SG1

~~P2690~~
P2500 Parallel Algorithms and P2300

Ruslan Arutyunyan

Motivation

- Parallel algorithms with execution policies (C++17) were a good start to express parallelism in C++ standard
- Schedulers/senders/receivers is more flexible abstraction for answering “where” the code is executed.
- P2300 is targeted to C++26 so we need the answer how the rest of C++ standard library interoperates with schedulers/senders/receivers

P2500 is intended to answer how C++17 parallel algorithms work together with P2300 std::execution

API on the user side

The API call should look like (based on std::for_each):

```
for_each(execute_on(scheduler, std::execution::par), begin, end,
         callable);
```

Design to achieve:

- Implementation should call customization of for_each if exists
- Otherwise, the default implementation is called.
- Customization of every particular algorithm should be allowed

for_each CPO underneath

```
struct __for_each
{
    template <std::policy_aware_scheduler Scheduler, typename It, typename Callable>
    void operator()(Scheduler s, It b, It e, Callable c) const {
        if constexpr (std::tag_invocable<__for_each, Scheduler, It, It, Callable>) {
            std::tag_invoke(*this, s, b, e, c);
        }
        else {
            // default implementation
        }
    }
};
inline constexpr __for_each for_each;
```

Why scheduler?

- Allows getting as many senders as algorithm wants to be able to build whatever dependency graph.

Alternatives:

- Could be “combinated tag” of scheduler and execution policy in that case generic implementation does not have much to do with that
- Could be execution policy but makes it harder to get necessary objects from (e.g., scheduler for default algorithm implementation or allocator, if available)

execute_on

```
struct __execute_on {
    policy_aware_scheduler auto operator()(scheduler auto sched, execution_policy auto policy) const {
        return std::tag_invoke(*this, sched, policy);
    }
};
inline constexpr __execute_on execute_on;
```

- Serves the purpose to tie scheduler and execution policy. It's up to scheduler customization to check if it can work with the passed execution policy.
- Might have the default implementation but it's an open question what the behavior it should implement.

Policy aware scheduler

```
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;
};
```

- Allows to get both execution policy type and execution policy object
- Allows to get scheduler type it was constructed over.
  - Necessary for parallel algorithm to be able to reuse existing implementation for “known” base_scheduler_type

Execution policy concept

```
template <typename ExecutionPolicy>
concept execution_policy = std::is_execution_policy_v<std::remove_cvref_t<ExecutionPolicy>>;
```

- Necessary if we want to constraint the return type of (some kind of) s.get_policy() method for policy_aware_scheduler
- Has a potential problem with user-defined policies support. We might need to allow is_execution_policy trait specialization.

Open questions

- Should execute_on have default implementation?
  - If yes, should it advice sequential execution using passed scheduler execution resources or the calling thread?
  - If no, what a default behavior should it advice for scheduler to implement?
- What if the scheduler is used in entry point to the binary as a polymorphic (or type-erased) scheduler? How would it know that customization appears?
- If execution_policy concept is necessary should specialization of is_execution_policy be allowed?

Further exploration

Explore the feasibility of the set of basic functions the rest of algorithm can be expressed with (code name: “parallel backend”

- Allows to customize just parallel backend instead of customizing every single algorithm.
- Might result in separate paper based on the analysis.

Namespace for new algorithms

Main topics during Kona (2022) SG1 review

- What should be the execution context argument(s)?
  - Is that a policy wrapping scheduler?
  - Is that a scheduler wrapping policy?
  - Is that a something else, like execution environment?
- Should execution context include knobs for algorithm tuning?
- Does it depend on async algorithm?

Backup

Alternative considered API

Alternative API might have both scheduler and execution_policy as operator() parameters.

```
struct __for_each {

    template <std::policy_aware_scheduler Scheduler, std::execution_policy ExecutionPolicy,
              typename It, typename Callable>
    void operator()(Scheduler s, ExecutionPolicy policy, It b, It e, Callable c) const;

};
```

- Complicates the API for the user (IMHO)
- Still requires scheduler to check if it can work with passed execution policy but on the later stage, after the algorithm call is resolved


intel