Round 2: Standard parallelism. Can we?
What this presentation is or is not?
Outline

std::simd
Abstraction for vectorization

Parallel Algorithms
First step towards standard support for parallelism

std::execution
Going further with design opportunities
SIMD abstractions
std::simd in C++26... maybe...

Why?

Performance!

But how?

By hand, with intrinsics... maybe no.

Let the compiler do it! Ok but... maybe no.

First step

So, can I write SIMD code like scalar code?
std::simd in C++26... maybe...

Developers libraries

Yes, we can ease this process with third party libraries.

There is history here.

Eve (boost.simd), xsimd, etc...

Should the standard jump in the train?

Yes it has, some work has been done in the Parallelism TS v2

Now targeting C++26
A data parallel type, defined as a class template of type T.

Width of a given simd instantiation is a constant expression, determined by the template parameters.

```
std::simd<T>
```

Arithmetic operators, comparisons, masking abilities, ABI compatibility concerns, small amount of simd aware algorithms ...

Intrinsics wrappers or more?

Will `std::simd` be on par with developer libraries?

Is this a problem if ...

We can go further with this?

Parallel execution policies and constrained algorithms.
Parallel algorithms
Standard algorithms

Works sequentially on iterators.

```cpp
std::transform( std::begin(my_container), std::end(my_container)
               , [] (auto e) { return std::cos(e); })
```

Great, standard sequential code...

But can I have a free parallel version of my code?
Standard parallel algorithms

Yes, since C++17 and parallel policies.

```cpp
std::transform( std::execution::par
    , std::begin(my_container), std::end(my_container)
    , [](auto e){ return std::cos(e); })
```

Different execution policies available

- `std::execution::seq` -> *op in the calling thread, indeterminately sequenced*
- `std::execution::unseq` -> *op in the calling thread, unsequenced*
- `std::execution::par` -> *potentially in multiple threads, indeterminately sequenced within each threads*
- `std::execution::par_unseq` -> *potentially in multiple threads, unsequenced*
Standard parallel algorithms

Fork and join.

Pretty cool but, what about merging two steps?

algorithm 1:
std::transform(..., f)

algorithm 2:
std::for_each(..., g)
Standard parallel algorithms

Yes, we can go further with the laziness of range adaptors from C++20.

```cpp
std::vector x{...};
auto v = std::views::transform(x, f);
std::for_each(std::begin(v), std::end(v), g);
```
Standard parallel algorithms

Now, what are we missing?

Unlock the fork and join model
Get rid of possible latencies
Say where things should execute

So, what do we need?

A model for asynchrony
A way to attach work to a computing resource
std::execution: Senders and Receivers
The standard answers the previous concerns with:

Senders and Receivers

Targeting C++26

Reference implementation available from NVIDIA.
namespace ex = std::execution;
// retrieve a scheduler
ex::scheduler auto sch = thread_pool.scheduler();
// start chain of work
ex::sender auto begin = ex::schedule(sch);
// compose work on top of the first sender,
// return a new sender that will complete on the same execution context
ex::sender auto hi = ex::then(begin, [](){ return 13; });
// adding more work here
ex::sender auto add = ex::then(hi, [](int a){ return a + 42; });
// we finally wait for completion
auto [res] = std::this_thread::sync_wait(add).value();
Schedulers are handles to execution contexts

Senders represent asynchronous work

Receivers process asynchronous signals
std::execution: Senders and Receivers
Schedulers **produce** senders that will produce work on the execution contexts **attach** to the schedulers. Once we have a **sender**, we can compose work on it.
std::execution: Senders and Receivers

We get senders with sender factories.

```cpp
schedule(), just(), transfer_just()...
```

We compose work on senders with sender adaptors.

```cpp
then(), bulk(), on(), transfer(), split(), when_all(), ensure_started()...
```

We start work by connecting sender graphs with sender consumers.

```cpp
sync_wait(), start_detached(), execute()
```
std::execution: Senders and Receivers

Senders and Receivers are compatible with C++20 std::coroutines

Coroutines are stackless functions that can suspend execution to be resumed later.

Distributed memory support?

It should, but not in a transparent way. (Extensions? Runtime support under the hood? *)

Error handling support

By design.

User facing design / Implementer facing design

Implementer side is open for adding support and extensions. (*)

Remember? I never talked about receivers... :)

Ínria
Let’s wrap it up!

Quick reminder: `std::span` (20), `std::mdspan` (23), `std::blas` (26)

Parallel algorithms available since C++17.

Targeting C++26

- `std::execution`:
  - reference implementation available, NVIDIA
  - designed to be composable and extended

- `std::simd`:
  - work available in Parallelism TS v2 (gcc11, clang in progress)
Let’s wrap it up!

**Concerns about:**

- Is the scope of these works covering our needs?
- If not, the c++ community may have already started some works on the side of the standard. Round 3?
- What are your concerns/ideas? Round 3!

Going further: standard c++ ecosystem starts to be powerful right!? Let’s see if we can make it even more...
To be continued...

...in round 3.

Any thoughts or questions?