Abstract

We propose to modify the unified executors programming model described by P0443R10 to increase consensus and ensure that executors merge with C++23 as planned.

Proposal Summary

We propose a compromise between competing requirements on executors by eliminating interface-changing properties, retaining behavioral properties, establishing a single Executor concept, and incorporating Senders and Receivers.

Proposed changes to P0443R10:

- Eliminate OneWayExecutor, BulkOneWayExecutor, and interface-changing properties oneway and bulk_oneway.
- Introduce an Executor concept based on a function named execute which eagerly creates a single execution agent.
- Introduce a customization point object named bulk_execute which eagerly creates multiple execution agents in bulk.
- Introduce a Scheduler concept based on a function named schedule which lazily creates a Sender of a subexecutor.
- Introduce Senders, Receivers, and related operations with additions described below.

Context of Proposed Compromise

P0443 depends on a separate proposal for properties allowing executor authors to impose requirements and preferences on execution via a novel API. LEWG voted with a small majority to forward this proposal to LWG for C++23. Meanwhile, discussion within the sg1-exec mailing list revealed that properties, especially “interface-changing” properties exposed via require_concept, remain controversial. Detractors argue that interface-changing properties and their implied disjoint concept hierarchy make code maintenance challenging. As an additional constraint, Executors must somehow integrate cleanly with Senders and Receivers. In order to address controversial parts of P0443 and build greater consensus, we suggest a compromise.
Suggestions for Properties

The controversy surrounding the require_concept operation and related interface-changing properties is great enough that we are not confident it will survive WG21 scrutiny. In order to increase consensus around P0443, we suggest removing dependence on require_concept and eliminating P0443’s two proposed interface-changing properties, execution::oneway and execution::bulk_oneway.

This retains P0443’s dependence on cross-cutting behavioral properties. We perceive these as less controversial than interface-changing properties because behavioral properties do not alter the APIs of objects. Yet, disagreement exists concerning behavioral properties’ ergonomics. However, after failing to build a larger consensus around a modification of behavioral properties, we do not propose to modify them at this time. What Nvidia requires of an executors programming model is a way to express execution agent behaviors and guarantees while avoiding a combinatorial explosion of executor basis operations. P0443’s behavioral properties, or some tasteful equivalent, are currently the only proposal for representing cross-cutting behavioral requirements and preferences. Ultimately, Nvidia will support whatever formulation of behavioral properties yields consensus.

Suggestions for Executors

P0443’s multiple disjoint executor concepts (OneWayExecutor and BulkOneWayExecutor) are a related source of opposition. Skeptics argue that a good executor concept hierarchy should have a single root in order to avoid intractable maintenance problems. Originally, the P0443 authors assumed the necessity of an extensible executor concept hierarchy allowing multiple roots. This is because they observed that low-level work creation platforms provide different modes of creating work. It seemed prudent to define multiple root concepts abstracting those modes. An executor satisfying the requirements of one of these concept roots could be converted into one satisfying another concept via a require_concept operation. For example, earlier revisions of P0443 defined roots such as TwoWayExecutor, BulkTwoWayExecutor, ThenExecutor, and BulkThenExecutor. We at Nvidia now understand that the proposed Sender and Receiver model makes such additional concepts superfluous because any one-way executor can be mechanically adapted to create two-way execution by executing a Sender’s submit operation. Absent extant examples of executor concepts which cannot be rooted at a single Executor concept, we believe there is no compelling reason to introduce controversial extensibility at this time.

To summarize, Nvidia believes a single Executor concept should be CopyConstructible, EqualityComparable, and provide a function named execute which eagerly creates a single execution agent. Executors abstracting bulk execution platforms such as GPUs or SIMD units would need to implement execute to satisfy the Executor requirements and also implement bulk_execute or similar operations to accelerate bulk tasks. We are agnostic as to whether or not execute should be one-way or two-way, but note that some kinds of execution agents are unable to throw exceptions. GPU agents, SIMD agents, and any agent in a program compiled without exception handling all are examples. An eager execute operation with flexible error handling responsibilities would generalize to all of these.

With a single Executor root concept, P0443 no longer needs require_concept nor interface-changing properties. P1393 proposes require_concept, and P0443R10 proposes interface-changing properties to be used with it. Nvidia is agnostic as to whether or not require_concept is necessary in general. However, we do not believe that the P0443 executors programming model requires it. To build consensus, we believe P0443’s interface-changing properties oneway and bulk_oneway should be eliminated in order to decouple executors from interface-changing properties. In this environment, all Executors would be required to provide the execute function. However, in order to retain the functionality offered by the bulk_oneway property, we propose a customization point object named bulk_execute callable with any Executor to create bulk execution.
Suggestions for Senders and Receivers

Distinguish Executors from Schedulers

P1341 suggests a single Executor concept defined by the basis operation schedule. As discussed, we are not opposed to a single Executor concept. Nor are we opposed to schedule in principle, but we do not believe that it is Executor’s defining operation. We believe that a concept’s basis operation should be the locus of customization, and that implementing the requirements for a concept should be fairly straightforward. For example, “Implement this eager execute function.” is the simplest and most useful answer to the question “How do I implement an executor?” This is because operations that eagerly create EA(s) are the kind of native operations typical of lowest-level interfaces for work creation.

On the other hand, P1341’s schedule operation does not directly create execution agents. Rather, it returns a special kind of Sender whose submit operation is analogous to P0443 execute. The resulting Sender is then composable with other Senders via a library of combinators. We agree that this kind of operation is invaluable, but it is not the best target for the lowest-level execution facilities Executors abstract. Moreover, schedule, and the accompanying Sender and Receiver machinery, are an excess of responsibilities for a low-level concept.

Rather than base Executor on schedule, a separate Scheduler concept layered on top of Executor is more composable. Scheduler’s basis operation would be schedule and would return a Sender as P1341 proposes. The implementor of a Scheduler could manage an Executor to create eager execution upon a call to submit. Such a factoring simplifies Executor author responsibilities by requiring only a customization of execute, rather than customizations of both schedule and the resulting Sender type. Such a layering enables flexible implementation strategies. Moreover, types are free to be both Executor and Scheduler simultaneously. Finally, a separate Scheduler concept avoids P1341’s recursive definition of Executor, which is currently inexpressable via C++ concepts.

Factor submit into more basic operations

Based on our experience prototyping, we believe that Senders could be enhanced by factoring the submit operation into two more primitive parts. The current proposal specifies submit as a basic operation which signals that a lazily constructed Sender is 1. ready for execution and 2. may be executed immediately. It would be valuable to separate these two conditions into named operations such that the cost of readying a Sender for execution can be decoupled from its launch. A separate operation for Sender readiness (for the sake of discussion, named finalize) quarantines expensive preparations for execution and prevents their incursion into the launch operation (for the sake of discussion, named start). To be clear, we do not propose replacing submit with finalize and start. All three operations are valuable.

Examples of expensive finalization operations we have encountered during our prototyping include:

- Memory allocation of temporary objects required during execution
- Just-in-time compilation of heterogeneous compute kernels
- Instantiation of task graphs
- Serialization of descriptions of work to be executed remotely

Allowing finalize to be a separate operation is essential because performance-conscious C++ programmers must be able to control where expensive things occur.

Enhance P0443 with Senders and Receivers

Finally, with the above additions, we suggest introducing Senders and Receivers and related operations described by P1341 into P0443.
Appendix: Before & After Proposed Changes

// One-Way execution

// Before: P0443R10
// compiles only if can_require_v<E, oneway_t>
std::require(ex, std::execution::oneway).execute(f);

// After: Proposed P0443R
// always compiles if ex is an Executor
std::tbd::execute(ex, f);

// One-Way Bulk Execution

// Before: P0443R10
// compiles only if can_require_v<E, bulk_oneway_t>
std::require(ex, std::execution::bulk_oneway).bulk_execute(f, shape, sf);

// After: Proposed P0443R11
// always compiles if ex is an Executor
std::tbd::bulk_execute(ex, f, shape, sf);

// Authoring bulk executors

// Before: P0443R10
struct simplest_bulk_executor {
    auto operator<=> const noexcept = default;

    template<class F, class SF>
    void bulk_execute(F f, size_t n, SF shared_factory) const {
        auto shared = shared_factory();
        for(size_t i = 0; i < n; ++i) {
            f(i, shared);
        }
    }
};

// After: Proposed P0443R11
struct simplest_bulk_executor {
    auto operator<=> const noexcept = default;

    template<class F>
    void execute(F f) const noexcept {
        f();
    }

    template<class F, class SF>
    void bulk_execute(F f, size_t n, SF shared_factory) const noexcept {
        auto shared = shared_factory();
        for(size_t i = 0; i < n; ++i) {
            f(i, shared);
        }
    }
}
Acknowledgements

Thanks to Eric Niebler, Michael Garland, Michał Dominiak, and Olivier Giroux for helpful feedback.