NVIDIA

STANDARD PARALLELISM

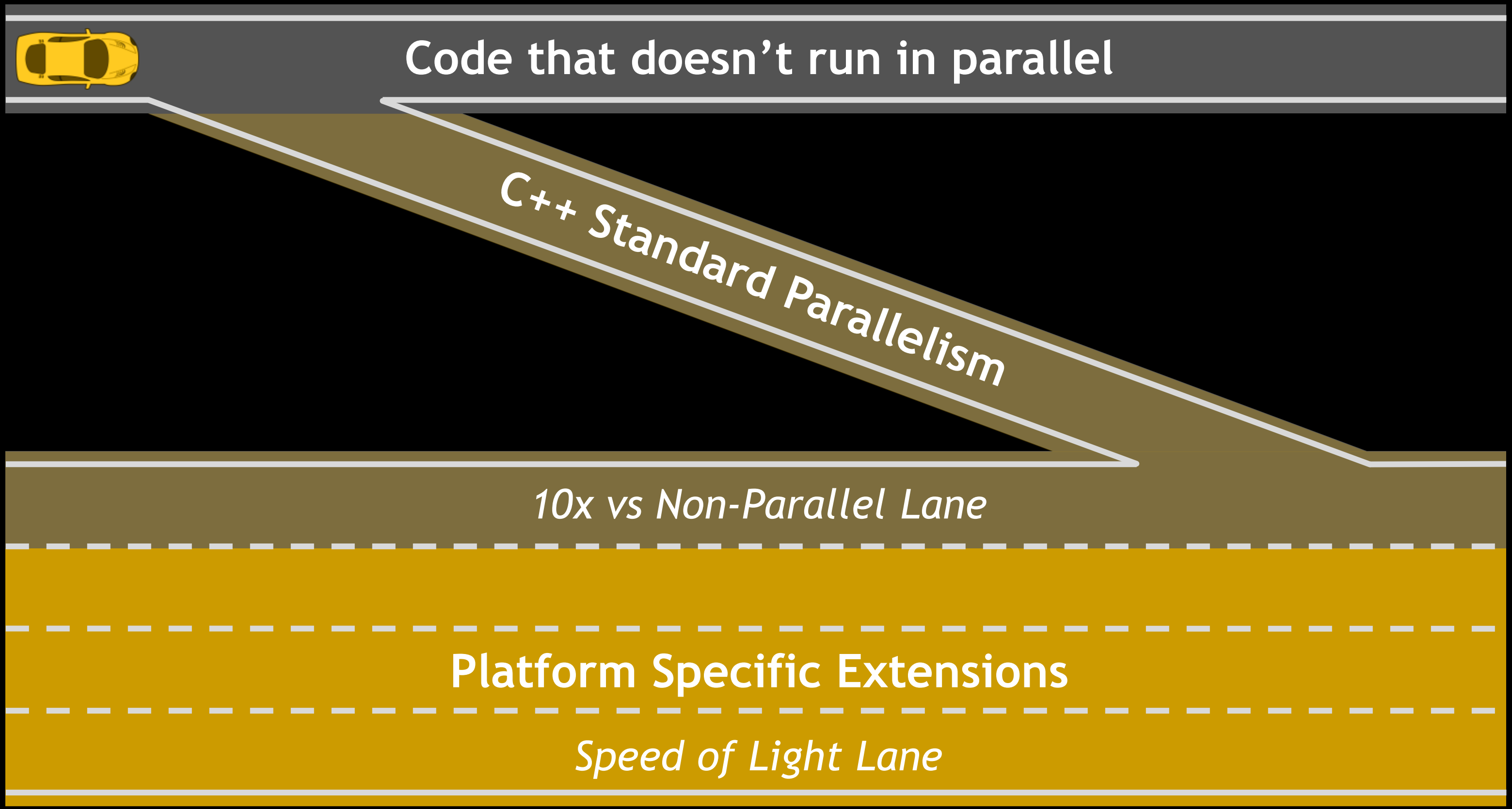
Bryce Adelstein Lelbach

HPC Programming Models Architect

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

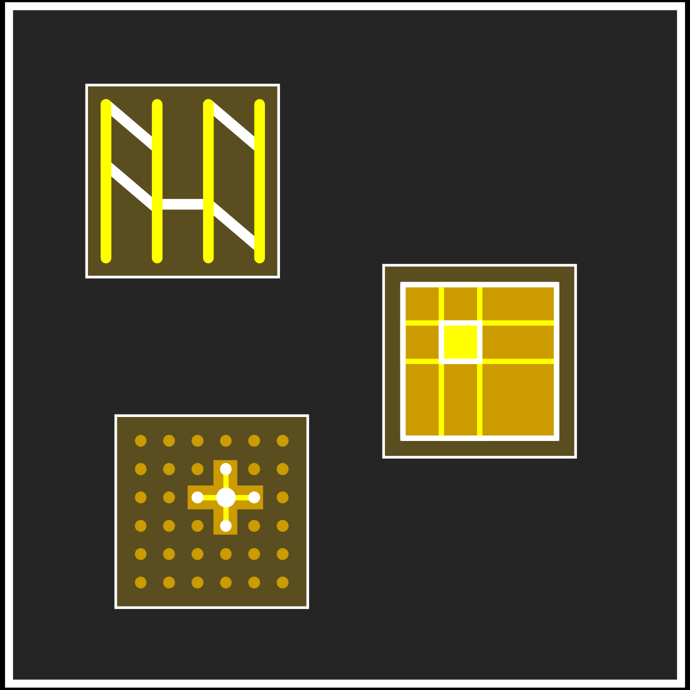
@blelbach

We Need On-Ramps



Pillars of C++ Standard Parallelism

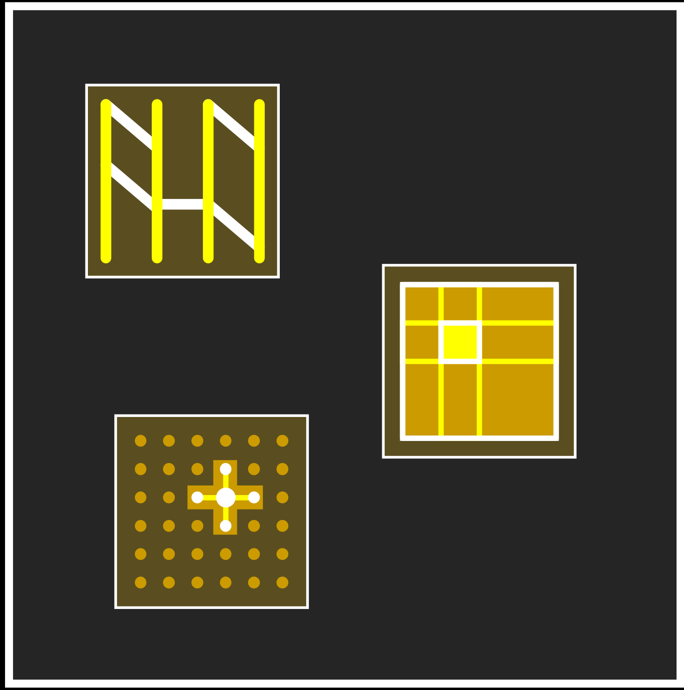
Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries

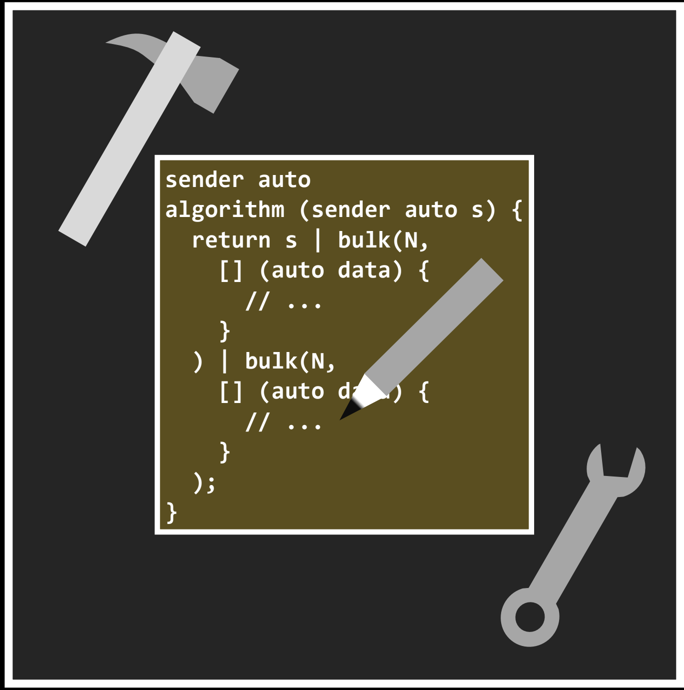


Pillars of C++ Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries

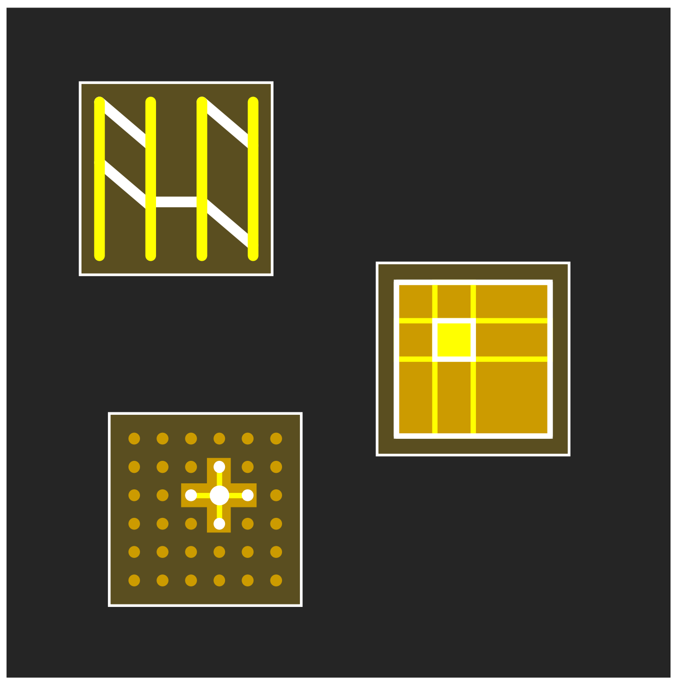
Tools to Write Your Own Parallel Algorithms that Run Anywhere



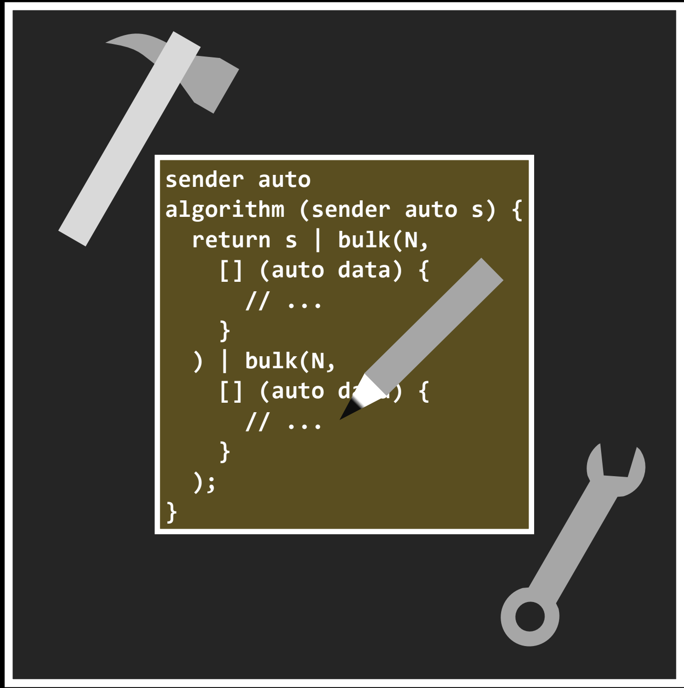


Pillars of C++ Standard Parallelism

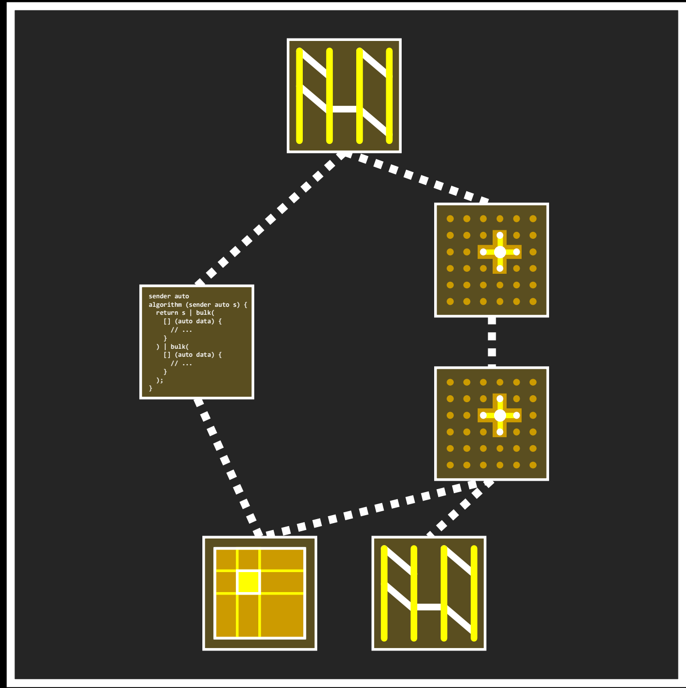
Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



Tools to Write Your Own Parallel Algorithms that Run Anywhere



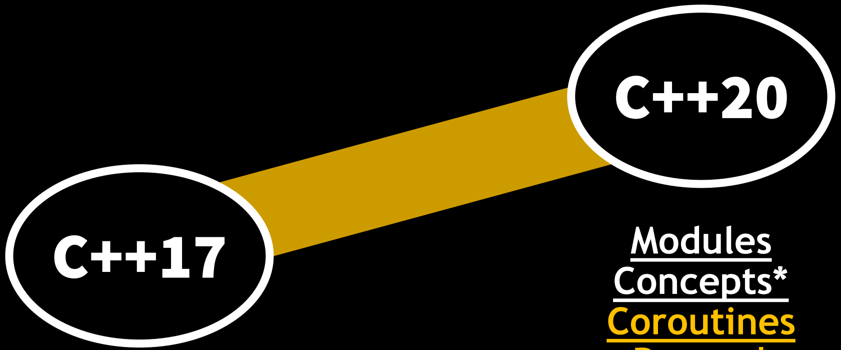
Mechanisms for Composing Parallel Invocations into Task Graphs



C++17

Parallel Algorithms*
Forward Progress*
New Memory Model*

* = Available now in NVC++



Parallel Algorithms*
Forward Progress*
New Memory Model*

Modules
Concepts*
Coroutines
Ranges*
atomic wait*
atomic_ref*
barrier*

* = Available now in NVC++

C++17

Parallel Algorithms*
Forward Progress*
New Memory Model*

C++20

Modules
Concepts*
Coroutines
Ranges*
`atomic wait*`
`atomic_ref*`
`barrier*`

C++23

Ranges*
`mdspan*`
`generator`
Extended FP

* = Available now in NVC++

C++17

Parallel Algorithms*
Forward Progress*
New Memory Model*

C++20

Modules
Concepts*
Coroutines
Ranges*
`atomic wait*`
`atomic_ref*`
`barrier*`

C++23

Ranges*
`mdspan*`
`generator`
Extended FP

C++26

Reflection
Senders*
Linear Algebra*
SIMD
RCU
Hazard Pointers

* = Available now in NVC++

Standard Algorithms

Serial (C++98)

```
std::vector<T> x{…};

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

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

std::transform(
 begin(x), end(x), begin(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

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

Standard Algorithms

Serial (C++98)

```
std::vector<T> x{…};

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

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

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

Parallel (C++17)

```
std::vector<T> x{…};

std::transform(
 ex::par_unseq,
 begin(x), end(x), begin(x)
 f);
std::transform(
 ex::par_unseq,
 begin(x), end(x), begin(x)
 g);
std::transform(
 ex::par_unseq,
 begin(x), end(x), begin(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 …
`std::execution::seq`	In the calling thread	Indeterminately sequenced

Execution Policy	Operations occur …	Operations are …
`std::execution::seq`	In the calling thread	Indeterminately sequenced
`std::execution::unseq`	In the calling thread	Unsequenced

Execution Policy	Operations occur …	Operations are …
`std::execution::seq`	In the calling thread	Indeterminately sequenced
`std::execution::unseq`	In the calling thread	Unsequenced
`std::execution::par`	Potentially in multiple threads	Indeterminately sequenced within each thread

Execution Policy	Operations occur …	Operations are …
`std::execution::seq`	In the calling thread	Indeterminately sequenced
`std::execution::unseq`	In the calling thread	Unsequenced
`std::execution::par`	Potentially in multiple threads	Indeterminately sequenced within each thread
`std::execution::par_unseq`	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
                         10001000001001001000100000001000000000
                         10010000100010001001000000001000000000
                         10010000010001000001000010010000100000;
```

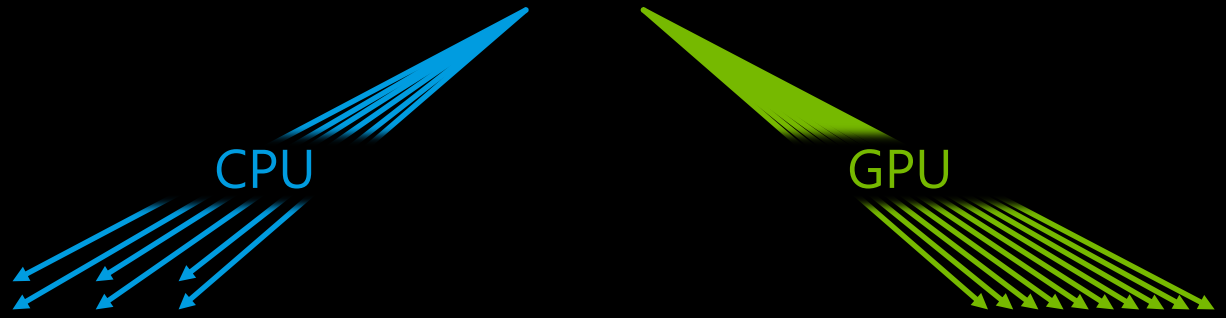
```
std::size_t word_count(std::string_view s) {
  if (s.empty()) return 0;
  return std::transform_reduce(ex::par_unseq,
    begin(s), end(s) - 1, begin(s) + 1,
    std::size_t(!std::isspace(s.front()) ? 1 : 0),
    …
    [] (char l, char r) { return std::isspace(l) && !std::isspace(r); }
  );
}

std::size_t result     = 10000010000010000010001010000010100000
                         10001000001001001000100000001000000000
                         10010000100010001001000000001000000000
                         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);
```

Standard Parallel Algorithms

```
std::vector<double> x(…), y(…);
double dot_product = std::transform_reduce(std::execution::par,
                                           x.begin(), x.end(), y.begin());
```



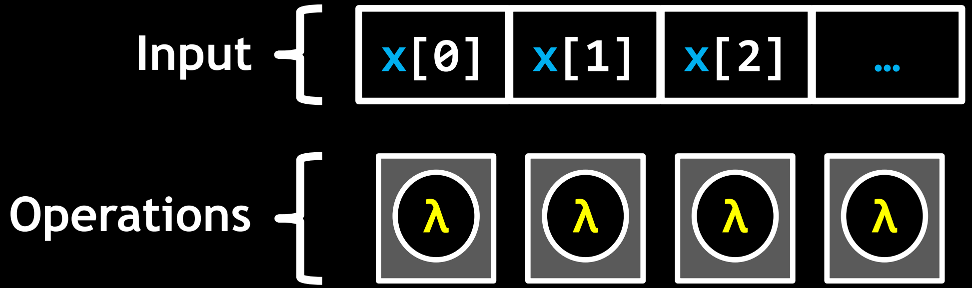
`nvc++ -stdpar=multicore`   `nvc++ -stdpar=gpu`

Since C++17 and the initial release of NVC++!

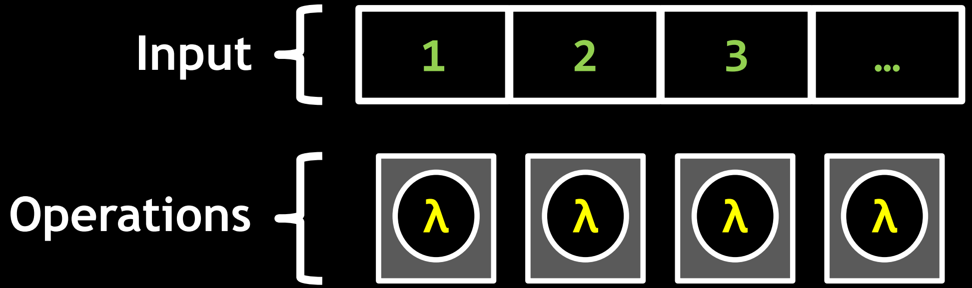
In C++20, the Standard Library introduced ranges.

Unlike iterators, ranges are composable and can be lazy.

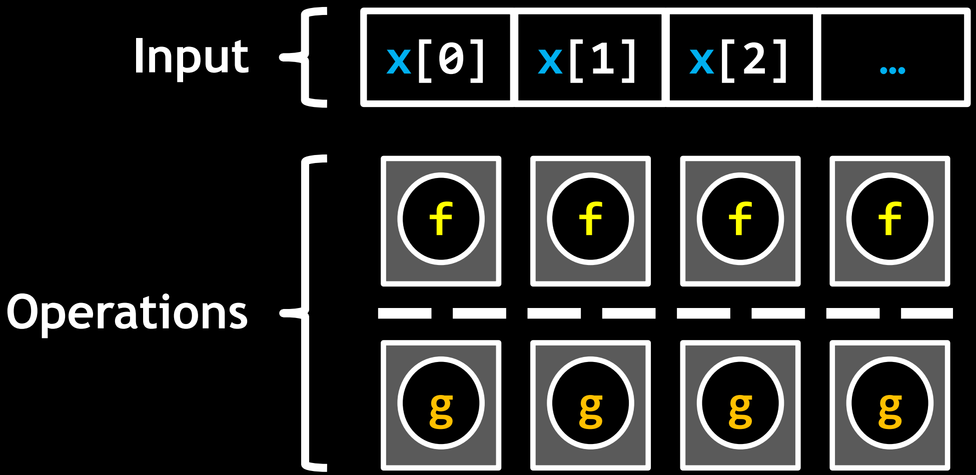
```
std::vector x{…};

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



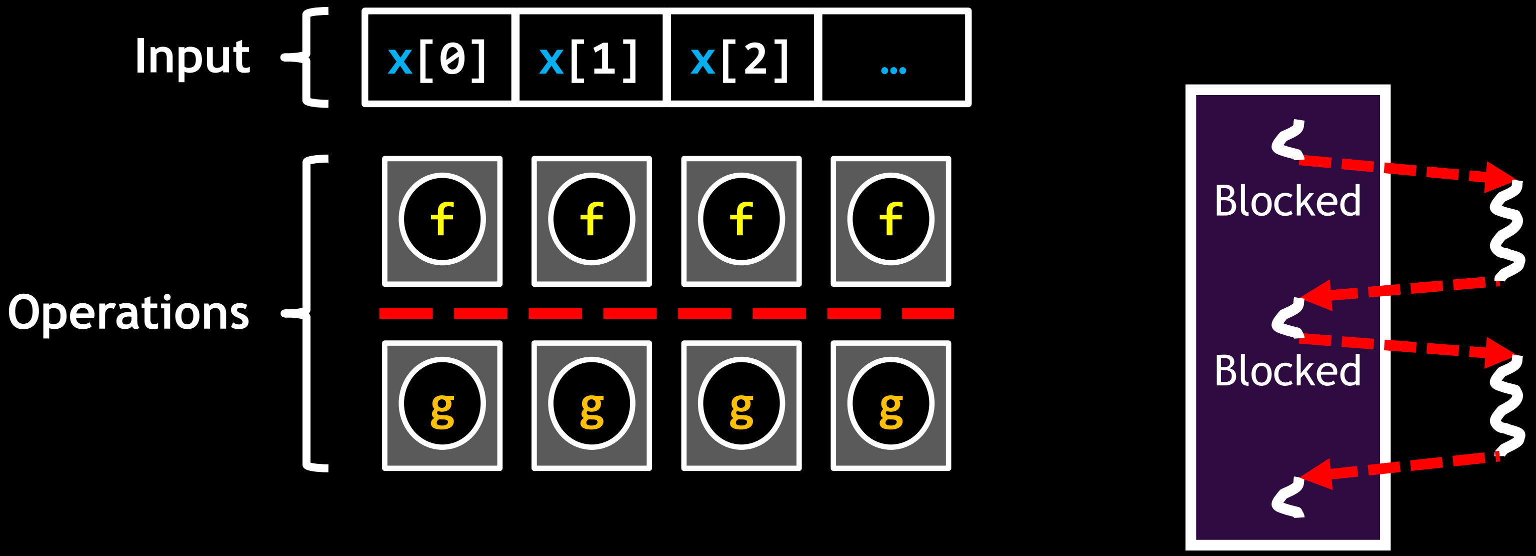
```
auto v = stdv::iota(1, N);

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



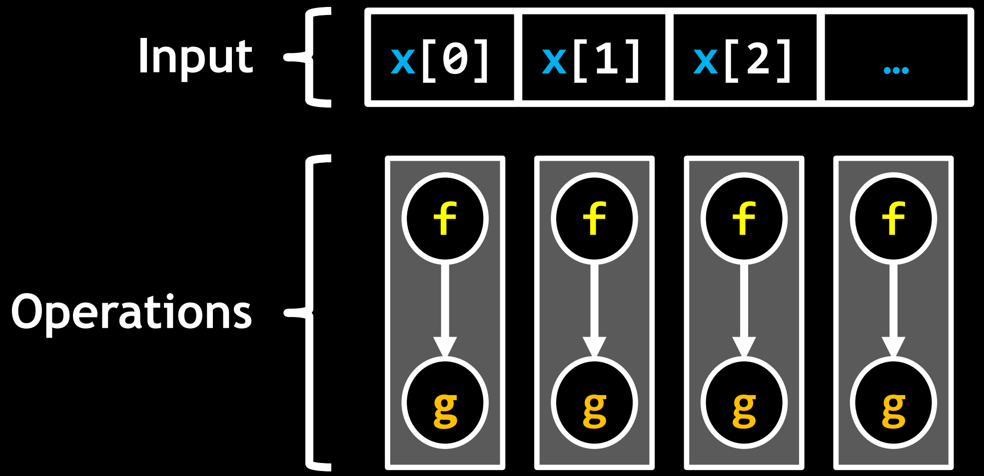
```
std::vector x{…};

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



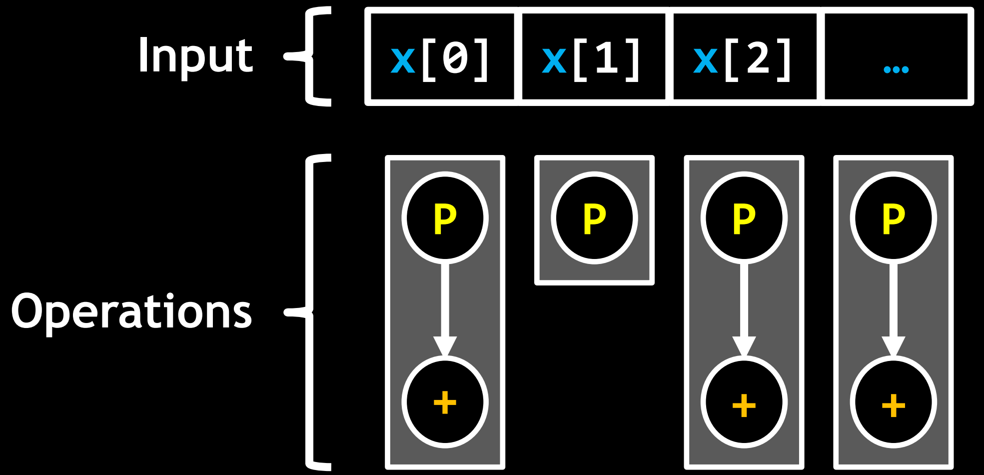
```
std::vector x{…};

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 = stdv::transform(x, f);

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

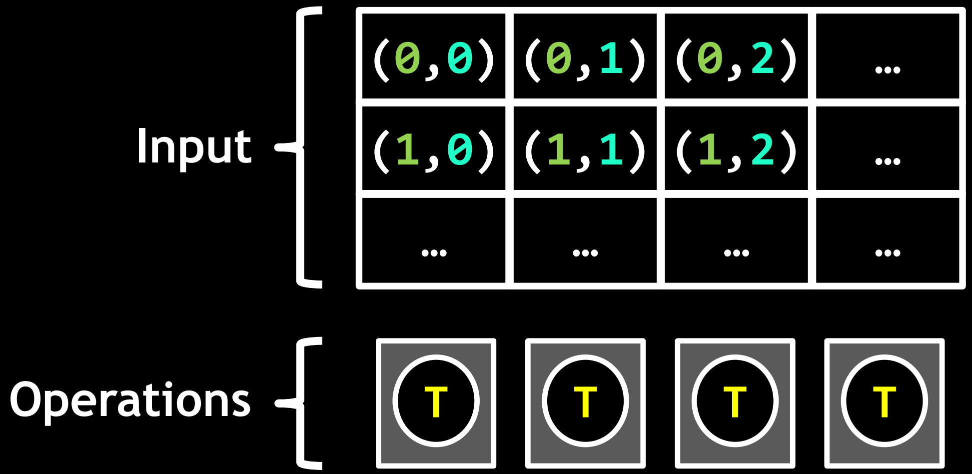


```
std::vector x{…};

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

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



```
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));
```



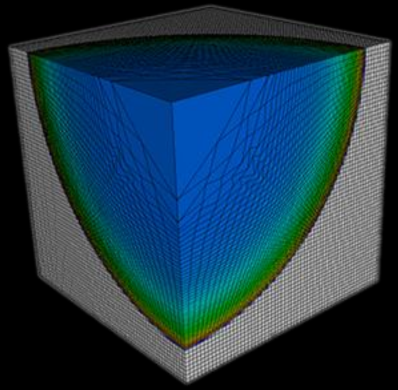
Standard Parallel Algorithms & Ranges

```
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];
  });
```

Available since C++23, NVC++ 22.5, and libstdc++ 13!

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++.
- Versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos, Standard C++, …



https://github.com/LLNL/LULESH

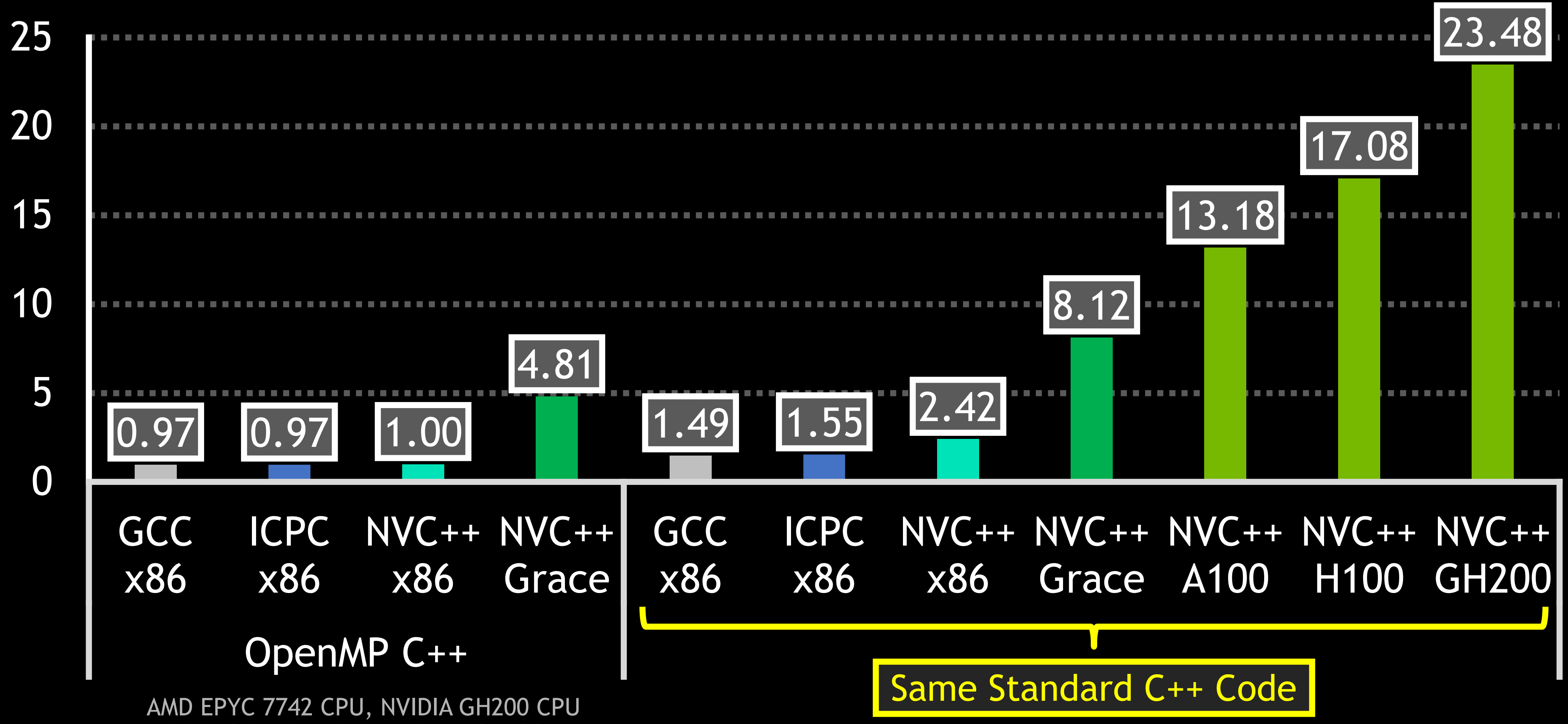
```
static inline void CalcHydroConstraintForElems(
  Domain &domain, Index_t length,
  Index_t *regElemlist, Real_t dvovmax, Real_t& dthydro) {
  #if _OPENMP
    const Index_t threads = omp_get_max_threads();
    Index_t hydro_elem_per_thread[threads];
    Real_t dthydro_per_thread[threads];
  #else
    Index_t threads = 1;
    Index_t hydro_elem_per_thread[1];
    Real_t dthydro_per_thread[1];
  #endif
  #pragma omp parallel firstprivate(length, dvovmax)
  {
    Real_t dthydro_tmp = dthydro;
    Index_t hydro_elem = -1;
    #if _OPENMP
      Index_t thread_num = omp_get_thread_num();
    #else
      Index_t thread_num = 0;
    #endif
    #pragma omp for
    for (Index_t i = 0 ; i < length ; ++i) {
      Index_t indx = regElemlist[i] ;

      if (domain.vdov(indx) != Real_t(0.)) {
        Real_t dtdvov = dvovmax / (FABS(domain.vdov(indx))+Real_t(1.e-20));
        if (dthydro_tmp > dtdvov) {
          dthydro_tmp = dtdvov;
          hydro_elem = indx;
        }
      }
    }
    dthydro_per_thread[thread_num] = dthydro_tmp ;
    hydro_elem_per_thread[thread_num] = hydro_elem ;
  }
  for (Index_t i = 1; i < threads; ++i) {
    if(dthydro_per_thread[i] < dthydro_per_thread[0]) {
      dthydro_per_thread[0] = dthydro_per_thread[i];
      hydro_elem_per_thread[0] = hydro_elem_per_thread[i];
    }
  }
  if (hydro_elem_per_thread[0] != -1) {
    dthydro = dthydro_per_thread[0];
  }
}
```

OpenMP C++

```
static inline void CalcHydroConstraintForElems(Domain &domain, Index_t length,
                                               Index_t *regElemlist,
                                               Real_t dvovmax,
                                               Real_t &dthydro) {
  dthydro = std::transform_reduce(
    std::execution::par, counting_iterator(0), counting_iterator(length),
    dthydro, [](Real_t a, Real_t b) { return a < b ? a : b; },
    [=, &domain](Index_t i) {
      Index_t indx = regElemlist[i];
      if (domain.vdov(indx) == Real_t(0.0)) {
        return std::numeric_limits<Real_t>::max();
      } else {
        return dvovmax / (std::abs(domain.vdov(indx)) + Real_t(1.e-20));
      }
    });
}
```

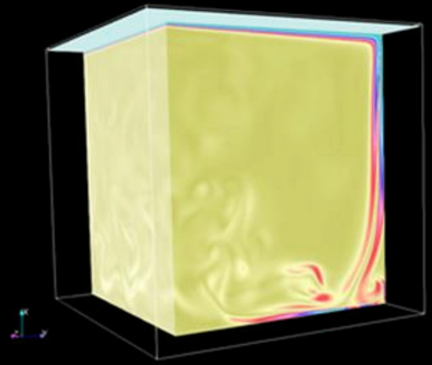
Standard C++

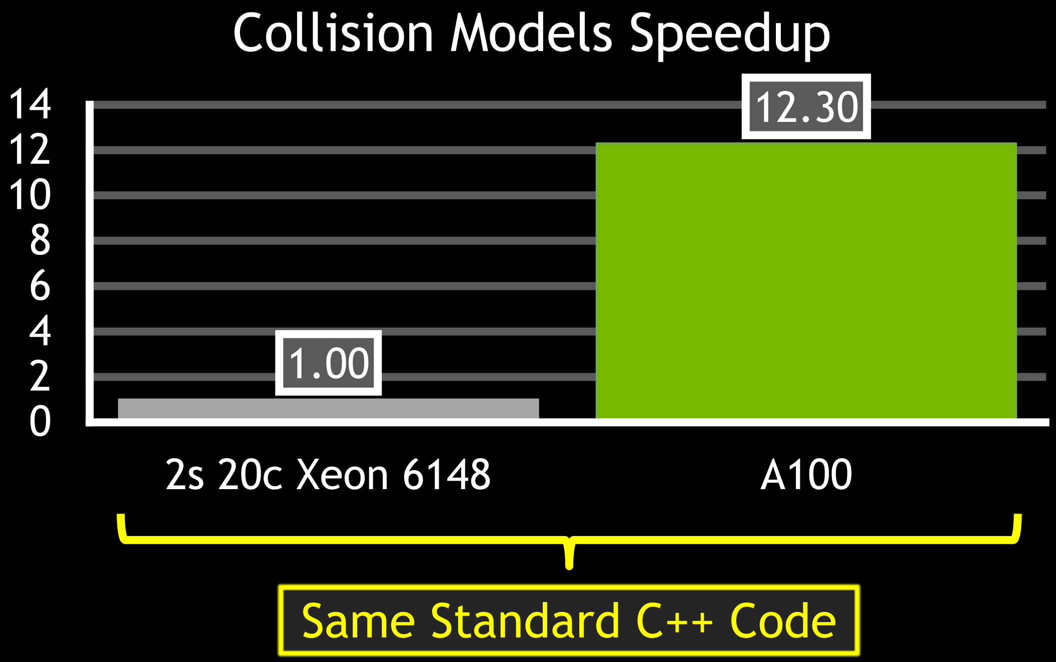
LULESH Speedup



AMD EPYC 7742 CPU, NVIDIA GH200 CPU
GCC 10.3, ICPC 2021.5.0, NVC++ 23.7

STLBM





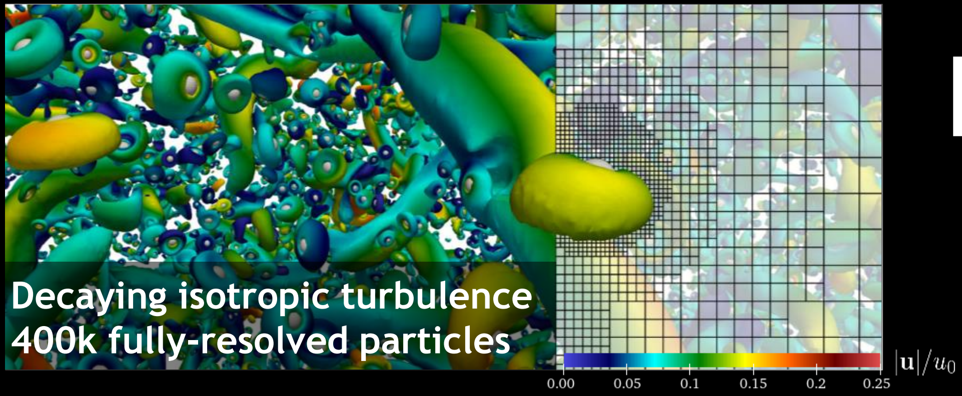
- Framework for parallel Lattice-Boltzmann simulations on multiple targets, including multicore CPUs & GPUs.
- Implemented with C++ Standard Parallelism.
- No language extensions, external libraries, vendor-specific code annotations, or pre-compilation steps.

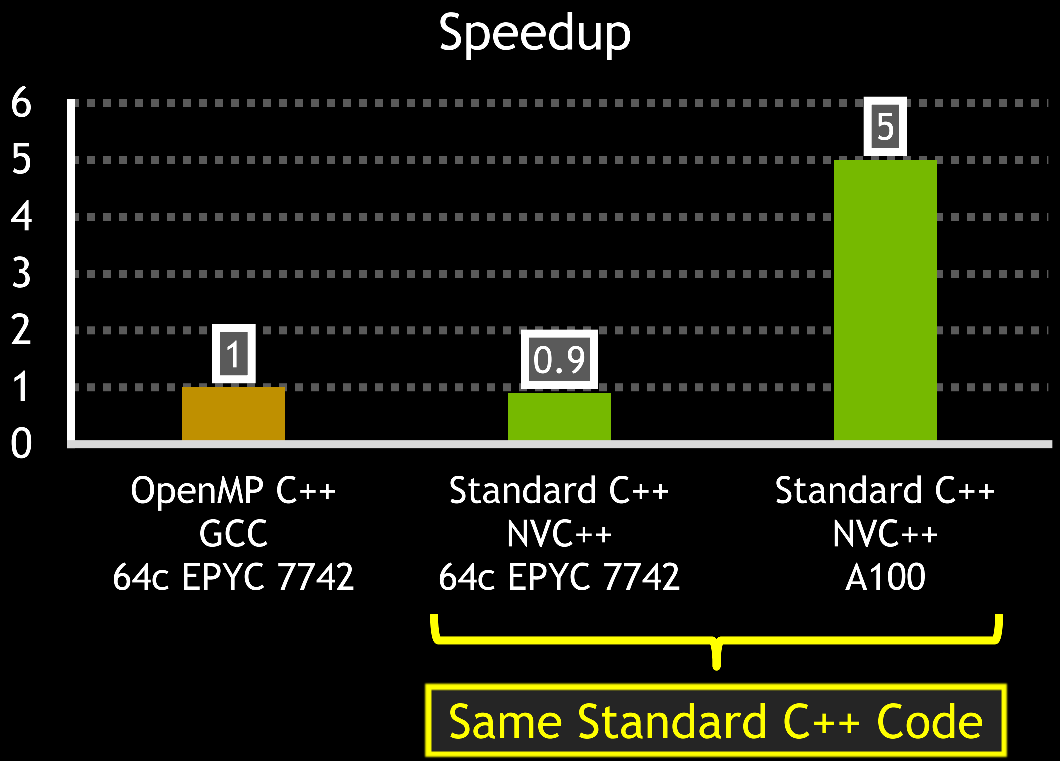
"We have with delight discovered the NVIDIA "stdpar" implementation of C++ Standard Parallel Algorithms. … We believe that the result produces state-of-the-art performance, is highly didactical, and introduces a paradigm shift in cross-platform CPU/GPU programming in the community."

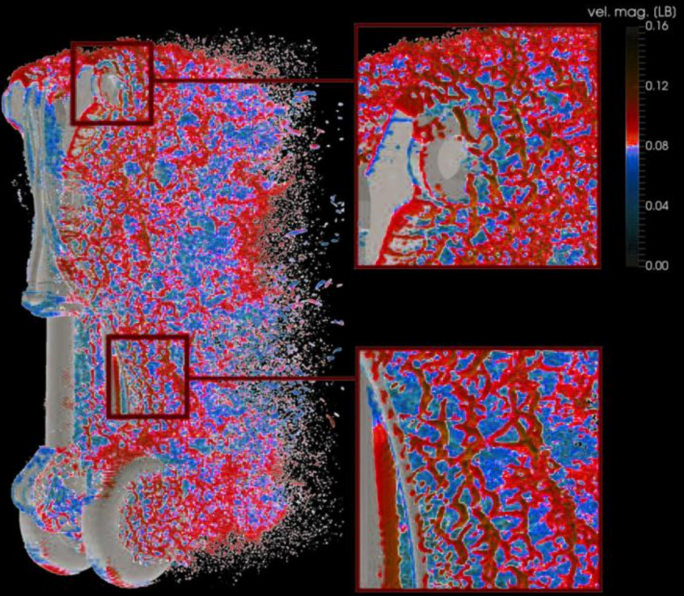
— Professor Jonas Latt, University of Geneva

https://gitlab.com/unigehpfs/stlbm

M-AIA







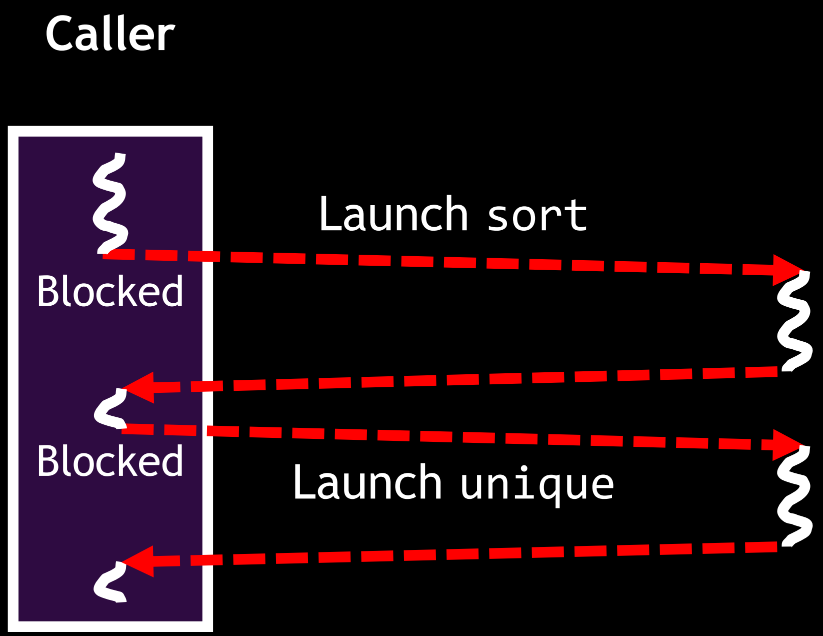
- Package for aerospace flow and noise simulations.
- Adaptive meshing and load balancing, supporting complex moving geometries.
- Solvers include Finite Volume, Navier-Stokes, and Lattice-Boltzmann.
- ~500k lines of C++, developed by ~20 engineers.
- Programming model: MPI & Standard Parallelism.

The C++ parallel algorithms introduced in C++17 are great,

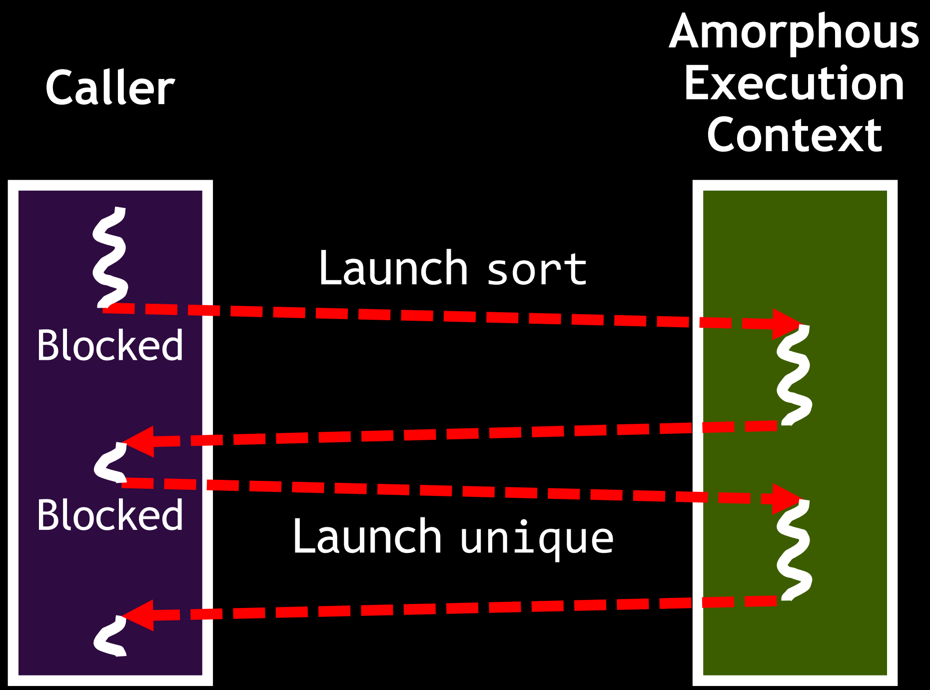
but they're just the start of the story.

```
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));
```



```
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

```cpp
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] = ex::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] = ex::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] = ex::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] = ex::sync_wait(add).value();
```

```cpp
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] = ex::sync_wait(add).value();
```

```cpp
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] = ex::sync_wait(add).value();
```

Schedulers are handles to execution contexts.

Schedulers are handles to execution contexts.

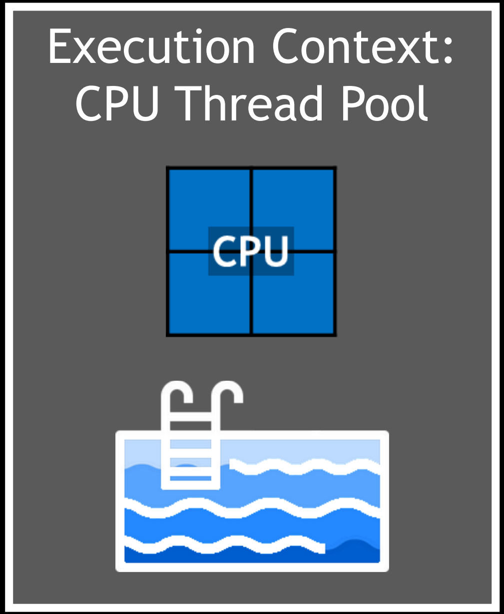
Senders represent asynchronous work.

Schedulers are handles to execution contexts.

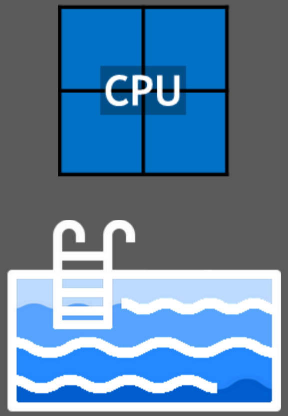
Senders represent asynchronous work.

Receivers process asynchronous signals.

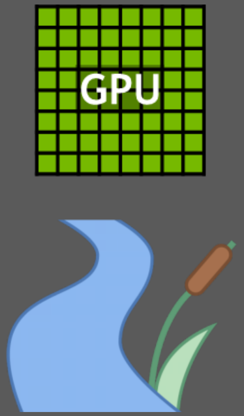
Schedulers are handles to execution contexts.


Execution Context:
CPU Thread Pool
CPU

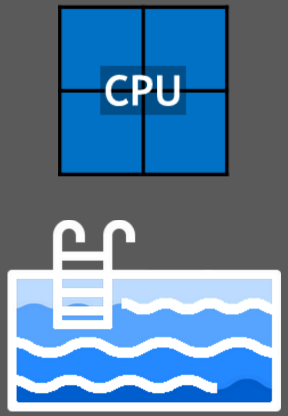
Execution Context:
CPU Thread Pool



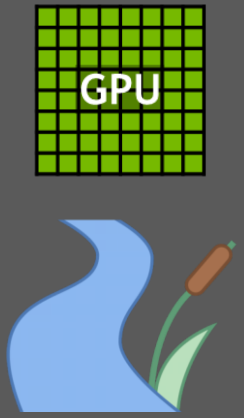
Execution Context:
GPU Stream



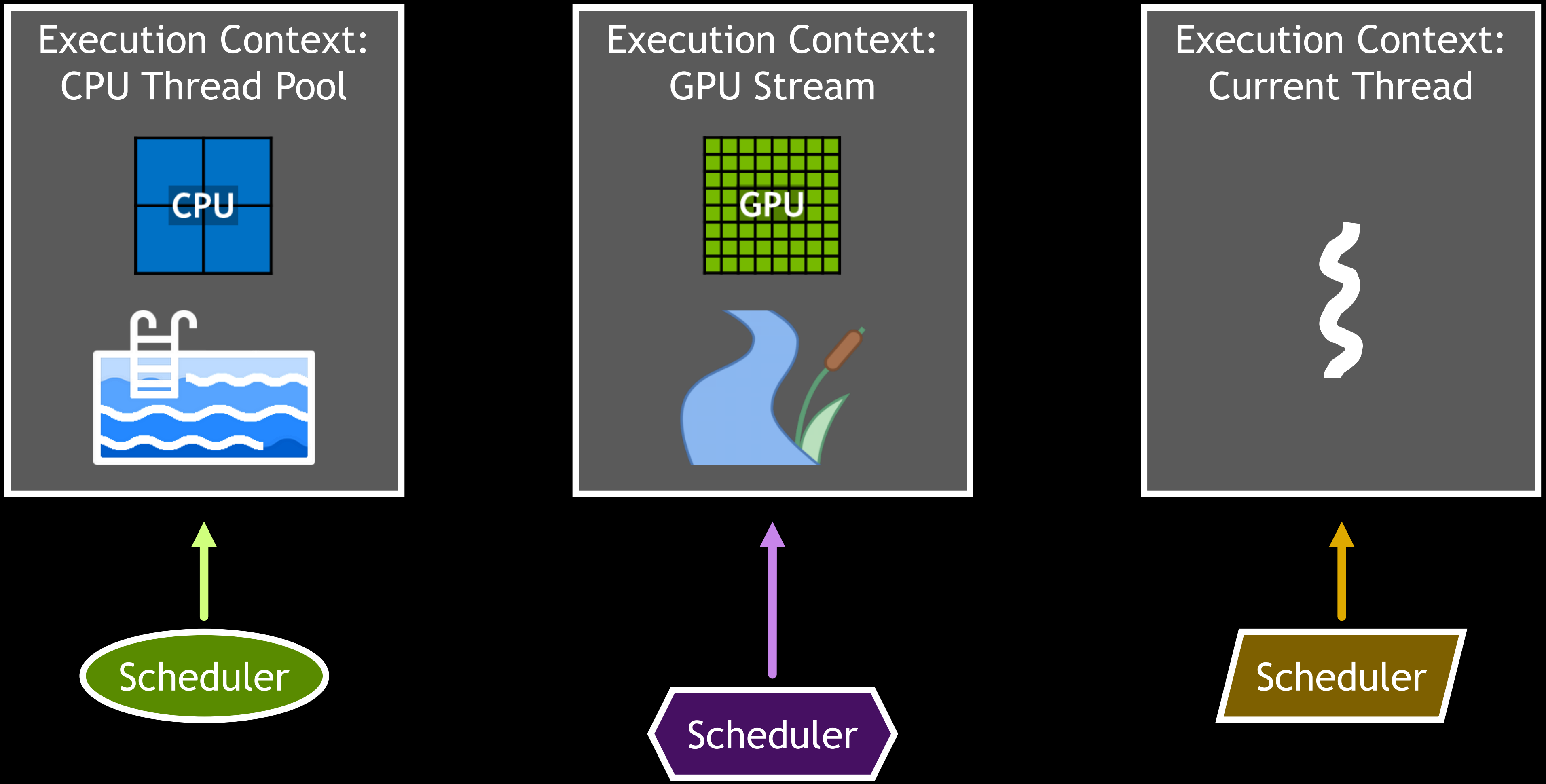
Execution Context:
CPU Thread Pool

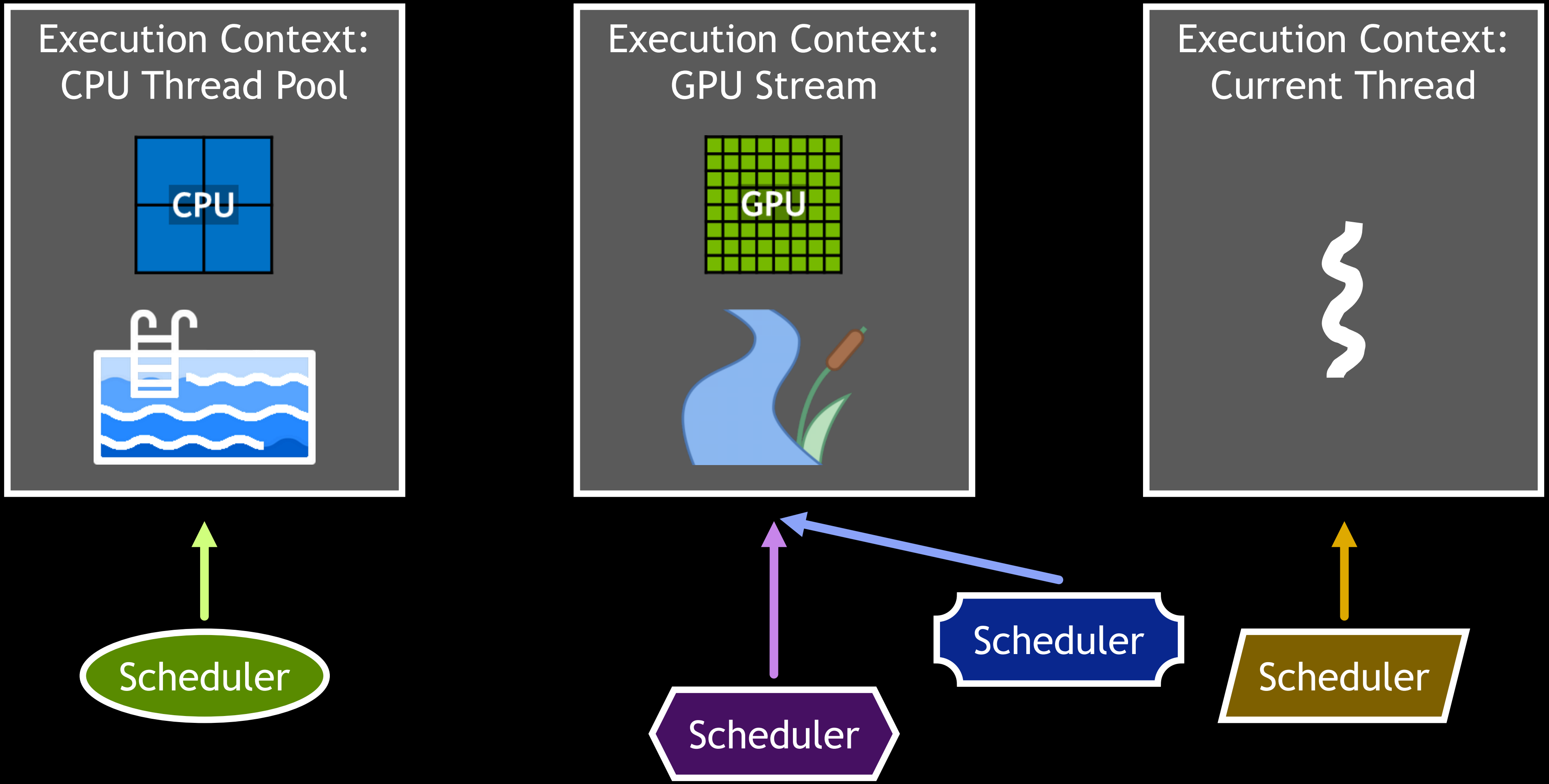


Execution Context:
GPU Stream

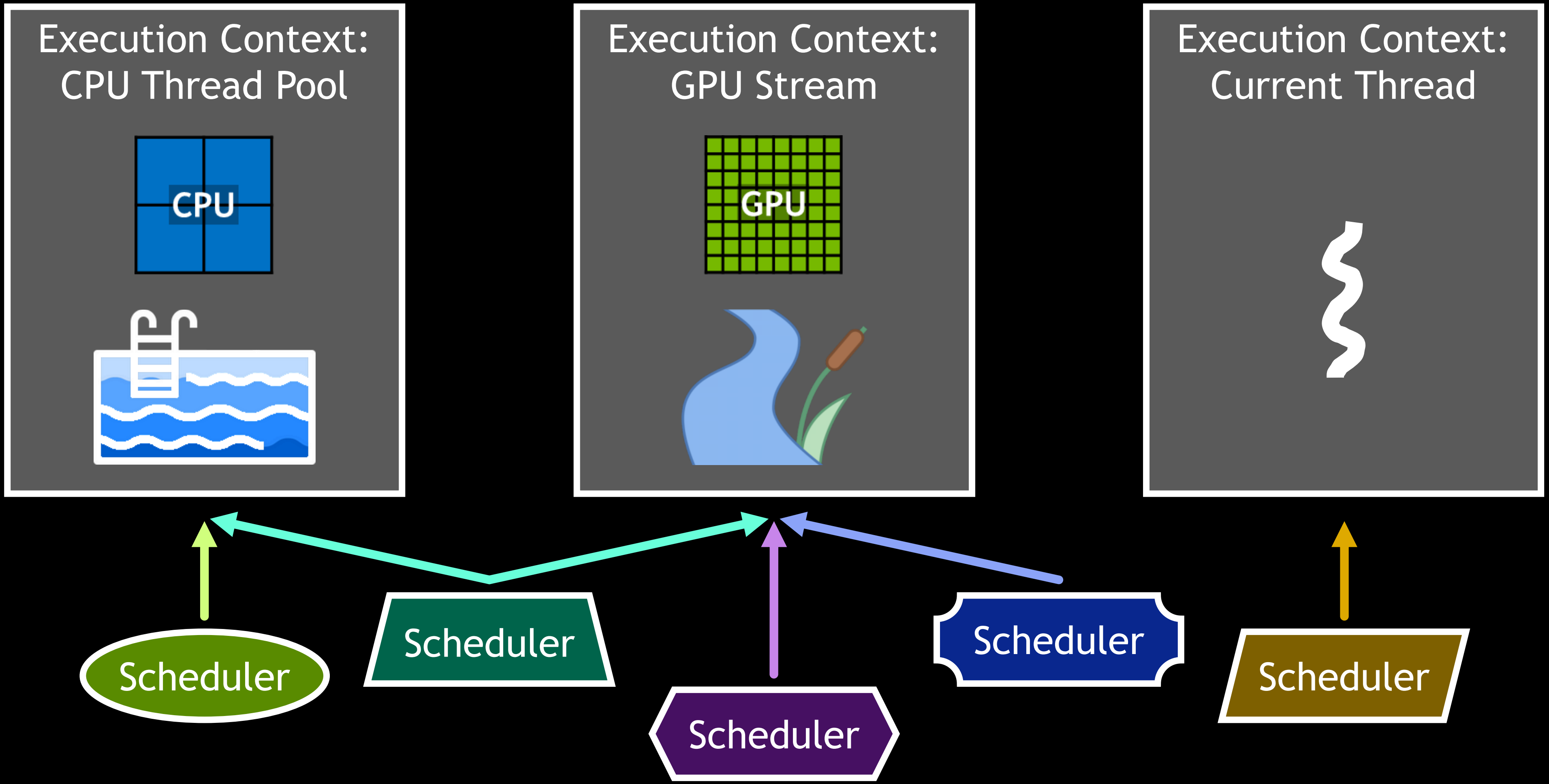


Execution Context:
Current Thread


Execution Context:
CPU Thread Pool
CPU
Execution Context:
GPU Stream
GPU
Execution Context:
Current Thread
Scheduler
Scheduler
Scheduler



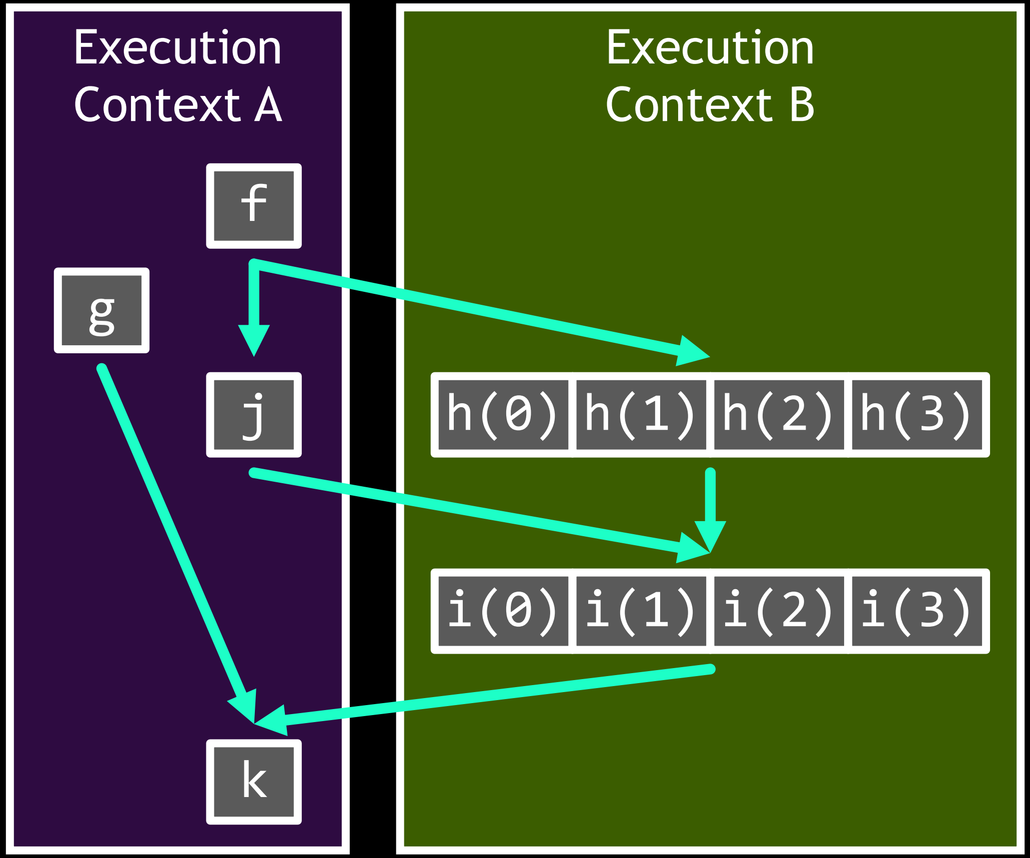
Execution Context:
CPU Thread Pool
CPU
Execution Context:
GPU Stream
GPU
Execution Context:
Current Thread
Scheduler
Scheduler
Scheduler
Scheduler


Execution Context: CPU Thread Pool
CPU
Execution Context: GPU Stream
GPU
Execution Context: Current Thread
Scheduler
Scheduler
Scheduler
Scheduler
Scheduler

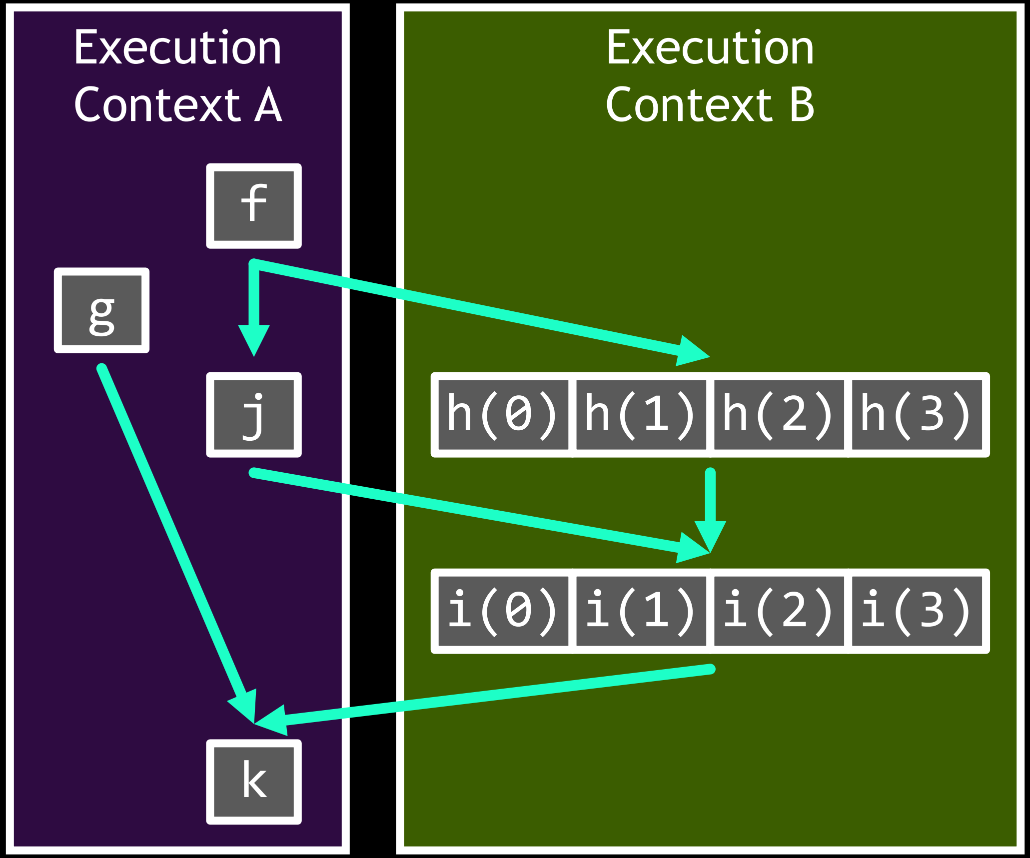
Schedulers produce senders.

- Senders represent asynchronous work.

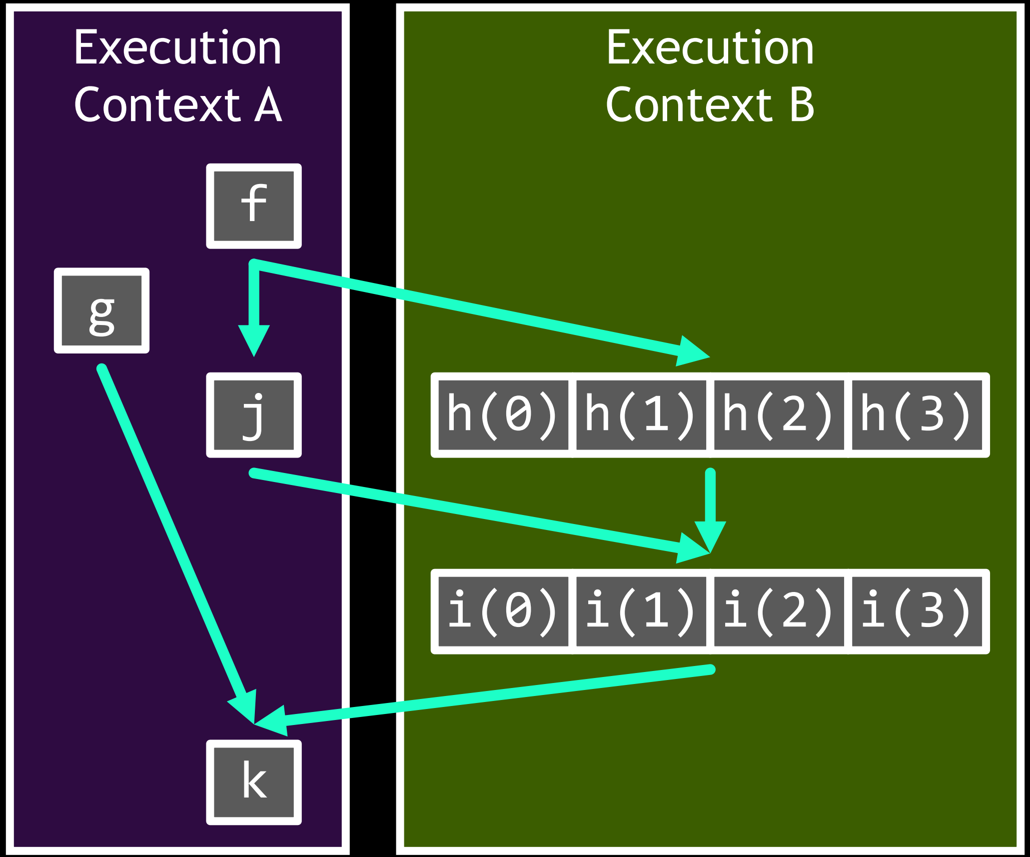
- Senders represent asynchronous work.
- Senders form the nodes of a task graph.



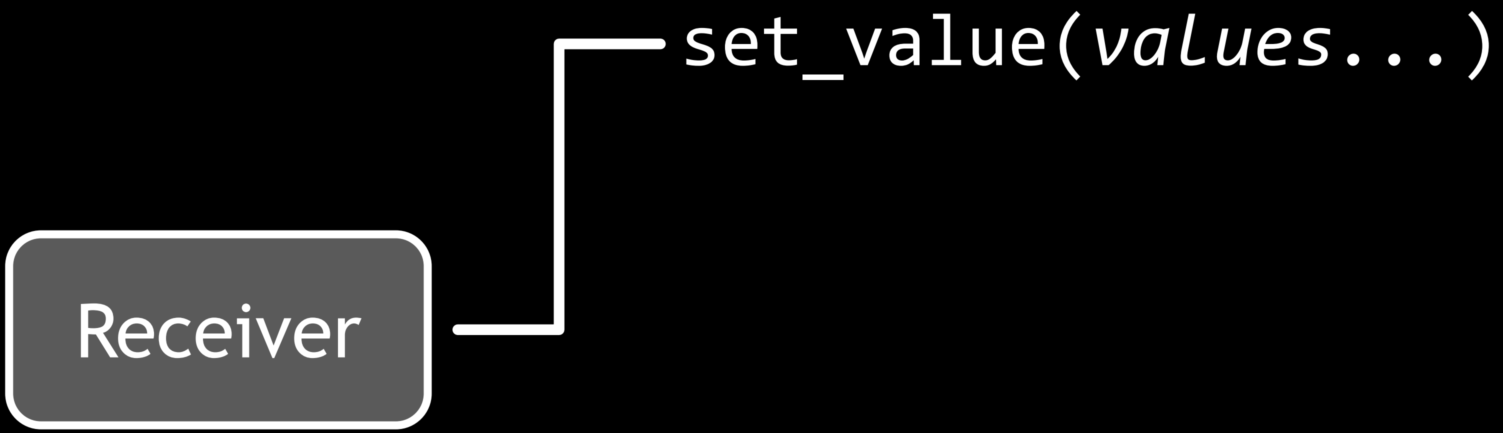
- Senders represent asynchronous work.
- Senders form the nodes of a task graph.
- Senders are lazy.

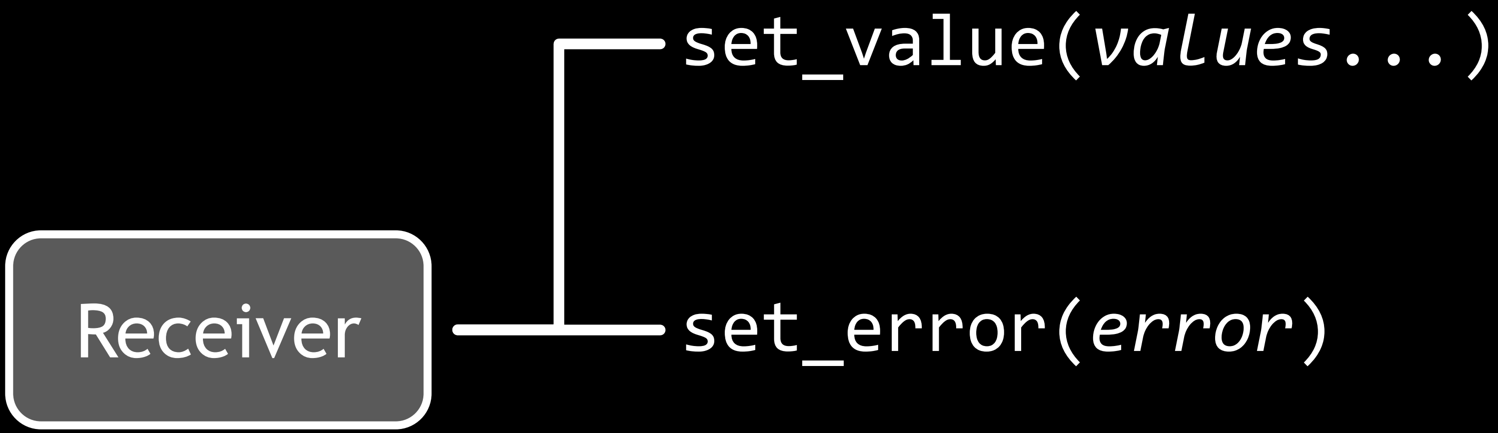


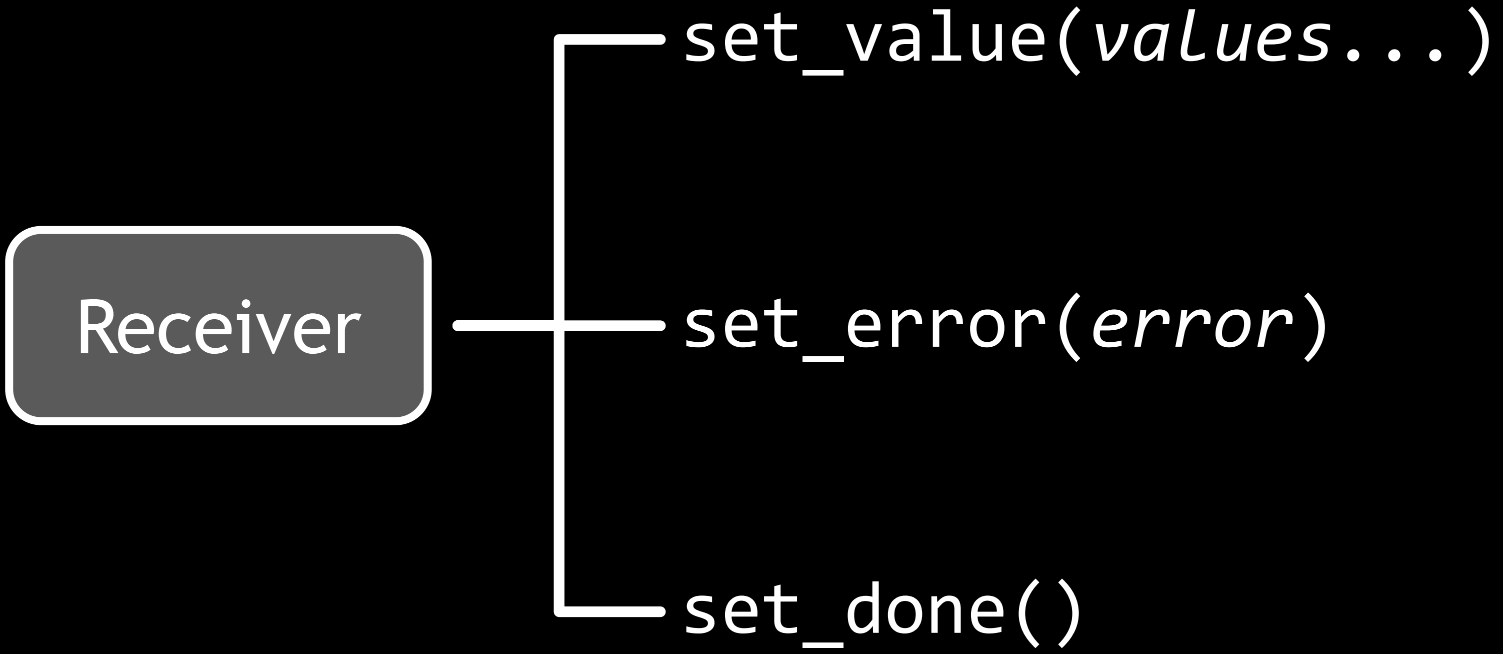
- Senders represent asynchronous work.
- Senders form the nodes of a task graph.
- Senders are lazy.
- When a sender's work completes, it sends a signal to the receivers attached to it.



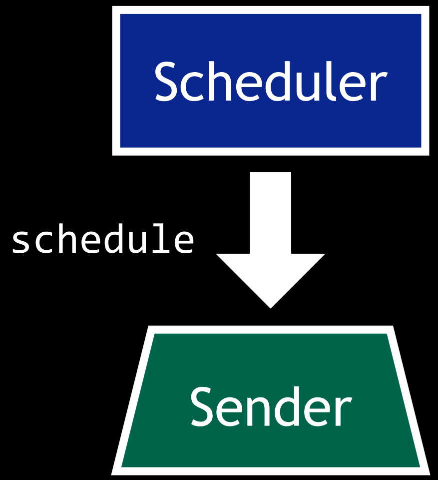
Receiver

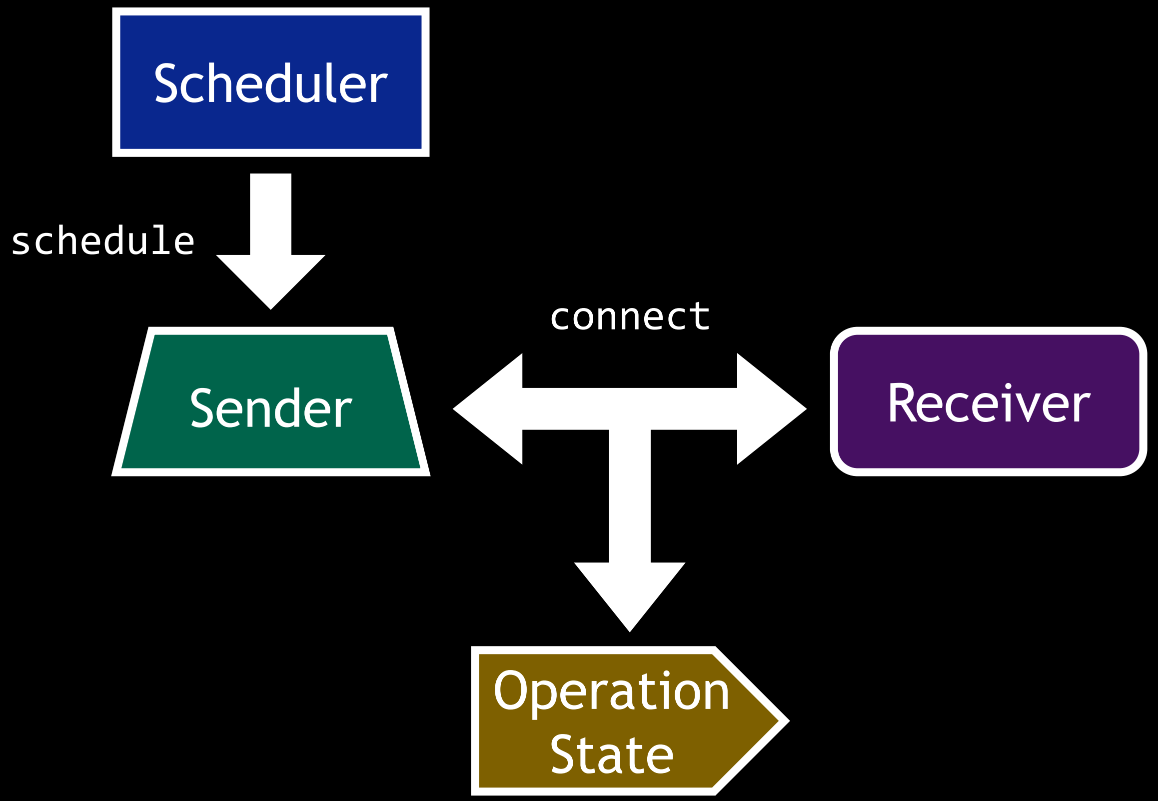

set_value(values...)
Receiver

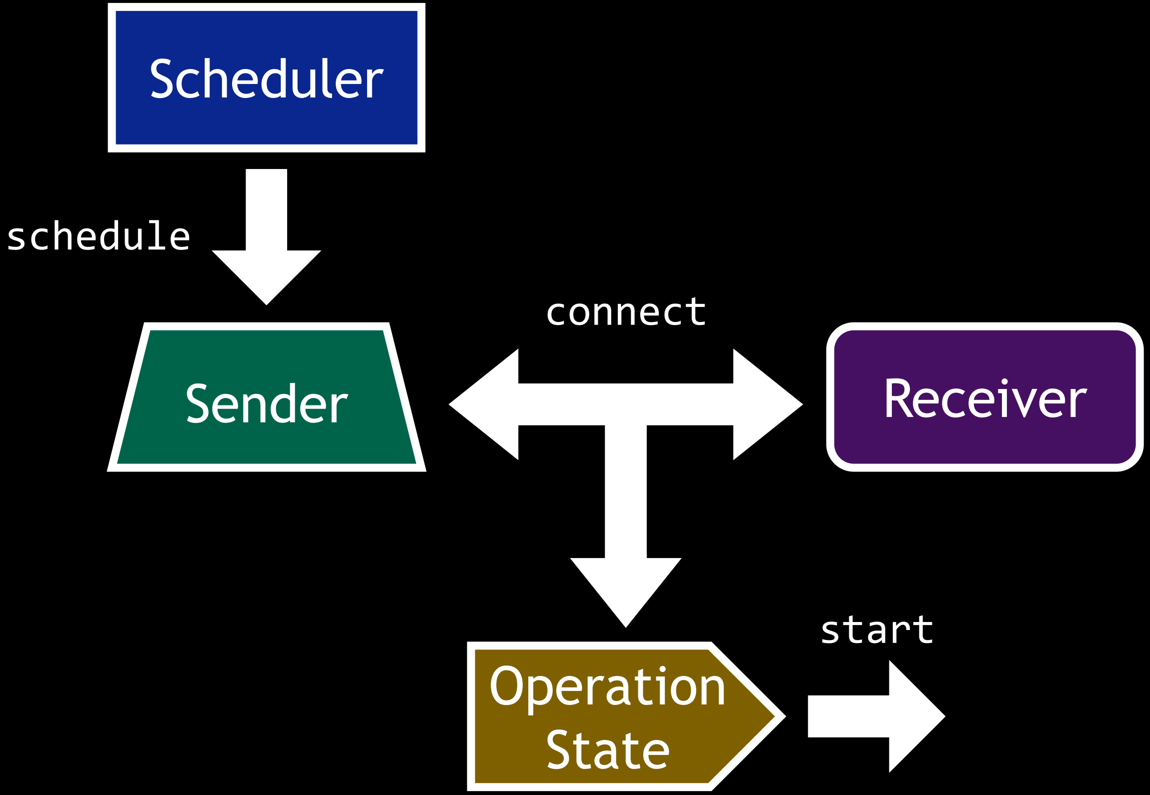

Receiver
set_value(values...)
set_error(error)

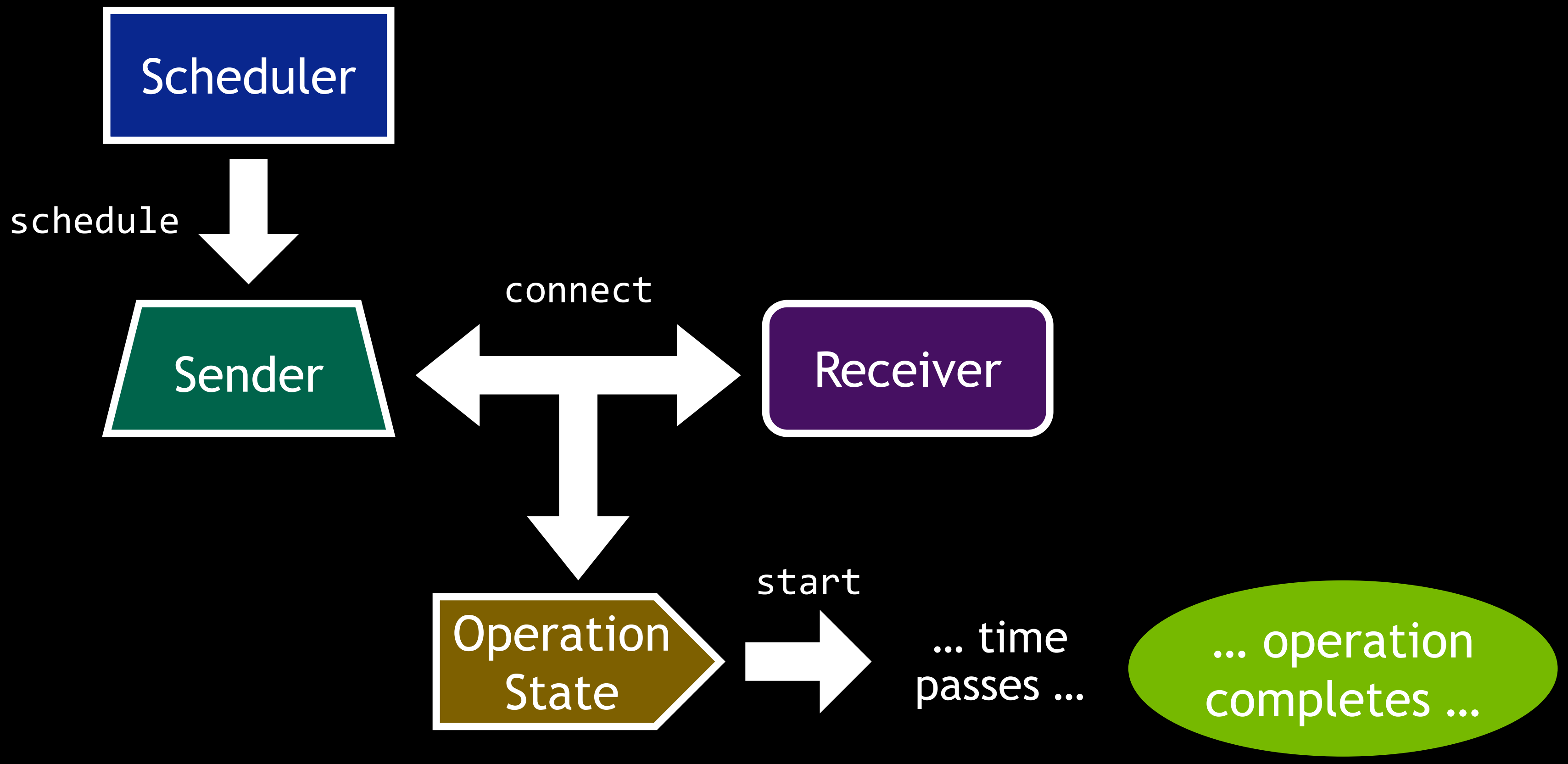


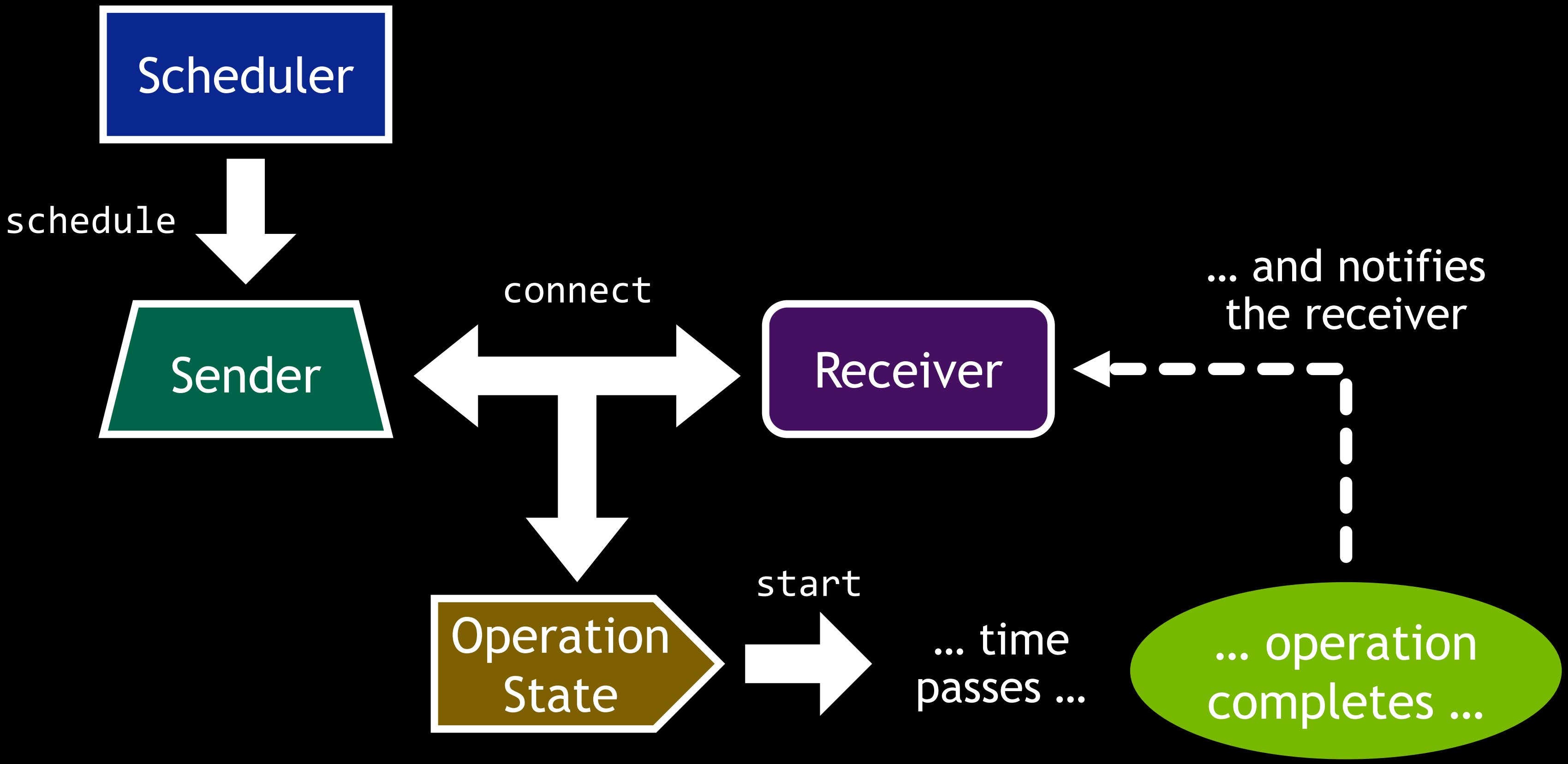
brycelelbach@gmail.com |


Scheduler
schedule
Sender


Scheduler
schedule
Sender
connect
Receiver
Operation
State


Scheduler
schedule
Sender
connect
Receiver
Operation State
start


Scheduler
schedule
Sender
connect
Receiver
Operation State
start
... time passes ...
... operation completes ...


Scheduler
schedule
Sender
connect
Receiver
... and notifies
the receiver
Operation
State
start
... time
passes ...
... operation
completes ...

```
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); }
                    );

ex::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); });

ex::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); });

ex::sync_wait(s);
```

Sender Adaptor	Semantics Of Returned Sender

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

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

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

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

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

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

Sender Factories	Semantics Of Returned Sender

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

Sender Factories	Semantics Of Returned Sender
schedule(scheduler auto sch)	Completes on sch.

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

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

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

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

Sender Consumers	Semantics

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

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

Sender Consumers	Semantics
sync_wait(sender auto snd) -> *values-sent-by-sender*	Block until snd completes and return or throw whatever it sent.

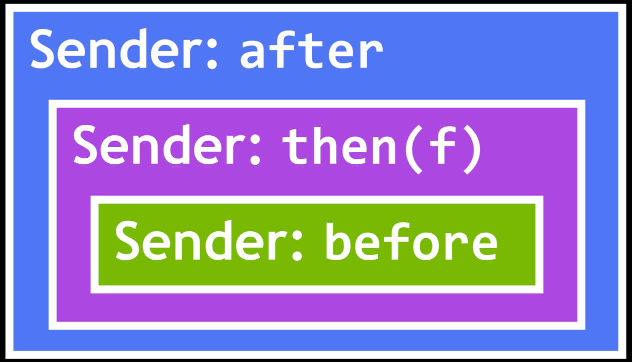
```
before | then(f) | after;
```

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);
```

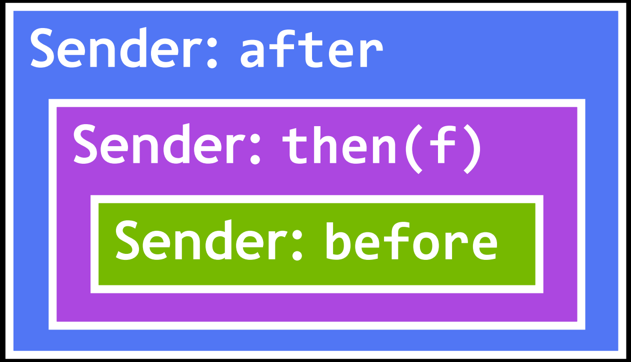
before | then(f) | after;

```
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);
                …
```

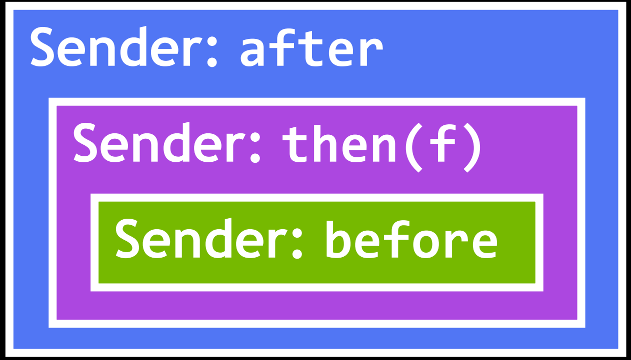


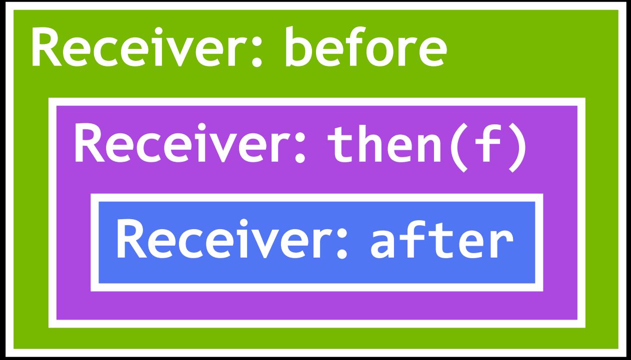
before | then(f) | after;

```
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 auto
async::inclusive_scan = [] (…) -> ex::sender auto {
  …
}
```

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

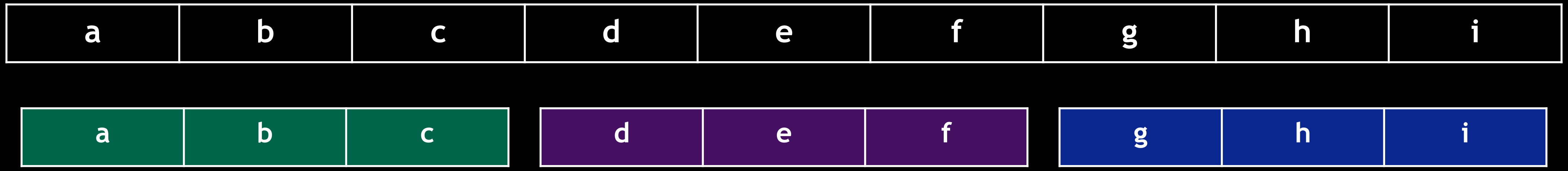
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                   …
                 })
  …
}
```

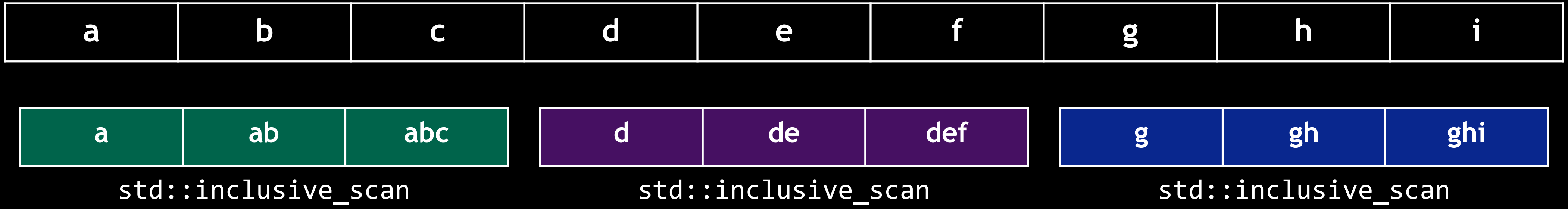
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count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

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

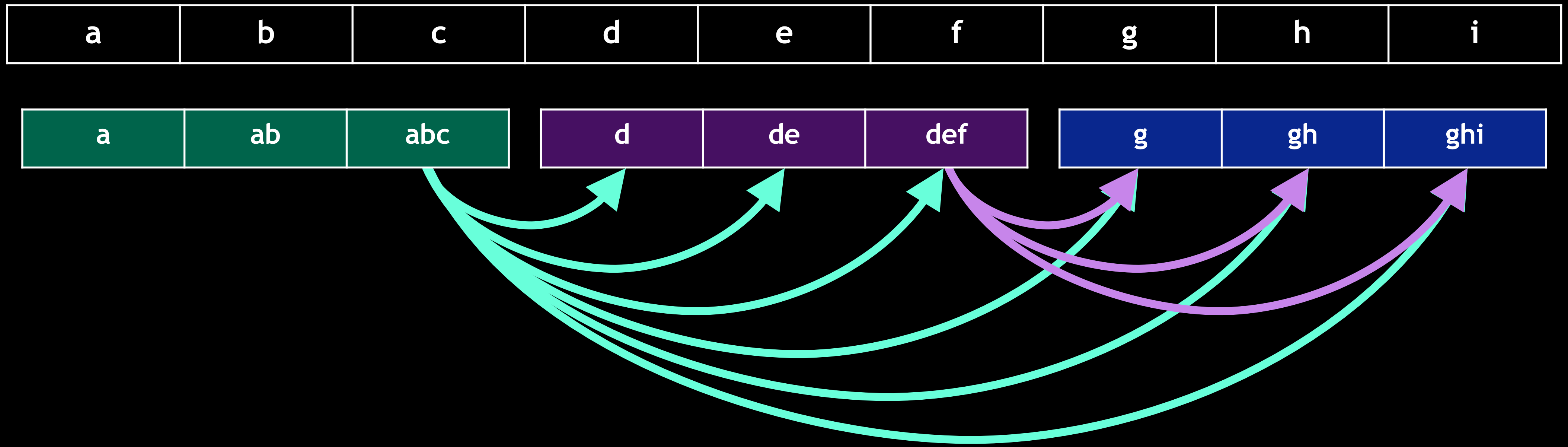
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count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

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | 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 b c
d e f
g h i

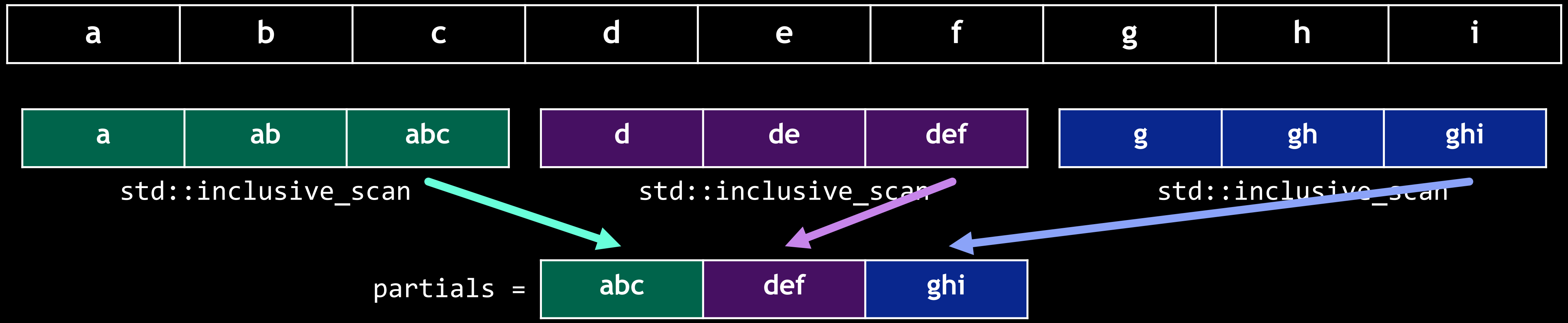

a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan

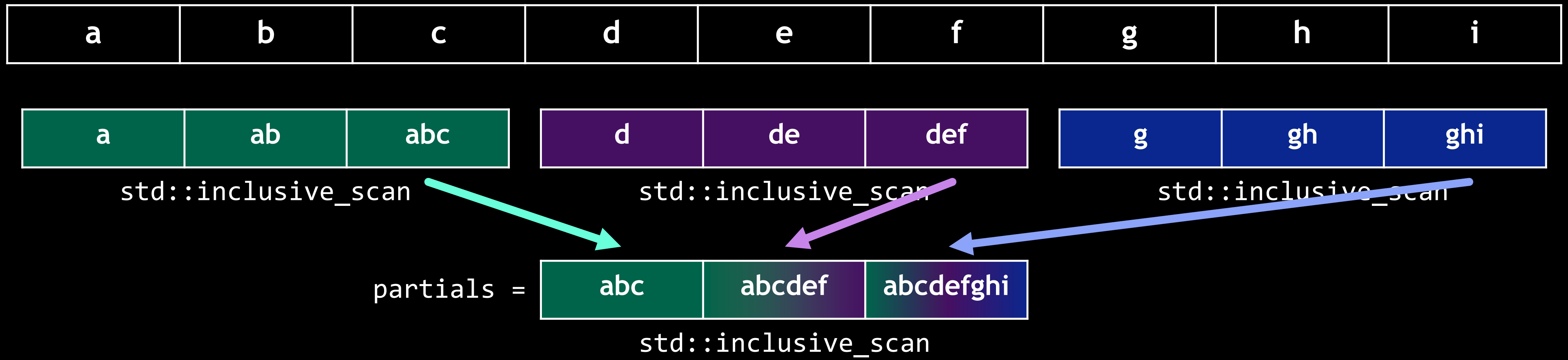
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                  std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count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

                })
  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi

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

                   })
  …
}
```

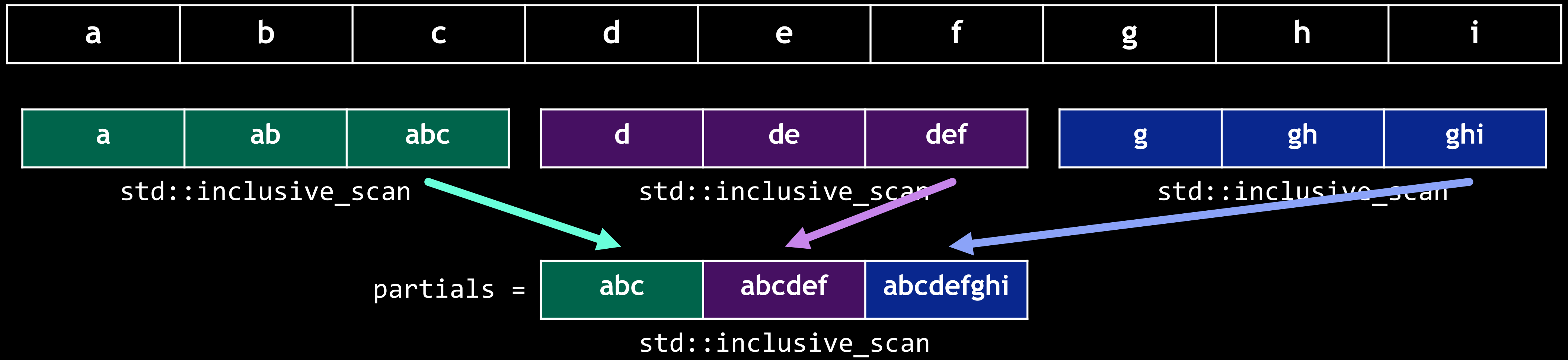
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                     std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count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


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
def
ghi


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
abcdef
abcdefghi
std::inclusive_scan

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | 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(partials), end(partials), begin(partials));
                    …
                  })
  …
}
```

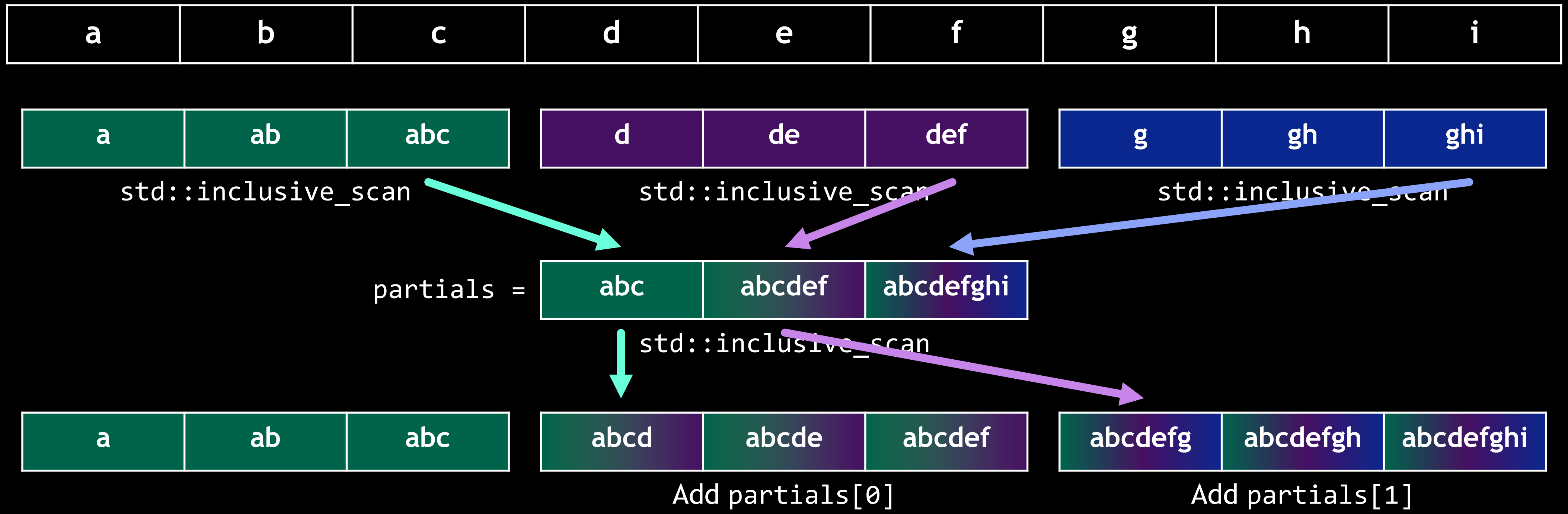
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                partials[0] = init;
                return send_values(input, std::move(partials));
              })
       | 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(partials), end(partials), begin(partials));
                return send_values(input, std::move(partials));
              })
  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
abcdef
abcdefghi
std::inclusive_scan

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

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | 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(partials), end(partials), begin(partials));
                    return send_values(input, std::move(partials));
                  })
       | 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 auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                   std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                   partials[0] = init;
                   return send_values(input, std::move(partials));
                 })
       | 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(partials), end(partials), begin(partials));
                   return send_values(input, std::move(partials));
                 })
       | 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; });
                 })
  …
}
```



a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
abcdef
abcdefghi
std::inclusive_scan
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
Add partials[0]
Add partials[1]

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

Standard Algorithms

Serial (C++98)

```
std::vector<T> x{…};

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

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

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

Parallel (C++17)

```
std::vector<T> x{…};

std::transform(
 ex::par_unseq,
 begin(x), end(x), begin(x)
 f);
std::transform(
 ex::par_unseq,
 begin(x), end(x), begin(x)
 g);
std::transform(
 ex::par_unseq,
 begin(x), end(x), begin(x)
 h);
```

Asynchronous

```
std::vector<T> x(…);

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

ex::sync_wait(s);
```

Planned for C++26 and coming soon to NVC++!

<table>
<tr><th>Async Algorithms</th><th>Returns</th><th>Semantics</th></tr>
<tr><td>transform(out, rngs..., f)
just(out, rngs...) | transform(f)</td><td>sender_of<range></td><td>out[i] = f(out[i], rngs[i]…)</td></tr>
<tr><td>reduce(rng, init, f)
just(rng, init) | reduce(f)</td><td>sender_of<T></td><td>f(f(f(init, rng[0]), rng[1]), …)</td></tr>
<tr><td>find_if(rng, f)
just(rng) | find_if(f)</td><td>sender_of<iterator></td><td>Returns the first element
for which f(rng[i]) == true</td></tr>
<tr><td>copy(from, to)
just(from, to) | copy</td><td>sender_of<range></td><td>to[i] = from[i]</td></tr>
<tr><td>copy_if(from, to, f)
just(from, to) | copy_if(f)</td><td>sender_of<range></td><td>to[i] = from[i]
for all i for which f(rng[i]) == true</td></tr>
<tr><td>*_scan(rng, init, f)
just(rng, init) | *_scan(f)</td><td>sender_of<range></td><td>rng[i] = rng[0] + … + rng[i]</td></tr>
<tr><td>sort(rng, f)
just(rng) | sort(f)</td><td>sender_of<range></td><td>Ensures f(rng[i+n], rng[i]) == false</td></tr>
<tr><td>unique(rng, f)
just(rng) | unique(f)</td><td>sender_of<range></td><td>Eliminates all elements
for which f(rng[i+1], rng[i])</td></tr>
<tr><td>partition(rng, f)
just(rng) | unique(f)</td><td>sender_of<range></td><td>Reorders such that all elements for which
f(rng[i]) == true precede all others</td></tr>
</table>

```
stdr::transform(rng, begin(rng), f);
stdr::transform(rng, begin(rng), g);
stdr::transform(rng, begin(rng), h);
```

```
stdr::transform(rng, begin(rng), f);
stdr::transform(rng, begin(rng), g);
stdr::transform(rng, begin(rng), h);
```

```
stdr::transform(ex::par_unseq,
  rng, begin(rng), f);
stdr::transform(ex::par_unseq,
  rng, begin(rng), g);
stdr::transform(ex::par_unseq,
  rng, begin(rng), h);
```

```
ex::sender_of<stdr::range> auto
async::transform(ex::sender_of<stdr::range> auto rng, std::invocable auto c);
```

```
stdr::transform(rng, begin(rng), f);
stdr::transform(rng, begin(rng), g);
stdr::transform(rng, begin(rng), h);
```

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

```
auto m = stdr::max_element(rng);

stdr::transform(
  rng, stdv::repeat(*m), begin(rng),
  std::divides);
```

```
ex::sender_of<stdr::range, stdr::range, …> auto
async::transform(ex::sender_of<stdr::range, stdr::range, …> auto rngs,
                 std::invocable auto f);

ex::sender_of<std::forward_iterator> auto
async::max_element(ex::sender_of<stdr::range> auto rng);
```

```
auto m = stdr::max_element(rng);

stdr::transform(
  rng, stdv::repeat(*m), begin(rng),
  std::divides);
```

```
ex::sender auto snd =
    ex::transfer_just(sch, rng)
  | ex::with(async::max_element)
  | ex::let_value(
      [] (auto r, auto m) {
        return ex::just(
          r, stdv::repeat(*m));
      })
  | async::transform(std::divides);
```

```
ex::sender_of<stdr::range, stdr::range, …> auto
async::transform(ex::sender_of<stdr::range, stdr::range, …> auto rngs,
                 std::invocable auto f);

ex::sender_of<std::forward_iterator> auto
async::max_element(ex::sender_of<stdr::range> auto rng);
```

```
auto m = stdr::max_element(rng);

stdr::transform(
  rng, stdv::repeat(*m), begin(rng),
  std::divides);
```

```
ex::sender auto snd =
    ex::transfer_just(sch, rng)
  | ex::with(async::max_element)
  | ex::let_value(
      [] (auto r, auto m) {
        return ex::just(
          r, stdv::repeat(*m));
      })
  | async::transform(std::divides);
```

```
ex::sender_of<stdr::range, stdr::range, …> auto
async::transform(ex::sender_of<stdr::range, stdr::range, …> auto rngs,
                 std::invocable auto f);

ex::sender_of<std::forward_iterator> auto
async::max_element(ex::sender_of<stdr::range> auto rng);
```

```
auto m = stdr::max_element(rng);

stdr::transform(
  rng, stdv::repeat(*m), begin(rng),
  std::divides);
```

```
ex::sender auto snd =
    ex::transfer_just(sch, rng)
  | ex::with(async::max_element)
  | ex::let_value(
      [] (auto r, auto m) {
        return ex::just(
          r, stdv::repeat(*m));
      })
  | async::transform(std::divides);
```

```
ex::sender_of<stdr::range, stdr::range, …> auto
async::transform(ex::sender_of<stdr::range, stdr::range, …> auto rngs,
                 std::invocable auto f);

ex::sender_of<std::forward_iterator> auto
async::max_element(ex::sender_of<stdr::range> auto rng);
```

```
auto m = stdr::max_element(rng);

stdr::transform(
  rng, stdv::repeat(*m), begin(rng),
  std::divides);
```

```
ex::sender auto snd =
    ex::transfer_just(sch, rng)
  | ex::with(async::max_element)
  | ex::let_value(
      [] (auto r, auto m) {
        return ex::just(
          r, stdv::repeat(*m));
      })
  | async::transform(std::divides);
```

Standard Senders

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

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

Synchronous

```
std::for_each(stdex::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[j, i] = A[i, j];
  });

std::matrix_product(stdex::par_unseq,
  A, B, B);
```

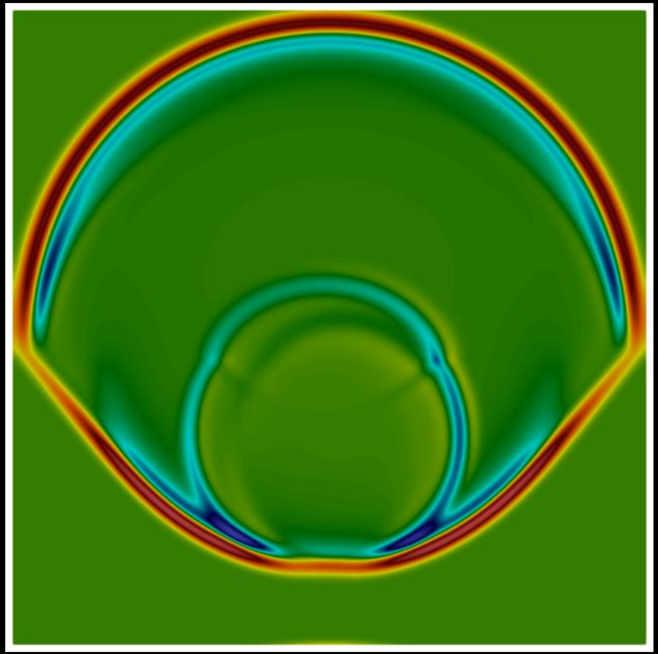
Asynchronous

```
auto s = stdex::just_on(sch, v)
  | stdex::bulk(N,
      [=] (auto idx) {
        auto [i, j] = idx;
        B[j, i] = A[i, j];
      })
  | async::matrix_product(B, B);
```

Planned for C++26 and available at github.com/nvidia/stdexec!

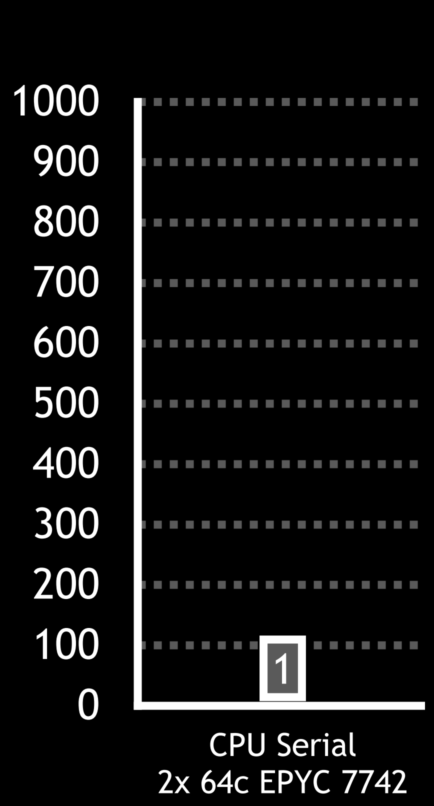
Maxwell's Equations

```
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

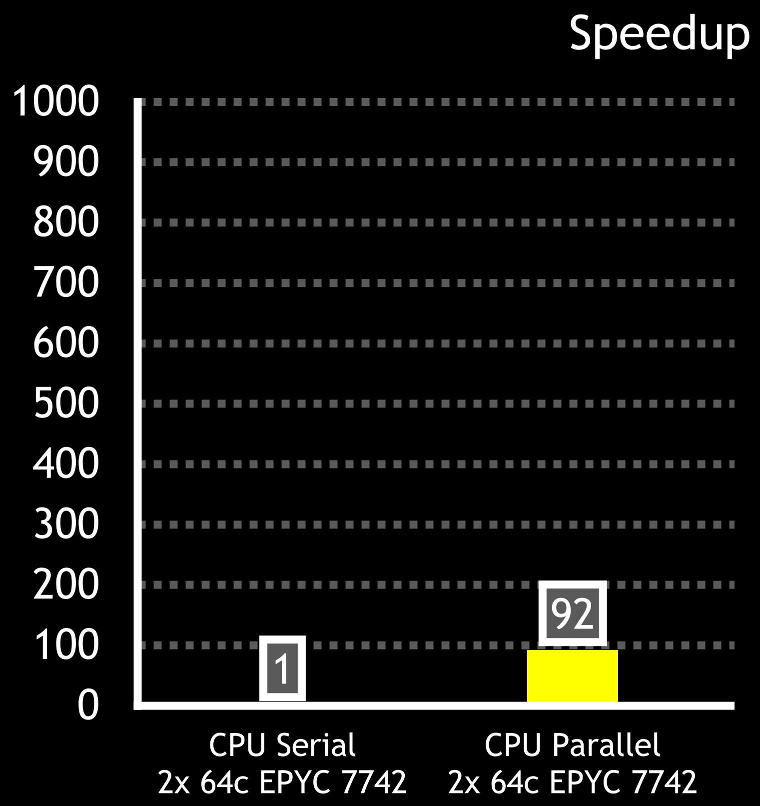
Change one line of code and scale from a single CPU thread…




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

Maxwell's Equations

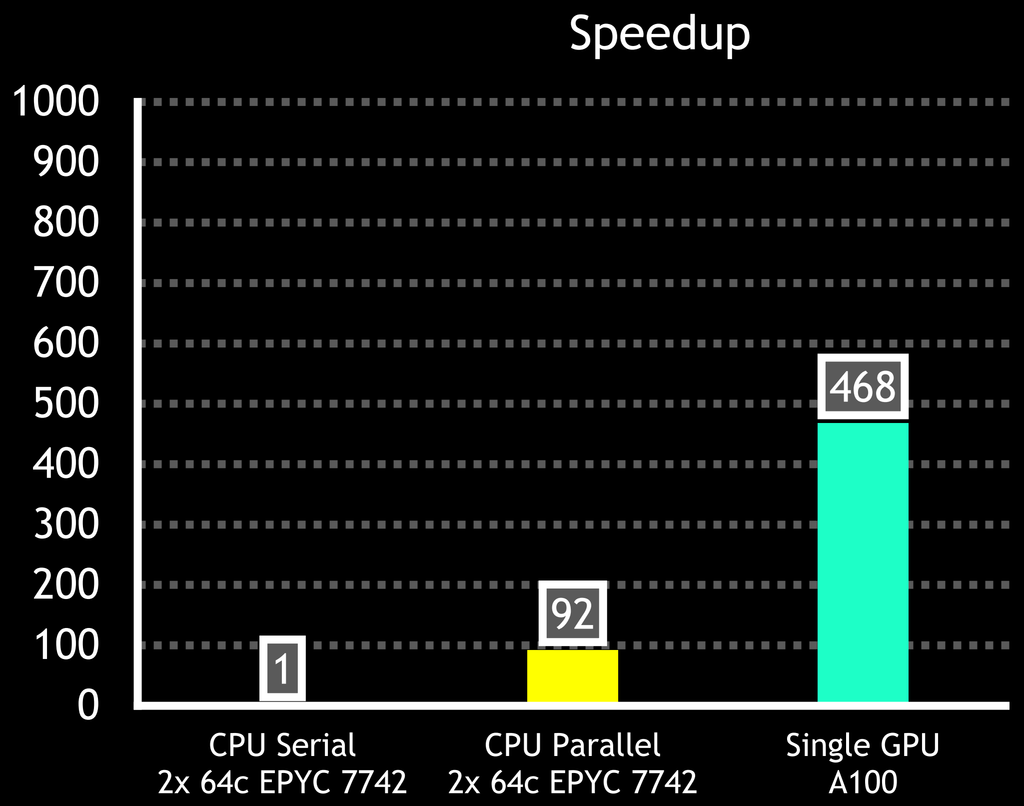
Change one line of code and scale from a single CPU thread up to multiple CPU threads…




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

Maxwell's Equations

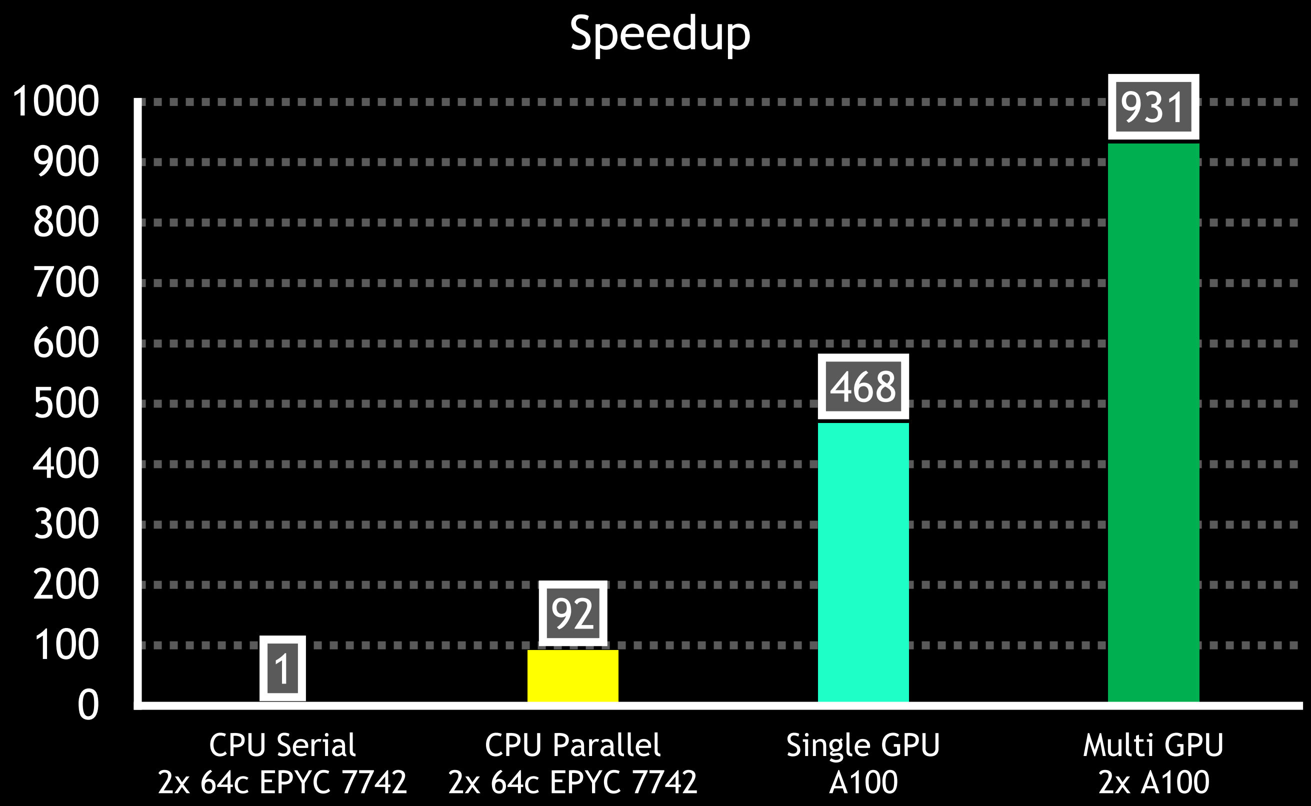
Change one line of code and scale from a single CPU thread up to a GPU…




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

Maxwell's Equations

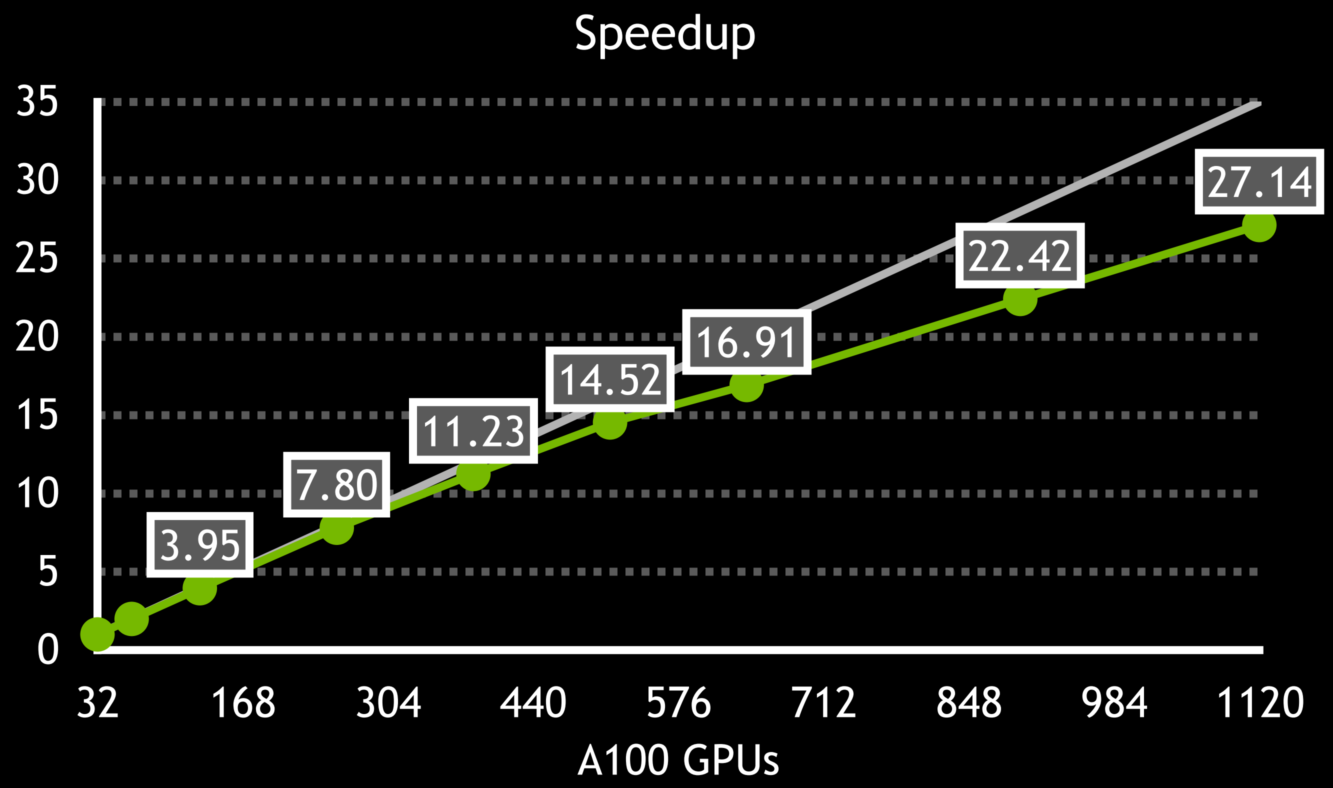
Change one line of code and scale from a single CPU thread up to multiple GPUs…




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

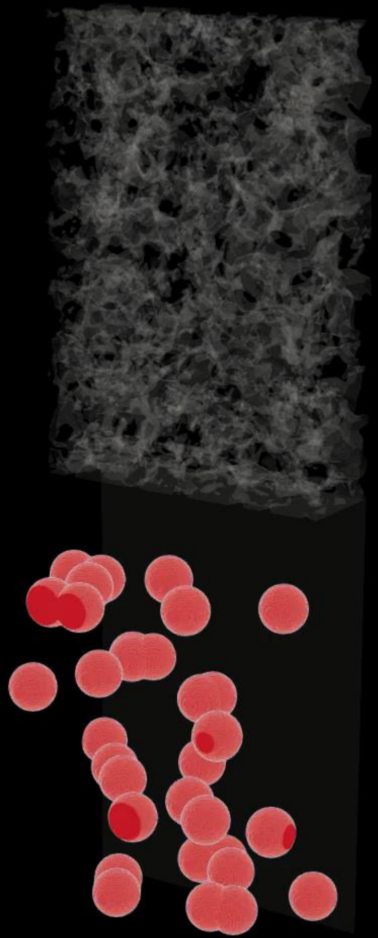
Maxwell's Equations

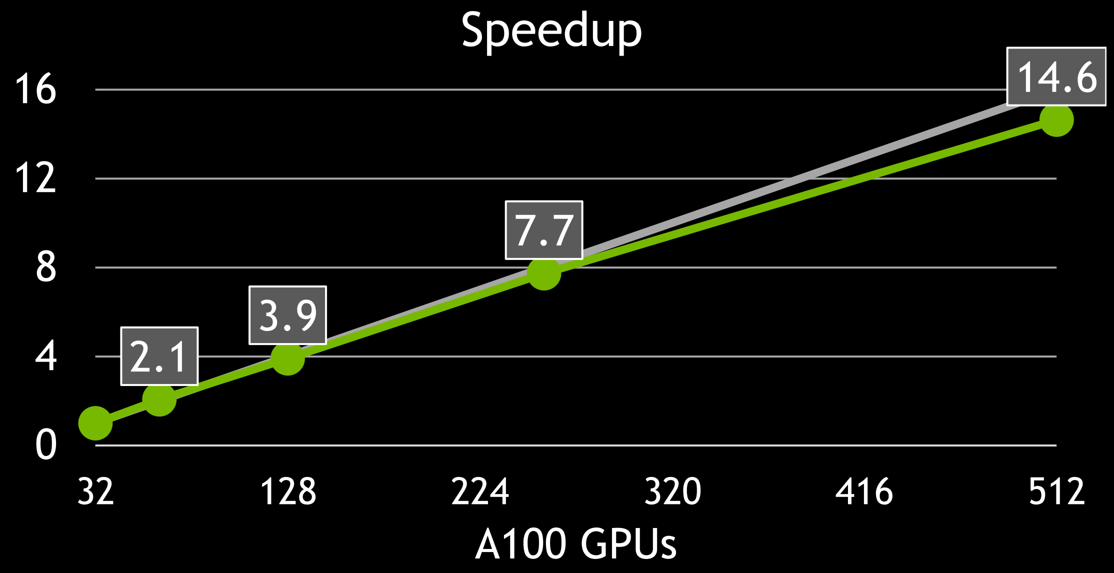
Change one line of code and scale from a single CPU thread up to a cluster of GPUs!




```
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.

Jonas Latt (University of Geneva), Christian Huber (Brown University)
Georgy Evtushenko (NVIDIA), Gonzalo Brito (NVIDIA)

Standard Senders

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

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

Synchronous

```
std::for_each(stdex::par_unseq,
  begin(v), end(v),
  [=] (auto idx) {
    auto [i, j] = idx;
    B[j, i] = A[i, j];
  });

std::matrix_product(stdex::par_unseq,
  A, B, B);
```

Asynchronous

```
auto s = stdex::just_on(sch, v)
  | stdex::bulk(N,
      [=] (auto idx) {
        auto [i, j] = idx;
        B[j, i] = A[i, j];
      })
  | async::matrix_product(B, B);
```

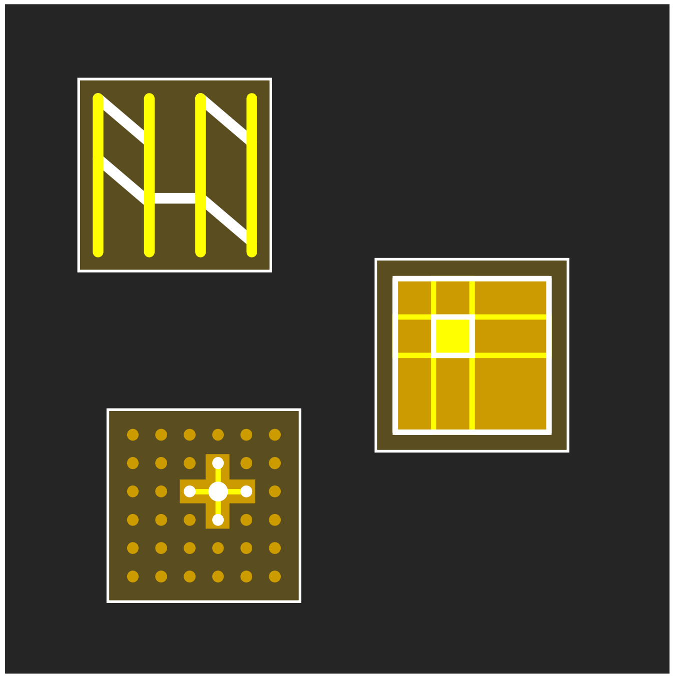
Planned for C++26 and available at github.com/nvidia/stdexec!

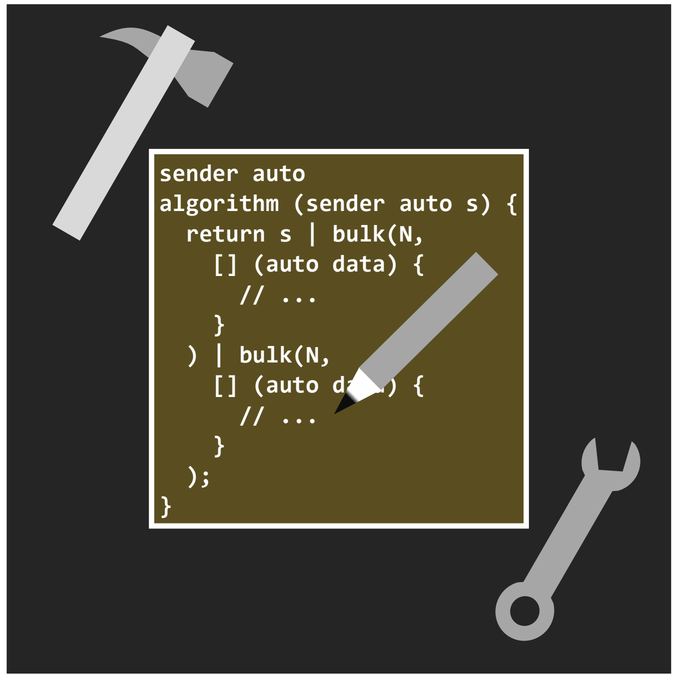
Pillars of C++ Standard Parallelism

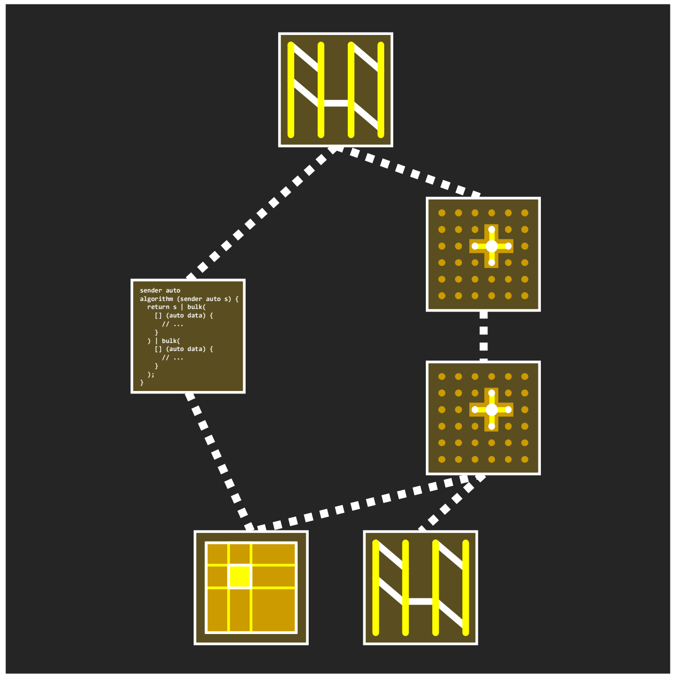
Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries

Tools to Write Your Own Parallel Algorithms that Run Anywhere

Mechanisms for Composing Parallel Invocations into Task Graphs







Today

With Senders

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 T,

                                                          >

class std::mdspan;
```

```
template <class T,
          class Extents,


                                                                  >
class std::mdspan;
```

```
template <class T,
          class Extents,
          class LayoutPolicy = std::layout_right,
                                                          >
class std::mdspan;
```

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

```cpp
template <class T,
          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 T,
          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 T,
          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}

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

NVIDIA

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
```

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

```
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
```

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
```

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 $(i, j, k, \ldots)$ to a data location.

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

Anyone can define a layout.

Layouts map $(i, j, k, \ldots)$ 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 $(i, j, k, \ldots)$ 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 $(i, j, k, \ldots)$ 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, O};

std::mdspan B{output, 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
  });
```


NVIDIA

```
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];
  });
```

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

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

Slice Specifier	Argument
Single	`index`

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

Slice Specifier	Argument
Single	`index`
Interval	`std::tuple{first, last}`

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

Slice Specifier	Argument
Single	`index`
Interval	`std::tuple{first, last}`
Strided Interval	`std::strided_slice{offset, length, stride}`

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

Slice Specifier	Argument
Single	`index`
Interval	`std::tuple{first, last}`
Strided Interval	`std::strided_slice{offset, length, stride}`
All	`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 m0{data, 64, 128, 32};

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()    == 1
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()     == 1
m3.extent(0) == 4
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()     == 1
m3.extent(0) == 4
m3[i]         == m0[i * 2 + 7]
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()     == 1
m3.extent(0) == 4
m3[i]         == m0[i * 2 + 7]
m3[0]         == m0[7]
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()     == 1
m3.extent(0)  == 4
m3[i]         == m0[i * 2 + 7]
m3[0]         == m0[7]
m3[1]         == m0[9]
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()      == 1
m3.extent(0)   == 4
m3[i]          == m0[i * 2 + 7]
m3[0]          == m0[7]
m3[1]          == m0[9]
m3[2]          == m0[11]
```

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

auto m3 = std::submdspan(m0,
            std::strided_slice{7, 8, 2},
            0,
            0
          );

m3.rank()     == 1
m3.extent(0) == 4
m3[i]         == m0[i * 2 + 7]
m3[0]         == m0[7]
m3[1]         == m0[9]
m3[2]         == m0[11]
m3[3]         == m0[13]
```

```
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 = 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)};

    …
  });
```

```
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];
  });
```

`mdspan` doesn’t provide ranges and iterators that enumerate its elements.

`mdspan` doesn't provide ranges and iterators that enumerate its elements.

Why not?

`mdspan` doesn't provide ranges and iterators that enumerate its elements.

Why not?

Performance.

Why do we want ranges and iterators?

- Parameterization.
- Composability.

They enable generic programming.

The Space Protocol

```
template <std::int64_t N, typename Space, typename... Outer>
stdr::range auto mdrange(Space&& space, std::tuple<Outer...>&& outer);

template <typename Space>
constexpr std::int64_t mdrank;
```

The Space Protocol

```
template <std::int64_t N, typename Space, typename... Outer>
stdr::range auto mdrange(Space&& space, std::tuple<Outer...>&& outer);

template <typename Space>
constexpr std::int64_t mdrank;
```

The Space Protocol

```
template <std::int64_t N, typename Space, typename... Outer>
stdr::range auto mdrange(Space&& space, std::tuple<Outer...>&& outer);

template <typename Space>
constexpr std::int64_t mdrank;
```

The Space Protocol

```
template <std::int64_t N, typename Space, typename... Outer>
stdr::range auto mdrange(Space&& space, std::tuple<Outer...>&& outer);

template <typename Space>
constexpr std::int64_t mdrank;
```

The Space Protocol

```
template <std::int64_t N, typename Space, typename... Outer>
stdr::range auto mdrange(Space&& space, std::tuple<Outer...>&& outer);

template <typename Space>
constexpr std::int64_t mdrank;
```

The function:

```
template <typename Space, typename UnaryFunction>
void for_each(Space&& space, UnaryFunction&& f);
```

is equivalent to:

```
constexpr auto N = mdrank<Space>;
for (auto k : mdrange<N - 1>(space))
  for (auto jk : mdrange<N – 2>(space, k))
    …
      for (auto ijk = mdrange<0>(space, …))
        f(ijk);
```

```
std::mdspan A{…, N, M};

space auto elements = A.elements();
space auto indices  = A.indices();

// Traditional ranges & iterators for elementwise access.
// Multidimensional indices are NOT exposed from these.
stdr::random_access_range auto    range = A;
stdr::random_access_iterator auto first = begin(a);
stdr::random_access_iterator auto last  = end(a);
```

```cpp
std::mdspan A{…, N, M, O};

// Just the main diagonal.
A.indices() | stdv::filter([] (auto [i, j, k]) { return i == j && j == k; });
```

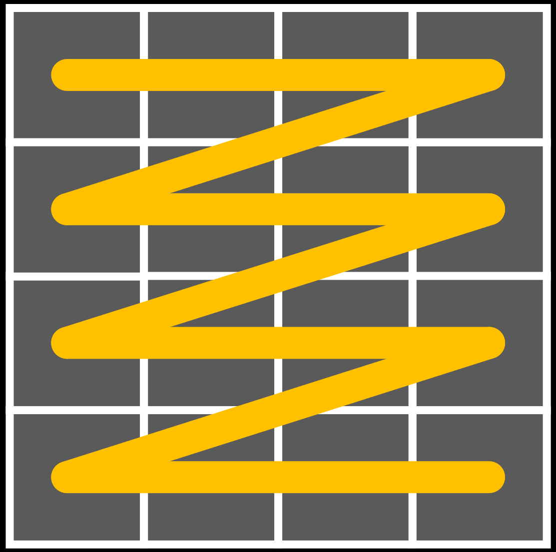
```
std::mdspan A{…, N, M, O};

// Just the main diagonal.
A.indices() | stdv::filter([] (auto [i, j, k]) { return i == j && j == k; });

// Just [i, 0, 0].
A.indices() | on_extent<1>(stdv::filter([] (auto [j, k]) { return j == 0; }))
            | on_extent<2>(stdv::filter([] (auto [k]) { return k == 0; }));
```

```
std::mdspan A{…, N, M, O};

// Just the main diagonal.
A.indices() | stdv::filter([] (auto [i, j, k]) { return i == j && j == k; });

// Just [i, 0, 0].
A.indices() | on_extent<1>(stdv::filter([] (auto [j, k]) { return j == 0; }))
            | on_extent<2>(stdv::filter([] (auto [k]) { return k == 0; }));

// Just interior points.
A.indices() | on_extent<0>(stdv::drop(1) | stdv::take(A.extent(0) - 2))
            | on_extent<1>(stdv::drop(1) | stdv::take(A.extent(1) - 2))
            | on_extent<2>(stdv::drop(1) | stdv::take(A.extent(2) - 2));
```

```
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];
  });
```

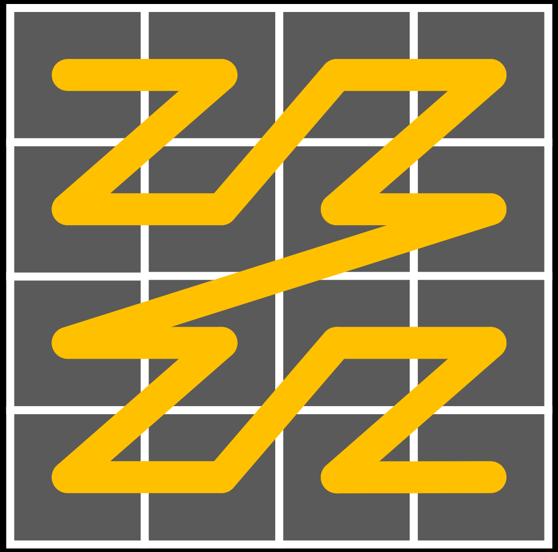
```
std::mdspan A{…, N, M};
```

Row-Major
AKA Right



Column-Major
AKA Left

Morton Order
AKA Z-Order Curve



```
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())
  | async::for_each(
      [=] (auto [i, j]) {
        B[j, i] = A[i, j];
      });
```

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

stdr::for_each(ex::par_unseq,
  A.indices(),
  [=] (auto [i, j, k]) {
    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,  N, M, O};
std::mdspan B{output, N, M, O};

// Just interior points.
stdr::for_each(ex::par_unseq,
  A.indices() | on_extent<2>(stdv::drop(1) | stdv::take(A.extent(2)-2))
              | on_extent<1>(stdv::drop(1) | stdv::take(A.extent(1)-2))
              | on_extent<0>(stdv::drop(1) | stdv::take(A.extent(0)-2)),
  [=] (auto [i, j, k]) {
    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,  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];
  });
```

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

auto outer = A.indices() | on_extent<1>(stdv::stride(T))
                         | on_extent<0>(stdv::stride(T));

stdr::for_each(ex::par_unseq, outer,
  [=] (auto [x, y]) {
    for (auto [i, j] : std::extents{{T * x, std::min(T * (x + 1), N)},
                                    {T * y, std::min(T * (y + 1), M)}})
      B[j, i] = A[i, j];
  });
```

Standard Multidimensional Spans

```
template <class T, class Extents, class LayoutPolicy = …, class AccessorPolicy = …>
class std::mdspan;
```

```
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}

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

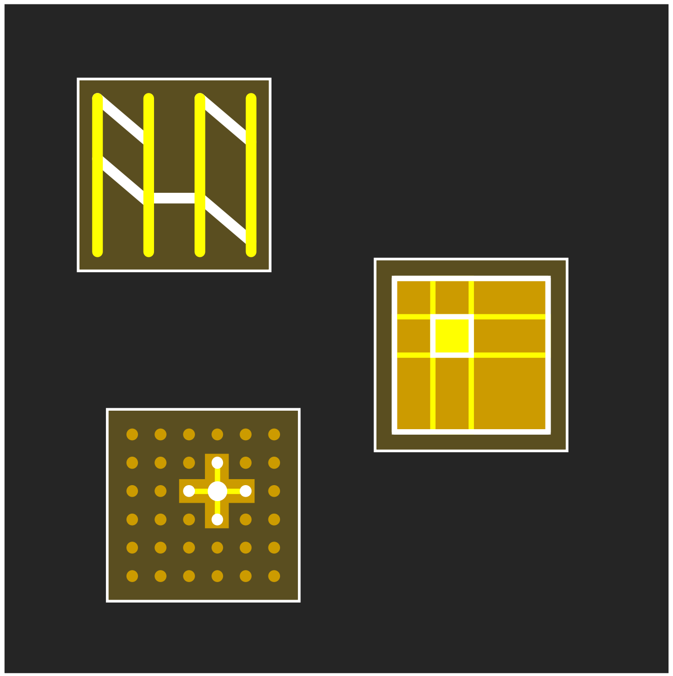
```
mdspan B{data, layout_left::mapping{N, M}};

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}

Available since C++23 and NVC++ 22.7!

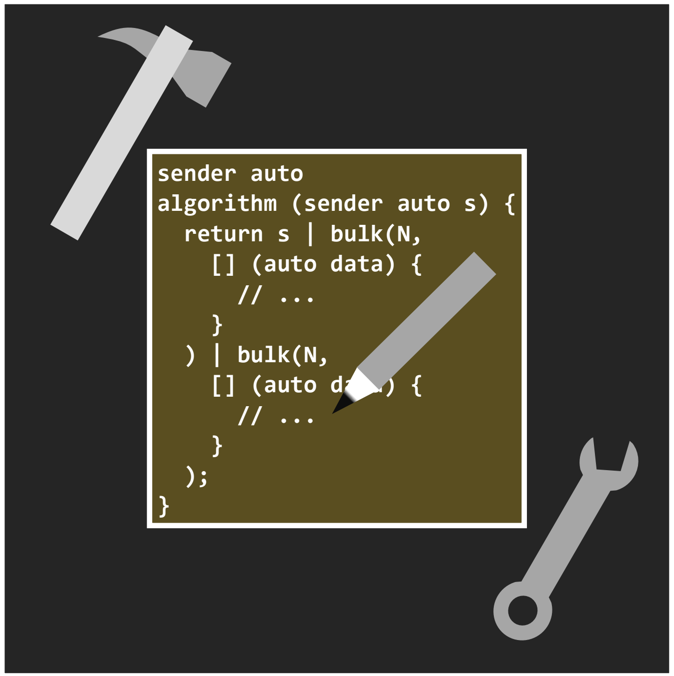
Pillars of C++ Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries

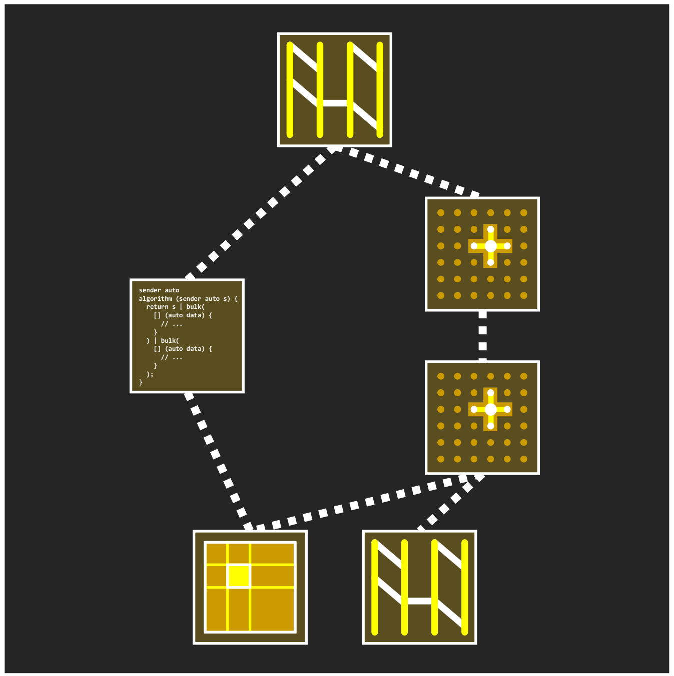


Expanding the Set

Tools to Write Your Own Parallel Algorithms that Run Anywhere



Mechanisms for Composing Parallel Invocations into Task Graphs



```
double *A = …, *x = …, *y = …;

double *dA, *dx, *dy;
cudaMalloc(&dA, N * M * sizeof(double));
cudaMalloc(&dx, M * sizeof(double));
cudaMalloc(&dy, N * sizeof(double));

cublasSetMatrix(N, M, sizeof(double), &A, N, dA, N);
cublasSetVector(M, sizeof(double), &x, 1, dx, 1);
cublasSetVector(N, sizeof(double), &y, 1, dy, 1);

cublasHandle_t handle;
cublasCreate(&handle);

double alpha = 3.0, beta = 2.0;
cublasSgemv(handle, CUBLAS_OP_N, N, M,
            &alpha, dA, N, dx, 1, &beta, dy, 1);

cublasGetVector(N, sizeof(double), &y, 1, dy, 1);
```

```
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);
```

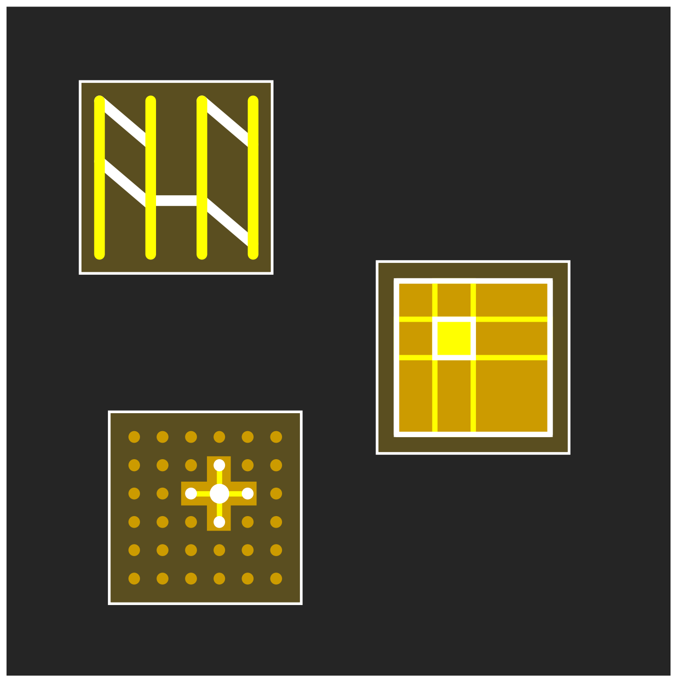
Standard Linear Algebra

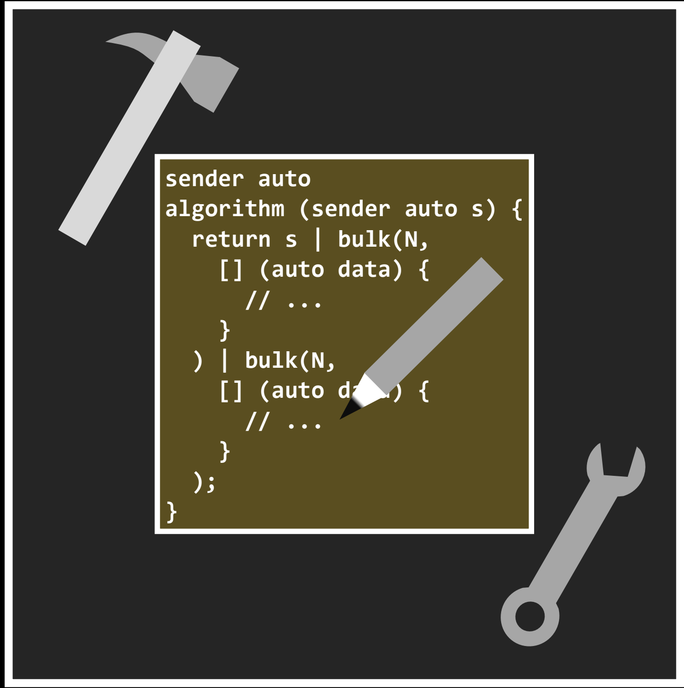
```
std::mdspan A{…, N, M}; std::mdspan x{…, M}; std::mdspan b{…, N};

// 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);
```

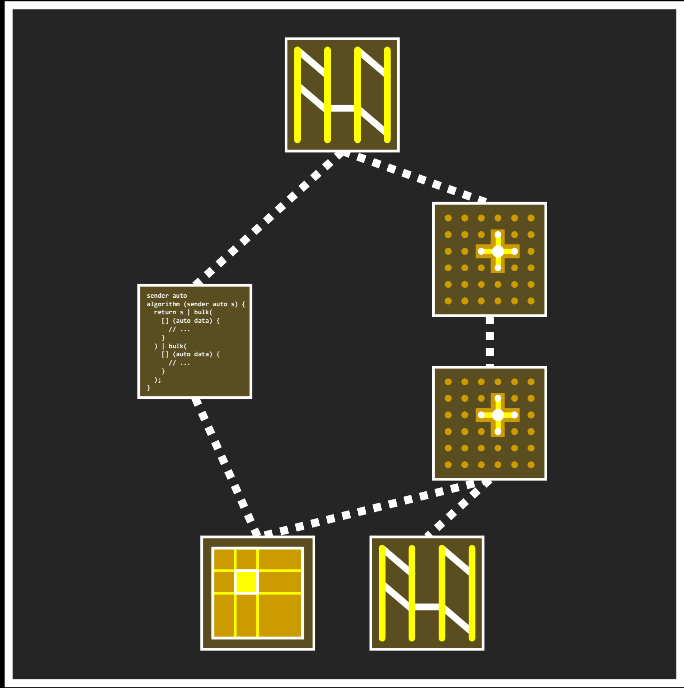
Planned for C++26 and available since NVC++ 22.7!

Pillars of C++ Standard Parallelism

Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries	Tools to Write Your Own Parallel Algorithms that Run Anywhere	Mechanisms for Composing Parallel Invocations into Task Graphs

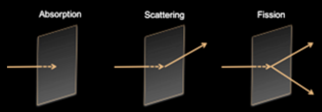




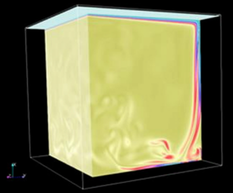


Standard Parallelism Adoption

abinit

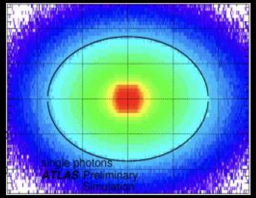


Quicksilver



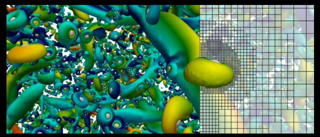
STLBM

GAMESS



FastCaloSim

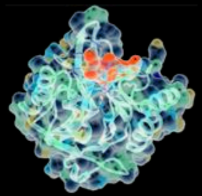
RAJAPerf



M-AIA

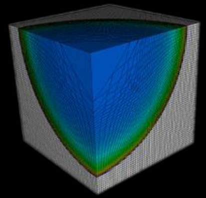
BabelStream

POT3D



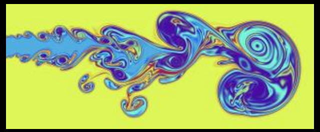
MiniBUDE

MAS



LULESH

Palabos



MiniWeather

Cloverleaf

DIFFUSE

JAEA MiniApps

HipFT

We Need On-Ramps