Document	D4007R0
Date:	2026-02-02
Reply-to:	Vinnie Falco <vinnie.falco@gmail.com>
Audience:	SG1, LEWG

std::execution Needs More Time

Abstract

`std::execution` brings valuable ideas to C++: structured concurrency, composable task graphs, and a clean separation of work description from execution policy. These contributions are real and worth preserving. However, this paper identifies a fundamental incompatibility between `std::execution`'s backward-flow context model and coroutine frame allocation that remains unresolved by P3826R3's proposed fixes. For coroutine-based I/O, allocator control is mandatory for performance, yet `std::execution` provides the allocator only *after* the coroutine frame is already allocated. We recommend deferring `std::execution` from C++26 to allow the design time to mature, so that when it ships, it can serve both CPU-bound and I/O-bound use cases with the quality the C++ community deserves.

1. What std::execution Gets Right

`std::execution` represents years of careful work and genuine progress in structured asynchronous programming for C++:

- **Structured concurrency** with well-defined lifetime guarantees for asynchronous operations
- **Composable sender/receiver pipelines** that separate work description from execution policy
- **Schedulers and execution contexts** that give callers control over where work runs
- **`run_loop`** and foundational primitives that enable deterministic testing of async code
- **A formal model** for reasoning about asynchronous completion signatures

These are real achievements. The question this paper raises is not whether `std::execution` has value—it does—but whether it is ready to serve the full scope of use cases it aspires to, particularly coroutine-based I/O.

2. The I/O Use Case

`std::execution` aspires to be a general-purpose asynchronous framework, including for networked I/O. Here is what that use case requires.

A typical networking application launches coroutines that perform I/O:

```
// User's coroutine - handles one client connection
task<void> handle_client(tcp_socket socket)
```

```
{
    char buf[1024];
    auto [ec, n] = co_await socket.async_read(buf);  // Suspends here
    if (!ec)
        process(buf, n);
}

// Application launches the coroutine
int main()
{
    io_context ctx;
    tcp_socket sock = accept_connection(ctx);
    auto alloc = ctx.get_frame_allocator();

    run_async(ctx.get_executor(), alloc)(    // Launcher knows the allocator
        handle_client(std::move(sock))       // Coroutine frame allocated HERE
    );

    ctx.run();
}
```

The critical timing:

```
run_async(executor, alloc)(handle_client(sock))
                                  │
                                  └─► promise_type::operator new runs HERE
                                      Allocator must already be known
```

When `handle_client(sock)` is evaluated, the compiler calls `promise_type::operator new` to allocate the coroutine frame. The allocator must be available at this moment—before the coroutine body executes, before any `co_await`, before any connection to a receiver.

---

3. The Cost of Uncontrolled Allocation

High-performance I/O servers cannot tolerate uncontrolled heap allocation:

```
task<void> handle_connection(tcp_socket socket)
{
    while (running) {
        co_await socket.async_read(buf);   // Frame allocated
        co_await socket.async_write(buf);  // Frame allocated
    }
}
// 1000 requests/sec × 2 operations = 2000 allocations/sec per connection
// 10,000 connections = 20 million allocations/sec
// Global heap contention becomes the bottleneck
```

The solution is **stateful allocators**:

- **Recycling allocators** — cache recently-freed frames for immediate reuse
- **Arena allocators** — tie allocation lifetime to connection scope
- **Pool allocators** — optimize for common frame sizes
- **Memory resources** — customize upstream allocation strategy
- **Standard allocator** - a useful tradeoff for resource-constrained devices

These allocators can have state: pointers to free lists, arena boundaries, pool metadata. This state must be accessible when `operator new` executes.

Different coroutine chains may require different allocation policies. A multi-tenant server might enforce per-tenant memory limits—each tenant's connections use an allocator bounded to that tenant's quota. A single global allocator cannot express these constraints.

---

4. Why HALO Cannot Help

One might ask: can't the compiler optimize away the allocation?

HALO (Heap Allocation eLision Optimization) allows compilers to elide coroutine frame allocation when the frame's lifetime is provably bounded by its caller. For I/O coroutines, this optimization cannot apply:

```
task<size_t> read_data(socket& s, buffer& buf)
{
    co_return co_await s.async_read(buf);  // Suspends for network I/O
}

void start_read()
{
    auto t = read_data(sock, buf);          // Frame allocated here
}                                           // Caller returns immediately
```

Why HALO cannot apply:

1. The coroutine suspends on I/O—waiting for a network packet
2. The caller returns immediately after initiating the operation
3. The frame must persist until the OS signals completion
4. Completion timing is unbounded—microseconds to seconds

The compiler cannot prove bounded lifetime because the lifetime depends on an external event. For any coroutine that suspends on I/O, allocation is mandatory.

SOMEONE must allocate the frame. That someone needs a stateful allocator.

---

5. The Only Customization Point

C++ provides exactly one mechanism to intercept coroutine frame allocation:

```
struct promise_type
{
    static void* operator new(std::size_t size)
    {
        // This is the ONLY place to intercept allocation
        // The allocator must be known RIGHT NOW

        auto* allocator = /* ??? */;  // Where does this come from?
        return allocator->allocate(size);
    }
};
```

The allocator—with its state—must be discoverable at this exact moment. No mechanism that provides the allocator later can help. The frame is already allocated by the time any later mechanism executes.

---

6. The allocator_arg Workaround

P3552R3 acknowledges the allocation problem:

> “When using coroutines there will probably be an allocation at least for the coroutine frame (the HALO optimisations can’t always work).”

The paper notes that existing implementations lack allocator support entirely:

> “The `unifex::task<T>` doesn’t have allocator support. When creating a task multiple objects are allocated on the heap: it seems there is a total of 6 allocations for each `unifex::task<T>` being created.”

The stdexec reference implementation fares no better:

> “Like the unifex task `exec::task<T, C>` doesn’t provide any allocator support.”

P3552’s solution uses `std::allocator_arg_t` to pass allocator information to `operator new` via the coroutine’s parameter list:

> “The idea is to pick up on a pair of arguments of type `std::allocator_arg_t` and an allocator type being passed and use the corresponding allocator if present… The arguments passed when creating the coroutine are made available to an `operator new` of the promise type.”

```
template<class... Args>
task<void> my_coro(int x, Args&&...) {
    co_return;
}

// Caller must pass allocator explicitly
co_await my_coro(42, std::allocator_arg, alloc);
```

P3552 describes how a *single* coroutine accepts an allocator but never addresses propagation through call chains. Every coroutine in a chain must accept and forward the allocator:

```
template<class... Args>
task<void> level_three(Args&&...) { co_return; }

template<class... Args>
task<void> level_two(int x, Args&&...) {
    auto alloc = co_await read_env(get_allocator);   // Query
    co_await level_three(std::allocator_arg, alloc); // Forward
}

template<class... Args>
task<int> level_one(int v, Args&&...) {
    auto alloc = co_await read_env(get_allocator);            // Query
    co_await level_two(42, std::allocator_arg, alloc);        // Forward
    co_return v;
}
```

This pattern imposes significant burdens:

- Every coroutine requires variadic template arguments
- Every coroutine must query and forward the allocator
- Function signatures no longer reflect algorithmic intent
- Forgetting to forward silently breaks the chain

The allocator becomes “viral”—polluting interfaces throughout the codebase. P3552’s silence on this propagation burden is telling: the paper demonstrates the mechanism but offers no solution for real-world call chains. This is not a practical solution for production I/O code.

---

7. A Timing Gap in std::execution’s Context Model

The sender/receiver model in `std::execution` flows context backward from receiver to sender. This is an elegant design for many use cases, but it creates a timing gap for coroutine frame allocation:

```
// std::execution timing:
task<int> my_coro();               // Step 1: operator new runs HERE
auto sndr = my_coro();             //         Frame already allocated

auto op = connect(sndr, receiver); // Step 2: Receiver connected HERE
                                   //         get_allocator(get_env(rcvr))
                                   //         available NOW—too late

start(op);                         // Step 3: Operation starts
```

P2300R4 (2022-01-18) acknowledges this timing explicitly:

> “In the sender/receiver model… contextual information is associated with the receiver and is queried by the sender and/or operation state **after** the sender and the receiver are `connect`-ed.”

P3826R3 (2026-01-05) confirms the consequence:

> "The receiver is not known during early customization. Therefore, early customization is irreparably broken."

The allocator is part of this contextual information. It becomes available only after `connect()`—but the coroutine frame was allocated in Step 1, before `connect()` was called.

For coroutine-based I/O, the allocator arrives too late. This is not a flaw in the sender/receiver formalism itself—it is a gap that emerges when that formalism meets the concrete requirements of coroutine frame allocation.

---

8. P3826R3 Addresses Algorithm Dispatch but Not Allocator Timing

P3826R3 proposes important fixes for sender algorithm customization—a real problem worth solving. However, its five proposed solutions all target algorithm dispatch. None address the allocator timing gap identified above:

Solution 4.1: Remove all std::execution

This would resolve the timing gap by deferral, giving the design time to mature before standardization.

Solution 4.2: Remove customizable sender algorithms

This removes `then`, `let_value`, `bulk`, etc. It does not change when the allocator becomes available. The timing problem remains.

Solution 4.3: Remove sender algorithm customization

This removes the ability to customize algorithms. It does not change when the allocator becomes available. The timing problem remains.

Solution 4.4: Ship as-is, fix via DR

This defers the fix. It does not change when the allocator becomes available. The timing problem remains.

Solution 4.5: Fix algorithm customization now

P3826's proposed fix passes the receiver's environment when querying a sender for its completion domain:

```
// P3826's fix:
get_completion_domain<set_value_t>(get_env(sndr), get_env(rcvr))
```

This tells senders where they will start, enabling correct algorithm dispatch. **It does not change when the allocator becomes available.** The receiver's environment—including the allocator—is still only queryable after `connect()`.

P3826 fixes an important question: "which algorithm implementation should run." It does not address a different but equally important question: "when is the allocator available." The timing gap remains open.

9. Conclusion

`std::execution` represents important progress in structured asynchronous programming for C++. Its contributions to the language—structured concurrency, composable pipelines, and a formal model for async completion—are valuable and worth building on.

However, for coroutine-based asynchronous I/O, a fundamental timing gap remains:

1. **Allocation is mandatory** — HALO cannot optimize away frames that outlive their callers
2. **Stateful allocators are required** — 20 million allocations/sec demands recycling, pooling, or arena strategies
3. **`operator new` is the only customization point** — the allocator must be known when the frame is allocated
4. **`std::execution` provides the allocator too late** — receiver environment is available only after `connect()`
5. **P3826's solutions do not address this** — they fix algorithm dispatch, not allocator timing

P2300R4 established the sender/receiver context model in January 2022. Four years later, P3826R3 acknowledges that "early customization is irreparably broken"—the author's own characterization of design issues discovered during implementation. The design continues to evolve: P3826R3 proposes significant architectural changes to `transform_sender`, removes early customization entirely, and restructures the relationship between `continues_on` and `schedule_from`. This level of active change is healthy for a maturing design, but it signals that the work is not yet finished.

Standardizing `std::execution` now would lock in an API before the I/O story is resolved. Deferring it gives the authors time to address the allocator timing gap, stabilize the architecture, and deliver an API that serves the full range of use cases—CPU-bound parallelism and I/O-bound networking alike—with the quality and stability the C++ community expects from the standard library.

We recommend deferring `std::execution` from C++26 to allow the design to mature.

References

- [P3826R3] Eric Niebler. "Fix Sender Algorithm Customization." 2026-01-05.
- [P2300R10] Michał Dominiak, et al. "std::execution."
- [P3552R3] Dietmar Kühl, Maikel Nadolski. "Add a Coroutine Task Type."
- [P4003R0] Vinnie Falco. "IoAwaitables: A Coroutines-First Execution Model"