constexpr ‘parallel’ algorithms?

Oliver Rosten

The conundrum

- `constexpr` is to facilitate compile-time programming
  - Relatively small computations
  - Must be structured enough not to require runtime input

- Parallel execution is for runtime acceleration
  - Medium through to extremely large computations
  - Inputs may be completely unstructured and likely determined at runtime

- Is there any overlap?
  - And, if yes, is the overlap big enough to be interesting?

The Case Against

- The intersection of these two cases is either
  - Non-existent [Rebuttal to follow]
  - Too small to be worthwhile [A matter of taste – that's why we're discussing this]

- Making the parallel algorithms `constexpr` requires
  - Work [Relatively easy, but tedious]
  - Testing
  - Maintenance

- If people want to do it, it's easy for them
  - Just an `if consteval` away [No longer strictly true, but still not hard]

- The cost/benefit ratio is considered too high to merit standardization

The Case For

- Regularity of the standard library
- Getting this right is no longer ***entirely*** trivial
- Opaque Usage of execution policies
- Lowering the path of resistance for compile time testing
- There are realistic use cases

Regularity of the Standard Library

- The direction of evolution of C++ has been to `constexpr` all the things

- For C++ 26 (apologies if I've missed anything):
  - Language
    - `placement new` [P2747R2]
    - casting from `void*` [P2738R1]
    - throwing exceptions [P3068R6]
  - Library
    - A greater range of `cmath` and `complex` functions [P1383R2]
    - The stable sorting algorithms [P2562R1]
    - The specialized memory algorithms [P2283R0, P3508R0]
    - `atomic` and `atomic_ref` [P3309R3]
    - Exception types [P3068R6, P3378R2]
    - Containers and adaptors [P3372R2]
    - `inplace_vector` for non-trivial types [P3074R7]

- Excluding the parallel algorithms carves out an irregular corner
  - `atomic` gives precedent for including things associated with runtime concurrency/parallelism
  - We may be getting `constexpr` coroutines in C++29

Getting `constexpr` parallel algos right is no longer trivial

- In essence, the implementation is an `if consteval` branch

```
if consteval { std::sort(first, last); }
else         { std::sort(exec, first, last); }
```

- But we can now throw exceptions during constant evaluation
  - The ***runtime*** semantic of the parallel algos is that uncaught exceptions → termination
  - If the above code is executed at compile time we get ***different semantics***
    - An exception will escape
    - It could be caught within the cone of constant evaluation, giving a constant expression

- A DIY approach to `constexpr` parallel algos could easily miss this subtle point

Opaque usage

- Functions may use a parallel algorithm under the bonnet

```
float sum(std::span<const float> s) { std::reduce(std::execution::par, s.begin(), s.end()); }
```

- To make this `constexpr` currently requires the DIY approach

```
constexpr float sum(std::span<const float> s) {
    if consteval {
        try {
            return std::reduce(s.begin(), s.end());
        }
        catch(...) {
            std::terminate();
        }
    }
    else {
        return std::reduce(std::execution::par, s.begin(), s.end());
    }
};
```

Hard to love.

`constexpr` tests

- Parallel algos will likely delegate to sequential ones
- So creating `constexpr` tests won’t probe the parallelized behaviour
- But there’s still value, to pick up some forms of UB

Possible Scenario

- I've unwittingly created UB

```
float sum(std::span<const float> s) { std::reduce(std::execution::par, s.end(), s.begin()); }
```

- In production, I only use this at runtime and it's a faff to make it `constexpr`

- But if only I had, a very simple test would have caught this

```
constexpr auto x = sum({std::array{1.0f}});
```

UB in ambient code

```
void foo() {
constexpr void foo() {
    // UB
    // Parallel algo
}
```

- The UB cannot be caught during constant evaluation (no `constexpr`)
- Making the parallel algos `constexpr` ⇒ path of very low resistance
    - Now the UB can be caught during constant evaluation with a simple test
    - People are more likely to do this, the easier we make it for them

Realistic Use Case 1: Polygons

- Triangles/quads are useful primitives for graphics
  - Why not create the vertices at compile time?

- Large-n polys are useful for e.g. computational geometry
  - Why not accelerate their runtime creation?

In Code

`array/vector` depending on the size

Rule for creating the container

Algorithm for generating verts

It would be nice to simply add this!

```
template<std::floating_point T>
struct vertex{ T x{}, y{}; };

template<std::floating_point T, std::size_t N>
class polygon {
    constexpr static bool small{N <= 64};
    using container_t = std::conditional_t<small, std::array<vertex<T>, N>, std::vector<vertex<T>>>;

    constexpr static container_t make_storage() {
        if constexpr(small)
            return container_t{};
        else
            return container_t(N);
    }

    container_t m_Values{};

    template<std::ranges::random_access_range R>
    constexpr static R&& build_poly(R&& r) {
        constexpr auto pi{std::numbers::pi_v<T>};
        std::ranges::transform(
            std::execution::par,
            std::views::iota(0uz, N),
            r.begin(),
            [](std::size_t i) -> vertex<T> {
                const auto theta{2 * pi * i / N};
                return {-std::sin(theta), std::cos(theta)};
            });

        return r;
    }
public:
    constexpr polygon() : m_Values{build_poly(make_storage())} {}
};
```

The Extra Wrinkle, if approved

- In ***principle*** we may open the door to constant evaluation on multiple threads

- The ***possibility*** could have ramifications for existing library implementations
    - Simple increments of `atomic` in an `if consteval` branch may have to be rethought

Conclusion

- There are good arguments both for* and against

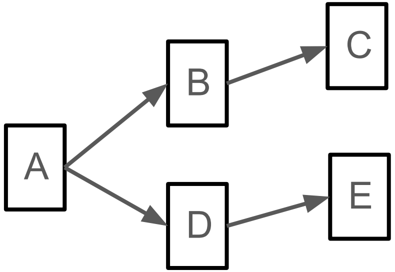
- There are benefits to making the parallel algorithms `constexpr`

- The question is whether the benefits are worth the effort

*Additional use case in the appendix

Realistic Use Case 2: Task-oriented programming

- Consider a dependency graph of tasks



- In memory, this could be laid out depth-first

Nodes	A	B	C	D	E

Edges per Node	2	1	0	1	0

Edge targets	1	3	2	4
	A→B	A→D	B→C	D→E

Data Transformations

- Topological sort, to obtain an execution order in terms of node indices [DFS; no acceleration]

- Set auxiliary data on each node [`transform`, may be accelerated]
  - Current index in storage
  - Execution order

- sort the new nodes, using the execution order [`sort`, may be accelerated]

- Fix-up the connectivity data to maintain the graph invariant [`transform`, may be accelerated]

- Repeatedly execute tasks with met dependencies not yet executed [inner `for_each` may be accelerated]