ExecutionPolicy>(exec), first, last, value, comp),
          upper_bound(forward<ExecutionPolicy>(exec), first, last, value, comp))
```

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

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

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

```
template<class ExecutionPolicy,
         class ForwardIterator, class T, class Compare>
  bool binary_search(ExecutionPolicy &&exec,
                     ForwardIterator first, ForwardIterator last,
                     const T& value, Compare comp);
```

1. *Requires:* The elements `e` of `[first, last)` are partitioned with respect to the expressions `e < value` and `!(value< e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` implies `!(value < e)` or `comp(e, value)` implies `!comp(value, e)`.
2. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
3. *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`.
4. *Complexity:* `O(lg (last - first))`.
5. *Remarks:* The signatures shall not participate in overload resolution if

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

11.2.33 Merge [alg.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`.

11.2.34 Includes [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`.

11.2.35 `set_union [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`.

11.2.36 `set_intersection [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`.

11.2.37 `set_difference [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`.

11.2.38 `set_symmetric_difference [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`.

11.2.39 Minimum and maximum `[alg.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`.

11.2.40 Lexicographical comparison [alg.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`.

11.3 Existing generalized numeric operations from `<numeric>`

11.3.1 Header `<numeric>` synopsis

```
namespace std {
  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 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);
}
```

```
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`.

```
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`.

12 Novel Algorithms Introduced by this Proposal

12.1 Novel specialized algorithms from `<memory>`

12.1.1 None.

12.2 Novel algorithms from `<algorithm>`

12.2.1 Header `<algorithm>` synopsis

```
namespace std {
  // non-modifying sequence operations
  template<class ExecutionPolicy,
```

```
          class ForwardIterator, class Function>
    ForwardIterator for_each(ExecutionPolicy &&exec,
                             ForwardIterator first, ForwardIterator last,
                             Function f);
  template<class ExecutionPolicy,
           class InputIterator, class Size, class Function>
    InputIterator for_each_n(ExecutionPolicy &&exec,
                             InputIterator first, Size n,
                             Function f);
} // namespace parallel
}
```

Novel non-modifying sequence operations

12.2.2 For each [alg.foreach]

```
template<class ExecutionPolicy,
         class ForwardIterator, class Function>
  ForwardIterator for_each(ExecutionPolicy &&exec,
                           ForwardIterator first, ForwardIterator 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`.

12.3 Novel generalized numeric operations from `<numeric>`

12.3.1 Header `<numeric>` synopsis

```
namespace std {
  template<class ExecutionPolicy,
           class InputIterator>
    typename iterator_traits<InputIterator>::value_type
      reduce(ExecutionPolicy &&exec,
             InputIterator first, InputIterator last);
```

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

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

12.3.2 Reduce [reduce]

```
template<class ExecutionPolicy,
         class InputIterator>
  typename iterator_traits<InputIterator>::value_type
    reduce(ExecutionPolicy &&exec,
```

```
          InputIterator first, InputIterator last);
```

1. *Effects:* The algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The result of the sum `T(0) + *iter_0 + *iter_1 + *iter_2 + ...` for every iterator `iter_i` 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 signature shall not participate in overload resolution if

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

```
template<class ExecutionPolicy,
         class InputIterator, class T>
  T reduce(ExecutionPolicy &&exec,
           InputIterator first, InputIterator last, T init);

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 algorithm's execution is parallelized as determined by `exec`.
2. *Returns:* The result of the generalized sum `init + *iter_0 + *iter_1 + *iter_2 + ...` or `binary_op(init, binary_op(*iter_0, binary_op(*iter_2, ...)))` for every iterator `iter_i` in the range `[first,last)`.

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

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

1. *Effects:* For each iterator `i` in `[result,result + (last - first))`, performs `*i = prefix_sum`, where `prefix_sum` is the result of the corresponding sum `init + *iter_0 + *iter_1 + *iter_2 + ...` or `binary_op(init, binary_op(*iter_0, binary_op(*iter_1, binary_op(*iter_2, ...)))` for every iterator `iter_j` in the range `[first,first + (i - result) - 1)`.

   The order of operands of the sum is unspecified.

   The algorithm's execution 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 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.

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

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

1. *Effects:* For each iterator `i` in `[result,result + (last - first))`, performs `*i = prefix_sum`, where `prefix_sum` is the result of the corresponding sum `init + *iter_0 + *iter_1 + *iter_2 + ...` or `binary_op(init, binary_op(*iter_0, binary_op(*iter_1, binary_op(*iter_2, ...)))` for every iterator `iter_j` in the range `[first,first + (i - result) - 1)`.

   The order of operands of the sum is unspecified.

   The algorithm's execution 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 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.

13 Existing Algorithms Left Unchanged by this Proposal

This proposal leaves some algorithms of `<memory>`, `<algorithm>`, and `<numeric>` unchanged because they either 1. have no parallelism, 2. have explicitly sequential semantics, 3. have an uncertain parallel implementation and for which we feel parallelization would be low priority.

For example, instead of parallelizing the standard heap algorithms, a better strategy may be to provide concurrent priority queues as a data structure as some have suggested.

13.1 Unchanged specialized algorithms from `<memory>`

- No parallelism
  - `addressof`

13.2 Unchanged algorithms from `<algorithm>`

- No parallelism
  - `iter_swap`
  - `min`
  - `max`
  - `minmax`
- Uncertain / low priority
  - `is_permutation`
  - `search`
  - `search_n`
  - `push_heap`
  - `pop_heap`
  - `make_heap`
  - `sort_heap`
  - `is_heap`
  - `is_heap_until`

  - next_permutation
  - prev_permutation

- Explicitly sequential
  - copy_backward
  - move_backward
  - random_shuffle
  - shuffle

13.3 Unchanged generalized numeric operations from <numeric>

- Explicitly sequential
  - accumulate
  - partial_sum
  - iota