



STANDARD PARALLELISM

Bryce Adelstein Lelbach

Principal Architect

Standard C++ Library Evolution Chair, US Programming Languages Chair



@blelbach

Conventions

Conventions

```
namespace stdv = std::views;
```

```
namespace stdr = std::ranges;
```

```
namespace ex = std::execution;
```

```
namespace this_thread = std::this_thread;
```

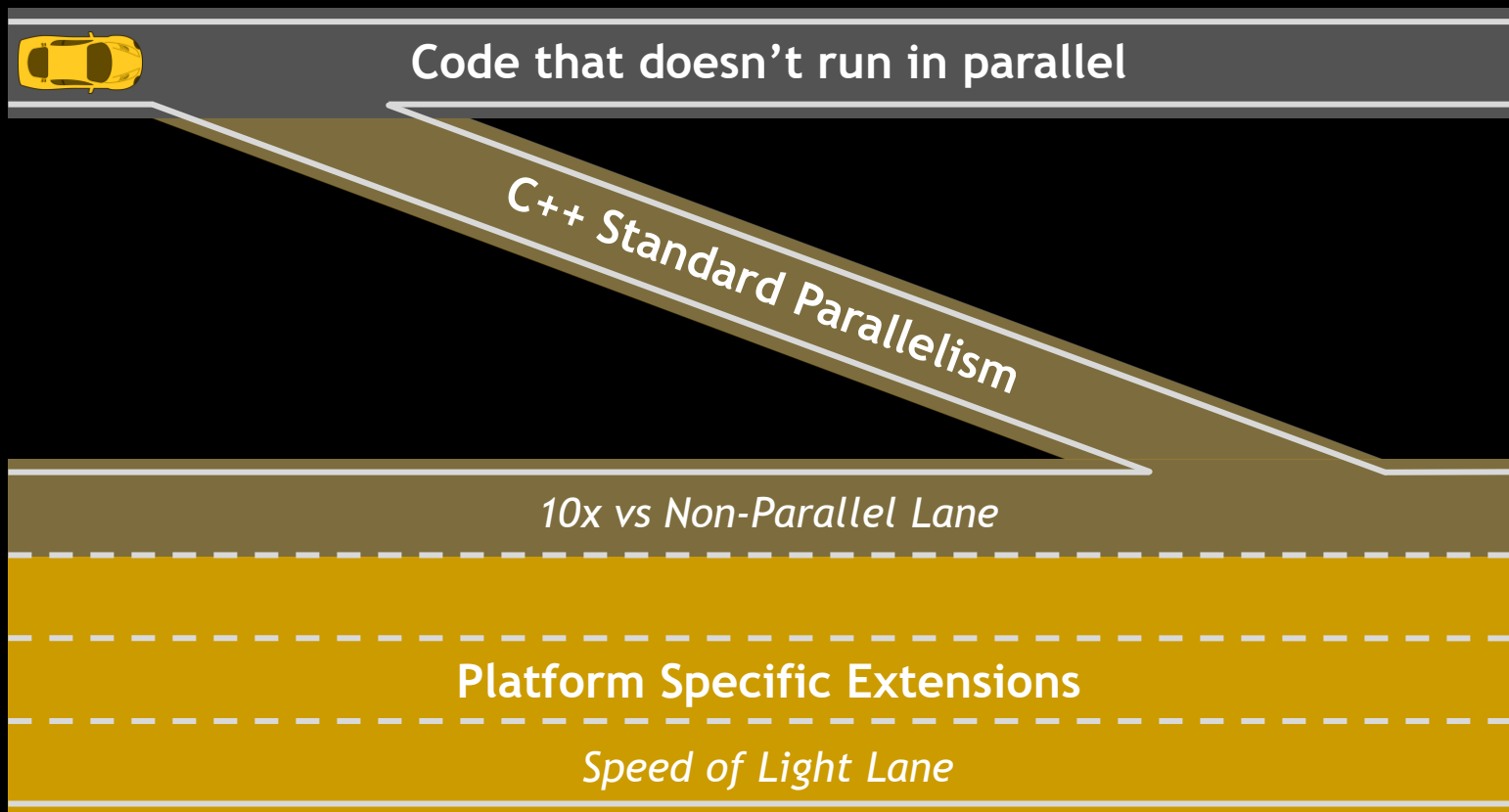
Conventions

```
namespace stdv = std::views;  
namespace stdr = std::ranges;  
namespace ex = std::execution;  
namespace this_thread = std::this_thread;
```

Class Template Argument Deduction (CTAD)

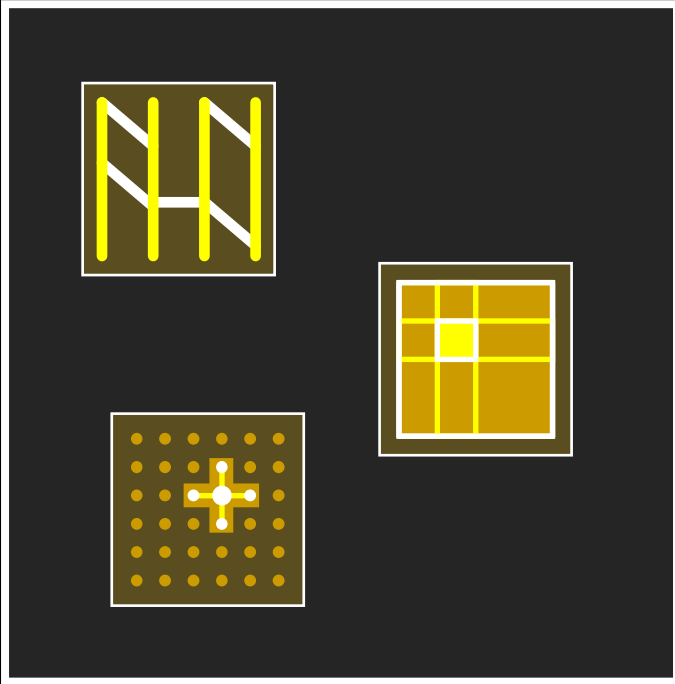
```
std::tuple t{3.14, 42}; → std::tuple<double, int>  
std::array a{0, 1, 1, 0}; → std::array<int, 4>
```

We Need On-Ramps



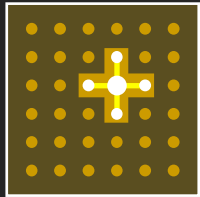
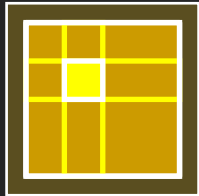
Pillars of Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



Pillars of Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



Tools to Write Your Own Parallel Algorithms that Run Anywhere

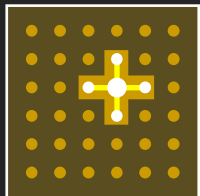
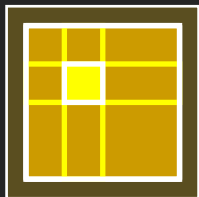


```
sender auto  
algorithm (sender auto s) {  
    return s | bulk(N,  
        [] (auto data) {  
            // ...  
        }  
    ) | bulk(N,  
        [] (auto data) {  
            // ...  
        }  
    );  
}
```



Pillars of Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



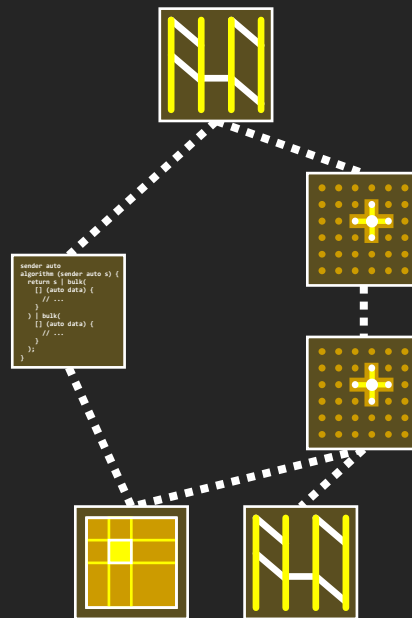
Tools to Write Your Own Parallel Algorithms that Run Anywhere



```
sender auto  
algorithm (sender auto s) {  
    return s | bulk(N,  
        [] (auto data) {  
            // ...  
        }) | bulk(N,  
        [] (auto data) {  
            // ...  
        });  
}
```



Mechanisms for Composing Parallel Invocations into Task Graphs





Standard Algorithms

Serial (C++98)

```
std::vector<T> x{...};
```

```
std::for_each(  
    begin(x), end(x),  
    f);
```

```
std::for_each(  
    begin(x), end(x),  
    g);
```

```
std::for_each(  
    begin(x), end(x),  
    h);
```

```
std::vector<double> x{...}, y{...};  
double dot_product = std::transform_reduce(begin(x), end(x),  
                                           begin(y));
```

```
std::span<std::string_view> s{...};  
std::sort(begin(s), end(s));
```

```
std::unordered_map<std::string_view, int> db{...};  
std::vector<std::pair<std::string_view, int>> m{...};  
std::copy_if(begin(db), end(db), begin(m),  
             [] (auto e) { return e.second > 0; });
```



Standard Algorithms

<code>adjacent_difference</code>	<code>is_sorted[_until]</code>	<code>rotate[_copy]</code>
<code>adjacent_find</code>	<code>lexicographical_compare</code>	<code>search[_n]</code>
<code>all_of</code>	<code>max_element</code>	<code>set_difference</code>
<code>any_of</code>	<code>merge</code>	<code>set_intersection</code>
<code>copy[_if _n]</code>	<code>min_element</code>	<code>set_symmetric_difference</code>
<code>count[_if]</code>	<code>minmax_element</code>	<code>set_union</code>
<code>equal</code>	<code>mismatch</code>	<code>sort</code>
<code>fill[_n]</code>	<code>move</code>	<code>stable_partition</code>
<code>find[_end _first_of _if _if_not]</code>	<code>none_of</code>	<code>stable_sort</code>
<code>for_each</code>	<code>nth_element</code>	<code>swap_ranges</code>
<code>generate[_n]</code>	<code>partial_sort[_copy]</code>	<code>transform</code>
<code>includes</code>	<code>partition[_copy]</code>	<code>uninitialized_copy[_n]</code>
<code>inplace_merge</code>	<code>remove[_copy _copy_if _if]</code>	<code>uninitialized_fill[_n]</code>
<code>is_heap[_until]</code>	<code>replace[_copy _copy_if _if]</code>	<code>unique</code>
<code>is_partitioned</code>	<code>reverse[_copy]</code>	<code>unique_copy</code>



Standard Algorithms

Serial (C++98)

```
std::vector<T> x{...};

std::for_each(
    begin(x), end(x),
    f);

std::for_each(
    begin(x), end(x),
    g);

std::for_each(
    begin(x), end(x),
    h);
```

Parallel (C++17)

```
std::vector<T> x{...};

std::for_each(
    ex::par_unseq,
    begin(x), end(x),
    f);

std::for_each(
    ex::par_unseq,
    begin(x), end(x),
    g);

std::for_each(
    ex::par_unseq,
    begin(x), end(x),
    h);
```

```
std::vector<double> x{...}, y{...};  
double dot_product = std::transform_reduce(ex::par_unseq,  
                                              begin(x), end(x),  
                                              begin(y));
```

```
std::span<std::string_view> s{...};  
std::sort(ex::par_unseq, begin(s), end(s));
```

```
std::unordered_map<std::string_view, int> db{...};  
std::vector<std::pair<std::string_view, int>> m{...};  
std::copy_if(ex::par_unseq, begin(db), end(db), begin(m),  
            [] (auto e) { return e.second > 0; });
```

Execution Policy	Operations occur ...	Operations are ...
------------------	----------------------	--------------------

Execution Policy	Operations occur ...	Operations are ...
<code>std::execution::seq</code>	In the calling thread	Indeterminately sequenced

Execution Policy	Operations occur ...	Operations are ...
<code>std::execution::seq</code>	In the calling thread	Indeterminately sequenced
<code>std::execution::unseq</code>	In the calling thread	Unsequenced

Execution Policy	Operations occur ...	Operations are ...
<code>std::execution::seq</code>	In the calling thread	Indeterminately sequenced
<code>std::execution::unseq</code>	In the calling thread	Unsequenced
<code>std::execution::par</code>	Potentially in multiple threads	Indeterminately sequenced within each thread

Execution Policy	Operations occur ...	Operations are ...
<code>std::execution::seq</code>	In the calling thread	Indeterminately sequenced
<code>std::execution::unseq</code>	In the calling thread	Unsequenced
<code>std::execution::par</code>	Potentially in multiple threads	Indeterminately sequenced within each thread
<code>std::execution::par_unseq</code>	Potentially in multiple threads	Unsequenced

```
std::size_t word_count(std::string_view s) {  
    ...  
}
```

```
std::string_view frost = "Whose woods these are I think I know.\n"  
                        "His house is in the village though; \n"  
                        "He will not see me stopping here \n"  
                        "To watch his woods fill up with snow.\n";  
  
std::size_t result = word_count(frost);
```

```
std::size_t word_count(std::string_view s) {  
    if (s.empty()) return 0;  
    return std::transform_reduce(ex::par_unseq, ...);  
}
```

```
std::string_view frost = "Whose woods these are I think I know.\n"  
                        "His house is in the village though; \n"  
                        "He will not see me stopping here \n"  
                        "To watch his woods fill up with snow.\n";  
  
std::size_t result = word_count(frost);
```

```
std::size_t word_count(std::string_view s) {  
    if (s.empty()) return 0;  
    return std::transform_reduce(ex::par_unseq,  
        begin(s), end(s) - 1, begin(s) + 1,  
        ...  
    );  
}
```

```
std::string_view frost = "Whose woods these are I think I know.\n"  
                        "His house is in the village though; \n"  
                        "He will not see me stopping here \n"  
                        "To watch his woods fill up with snow.\n";  
  
std::size_t result = word_count(frost);
```

```
std::size_t word_count(std::string_view s) {  
    if (s.empty()) return 0;  
    return std::transform_reduce(ex::par_unseq,  
        begin(s), end(s) - 1, begin(s) + 1,  
        ...  
    );  
}
```

```
std::string_view frost = "Whose woods these are I think I know.\n"  
                        "His house is in the village though; \n"  
                        "He will not see me stopping here \n"  
                        "To watch his woods fill up with snow.\n";  
  
std::size_t result = word_count(frost);
```

```

std::size_t word_count(std::string_view s) {
    if (s.empty()) return 0;
    return std::transform_reduce(ex::par_unseq,
        begin(s), end(s) - 1, begin(s) + 1,
        ...

    [](char l, char r) { return std::isspace(l) && !std::isspace(r); }
);
}

```

```

std::string_view frost = "Whose woods these are I think I know.\n"
                        "His house is in the village though; \n"
                        "He will not see me stopping here \n"
                        "To watch his woods fill up with snow.\n";

```

```

std::size_t result = word_count(frost);

```

```

std::size_t word_count(std::string_view s) {
    if (s.empty()) return 0;
    return std::transform_reduce(ex::par_unseq,
        begin(s), end(s) - 1, begin(s) + 1,
        ...

    [](char l, char r) { return std::isspace(l) && !std::isspace(r); }
    );
}

```

```

std::size_t result      =  0000010000010000010001010000010100000
                           1000100000100100100010000000100000000
                           100100001000100010010010000000010000000
                           10010000010001000001000010010000100000;

```



```

std::size_t word_count(std::string_view s) {
    if (s.empty()) return 0;
    return std::transform_reduce(ex::par_unseq,
        begin(s), end(s) - 1, begin(s) + 1,
        std::size_t(!std::isspace(s.front())) ? 1 : 0),
        ...
        [] (char l, char r) { return std::isspace(l) && !std::isspace(r); }
    );
}

```

```

std::size_t result      = 10000010000010000010001010000010100000
                          10001000001001001000100000001000000000
                          100100001000100010010010000000001000000000
                          10010000010001000001000010010000100000;

```

```

std::size_t word_count(std::string_view s) {
    if (s.empty()) return 0;
    return std::transform_reduce(ex::par_unseq,
        begin(s), end(s) - 1, begin(s) + 1,
        std::size_t(!std::isspace(s.front())) ? 1 : 0),
        std::plus(),
        [] (char l, char r) { return std::isspace(l) && !std::isspace(r); }
    );
}

```

```

std::size_t result      = 1  + 1  + 1  + 1 + 1+1  + 1+1  +
                          1 + 1  + 1 +1 +1 + 1  + 1  +
                          1 +1  + 1 + 1 + 1 +1  + 1  +
                          1 +1  + 1 + 1  + 1  + 1 +1  + 1 ;

```

```
std::size_t word_count(std::string_view s) {  
    if (s.empty()) return 0;  
    return std::transform_reduce(ex::par_unseq,  
        begin(s), end(s) - 1, begin(s) + 1,  
        std::size_t(!std::isspace(s.front())) ? 1 : 0),  
        std::plus(),  
        [] (char l, char r) { return std::isspace(l) && !std::isspace(r); }  
    );  
}
```

```
std::string_view frost = "Whose woods these are I think I know.\n"  
    "His house is in the village though; \n"  
    "He will not see me stopping here \n"  
    "To watch his woods fill up with snow.\n";
```

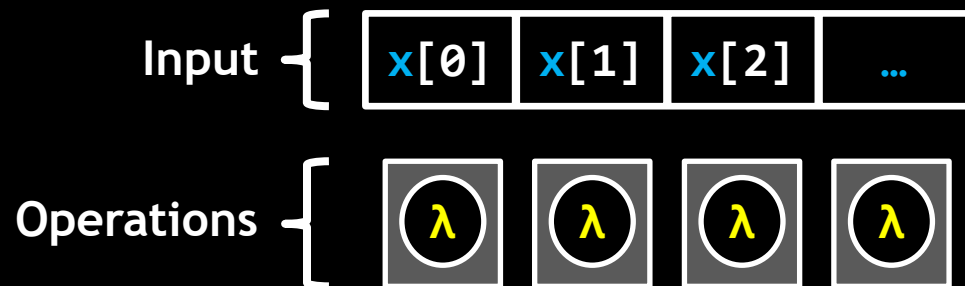
```
std::size_t result = word_count(frost);
```

**In C++20, the Standard Library
introduced ranges.**

**Unlike iterators, ranges are
composable and can be lazy.**

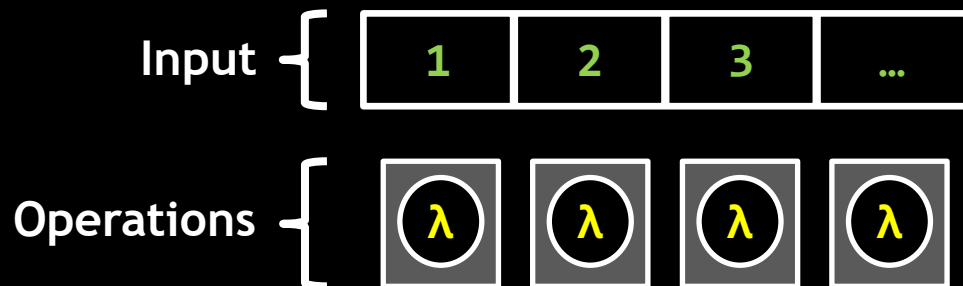
```
std::vector x{...};
```

```
std::for_each(  
    ex::par_unseq,  
    begin(x), end(x),  
    [...] (auto& obj) { ... } );
```



```
auto v = stdv::iota(1, N);
```

```
std::for_each(  
    ex::par_unseq,  
    begin(v), end(v),  
    [...] (auto idx) { ... });
```



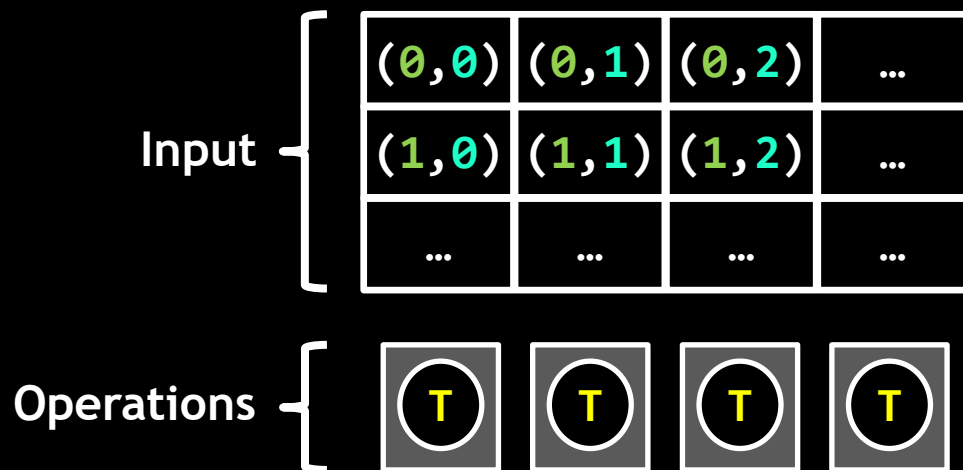
```
std::span A{input,  N * M};
std::span B{output, M * N};

auto v = stdv::cartesian_product(
    stdv::iota(0, N),
    stdv::iota(0, M));

std::for_each(ex::par_unseq,
    begin(v), end(v),
    [=] (auto idx) {
        auto [i, j] = idx;
        B[i + j * N] = A[i * M + j];
    });
```

```
std::span A{input,  N * M};  
std::span B{output, M * N};
```

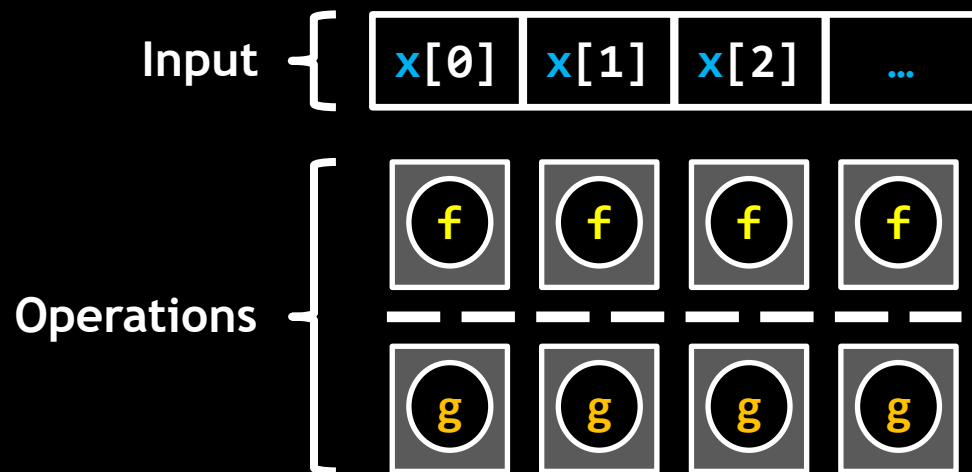
```
auto v = stdv::cartesian_product(  
    stdv::iota(0, N),  
    stdv::iota(0, M));
```




```
std::vector x{...};
```

```
std::for_each(ex::par unseq,  
              begin(x), end(x), f);
```

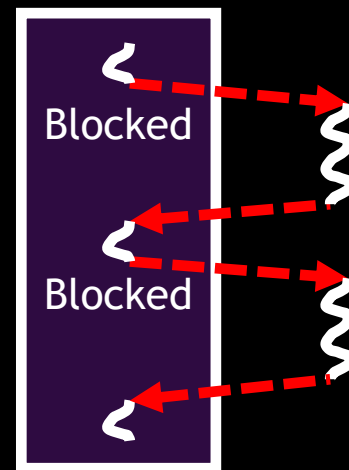
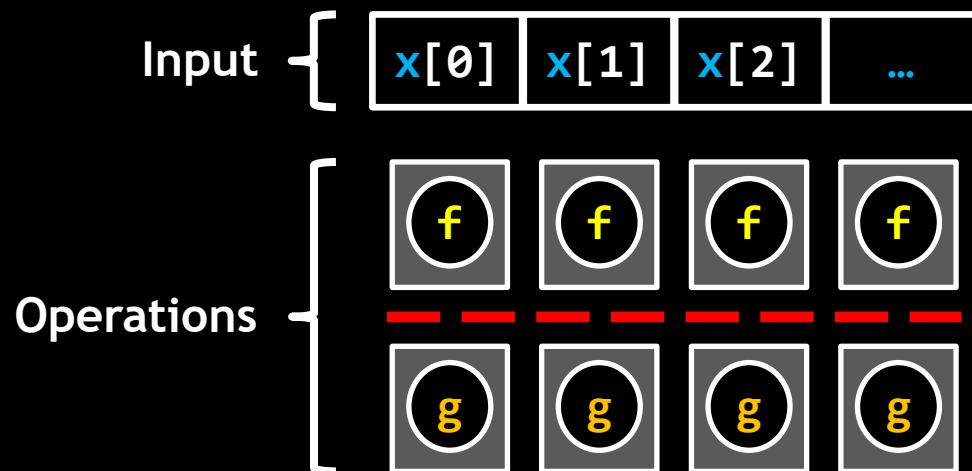
```
std::for_each(ex::par unseq,  
              begin(x), end(x), g);
```



```
std::vector x{...};
```

```
std::for_each(ex::par_unseq,  
             begin(x), end(x), f);
```

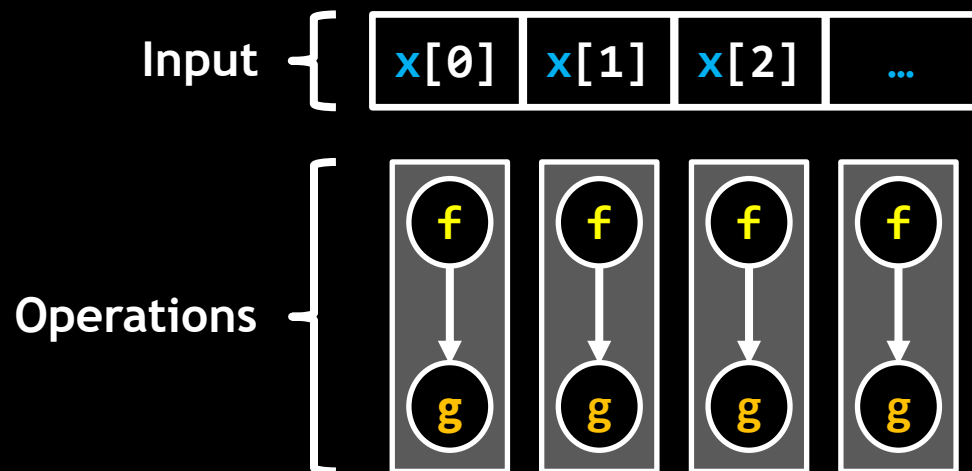
```
std::for_each(ex::par_unseq,  
             begin(x), end(x), g);
```



```
std::vector x{...};
```

```
auto v = std::transform(x, f);
```

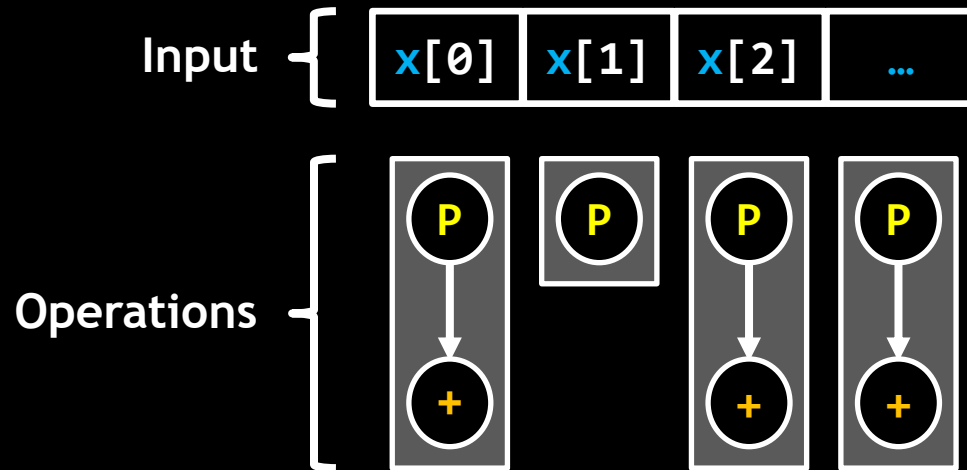
```
std::for_each(ex::par_unseq,  
             begin(v), end(v), g);
```



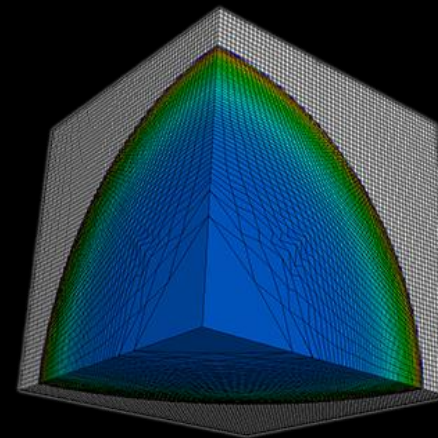
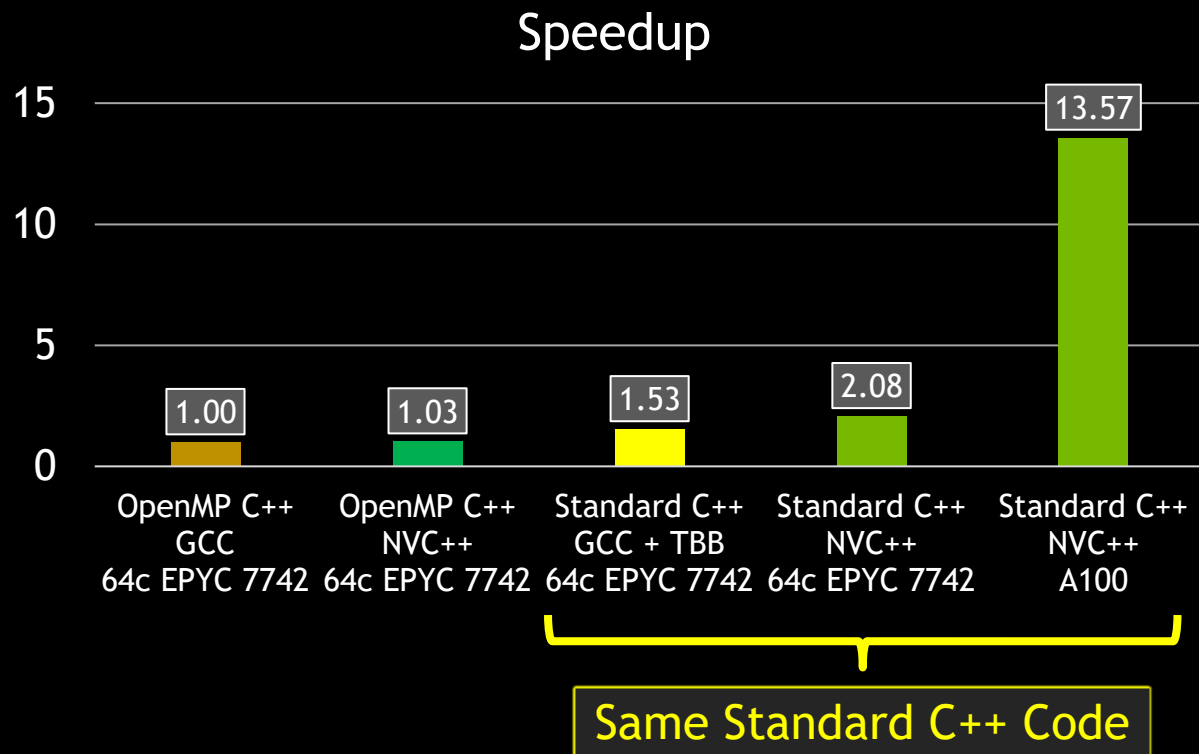
```
std::vector x{...};
```

```
auto v = std::filter(x,  
    [] (auto e) { return e > 0; });
```

```
std::reduce(ex::par_unseq,  
    begin(v), end(v));
```



LULESH

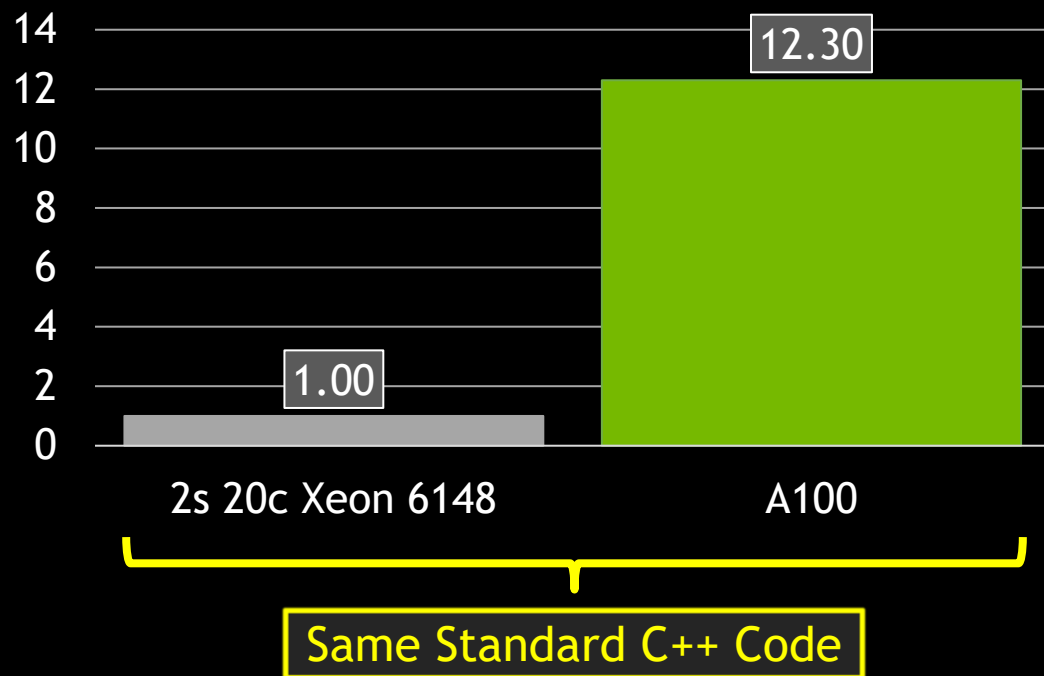


- Mini app for Lagrangian explicit shock hydrodynamics on an unstructured grid.
- Designed to stress vectorization, parallel overheads, & on-node parallelism.
- ~9000 lines of C++.
- Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos, Standard C++, ...

<https://github.com/LLNL/LULESH>

STLBM

Collision Models Speedup



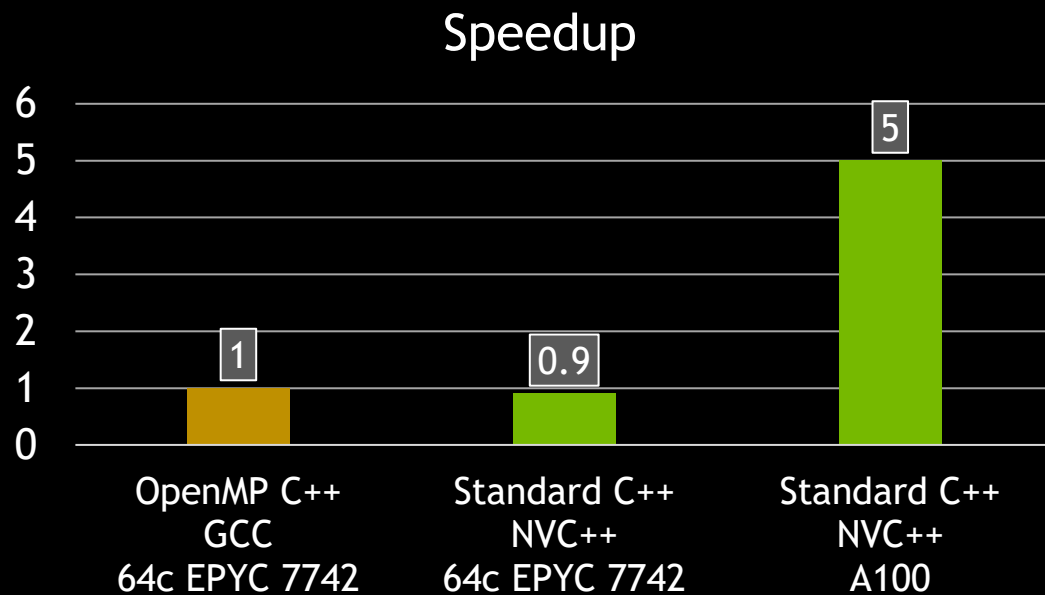
- Framework for parallel Lattice-Boltzmann simulations on multiple targets, including multicore CPUs & GPUs.
- Implemented with C++ Standard Parallelism.
- No language extensions, external libraries, vendor-specific code annotations, or pre-compilation steps.

*"We have with delight discovered the NVIDIA "stdpar" implementation of C++ Standard Parallel Algorithms. ... We believe that the result produces state-of-the-art performance, is highly didactical, and introduces **a paradigm shift in cross-platform CPU/GPU programming** in the community."*

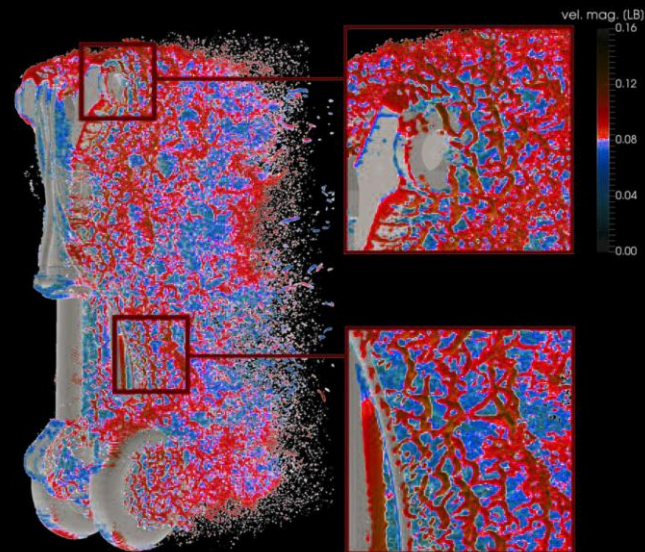
— Professor Jonas Latt, University of Geneva

<https://gitlab.com/unigehpfs/stlbn>

M-AIA



Same Standard C++ Code



- Package for aerospace flow and noise simulations.
- Solvers include Finite Volume, Navier-Stokes, and Lattice-Boltzmann.
- Switching from OpenMP to C++ Standard Parallelism.

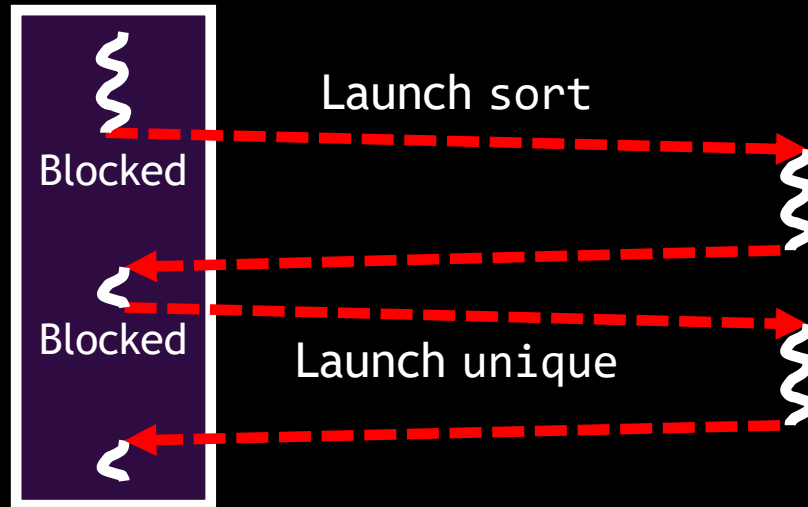
**The C++ parallel algorithms
introduced in C++17 are great,
but they're just the
start of the story.**


```
std::vector<std::string_view> s{...};  
  
std::sort(ex::par_unseq, begin(s), end(s));  
std::unique(ex::par_unseq, begin(s), end(s));
```

```
std::vector<std::string_view> s{...};
```

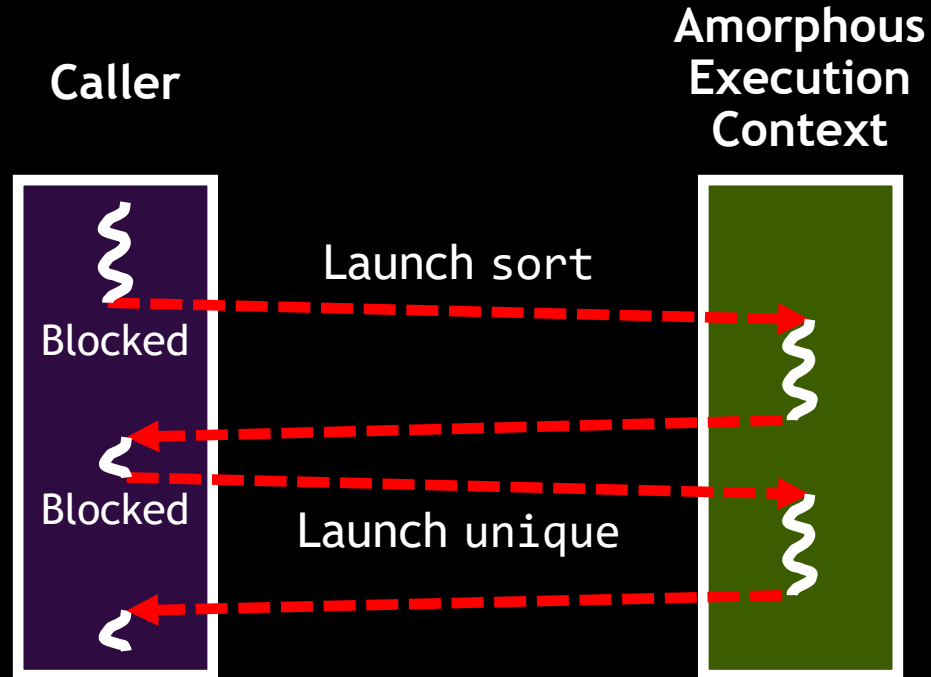
```
std::sort(ex::par_unseq, begin(s), end(s));  
std::unique(ex::par_unseq, begin(s), end(s));
```

Caller



```
std::vector<std::string_view> s{...};
```

```
std::sort(ex::par_unseq, begin(s), end(s));  
std::unique(ex::par_unseq, begin(s), end(s));
```



Today, C++ has:

- No standard model for asynchrony.
- No standard way to express where things should execute.

Today, C++ has:

- No standard model for asynchrony.
- No standard way to express where things should execute.

The solution is coming soon:

Senders & Receivers

```
ex::scheduler auto sch = thread_pool.scheduler();  
  
ex::sender auto begin = ex::schedule(sch);  
ex::sender auto hi     = ex::then(begin, [] { return 13; });  
ex::sender auto add    = ex::then(hi, [] (int a) { return a + 42; });  
  
auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();
```

```
ex::sender auto begin = ex::schedule(sch);
```

```
ex::sender auto hi      = ex::then(begin, [] { return 13; });
```

```
ex::sender auto add     = ex::then(hi, [] (int a) { return a + 42; });
```

```
auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();
```

```
ex::sender auto begin = ex::schedule(sch);
```

```
ex::sender auto hi      = ex::then(begin, [] { return 13; });
```

```
ex::sender auto add     = ex::then(hi, [] (int a) { return a + 42; });
```

```
auto [i] = this_thread::sync_wait(add).value();
```



```
ex::scheduler auto sch = thread_pool.scheduler();
```

```
ex::sender auto begin = ex::schedule(sch);
```

```
ex::sender auto hi     = ex::then(begin, [] { return 13; });
```

```
ex::sender auto add     = ex::then(hi, [] (int a) { return a + 42; });
```

```
auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();  
  
ex::sender auto begin = ex::schedule(sch);  
ex::sender auto hi      = ex::then(begin, [] { return 13; });  
ex::sender auto add     = ex::then(hi, [] (int a) { return a + 42; });  
  
auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();  
  
ex::sender auto begin = ex::schedule(sch);  
ex::sender auto hi     = ex::then(begin, [] { return 13; });  
ex::sender auto add    = ex::then(hi, [] (int a) { return a + 42; });  
  
auto [i] = this thread::sync wait(add).value();
```

Schedulers are handles to execution contexts.

Schedulers are handles to execution contexts.

Senders represent asynchronous work.

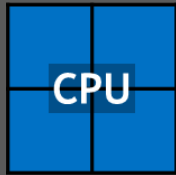
Schedulers are handles to execution contexts.

Senders represent asynchronous work.

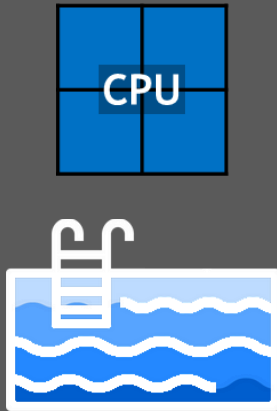
Receivers process asynchronous signals.

**Schedulers are handles to
execution contexts.**

Execution Context: CPU Thread Pool



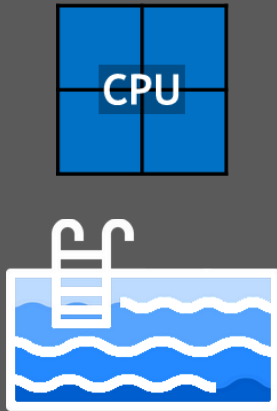
Execution Context:
CPU Thread Pool



Execution Context:
GPU Stream



Execution Context:
CPU Thread Pool

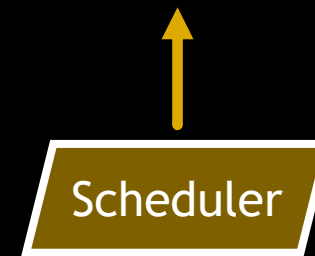
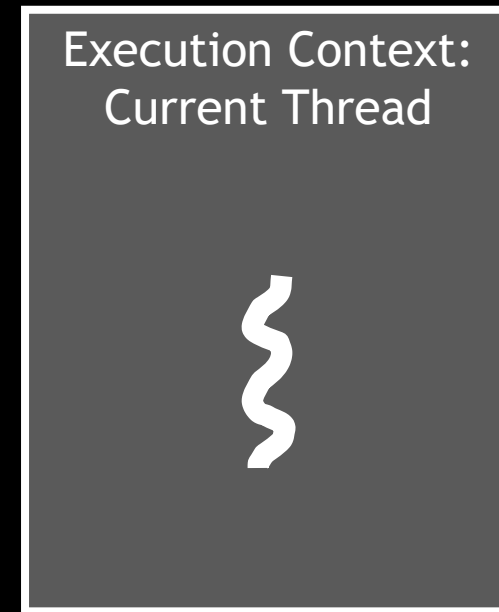
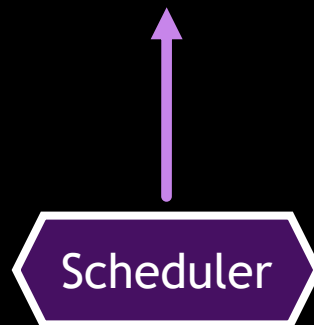
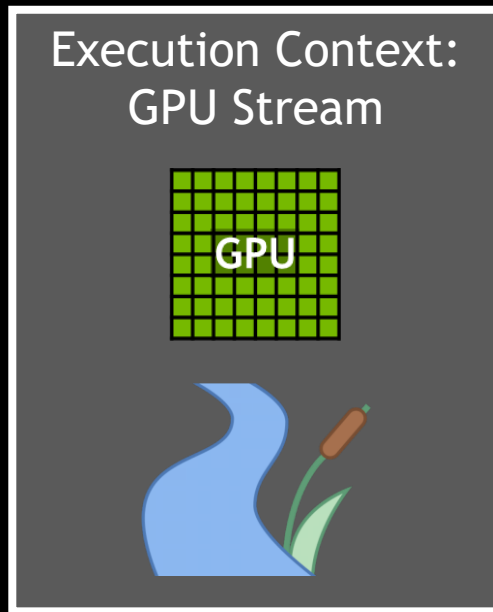
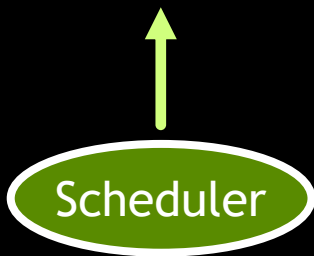
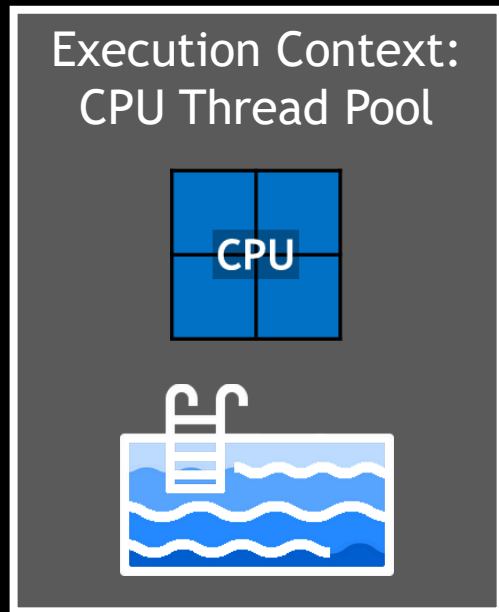


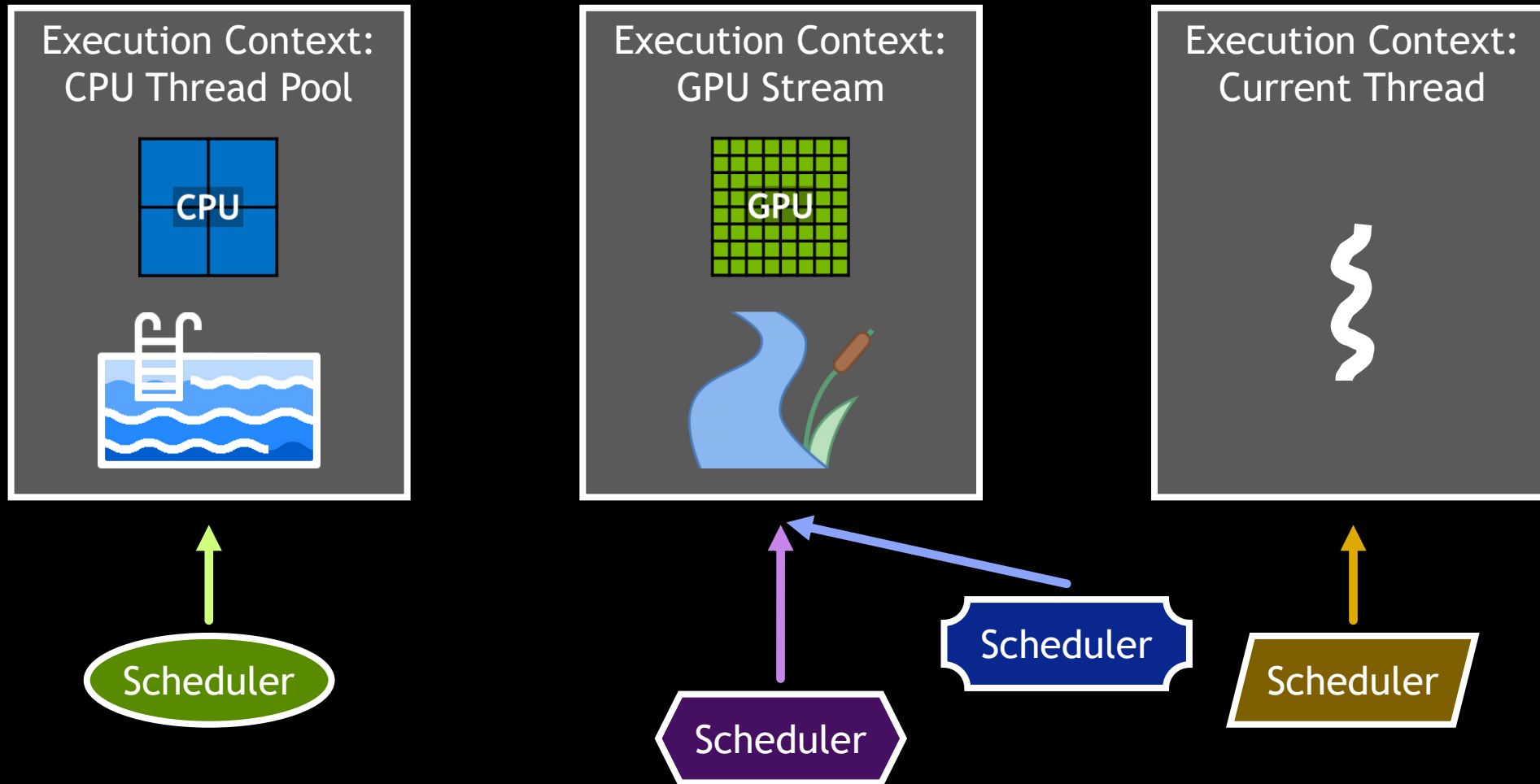
Execution Context:
GPU Stream

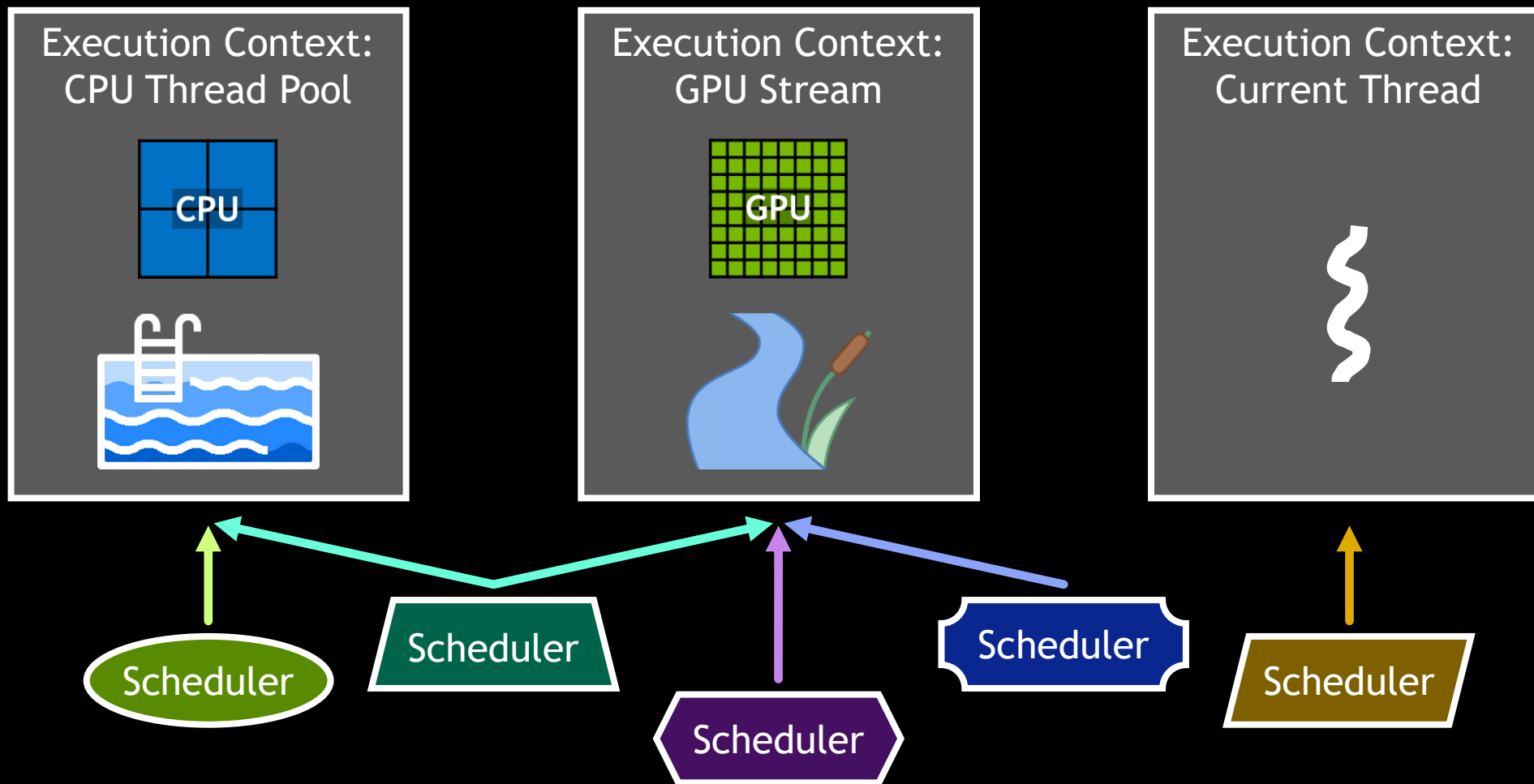


Execution Context:
Current Thread





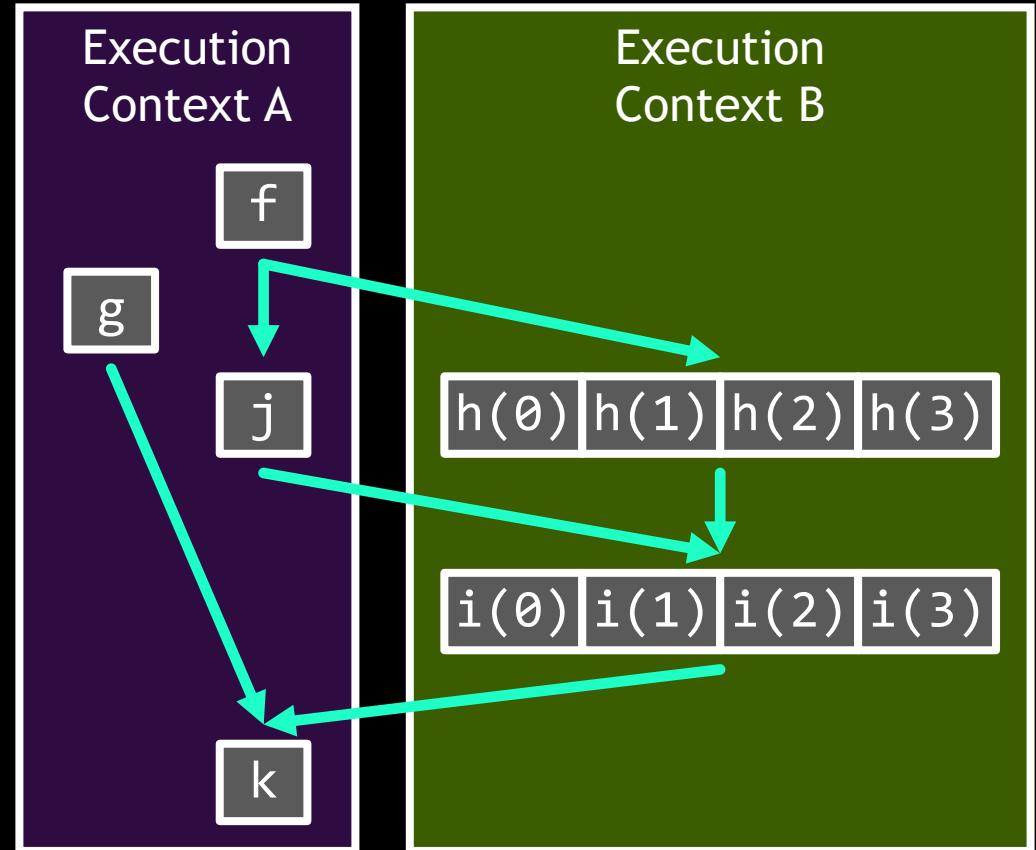




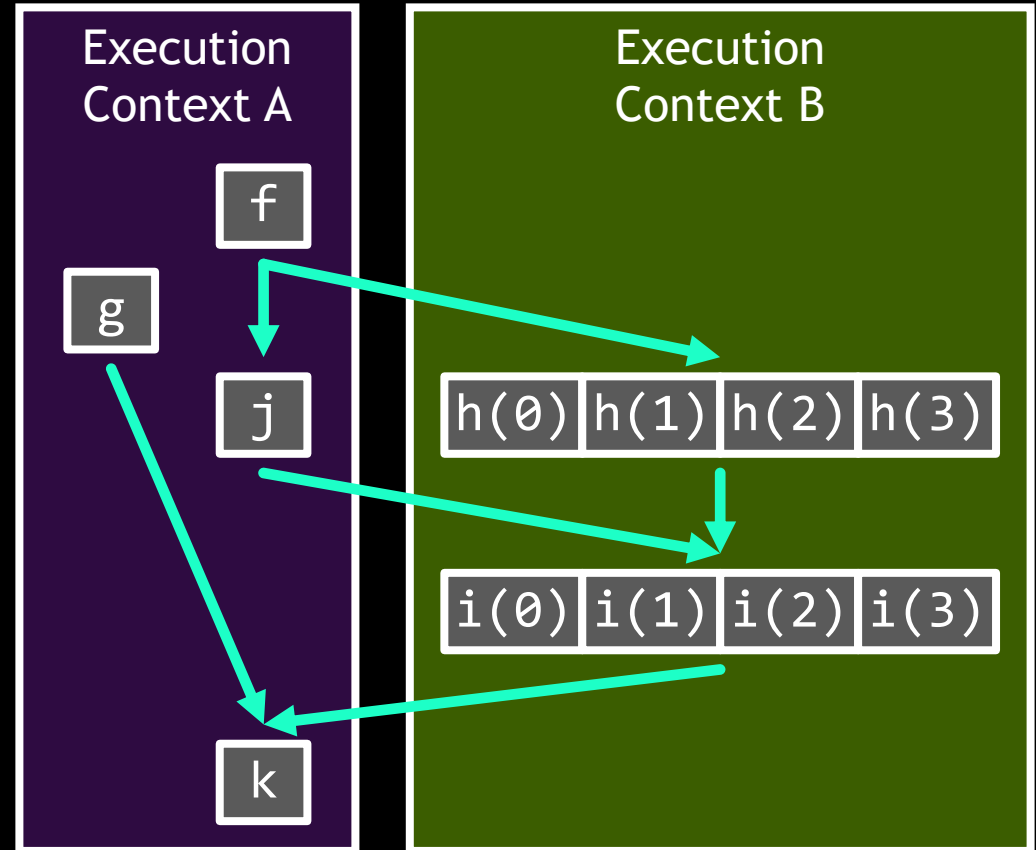
Schedulers produce senders.

- Senders represent asynchronous work.

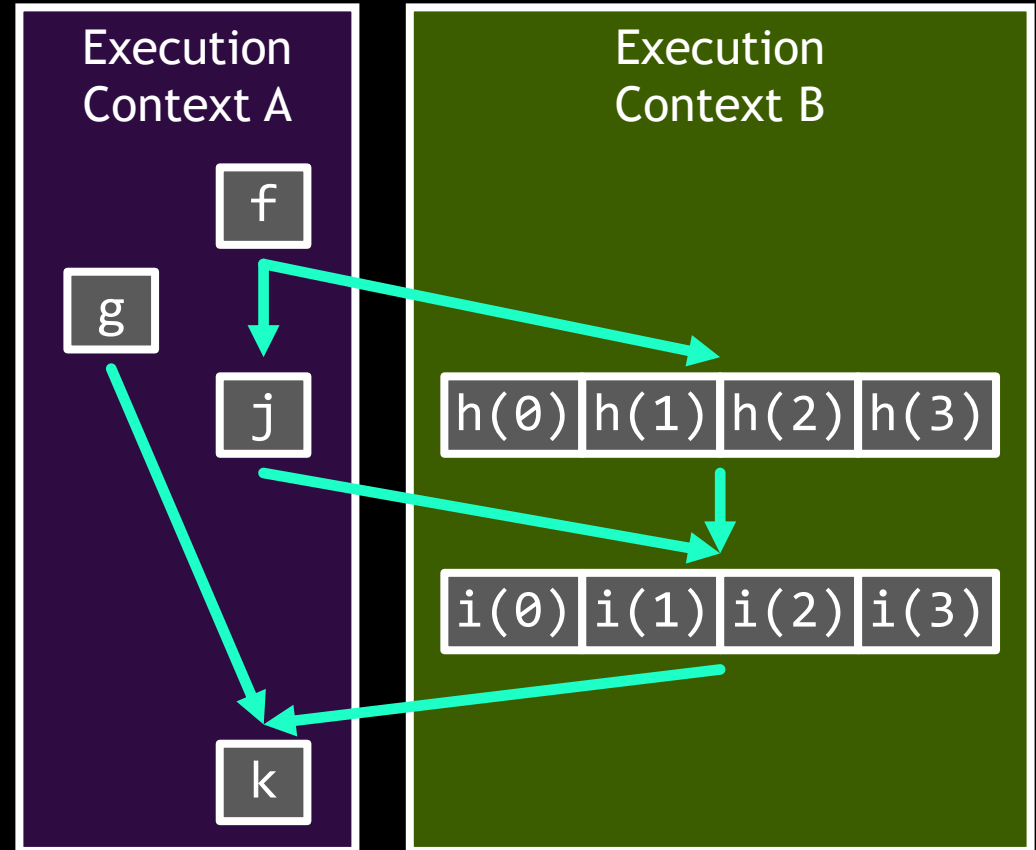
- Senders represent asynchronous work.
- Senders form the nodes of a task graph.



- Senders represent asynchronous work.
- Senders form the nodes of a task graph.
- Senders are lazy.

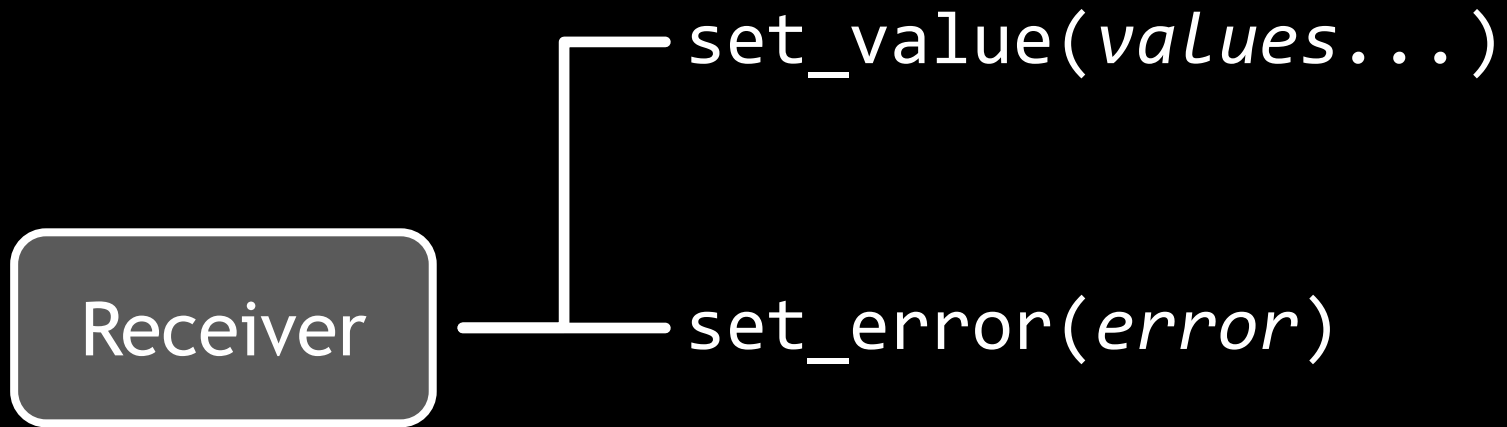


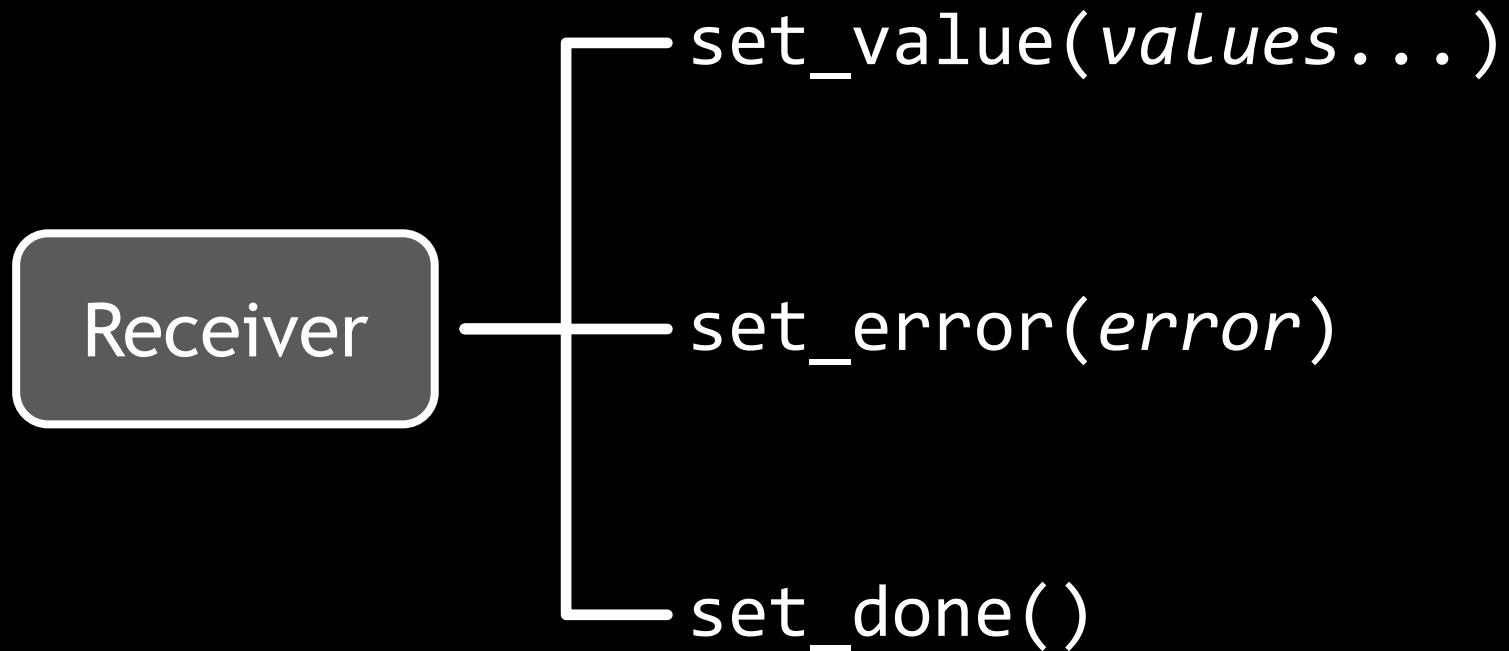
- Senders represent asynchronous work.
- Senders form the nodes of a task graph.
- Senders are lazy.
- When a sender's work completes, it sends a signal to the receivers attached to it.

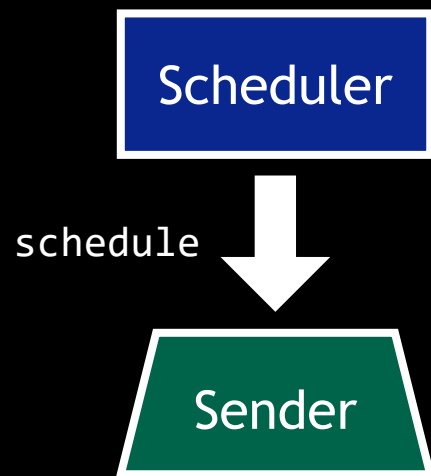


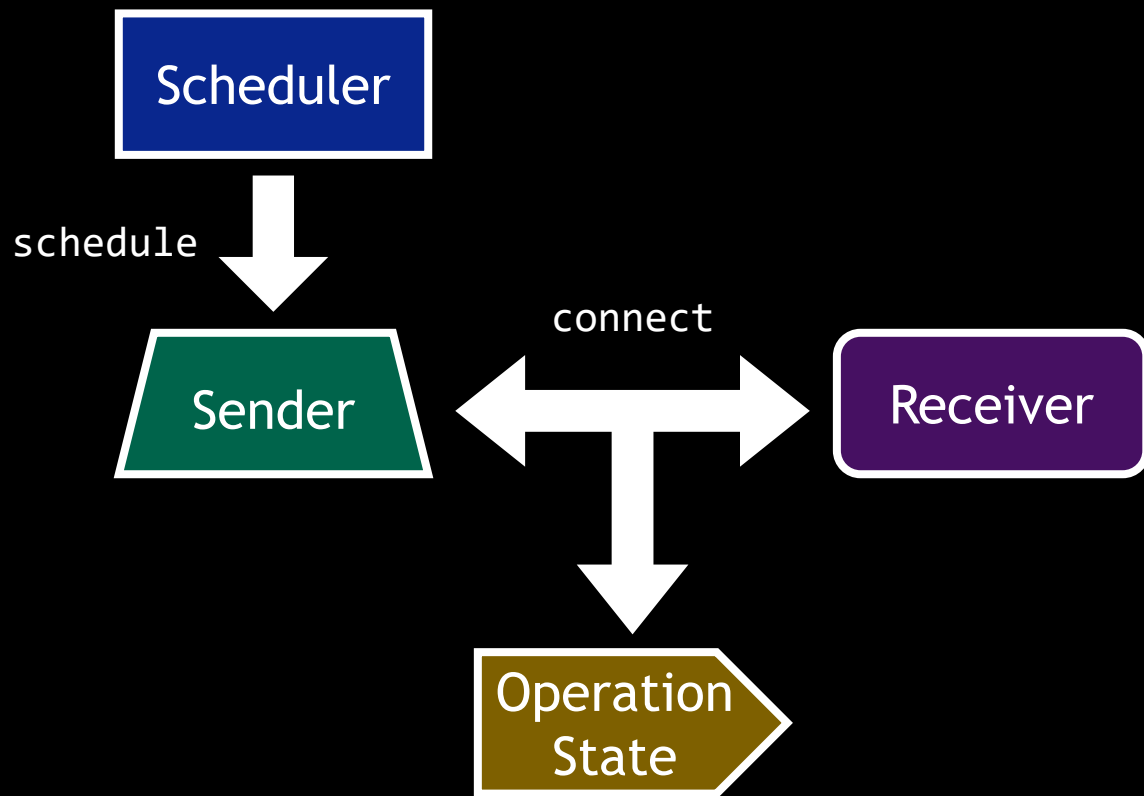
Receiver

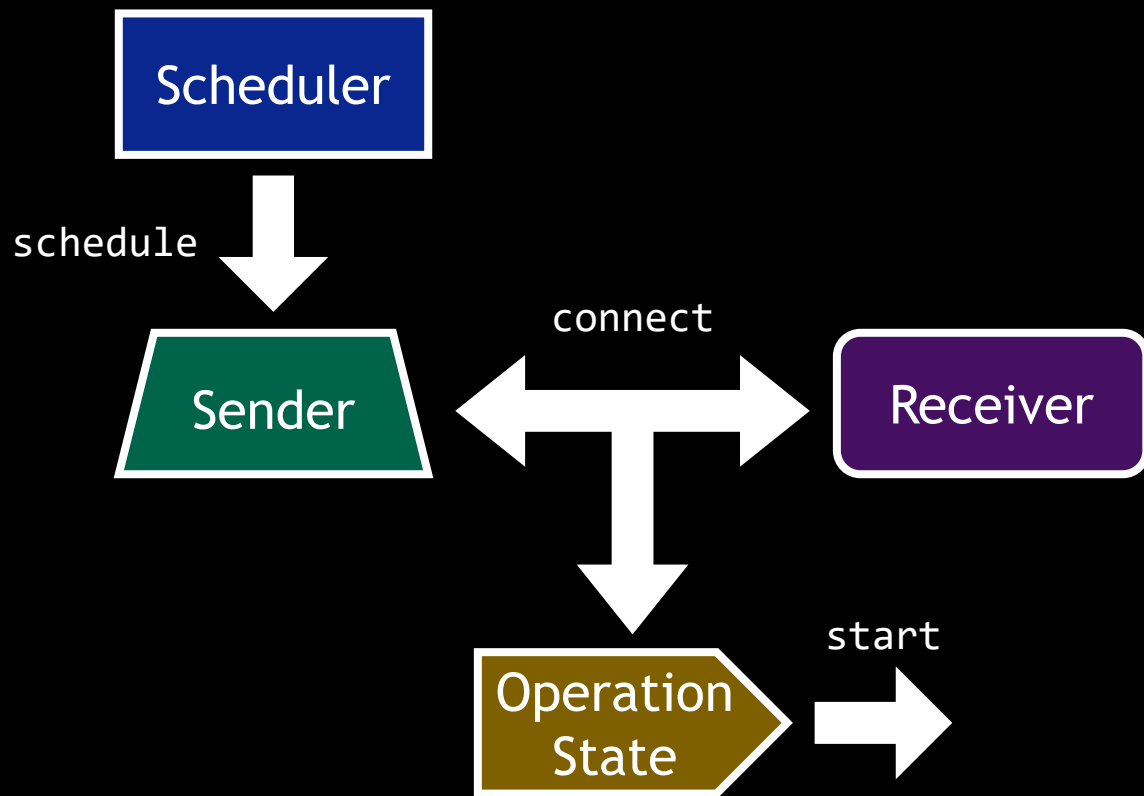


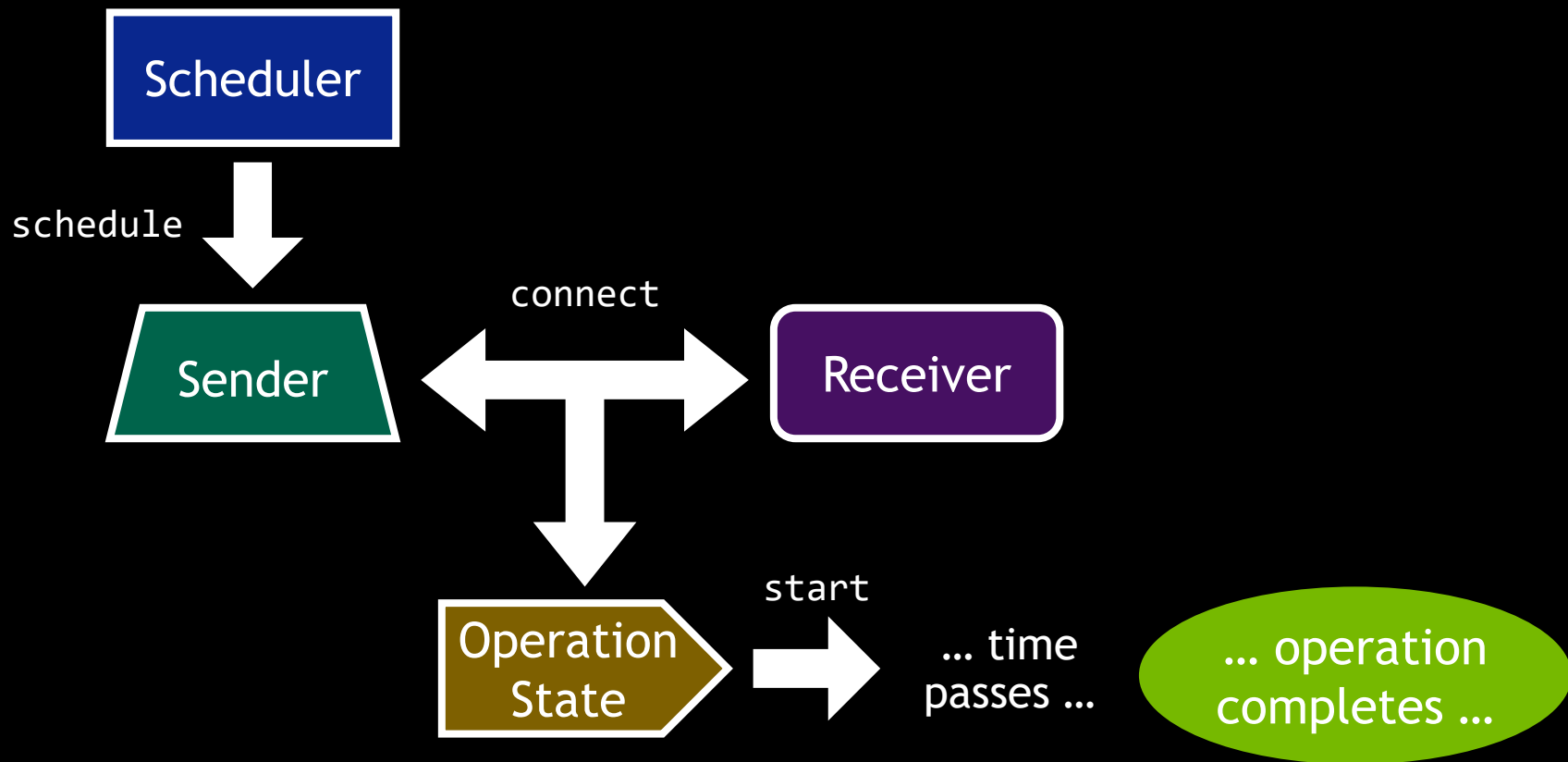


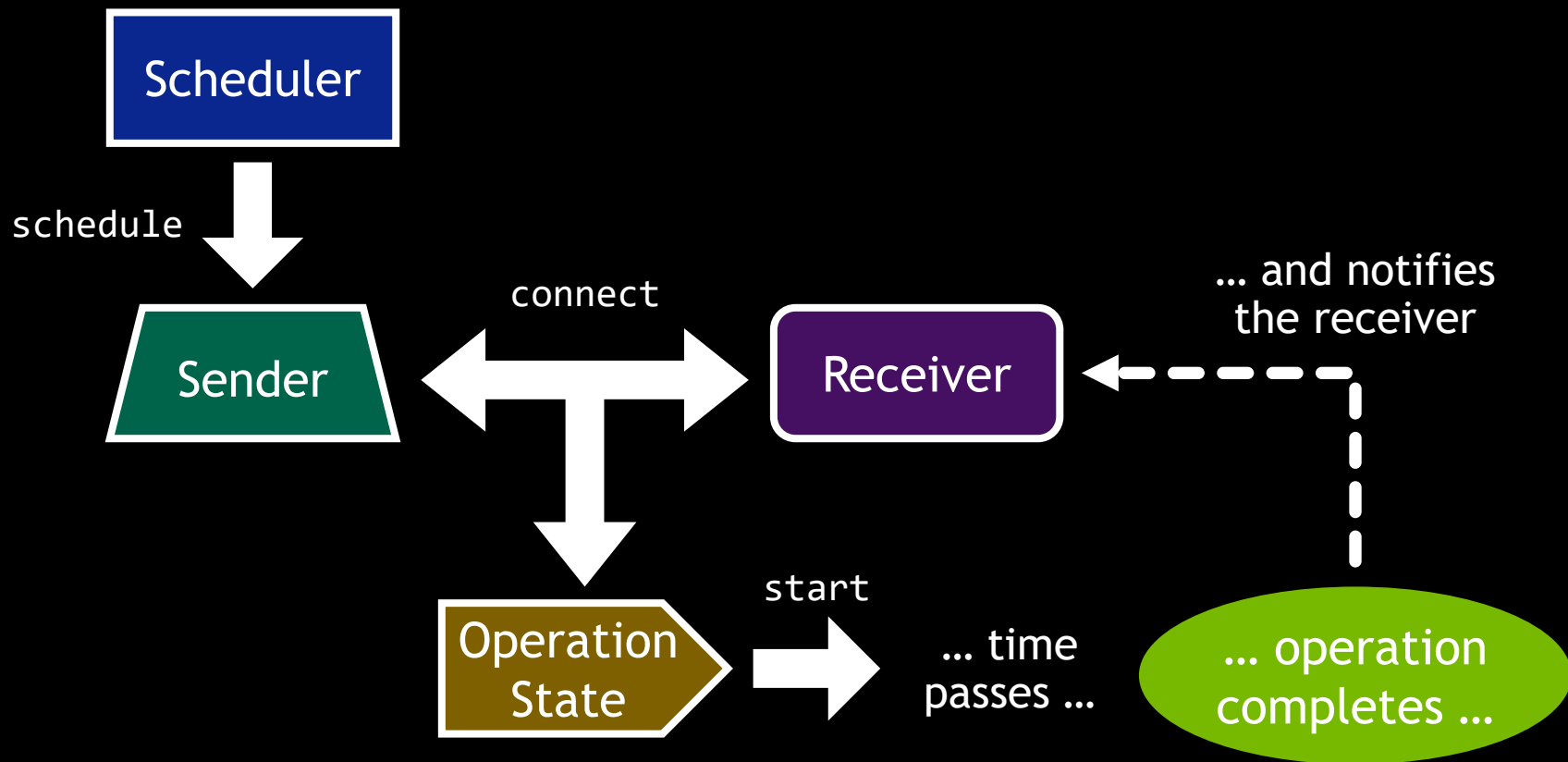












```
sender auto f(sender auto p, ...);
```

```
sender auto f(sender auto p, ...);
```

➤ Takes one or more senders.

```
sender auto f(sender auto p, ...);
```

- Takes one or more senders.
- Return a sender.

```
sender auto f(sender auto p, ...);
```

- Takes one or more senders.
- Return a sender.
- Pipeable (think *nix shells):

`snd | f | g`

is equivalent to

`g(f(snd))`

```

std::vector<std::string_view> v{...};

ex::sender auto s = for_each_async(
    ex::transfer(
        unique_async(
            sort_async(
                ex::transfer_just(gpu_stream_scheduler{}, v)
            )
        ),
        thread_pool.scheduler()
    ),
    [] (std::string_view e)
    { std::print(file, "{}\n", e); }
);

this_thread::sync_wait(s);

```



```
std::vector<std::string_view> v{...};

ex::sender auto s0 = ex::transfer_just(gpu_stream_scheduler{}, v);
ex::sender auto s1 = sort_async(s0);
ex::sender auto s2 = unique_async(s1);
ex::sender auto s3 = ex::transfer(s2, thread_pool.scheduler());
ex::sender auto s4 = for_each_async(s3, [] (std::string_view e)
                                   { std::print(file, "{}\n", e); });

this_thread::sync_wait(s);
```

```
std::vector<std::string_view> v{...};

ex::sender auto s = ex::transfer_just(gpu_stream_scheduler{}, v)
    | sort_async
    | unique_async
    | ex::transfer(thread_pool.scheduler())
    | for_each_async([] (std::string_view e)
        { std::print(file, "{}\n", e); });

this_thread::sync_wait(s);
```


Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected to multiple receivers.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.
<u>ensure started</u> (sender auto last)	Connects and starts last.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.
<u>ensure started</u> (sender auto last)	Connects and starts last.

Sender Factories	Semantics Of Returned Sender
------------------	------------------------------

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.
<u>ensure started</u> (sender auto last)	Connects and starts last.
Sender Factories	Semantics Of Returned Sender
<u>schedule</u> (scheduler auto sch)	Completes on sch.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.
<u>ensure started</u> (sender auto last)	Connects and starts last.

Sender Factories	Semantics Of Returned Sender
<u>schedule</u> (scheduler auto sch)	Completes on sch.
<u>just</u> (T&&... ts)	Send the values ts.

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.
<u>ensure started</u> (sender auto last)	Connects and starts last.

Sender Factories	Semantics Of Returned Sender
<u>schedule</u> (scheduler auto sch)	Completes on sch.
<u>just</u> (T&&... ts)	Send the values ts.

Sender Consumers	Semantics
------------------	-----------

Sender Adaptor	Semantics Of Returned Sender
<u>then</u> (sender auto last, invocable auto f)	Call f with the value sent by last.
<u>bulk</u> (sender auto last, shape auto n, invocable auto body)	Call body for every index in n with the value sent by last.
<u>transfer</u> (sender auto last, scheduler auto sch)	Transition to sch for the next sender.
<u>split</u> (sender auto last)	Can be connected multiple times.
<u>when all</u> (sender auto... inputs)	Combines multiple senders into an aggregate.
<u>ensure started</u> (sender auto last)	Connects and starts last.

Sender Factories	Semantics Of Returned Sender
<u>schedule</u> (scheduler auto sch)	Completes on sch.
<u>just</u> (T&&... ts)	Send the values ts.

Sender Consumers	Semantics
<u>sync wait</u> (sender auto snd) -> <i>values-sent-by-sender</i>	Block until snd completes and return or throw whatever it sent.

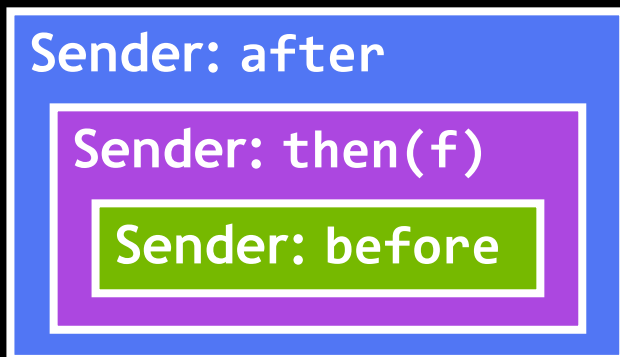
before | **then(f)** | **after;**

before | **then(f)** | **after**;

```
sender auto before_snd = ...;  
sender auto then_f_snd = then_sender(before_snd, f);  
sender auto after_snd = after_sender(then_f_snd);
```

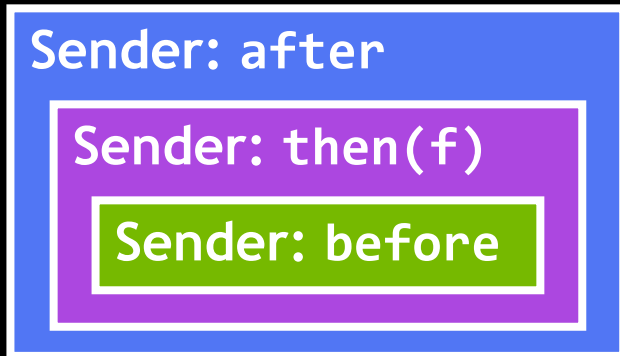

before | **then(f)** | **after**;

```
sender auto before_snd = ...;  
sender auto then_f_snd = then_sender(before_snd, f);  
sender auto after_snd = after_sender(then_f_snd);
```



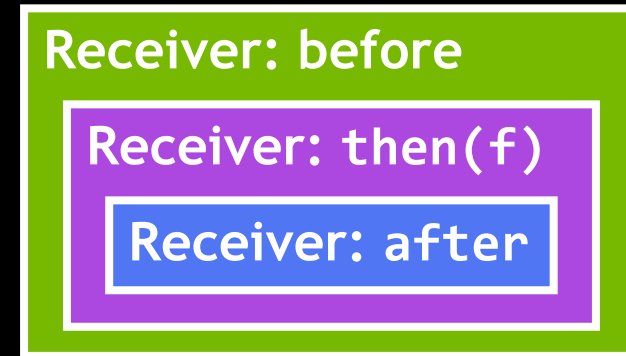
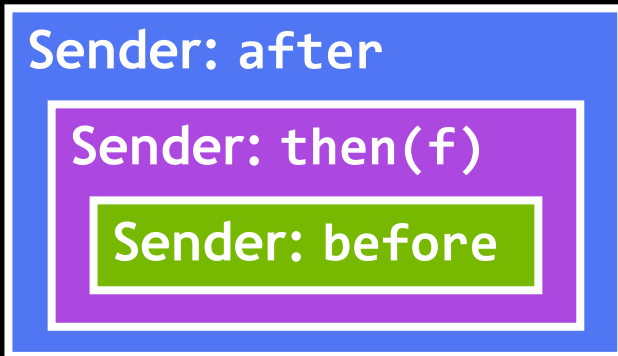
before | **then(f)** | **after**;

```
sender auto before_snd = ...;  
sender auto then_f_snd = then_sender(before_snd, f);  
sender auto after_snd = after_sender(then_f_snd);  
...  
    return connect(after_snd, ...);  
        return connect(then_f_snd, after_rcv);  
            return connect(before_snd, then_f_rcv);  
        ...
```



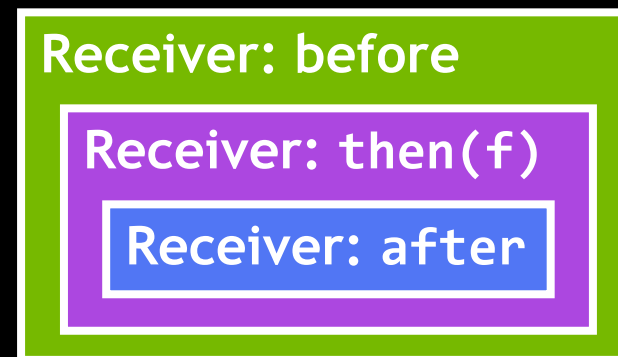
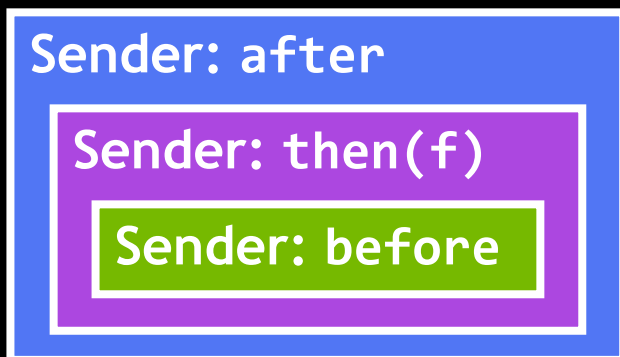
before | **then(f)** | **after**;

```
sender auto before_snd = ...;  
sender auto then_f_snd = then_sender(before_snd, f);  
sender auto after_snd = after_sender(then_f_snd);  
...  
    return connect(after_snd, ...);  
        return connect(then_f_snd, after_rcv);  
            return connect(before_snd, then_f_rcv);  
        ...
```



before | **then(f)** | **after**;

```
sender auto before_snd = ...;  
sender auto then_f_snd = then_sender(before_snd, f);  
sender auto after_snd = after_sender(then_f_snd);  
...  
return connect(after_snd, ...);  
return connect(then_f_snd, after_rcv);  
return connect(before_snd, then_f_rcv);  
...
```



```
...  
set_value(before_rcv, ...);  
set_value(then_f_rcv, before_val);  
set_value(after_rcv, f(before_val));  
...
```

```
inline constexpr sender_adaptor  
inclusive_scan_async = [] (...) -> ex::sender auto {  
    ...  
}
```

```
inline constexpr sender_adaptor  
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {  
    ...  
}
```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then([=] (std::random_access_range auto input) {
            ...
        })
    ...
}

```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then([=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        ...
}

```



```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then([=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        ...
}

```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                ...
            })
        ...
    }
}

```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                ...
            })
        ...
    }
}

```

a	b	c	d	e	f	g	h	i
---	---	---	---	---	---	---	---	---

a	ab	abc	d	de	def	g	gh	ghi
---	----	-----	---	----	-----	---	----	-----

a	b	c	d	e	f	g	h	i
---	---	---	---	---	---	---	---	---

a	ab	abc
---	----	-----

`std::inclusive_scan`

d	de	def
---	----	-----

`std::inclusive_scan`

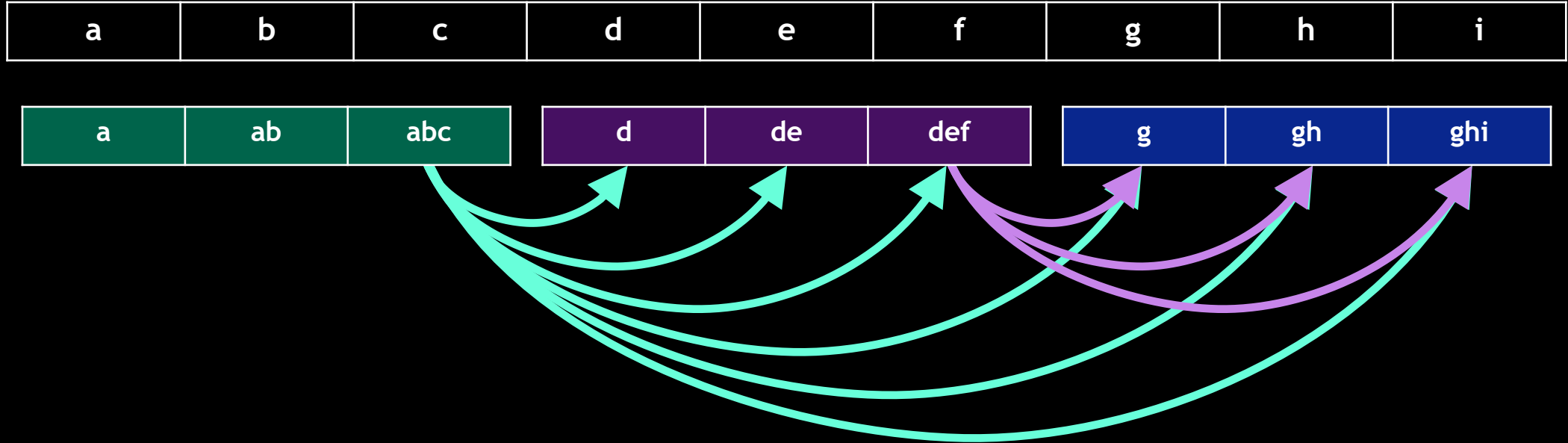
g	gh	ghi
---	----	-----

`std::inclusive_scan`

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                ...
                std::inclusive_scan(begin(input) + start,
                                    begin(input) + end,
                                    begin(input) + start);
            })
        ...
    }
}

```



```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                ...            = *--std::inclusive_scan(begin(input) + start,
                                begin(input) + end,
                                begin(input) + start);
            })
        ...
    }
}

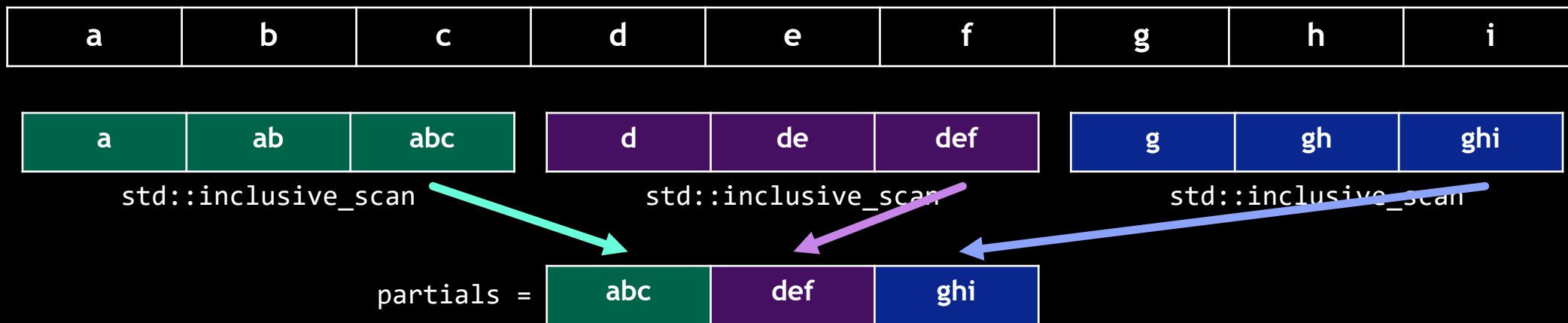
```



```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then([=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                            begin(input) + end,
                                                            begin(input) + start);
            })
        ...
    }

```





`std::inclusive_scan`

`std::inclusive_scan`

`std::inclusive_scan`

`partials =`



`std::inclusive_scan`

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then([ ] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            ...
        })
        ...
    }
}

```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then([=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then([] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            return send_values(input, std::move(partial));
        })
        ...
    }
}

```



`std::inclusive_scan`

`std::inclusive_scan`

`std::inclusive_scan`

`partials =`



`std::inclusive_scan`

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then( [] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                ...
            })
        ...
    }
}

```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then( [] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                ...
            })
        ...
    }
}

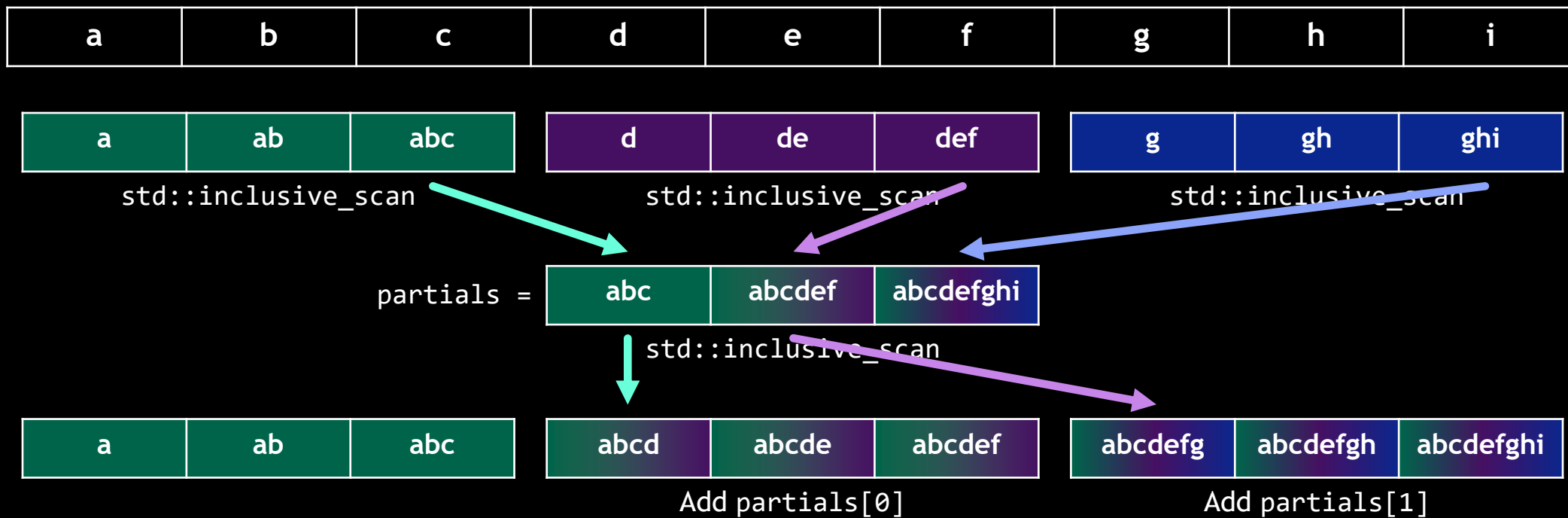
```



```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then([=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then([] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                std::for_each(begin(input) + start, begin(input) + end,
                    [&] (auto& e) { e = partials[i] + e; });
            })
        ...
    }
}

```



```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then( [=] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                std::for_each(begin(input) + start, begin(input) + end,
                              [&] (auto& e) { e = partials[i] + e; });
            })
        | ex::then( [=] (auto input, auto partials) { return input; });
}

```

```

inline constexpr sender_adaptor
inclusive_scan_async = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
    return last
        | ex::then( [=] (std::random_access_range auto input) {
            std::vector<std::range_value_t<decltype(input)>> partials(tile_count + 1);
            partials[0] = init;
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                         begin(input) + end,
                                                         begin(input) + start);
            })
        | ex::then( [] (auto input, auto partials) {
            std::inclusive_scan(begin(partial), end(partial), begin(partial));
            return send_values(input, std::move(partial));
        })
        | ex::bulk(tile_count,
            [=] (std::size_t i, auto input, auto partials) {
                auto tile_size = (input.size() + tile_count - 1) / tile_count;
                auto start     = i * tile_size;
                auto end       = std::min(input.size(), (i + 1) * tile_size);
                std::for_each(begin(input) + start, begin(input) + end,
                             [&] (auto& e) { e = partials[i] + e; });
            })
        | ex::then( [=] (auto input, auto partials) { return input; });
}

```

```
stdr::for_each(rng, f);  
stdr::for_each(rng, g);  
stdr::for_each(rng, h);
```

```
ex::sender_of<std::range> auto  
transform_async(ex::sender_of<std::range> auto rng, std::callable auto c);
```

```
std::for_each(rng, f);  
std::for_each(rng, g);  
std::for_each(rng, h);
```

```
ex::sender snd =  
    ex::transfer_just(sch, rng)  
    | transform(f)  
    | transform(g)  
    | transform(h);
```

```
auto max = *stdr::max_element(rng);  
  
stdr::for_each(rng,  
    [=] (auto x) { return x / max; });
```

```
ex::sender_of<std::range, std::range, ...> auto  
transform_async(ex::sender_of<std::range, std::range, ...> auto rngs,  
                std::callable auto f);
```

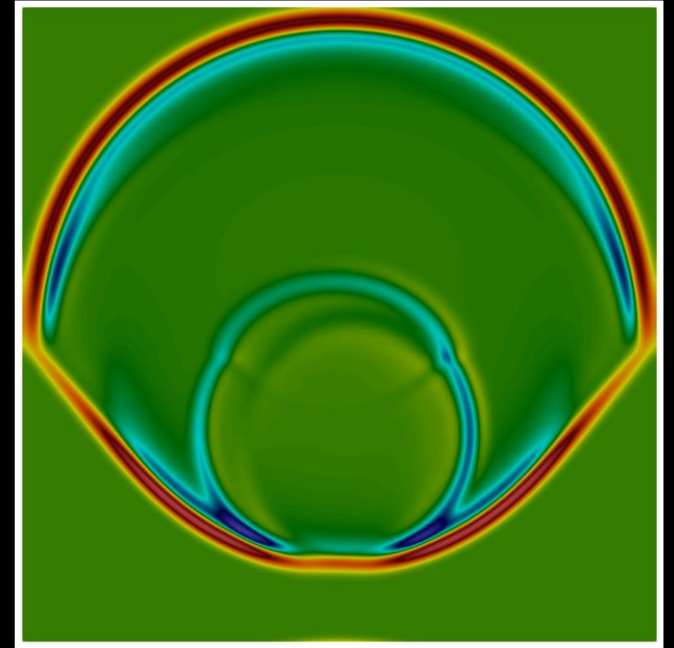
```
ex::sender_of<std::range, std::forward_iterator> auto  
max_element_async(ex::sender_of<std::range> auto rng);
```

```
auto max = *std::max_element(rng);  
  
std::for_each(rng,  
              [=] (auto x) { return x / max; });
```

```
ex::sender auto snd =  
    ex::transfer_just(sch, rng)  
    | max_element_async  
    | ex::let_value(  
        [] (auto r, auto max) {  
            return ex::just(r, max);  
        })  
    | transform_async(std::divides);
```

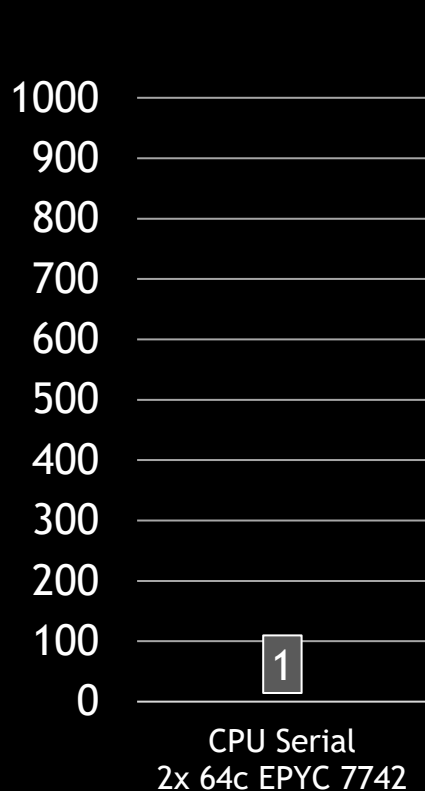

Maxwell's Equations

```
sender auto maxwell_eqs(scheduler auto &compute,  
                        grid_accessor A, ...) {  
    return repeat_n(n_outer_iterations,  
        repeat_n(n_inner_iterations,  
            schedule(compute)  
            | bulk(G.cells, update_h(G))  
            | halo_exchange(G, hx, hy)  
            | bulk(G.cells, update_e(time, dt, G))  
            | halo_exchange(G, hx, hy))  
            | transfer(cpu_serial_scheduler)  
            | then(output_results))  
        );  
}
```



Maxwell's Equations

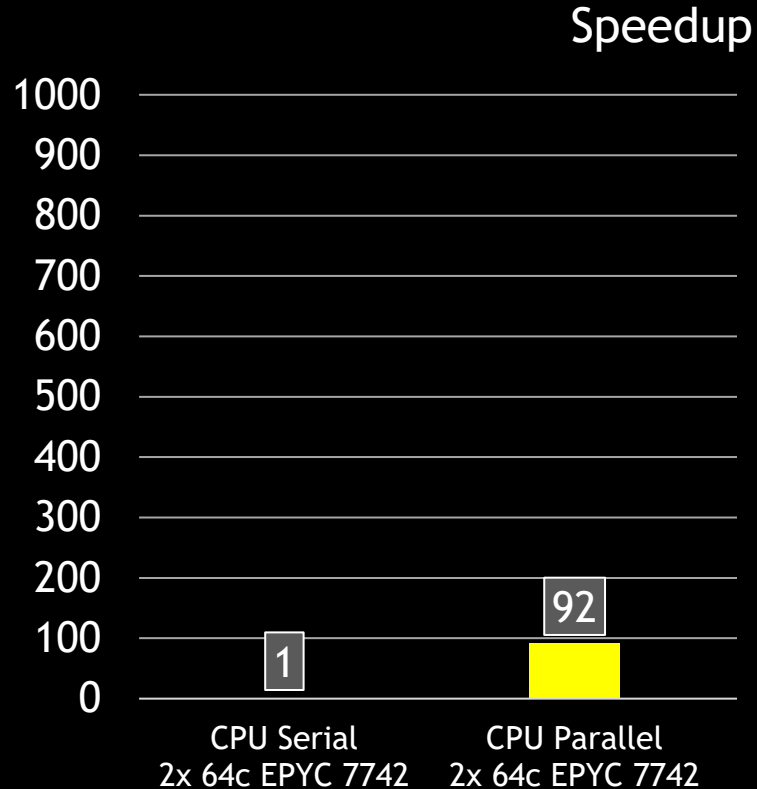
Change one line of code and scale from a single CPU thread...



```
sync_wait(maxwell_eqs(cpu_serial_scheduler), ...);
```

Maxwell's Equations

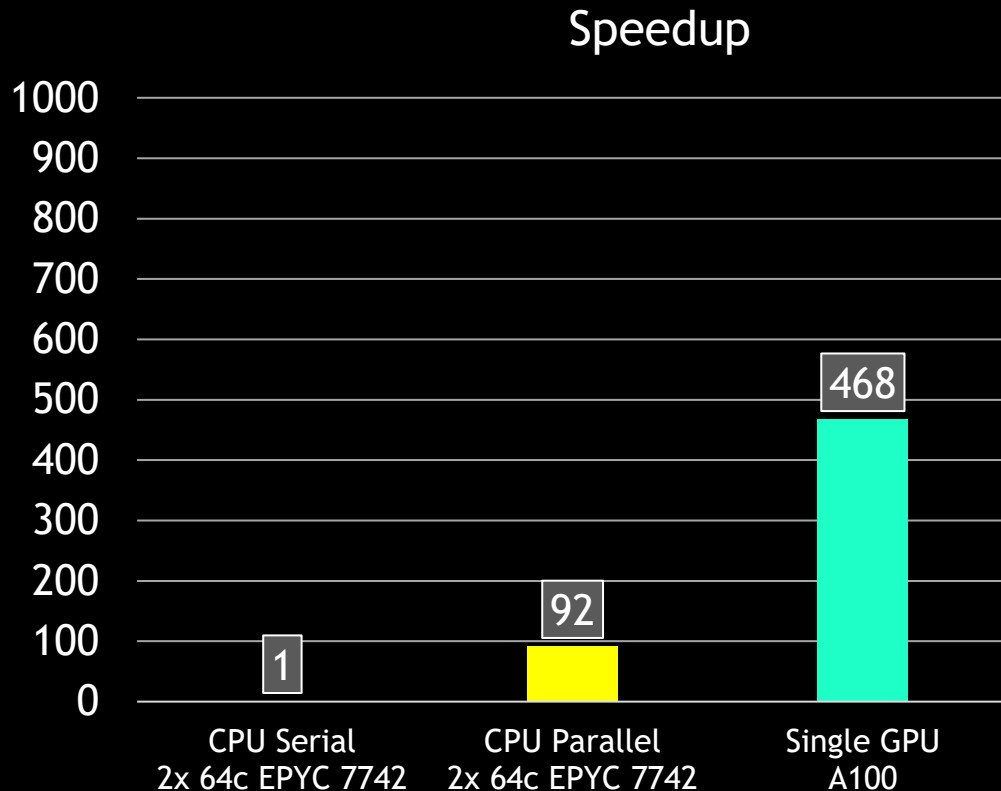
Change one line of code and scale from a single CPU thread up to multiple CPU threads...



```
sync_wait(maxwell_eqs(cpu_parallel_scheduler), ...);
```

Maxwell's Equations

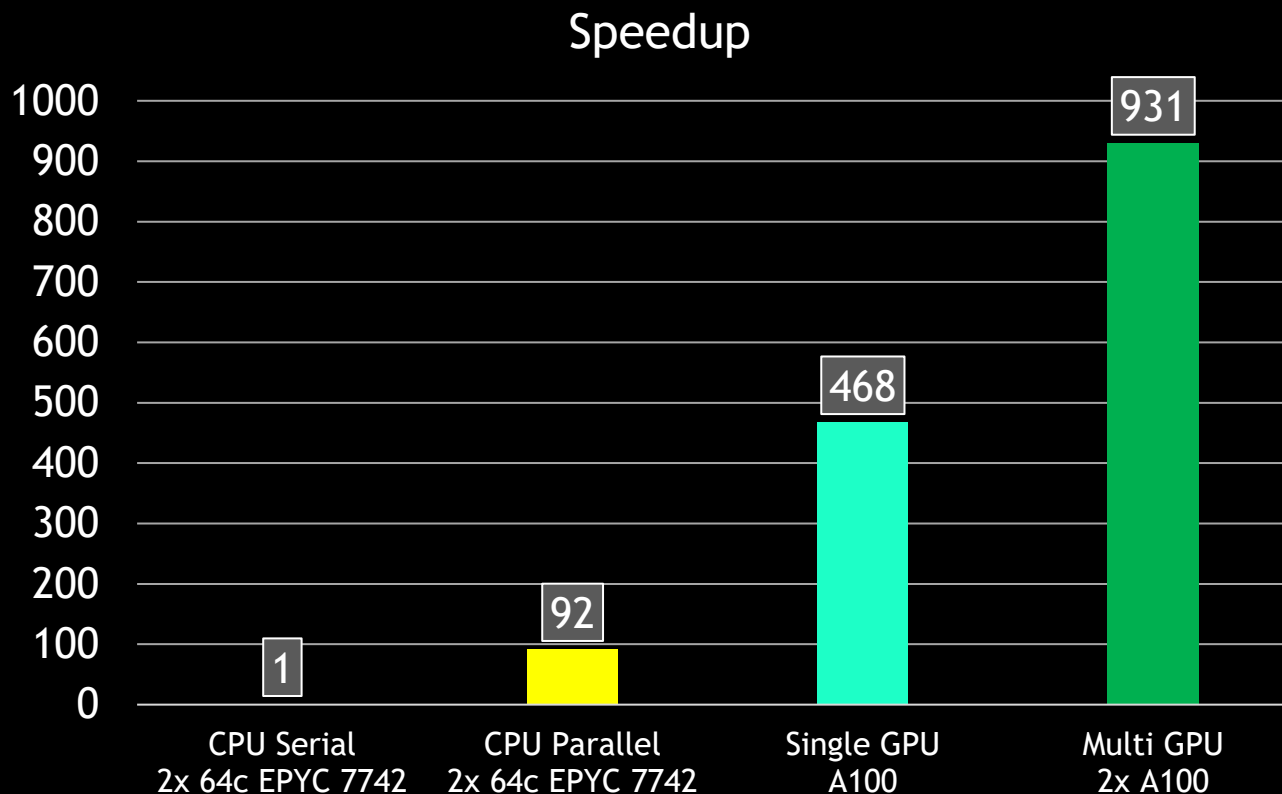
Change one line of code and scale from a single CPU thread up to a GPU...



```
sync_wait(maxwell_eqs(single_gpu_scheduler), ...);
```

Maxwell's Equations

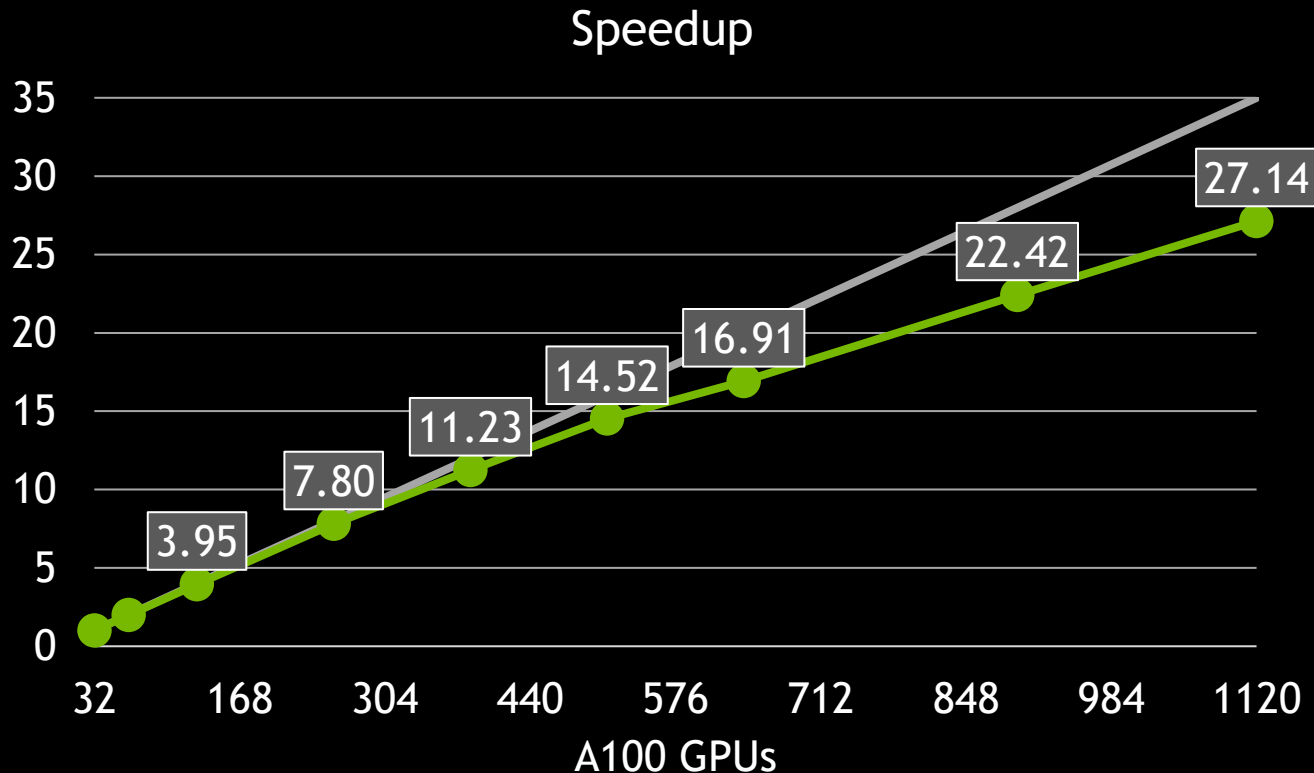
Change one line of code and scale from a single CPU thread up to multiple GPUs...



```
sync_wait(maxwell_eqs(multi_gpu_scheduler), ...);
```

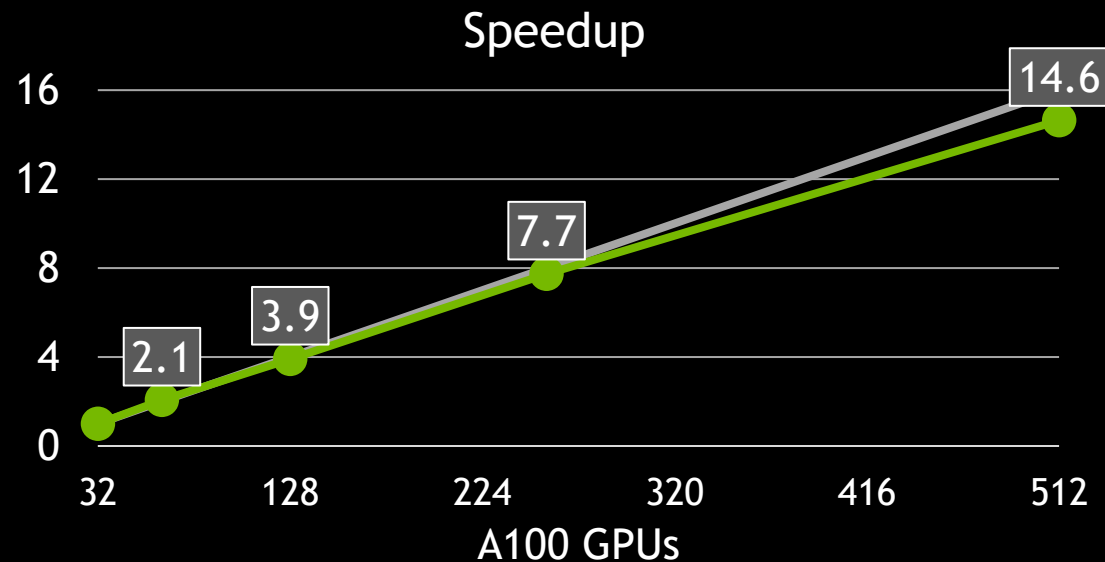
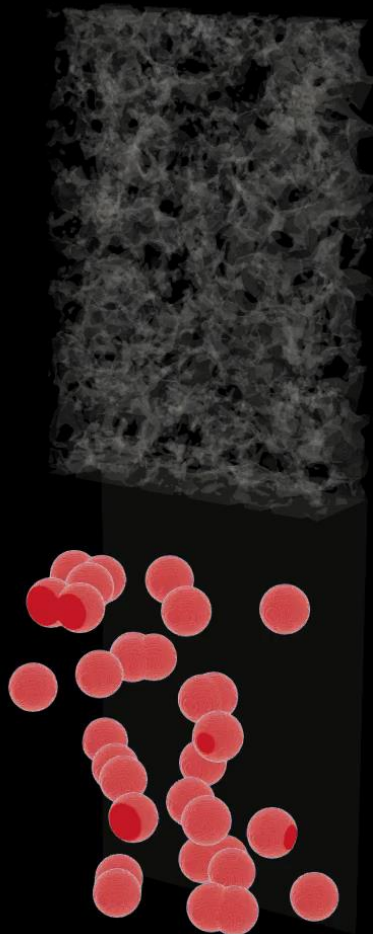
Maxwell's Equations

Change one line of code and scale from a single CPU thread up to a cluster of GPUs!



```
sync_wait(maxwell_eqs(multi_node_gpu_scheduler), ...);
```

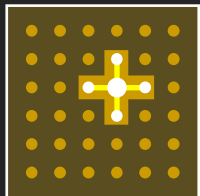
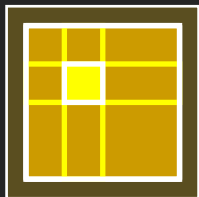
Palabos Carbon Sequestration



- Palabos is a framework for parallel computational fluid dynamics simulations using the Lattice-Boltzmann method.
- Code for multi-component flow through a porous media ported to C++ Senders and Receivers.
- Application: simulating carbon sequestration in sandstone.

Pillars of Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



Today

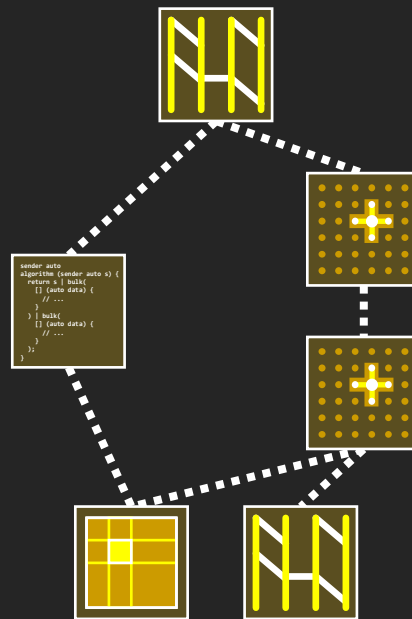
Tools to Write Your Own Parallel Algorithms that Run Anywhere



```
sender auto  
algorithm (sender auto s) {  
    return s | bulk(N,  
        [] (auto data) {  
            // ...  
        }) | bulk(N,  
        [] (auto data) {  
            // ...  
        });  
}
```



Mechanisms for Composing Parallel Invocations into Task Graphs



With Senders & Receivers



Standard Algorithms

Serial (C++98)

```
std::vector<T> x{...};

std::for_each(
    begin(x), end(x),
    f);

std::for_each(
    begin(x), end(x),
    g);

std::for_each(
    begin(x), end(x),
    h);
```

Parallel (C++17)

```
std::vector<T> x{...};

std::for_each(
    ex::par_unseq,
    begin(x), end(x),
    f);

std::for_each(
    ex::par_unseq,
    begin(x), end(x),
    g);

std::for_each(
    ex::par_unseq,
    begin(x), end(x),
    h);
```

Asynchronous

```
std::vector<T> x(...);

ex::sender auto s
    = ex::transfer_just(sch, x)
    | for_each_async(f)
    | for_each_async(g)
    | for_each_async(h);

this thread::sync wait(s);
```

Today, C++ has no reasonable abstraction for multi-dimensional data.

Today, C++ has no reasonable abstraction for
multi-dimensional data.

The solution is coming in C++23:

`std::mdspan`

`std::mdspan`

- Non-owning; pointer + metadata.

`std::mdspan`

- Non-owning; pointer + metadata.
- Metadata can be dynamic or static.

`std::mdspan`

- Non-owning; pointer + metadata.
- Metadata can be dynamic or static.
- Parameterizes layout.

`std::mdspan`

- Non-owning; pointer + metadata.
- Metadata can be dynamic or static.
- Parameterizes layout and access.

```
template <std::size_t... Extents>  
class std::extents;
```



```
template <std::size_t... Extents>  
class std::extents;
```

```
std::extents e0{16, 32};
```

```
// Equivalent to:
```

```
std::extents<std::dynamic_extent, std::dynamic_extent> e1{16, 32};
```

```
e0.rank() == 2
```

```
e0.extent(0) == 16
```

```
e0.extent(1) == 32
```

```
template <std::size_t... Extents>  
class std::extents;
```

```
std::extents e0{16, 32};
```

```
// Equivalent to:
```

```
std::extents<std::dynamic_extent, std::dynamic_extent> e1{16, 32};  
std::dextents<2> e2{16, 32};
```

```
e0.rank() == 2
```

```
e0.extent(0) == 16
```

```
e0.extent(1) == 32
```

```
template <std::size_t... Extents>  
class std::extents;
```

```
std::extents e0{16, 32};
```

```
// Equivalent to:
```

```
std::extents<std::dynamic_extent, std::dynamic_extent> e1{16, 32};  
std::dextents<2> e2{16, 32};
```

```
e0.rank()      == 2  
e0.extent(0)   == 16  
e0.extent(1)   == 32
```

```
std::extents<16, 32> e3;
```

```
template <std::size_t... Extents>  
class std::extents;
```

```
std::extents e0{16, 32};
```

```
// Equivalent to:
```

```
std::extents<std::dynamic_extent, std::dynamic_extent> e1{16, 32};  
std::dextents<2> e2{16, 32};
```

```
e0.rank()      == 2  
e0.extent(0)   == 16  
e0.extent(1)   == 32
```

```
std::extents<16, 32> e3;
```

```
std::extents<16, std::dynamic_extent> e4{32};
```

```
template <std::size_t... Extents>  
class std::extents;
```

```
std::extents e0{16, 32};
```

```
// Equivalent to:
```

```
std::extents<std::dynamic_extent, std::dynamic_extent> e1{16, 32};  
std::dextents<2> e2{16, 32};
```

```
e0.rank()      == 2  
e0.extent(0)   == 16  
e0.extent(1)   == 32
```

```
std::extents<16, 32> e3;
```

```
std::extents<16, std::dynamic_extent> e4{32};
```

```
std::extents e5{16, 32, 48, 4};
```

```
template <
```

```
>
```

```
class std::mdspan;
```

```
template <class I,
```

```
>
```

```
class std::mdspan;
```

```
template <class I,  
         class Extents,
```

>

```
class std::mdspan;
```



```
template <class I,  
         class Extents,  
         class LayoutPolicy = std::layout right,
```

>

```
class std::mdspan;
```

```
template <class I,  
          class Extents,  
          class LayoutPolicy = std::layout right,  
          class AccessorPolicy = std::default accessor<T>>  
class std::mdspan;
```

```
template <class I,  
         class Extents,  
         class LayoutPolicy = std::layout right,  
         class AccessorPolicy = std::default accessor<T>>  
class std::mdspan;  
  
std::mdspan m0{data, 16, 32};  
// Equivalent to:  
std::mdspan<double, std::dextents<2>> m1{data, 16, 32};
```

```

template <class I,
          class Extents,
          class LayoutPolicy = std::layout_right,
          class AccessorPolicy = std::default_accessor<T>>
class std::mdspan;

std::mdspan m0{data, 16, 32};
// Equivalent to:
std::mdspan<double, std::dextents<2>> m1{data, 16, 32};

m0[i, j] == data[i * M + j]

```

```

template <class I,
          class Extents,
          class LayoutPolicy = std::layout right,
          class AccessorPolicy = std::default accessor<T>>
class std::mdspan;

std::mdspan m0{data, 16, 32};
// Equivalent to:
std::mdspan<double, std::dextents<2>> m1{data, 16, 32};

m0[i, j] == data[i * M + j]

std::mdspan m2{data, std::extents<16, 32>{}};
// Equivalent to:
std::mdspan<double, std::extents<16, 32>> m3{data};

std::mdspan m4{data, std::extents<16, std::dynamic_extent>{32}};

```

Row-Major AKA Right

- C++, NumPy (default)
- Rightmost extent is contiguous

```
mdspan A{data, N, M};  
mdspan A{data, layout_right::mapping{N, M}};
```

```
A[i, j]      == data[i * M + j]  
A.stride(0) == M  
A.stride(1) == 1
```

Row-Major AKA Right

- C++, NumPy (default)
- Rightmost extent is contiguous

```
mdspan A{data, N, M};  
mdspan A{data, layout_right::mapping{N, M}};
```

```
A[i, j] == data[i * M + j]  
A.stride(0) == M  
A.stride(1) == 1
```

Location	Element
0	a_{11}
1	a_{12}
2	a_{21}
3	a_{22}

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Row-Major AKA Right

- C++, NumPy (default)
- Rightmost extent is contiguous

Column-Major AKA Left

- Fortran, MATLAB
- Leftmost extent is contiguous

```
mdspan A{data, N, M};  
mdspan A{data, layout right::mapping{N, M}};
```

```
A[i, j] == data[i * M + j]  
A.stride(0) == M  
A.stride(1) == 1
```

```
mdspan B{data, layout left::mapping{N, M}};
```

```
B[i, j] == data[i + j * N]  
B.stride(0) == 1  
B.stride(1) == N
```

Location	Element
0	a_{11}
1	a_{12}
2	a_{21}
3	a_{22}

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Row-Major AKA Right

- C++, NumPy (default)
- Rightmost extent is contiguous

```
mdspan A{data, N, M};  
mdspan A{data, layout right::mapping{N, M}};
```

```
A[i, j] == data[i * M + j]  
A.stride(0) == M  
A.stride(1) == 1
```

Location	Element
0	a_{11}
1	a_{12}
2	a_{21}
3	a_{22}

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Column-Major AKA Left

- Fortran, MATLAB
- Leftmost extent is contiguous

```
mdspan B{data, layout left::mapping{N, M}};
```

```
B[i, j] == data[i + j * N]  
B.stride(0) == 1  
B.stride(1) == N
```

Location	Element
0	a_{11}
1	a_{21}
2	a_{12}
3	a_{22}

Row-Major AKA Right

- C++, NumPy (default)
- Rightmost extent is contiguous

Column-Major AKA Left

- Fortran, MATLAB
- Leftmost extent is contiguous

```
mdspan A{data, N, M};  
mdspan A{data, layout right::mapping{N, M}};
```

```
A[i, j]      == data[i * M + j]  
A.stride(0) == M  
A.stride(1) == 1
```

```
mdspan B{data, layout left::mapping{N, M}};
```

```
B[i, j]      == data[i + j * N]  
B.stride(0) == 1  
B.stride(1) == N
```

User-Defined Strides

```
mdspan C{data, layout stride::mapping{extents{N, M}, {X, Y}};
```

```
A[i, j]      == data[i * X + j * Y]  
A.stride(0) == X  
A.stride(1) == Y
```

Layouts map (i, j, k, \dots) to a data location.

Layouts map (i, j, k, \dots) to a data location.

Anyone can define a layout.

Layouts map (i, j, k, \dots) to a data location.

Anyone can define a layout.

Layouts may:

➤ Be non-contiguous.

Layouts map (i, j, k, \dots) to a data location.

Anyone can define a layout.

Layouts may:

- Be non-contiguous.
- Map multiple indices to the same location.

Layouts map (i, j, k, \dots) to a data location.

Anyone can define a layout.

Layouts may:

- Be non-contiguous.
- Map multiple indices to the same location.
- Perform complicated computations.

Layouts map (i, j, k, \dots) to a data location.

Anyone can define a layout.

Layouts may:

- Be non-contiguous.
- Map multiple indices to the same location.
- Perform complicated computations.
- Have or refer to state.

**Parametric layout enables
generic multi-dimensional algorithms.**

```
void your_function(Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>& m);
```

```
void your_function(Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>& m);  
your_function(Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>{...});
```

```
void your_function(Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>& m);
```

```
your_function(Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>{...});  
your_function(boost::numeric::ublas::matrix<double>{...});  
your_function(Mat{...}); // PETSc  
your_function(blaze::DynamicMatrix<double, blaze::rowMajor>{...});  
your_function(cutlass::HostTensor<float, cutlass::layout::ColumnMajor>{...});  
// ...
```

```
void your_function(std::mdspan<T, Extents, Layout, Accessor> m);  
  
your_function(Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>{...});  
your_function(boost::numeric::ublas::matrix<double>{...});  
your_function(Mat{...}); // PETSc  
your_function(blaze::DynamicMatrix<double, blaze::rowMajor>{...});  
your_function(cutlass::HostTensor<float, cutlass::layout::ColumnMajor>{...});  
// ...
```

```

struct my_matrix {
public:
    my_matrix(std::size_t N, std::size_t M)
        : num_rows_(N), num_cols_(M), storage_(num_rows_ * num_cols_) {}

    double& operator()(size_t i, size_t j)
    { return storage_[i * num_cols_ + j]; }
    const double& operator()(size_t i, size_t j) const
    { return storage_[i * num_cols_ + j]; }

    std::size_t num_rows() const { return num_rows_; }
    std::size_t num_cols() const { return num_cols_; }

private:
    std::size_t          num_rows_, num_cols_;
    std::vector<double> storage_;
};

```

```

struct my_matrix {
public:
    my_matrix(std::size_t N, std::size_t M)
        : num_rows_(N), num_cols_(M), storage_(num_rows_ * num_cols_) {}

    double& operator()(size_t i, size_t j)
    { return storage_[i * num_cols_ + j]; }
    const double& operator()(size_t i, size_t j) const
    { return storage_[i * num_cols_ + j]; }

    std::size_t num_rows() const { return num_rows_; }
    std::size_t num_cols() const { return num_cols_; }

    operator std::mdspan<double, std::dextents<2>>() const
{ return {storage_, num_rows_, num_cols_}; }

private:
    std::size_t          num_rows_, num_cols_;
    std::vector<double> storage_;
};

```

```
std::mdspan A{input, N, M, 0};
```

```
std::mdspan B{output, N, M, 0};
```

```
auto v = stdv::cartesian_product(  
    stdv::iota(1, A.extent(0) - 1),  
    stdv::iota(1, A.extent(1) - 1),  
    stdv::iota(1, A.extent(2) - 1));
```

```
std::for_each(ex::par_unseq,  
    begin(v), end(v),  
    [=] (auto idx) {  
        auto [i, j, k] = idx;  
        B[i, j, k] = ( A[i, j, k-1] +  
                      A[i-1, j, k] +  
                      A[i, j-1, k] + A[i, j, k] + A[i, j+1, k]  
                      + A[i+1, j, k]  
                      + A[i, j, k+1] ) / 7.0  
    });
```



```

std::mdspan A{input,
               std::layout_left::mapping{N, M, O}};
std::mdspan B{output,
               std::layout_left::mapping{N, M, O}};

```

```

auto v = stdv::cartesian_product(
    stdv::iota(1, A.extent(0) - 1),
    stdv::iota(1, A.extent(1) - 1),
    stdv::iota(1, A.extent(2) - 1));

std::for_each(ex::par_unseq,
    begin(v), end(v),
    [=] (auto idx) {
        auto [i, j, k] = idx;
        B[i, j, k] = ( A[i, j, k-1] +
                       A[i-1, j, k] +
                       A[i, j-1, k] + A[i, j, k] + A[i, j+1, k]
                       + A[i+1, j, k]
                       + A[i, j, k+1] ) / 7.0
    });

```

```
std::span A{input,  N * M};  
std::span B{output, M * N};  
  
auto v = stdv::cartesian_product(  
    stdv::iota(0, N),  
    stdv::iota(0, M));  
  
std::for_each(ex::par_unseq,  
    begin(v), end(v),  
    [=] (auto idx) {  
        auto [i, j] = idx;  
        B[i + j * N] = A[i * M + j];  
    });
```

```
std::mdspan A{input,  N, M};  
std::mdspan B{output, M, N};  
  
auto v = stdv::cartesian_product(  
    stdv::iota(0, A.extent(0)),  
    stdv::iota(0, A.extent(1)));  
  
std::for_each(ex::par_unseq,  
    begin(v), end(v),  
    [=] (auto idx) {  
        auto [i, j] = idx;  
        B[j, i] = A[i, j];  
    });
```

```
std::mdspan A{input,  N, M};  
std::mdspan B{output, M, N};
```

```
stdr::for_each(  
    ex::par_unseq,  
    A.indices(),  
    [=] (auto [i, j]) {  
        B[j, i] = A[i, j];  
    });
```

```
std::mdspan A{input,  N, M};  
std::mdspan B{output, M, N};
```

```
ex::sender auto s =  
    ex::transfer_just(sch, A.indices())  
    | for_each_async(  
        [=] (auto [i, j]) {  
            B[j, i] = A[i, j];  
        });
```

```
submdspan(mdspan<...> m, SliceSpecifiers... ss)  
    -> mdspan<...>
```

submdspan(mdspan<...> m, SliceSpecifiers... ss)
-> mdspan<...>

Slice Specifier	Argument	Reduces Rank?
Single Index	Integral	

submdspan(mdspan<...> m, SliceSpecifiers... ss)
-> mdspan<...>

Slice Specifier	Argument	Reduces Rank?
Single Index	Integral	✓
Range of Indices	std::pair<Integral, Integral> std::tuple<Integral, Integral>	✗

submdspan(mdspan<...> m, SliceSpecifiers... ss)
-> mdspan<...>

Slice Specifier	Argument	Reduces Rank?
Single Index	Integral	✓
Range of Indices	std::pair<Integral, Integral> std::tuple<Integral, Integral>	✗
All Indices	std::full_extent	✗

```
std::mdspan m0{data, 64, 128, 32};  
  
auto m1 = std::submdspan(m0, std::tuple{15, 23},  
                           std::tuple{31, 39},  
                           std::tuple{ 7, 15});
```

```
std::mdspan m0{data, 64, 128, 32};  
  
auto m1 = std::submdspan(m0, std::tuple{15, 23},  
                           std::tuple{31, 39},  
                           std::tuple{ 7, 15});  
  
m1.rank() == 3
```

```
std::mdspan m0{data, 64, 128, 32};

auto m1 = std::submdspan(m0, std::tuple{15, 23},
                           std::tuple{31, 39},
                           std::tuple{ 7, 15});

m1.rank()      == 3
m1.extent(0)   == 8
m1.extent(1)   == 8
m1.extent(2)   == 8
```

```
std::mdspan m0{data, 64, 128, 32};

auto m1 = std::submdspan(m0, std::tuple{15, 23},
                          std::tuple{31, 39},
                          std::tuple{ 7, 15});

m1.rank()      == 3
m1.extent(0)   == 8
m1.extent(1)   == 8
m1.extent(2)   == 8
m1[i, j, k]    == m0[i + 15, j + 31, k + 7]
```

```

std::mdspan m0{data, 64, 128, 32};

auto m1 = std::submdspan(m0, std::tuple{15, 23},
                          std::tuple{31, 39},
                          std::tuple{ 7, 15});

m1.rank()      == 3
m1.extent(0)   == 8
m1.extent(1)   == 8
m1.extent(2)   == 8
m1[i, j, k]    == m0[i + 15, j + 31, k + 7]

auto m2 = std::submdspan(m0, 15,
                          std::full_extent,
                          31);

```

```

std::mdspan m0{data, 64, 128, 32};

auto m1 = std::submdspan(m0, std::tuple{15, 23},
                        std::tuple{31, 39},
                        std::tuple{ 7, 15});

m1.rank()      == 3
m1.extent(0)   == 8
m1.extent(1)   == 8
m1.extent(2)   == 8
m1[i, j, k]    == m0[i + 15, j + 31, k + 7]

auto m2 = std::submdspan(m0, 15,
                        std::full_extent,
                        31);

m2.rank()      == 1

```

```

std::mdspan m0{data, 64, 128, 32};

auto m1 = std::submdspan(m0, std::tuple{15, 23},
                        std::tuple{31, 39},
                        std::tuple{ 7, 15});

m1.rank()      == 3
m1.extent(0)   == 8
m1.extent(1)   == 8
m1.extent(2)   == 8
m1[i, j, k]    == m0[i + 15, j + 31, k + 7]

auto m2 = std::submdspan(m0, 15,
                        std::full_extent,
                        31);

m2.rank()      == 1
m2.extent(0)   == 128

```



```

std::mdspan m0{data, 64, 128, 32};

auto m1 = std::submdspan(m0, std::tuple{15, 23},
                        std::tuple{31, 39},
                        std::tuple{ 7, 15});

m1.rank()      == 3
m1.extent(0)   == 8
m1.extent(1)   == 8
m1.extent(2)   == 8
m1[i, j, k]    == m0[i + 15, j + 31, k + 7]

auto m2 = std::submdspan(m0, 15,
                        std::full_extent,
                        31);

m2.rank()      == 1
m2.extent(0)   == 128
m2[j]          == m0[15, j, 31]

```

```
std::mdspan A{input,  N,  M};  
std::mdspan B{output, M,  N};  
std::size_t T = ...;
```

```
std::mdspan A{input,  N, M};  
std::mdspan B{output, M, N};  
std::size_t T = ...;  
  
auto outer = stdv::cartesian_product(stdv::iota(0, (N + T - 1) / T),  
                                     stdv::iota(0, (M + T - 1) / T));
```

```

std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};
std::size_t T = ...;

auto outer = stdv::cartesian_product(stdv::iota(0, (N + T - 1) / T),
                                     stdv::iota(0, (M + T - 1) / T));

std::for_each(ex::par_unseq, begin(outer), end(outer),
  [=] (auto tile) {
    auto [x, y] = tile;
    ...
  });

```

```

std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};
std::size_t T = ...;

auto outer = std::cartesian_product(std::iota(0, (N + T - 1) / T),
                                     std::iota(0, (M + T - 1) / T));

std::for_each(ex::par_unseq, begin(outer), end(outer),
  [=] (auto tile) {
    auto [x, y] = tile;
    std::tuple selectN{T * x, std::min(T * (x + 1), N)};
    std::tuple selectM{T * y, std::min(T * (y + 1), M)};

    ...
  });

```

```

std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};
std::size_t T = ...;

auto outer = stdv::cartesian_product(stdv::iota(0, (N + T - 1) / T),
                                     stdv::iota(0, (M + T - 1) / T));

std::for_each(ex::par_unseq, begin(outer), end(outer),
  [=] (auto tile) {
    auto [x, y] = tile;
    std::tuple selectN{T * x, std::min(T * (x + 1), N)};
    std::tuple selectM{T * y, std::min(T * (y + 1), M)};

    auto TA = std::submdspan(A, selectN, selectM);
    auto TB = std::submdspan(B, selectM, selectN);

    ...
  });

```

```

std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};
std::size_t T = ...;

auto outer = stdv::cartesian_product(stdv::iota(0, (N + T - 1) / T),
                                     stdv::iota(0, (M + T - 1) / T));

std::for_each(ex::par_unseq, begin(outer), end(outer),
  [=] (auto tile) {
    auto [x, y] = tile;
    std::tuple selectN{T * x, std::min(T * (x + 1), N)};
    std::tuple selectM{T * y, std::min(T * (y + 1), M)};

    auto TA = std::submdspan(A, selectN, selectM);
    auto TB = std::submdspan(B, selectM, selectN);

    auto inner = stdv::cartesian_product(stdv::iota(0, TA.extent(0)),
                                          stdv::iota(0, TA.extent(1)));

    ...
  });

```

```

std::mdspan A{input,  N, M};
std::mdspan B{output, M, N};
std::size_t T = ...;

auto outer = stdv::cartesian_product(stdv::iota(0, (N + T - 1) / T),
                                     stdv::iota(0, (M + T - 1) / T));

std::for_each(ex::par_unseq, begin(outer), end(outer),
  [=] (auto tile) {
    auto [x, y] = tile;
    std::tuple selectN{T * x, std::min(T * (x + 1), N)};
    std::tuple selectM{T * y, std::min(T * (y + 1), M)};

    auto TA = std::submdspan(A, selectN, selectM);
    auto TB = std::submdspan(B, selectM, selectN);

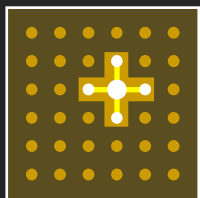
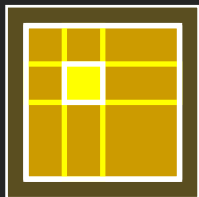
    auto inner = stdv::cartesian_product(stdv::iota(0, TA.extent(0)),
                                          stdv::iota(0, TA.extent(1)));

    for (auto [i, j] : inner)
      TB[j, i] = TA[i, j];
  });

```


Pillars of Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



Expanding the Set

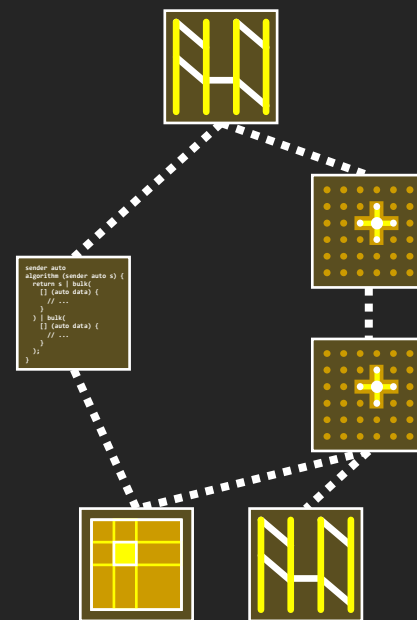
Tools to Write Your Own Parallel Algorithms that Run Anywhere



```
sender auto  
algorithm (sender auto s) {  
    return s | bulk(N,  
        [] (auto data) {  
            // ...  
        }) | bulk(N,  
        [] (auto data) {  
            // ...  
        });  
}
```



Mechanisms for Composing Parallel Invocations into Task Graphs



```
std::mdspan A{..., N, M};  
std::mdspan x{..., M};  
std::mdspan y{..., N};  
  
//  $y = 3.0 A x + 2.0 y$   
std::matrix_vector_product(  
    ex::par_unseq,  
    std::scaled(3.0, A), x,  
    std::scaled(2.0, y), y);
```

```
std::mdspan A{..., N, M};  
std::mdspan x{..., M};  
std::mdspan y{..., N};  
  
//  $y = 3.0 A x + 2.0 y$   
std::matrix_vector_product(  
    ex::par_unseq,  
    std::scaled(3.0, A), x,  
    std::scaled(2.0, y), y);
```

```
std::mdspan A{..., N, M};  
std::mdspan x{..., M};  
std::mdspan y{..., N};  
  
//  $y = 3.0 A x + 2.0 y$   
std::matrix_vector_product(  
    ex::par_unseq,  
    std::scaled(3.0, A), x,  
    std::scaled(2.0, y), y);
```

```

std::mdspan A{..., N, M};
std::mdspan x{..., M};
std::mdspan b{..., N};

// Solve  $A x = b$  where  $A = U^T U$ 

// Solve  $U^T c = b$ , using  $x$  to store  $c$ 
std::triangular_matrix_vector_solve(ex::par_unseq,
                                   std::transposed(A),
                                   std::upper_triangle, std::explicit_diagonal,
                                   b, x);

// Solve  $U x = c$ , overwriting  $x$  with result
std::triangular_matrix_vector_solve(ex::par_unseq,
                                   A,
                                   std::upper_triangle, std::explicit_diagonal,
                                   x);

```

```

std::mdspan A{..., N, M};
std::mdspan x{..., M};
std::mdspan b{..., N};

// Solve  $A x = b$  where  $A = U^T U$ 

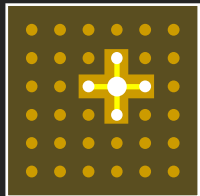
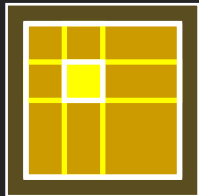
// Solve  $U^T c = b$ , using  $x$  to store  $c$ 
std::triangular_matrix_vector_solve(ex::par_unseq,
                                   std::transposed(A),
                                   std::upper_triangle, std::explicit_diagonal,
                                   b, x);

// Solve  $U x = c$ , overwriting  $x$  with result
std::triangular_matrix_vector_solve(ex::par_unseq,
                                   A,
                                   std::upper_triangle, std::explicit_diagonal,
                                   x);

```

Pillars of Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



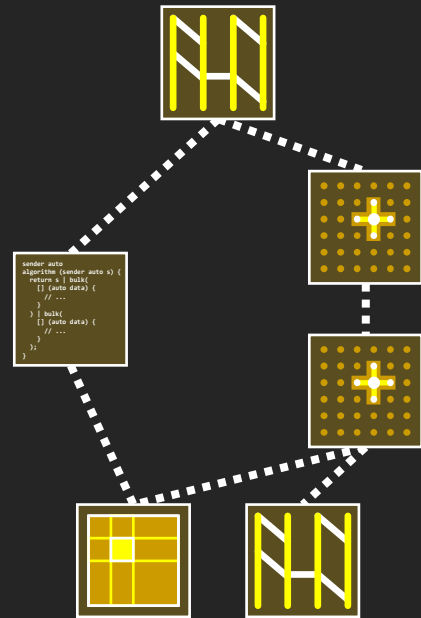
Tools to Write Your Own Parallel Algorithms that Run Anywhere



```
sender auto  
algorithm (sender auto s) {  
    return s | bulk(N,  
        [] (auto data) {  
            // ...  
        }) | bulk(N,  
        [] (auto data) {  
            // ...  
        });  
}
```



Mechanisms for Composing Parallel Invocations into Task Graphs



We Need On-Ramps

