The C++20 Synchronization Library

Bryce Adelstein Lelbach, Meeting C++ 2019

```
unique_future<std::uint64_t>
fibonacci(execution_policy auto&& s, std::uint64_t n) {
  if (n < 2) co_return n;

  auto n1 = async(s, fibonacci<decltype(s)>, s, n - 1);
  auto n2 = fibonacci(s, n - 2);

  co_return co_await n1 + co_await n2;
}
```

THE C++20 SYNCHRONIZATION LIBRARY

Bryce Adelstein Lelbach
@blelbach
CUDA C++ Core Libraries Lead
ISO C++ Library Evolution Incubator Chair, ISO C++ Tooling Study Group Chair

#include <C++>

includecpp.org

THE C++20 SYNCHRONIZATION LIBRARY

Bryce Adelstein Lelbach

@blelbach

CUDA C++ Core Libraries Lead

ISO C++ Library Evolution Incubator Chair, ISO C++ Tooling Study Group Chair

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

void f(std::invocable auto&&);
// ^ Constrained template.
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- Parallel algorithms.

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- Parallel algorithms.

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};
```

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    thread_group tg(6, [&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    thread_group tg(6, [&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
struct thread_group {
private:
  std::vector<std::thread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }

  ~thread_group() {
    stdr::for_each(members, [] (std::thread& t) { t.join(); });
  }
};
```

```
struct thread_group {
private:
  std::vector<std::jthread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};
```

`std::jthread`

Joining Thread

- Just like `std::thread`, except:
- When destroyed, if the thread is joinable, it joins instead of calling `terminate`.

```
struct thread_group {
private:
  std::vector<std::jthread> members;

public:
  thread_group(std::uint64_t n, std::invocable auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};
```

```
struct thread_group {
private:
  std::vector<std::jthread> members;

public:
  thread_group(std::uint64_t n, std::invocable<std::stop_token> auto&& f) {
    for (auto i : stdv::iota(0, n)) members.emplace_back(f);
  }
};

int main() {
  std::atomic<std::uint64_t> count(0);

  {
    thread_group tg(6,
      [&] (std::stop_token s) { while (!s.stop_requested()) ++count; }
    );
  }

  std::cout << count << "\n";
}
```

`std::jthread`

Joining Thread

- Just like `std::thread`, except:
- When destroyed, if the thread is joinable, it joins instead of calling `terminate`.
- It supports interruption.
  - `std::jthread` invocables will be passed a `std::stop_token` parameter if they support it.
- Interruption API:

```
[[nodiscard]] stop_source std::jthread::get_stop_source() noexcept;
[[nodiscard]] stop_token std::jthread::get_stop_token() const noexcept;
bool std::jthread::request_stop() noexcept;
```

`std::stop_*`
Interruption Facilities

- `std::stop_source` (analogous to a `promise`)
  - Producer of stop requests.
  - Owns the shared state (if any).
- `std::stop_token` (analogous to `future`)
  - Handle to a `std::stop_source`.
  - Consumer only; can query for stop requests, but can't make them.
- `std::stop_callback` (analogous to `future::then`)
  - Mechanism for registering invocables to be run upon receiving a stop request.

CV Interruption Support

```
struct condition_variable_any {
  template <typename Lock, typename Predicate>
    bool wait(Lock& lock, stop_token stoken, Predicate pred);
  template <typename Lock, class Clock, typename Duration, typename Predicate>
    bool wait_until(Lock& lock, stop_token stoken,
                    chrono::time_point<Clock, Duration> const& abs, Predicate pred);
  template <typename Lock, typename Rep, typename Period, typename Predicate>
    bool wait_for(Lock& lock, stop_token stoken,
                  const chrono::duration<Rep, Period>& rel, Predicate pred);
};
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- Parallel algorithms.

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
template <ptrdiff_t least_max_value = implementation-defined>
struct counting_semaphore {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit counting_semaphore(ptrdiff_t desired);

  void release(ptrdiff_t update = 1);

  void acquire();
  bool try_acquire() noexcept;
  template <typename Rep, typename Period>
    bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
  template <typename Clock, typename Duration>
    bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);
};
```

std::counting_semaphore

```
using binary_semaphore = counting_semaphore<1>;
```

`std::mutex` vs `std::counting_semaphore<N>`

`std::mutex`

- Ensures a resource is only accessed by one thread at a time.
- Each thread which needs the resource blocks until it receives it.
- Thread identity:
    - Only the locking thread may unlock.
    - A locked mutex is unlocked once.

`std::counting_semaphore<N>`

- Does not limit how many threads access resources concurrently.
- Each thread which needs a resource blocks until it receives one.
- No thread identity:
    - Any thread may release.
    - A thread may release up to `N` count.

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u)
  {
    std::scoped_lock l(items_mtx);
    items.emplace(std::forward<decltype(u)>(u));
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u)
  {
    std::scoped_lock l(items_mtx);
    items.emplace(std::forward<decltype(u)>(u));
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u)
  {
    std::scoped_lock l(items_mtx);
    items.emplace(std::forward<decltype(u)>(u));
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    push(std::forward<decltype(u)>(u));
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    push(std::forward<decltype(u)>(u));
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    push(std::forward<decltype(u)>(u));
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  void enqueue(std::convertible_to<T> auto&& u) {
    remaining_space.acquire();
    push(std::forward<decltype(u)>(u));
    items_produced.release();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop() {
    std::optional<T> tmp;
    std::scoped_lock l(items_mtx);
    tmp = std::move(items.front());
    items.pop();
    return std::move(*tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop() {
    std::optional<T> tmp;
    std::scoped_lock l(items_mtx);
    tmp = std::move(items.front());
    items.pop();
    return std::move(*tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  T pop() {
    std::optional<T> tmp;
    std::scoped_lock l(items_mtx);
    tmp = std::move(items.front());
    items.pop();
    return std::move(*tmp);
  }
};
```

```cpp
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  T dequeue() {
    items_produced.acquire();
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  std::optional<T> try_dequeue() {
    if (!items_produced.try_acquire()) return {};
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

public:
  std::optional<T> try_dequeue() {
    if (!items_produced.try_acquire()) return {};
    T tmp = pop();
    remaining_space.release();
    return std::move(tmp);
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  std::mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    while (flag.test_and_set(std::memory_order_acquire))
      ;
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    for (std::uint64_t k = 0; flag.test_and_set(std::memory_order_acquire); ++k) {
      if (k < 16) __asm__ __volatile__( "rep; nop" : : : "memory" );
      else if (k < 64) sched_yield();
      else {
        timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0; rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, nullptr);
      }
    }
  }

  void unlock() {
    flag.clear(std::memory_order_release);
  }
};
```

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    while (flag.test_and_set(std::memory_order_acquire))
      flag.wait(true, std::memory_order_relaxed);
  }

  void unlock() {
    flag.clear(std::memory_order_release);
    flag.notify_one();
  }
};
```

```
struct spin_mutex {
private:
  std::atomic<bool> flag = ATOMIC_VAR_INIT(false);

public:
  void lock() {
    while (flag.exchange(true, std::memory_order_acquire))
      flag.wait(true, std::memory_order_relaxed);
  }

  void unlock() {
    flag.store(false, std::memory_order_release);
    flag.notify_one();
  }
};
```

std::atomic{_flag}::wait/notify

```
template <typename T>
struct atomic {
  void wait(T old, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(T old, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

std::atomic{_flag}::wait/notify

```
template <typename T>
struct atomic {
  void wait(T old, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(T old, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

std::atomic{_flag}::wait/notify

```
template <typename T>
struct atomic {
  void wait(T old, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(T old, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

std::atomic{_flag}::wait/notify

```
struct atomic_flag {
  void wait(bool old, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(bool old, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;
};
```

std::atomic_flag::test

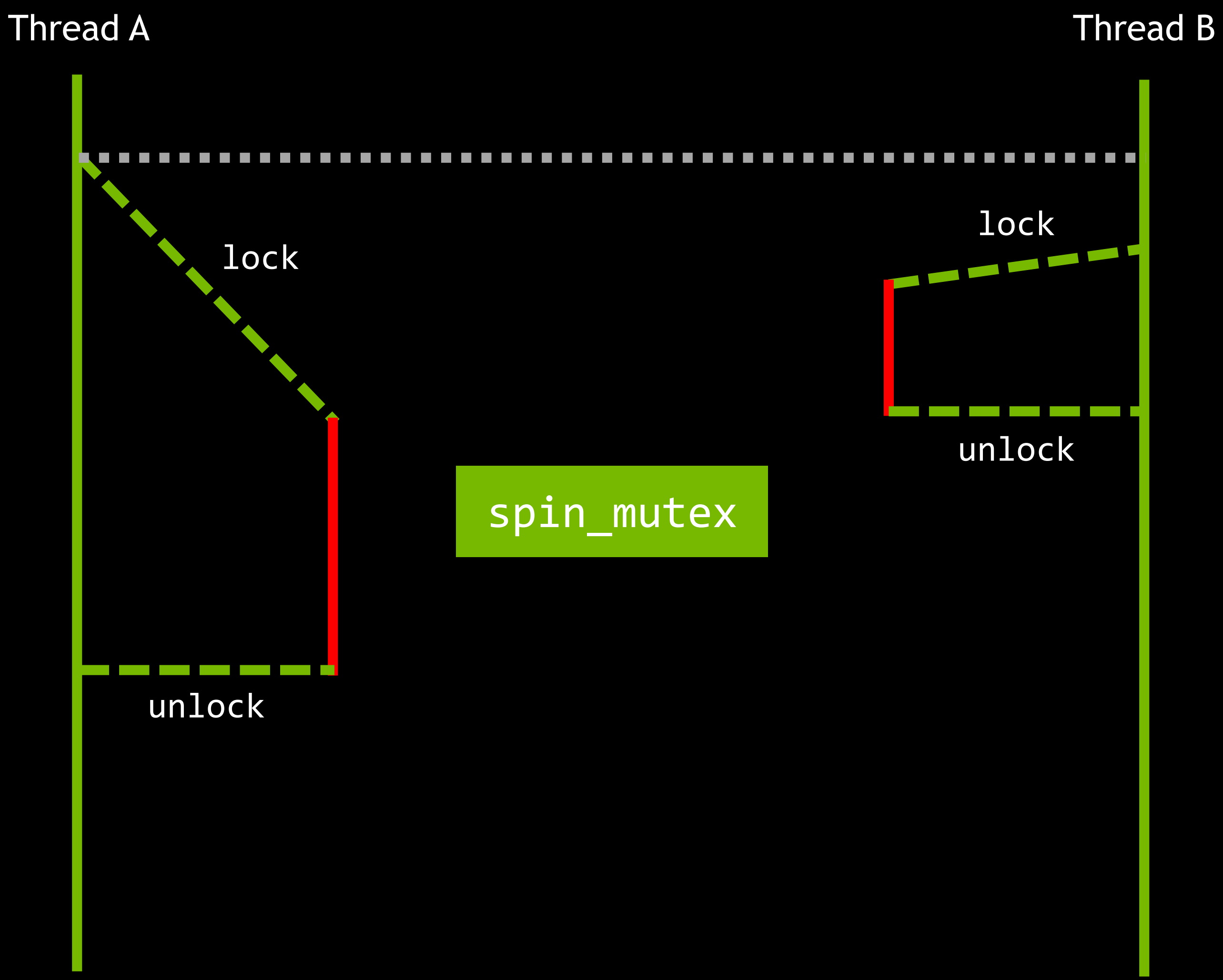
```
struct atomic_flag {
  void wait(bool old, memory_order = memory_order::seq_cst) const volatile noexcept;
  void wait(bool old, memory_order = memory_order::seq_cst) const noexcept;
  void notify_one() volatile noexcept;
  void notify_one() noexcept;
  void notify_all() volatile noexcept;
  void notify_all() noexcept;

  bool test(memory_order = memory_order::seq_cst) const volatile noexcept;
  bool test(memory_order = memory_order::seq_cst) const noexcept;
};
```

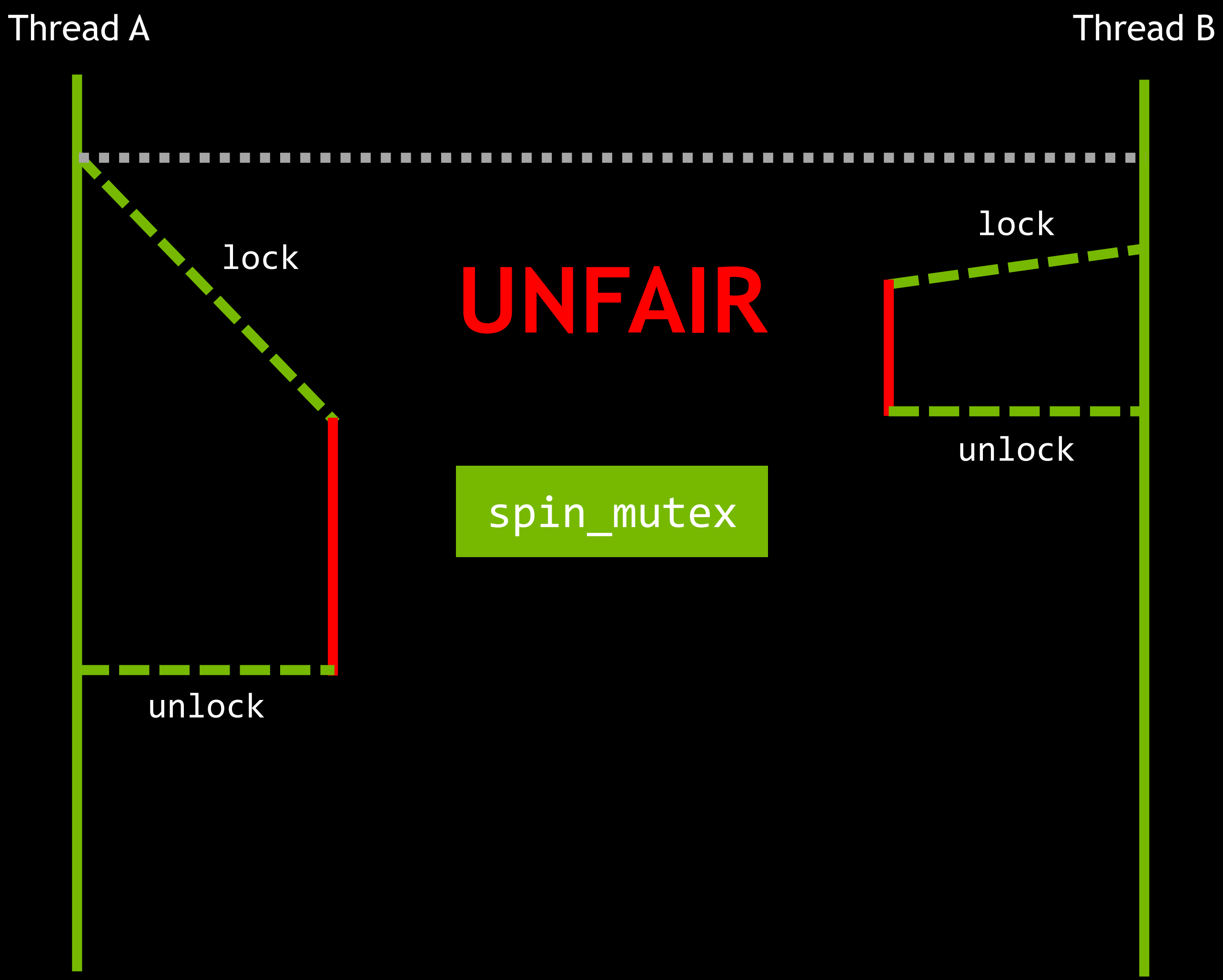
`std::atomic{_flag}` wait and notify

Some possible implementation strategies

- Futex. Supported for certain size objects on Linux and Windows.
- Condition Variables. Supported for certain size objects on Linux and Mac.
- Contention Table. Used to optimize futex notify or to hold CVs.
- Timed back-off. Supported on everything.
- Spinlock. Supported on everything. Note: performance is terrible.

```
struct spin_mutex {
private:
  std::atomic_flag flag = ATOMIC_FLAG_INIT;

public:
  void lock() {
    while (flag.test_and_set(std::memory_order_acquire))
      flag.wait(true, std::memory_order_relaxed);
  }

  void unlock() {
    flag.clear(std::memory_order_release);
    flag.notify_one();
  }
};
```


Thread A
Thread B
lock
lock
unlock
unlock
spin_mutex


Thread A
Thread B
lock
lock
UNFAIR
unlock
spin_mutex
unlock

```
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```cpp
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

PLEASE Take A Number
NOW SERVING
52

```
struct ticket_mutex {
private:
  std::atomic<int> in  = ATOMIC_VAR_INIT(0);
  std::atomic<int> out = ATOMIC_VAR_INIT(0);



public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
struct ticket_mutex {
private:
  alignas(std::hardware_destructive_interference_size) std::atomic<int> in
    = ATOMIC_VAR_INIT(0);
  alignas(std::hardware_destructive_interference_size) std::atomic<int> out
    = ATOMIC_VAR_INIT(0);

public:
  void lock() {
    auto const my = in.fetch_add(1, std::memory_order_acquire);
    while (true) {
      auto const now = out.load(std::memory_order_acquire);
      if (now == my) return;
      out.wait(now, std::memory_order_relaxed);
    }
  }

  void unlock() {
    out.fetch_add(1, std::memory_order_release);
    out.notify_all();
  }
};
```

```
template <typename T, std::uint64_t QueueDepth>
struct concurrent_bounded_queue {
private:
  std::queue<T> items;
  ticket_mutex items_mtx;
  std::counting_semaphore<QueueDepth> items_produced{0};
  std::counting_semaphore<QueueDepth> remaining_space{QueueDepth};

  void push(std::convertible_to<T> auto&& u);
  T pop();

public:
  constexpr concurrent_bounded_queue() = default;

  void enqueue(std::convertible_to<T> auto&& u);

  T dequeue();
  std::optional<T> try_dequeue();
};
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- **Termination detection mechanism.**
- Parallel algorithms.

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

public:
  void submit(std::invocable auto&& f) {
    tasks.enqueue(std::forward<decltype(f)>(f));
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s);

public:
  bounded_depth_task_manager(std::uint64_t n);

  void submit(std::invocable auto&& f);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    for (auto i : stdv::iota(0, 256))
      tm.submit([&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
int main() {
  std::atomic<std::uint64_t> count(0);

  {
    bounded_depth_task_manager<64> tm(6);

    for (auto i : stdv::iota(0, 256))
      tm.submit([&] { ++count; });
  }

  std::cout << count << "\n";
}
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  concurrent_bounded_queue<std::any_invocable<void()>, QueueDepth> tasks;
  thread_group threads;

  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }

public:
  bounded_depth_task_manager(std::uint64_t n)
    : threads(n, [&] (std::stop_token s) { process_tasks(s); })
  {}
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

std::latch

```
struct latch {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

`std::latch`

```
struct latch {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

std::latch

```
struct latch {
  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit latch(ptrdiff_t expected);

  latch(const latch&) = delete;
  latch& operator=(const latch&) = delete;

  void count_down(ptrdiff_t update = 1);
  bool try_wait() const noexcept;
  void wait() const;
  void arrive_and_wait(ptrdiff_t update = 1);
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
    while (true) {
      if (auto f = tasks.try_dequeue()) std::move(*f)();
      else break;
    }
  }
public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

```
template <std::uint64_t QueueDepth>
struct bounded_depth_task_manager {
private:
  void process_tasks(std::stop_token s) {
    while (!s.stop_requested())
      tasks.dequeue()();
  }




public:
 ~bounded_depth_task_manager() {
    std::latch l(threads.size() + 1);
    for (auto i : stdv::iota(0, threads.size()))
      submit([&] { l.arrive_and_wait(); });
    threads.request_stop();
    l.count_down();
  }
};
```

Recipe For a Tasking Runtime

- Worker threads.
- Multi-consumer, multi-producer concurrent queue.
- Termination detection mechanism.
- **Parallel algorithms.**

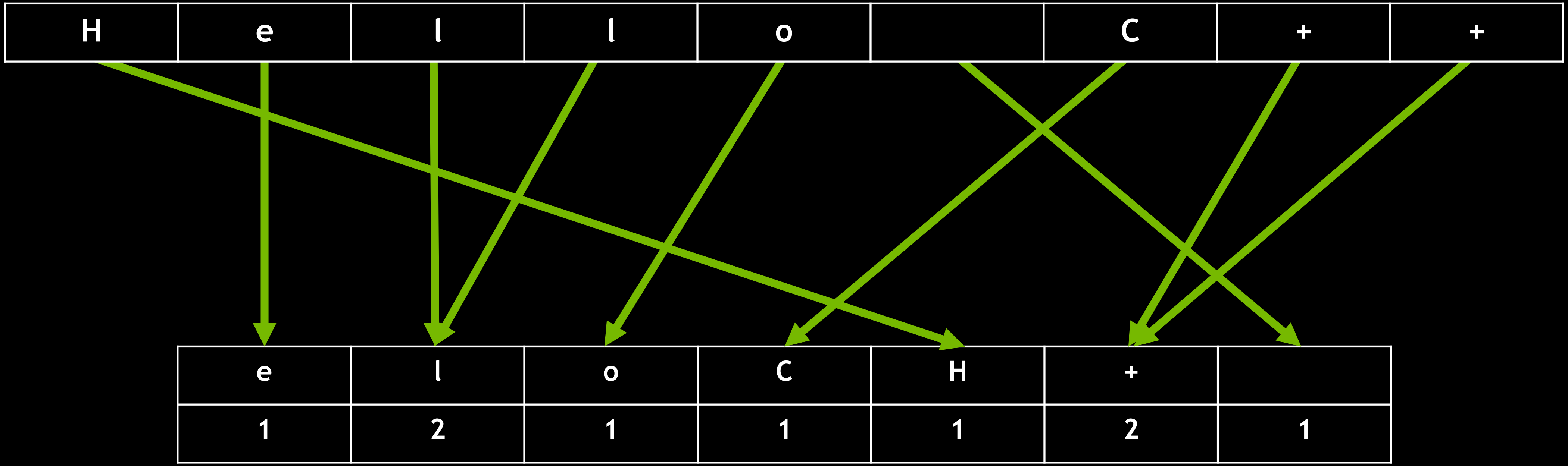
```
template <stdr::range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(I&& input, O output, T inc, OP op) {
  stdr::for_each(input, [&] (auto&& t) { output[op(t)] += inc; });
}
```

```
template <stdr::range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(I&& input, O output, T inc, OP op) {
  stdr::for_each(input, [&] (auto&& t) { output[op(t)] += inc; });
}
```

Histogram

H	e	l	l	o		C	+	+

Histogram



e	l	o	C	H	+	
1	2	1	1	1	2	1

```
template <execution_policy EP,
          stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(EP&& exec, I&& input, O output, T inc, OP op);
```

```
template <execution_policy EP,
          stdr::random_access_range I, std::random_access_iterator O,
          typename T, std::invocable</* ... */> BO>
  requires /* ... */
void histogram(EP&& exec, I&& input, O output, T inc, OP op);
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents() * 4;
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents() * 4;
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents() * 4;
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents() * 4;
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  std::latch l(chunks);

 // ...
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      // ...
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      // ...
    );
}
```

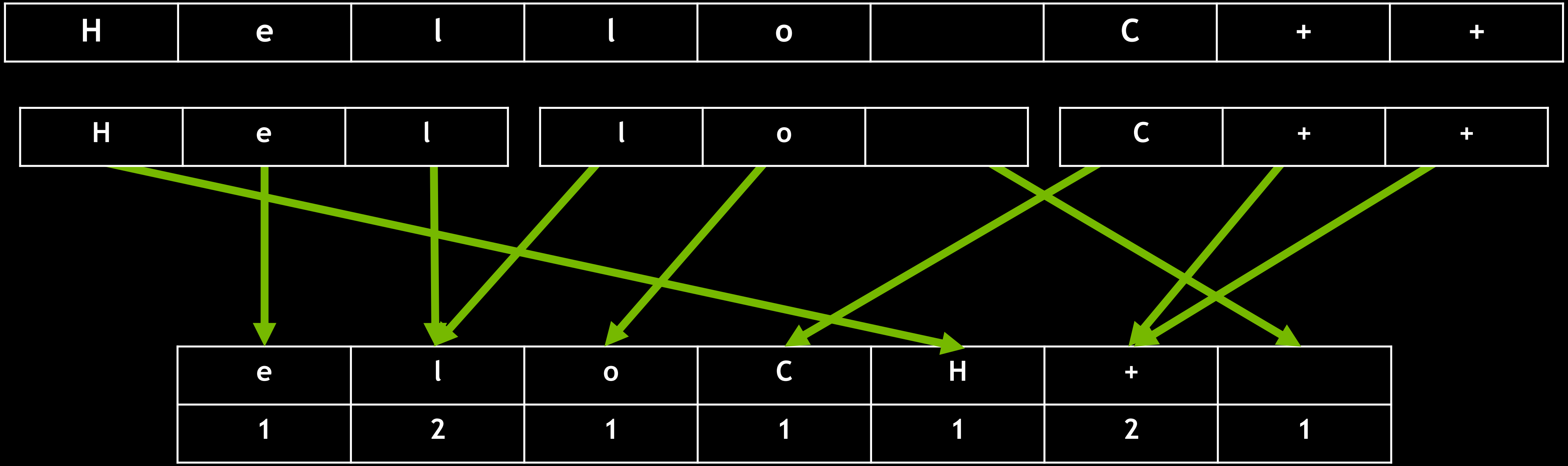
```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        // ...
      }
    );
}
```

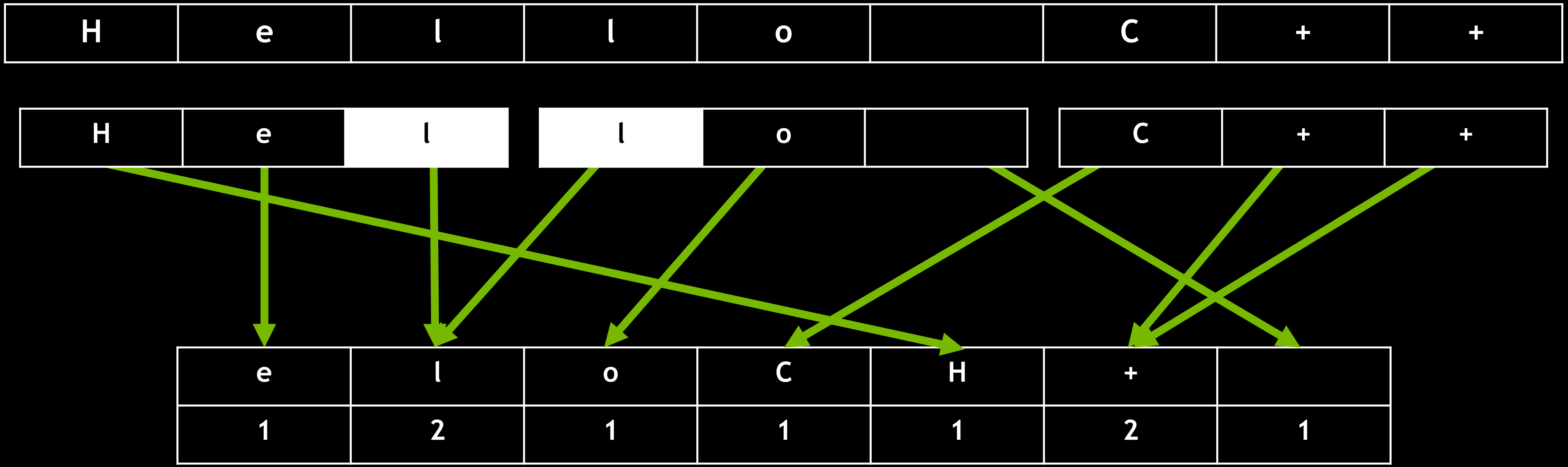
```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        stdr::for_each(stdr::begin(input) + my_begin,
                       stdr::begin(input) + my_end,
                       [&] (auto&& t) {
                         output[op(t)] += inc;
                       });

        // ...
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        stdr::for_each(stdr::begin(input) + my_begin,
                       stdr::begin(input) + my_end,
                       [&] (auto&& t) {
                         output[op(t)] += inc;
                       });

        // ...
      }
    );
}
```

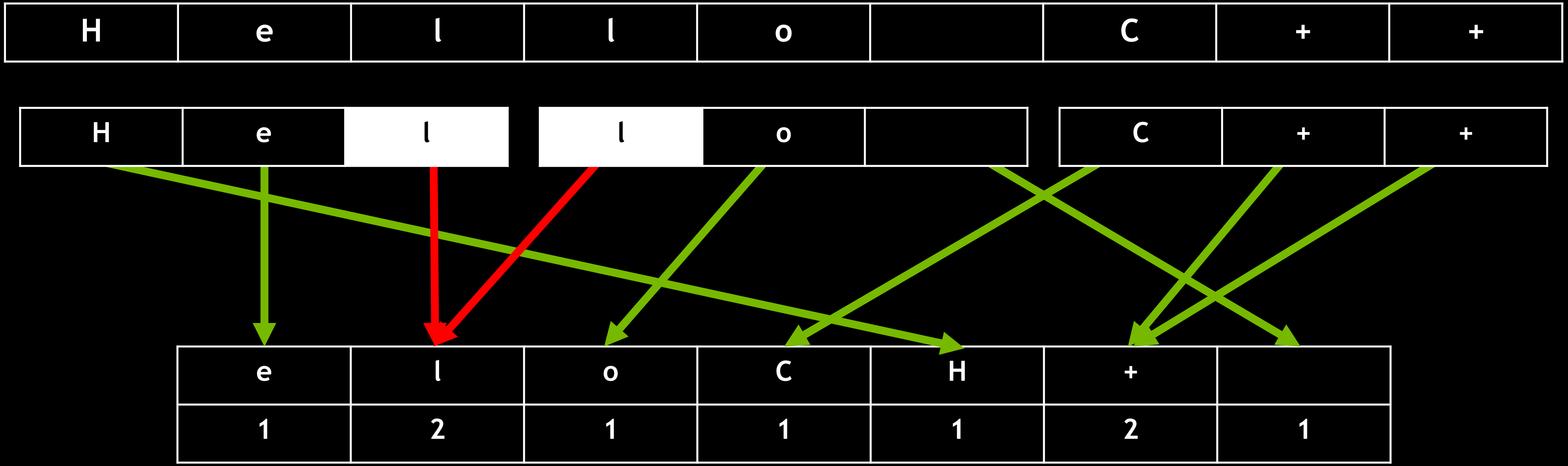
Histogram



Histogram



Histogram



```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        stdr::for_each(stdr::begin(input) + my_begin,
                       stdr::begin(input) + my_end,
                       [&] (auto&& t) {
                         std::atomic_ref r(output[op(t)]);
                         r.fetch_add(inc, std::memory_order_relaxed);
                       });


        // ...
      }
    );
}
```

std::atomic_ref<T>

std::atomic<T> holds a T.

```
template <struct T>
struct atomic {
private:
  T data; // exposition only
public:
  // ...
};
```

std::atomic_ref<T> does not hold a T.

```
template <struct T>
struct atomic_ref {
private:
  T* ptr; // exposition only
public:
  explicit atomic_ref(T&);

  // Otherwise, same API as std::atomic.
};
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        stdr::for_each(stdr::begin(input) + my_begin,
                       stdr::begin(input) + my_end,
                       [&] (auto&& t) {
                         std::atomic_ref r(output[op(t)]);
                         r.fetch_add(inc, std::memory_order_relaxed);
                       });

        // ...
      }
    );
}
```

std::atomic<*floating-point*>

```
template<> struct atomic<floating-point> {
  floating-point fetch_add(floating-point,
                           memory_order = memory_order_seq_cst) volatile noexcept;
  floating-point fetch_add(floating-point,
                           memory_order = memory_order_seq_cst) noexcept;
  floating-point fetch_sub(floating-point,
                           memory_order = memory_order_seq_cst) volatile noexcept;
  floating-point fetch_sub(floating-point,
                           memory_order = memory_order_seq_cst) noexcept;
};
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      [=, &input, &l] {
        auto const my_begin = chunk * chunk_size;
        auto const my_end   = std::min(elements, (chunk + 1) * chunk_size);

        stdr::for_each(stdr::begin(input) + my_begin,
                       stdr::begin(input) + my_end,
                       [&] (auto&& t) {
                         std::atomic_ref r(output[op(t)]);
                         r.fetch_add(inc, std::memory_order_relaxed);
                       });

        l.count_down();
      }
    );
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents() * 4;
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  std::latch l(chunks);

  for (std::uint64_t chunk = 0; chunk < chunks; ++chunk)
    exec.submit(
      // ...
    );

  l.wait();
}
```

```
void histogram(EP&& exec, I&& input, O output, T inc, OP op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents() * 4;
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  std::latch l(chunks);

  for (std::uint64_t chunk = 0; chunk < chunks; ++chunk)
    exec.submit(
      // ...
    );

  l.wait();
}
```

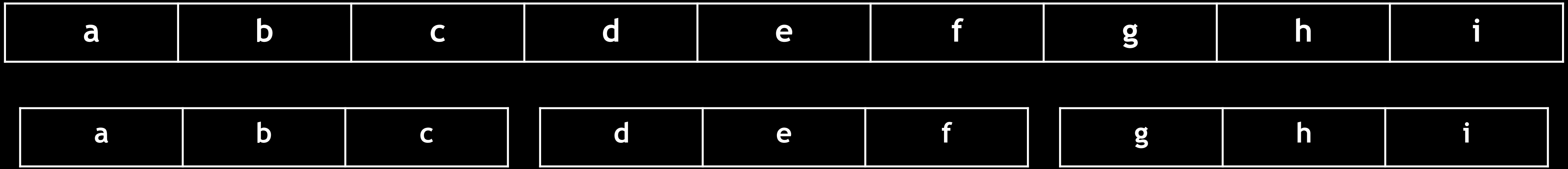
```
template <execution_policy EP,
          stdr::random_access_range I, std::random_access_iterator O,
          std::invocable</* ... */> BO>
  requires /* ... */
void inclusive_scan(EP&& exec, I&& input, O output, OP op);
```

Inclusive Scan

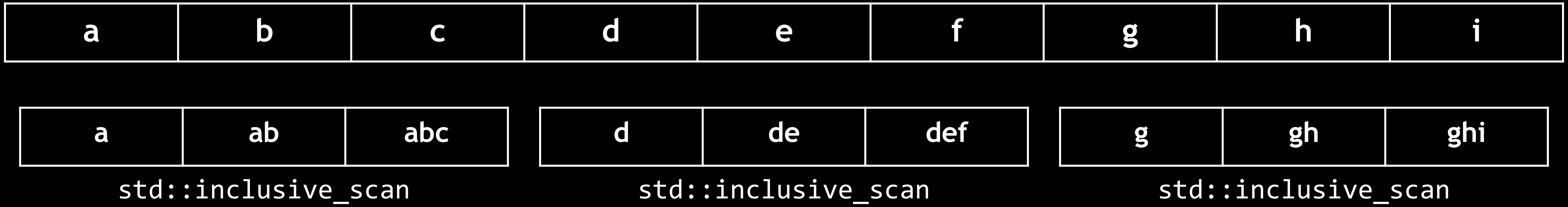
a	b	c	d	e	f	g	h	i

a	ab	abc	abcd	abcde	abcdef	abcdefg	abcdefgh	abcdefghi

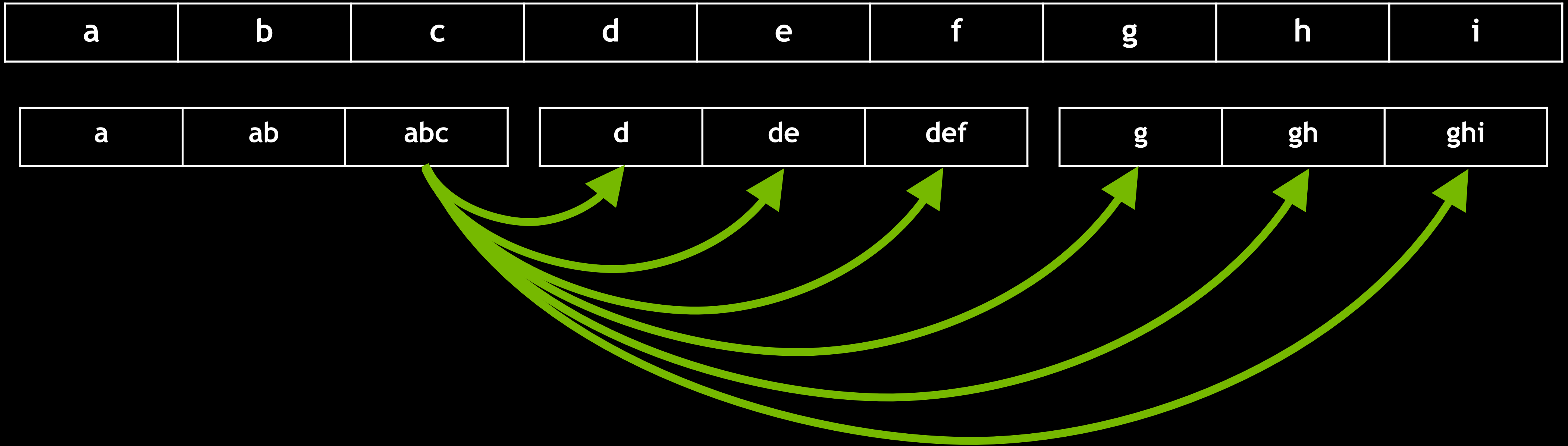
Inclusive Scan



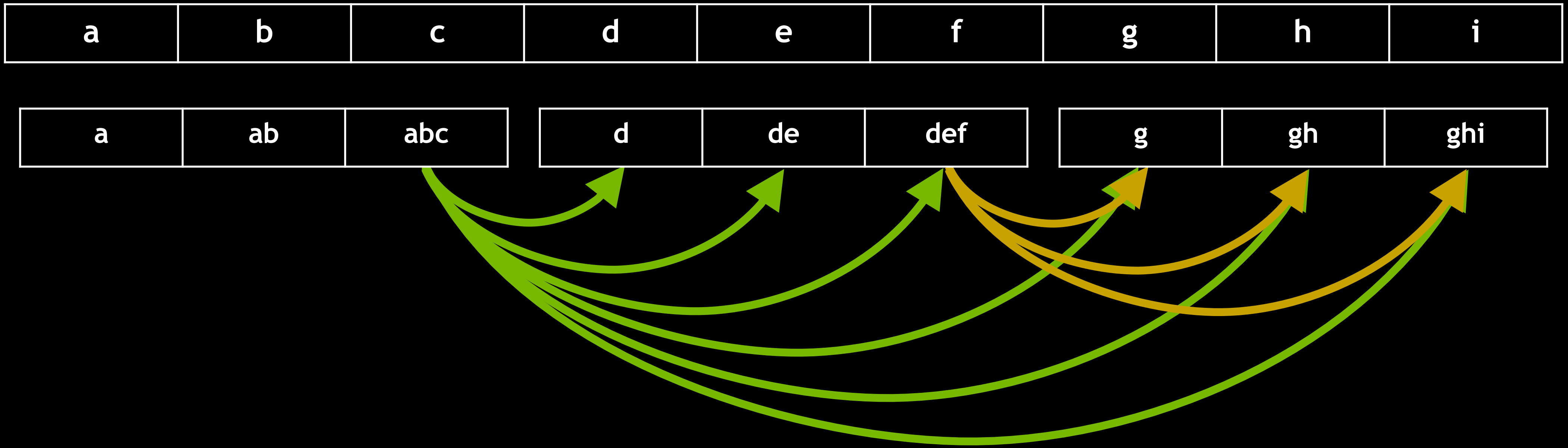
Inclusive Scan



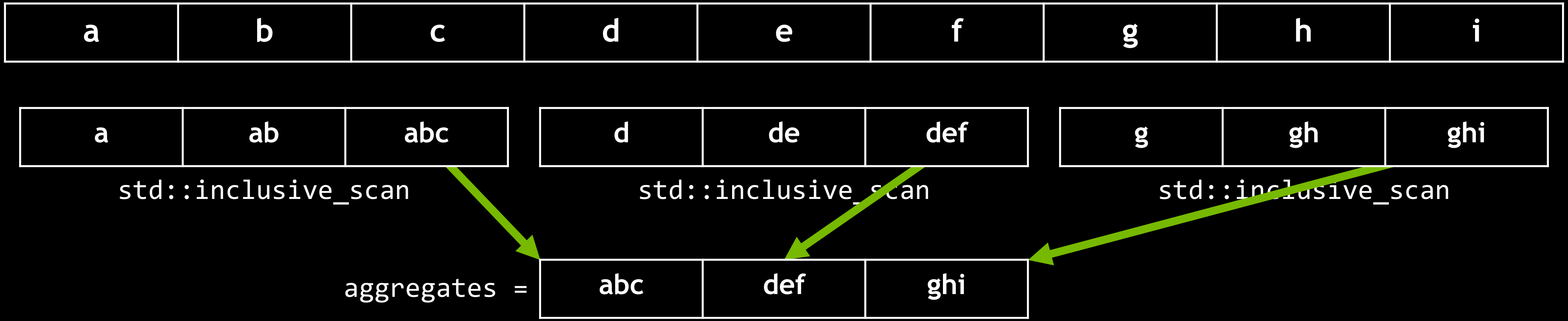
Inclusive Scan



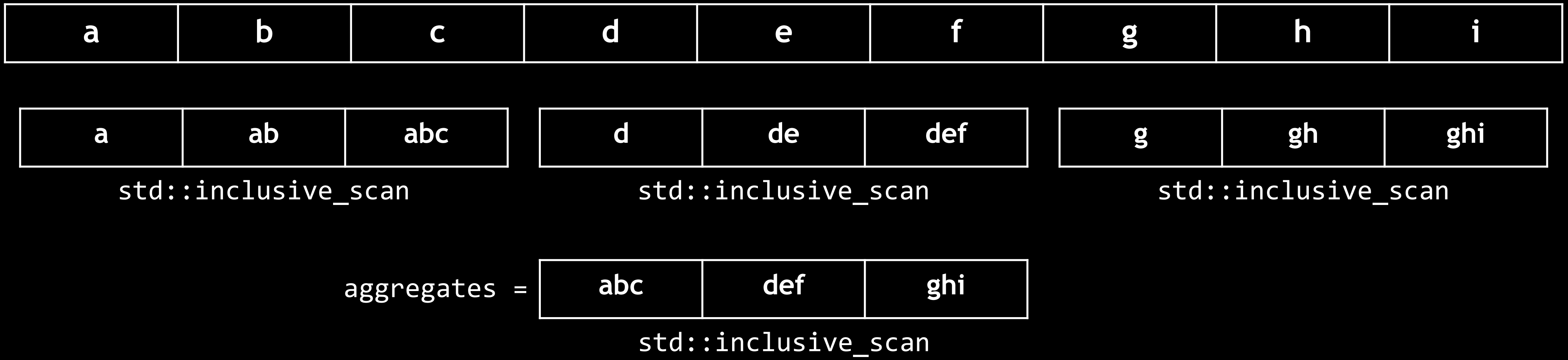
Inclusive Scan



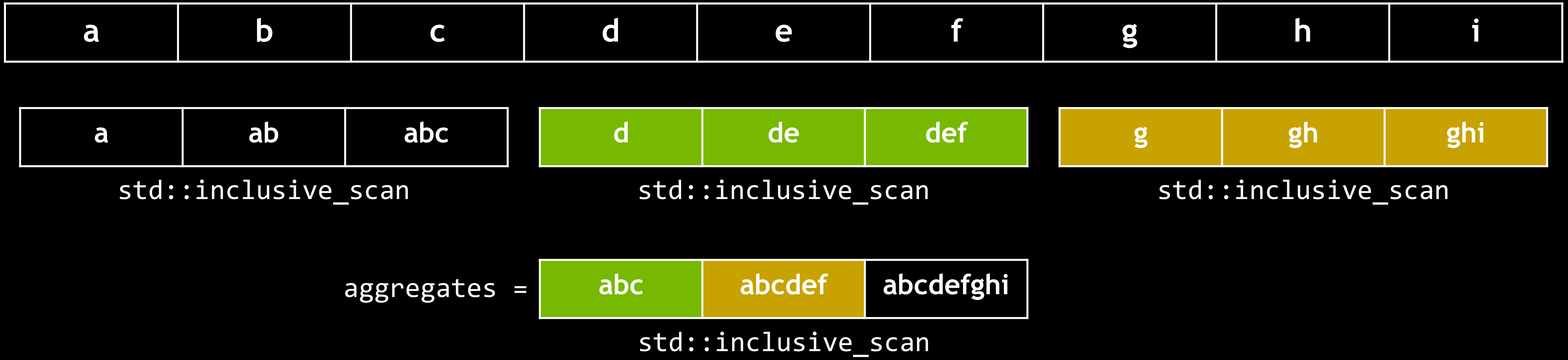
Inclusive Scan



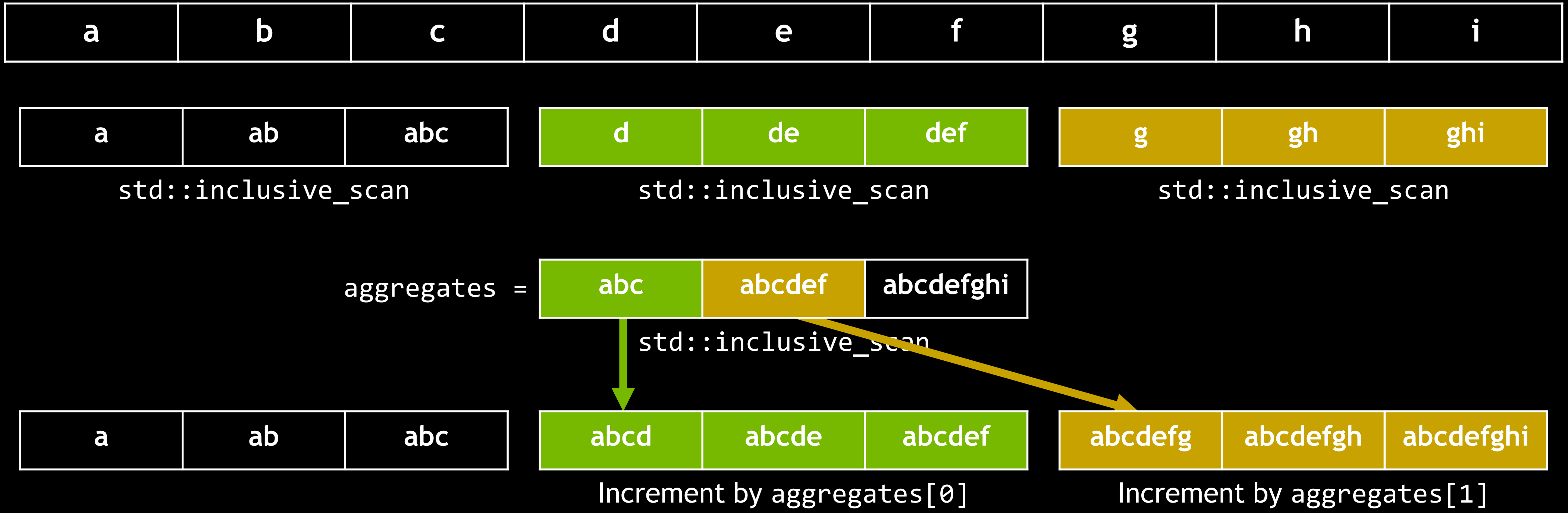
Inclusive Scan



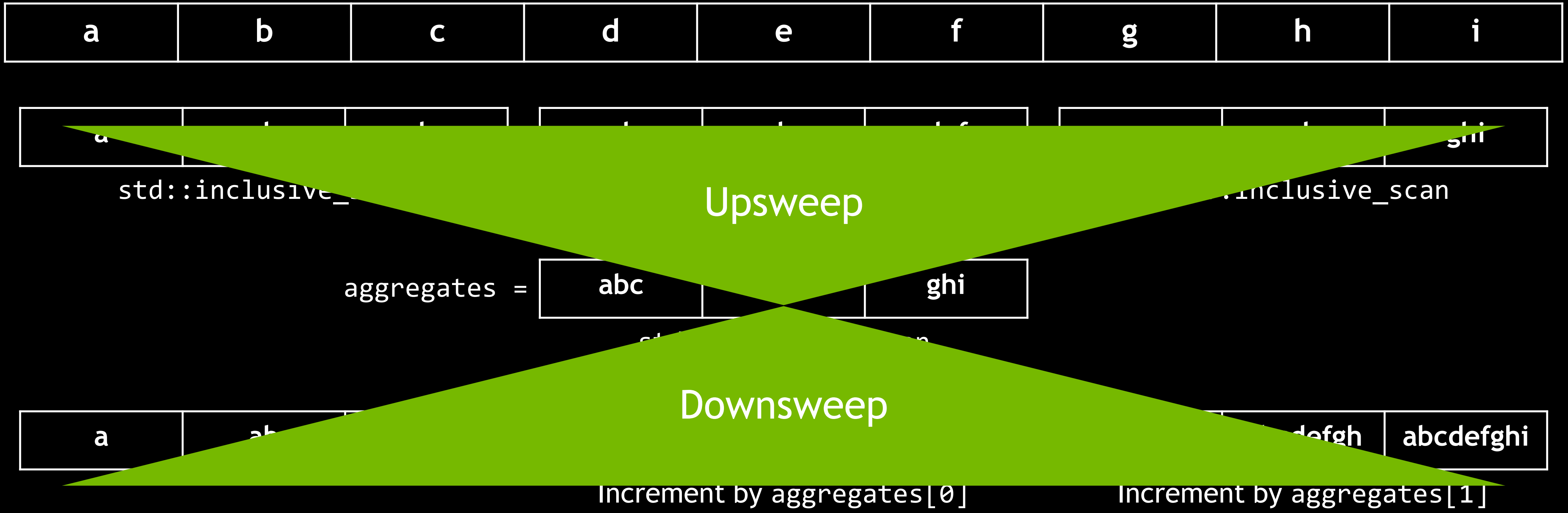
Inclusive Scan



Inclusive Scan



Inclusive Scan



```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier</* ... */> upsweep_barrier(chunks, /* ... */);

  std::latch              downsweep_latch(chunks);

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier</* ... */> upsweep_barrier(chunks, /* ... */);

  std::latch              downsweep_latch(chunks);

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier</* ... */> upsweep_barrier(chunks, /* ... */);

  std::latch              downsweep_latch(chunks);

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier</* ... */> upsweep_barrier(chunks, /* ... */);

  std::latch              downsweep_latch(chunks);

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier</* ... */> upsweep_barrier(chunks, /* ... */);

  std::latch              downsweep_latch(chunks);

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op);


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op);


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op);


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op);


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op);


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier<std::function<void()>> upsweep_barrier(chunks,
    [&] { stdr::inclusive_scan(aggregates, aggregates.begin(), op); });
  std::latch                          downsweep_latch(chunks);

  // ...
}
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

std::barrier

```
template <typename CompletionFunction = see below>
struct barrier {
  using arrival_token = see below;

  static constexpr ptrdiff_t max() noexcept;

  constexpr explicit barrier(ptrdiff_t expected,
                             CompletionFunction f = CompletionFunction());

  [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
  void wait(arrival_token&& arrival) const;

  void arrive_and_wait();
  void arrive_and_drop();
};
```

`std::latch` vs `std::barrier`

`std::latch`

- Supports asynchronous arrival.
- Single phase.
- No thread identity:
  - Threads may arrive multiple times.
  - Any thread may wait on a latch.
- No completion function.

`std::barrier`

- Supports asynchronous arrival.
- Multi phase.
- Thread identity:
  - A thread may arrive only once per phase.
  - Only a thread who has arrived may wait.
- Supports completion functions.

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  using T = stdr::ranges_value_t<I>;
  std::vector<T> aggregates(chunks);

  std::barrier<std::function<void()>> upsweep_barrier(chunks,
    [&] { stdr::inclusive_scan(aggregates, aggregates.begin(), op); });
  std::latch                          downsweep_latch(chunks);

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op),


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      auto const this_begin   = chunk * chunk_size;
      auto const this_end     = std::min(elements, (chunk + 1) * chunk_size);
      aggregates[chunk] = *--stdr::inclusive_scan(stdr::begin(input) + this_begin,
                                                  stdr::begin(input) + this_end,
                                                  output + this_begin,
                                                  op),


      upsweep_barrier.arrive_and_wait();

      // ...
    }));

  // ...
}
```

`std::barrier`

Synchronization

$$\begin{tikzcd} \text{N x arrives} \\ \text{1 x completion} \arrow[u, bend right, "\text{happens-before}"'] \\ \text{N x waits complete} \arrow[u, bend right, "\text{happens-before}"'] \end{tikzcd}$$

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      // ...

      upsweep_barrier.arrive_and_wait();

      if (0 != chunk)
        stdr::for_each(output + this_begin, output + this_end,
                       [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk - 1]); });

      downsweep_latch.count_down();
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      // ...

      upsweep_barrier.arrive_and_wait();

      if (0 != chunk)
        stdr::for_each(output + this_begin, output + this_end,
                       [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk - 1]); });

      downsweep_latch.count_down();
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      // ...

      upsweep_barrier.arrive_and_wait();

      if (0 != chunk)
        stdr::for_each(output + this_begin, output + this_end,
                       [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk - 1]); });

      downsweep_latch.count_down();
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  // ...

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit([=, &aggregates, &upsweep_barrier, &downsweep_latch] {
      // ...

      upsweep_barrier.arrive_and_wait();

      if (0 != chunk)
        stdr::for_each(output + this_begin, output + this_end,
                       [&, chunk] (auto& t) { t = op(std::move(t), aggregates[chunk - 1]); });

      downsweep_latch.count_down();
    }));

  // ...
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  std::vector<T> aggregates(chunks);

  std::barrier<std::function<void()>> upsweep_barrier(chunks,
    [&] { stdr::inclusive_scan(aggregates, aggregates.begin(), op); });
  std::latch                          downsweep_latch(chunks);

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      // ...
    );

  downsweep_latch.wait();
}
```

```
void inclusive_scan(EP&& exec, I&& input, O&& output, BO&& op) {
  std::uint64_t const elements   = stdr::distance(input);
  std::uint64_t const chunks     = exec.concurrent_agents();
  std::uint64_t const chunk_size = (elements + chunks - 1) / chunks;

  std::vector<T> aggregates(chunks);

  std::barrier<std::function<void()>> upsweep_barrier(chunks,
    [&] { stdr::inclusive_scan(aggregates, aggregates.begin(), op); });
  std::latch                          downsweep_latch(chunks);

  for (auto chunk : stdv::iota(0, chunks))
    exec.submit(
      // ...
    );

  downsweep_latch.wait();
}
```

C++20 Synchronization Library

- `std::atomic<T>` et al
  - `wait`/`notify` interface
  - `std::atomic_ref<T>`
  - `test` interface for `std::atomic_flag`
  - Floating-point specializations

- `std::latch` & `std::barrier`
- `std::counting_semaphore`
- `std::jthread`
  - Joining destructor
  - `std::stop_*` interruption mechanism

libcu++

The CUDA C++ Standard Library

- Opt-in, heterogeneous, incremental C++ standard library for CUDA.
- Port of LLVM's libc++; contributed C++20 sync library upstream.
- **Version 1 (next week): `<atomic>` (Pascal+), `<type_traits>`.**
- Version 2 (1H 2020): `atomic<T>::wait/notify` (Volta+), `<barrier>` (Volta+), `<latch>` (Volta+), `<counting_semaphore>` (Volta+), `<chrono>`, `<ratio>`, `<functional>` minus `function`.
- Future priorities: `<complex>`, `<tuple>`, `<array>`, `<utility>`, `<cmath>`, string processing, …

```
#include <atomic>
std::atomic<int> x;

#include <cuda/std/atomic>
cuda::std::atomic<int> x;

#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

`std::`	ISO C++, __host__ only.
`cuda::std::`	CUDA C++, __host__ __device__. Strictly conforming to ISO C++.
`cuda::`	CUDA C++, __host__ __device__. Conforming extensions to ISO C++.

```
#include <atomic>
std::atomic<int> x;

#include <cuda/std/atomic>
cuda::std::atomic<int> x;

#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

CUDA is the only GPU platform that implements C++ parallel forward progress and the C++ memory model; not possible with OpenCL or SYCL.

`std::`	ISO C++, `__host__` only.
`cuda::std::`	CUDA C++, `__host__ __device__`. Strictly conforming to ISO C++.
`cuda::`	CUDA C++, `__host__ __device__`. Conforming extensions to ISO C++.

@blelbach