Parallel Range Algorithms

The Evolution of Parallelism in C++

RUSLAN ARUTYUNYAN

Cppcon
The C++ Conference

2025
September 13 - 19

About myself

- Working for Intel

About myself

- Working for Intel
- oneAPI Data Parallel C++ (oneDPL) lead developer

About myself

- Working for Intel
- oneAPI Data Parallel C++ (oneDPL) lead developer
- Significant contributions to other threading engines including oneAPI Threading Building Blocks (oneTBB)

About myself

- Working for Intel
- oneAPI Data Parallel C++ (oneDPL) lead developer
- Significant contributions to other threading engines including oneAPI Threading Building Blocks (oneTBB)
- Contributor to SYCL (in the past)

About myself

- Working for Intel
- oneAPI Data Parallel C++ (oneDPL) lead developer
- Significant contributions to other threading engines including oneAPI Threading Building Blocks (oneTBB)
- Contributor to SYCL (in the past)
- Contributions to C++ standard including `std::simd`, `std::execution`, and more

About myself

- Working for Intel
- oneAPI Data Parallel C++ (oneDPL) lead developer
- Significant contributions to other threading engines including oneAPI Threading Building Blocks (oneTBB)
- Contributor to SYCL (in the past)
- Contributions to C++ standard including **`std::simd`**, **`std::execution`**, and more
- SG1: Concurrency and Parallelism co-chair in the C++ committee

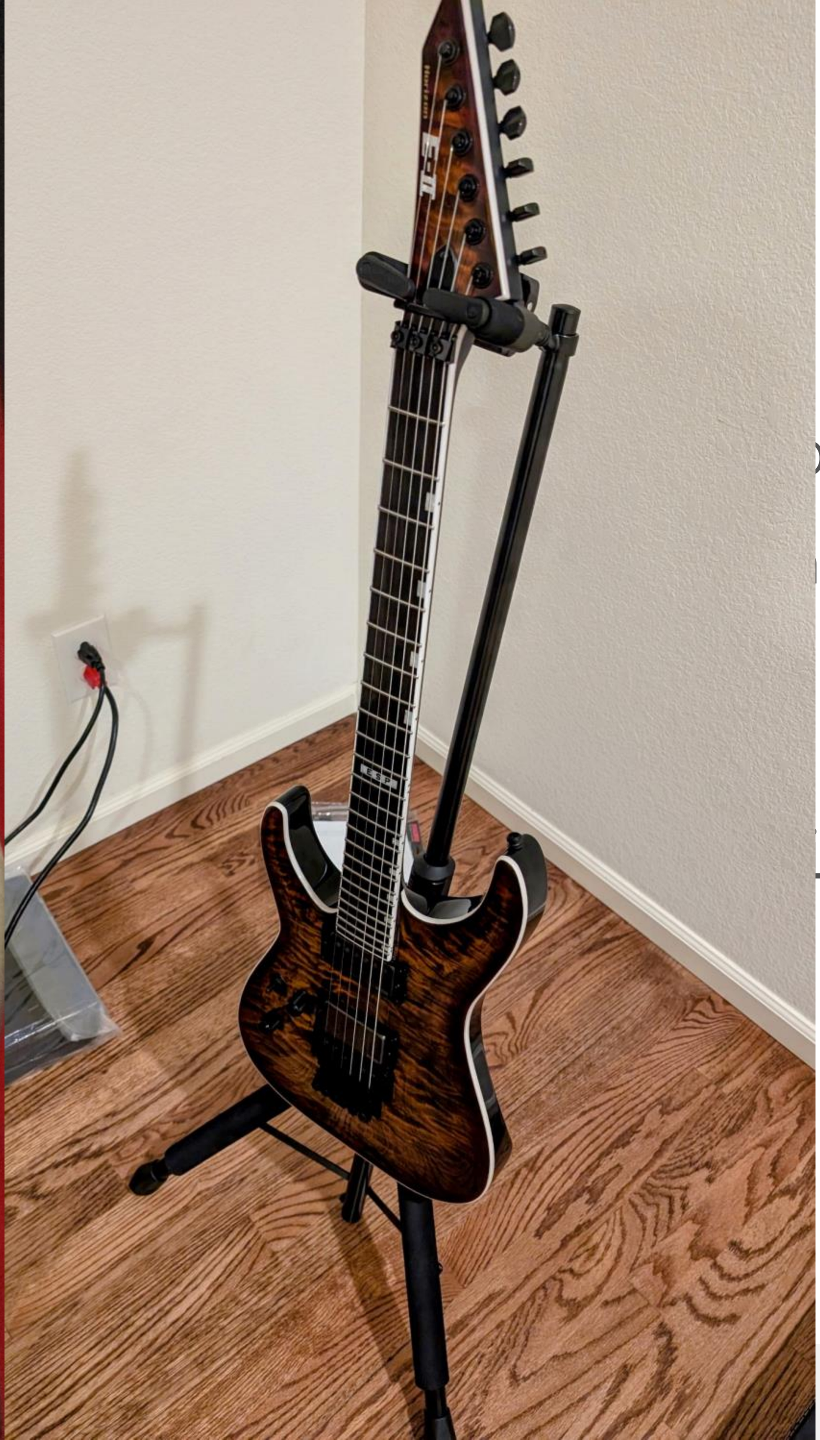
++ (oneDPL) lead developer

ns to other threading engines including
ding Blocks (oneTBB)

n the past)

andard including `std::simd`,
more

Parallelism co-chair

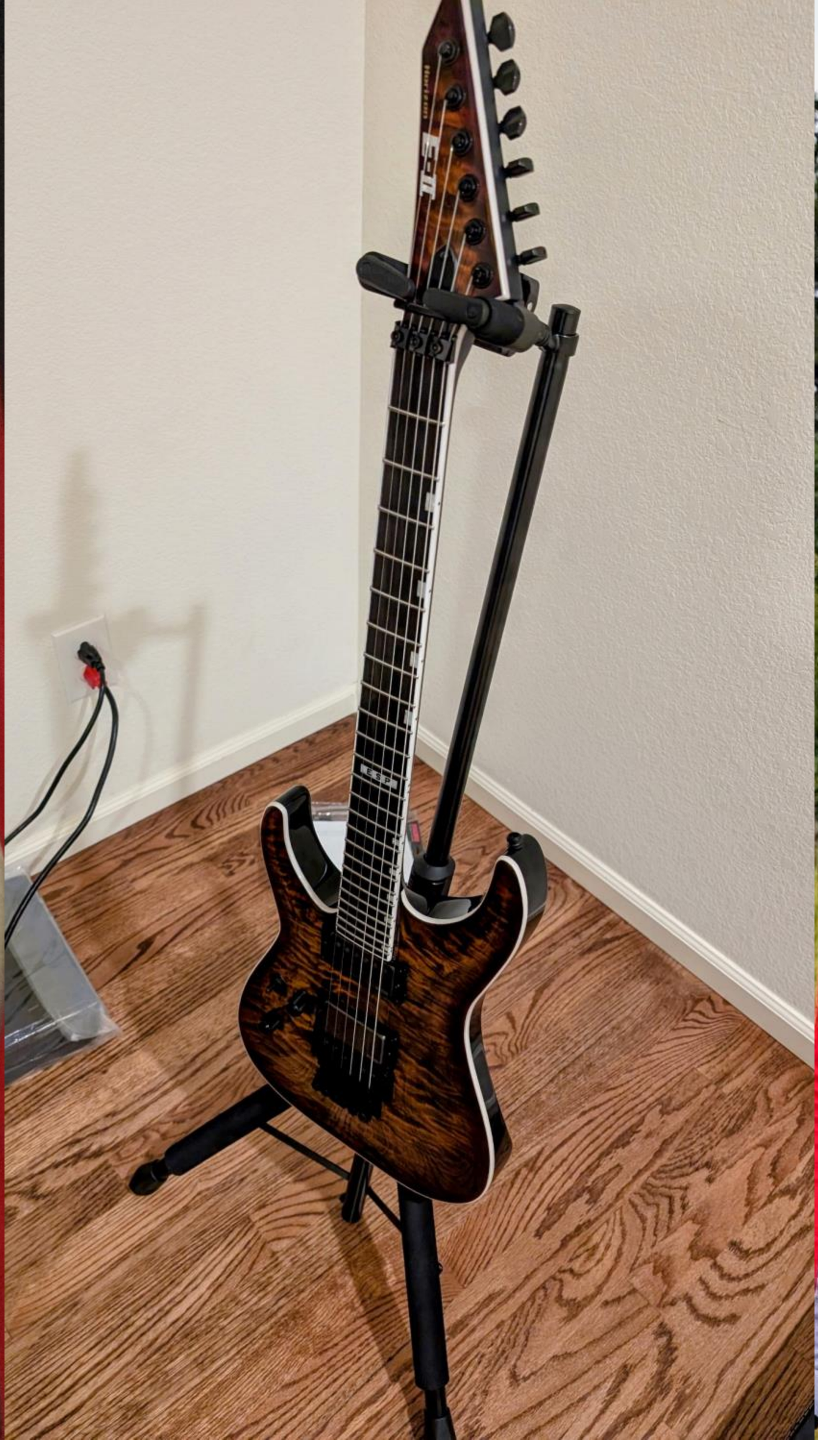


per

es including

md,

DAM


ESP

THE WORLD'S FASTEST

Parallel Range Algorithms

Parallel Algorithms (C++17)

Algorithms:

- With the first `ExecutionPolicy` template parameter
- In `std` namespace
- Taking iterators

Parallel Algorithms (C++17)

Algorithms:

- With the first `ExecutionPolicy` template parameter
- In `std` namespace
- Taking iterators

```
// serial
auto res = std::find_if(std::begin(input), std::end(input), pred);

// parallel
auto res = std::find_if(std::execution::par, std::begin(input), std::end(input), pred);
```

Parallel Range algorithms (P3179 proposal)

Algorithms:

- With the first template parameter constrained by the ***`execution-policy`*** concept
- In **`ranges`** namespace
- Taking
  - ranges
  - iterators and sentinels

Parallel Range algorithms (P3179 proposal)

Algorithms:

- With the first template parameter constrained by the ***`execution-policy`*** concept
- In **`ranges`** namespace
- Taking
  - ranges
  - iterators and sentinels

```
// serial
auto res = std::ranges::find_if(input, pred);

// parallel with P3179
auto res = std::ranges::find_if(std::execution::par, input, pred);
```

Motivation

Combining the powerful ranges API with parallelism:

- Opportunity to fuse several parallel algorithm invocations into one
- Better expressiveness and productivity for parallel code
- Ease of use

Example with C++17 Parallel Algorithms

```
std::transform(policy, std::begin(data), std::end(data), std::begin(result),
               [](auto i){ return i + 1; });
std::reverse(policy, std::begin(result), std::end(result));
auto res = std::find_if(policy, std::begin(result), std::end(result), pred);
```

Example with C++17 Parallel Algorithms

```
std::transform(policy, std::begin(data), std::end(data), std::begin(result),
               [](auto i){ return i + 1; });
std::reverse(policy, std::begin(result), std::end(result));
auto res = std::find_if(policy, std::begin(result), std::end(result), pred);
```

- Three algorithm invocations, each invocation adds its own overhead
- The unnecessary work might be skipped only for the third algorithm call

Example with fancy iterators and Parallel Algorithms

```
auto res = std::find_if(policy,
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::end(data), [](auto i){ return i + 1; })),
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::begin(data), [](auto i){ return i + 1; })),
             pred);
```

Example with fancy iterators and Parallel Algorithms

```
auto res = std::find_if(policy,
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::end(data), [](auto i){ return i + 1; })),
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::begin(data), [](auto i){ return i + 1; })),
             pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped

Example with fancy iterators and Parallel Algorithms

```
auto res = std::find_if(policy,
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::end(data), [](auto i){ return i + 1; })),
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::begin(data), [](auto i){ return i + 1; })),
             pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped

But:

- A lot of verbosity
- `end(data)` is a `reverse_iterator` begin

Example with fancy iterators and Parallel Algorithms

```
auto res = std::find_if(policy,
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::end(data), [](auto i){ return i + 1; })),
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::begin(data), [](auto i){ return i + 1; })),
             pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped

But:

- A lot of verbosity
- `end(data)` is a `reverse_iterator` begin
- The code does not compile as is

Example with fancy iterators and Parallel Algorithms

```
auto add_one = [](auto i){ return i + 1; };
auto res = std::find_if(policy,
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::end(data), add_one)),
             std::make_reverse_iterator(
                    dpl::make_transform_iterator(std::begin(data), add_one)),
             pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped

But:

- A lot of verbosity
- `end(data)` is a `reverse_iterator` begin

Example with C++17 parallel algorithms and ranges

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::find_if(policy, std::ranges::begin(pipeline), std::ranges::end(pipeline),
                        pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped

Example with C++17 parallel algorithms and ranges

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::find_if(policy, std::ranges::begin(pipeline), std::ranges::end(pipeline),
                        pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped

But:

- Still unnecessary verbosity

Example with P3179

```
auto res = std::ranges::find_if(policy,
                 data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse,
                 pred);
```

Example with P3179

```
auto res = std::ranges::find_if(policy,
                 data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse,
                 pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped
- Concise

Example with P3179

```
auto res = std::ranges::find_if(policy,
                 data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse,
                 pred);



*res;   // compile-time error because decltype(res) is std::ranges::dangling
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped
- Concise

But:

- `res` is unusable since `transform_view` is not a `borrowed_range`

Example with P3179

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::ranges::find_if(policy, pipeline, pred);
```

Example with P3179

```cpp
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::ranges::find_if(policy, pipeline, pred);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped
- Concise

Example with P3179

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::ranges::find_if(policy, pipeline, pred, proj);
```

- One algorithm invocation, less overhead for parallelism
- The unnecessary work might be skipped
- Concise
- Ability to use projections

Key differences to existing algorithms

Key differences to existing algorithms

Comparing to the existing algorithms, we propose the following modifications:

a) The execution policy parameter is added

Key differences to existing algorithms

Comparing to the existing algorithms, we propose the following modifications:

a) The execution policy parameter is added

b) Parallel algorithms require `random_access_{iterator,range}`

Key differences to existing algorithms

Comparing to the existing algorithms, we propose the following modifications:

a) The execution policy parameter is added

b) Parallel algorithms require `random_access_{iterator,range}`

c) Parallel algorithms require sized ranges

Key differences to existing algorithms

Comparing to the existing algorithms, we propose the following modifications:

a) The execution policy parameter is added

b) Parallel algorithms require `random_access_{iterator,range}`

c) Parallel algorithms require sized ranges

d) Parallel range algorithms take a range, not an iterator, as the output for the overloads with ranges, and additionally take an output sentinel for the "iterator and sentinel" overloads

Starting from the serial signature

```
template<std::input_iterator I, std::sentinel_for<I> S, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(I first1, S last1, O result, F op, Proj proj = {});
```

```
template<ranges::input_range R, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O,
    std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
unary_transform_result<ranges::borrowed_iterator_t<R>, O>
  transform(R&& r, O result, F op, Proj proj = {});
```

a) Adding an execution policy parameter

```
template<execution-policy Ep, std::input_iterator I, std::sentinel_for<I> S,
         std::weakly_incrementable O, std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(Ep&& exec, I first1, S last1, O result, F op, Proj proj = {});
```

```
template<execution-policy Ep, ranges::input_range R, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O,
    std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
unary_transform_result<ranges::borrowed_iterator_t<R>, O>
  transform(Ep&& exec, R&& r, O result, F op, Proj proj = {});
```

```
template<class Ep>
concept execution-policy = // exposition only
  std::is_execution_policy_v<std::remove_cvref_t<Ep>>;
```

b) Requiring random access

```
template<execution-policy Ep, std::random_access_iterator I, std::sentinel_for<I> S,
         std::random_access_iterator O, std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(Ep&& exec, I first1, S last1, O result, F op, Proj proj = {});
```

```
template<execution-policy Ep, ranges::random_access_range R, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O,
    std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
unary_transform_result<ranges::borrowed_iterator_t<R>, O>
  transform(Ep&& exec, R&& r, O result, F op, Proj proj = {});
```

c) Requiring sized range

```
template<execution-policy Ep, std::random_access_iterator I, std::sized_sentinel_for<I> S,
         std::random_access_iterator O, std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(Ep&& exec, I first1, S last1, O result, F op, Proj proj = {});
```

```
template<execution-policy Ep, sized-random-access-range R, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O,
    std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
unary_transform_result<ranges::borrowed_iterator_t<R>, O>
  transform(Ep&& exec, R&& r, O result, F op, Proj proj = {});
```

```
template<class R>
concept sized-random-access-range = // exposition only
    ranges::random_access_range<R> && ranges::sized_range<R>;
```

d) Using a range for the output

```
template<execution-policy Ep, std::random_access_iterator I, std::sized_sentinel_for<I> S,
         std::random_access_iterator O, std::sized_sentinel_for<O> OutS,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(Ep&& exec, I first1, S last1, O result, OutS result_last, F op, Proj proj = {});
```

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<ranges::iterator_t<OutR>,
    std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
unary_transform_result<ranges::borrowed_iterator_t<R>, ranges::borrowed_iterator_t<OutR>>
  transform(Ep&& exec, R&& r, OutR&& result_r, F op, Proj proj = {});
```

Comparing side-by-side (Iterator and Sentinel overload)

```
// serial
template<std::input_iterator I, std::sentinel_for<I> S, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(I first1, S last1, O result, F op, Proj proj = {});



// parallel
template<execution-policy Ep, std::random_access_iterator I, std::sized_sentinel_for<I> S,
         std::random_access_iterator O, std::sized_sentinel_for<O> OutS,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O, std::indirect_result_t<F&, std::projected<I, Proj>>>
unary_transform_result<I, O>
  transform(Ep&& exec, I first1, S last1, O result, OutS result_last, F op, Proj proj = {});
```

Comparing side-by-side (range overload)

```
// serial
template<ranges::input_range R, std::weakly_incrementable O,
         std::copy_constructible F, class Proj = std::identity>
  requires std::indirectly_writable<O,
      std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
ranges::unary_transform_result<ranges::borrowed_iterator_t<R>, O>
  ranges::transform(R&& r, O result, F op, Proj proj = {});
```

```
// parallel
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR,
         std::copy_constructible F, class Proj = std::identity>
  requires indirectly_writable<ranges::iterator_t<OutR>,
      std::indirect_result_t<F&, std::projected<ranges::iterator_t<R>, Proj>>>
ranges::unary_transform_result<ranges::borrowed_iterator_t<R>, ranges::borrowed_iterator_t<OutR>>
  ranges::transform(Ep&& exec, R&& r, OutR&& result_r, F op, Proj proj = {});
```

More about design

sized-random-access-range

- C++17 parallel algorithms require *Cpp17ForwardIterator*
  - Intel oneDPL, Nvidia Thrust, GNU libstdc++ implementations are based on random access
  - Only Microsoft STL supports forward iterators*

sized-random-access-range

- C++17 parallel algorithms require *Cpp17ForwardIterator*
  - Intel oneDPL, Nvidia Thrust, GNU libstdc++ implementations are based on random access
  - Only Microsoft STL supports forward iterators*
- Random access is the best abstraction in the standard (for now)
  - Some potentially useful views are not supported (e.g., `filter_view`)
  - Might be relaxed in the future

sized-random-access-range

- C++17 parallel algorithms require *Cpp17ForwardIterator*
  - Intel oneDPL, Nvidia Thrust, GNU libstdc++ implementations are based on random access
  - Only Microsoft STL supports forward iterators*
- Random access is the best abstraction in the standard (for now)
  - Some potentially useful views are not supported (e.g., `filter_view`)
  - Might be relaxed in the future
- Size is necessary to be known in advance for parallelization
  - memory safety: everything is bounded, including the output
  - performance: no need to do unnecessary work

Range-as-the-output

`copy[_n]`, `move`, `transform`, `replace_copy`, `replace_copy_if`, `merge`, `partial_sort_copy`, `uninitialized_copy[_n]`, `uninitialized_move[_n]`

Range-as-the-output

- Pros:
  - Ease of use
  - Better memory safety
  - Potentially better performance
  - Ability to detect an error

Range-as-the-output

- Pros:
  - Ease of use
  - Better memory safety
  - Potentially better performance
  - Ability to detect an error
- The objection was:
  - More complicated switch between serial and parallel algorithms
  - Inconsistency with serial range algorithms
  - Unclear semantics

Evolution of algorithm input

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

Evolution of algorithm input

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

template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2,
         weakly_incrementable O, copy_constructible F, class Proj1 = identity,
         class Proj2 = identity>
  requires indirectly_writable<O, indirect_result_t<F&, projected<I1, Proj1>, projected<I2, Proj2>>>
constexpr ranges::binary_transform_result<I1, I2, O>
  ranges::transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                    F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});

template<input_range R1, input_range R2, weakly_incrementable O,
         copy_constructible F, class Proj1 = identity, class Proj2 = identity>
  requires indirectly_writable<O, indirect_result_t<F&, projected<iterator_t<R1>, Proj1>,
                               projected<iterator_t<R2>, Proj2>>>
constexpr ranges::binary_transform_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O>
  ranges::transform(R1&& r1, R2&& r2, O result, F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});
```

Output inconsistency

```
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
constexpr ranges::copy_result<I, O> ranges::copy(I first, S last, O result);

template<input_range R, weakly_incrementable O>
  requires indirectly_copyable<iterator_t<R>, O>
constexpr ranges::copy_result<borrowed_iterator_t<R>, O> ranges::copy(R&& r, O result);
```

Output inconsistency

```
template<input_iterator I, sentinel_for<I> S, weakly_incrementable O>
  requires indirectly_copyable<I, O>
constexpr ranges::copy_result<I, O> ranges::copy(I first, S last, O result);

template<input_range R, weakly_incrementable O>
  requires indirectly_copyable<iterator_t<R>, O>
constexpr ranges::copy_result<borrowed_iterator_t<R>, O> ranges::copy(R&& r, O result);


template<input_iterator I, sentinel_for<I> S1, nothrow-forward-iterator O, nothrow-sentinel-for<O> S2>
  requires constructible_from<iter_value_t<O>, iter_reference_t<I>>
constexpr uninitialized_copy_result<I, O>
  uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);

template<input_range IR, nothrow-forward-range OR>
  requires constructible_from<range_value_t<OR>, range_reference_t<IR>>
constexpr uninitialized_copy_result<borrowed_iterator_t<IR>, borrowed_iterator_t<OR>>
  uninitialized_copy(IR&& in_range, OR&& out_range);
```

Unclear semantics

```
std::vector<int> v1{1,2,3,4,5};
std::vector<int> v2(3);
std::ranges::copy(v1, v2); // might appear that copy allocates for v2
```

Addressing unclear semantics

```
std::vector<int> v1{1,2,3,4,5};
std::vector<int> v2(3);
std::ranges::copy(v1, v2); // might appear that copy allocates for v2
```

Our proposal: Execute an algorithm until any of the ranges ends

Addressing unclear semantics

```
std::vector<int> v1{1,2,3,4,5};
std::vector<int> v2(3);
std::ranges::copy(v1, v2); // might appear that copy allocates for v2
```

Our proposal: Execute an algorithm until any of the ranges ends

Algorithm with the same semantics:

- uninitialized_copy

Addressing unclear semantics

```
std::vector<int> v1{1,2,3,4,5};
std::vector<int> v2(3);
std::ranges::copy(v1, v2); // might appear that copy allocates for v2
```

Our proposal: Execute an algorithm until any of the ranges ends

Algorithms with the same semantics:

- uninitialized_copy
- uninitialized_move
- partial_sort_copy

Range-as-the-output

- Pros:
  - Ease of use
  - Better memory safety
  - Potentially better performance
  - Ability to detect an error
- The objection was:
  - More complicated switch between serial and parallel algorithms
  - Inconsistency with serial range algorithms
  - Unclear semantics

Range-as-the-output

- Pros:
  - Ease of use
  - Better memory safety
  - Potentially better performance
  - Ability to detect an error
- The objection was:
  - More complicated switch between serial and parallel algorithms
  - ~~Inconsistency with serial range algorithms~~
  - ~~Unclear semantics~~

Range-as-the-output

- Pros:
  - Ease of use
  - Better memory safety
  - Potentially better performance
  - Ability to detect an error
- The objection was:
  - More complicated switch between serial and parallel algorithms
  - ~~Inconsistency with serial range algorithms~~
  - ~~Unclear semantics~~

P3490 proposal has more detail about range-as-the-output design aspect

Algorithms with output and gaps

`copy_if`, `remove_copy`, `remove_copy_if`, `unique_copy`, `partition_copy`, `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`

`copy_if` parallel signature

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR,
         class Proj = std::identity,
         std::indirect_unary_predicate<std::projected<std::ranges::iterator_t<R>, Proj>> Pred>
  requires std::indirectly_copyable<std::ranges::iterator_t<R>, std::ranges::iterator_t<OutR>>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {})
{



}
```

For demonstration purposes only

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR,
         class Proj = std::identity,
         std::indirect_unary_predicate<std::projected<std::ranges::iterator_t<R>, Proj>> Pred>
  requires std::indirectly_copyable<std::ranges::iterator_t<R>, std::ranges::iterator_t<OutR>>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {})
{
    std::size_t size = std::ranges::size(r);

    std::ranges::for_each(exec, std::views::iota(std::size_t(0), size), [=, &r, &result_r](auto i) {
        if (std::invoke(pred, std::invoke(proj, r[i])))
            result_r[i] = r[i];
    });

    return {std::ranges::begin(r) + size, std::ranges::begin(result_r) + size};
}
```

`copy_if` parallel implementation (wrong)

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR,
         class Proj = std::identity,
         std::indirect_unary_predicate<std::projected<std::ranges::iterator_t<R>, Proj>> Pred>
  requires std::indirectly_copyable<std::ranges::iterator_t<R>, std::ranges::iterator_t<OutR>>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {})
{
    std::size_t size = std::ranges::size(r);

    std::ranges::for_each(exec, std::views::iota(std::size_t(0), size), [=, &r, &result_r](auto i) {
        if (std::invoke(pred, std::invoke(proj, r[i])))
            result_r[i] = r[i];
    });

    return {std::ranges::begin(r) + size, std::ranges::begin(result_r) + size};
}

// Input:   {1,7,4,4,4,6,5,2,3,7,9}, pred: [](auto x) { return (x & 0x01) == 1; }
// Output:  {0,0,0,0,0,0,0,0,0,0,0}
// Result:  {1,7,0,0,0,0,5,0,3,7,9}
```

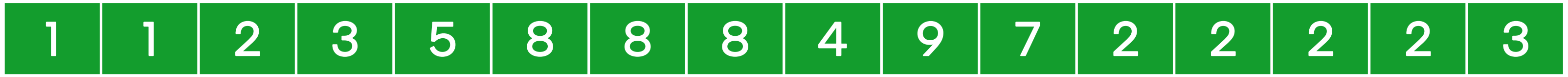
`copy_if` parallel implementation (wrong)

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR,
         class Proj = std::identity,
         std::indirect_unary_predicate<std::projected<std::ranges::iterator_t<R>, Proj>> Pred>
  requires std::indirectly_copyable<std::ranges::iterator_t<R>, std::ranges::iterator_t<OutR>>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {})
{
    std::size_t size = std::ranges::size(r);

    std::ranges::for_each(exec, std::views::iota(std::size_t(0), size), [=, &r, &result_r](auto i) {
        if (std::invoke(pred, std::invoke(proj, r[i])))
            result_r[i] = r[i];
    });

    return {std::ranges::begin(r) + size, std::ranges::begin(result_r) + size};
}

// Input:   {1,7,4,4,4,6,5,2,3,7,9}, pred: [](auto x) { return (x & 0x01) == 1; }
// Output:  {0,0,0,0,0,0,0,0,0,0,0}
// Result:  {1,7,0,0,0,0,5,0,3,7,9}
// Correct: {1,7,5,3,7,9,0,0,0,0,0}
```

For demonstration purposes only

`copy_if` example

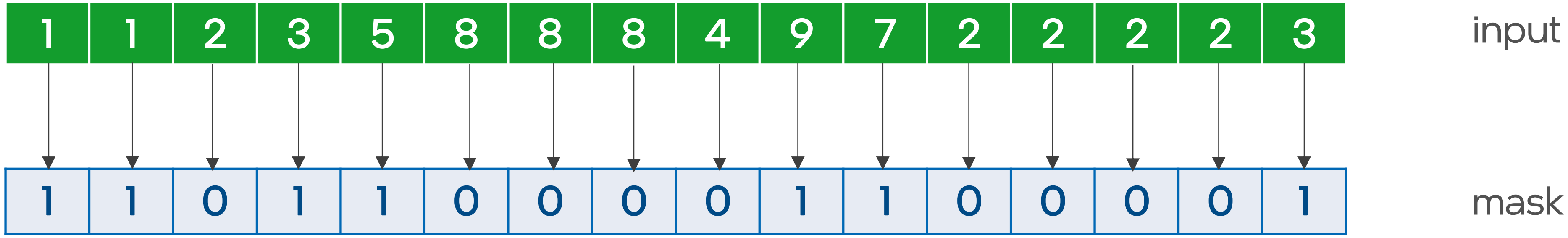
```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(data.size());

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3	input

copy_if example

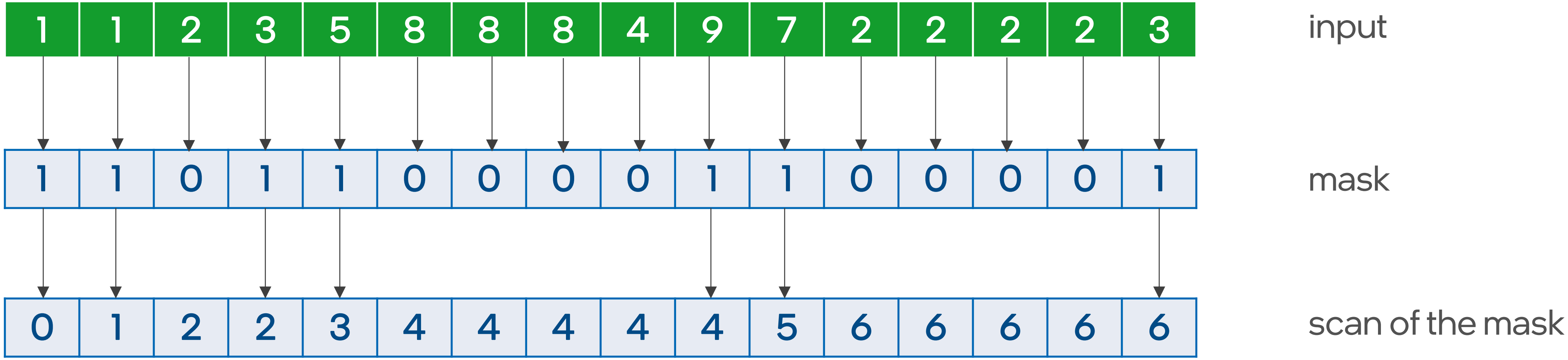
```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(data.size());

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3	input
1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	mask

copy_if example

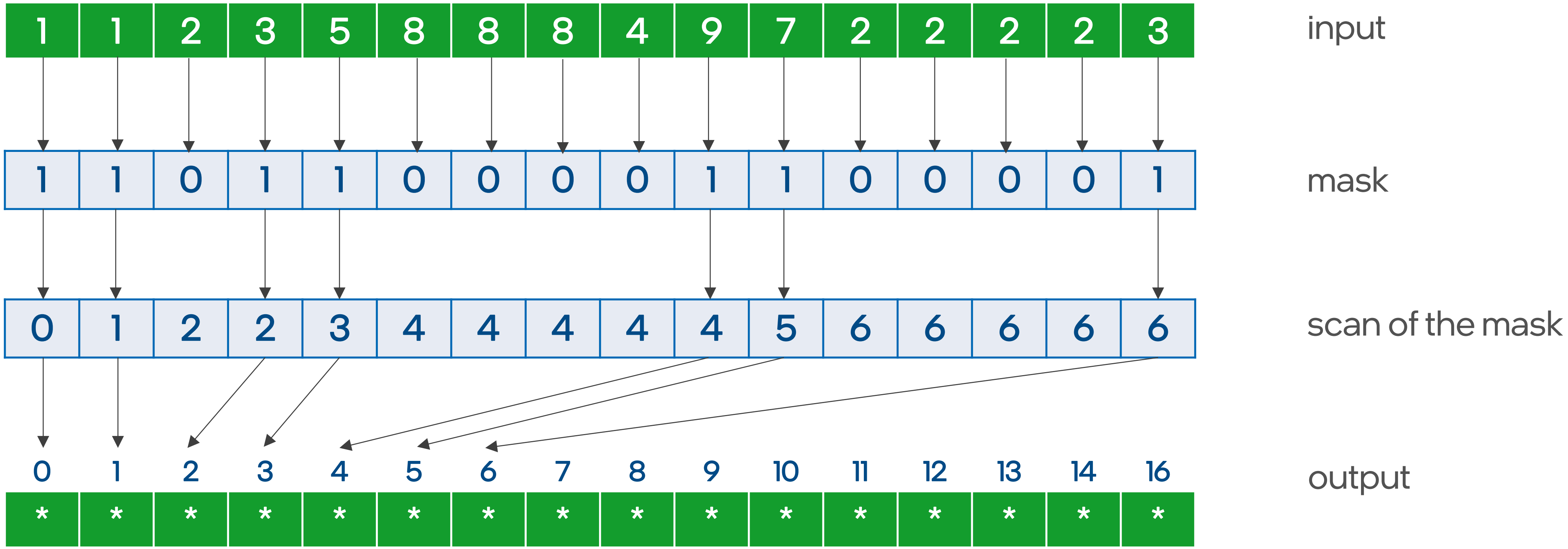
```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(data.size());

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3	input
1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	mask
0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6	scan of the mask

copy_if example

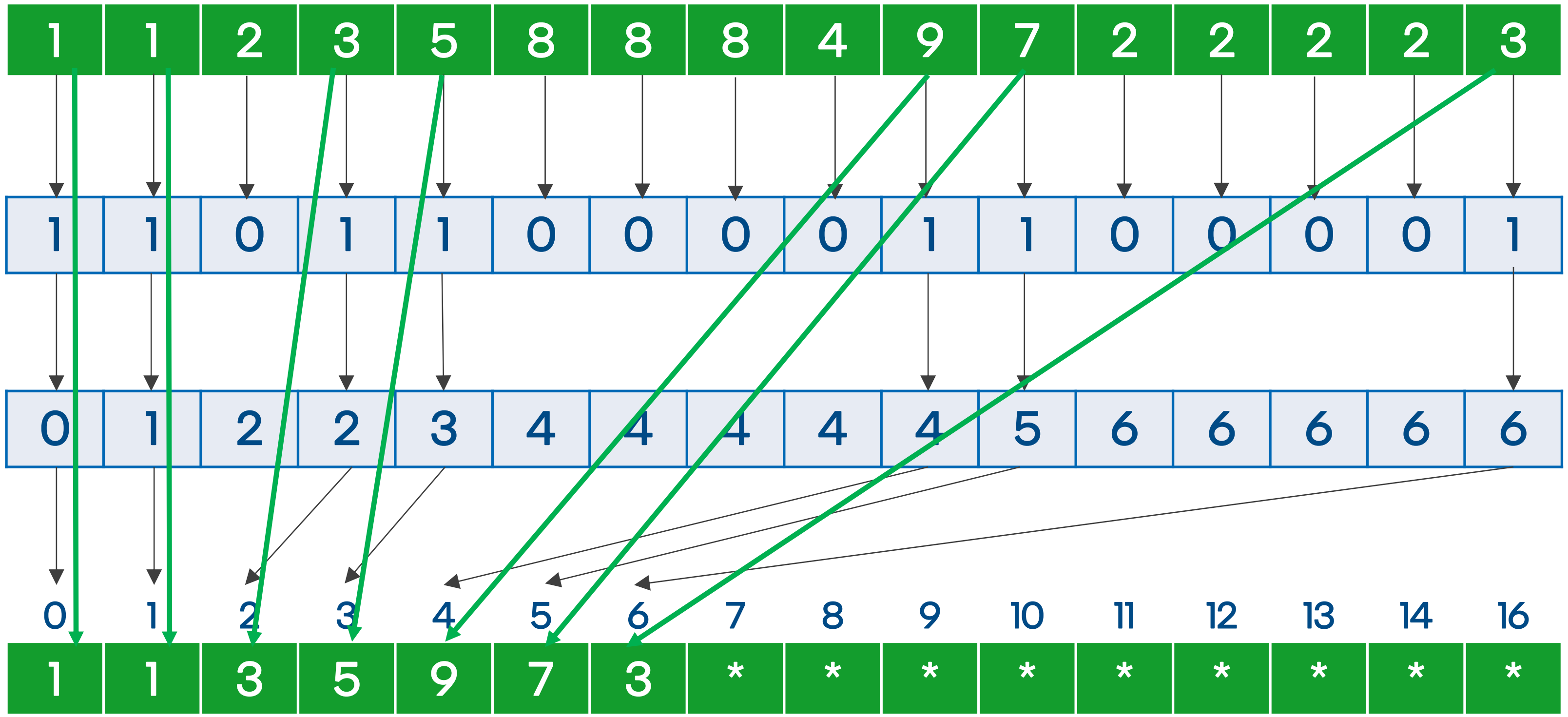
```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(data.size());

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3	input
1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	mask
0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6	scan of the mask
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	output
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	

`copy_if` example

```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(data.size());

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



input	1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3
mask	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1
scan of the mask	0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16
output	1	1	3	5	9	7	3	*	*	*	*	*	*	*	*	*

`copy_if` parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {



















}
```

For demonstration purposes only

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));


    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });




}
```

For demonstration purposes only

`copy_if` parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });

    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);



}
```

For demonstration purposes only

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });

    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);




    auto zip = std::views::zip(r, mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });


}
```

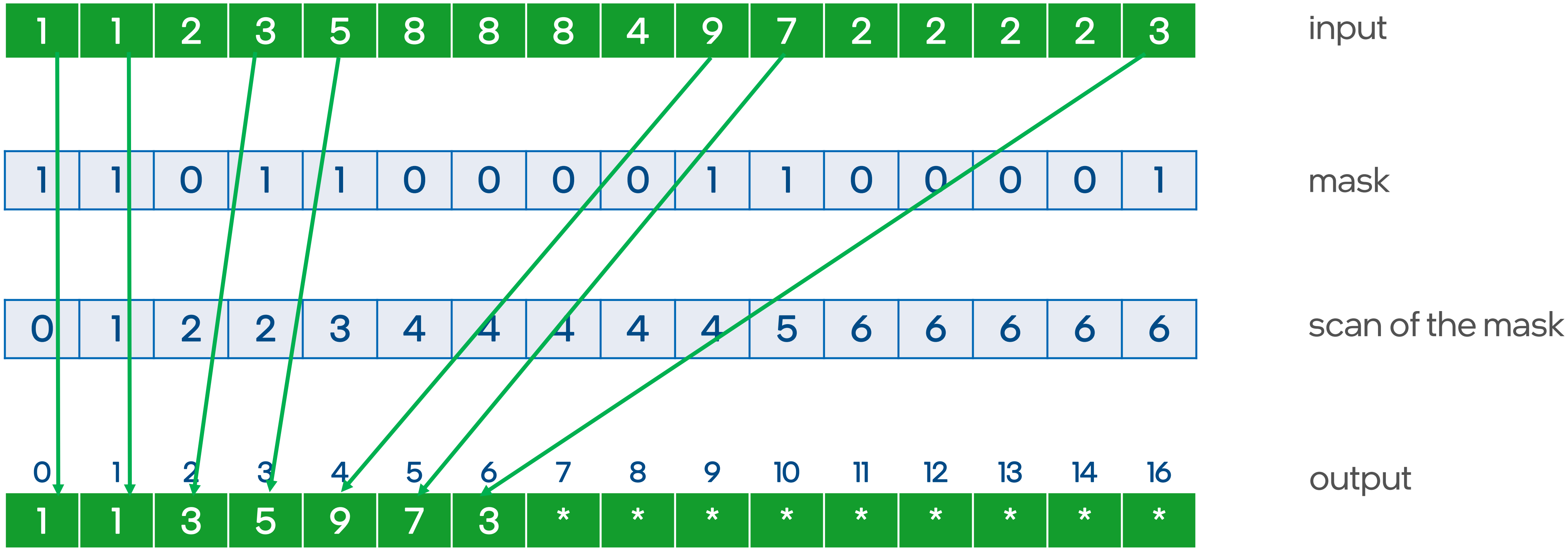
copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
 copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
  std::vector<std::size_t> mask(std::ranges::size(r));
  std::vector<std::size_t> scan_result(std::ranges::size(r));

  std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
    return std::size_t(std::invoke(pred, std::invoke(proj, x))); });

  std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);




  auto zip = std::views::zip(r, mask, scan_result);
  std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
    auto [in, mask, scan_result] = tuple;
    if (mask == 1) result_r[scan_result] = in;
  });
  auto copied_elements = scan_result.back() + mask.back();
  return {std::ranges::begin(r) + std::ranges::size(r), std::ranges::begin(result_r) + copied_elements};
}
```

For demonstration purposes only

copy_if example

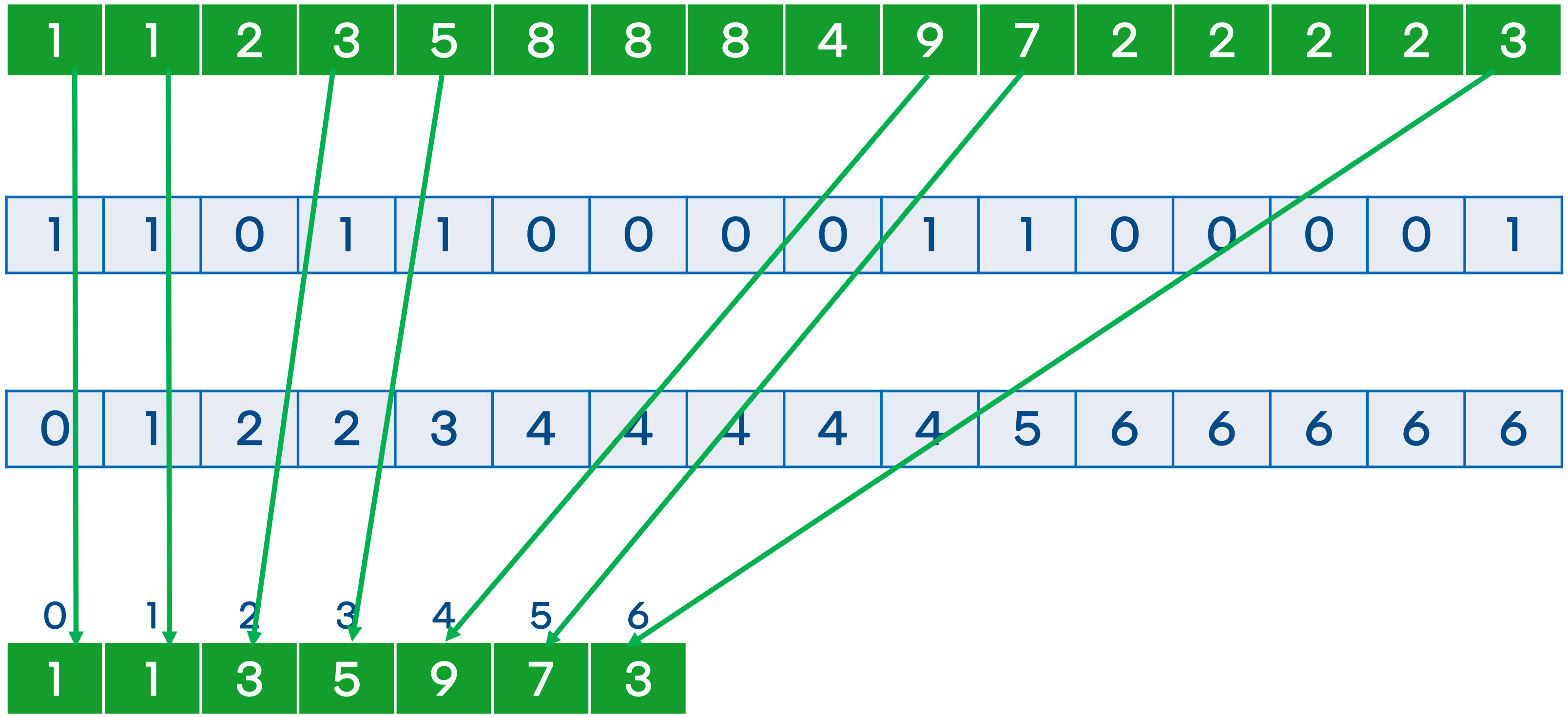
```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(data.size());

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3	input
1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	mask
0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6	scan of the mask
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	16	
1	1	3	5	9	7	3	*	*	*	*	*	*	*	*	*	output

`copy_if` example

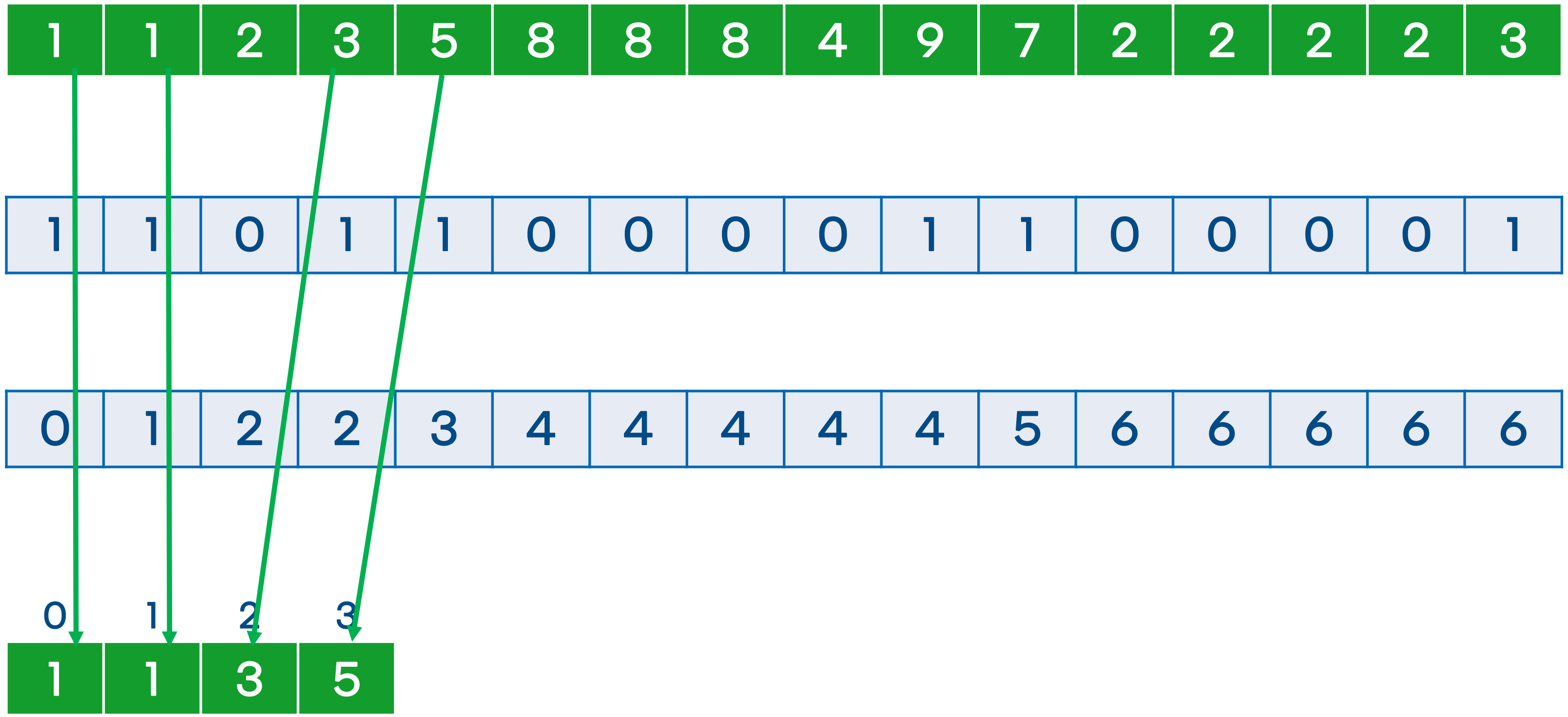
```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(7);

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



input	1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3
mask	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1
scan of the mask	0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6
	0	1	2	3	4	5	6									
output	1	1	3	5	9	7	3									

copy_if example

```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(4);

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```

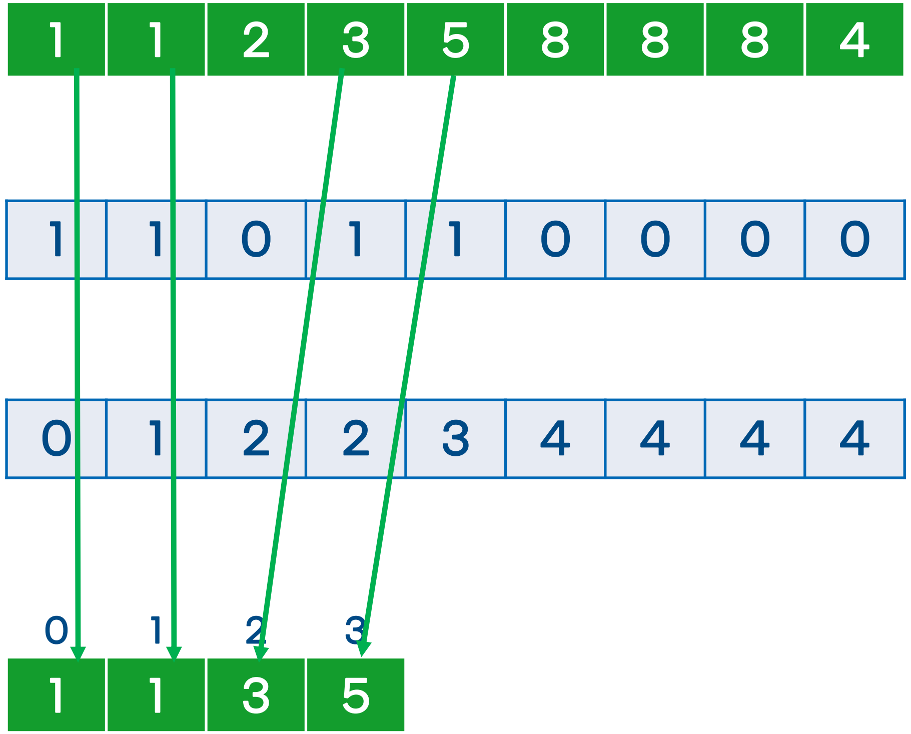


input	1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3
mask	1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1
scan of the mask	0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6

	0	1	2	3
output	1	1	3	5

`copy_if` example

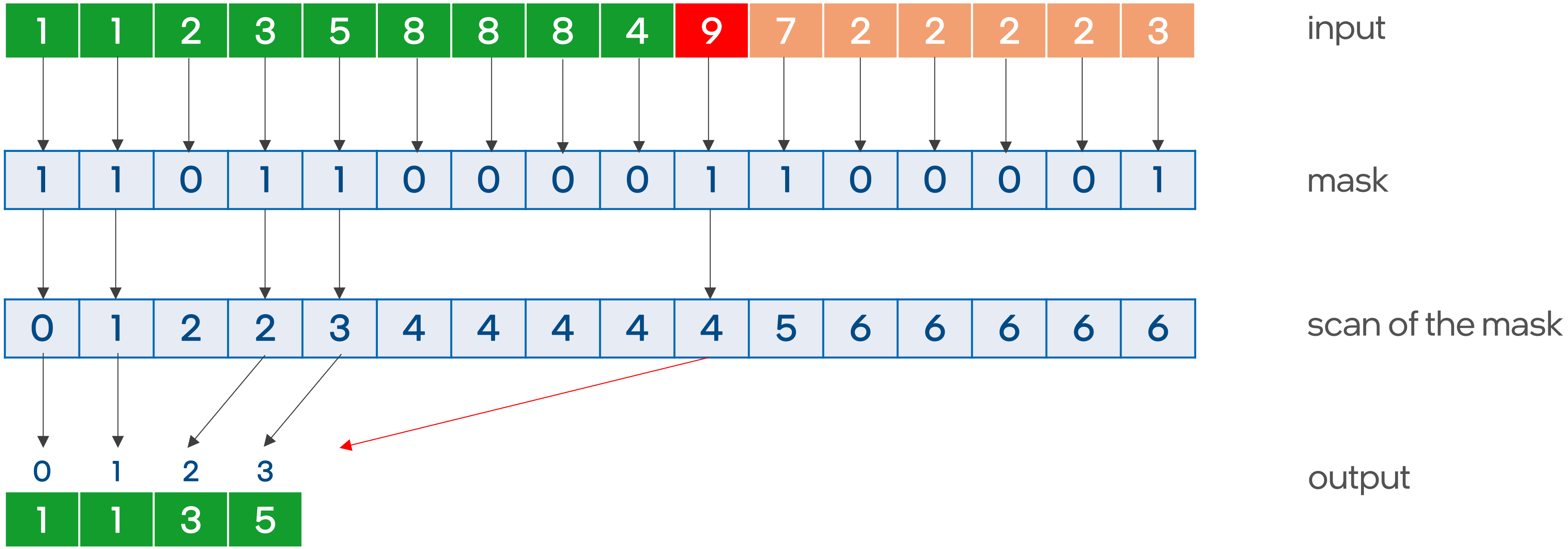
```
std::vector data{1,1,2,3,5,8,8,8,4};
std::vector<int> out(4);

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



input	1	1	2	3	5	8	8	8	4
mask	1	1	0	1	1	0	0	0	0
scan of the mask	0	1	2	2	3	4	4	4	4
	0	1	2	3					
output	1	1	3	5					

`copy_if` example

```
std::vector data{1,1,2,3,5,8,8,8,4,9,7,2,2,2,2,3};
std::vector<int> out(4);

auto res = std::ranges::copy_if(std::execution::par, data, out, [](auto x) { return (x & 0x01) == 1; });
```



1	1	2	3	5	8	8	8	4	9	7	2	2	2	2	3	input
1	1	0	1	1	0	0	0	0	1	1	0	0	0	0	1	mask
0	1	2	2	3	4	4	4	4	4	5	6	6	6	6	6	scan of the mask
0	1	2	3													
1	1	3	5													output

`copy_if` parallel implementation

```cpp
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });
    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);




    auto zip = std::views::zip(r, mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });
    auto copied_elements = scan_result.back() + mask.back();
    return {std::ranges::begin(r) + std::ranges::size(r), std::ranges::begin(result_r) + copied_elements};
}
```

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });
    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);

    auto last_iterator = std::ranges::upper_bound(scan_result, std::ranges::size(result_r)) - 1;




    auto zip = std::views::zip(r, mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });
    auto copied_elements = scan_result.back() + mask.back();
    return {std::ranges::begin(r) + std::ranges::size(r), std::ranges::begin(result_r) + copied_elements};
}
```

For demonstration purposes only

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });
    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);

    auto last_iterator = std::ranges::upper_bound(scan_result, std::ranges::size(result_r)) - 1;
    bool enough_space = std::ranges::size(result_r) > scan_result.back();



    auto zip = std::views::zip(r, mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });
    auto copied_elements = scan_result.back() + mask.back();
    return {std::ranges::begin(r) + std::ranges::size(r), std::ranges::begin(result_r) + copied_elements};
}
```

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });
    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);

    auto last_iterator = std::ranges::upper_bound(scan_result, std::ranges::size(result_r)) - 1;
    bool enough_space = std::ranges::size(result_r) > scan_result.back();
    last_iterator += std::size_t(enough_space || (*last_iterator == scan_result.back() && mask.back() == 0));


    auto zip = std::views::zip(r, mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });
    auto copied_elements = scan_result.back() + mask.back();
    return {std::ranges::begin(r) + std::ranges::size(r), std::ranges::begin(result_r) + copied_elements};
}
```

For demonstration purposes only

`copy_if` parallel implementation

```cpp
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });
    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);

    auto last_iterator = std::ranges::upper_bound(scan_result, std::ranges::size(result_r)) - 1;
    bool enough_space = std::ranges::size(result_r) > scan_result.back();
    last_iterator += std::size_t(enough_space || (*last_iterator == scan_result.back() && mask.back() == 0));
    std::size_t distance = std::ranges::distance(scan_result.begin(), last_iterator);

    auto zip = std::views::zip(r | std::views::take(distance), mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });
    auto copied_elements = scan_result.back() + mask.back();
    return {std::ranges::begin(r) + std::ranges::size(r), std::ranges::begin(result_r) + copied_elements};
}
```

For demonstration purposes only

copy_if parallel implementation

```
template<execution-policy Ep, sized-random-access-range R, sized-random-access-range OutR, //, Pred, Proj>
std::ranges::copy_if_result<std::ranges::borrowed_iterator_t<R>, std::ranges::borrowed_iterator_t<OutR>>
  copy_if(Ep&& exec, R&& r, OutR&& result_r, Pred pred, Proj proj = {}) {
    std::vector<std::size_t> mask(std::ranges::size(r));
    std::vector<std::size_t> scan_result(std::ranges::size(r));

    std::ranges::transform(exec, r, mask, [pred = std::move(pred), proj = std::move(proj)](auto x) {
        return std::size_t(std::invoke(pred, std::invoke(proj, x))); });
    std::exclusive_scan(exec, mask.begin(), mask.end(), scan_result.begin(), 0);

    auto last_iterator = std::ranges::upper_bound(scan_result, std::ranges::size(result_r)) - 1;
    bool enough_space = std::ranges::size(result_r) > scan_result.back();
    last_iterator += std::size_t(enough_space || (*last_iterator == scan_result.back() && mask.back() == 0));
    std::size_t distance = std::ranges::distance(scan_result.begin(), last_iterator);

    auto zip = std::views::zip(r | std::views::take(distance), mask, scan_result);
    std::ranges::for_each(exec, zip, [&result_r](auto tuple) {
        auto [in, mask, scan_result] = tuple;
        if (mask == 1) result_r[scan_result] = in;
    });
    auto copied_elements = scan_result[distance - 1] + mask[distance - 1];
    return {std::ranges::begin(r) + distance, std::ranges::begin(result_r) + copied_elements};
}
```

For demonstration purposes only

Heterogeneity and performance

Heterogeneity with oneDPL

oneAPI DPC++ library (oneDPL):

- Implementation of standard Parallel Algorithms made by Intel
- Evolution of former Parallel STL
  - Donated to LLVM, used as GNU libstc++ parallel algorithms implementation
- Supports heterogeneous execution via SYCL
- Currently a part of UXL Foundation: both specification and source code
- ~35 parallel range algorithms available as a product quality
- ~15 parallel range algorithms coming

Heterogeneous example

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::find_if(std::execution::par, pipeline, pred);
```

Heterogeneous example

```
// Has an associated device
sycl::queue q;
```

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::find_if(std::execution::par, pipeline, pred);
```

Heterogeneous example

```
// Has an associated device
sycl::queue q;

// Accessible memory on both the host and the device
using alloc_type = sycl::usm_allocator<std::size_t, sycl::usm::alloc::shared>;

// Creating an allocator object
alloc_type alloc(q);
```

```
auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::find_if(std::execution::par, pipeline, pred);
```

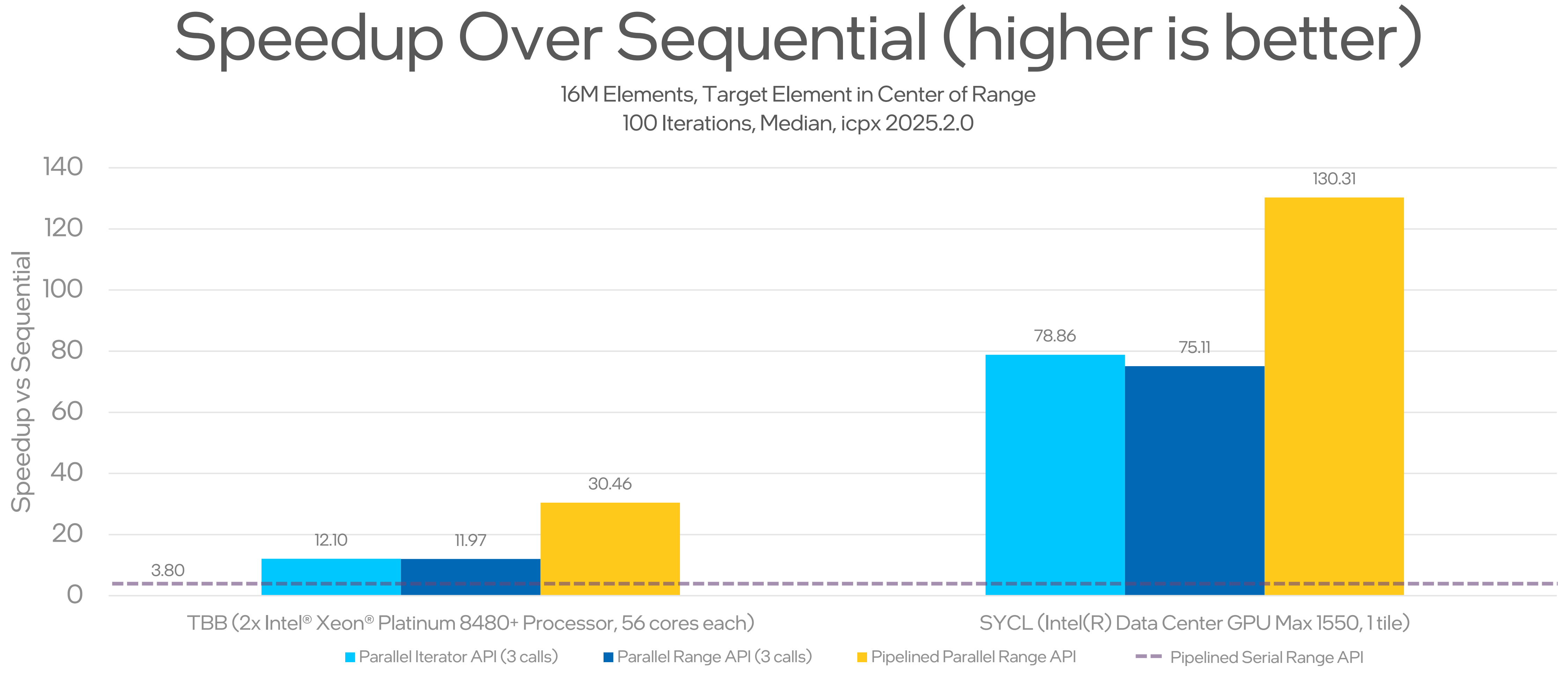
Heterogeneous example

```cpp
// Has an associated device
sycl::queue q;

// Accessible memory on both the host and the device
using alloc_type = sycl::usm_allocator<std::size_t, sycl::usm::alloc::shared>;

// Creating an allocator object
alloc_type alloc(q);

// Creating data
std::vector<std::size_t, alloc_type> v(size, alloc);
std::ranges::subrange data{v.begin(), v.end()};

auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = std::find_if(std::execution::par, pipeline, pred);
```

Heterogeneous example

```cpp
// Has an associated device
sycl::queue q;

// Accessible memory on both the host and the device
using alloc_type = sycl::usm_allocator<std::size_t, sycl::usm::alloc::shared>;

// Creating an allocator object
alloc_type alloc(q);

// Creating data
std::vector<std::size_t, alloc_type> v(size, alloc);
std::ranges::subrange data{v.begin(), v.end()};

auto pipeline = data | std::views::transform([](auto i){ return i + 1; }) | std::views::reverse;
auto res = dpl::find_if(dpl::make_device_policy(q), pipeline, pred);
```

Performance



Scope and status

79 algorithms in namespace std already with policies

all_of	search[_n]	remove_copy	is_sorted	is_heap
any_of	copy[_n]	remove_copy_if	is_sorted_until	is_heap_until
none_of	copy_if	unique	nth_element	min_element
for_each[_n]	move	unique_copy	is_partitioned	max_element
find	swap_ranges	reverse	partition	minmax_element
find_if	transform	reverse_copy	stable_partition	lexicographical_compare
find_if_not	replace	rotate	partition_copy	uninitialized_default_construct[_n]
find_end	replace_if	rotate_copy	merge	uninitialized_value_construct[_n]
find_first_of	replace_copy	shift_left	inplace_merge	uninitialized_copy[_n]
adjacent_find	replace_copy_if	shift_right	includes	uninitialized_move[_n]
count	fill[_n]	sort	set_union	uninitialized_fill[_n]
count_if	generate[_n]	stable_sort	set_intersection	destroy[_n]
mismatch	remove	partial_sort	set_difference	
equal	remove_if	partial_sort_copy	set_symmetric_difference	

oneDPL: implemented + coming

all_of	search[_n]	remove_copy	is_sorted	is_heap
any_of	copy[_n]	remove_copy_if	is_sorted_until	is_heap_until
none_of	copy_if	unique	nth_element	min_element
for_each[_n]	move	unique_copy	is_partitioned	max_element
find	swap_ranges	reverse	partition	minmax_element
find_if	transform	reverse_copy	stable_partition	lexicographical_compare
find_if_not	replace	rotate	partition_copy	uninitialized_default_construct[_n]
find_end	replace_if	rotate_copy	merge	uninitialized_value_construct[_n]
find_first_of	replace_copy	shift_left	inplace_merge	uninitialized_copy[_n]
adjacent_find	replace_copy_if	shift_right	includes	uninitialized_move[_n]
count	fill[_n]	sort	set_union	uninitialized_fill[_n]
count_if	generate[_n]	stable_sort	set_intersection	destroy[_n]
mismatch	remove	partial_sort	set_difference	
equal	remove_if	partial_sort_copy	set_symmetric_difference	

10 algorithms only in namespace `std::ranges`

std::ranges algorithms to add policies	**std algorithms used as the guidance**
contains	*find*
contains_subrange	*search*
find_last	*find*
find_last_if	*find_if*
find_last_if_not	*find_if_not*
starts_with	*mismatch*
ends_with	*equal*
min	*min_element*
max	*max_element*
minmax	*minmax_element*

oneDPL: implemented

std::ranges algorithms to add policies	**std algorithms used as the guidance**
contains	*find*
contains_subrange	*search*
find_last	*find*
find_last_if	*find_if*
find_last_if_not	*find_if_not*
starts_with	*mismatch*
ends_with	*equal*
min	*min_element*
max	*max_element*
minmax	*minmax_element*

Out of scope algorithms

- All algorithms that do not have an `ExecutionPolicy` in their C++17 counterpart
- Algorithms from `<numeric>`
  - No serial `<numeric>` algorithms in `std::ranges` namespace (except `iota`)
- Algorithms only in namespace `std::ranges` other than mentioned before:
  - `fold` algorithm family
  - `generate_random`

P3179 proposal status

Parallel Range Algorithms accepted for C++26!

Further work

- Numeric range algorithms (P3732)
- Synchronous parallel algorithms and Senders/Receivers (P2500)
- Asynchronous parallel algorithms (P3300)
- Stretch goal: Range-as-the-output for serial range algorithms

Special thanks

- Alexey Kukanov

Special thanks

- Alexey Kukanov
- Jonathan Mueller

Special thanks

- Alexey Kukanov
- Jonathan Mueller
- Inbal Levi

Special thanks

- Alexey Kukanov
- Jonathan Mueller
- Inbal Levi
- Jeff Garland and Jonathan Wakely

Special thanks

- Alexey Kukanov
- Jonathan Mueller
- Inbal Levi
- Jeff Garland and Jonathan Wakely
- Zach Laine and Jonathan Mueller for reviewing the abstract for C++Now

Special thanks

- Alexey Kukanov
- Jonathan Mueller
- Inbal Levi
- Jeff Garland and Jonathan Wakely
- Zach Laine and Jonathan Mueller for reviewing the abstract for C++Now
    - Feedback: “The abstract looks good as is” ☺

Useful links

- Parallel Range Algorithms proposal: https://wg21.link/P3179
- Range-as-the-output paper: https://wg21.link/P3490
- Reconsider `rotate_copy` and `reverse_copy`: https://wg21.link/P3709
- oneDPL source code:
  https://github.com/uxlfoundation/oneDPL
- oneDPL specification:
  https://oneapi-spec.uxlfoundation.org/specifications/oneapi/latest/elements/onedpl/source/
- SYCL specification:
  https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

Bonus

Count words example with C++ 17 parallel algorithms

```
int word_count(const std::string& text) {

    // check for empty string
    if (text.empty()) return 0;

    // compute the number characters that start a new word
    int result = std::transform_reduce(std::execution::par,
        text.begin(), text.end() - 1,            // sequence of left characters
        text.begin() + 1,                        // sequence of right characters
        0,                                       // initial value
        [](int s1, int s2) { return s1 + s2; },  // sum values together
        [](char s1, char s2) {
            return int(!std::isalpha(s1)
                       && std::isalpha(s2));     // check if the right character starts the word
    });

    // if the first character is alphabetical, then it also begins a word
    if (std::isalpha(*text.begin())) ++result;

    return result;
}
```

Count words example with parallel range algorithms

```
int word_count(const std::string& text) {
    using namespace std;
    // check for empty string
    if (text.empty()) return 0;

    // compute the number characters that start a new word
    int result = ranges::count_if(std::execution::par,
        views::zip(text | views::take(text.size() - 1),
                   text | views::drop(1)),
        [](auto v) {
            auto [s1, s2] = v;
            return !std::isalpha(s1)
                   && std::isalpha(s2);
        }
    );

    // if the first character is alphabetical, then it also begins a word
    if (std::isalpha(*text.begin())) ++result;

    return result;
}
```

intel®