`constexpr` 'Parallel' Algorithms

Document: P2902R1
Date: April 15, 2025
Project: Programming Language C++, Library Working Group
Audience: LEWG & LWG
Reply to: Oliver J. Rosten (`oliver.rosten@gmail.com`)

Abstract

It is proposed to render the overloads of various algorithms accepting an execution policy usable in constant expressions. The goal is to cleanly facilitate the use of algorithms built out of `std` components in runtime, runtime accelerated and compile time contexts. The possibility that constant evaluation itself is accelerated is a byproduct of the flexibility conferred, albeit one unlikely to be exploited in the near future.

CONTENTS

I. REVISION HISTORY

R0 → R1

1. Motivation considerably extended.
   (a) A list of language and library features usable in constant expressions since C++26 is provided to better contextualize this proposal.
   (b) A much more detailed analysis is presented of the cases for and against.
2. Ramifications of the possibility of compile time acceleration discussed.
3. Wording rebased to the latest standard.

II. INTRODUCTION

Parallel algorithms, first appearing in C++17, are a powerful addition to the standard library. They facilitate runtime acceleration without clients having to pay attention to manual synchronization or other low-level implementation details.[1] As such, they provide a natural, declarative abstraction which can offer significant performance gains.

In a seemingly orthogonal development, most `std` algorithms and their `ranges` counterparts have been declared `constexpr` [P0202R3, P0896R4, P2562R1, P2283R0, P3508R0, P3369R0]. Given that the purpose of the parallel algorithms is to improve runtime behaviour, at first sight it makes little sense to add them to the `constexpr` club. However, algorithms naturally compose both with each other and other components. Allowing parallel algorithms to be used in constant expressions implies that composite algorithms can be straightforwardly implemented such that they can be used in compile time / runtime / runtime accelerated contexts, as desired.

Allowing all `std` / `ranges` algorithms to be usable in constant expressions has an appealing regularity and would bring an arc of language/library evolution to a satisfying conclusion; what is proposed in this paper takes us one step closer.

Accepting the premise of this paper, the main design consideration is whether all current execution policies should be usable in constant expressions, or just `execution::seq`. This proposal errs towards the former, for reasons described below.

III. MOTIVATION

A. General Considerations

Suppose an author builds a function out of `std` algorithms (and potentially other components). Suppose further that they wish to use their function in both a compile time context and a runtime one, where the latter is amenable to acceleration. As things stand, this is possible but inelegant, requiring code duplication and a static branch. For example:

[1] Notwithstanding subtleties such as using code which is safe for vectorization with `par_unseq`.

```
template<class Exec, class It>
constexpr void do_thing(Exec exec, It first, It last)
{
  if consteval {
    std::sort(first, last);
    // More code
  }
  else {
    std::sort(exec, first, last);
    // More code
  }
}
```

Of course, in this case it would be advisable for the author to refactor `sort` into a `constexpr` version, minimizing the scope of the static branches. Depending on the other code present, it may be worthwhile to similarly refactor other components. However, this begs the question as to why the standard doesn't already do this, to save clients the work. A simple, general approach would be to allow parallel algorithms to be usable in constant expressions, with a tacit understanding that, when so used, they almost certainly delegate to their serial counterparts. As above, this is straightforward to achieve using `if consteval` [Implementation].

However, is standardization really justified? In the following subsections, we will review the cases for and against.

B. The Case For

1. Regularity of the Language and Library

Since its inception in C++11, the range of both library and language constructs usable in constant expressions has steadily grown. The case for this was eloquently made in the famous talk from 2017, "Constexpr ALL the things" [ConstexprAll]. However, a noteworthy counterpoint was provided in [P2043R0], which advocates instead for a Circle-like approach. Nevertheless, it is apparent that the standardization committee has collectively come down strongly in favour of the former. Indeed, nearly a decade on from the exhortation to constexpr all the things, this program is still being actively pursued and C++26 has seen the following language features made usable in constant expressions:

1. placement `new` [P2747R2];
2. casting from `void*` [P2738R1];
3. throwing exceptions [P3068R6].

These have been complimented by a considerable growth of library components which are now `constexpr`:

1. A greater range of `cmath` and `complex` functions [P1383R2];
2. The stable sorting algorithms [P2562R1];
3. The specialized memory algorithms [P2283R0, P3508R0];
4. `atomic` and `atomic_ref` [P3309R3];
5. Exception types [P3068R6, P3378R2];
6. Containers and adaptors [P3372R2];
7. `inplace_vector` for non-trivial types [P3074R7].

In this context, deciding against enabling parallel algorithms in constant expressions carves out an irregular corner of C++.

2. Opaque Usage

It may be desirable to make use of an execution policy as an implementation detail, invisible to the user except (presumably) through good runtime performance, for example:[2]

```
constexpr float sum(span<const float> s) {
  return reduce(par, s.begin(), s.end());
}
```

In such cases, it would be nice to be able to directly render such functions `constexpr` without the need for an internal static branch.

3. Composition involving Parallel Algorithms

In this section, we give an example of a case where both runtime acceleration and constant evaluation may be desirable.

The vertices of regular polygons may be straightforwardly computed for any number of vertices, N. For low-N this gives familiar primitives such as triangles and squares; as N becomes large we approximate a circle with increasing accuracy. It is straightforward to compose various components of the standard library to do this. Here is a sketch of a building-block which could be used either at compile time or runtime:

```
struct vertex { float x, y; };

template<std::ranges::input_range R>
constexpr void make_poly(R&& r)
{
  constexpr auto pi{std::numbers::pi_v<float>};
  const std::size_t N{std::ranges::size(r)};
  std::ranges::transform(
    std::views::iota(0uz, N),
    r.begin(),
    [N](int i) -> vertex {
```

[2] Thanks to Davis Herring for pointing this out.

```
      const auto theta{2*pi*i/N};
      return {-std::sin(theta), std::cos(theta)};
     });
}
```

Notwithstanding that the `ranges` library does not currently support the parallel overloads (though that is likely to change soon [P3179R7]) it is reasonable for this code (or its non-`ranges` analogue) to take an execution policy: why not have the option to accelerate the generation of polygon vertices for large values of N? On the other hand, it seems perfectly reasonable for this algorithm to be available at compile time: why not compute the vertices of a triangle or square statically?

The least bad way to allow both options with the tools currently available is to write your own `transform` containing the by-now-familiar `if consteval` branch. As before, this begs the question: why doesn't the standard provide this out of the box?

4. Testing for UB

Certain kinds of UB can be detected and reported at compile time. Therefore, it may be profitable to build test cases for this purpose. Were the parallel algorithms to be rendered `constexpr` they could be included in such a testing strategy. It is important to note, though, that such checking would probably not be exhaustive: most likely, at compile time, the parallel algorithms would revert to a sequential implementation. If so, there would be no sensitivity to UB arising from e.g. using `par_unseq` with code that is not vectorization-safe. Nevertheless, that does not mean that tests of this type are without merit, and having them available—even with limited fidelity—may be useful.

C. The Case Against

1. Runtime Acceleration and Constant Evaluation are Disjoint

It is a very reasonable criticism to wonder if there are any domains for which runtime acceleration and compile time programming are both plausibly relevant. In other words, are there any problems that this proposal will realistically solve.

The tension here is an interesting one. Naïvely, one might view `constexpr` as a universal performance panacea. Of course, the reality is that the vast majority of practical calculations take input which is not know (and in many cases cannot be known) until runtime. It is generally these sorts of calculations for which we are interested in parallel acceleration. Contrariwise, for something to be useful for compile time computation, it typically requires inputs to be known by the programmer. The latter implies a degree of structure which is by no means present in every problem.

To exemplify this, imagine performing a numerical simulation of some system. In most cases of interest there is no mathematical formula specifying the boundary conditions. Geometry is placed in the domain however the designer of the product chose to place them; similarly the shape of the geometry and other properties of the system. However, there are exceptions which may apply to the system as a whole or parts thereof. One such is given above: regular polygons are both sufficiently structured and useful.

D. Analysis

There are good reasons to be sceptical of allowing the parallel algorithms to be used at compile time. It requires standardization time, implementation time and it must be maintained. On the other hand, this paper outlines plausible use cases for `constexpr` parallel algorithms: the question is not whether such systems exist, it is whether there are enough of them to justify standardization. There is no definitive answer to this question and so it is useful to look for precedent.

The standardization of `constexpr` atomics [P3309R3] has overlapping considerations with the parallel algorithms: in both cases there is a question of whether a utility originally designed for runtime considerations (threading and/or acceleration) has domains in which code may reasonably be used in constant expressions. Furthermore, although parallelism and concurrency are distinct concepts it is noteworthy that there is a reasonable likelihood of coroutines becoming `constexpr` in C++29 [P3367R0].

As a committee, we have consistently expanded the reach of `constexpr` for nearly 15 years and it shows no signs of slowing down. Given that this paper outlines plausible use cases for `constexpr` parallel algorithms, leaving them out of the `constexpr` club risks unnecessary irregularity of C++.

IV. SCOPE

This paper focuses solely on the parallel overloads in the `<algorithm>` and `<numeric>` headers but excludes the specialized memory algorithms. If there is consensus to do so, the latter could be added to this paper.

It is worth pointing out that adding the various parallel overloads to `ranges` is at an advanced stage [P3179R7]. If, as is likely, the latter proposal is adopted soon then, if there is consensus to do so, this paper could be appropriately updated.

If the premise of allowing the parallel algorithms to be used in constant expressions is accepted, then there is a question as to the scope. There are two reasonable options as to precisely what is allowed in constant expressions: only `execution::seq` or all of the `execution` policies. There are various considerations.

A. Specific Options

1. *The case for only* `execution::seq`

1. Since constant evaluation of a parallel algorithm will almost certainly delegate to the standard version, there can be no misunderstanding as to what is happening. In particular, there will not be any potential confusion arising from clients expecting compile time acceleration.

2. Switching from runtime evaluation to compile time evaluation will (probably) give the same answer. This may not be true for other execution policies for certain algorithms were they to be allowed in constant expressions. For example, `std::reduce` states that the behaviour is non-deterministic if the relevant binary operator is not associative (or is not commutative). It is easy to cook up an example that demonstrates this, since floating-point arithmetic is not associative. Given an integer, `N`, consider the following view:

   ```
   auto series = views::iota(1, N) |
     views::transform([](int i){ return 1.0f/i; });
   ```

   Feeding this view to `reduce`, the various `execution` policies may lead to different answers. For MSVC, these differences appear for `N` at least as small as 255. By using a `constexpr` to force all evaluation to occur at compile time, the differences will disappear, meaning that some of the answers may change. Only allowing `execution::seq` in constant expressions will likely mean that the answer is the same at runtime and compile time, at least on a given platform. However, strictly speaking this isn't actually guaranteed by the standard (see the critique below).

2. *The case for all the* `execution` *policies*

1. Demanding constant evaluation of existing code just works. There is no need to switch the execution policy to `execution::seq`.

2. Functions containing opaque usage of the `execution` policies may be directly rendered `constexpr` (see section III B 2).

3. This approach does not unnecessarily constrain implementors:

   > `std::par` and friends merely relax requirements on the implementation, in hopes that implementations will do something useful with that freedom. Saying that I'm not allowed to give the implementation such freedom during constant evaluation doesn't make a lot of sense even if we all know that (typical, current) implementation will not do anything with it. —Davis Herring

B. Critique

On balance, the second option—allowing all `execution` policies in constant evaluation—is preferred, for the following reasons. First, ensuring compile time/runtime consistency in this context is not compelling. Expecting a definitive answer from a non-deterministic algorithm is to misunderstand the algorithm. Indeed, even successive invocations may give different answers. This is the case for the `series` example above, using `reduce` with `par_unseq` on MSVC. Furthermore, there may anyway be differences across different library implementations, and the result that a particular implementation gives today could legitimately be different tomorrow if the internal details change. Secondly, each item of the case for allowing all policies has merit, with the final one having a particular logical appeal.

Therefore, this paper advocates option 2, above: allowing all `execution` policies to appear in constant expressions.

V. STATE OF THE ART

Both MSVC and libstdc++ have supported the parallel algorithms for some time. Sadly, libc++ is lagging behind. However, this in of itself does not seem like a strong reason for holding back on development in this area, particularly where this development is very much incremental.

It is straightforward to adapt existing implementations for the purposes of this paper. See [Implementation] for a selection of parallel algorithms, based on libstdc++, which are usable in constant expressions.

VI. IMPACT ON THE STANDARD

This is a pure library extension amounting to a liberal sprinkling of `constexpr`. However, there may be a need for some additional wording. This proposal opens up the possibility of constant evaluation being performed on multiple threads. Regardless of whether this is actively exploited, the fact that it can be done in principle may prohibit certain implementations of other library functionality. For example, an implementation of `std::atomic` could not use an `if consteval` branch to perform a simple increment, bypassing the atomic machinery of the runtime branch.[3]

[3] I am grateful to Mark Hoemmen for pointing this out to me.

ACKNOWLEDGMENTS

I would like to thank to Mark Hoemmen for insightful comments on the manuscript. Thanks also to Jon Wakely for bringing some useful papers/resources to my attention.

REFERENCES

[N5008] Thomas Köppe, ed., Working Draft, Standard for Programming Language C++.

[P0202R3] Antony Polukhin, Add Constexpr Modifiers to Functions in `<algorithm>` and `<utility>` Headers

[P0896R4] Eric Niebler, Casey Carter, Christopher Di Bella The One Ranges Proposal

[P2562R1] Oliver J. Rosten, `constexpr` Stable Sorting

[P2283R0] Michael Schellenberger Costa, constexpr for specialized memory algorithms

[P3508R0] Giuseppe D'Angelo, Wording for "constexpr for specialized memory algorithms"

[P3369R0] Giuseppe D'Angelo, constexpr for uninitialized_default_construct

[P1383R2] Oliver J. Rosten, More `constexpr` for `<cmath>` and `<complex>`

[P3309R3] Hana Dusiková, atomic and atomic_ref

[ConstexprAll] Ben Dean, Jason Turner, C++ Now (2017), Constexpr ALL the things

[P2043R0] David Sankel, Don't `constexpr` All The Things

[P2747R2] Barry Revzin, constexpr placement new

[P2738R1] Corentin Jarbot, David Ledger, `constexpr` cast from `void*`: towards `constexpr` type-erasure

[P3068R6] Hana Dusiková, Allowing exception throwing in constant-evaluation

[P3074R7] Barry Revzin, trivial unions (was `std::uninitialized<T>`)

[P3378R2] Hana Dusiková, constexpr exception types

[P3372R2] Hana Dusiková, constexpr containers and adaptors

[P3367R0] Hana Dusiková, constexpr coroutines

[P3179R7] Ruslan Arutyunyan, Alexey Kukanov, Bryce Adelstein Lelbach, C++ parallel range algorithms

[Implementation] `https://github.com/ojrosten/sequoia/blob/constexpr_parallel/Tests/Experimental/ExperimentalTest.cpp`

VII. PROPOSED WORDING

The following proposed changes refer to the Working Paper [N5008].

A. Modifications to "Header `<algorithm>` synopsis" [algorithm.syn]

```
// [alg.all.of], all of
template<class InputIterator, class Predicate>
  constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr bool all_of(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                        ForwardIterator first, ForwardIterator last, Predicate pred);

...

// [alg.any.of], any of
template<class InputIterator, class Predicate>
  constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr bool any_of(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                        ForwardIterator first, ForwardIterator last, Predicate pred);

...

// [alg.none.of], none of
template<class InputIterator, class Predicate>
  constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr bool none_of(ExecutionPolicy&& exec,                   // see [algorithms.parallel.overloads]
                         ForwardIterator first, ForwardIterator last, Predicate pred);

...

// [alg.foreach], for each
template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

```
template<class ExecutionPolicy, class ForwardIterator, class Function>
  constexpr void for_each(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                          ForwardIterator first, ForwardIterator last, Function f);

...

template<class InputIterator, class Size, class Function>
  constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
  constexpr ForwardIterator for_each_n(ExecutionPolicy&& exec,     // see [algorithms.parallel.overloads]
                                       ForwardIterator first, Size n, Function f);

...

// [alg.find], find

template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  constexpr ForwardIterator find(ExecutionPolicy&& exec,           // see [algorithms.parallel.overloads]
                                 ForwardIterator first, ForwardIterator last,
                                 const T& value);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr ForwardIterator find_if(ExecutionPolicy&& exec,        // see [algorithms.parallel.overloads]
                                    ForwardIterator first, ForwardIterator last,
                                    Predicate pred);
template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr ForwardIterator find_if_not(ExecutionPolicy&& exec,    // see [algorithms.parallel.overloads]
                                        ForwardIterator first, ForwardIterator last,
                                        Predicate pred);
...

// [alg.find.end], find end

template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_end(ExecutionPolicy&& exec,                               // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    find_end(ExecutionPolicy&& exec,                               // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
...

// [alg.find.first.of], find first

template<class InputIterator, class ForwardIterator>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
```

```
                ForwardIterator first2, ForwardIterator last2);
template<class InputIterator, class ForwardIterator, class BinaryPredicate>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                ForwardIterator first2, ForwardIterator last2,
                BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator1 first1, ForwardIterator1 last1,
                ForwardIterator2 first2, ForwardIterator2 last2,
                BinaryPredicate pred);
...
```

// [alg.adjacent.find], *adjacent find*

```
template<class ForwardIterator>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ForwardIterator first, ForwardIterator last,
                BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator
    adjacent_find(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator
    adjacent_find(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last,
                BinaryPredicate pred);
...
```

// [alg.count], *count*

```
template<class InputIterator, class T>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count(InputIterator first, InputIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  constexpr typename iterator_traits<ForwardIterator>::difference_type
    count(ExecutionPolicy&& exec,                                  // see [algorithms.parallel.overloads]
          ForwardIterator first, ForwardIterator last, const T& value);
template<class InputIterator, class Predicate>
  constexpr typename iterator_traits<InputIterator>::difference_type
    count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr typename iterator_traits<ForwardIterator>::difference_type
    count_if(ExecutionPolicy&& exec,                               // see [algorithms.parallel.overloads]
             ForwardIterator first, ForwardIterator last, Predicate pred);
...
```

// [mismatch], *mismatch*

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

```
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
...
```

// [alg.equal], *equal*

```
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred);
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool equal(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                       ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
 constexpr bool equal(ExecutionPolicy&& exec,                     // see [algorithms.parallel.overloads]
                      ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool equal(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                       ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr bool equal(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                       ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2,
                       BinaryPredicate pred);
...
```

```
// [alg.search], search
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
  constexpr ForwardIterator1
    search(ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    search(ExecutionPolicy&& exec,                                  // see [algorithms.parallel.overloads]
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator1
    search(ExecutionPolicy&& exec,                                  // see [algorithms.parallel.overloads]
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           BinaryPredicate pred);

...

template<class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ForwardIterator first, ForwardIterator last,
             Size count, const T& value, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
  constexpr ForwardIterator
    search_n(ExecutionPolicy&& exec,                                // see [algorithms.parallel.overloads]
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class Size, class T,
         class BinaryPredicate>
  constexpr ForwardIterator
    search_n(ExecutionPolicy&& exec,                                // see [algorithms.parallel.overloads]
             ForwardIterator first, ForwardIterator last,
             Size count, const T& value,
             BinaryPredicate pred);

...

// [alg.modifying.operations], mutating sequence operations
// [alg.copy], copy
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator copy(InputIterator first, InputIterator last,
                                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2 copy(ExecutionPolicy&& exec,           // see [algorithms.parallel.overloads]
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result);

...

template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class Size,
         class ForwardIterator2>
  constexpr ForwardIterator2 copy_n(ExecutionPolicy&& exec,         // see [algorithms.parallel.overloads]
                                    ForwardIterator1 first, Size n,
```

```
                                    ForwardIterator2 result);

...

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  constexpr ForwardIterator2 copy_if(ExecutionPolicy&& exec,       // see [algorithms.parallel.overloads]
                                     ForwardIterator1 first, ForwardIterator1 last,
                                     ForwardIterator2 result, Predicate pred);

...
```

// [alg.move], *move*

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator move(InputIterator first, InputIterator last,
                                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1,
         class ForwardIterator2>
  constexpr ForwardIterator2 move(ExecutionPolicy&& exec,          // see [algorithms.parallel.overloads]
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result);

...
```

// [alg.swap], *swap*

```
template<class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
                                         ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2 swap_ranges(ExecutionPolicy&& exec,   // see [algorithms.parallel.overloads]
                                         ForwardIterator1 first1, ForwardIterator1 last1,
                                         ForwardIterator2 first2);

...
```

// [alg.transform], *transform*

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first1, InputIterator last1,
              OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  constexpr ForwardIterator2
    transform(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  constexpr ForwardIterator
    transform(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);

...
```

// [alg.replace], *replace*

```
template<class ForwardIterator, class T>
```

```
  constexpr void replace(ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  constexpr void replace(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                         ForwardIterator first, ForwardIterator last,
                         const T& old_value, const T& new_value);
template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ExecutionPolicy&& exec,                 // see [algorithms.parallel.overloads]
                            ForwardIterator first, ForwardIterator last

...

template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator replace_copy(InputIterator first, InputIterator last,
                                        OutputIterator result,
                                        const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  constexpr ForwardIterator2 replace_copy(ExecutionPolicy&& exec,  // see [algorithms.parallel.overloads]
                                          ForwardIterator1 first, ForwardIterator1 last,
                                          ForwardIterator2 result,
                                          const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                                           OutputIterator result,
                                           Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  constexpr ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec,                        // see
[algorithms.parallel.overloads]
                                             ForwardIterator1 first, ForwardIterator1 last,
                                             ForwardIterator2 result,
                                             Predicate pred, const T& new_value);

...

// [alg.fill], fill

template<class ForwardIterator, class T>
  constexpr void fill(ForwardIterator first, ForwardIterator last, const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  constexpr void fill(ExecutionPolicy&& exec,                       // see [algorithms.parallel.overloads]
                      ForwardIterator first, ForwardIterator last, const T& value);
template<class OutputIterator, class Size, class T>
  constexpr OutputIterator fill_n(OutputIterator first, Size n, const T& value);
template<class ExecutionPolicy, class ForwardIterator,
         class Size, class T>
  constexpr ForwardIterator fill_n(ExecutionPolicy&& exec,          // see [algorithms.parallel.overloads]
                                   ForwardIterator first, Size n, const T& value);

...

// [alg.generate], generate

template<class ForwardIterator, class Generator>
  constexpr void generate(ForwardIterator first, ForwardIterator last,
                          Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
  constexpr void generate(ExecutionPolicy&& exec,                   // see [algorithms.parallel.overloads]
                          ForwardIterator first, ForwardIterator last,
                          Generator gen);
template<class OutputIterator, class Size, class Generator>
  constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
  constexpr ForwardIterator generate_n(ExecutionPolicy&& exec,      // see [algorithms.parallel.overloads]
                                       ForwardIterator first, Size n, Generator gen);
```

```
...
```

// [alg.remove], *remove*

```
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                   const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  constexpr ForwardIterator remove(ExecutionPolicy&& exec,         // see [algorithms.parallel.overloads]
                                   ForwardIterator first, ForwardIterator last,
                                   const T& value);
template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ExecutionPolicy&& exec,      // see [algorithms.parallel.overloads]
                                      ForwardIterator first, ForwardIterator last,
                                      Predicate pred);

...

template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  constexpr ForwardIterator2
    remove_copy(ExecutionPolicy&& exec,                            // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);
template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  constexpr ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, Predicate pred);

 ...
```

// [alg.unique], *unique*

```
template<class ForwardIterator>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator unique(ExecutionPolicy&& exec,         // see [algorithms.parallel.overloads]
                                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
  constexpr ForwardIterator unique(ExecutionPolicy&& exec,         // see [algorithms.parallel.overloads]
                                   ForwardIterator first, ForwardIterator last,
                                   BinaryPredicate pred);

...

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

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);

...

// [alg.reverse], reverse

template<class BidirectionalIterator>
  constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator>
  constexpr void reverse(ExecutionPolicy&& exec,                 // see [algorithms.parallel.overloads]
                         BidirectionalIterator first, BidirectionalIterator last);

...

template<class BidirectionalIterator, class OutputIterator>
  constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                 OutputIterator result);
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
  constexpr ForwardIterator
    reverse_copy(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                 BidirectionalIterator first, BidirectionalIterator last,
                 ForwardIterator result);

...

// [alg.rotate], rotate

template<class ForwardIterator>
  constexpr ForwardIterator rotate(ForwardIterator first,
                                   ForwardIterator middle,
                                   ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator rotate(ExecutionPolicy&& exec,       // see [algorithms.parallel.overloads]
                                   ForwardIterator first,
                                   ForwardIterator middle,
                                   ForwardIterator last);
...

template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle,
                ForwardIterator last, OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                ForwardIterator1 first, ForwardIterator1 middle,
                ForwardIterator1 last, ForwardIterator2 result);

...

// [alg.shift], shift

template<class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator
```

```
    shift_left(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
               ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);

...

template<class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);

...

// [alg.sorting], sorting and related operations
// [alg.sort], sorting
template<class RandomAccessIterator>
  constexpr void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr void sort(RandomAccessIterator first, RandomAccessIterator last,
                      Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr void sort(ExecutionPolicy&& exec,                   // see [algorithms.parallel.overloads]
                      RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr void sort(ExecutionPolicy&& exec,                   // see [algorithms.parallel.overloads]
                      RandomAccessIterator first, RandomAccessIterator last,
                      Compare comp);

...

template<class RandomAccessIterator>
  constexpr void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                             Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr void stable_sort(ExecutionPolicy&& exec,            // see [algorithms.parallel.overloads]
                             RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr void stable_sort(ExecutionPolicy&& exec,            // see [algorithms.parallel.overloads]
                             RandomAccessIterator first, RandomAccessIterator last,
                             Compare comp);

 ...

template<class RandomAccessIterator>
  constexpr void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                              RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                              RandomAccessIterator last, Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr void partial_sort(ExecutionPolicy&& exec,           // see [algorithms.parallel.overloads]
                              RandomAccessIterator first, RandomAccessIterator middle,
                              RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr void partial_sort(ExecutionPolicy&& exec,           // see [algorithms.parallel.overloads]
                              RandomAccessIterator first, RandomAccessIterator middle,
                              RandomAccessIterator last, Compare comp);

...
```

```
template<class InputIterator, class RandomAccessIterator>
  constexpr RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    partial_sort_copy(InputIterator first, InputIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
  constexpr RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,                       // see [algorithms.parallel.overloads]
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last);
template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
  constexpr RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,                       // see [algorithms.parallel.overloads]
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);

...

template<class ForwardIterator>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                           Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr bool is_sorted(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                           ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  constexpr bool is_sorted(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                           ForwardIterator first, ForwardIterator last,
                           Compare comp);

...

template<class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ForwardIterator first, ForwardIterator last,
                    Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                    ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  constexpr ForwardIterator
    is_sorted_until(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                    ForwardIterator first, ForwardIterator last,
                    Compare comp);

...

// [alg.nth.element], Nth element

template<class RandomAccessIterator>
  constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                             RandomAccessIterator last);
```

```
  template<class RandomAccessIterator, class Compare>
    constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                               RandomAccessIterator last, Compare comp);
  template<class ExecutionPolicy, class RandomAccessIterator>
    constexpr void nth_element(ExecutionPolicy&& exec,              // see [algorithms.parallel.overloads]
                               RandomAccessIterator first, RandomAccessIterator nth,
                               RandomAccessIterator last);
  template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
    constexpr void nth_element(ExecutionPolicy&& exec,              // see [algorithms.parallel.overloads]
                               RandomAccessIterator first, RandomAccessIterator nth,
                               RandomAccessIterator last, Compare comp);

  ...

  // [alg.partitions], partitions

  template<class InputIterator, class Predicate>
    constexpr bool is_partitioned(InputIterator first, InputIterator last, Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    constexpr bool is_partitioned(ExecutionPolicy&& exec,           // see [algorithms.parallel.overloads]
                                  ForwardIterator first, ForwardIterator last, Predicate pred);

  ...

  template<class ForwardIterator, class Predicate>
    constexpr ForwardIterator partition(ForwardIterator first,
                                        ForwardIterator last,
                                        Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    constexpr ForwardIterator partition(ExecutionPolicy&& exec,     // see [algorithms.parallel.overloads]
                                        ForwardIterator first,
                                        ForwardIterator last,
                                        Predicate pred);

  ...

  template<class BidirectionalIterator, class Predicate>
    constexpr BidirectionalIterator stable_partition(BidirectionalIterator first,
                                                     BidirectionalIterator last,
                                                     Predicate pred);
  template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
    constexpr BidirectionalIterator stable_partition(ExecutionPolicy&& exec,                    // see
[algorithms.parallel.overloads]
                                                     BidirectionalIterator first,
                                                     BidirectionalIterator last,
                                                     Predicate pred);

  ...

  template<class InputIterator, class OutputIterator1,
           class OutputIterator2, class Predicate>
    constexpr pair<OutputIterator1, OutputIterator2>
      partition_copy(InputIterator first, InputIterator last,
                     OutputIterator1 out_true, OutputIterator2 out_false,
                     Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
           class ForwardIterator2, class Predicate>
    constexpr pair<ForwardIterator1, ForwardIterator2>
      partition_copy(ExecutionPolicy&& exec,                        // see [algorithms.parallel.overloads]
                     ForwardIterator first, ForwardIterator last,
                     ForwardIterator1 out_true, ForwardIterator2 out_false,
                     Predicate pred);

  ...

  // [alg.merge], merge

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

```
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
         class Compare>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    merge(ExecutionPolicy&& exec,                                  // see [algorithms.parallel.overloads]
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    merge(ExecutionPolicy&& exec,                                  // see [algorithms.parallel.overloads]
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result, Compare comp);

...

template<class BidirectionalIterator>
  constexpr void inplace_merge(BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
  constexpr void inplace_merge(BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator>
  constexpr void inplace_merge(ExecutionPolicy&& exec,             // see [algorithms.parallel.overloads]
                               BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
  constexpr void inplace_merge(ExecutionPolicy&& exec,             // see [algorithms.parallel.overloads]
                               BidirectionalIterator first,
                               BidirectionalIterator middle,
                               BidirectionalIterator last, Compare comp);

...
```

// [alg.set.operations], *set operations*

```
template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool includes(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  constexpr bool includes(ExecutionPolicy&& exec,                  // see [algorithms.parallel.overloads]
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          Compare comp);
```

```
...

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
              set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    set_union(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_union(ExecutionPolicy&& exec,                              // see [algorithms.parallel.overloads]
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result, Compare comp);

...

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    set_intersection(ExecutionPolicy&& exec,                       // see [algorithms.parallel.overloads]
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_intersection(ExecutionPolicy&& exec,                       // see [algorithms.parallel.overloads]
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result, Compare comp);

...

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
```

```
  constexpr ForwardIterator
    set_difference(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_difference(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result, Compare comp);

...

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,                // see [algorithms.parallel.overloads]
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,                // see [algorithms.parallel.overloads]
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result, Compare comp);

...

// [alg.heap.operations], heap operations

template<class RandomAccessIterator>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr bool is_heap(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                         RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr bool is_heap(ExecutionPolicy&& exec,                    // see [algorithms.parallel.overloads]
                         RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);

...

template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr RandomAccessIterator
```

```
    is_heap_until(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                  RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

...
```

// [alg.min.max], *minimum and maximum*

```
...

template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator min_element(ExecutionPolicy&& exec,    // see [algorithms.parallel.overloads]
                                        ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  constexpr ForwardIterator min_element(ExecutionPolicy&& exec,    // see [algorithms.parallel.overloads]
                                        ForwardIterator first, ForwardIterator last,
                                        Compare comp);

...

template<class ForwardIterator>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                        Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator max_element(ExecutionPolicy&& exec,    // see [algorithms.parallel.overloads]
                                        ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  constexpr ForwardIterator max_element(ExecutionPolicy&& exec,    // see [algorithms.parallel.overloads]
                                        ForwardIterator first, ForwardIterator last,
                                        Compare comp);

...

template<class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                   ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
  constexpr pair<ForwardIterator, ForwardIterator>
    minmax_element(ExecutionPolicy&& exec,                         // see [algorithms.parallel.overloads]
                   ForwardIterator first, ForwardIterator last, Compare comp);

...
```

// [alg.lex.comparison], *lexicographical comparison*

```
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
```

```
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool
    lexicographical_compare(ExecutionPolicy&& exec,               // see [algorithms.parallel.overloads]
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  constexpr bool
    lexicographical_compare(ExecutionPolicy&& exec,               // see [algorithms.parallel.overloads]
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);
```

...

B. Modifications to "Non-modifying sequence operations" [alg.nonmodifying]

All of [alg.all.of]

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

...

Any of [alg.any.of]

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

...

None of [alg.none.of]

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

...

For each [alg.foreach]

...

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

...

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

...

Find [alg.find]

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

template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  constexpr ForwardIterator find_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                                    Predicate pred);

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

...

Find end [alg.find.end]

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

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

...

Find first [alg.find.first.of]

```
template<class InputIterator, class ForwardIterator>
  constexpr InputIterator
    find_first_of(InputIterator first1, InputIterator last1,
                  ForwardIterator first2, ForwardIterator last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator1
    find_first_of(ExecutionPolicy&& exec,
                  ForwardIterator1 first1, ForwardIterator1 last1,
                  ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator, class ForwardIterator,
         class BinaryPredicate>
  constexpr InputIterator
```

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

...

Adjacent find [alg.adjacent.find]

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

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

...

Count [alg.count]

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

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

...

[alg.mismatch]

```
template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
```

```
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, InputIterator2 last2,
             BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    mismatch(ExecutionPolicy&& exec,
             ForwardIterator1 first1, ForwardIterator1 last1,
             ForwardIterator2 first2, ForwardIterator2 last2,
             BinaryPredicate pred);
...
```

Equal **[alg.equal]**

```
template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool equal(ExecutionPolicy&& exec,
                       ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr bool equal(ExecutionPolicy&& exec,
                       ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool equal(ExecutionPolicy&& exec,
                       ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                       BinaryPredicate pred);
```

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

...
```

Search [alg.search]

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

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

...

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

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

...
```

C. Modifications to "Mutating sequence operations" [alg.modifying.operations]

Copy [alg.copy]

```
...

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2 copy(ExecutionPolicy&& policy,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result);

...

template<class InputIterator, class Size, class OutputIterator>
  constexpr OutputIterator copy_n(InputIterator first, Size n,
                                  OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class Size, class ForwardIterator2>
   constexpr ForwardIterator2 copy_n(ExecutionPolicy&& exec,
                                     ForwardIterator1 first, Size n,
                                     ForwardIterator2 result);

...

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
                                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  constexpr ForwardIterator2 copy_if(ExecutionPolicy&& exec,
                                     ForwardIterator1 first, ForwardIterator1 last,
                                     ForwardIterator2 result, Predicate pred);

...
```

Move [alg.move]

```
...

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2 move(ExecutionPolicy&& policy,
                                  ForwardIterator1 first, ForwardIterator1 last,
                                  ForwardIterator2 result);

...
```

Swap [alg.swap]

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

...
```

Transform [alg.transform]

```
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
  constexpr OutputIterator
    transform(InputIterator first1, InputIterator last1,
              OutputIterator result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
  constexpr ForwardIterator2
    transform(ExecutionPolicy&& exec,
```

```
            ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class BinaryOperation>
  constexpr OutputIterator
    transform(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, OutputIterator result,
              BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class BinaryOperation>
  constexpr ForwardIterator
    transform(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator result,
              BinaryOperation binary_op);

...
```

Replace [alg.replace]

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

template<class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
  constexpr void replace_if(ExecutionPolicy&& exec,
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred, const T& new_value);

...

template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    replace_copy(InputIterator first, InputIterator last,
                 OutputIterator result,
                 const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  constexpr ForwardIterator2
    replace_copy(ExecutionPolicy&& exec,
                 ForwardIterator1 first, ForwardIterator1 last,
                 ForwardIterator2 result,
                 const T& old_value, const T& new_value);

template<class InputIterator, class OutputIterator, class Predicate, class T>
  constexpr OutputIterator
    constexpr replace_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result,
                              Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate, class T>
  constexpr ForwardIterator2
    replace_copy_if(ExecutionPolicy&& exec,
                    ForwardIterator1 first, ForwardIterator1 last,
                    ForwardIterator2 result,
                    Predicate pred, const T& new_value);
```

Fill [alg.fill]

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

```
  constexpr void fill(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last, const T& value);

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

...
```

Generate [alg.generate]

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

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

...
```

Remove [alg.remove]

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

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

...

template<class InputIterator, class OutputIterator, class T>
  constexpr OutputIterator
    remove_copy(InputIterator first, InputIterator last,
                OutputIterator result, const T& value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class T>
  constexpr ForwardIterator2
    remove_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, const T& value);

template<class InputIterator, class OutputIterator, class Predicate>
  constexpr OutputIterator
    remove_copy_if(InputIterator first, InputIterator last,
                   OutputIterator result, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Predicate>
  constexpr ForwardIterator2
    remove_copy_if(ExecutionPolicy&& exec,
```

```
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, Predicate pred);

...
```

Unique [alg.unique]

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

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

...

template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result);

template<class InputIterator, class OutputIterator,
         class BinaryPredicate>
  constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last,
                OutputIterator result, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  constexpr ForwardIterator2
    unique_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result, BinaryPredicate pred);

...
```

Reverse [alg.reverse]

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

...

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

...
```

Rotate [alg.rotate]

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

...

template<class ForwardIterator, class OutputIterator>
  constexpr OutputIterator
    rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last,
                OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    rotate_copy(ExecutionPolicy&& exec,
                ForwardIterator1 first, ForwardIterator1 middle, ForwardIterator1 last,
                ForwardIterator2 result);

...
```

Shift [alg.shift]

```
template<class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator
    shift_left(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
               typename iterator_traits<ForwardIterator>::difference_type n);

...

template<class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);
template<class ExecutionPolicy, class ForwardIterator>
  constexpr ForwardIterator
    shift_right(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                typename iterator_traits<ForwardIterator>::difference_type n);

...
```

D. Modifications to "Sorting and related operations" [alg.sorting]

Sorting [alg.sort]

```
...

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

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

```
                    RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);

...

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

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

...

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

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

...

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

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

```
  constexpr RandomAccessIterator
    partial_sort_copy(ExecutionPolicy&& exec,
                      ForwardIterator first, ForwardIterator last,
                      RandomAccessIterator result_first,
                      RandomAccessIterator result_last,
                      Compare comp);

...

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

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

...
```

Nth element [alg.nth.element]

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

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

...
```

Partitions [alg.partitions]

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

...

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

...
```

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

...

template<class InputIterator, class OutputIterator1, class OutputIterator2, class Predicate>
  constexpr pair<OutputIterator1, OutputIterator2>
    partition_copy(InputIterator first, InputIterator last,
                   OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
         class ForwardIterator2, class Predicate>
  constexpr pair<ForwardIterator1, ForwardIterator2>
    partition_copy(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last,
                   ForwardIterator1 out_true, ForwardIterator2 out_false, Predicate pred);

...
```

Merge [alg.merge]

```
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    merge(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    merge(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          ForwardIterator result, Compare comp);

...

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

template<class BidirectionalIterator, class Compare>
```

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

...
```

Set operations on sorted structures **[alg.set.operations]**

```
...

template<class InputIterator1, class InputIterator2>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool includes(ExecutionPolicy&& exec,
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Compare>
  constexpr bool includes(ExecutionPolicy&& exec,
                          ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                          Compare comp);

...

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr  ForwardIterator
    set_union(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
    set_union(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr  ForwardIterator
    set_union(ExecutionPolicy&& exec,
              ForwardIterator1 first1, ForwardIterator1 last1,
              ForwardIterator2 first2, ForwardIterator2 last2,
              ForwardIterator result, Compare comp);

...

template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
```

```
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    set_intersection(ExecutionPolicy&& exec,
                    ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_intersection(ExecutionPolicy&& exec,
                    ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    ForwardIterator result, Compare comp);

...

template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    set_difference(ExecutionPolicy&& exec,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_difference(ExecutionPolicy&& exec,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result, Compare comp);

...

template<class InputIterator1, class InputIterator2,
         class OutputIterator>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
  constexpr ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
```

```
                            ForwardIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2,
                             OutputIterator result, Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
  constexpr ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
                             ForwardIterator1 first1, ForwardIterator1 last1,
                             ForwardIterator2 first2, ForwardIterator2 last2,
                             ForwardIterator result, Compare comp);

...
```

Heap operations **[alg.heap.operations]**

```
...

template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr bool is_heap(ExecutionPolicy&& exec,
                         RandomAccessIterator first, RandomAccessIterator last);

...

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr bool is_heap(ExecutionPolicy&& exec,
                         RandomAccessIterator first, RandomAccessIterator last,
                         Compare comp);

...

template<class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
  constexpr RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  constexpr RandomAccessIterator
    is_heap_until(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

...
```

Minimum and maximum **[alg.min.max]**

```
...

template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

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

template<class ForwardIterator, class Compare>
```

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

...

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

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

...

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

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

...
```

Lexicographical comparison [alg.lex.comparison]

```
template<class InputIterator1, class InputIterator2>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  constexpr bool
    lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                            InputIterator2 first2, InputIterator2 last2,
                            Compare comp);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class Compare>
  constexpr bool
    lexicographical_compare(ExecutionPolicy&& exec,
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            Compare comp);

...
```

E. Modifications to "Header <numeric> synopsis" [numeric.ops.overview]

```
// [reduce], reduce

...

template<class ExecutionPolicy, class ForwardIterator>
  constexpr typename iterator_traits<ForwardIterator>::value_type
    reduce(ExecutionPolicy&& exec,                                  // see [algorithms.parallel.overloads]
           ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class T>
  constexpr T reduce(ExecutionPolicy&& exec,                        // see [algorithms.parallel.overloads]
                     ForwardIterator first, ForwardIterator last, T init);
template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
  constexpr T reduce(ExecutionPolicy&& exec,                        // see [algorithms.parallel.overloads]
                     ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op);

...

// [transform.reduce], transform reduce

...

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T>
  constexpr T transform_reduce(ExecutionPolicy&& exec,              // see [algorithms.parallel.overloads]
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2, T init);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  constexpr T transform_reduce(ExecutionPolicy&& exec,              // see [algorithms.parallel.overloads]
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2, T init,
                               BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
template<class ExecutionPolicy, class ForwardIterator, class T,
         class BinaryOperation, class UnaryOperation>
  constexpr T transform_reduce(ExecutionPolicy&& exec,              // see [algorithms.parallel.overloads]
                               ForwardIterator first, ForwardIterator last, T init,
                               BinaryOperation binary_op, UnaryOperation unary_op);

...

// [exclusive.scan], exclusive scan

...

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  constexpr ForwardIterator2
    exclusive_scan(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, T init);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation>
  constexpr ForwardIterator2
    exclusive_scan(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, T init, BinaryOperation binary_op);

...

// [inclusive.scan], inclusive scan

...

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    inclusive_scan(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
```

```
                  ForwardIterator1 first, ForwardIterator1 last,
                  ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  constexpr ForwardIterator2
    inclusive_scan(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                  ForwardIterator1 first, ForwardIterator1 last,
                  ForwardIterator2 result, BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class T>
  constexpr ForwardIterator2
    inclusive_scan(ExecutionPolicy&& exec,                          // see [algorithms.parallel.overloads]
                  ForwardIterator1 first, ForwardIterator1 last,
                  ForwardIterator2 result, BinaryOperation binary_op, T init);
```

// [transform.exclusive.scan], *transform exclusive scan*

...

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation, class UnaryOperation>
  constexpr ForwardIterator2
    transform_exclusive_scan(ExecutionPolicy&& exec,                // see [algorithms.parallel.overloads]
                             ForwardIterator1 first, ForwardIterator1 last,
                             ForwardIterator2 result, T init,
                             BinaryOperation binary_op, UnaryOperation unary_op);
```

// [transform.inclusive.scan], *transform inclusive scan*

...

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation>
  constexpr ForwardIterator2
    transform_inclusive_scan(ExecutionPolicy&& exec,                // see [algorithms.parallel.overloads]
                             ForwardIterator1 first, ForwardIterator1 last,
                             ForwardIterator2 result, BinaryOperation binary_op,
                             UnaryOperation unary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation, class T>
  constexpr ForwardIterator2
    transform_inclusive_scan(ExecutionPolicy&& exec,                // see [algorithms.parallel.overloads]
                             ForwardIterator1 first, ForwardIterator1 last,
                             ForwardIterator2 result,
                             BinaryOperation binary_op, UnaryOperation unary_op, T init);
```

// [adjacent.difference], *adjacent difference*

...

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    adjacent_difference(ExecutionPolicy&& exec,                     // see [algorithms.parallel.overloads]
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  constexpr ForwardIterator2
    adjacent_difference(ExecutionPolicy&& exec,                     // see [algorithms.parallel.overloads]
                        ForwardIterator1 first, ForwardIterator1 last,
                        ForwardIterator2 result, BinaryOperation binary_op);
```

F. Modifications to "Generalized numeric operations" [numeric.ops]

Reduce **[reduce]**

```
...

template<class ExecutionPolicy, class ForwardIterator>
  constexpr typename iterator_traits<ForwardIterator>::value_type
    reduce(ExecutionPolicy&& exec,
           ForwardIterator first, ForwardIterator last);

...

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

...

template<class InputIterator, class T, class BinaryOperation>
  constexpr T reduce(InputIterator first, InputIterator last, T init,
                     BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
  constexpr T reduce(ExecutionPolicy&& exec,
                     ForwardIterator first, ForwardIterator last, T init,
                     BinaryOperation binary_op);

...
```

Transform reduce [transform.reduce]

```
...

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T>
  constexpr T transform_reduce(ExecutionPolicy&& exec,
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2,
                               T init);

...

template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  constexpr T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                               InputIterator2 first2,
                               T init,
                               BinaryOperation1 binary_op1,
                               BinaryOperation2 binary_op2);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
  constexpr T transform_reduce(ExecutionPolicy&& exec,
                               ForwardIterator1 first1, ForwardIterator1 last1,
                               ForwardIterator2 first2,
                               T init,
                               BinaryOperation1 binary_op1,
                               BinaryOperation2 binary_op2);

...

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

...

Exclusive scan **[exclusive.scan]**

...

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
  constexpr ForwardIterator2
    exclusive_scan(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, T init);

...

template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
  constexpr OutputIterator
    exclusive_scan(InputIterator first, InputIterator last,
                   OutputIterator result, T init, BinaryOperation binary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
  constexpr ForwardIterator2
    exclusive_scan(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, T init, BinaryOperation binary_op);
```

...

Inclusive scan **[inclusive.scan]**

...

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    inclusive_scan(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result);

...

template<class InputIterator, class OutputIterator, class BinaryOperation>
  constexpr OutputIterator
    inclusive_scan(InputIterator first, InputIterator last,
                   OutputIterator result, BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  constexpr ForwardIterator2
    inclusive_scan(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, BinaryOperation binary_op);

template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
  constexpr OutputIterator
    inclusive_scan(InputIterator first, InputIterator last,
                   OutputIterator result, BinaryOperation binary_op, T init);
  template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class BinaryOperation, class T>
  constexpr ForwardIterator2
    inclusive_scan(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last,
                   ForwardIterator2 result, BinaryOperation binary_op, T init);
```

...

Transform exclusive scan **[transform.exclusive.scan]**

```
 template<class InputIterator, class OutputIterator, class T,
          class BinaryOperation, class UnaryOperation>
  constexpr OutputIterator
```

```
    transform_exclusive_scan(InputIterator first, InputIterator last,
                             OutputIterator result, T init,
                             BinaryOperation binary_op, UnaryOperation unary_op);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation, class UnaryOperation>
  constexpr ForwardIterator2
    transform_exclusive_scan(ExecutionPolicy&& exec,
                             ForwardIterator1 first, ForwardIterator1 last,
                             ForwardIterator2 result, T init,
                             BinaryOperation binary_op, UnaryOperation unary_op);
```

...

Transform inclusive scan [transform.inclusive.scan]

```
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation>
  constexpr OutputIterator
    transform_inclusive_scan(InputIterator first, InputIterator last,
                             OutputIterator result,
                             BinaryOperation binary_op, UnaryOperation unary_op);
  template<class ExecutionPolicy,
           class ForwardIterator1, class ForwardIterator2,
           class BinaryOperation, class UnaryOperation>
    constexpr ForwardIterator2
      transform_inclusive_scan(ExecutionPolicy&& exec,
                               ForwardIterator1 first, ForwardIterator1 last,
                               ForwardIterator2 result,
                               BinaryOperation binary_op, UnaryOperation unary_op);
template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation, class T>
  constexpr OutputIterator
    transform_inclusive_scan(InputIterator first, InputIterator last,
                             OutputIterator result,
                             BinaryOperation binary_op, UnaryOperation unary_op,
                             T init);
template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation, class T>
  constexpr ForwardIterator2
    transform_inclusive_scan(ExecutionPolicy&& exec,
                             ForwardIterator1 first, ForwardIterator1 last,
                             ForwardIterator2 result,
                            BinaryOperation binary_op, UnaryOperation unary_op,
                            T init);
```

...

Adjacent difference [adjacent.difference]

```
template<class InputIterator, class OutputIterator>
  constexpr OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
  constexpr ForwardIterator2
    adjacent_difference(ExecutionPolicy&& exec,
                        ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
  constexpr OutputIterator
    adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result, BinaryOperation binary_op);
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
           class BinaryOperation>
  constexpr ForwardIterator2
    adjacent_difference(ExecutionPolicy&& exec,
                        ForwardIterator1 first, ForwardIterator1 last,
```

```
                    ForwardIterator2 result, BinaryOperation binary_op);
...
```

G. Modifications to Modifications to "Header <version> synopsis" [version.syn]

```
#define __cpp_lib_constexpr_algorithms   ~~201806L~~ 20????L   // also in <algorithm>
#define __cpp_lib_constexpr_numeric      ~~201911L~~ 20????L   // also in <numeric>
```