Remove `std::execution::split`

Document Number: P3682R0
Date: 2025-05-05
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: LEWG

Abstract

This paper proposes removing `std::execution::split`.

Background

P2300 added `std::execution::split` to the C++26 working draft ([1] at §34.9.11.10). As described by Eric Niebler the purpose of this algorithm is to "represent[] a fork in the execution graph." This extrinsic description of the purpose of `std::execution::split` is necessary because P2300's high level description thereof (id. at §4.20.10) does not accurately describe the algorithm:

"If the provided sender is a multi-shot sender, returns that sender. Otherwise, returns a multi-shot sender which sends values equivalent to the values sent by the provided sender."

The erroneous description gives the false impression that the purpose of `std::execution::split` is to transform single shot senders into multishot senders. While it does accomplish that goal there is an important subtlety.

Particularly the above-quoted description of `std::execution::split` errs in that `std::execution::split` is not the identity transformation when applied to multi-shot senders. Instead `std::execution::split` behaves identically when passed single shot and multi-shot senders.

In actual fact `std::execution::split` yields a multi-shot sender associated with a shared state. When that sender is connected and started the resulting asynchronous operation:

1. Checks to see if the shared state encapsulates a completion, if so completes therewith, otherwise
2. Checks to see if the shared state encapsulates an ongoing operation, if so waits for the completion thereof and then completes with the results sent thereby, otherwise
3. Starts the asynchronous operation obtained by connecting and starting the sender provided to `std::execution::split`
4. On completion of the above stores those values in the shared state and awakens all operations waiting at step 2

Discussion

Deficiencies of `std::execution::split`

Dynamic Allocation

When a `std::execution::split` sender is created a state is created therefor. This state must be dynamically allocated due to the fact it is shared by:

- The original sender,
- All senders derived by copying the above, and
- All operation states derived by connecting either of the above

Not only is this an issue due to the performance overhead of dynamic allocation, but the dynamic allocation occurs so early in the `std::execution` workflow (i.e. at sender creation time) that the allocation cannot be parameterized by the custom allocator provided by the receiver's environment (i.e. at connect time).

Shared Ownership

As detailed in the preceding section the state associated with a `std::execution::split` sender is shared. Not only does this involve reference counting which can usually be avoided through structured concurrency ([1] at §1.4.1.1) but it also causes issues when reasoning about lifetimes. Particularly the lifetime of the resulting operation state [2] and result therefrom are stored in the shared state and therefore persist until the end of the lifetime of the last sender or operation state referring thereto.

Contrast the above with other asynchronous operations wherein the sender's lifetime has nothing to do with the lifetime of the operation state and/or completions transmitted thereby.

Eagerness

P2300R10 has the following to say about eagerness ([1] at §4.10):

"In an earlier revision of this paper, some of the proposed algorithms supported executing their logic eagerly; i.e., before the returned sender has been connected to a receiver and started. These algorithms were removed because eager execution has a number of negative semantic and performance implications."

Despite this `std::execution::split` represents execution which is, conditionally and from certain points of view, eager.

It is true that the first time a `std::execution::split` sender is connected and started it is lazy, as is expected from the `std::execution` model. From the point of view of subsequent consumers, however, the operation is eager since it was already started by the first consumer thereof. This means that `std::execution::split` suffers from all the concerns long understood regarding attaching continuations to eager execution [3].

Naming

"Split" is a very short (i.e. good) name. Reserving it for an operation which is so esoteric in the face of `std::execution`'s norms (see above) seems ill-advised.

Alternatives to `std::execution::split`

Rather than using reference counting and shared ownership to share the value(s) computed by a shared predecessor, users can instead use operations which adhere to structured concurrency principles. Consider the following example:

```
std::execution::sender auto shared = /* ... */;
std::execution::sender auto split = std::execution::split(std::move(shared));
std::execution::sender auto a = split | /* ... */;
std::execution::sender auto b = split | /* ... */;
std::execution::sender auto c = std::move(split) | /* ... */;
(void)std::this_thread::sync_wait(
  std::execution::when_all(
    std::move(a),
    std::move(b),
    std::move(c)));
```

This can be written without the overhead of dynamic allocation, shared ownership, and eager continuation attachment as:

```
std::execution::sender auto shared = /* ... */;
(void)std::this_thread::sync_wait(
  std::move(shared) | std::execution::let_value([](auto&&... values) {
    std::execution::sender auto a = std::execution::just(std::ref(values)...)
      | /* ... */;
    std::execution::sender auto b = std::execution::just(std::ref(values)...)
      | /* ... */;
    std::execution::sender auto c = std::execution::just(std::ref(values)...)
      | /* ... */;
    return std::execution::when_all(
      std::move(a),
      std::move(b),
      std::move(c));
  }));
```

For more complex patterns async scopes [4][5] can be used.

Proposal

Remove `std::execution::split`. Replace it with nothing.

Wording

[execution.syn]

```
struct starts_on_t { unspecified };
struct continues_on_t { unspecified };
struct on_t { unspecified };
struct schedule_from_t { unspecified };
struct then_t { unspecified };
struct upon_error_t { unspecified };
struct upon_stopped_t { unspecified };
struct let_value_t { unspecified };
struct let_error_t { unspecified };
struct let_stopped_t { unspecified };
struct bulk_t { unspecified };
~~struct split_t { unspecified };~~
struct when_all_t { unspecified };
struct when_all_with_variant_t { unspecified };
struct into_variant_t { unspecified };
struct stopped_as_optional_t { unspecified };
struct stopped_as_error_t { unspecified };

inline constexpr starts_on_t starts_on{};
inline constexpr continues_on_t continues_on{};
inline constexpr on_t on{};
inline constexpr schedule_from_t schedule_from{};
inline constexpr then_t then{};
inline constexpr upon_error_t upon_error{};
inline constexpr upon_stopped_t upon_stopped{};
inline constexpr let_value_t let_value{};
inline constexpr let_error_t let_error{};
inline constexpr let_stopped_t let_stopped{};
inline constexpr bulk_t bulk{};
~~inline constexpr split_t split{};~~
inline constexpr when_all_t when_all{};
inline constexpr when_all_with_variant_t when_all_with_variant{};
inline constexpr into_variant_t into_variant{};
inline constexpr stopped_as_optional_t stopped_as_optional{};
inline constexpr stopped_as_error_t stopped_as_error{};
```

[exec.split]

Remove entire section.

References

[1] M. Dominiak et al. std::execution P2300R10
[2] R. Leahy. Of Operation States and Their Lifetimes P3373R1
[3] C. Kohlhoff. A Universal Model for Asynchronous Operations N3747
[4] I. Petersen et al. `async_scope` – Creating scopes for non-sequential concurrency P3149R9
[5] A. Williams. `let_async_scope` P3296R4