Parallel Algorithms, Ranges and oneDPL

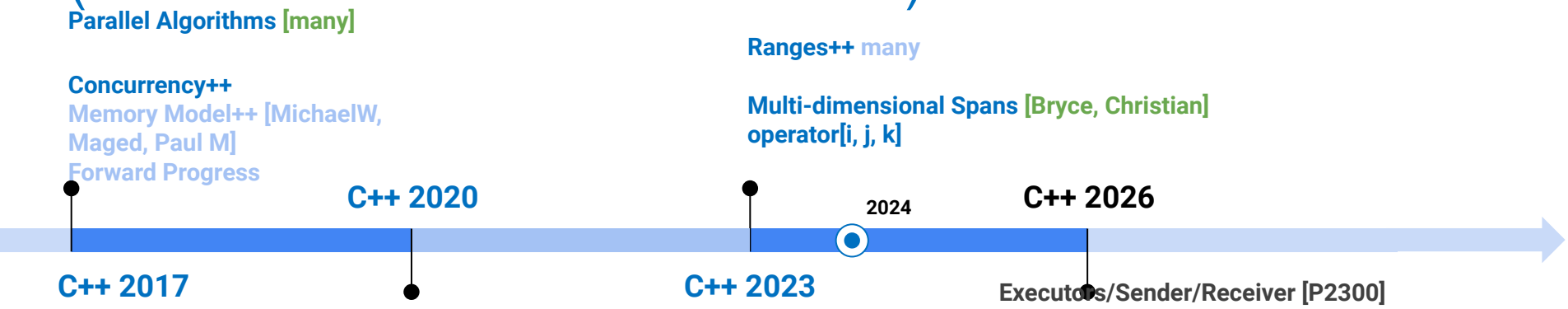
Abhilash Majumder

Agenda

1. C++ Parallel Algorithms(P2300, P2500) , Senders/Receivers concept
2. Parallel Range algorithms (P3179)
3. oneDPL , Thrust use cases
4. QnA

C++ Parallel Algorithms (P2300)

ISO C++ Parallelism/Concurrency Programming Language (based on Gonzalo's ISC C++ BoF)



Motivation

Why P2300?

- The need for a flexible abstraction that answers "where" the code should be executed.
- Limitations of current execution policies in specifying hardware execution contexts. (std::future/std::promise)

C++17 parallel algorithms: A good start

Current limitation: No control over execution hardware

P2300 introduces flexible schedulers

Need: Integrate schedulers with parallel algorithms

The Need for Integration

- C++ parallel algorithms offer parallelism, but lack control over execution hardware ("where").
- P2300 introduces the "scheduler" concept, representing execution contexts, addressing the "where."
- The integration of these two is crucial for leveraging hardware capabilities effectively.

Sender Receiver Example

Example

```
using namespace std::execution;

scheduler auto sch = thread_pool.scheduler();

sender auto begin = schedule(sch);
sender auto hi = then(begin, []{
    std::cout << "Hello world! Have an int.";
    return 13;
});
sender auto add_42 = then(hi, [](int arg) { return arg + 42; });

auto [i] = this_thread::sync_wait(add_42).value();
```

Usage Example

- Initiate a Scheduler
- Call execution::schedule returning a sender which completes the scheduler.
- Use sender algorithms for async work, execution::then is a sender adapter which sends the results of the sender invocation
- Submit the async pipeline (thread::sync_wait) for completion

Sender Factory APIs

Schedule & Just API

```
Execution::sender auto
schedule(
  execution::scheduler
auto scheduler);
```

```
execution::sender auto
just(
  auto ...&& values);
```

Usage Example

```
execution::scheduler auto sch1 =
get_system_thread_pool().scheduler();

/* sender describing start of task graph */;

execution::sender auto snd1 =
execution::schedule(sch1);

/* either moved or decay copied to receiver
*/;

execution::sender auto snd1 =
execution::just(1.0f)
```

Sender Factory APIs

transfer_just, just_error and just_stopped API

```
execution::sender auto
transfer_just(
  execution::scheduler auto
scheduler,

auto …&values);

execution::sender auto just_error(
  auto && error);

Execution::sender auto
just_stopped();
```

Usage Example

```
execution::sender auto vals =
execution::transfer_just(get_system_threadpoo
l.scheduler(), 1,2,3);

/* similar to just*/;

execution::sender auto send =
execution::then(vals, [](auto …args){

Std::print(args…)});
```

Sender Adapter APIs

Then, upon_*, let_* API

```
Execution::sender auto then(
  execution::sender auto input,
std::invocable<…> function);

execution::sender upon_error(
  execution::sender auto
input,std::invocable<…>
function);

Execution::sender
let_value(execution::sender auto
input,std::invocable<…>
function);
```

Usage Example

```
execution::scheduler auto sch1 =
execution::then(input, [](auto…
args){std::print(args);});

/* then returns the sender describing the
task graph described by the input sender with
added invocable.  */

/*upon_error & upon_stopped are similar to
then but work with errors and stop signals*/

/*let_value/error/stopped are similar to then
but return the sender and performs callback –
similar to “future_unwrapping” in
std::future*/;
```

Sender Adapter APIs

Then, upon_*, let_* API

```
Execution::sender auto then(
  execution::sender auto input,
std::invocable<…> function);

execution::sender upon_error(
  execution::sender auto
input,std::invocable<…>
function);

Execution::sender
let_value(execution::sender auto
input,std::invocable<…>
function);
```

Usage Example

```
execution::scheduler auto sch1 =
execution::then(input, [](auto…
args){std::print(args);});

/* then returns the sender describing the
task graph described by the input sender with
added invocable.  */

/*upon_error & upon_stopped are similar to
then but work with errors and stop signals*/

/*let_value/error/stopped are similar to then
but return the sender and performs callback –
similar to “future_unwrapping” in
std::future*/;
```

Sender Adapter APIs

on, into_variant, bulk and split API

```
execution::sender auto on(
  execution::scheduler auto sched,
execution::sender auto snd);

execution::sender auto into_variant(
  execution::sender auto snd);

Execution::sender auto bulk
(execution::sender auto
input,std::integral auto size,
std::invocable<decltype(size),...>
function);

Execution::sender auto
split(execution::sender auto sender);
```

Usage Example

```
/* on will start the provided sender on an
execution agent belonging to a context by
scheduler.  */

/*into_variant sends a variant of tuples of
all the possible sets of types sent by the
input sender*/

/*bulk returns a sender describing the task
of invoking the provided function with every
index in the provided shape along with the
values sent by the input sender.*/;

/*split returns a sender if the provided
sender is a multishot sender*/
```

Sender Adapter APIs

other API (transfer_when_all, when_all, ensure_started, stopped_as_*)

```
execution::sender auto
transfer_when_all(
  execution::scheduler auto sched,
execution::sender auto …inputs);

Execution::sender auto
when_all(execution::sender auto
…inputs);

Execution::sender auto
ensure_started(execution::sender
auto sender);
```

Usage Example

```
/* similar to when_all but returns a sender
whose completed value is provided scheduleron
will start the provided sender on an
execution agent belonging to a context by
scheduler.  */

/*when_all returns a sender that completes
once all of the input senders have completed.
It is constrained to only accept senders that
can complete with a single set of values
(_i.e._, it only calls one overload of
set_value on its receiver)..*/;

/*ensure_started implies that  provided
sender has connected through
execution::connect and start is called on Op
state*/
```

Sender Consumers APIs

start_detached,this_thread::sync_wait

```
void
start_detached(){execution::
sender auto sender;}
```

```
auto
sync_wait(execution::sender
auto sender) ->
std::optional<std::typle<val
ues-sent-by(sender)>>
```

Usage Example

```
/* Like ensure_started, but does not return a
value; if the provided sender sends an error
instead of a value, std::terminate is
called.*/
```

```
/*this_thread::sync_wait is a sender consumer
that submits the work described by the
provided sender for execution, similarly to
ensure_started, except that it blocks the
current std::thread or thread of main until
the work is completed, and returns an
optional tuple of values that were sent by
the provided sender on its completion of
work.*/
```

Main features of Senders Receivers P2300

Senders & Receivers

- Senders represent work composable through sender algorithms, can be forked or joined.
- Senders can be multishot or single shot
- Sender factory and adapter models are lazy and they support cancellation of scheduled tasks (stopped)
- Many senders can be trivially made awaitable and all awaitables are senders.
- Receivers are glue between senders
- Execution::connect is customization point which serves as connection between senders and receivers.
- Lazy senders provide optimization by fusing multiple ops /functions that can be submitted via an execution context.
- Customization of transfer with Schedulers: The transfer function in C++ does not allow direct customization by the target scheduler because specialized schedulers (like CUDA or remote node schedulers) may require specific runtime calls for transitioning between execution contexts, especially for accelerators. To address this, these specialized schedulers can inject a sender to handle transitions to a regular CPU execution context.
- Introducing schedule_from Adaptor: To enable full customization of transitions by both the source and target schedulers, the proposal introduces a schedule_from adaptor, which takes a scheduler and a sender as arguments. This adaptor allows both schedulers to customize the transition, ensuring proper execution in specialized contexts like GPUs. The default implementation of transfer(snd, sched) is equivalent to schedule_from(sched, snd).

Sender Consumers Example

Pipe in a CUDA execution context (except: when_all*, on)

```
auto snd =
execution::schedule(thread_pool.schedu
ler())          | execution::then([]{
return 123; }) |
execution::transfer(cuda::new_stream_s
cheduler()   | execution::then([](int
i){ return 123 * 5; })   |
execution::transfer(thread_pool.schedu
ler())          |
execution::then([](int i){ return i -
5; });

auto [result] =
this_thread::sync_wait(snd).value();
```

Usage Example

Piping enables us to compose together senders with a linear syntax. Without it, we would have to use either nested function call syntax, which would cause a syntactic inversion of the direction of control flow, or have to introduce a temporary variable for each stage of the pipeline. Consider the following example where we want to execute first on a CPU thread pool, then on a CUDA GPU, then back on the CPU thread pool

C++ Parallel Algorithms (P2500)

P2500R2 C++ parallel algorithms and P2300

Overview:

- The evolution of parallelism in C++.
- Why P2300 is important for C++26.

Goal of the Talk:

- Discuss the integration of C++ parallel algorithms with the facilities introduced in P2300.
- https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2500r2.htm

Further evolution of parallel algorithms (P2500)

```
//C++ 11/14
namespace stde = std::execution;
std::vector<int> vec{N};

// https://wg21.link/P3179: parallel range algorithms
std::ranges::generate( vec, std::minstd_rand{} );
std::ranges::sort( stde::par, vec );

// https://wg21.link/P2500: support schedulers to specify where to execute,  targeted to cover synchronous parallel algorithms integration with P2300.
std::ranges::generate(stde::execute_on(some_sched, stde::seq), vec, std::minstd_rand{} );
std::ranges::sort( stde::execute_on(some_sched, stde::par), vec );
```

Design Overview

The Proposed Solution in P2500

- The paper proposes extending C++ algorithms to accept a "policy-aware scheduler."
- This scheduler combines an execution policy ("how") and a scheduler ("where").
- The `execute_on` function facilitates the creation of such policy-aware schedulers.

Design Goals:

- Extending C++ parallel algorithms with policy-aware schedulers.
- Allowing customization for different execution contexts.

- Preserve core semantics of algorithms and policies
- Cover both "classic" and range-based algorithms

- Minimal API changes: The design aims to preserve the existing usage patterns of C++ algorithms.
- Flexibility: It allows execution semantics to be adjusted based on the capabilities of the execution context.
- Customization: Implementers of execution contexts can customize the implementation of standard algorithms for optimal performance.

Key Features:

- Combining scheduler and policy.
- Minimal, incremental API changes.

Combining Scheduler with Policy

Why Use Schedulers?

- Schedulers represent execution contexts and provide flexibility.
- API overview

Key Concepts

1. policy_aware_scheduler
2. execute_on
3. Customizable functions

    - The `execution_policy` concept defines the requirements for execution policies.
    - The `policy_aware_scheduler` concept represents an entity combining a scheduler and an execution policy.
    - The `execute_on` customization point binds a scheduler and an execution policy.
    - Parallel algorithms are defined as customizable functions, allowing customization for specific policy-aware schedulers.

policy_aware_scheduler Concept

API

```
template <typename S>
concept policy_aware_scheduler =
scheduler<S> && requires (S s) {
  typename
S::base_scheduler_type;
  typename S::policy_type;
  { s.get_policy() } ->
execution_policy;
};
```

Usage Example

```
struct MyScheduler {    using
base_scheduler_type = /* some scheduler type
*/;

    using policy_type = /* some execution
policy type */;

    policy_type get_policy() const {

        return /* return the associated
policy */;

    }};

  static_assert(policy_aware_scheduler<MySch
  eduler>);
```

execute_on Function API and Usage example

```
inline namespace
__execute_on_fn_namespace
{

  inline constexpr
__detail::__execute_on_fn
execute_on;

 }
```

```
auto
policy_aware_sched =
std::execute_on(my_sc
heduler,
std::execution::par);
```

Proposed API (Example with for_each)

```
// Existing API
template<class ExecutionPolicy, class It, class Fun>
      constexpr void for_each(ExecutionPolicy&& policy, It first, It last, Fun f);
// New Policy-based API
template<execution_policy Policy, input_iterator I, sentinel_for<I> S, class Proj = identity,
         indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr ranges::for_each_result<I, Fun>
  ranges::for_each(Policy&& policy, I first, S last, Fun f, Proj proj = {});
// New Scheduler-based API
template<policy_aware_scheduler Scheduler, input_iterator I, sentinel_for<I> S,
         class Proj = identity, indirectly_unary_invocable<projected<I, Proj>> Fun>
constexpr ranges::for_each_result<I, Fun>
  ranges::for_each(Scheduler sched, I first, S last, Fun f, Proj proj = {}) /*customizable*/;
```

Allowing schedulers with C++ algorithms.
Blocking behavior similar to C++17 parallel algorithms.

```
template<policy_aware_scheduler Scheduler,
typename ForwardIterator, typename Function>
void for_each(Scheduler&& sched, ForwardIterator
first, ForwardIterator last, Function f) {
    // Implementation using scheduler and policy
    sched.execute([&]() {
        for (; first != last; ++first) {
            f(*first);
        }
    });
}
```

Using the API

```
std::for_each(

std::execute_on(my_gpu_scheduler,
std::execution::par),
    begin(data),
    end(data),
    [](auto& item) {
item.process(); }
 );
```

Customization and Extensibility

Parallel Algorithms as Customizable Functions:

- Customization through policy-aware schedulers.
- Flexibility to support platform-specific optimizations.

Example Customization:

- Using CUDA-specific scheduler for `std::for_each`.

```
namespace cuda {
  struct scheduler {
    friend constexpr auto
    tag_invoke(std::tag_t<ranges::for_each>, scheduler,
/*...*/) {
      // CUDA-optimized implementation
      cuda_kernel<<<blocks, threads>>>(/*...*/);
      return std::ranges::for_each_result{/*...*/};
    }
  };
}
```

Covering Classic and Range Algorithms

Support for Both Types:

- Customizable algorithms defined in `std::ranges`.
- Parallel and range-based algorithms integration with schedulers

Possible Implementations:

- Default and customized implementations using `tag_invoke`.
- Extending existing function objects with new constrained overloads.

Example Implementation:

- Code snippets demonstrating the implementation for `std::ranges::for_each`.

.

proposed API can be implemented with customization points

```
namespace ranges {
    struct __for_each_fn {
template<policy_aware_scheduler Scheduler, input_iterator I, sentinel_for<I> S,
         class Proj = identity, indirectly_unary_invocable<projected<I, Proj>>
Fun>
constexpr for_each_result<I, Fun>
operator()(Scheduler sched, I first, S last, Fun f, Proj proj = {}) const {

if constexpr (std::tag_invocable<__for_each_fn, Scheduler, I, S, Fun, Proj>) {

    std::tag_invoke(*this, sched, first, last, f, proj);

} else {

    // default implementation

  }

    }

  };
```

Benefits

Unified API for different execution contexts

Optimized implementations per platform

Extensible to future scheduling needs

Covers both iterator and range-based algorithms

Open Questions

Exact customization mechanism

- tag_invoke vs language support

Impact on execution rules
Set of basic parallel functions

Next Steps

- Finalize customization approach
- Specify behavior for all parallel algorithms
- Explore "parallel backend" functions
- Gather community feedback

- Explore specifying a set of basic functions as a "parallel backend."
- Further customization for different execution policies and schedulers.

Summary

P2300 offers significant flexibility and control over execution contexts.

P2500 Integration with parallel algorithms is crucial for modern C++ development.

The future of C++ parallelism lies in customizable and extensible algorithms.

Parallel Range Algorithms (P3179)

Design Overview

Key Modifications:

- The parallel range algorithms should be close to C++17 classic ones to use the code with minimal required changes.
- The parallel range algorithms should return the same type as the corresponding serial range algorithms.
- The proposed algorithms should be special non-ADL discoverable functions, same as serial range algorithms.
- The required range and iterator categories should at least be random access for all but std::execution::seq execution policies.
- The proposed API should require any callable object passed to an algorithm to have const-qualified operator().
- The proposed API is not a customization point.
- The proposed API is not constexpr.

Switch from Parallel Range Algorithms Example

```
// Before
std::for_each(std::execut
ion::par, v.begin(),
v.end(), [](auto& x) {
++x; });
```

```
// After

//Using Iterator and Sentinel

std::ranges::for_each(std::execution
::par, v.begin(), v.end(), [](auto&
x) { ++x; });

//Or use a Range directly

std::ranges::for_each(std::execution
::par, v, [](auto& x) { ++x; });
```

More expressive & unified calls

```
// Before
reverse(policy, begin(data),
end(data));
transform(policy, begin(data),
end(data), begin(result), [](auto i){
return i * i; });
auto res = any_of(policy,
begin(result), end(result), pred);
```

```
// After

//With Ranges API

auto res = any_of(policy, data |
views::reverse |
views::transform([](auto i){ return
i * i; }), pred);
```

Bringing parallelism to std::ranges algorithms (P3179)

```
// C++03
template<class RandomAccessIterator, class Compare>
 void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// C++17
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
 void sort(ExecutionPolicy&& exec, RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// C++20
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
 constexpr borrowed_iterator_t<R> ranges::sort(R&& r, Comp comp = {}, Proj proj = {});

// C++26? (https://wg21.link/P3179)
template<class ExecutionPolicy, random_access_range R, class Comp = ranges::less, class Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
 constexpr borrowed_iterator_t<R> ranges::sort(ExecutionPolicy&& exec, R&& r, Comp comp = {}, Proj proj = {});
```

C++ Parallel Range Algorithms: Unifying Parallelism and Ranges

Why Combine Parallelism with Ranges?

- Ranges offer a productive API with opportunities for optimization.
- Users are already using ranges with non-range parallel algorithms; integrating execution policies simplifies and streamlines code.

The Power of Ranges and Parallelism

- The C++ Ranges library provides a powerful way to express and compose computations lazily.
- C++17 introduced parallel algorithms, but they don't integrate seamlessly with ranges.
- This paper proposes adding parallel algorithms that work directly with ranges, combining the benefits of both worlds.

The Need for Parallel Range Algorithms

- Users often combine ranges and parallel algorithms, but the current approach is verbose and error-prone.
- The proposed parallel range algorithms offer a more natural and expressive way to parallelize range-based computations

.

Design Overview

Key Modifications:

- Execution policy parameter added to range algorithms.
- Introduction of bounded ranges for better parallel performance.

- **Execution policies:** Parallel range algorithms accept execution policies to control parallelism.
- **Random access ranges:** Algorithms require random-access ranges for efficient parallelization.
- **Bounded ranges:** At least one input and the output range must be bounded for safety and performance.
- **Algorithm return types:** Consistent with serial range algorithms for easy migration.

- Enable single-call fusion of multiple operations
- Preserve the expressiveness of ranges

Key Design Decisions

1. Return types match serial range algorithms
2. Require random_access_range (for now)
3. Take range as output
4. Require bounded ranges
5. Preserve callable requirements from C++17 parallel algorithms

```
template <class ExecutionPolicy,
random_access_range R,
          class Proj = identity,

indirectly_unary_invocable<projected<i
terator_t<R>, Proj>> Fun>

requires
sized_sentinel_for<ranges::sentinel_t<
R>, ranges::iterator_t<R>>

ranges::borrowed_iterator_t<R>

ranges::for_each(ExecutionPolicy&&
policy, R&& r, Fun f, Proj proj = {});
```

Differences to C++17 Parallel Algorithms

Key Differences:

- Parallel range algorithms require random access ranges.
- Output can now be a range instead of just an iterator.

Fusion Example

```
// Before (multiple parallel algorithm calls)
reverse(policy, begin(data), end(data));
transform(policy, begin(data), end(data), begin(result),
          [](auto i){ return i * i; });
auto res = any_of(policy, begin(result), end(result), pred);
```

```
// After (fusing operations with parallel range algorithms)
auto res = ranges::any_of(policy,
    data | views::reverse
         | views::transform([](auto i){ return i * i; }),
    pred);
```

Benefits

Parallel range algorithms offer a natural and efficient way to parallelize range-based computations.

The proposed design integrates seamlessly with the Ranges library and existing parallel algorithms.

This feature will enhance the expressiveness and performance of parallel code in C++.

More expressive code

Potential for better performance

Safer APIs (bounded ranges, range outputs)

Simplified migration from serial to parallel code

oneDPL, Thrust use cases

Thrust

Thrust is the C++ parallel algorithms library which inspired the introduction of parallel algorithms to the C++ Standard Library.

The Thrust library is built on top of CUDA, OpenMP, TBB and provides many general purpose parallel algorithms for standard C++ library.

Thrust can be used to perform parallel operations such as reduce, excusive_scan, for_each, gather etc. Has both host and device execution policies.

Thrust can choose different backends for GPUs and CPUs (through TBB) to perform parallelization.

Host or Device Execution Policy Example

```
// Template impl
Template<typename
DerivedPolicy>
struct host_execution_policy :
public
thrust::system::_THRUST_HOST_S
YSTEM_NAMESPACE::execution_pol
icy<DerivedPolicy>

Template<typename
DerivedPolicy>
struct device_execution_policy
: public
thrust::system::_THRUST_HOST_S
YSTEM_NAMESPACE::execution_pol
icy<DerivedPolicy>
```

```
// Calling host_policy for parallel
range algorithms)

Struct mypolicy :
thrust::host_execution_policy<mypoli
cy>();

// Calling host_policy  for
transform

Thrust::transform(mypolicy, data,
data+1, data,
thrust::identity<int>());
```

oneDPL (ParallelSTL)

oneAPI DPC++ Library (oneDPL) works with the Intel oneAPI DPC++/C++ Compiler to provide high-productivity APIs to developers, which can minimize SYCL* programming efforts across devices for high performance parallel applications.

oneDPL provides parallel api, api mapping for SYCL kernels, and macros; provides different device specific implementation for parallel algorithms on SYCL runtime. (vectorized algorithms, segmented scan etc)

oneDPL is involved in device execution policies and follows std::execution (seq, unseq, par, par_unseq, dpcpp_default, dpcpp_fpga)

oneDPL Execution Policy Example

```
// C++ std exec policies
std::fill(oneapi::dpl::execution::
par_unseq, data.begin(),
data.end(), 42);
```

```
//SYCL  device policies

//Calling device_policy for  SYCL
gpus (Intel)

auto policy_b = device_policy<class
PolicyB> {device{gpu_selector_v}};

std::for_each(policy_b, ...);

// Calling dpcpp_default

auto policy_b = device_policy<class
PolicyB> {dpcpp_default};

std::for_each(policy_b, ...);
```

oneDPL Ranges Example

```
using namespace oneapi::dpl::experimental::ranges;
{
    sycl::buffer<int> A(data, sycl::range<1>(max_n));
    sycl::buffer<int> B(data2, sycl::range<1>(max_n));
    auto view = all_view(A) | views::reverse();
    auto range_res = all_view<int, sycl::access::mode::write>(B);
    copy(oneapi::dpl::execution::dpcpp_default, view, range_res);
}
```

Challenges and Questions

Challenges and Open Questions

Thread safety of views

Support for forward ranges

Customization points

Interaction with schedulers (P2300)

Next Steps

Finalize API design

Address thread safety concerns

Explore integration with P2300 (schedulers)

Implement and benchmark

Parallel Algorithms, Ranges and oneDPL

Abhilash Majumder

Resources

1. P2300. P2500
2. Parallel Range algorithms (P3179)
3. oneDPL , Thrust use cases