Part 7

Tyler Swann

Agenda

- Parallel Algorithms
- Atomics
- Threads
- Mutexes & Locks
- Aync
- Discussion

Parallel Algorithms

Parallel Algorithms

- Most of C++ standard algorithms have overloads used to execute the algorithms in a parallel context.
- These use unspecified strategies that are implemented by the compiler.
- Some numerical algorithms do not have parallel overloads due to issues with the existing algorithms constraining (by assumption) the binary operator to have certain commutive and associative properties.
- Instead, there are new algorithms that have different semantics that are required to for possible parallel execution.
- Programs that use parallel algorithms must link against an external threading library such as Intel's Thread Building Blocks (TBB) so that the compiler has the means to implement the parallel execution.

EXECUTION POLICIES

- To execute a parallel algorithm, you supply it with an execution policy as the first argument.
- These policies indicate to the compiler that the algorithm can use other means of execution.
- These execution policies are found in the `std::execution` namespace from the `<execution>` header.
- There are four standard execution policies, sequential, parallel, sequenced and parallel-unsequenced.

Execution Policies

Policy	Description
`std::execution::seq`	Forbids an algorithm from being parallelized. Invocations of element access functions within the algorithm are indeterminately sequenced in the calling thread.
`std::execution::par`	Specifies that an algorithm can be parallelized. Invocations of element access functions within the algorithm are permitted to execute in the working thread or in a thread implicitly created by the underlying parallel library. Invocations executing in the same thread are indeterminately sequenced in the calling thread.
`std::execution::unseq`	Specifies that an algorithm can be vectorized such that a single thread using instructions that operate on multiple data items.
`std::execution::par_unseq`	Specifies that an algorithm can be parallelized, vectorized or migrated across threads. Invocations of element access functions within the algorithm are permitted to execute in unordered fashion in unspecified threads and can be un-sequenced with respect to one another within each thread.

Alternative Algorithms Example - Reduce

Algorithm	Execution Policy	Binary Op	Time	Result
`std::accumulate`	Serial	+	151,861 us	10,000,000.7
`std::reduce`	Sequential	+	76,011 us	10,000,000.7
`std::reduce`	Parallel	+	21,098 us	10,000,000.7
`std::reduce`	Unsequenced	+	135,906 us	10,000,000.7
`std::reduce`	Parallel-Unsequenced	+	23,752 us	10,000,000.7

Alternative Algorithms Example – Transform-Reduce

Algorithm	Execution Policy	Binary Op	Time	Result
`std::inner_product`	Serial	(*) → (+)	144,255 us	7,598,000.5455
`std::transform_reduce`	Sequential	(*) → (+)	119,467 us	7,598,000.5455
`std::transform_reduce`	Parallel	(*) → (+)	53,172 us	7,598,000.5318
`std::transform_reduce`	Unsequenced	(*) → (+)	131,677 us	7,598,000.5455
`std::transform_reduce`	Parallel-Unsequenced	(*) → (+)	51,095 us	7,598,000.5319

Alternative Algorithms Example – Exclusive Scan

Algorithm	Execution Policy	Binary Op	Time	Result
`std::partial_sum`	Serial	+	119,096 us	[0.1, 0.2, ..., 10,000,000.6, 10,000,000.7]
`std::exclusive_scan`	Sequential	+	143,338 us	[0.0, 0.1, ..., 10,000,000.6, 10,000,000.6]
`std::exclusive_scan`	Parallel	+	146,967 us	[0.0, 0.1, ..., 10,000,000.6, 10,000,000.6]
`std::exclusive_scan`	Unsequenced	+	140,900 us	[0.0, 0.1, ..., 10,000,000.6, 10,000,000.6]
`std::exclusive_scan`	Parallel-Unsequenced	+	145,098 us	[0.0, 0.1, ..., 10,000,000.6, 10,000,000.6]

Alternative Algorithms Example – Inclusive Scan

Algorithm	Execution Policy	Binary Op	Time	Result
`std::partial_sum`	Serial	+	121,801 us	[0.1, 0.2, …, 10,000,000.6, 10,000,000.7]
`std::inclusive_scan`	Sequential	+	120,705 us	[0.1, 0.2, …, 10,000,000.6, 10,000,000.7]
`std::inclusive_scan`	Parallel	+	150,662 us	[0.1, 0.2, …, 10,000,000.6, 10,000,000.7]
`std::inclusive_scan`	Unsequenced	+	120,440 us	[0.1, 0.2, …, 10,000,000.6, 10,000,000.7]
`std::inclusive_scan`	Parallel-Unsequenced	+	145,441 us	[0.1, 0.2, …, 10,000,000.6, 10,000,000.7]

Alternative Algorithms Example – Transform-Exclusive Scan

Algorithm	Execution Policy	Binary Op	Time	Result
`std::transform_exclusive_scan`	Sequential	(*2) → (+)	125,675 us	[0.0, 0.2, ..., 20,000,001.0, 20,000,001.2]
`std::transform_exclusive_scan`	Parallel	(*2) → (+)	150,095 us	[0.0, 0.2, ..., 20,000,001.0, 20,000,001.2]
`std::transform_exclusive_scan`	Unsequenced	(*2) → (+)	167,813 us	[0.0, 0.2, ..., 20,000,001.0, 20,000,001.2]
`std::transform_exclusive_scan`	Parallel-Unsequenced	(*2) → (+)	146,167 us	[0.0, 0.2, ..., 20,000,001.0, 20,000,001.2]

Alternative Algorithms Example – Transform-Inclusive Scan

Algorithm	Execution Policy	Binary Op	Time	Result
`std::transform_inclusive_scan`	Sequential	(*2) → (+)	120,220 us	[0.2, 0.4, ..., 20,000,001.2, 20,000,001.4]
`td::transform_inclusive_scan`	Parallel	(*2) → (+)	148,472 us	[0.2, 0.4, ..., 20,000,001.2, 20,000,001.4]
`td::transform_inclusive_scan`	Unsequenced	(*2) → (+)	135,489 us	[0.2, 0.4, ..., 20,000,001.2, 20,000,001.4]
`td::transform_inclusive_scan`	Parallel-Unsequenced	(*2) → (+)	150,443 us	[0.2, 0.4, ..., 20,000,001.2, 20,000,001.4]

Atomics

Atomics

DATA RACES & SHARED RESOURCES

- A data race is a condition in which shared data is being read from and written to by separate threads of execution simultaneous.
- This causes UB as we cannot meaningfully restrict which operation occurs first.
- To prevent race conditions, we use synchronization primitives to ensure that access to a resource is synchronized.

ATOMIC TYPES

- One type of synchronization primitive are atomic types.
- Atomic types are data that exists in a share memory space and can only be access atomically.
- Atomic operations are dictated by a `std::memory_order` object which controls how memory is access, allowing operations to atomics to be synchronized.
- C++ atomic type is `std::atomic<T>` where `T` can be integrals, floating-points, pointers and `std::shared_ptr<T>` or `std::weak_ptr<T>`.
- C++ also has `std::atomic_ref<T>` which creates an atomic reference to an existing object which becomes atomic for the lifetime of the atomic reference.

Atomics Example

```cpp
#include <algorithm>
#include <atomic>
#include <chrono>
#include <execution>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <utility>
#include <vector>

/// execution timer ...

auto main() -> int
{
    auto count = std::atomic<int>{ 0 };
    auto v = std::vector<double>(100'000'007, 0.1);

    auto alg = [&count](const auto& v)
    {
        return std::reduce(
            std::execution::par_unseq,
            v.begin(),
            v.end(),
            0.0,
            [&count](const auto& x, const auto& y) {
                count.fetch_add(1, std::memory_order_relaxed);
                return x + y;
            }
        );
    };

    std::cout.imbue(std::locale("en_US.UTF-8"));
    std::cout << std::fixed << std::setprecision(4);

    auto [time, result] = measure<>::execution(alg, v);

    std::cout << "std::reduce (parallel-unsequenced execution):\n"
              << "Result: " << result << "\n"
              << "Time: " << time << " us\n"
              << "Atomic Count: " << count.load() << std::endl;

    return 0;
}
```

Threads

Threads

- A thread is the smallest sequence of instructions that is managed by the OS scheduler.
- Threads are a sub-object of a process where a process is a running system or service.
- A process can multiple threads which are used to run parts of a process concurrently.
- Threads are spawned from a `std::thread` object which will run a function until completion and then must be rejoined to the main thread or dethatched.
- `std::thread` constructor takes a function as there first argument and any arguments that need to be forwarded to the job function.

```cpp
#include <atomic>
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

using namespace std::literals;

auto work = [](std::atomic_ref<int> counter)
{
    std::osyncstream(std::cout) << "Doing work on thread: "
                                << std::this_thread::get_id()
                                << " ...\n";
    auto count = counter.fetch_add(1, std::memory_order_relaxed);
    std::osyncstream(std::cout) << "Call count: " << count << "\n";
    std::this_thread::sleep_for(1.5s);
    std::osyncstream(std::cout) << "Thread "
                                << std::this_thread::get_id()
                                << " Done!\n";
};

auto main() -> int
{
    auto counter { 1 };
    auto atomic_counter = std::atomic_ref<int>{ counter };

    std::thread t1(work, atomic_counter);
    std::thread t2(work, atomic_counter);
    std::thread t3(work, atomic_counter);
    std::thread t4(work, atomic_counter);

    std::cout << "Waiting in main...\n";

    t1.join();
    t2.join();
    t3.join();
    t4.join();

    return 0;
}
```

Automatic Threads

- C++20 added support for automatic thread objects that will automatically join to main thread on their destruction called `std::jthread`.
- These thread objects also support the preemptive cancellation using stop tokens.
- The stop token (`std::stop_token`) of a `std::jthread` can be obtained by making the first parameter of the job function a `std::stop_token` object.
- Other threads can get a `std::stop_source` so that they can cancel the execution of a `std::jthread`.

```cpp
#include <atomic>
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>

using namespace std::literals;

auto job = [](std::stop_token tkn)
{
    for (auto i { 10 }; i; --i)
    {
        std::this_thread::sleep_for(150ms);

        if (tkn.stop_requested())
        {
            std::cout << "  The job has be requested to stop\n";
            return;
        }

        std::cout << " Continuing with job\n";
    }
};

auto stop_job = [](std::stop_source source)
{
    std::this_thread::sleep_for(500ms);
    std::cout << "Request stop for worker via source\n";
    source.request_stop();
};

auto main() -> int
{
    auto worker = std::jthread(job);

    std::cout << "\nPass source to other thread:\n";

    std::stop_source stop_source = worker.get_stop_source();
    auto stopper = std::thread(stop_job, stop_source);
    stopper.join();

    std::this_thread::sleep_for(250ms);

    return 0;
}
```

Thread Pool

- Thread pools are a common design pattern in Computer Science.
- The basic premise involves creating an array of idle thread objects that can be activated by pushing jobs to a queue.
- Idle threads will be assigned jobs until either there is no jobs left or no more idle threads.
- Once the threads are finished, the become idle again and wait for a new job.
- The easiest way to accomplish this in C++ is to use a vector of threads and use `std::vector::emplace_back` to construct a new thread with the given job in place in the vector and loop thread the threads and rejoin them to the main thread.

```cpp
#include <chrono>
#include <iostream>
#include <syncstream>
#include <thread>
#include <vector>

using namespace std::literals;

auto job = [](auto job_id)
{
    std::this_thread::sleep_for(150ms);
    std::osyncstream(std::cout) << "Thread: "
                                << std::this_thread::get_id()
                                << " is running job: "
                                << job_id
                                << "\n";
    std::this_thread::sleep_for(150ms);
};

auto main() -> int
{
    auto thr_count { std::thread::hardware_concurrency() };
    auto pool = std::vector<std::thread>(thr_count);

    /// Queue jobs
    for (auto i { 0u }; i < thr_count; ++i)
        pool.emplace_back(job, i);

    std::this_thread::sleep_for(200ms);

    /// Join all job threads
    for (auto& th : pool)
        if (th.joinable())
            th.join();

    return 0;
}
```

Mutexes & Locks

Mutex

- A mutex is a *mutually-exclusive-object* that is used as a synchronization mechanism for shared memory across multiple threads.
- Mutexes are acquired by trying to lock it and are released by unlocking.
- Locking operations will block the current thread until the lock can be acquired while try-locking will return a Boolean indicating the result of attempting to lock.
- C++ primitive mutex type is `std::mutex` and is often implemented from an OS or assembly primitive.
- Mutexes are non-copyable as well as non-moveable and are not RAII compliant.

```cpp
#include <chrono>
#include <iostream>
#include <map>
#include <mutex>
#include <sstream>
#include <thread>
#include <vector>

using namespace std::literals;

auto mx  = std::mutex{};
auto map = std::map<int, long long>{};

auto job = [](auto job_id)
{
    std::this_thread::sleep_for(150ms);
    auto ss = std::stringstream{};
    ss << std::this_thread::get_id();
    auto thread_id = std::stoll(ss.str());

    while (!mx.try_lock())
        std::this_thread::sleep_for(150ms);

    map.insert({ job_id, thread_id });
    mx.unlock();
    std::this_thread::sleep_for(150ms);
};

auto main() -> int
{
    auto thr_count { std::thread::hardware_concurrency() };
    auto pool = std::vector<std::thread>(thr_count);

    /// Queue jobs
    for (auto i { 0u }; i < thr_count; ++i)
        pool.emplace_back(job, i);

    std::this_thread::sleep_for(200ms);

    /// Join all job threads
    for (auto& th : pool)
        if (th.joinable())
            th.join();

    std::cout << "{ ";
    for (auto i { map.size() }; auto& [k, v] : map)
        std::cout << k << ": " << v << (i-- ? ", " : "");
    std::cout << " }" << std::endl;

    return 0;
}
```

Other Mutex Types

Mutex Type	Description
`std::timed_utex`	Mutex that offers timeout-based locking methods. Locking will be attempted for a certain duration.
`std::recursive_mutex`	Mutex that can be repeatedly locked by the same thread multiple times. Must be unlocked the same number of times to become fully unlocked.
`std::timed_recursive_mutex`	Recursive mutex with timeout locking.
`std::shared_mutex`	A mutex that offers to levels of access, *shared* or *exclusive*. Shared locking allows for multiple threads to share a mutex and read the shared memory resources while exclusive only allows one thread to access the shared resources with write privileges. If one thread has a shared lock an a mutex other threads can only gain a shared lock on it as well prohibiting the ability to gain exclusive access from another thread until all threads have unlocked the shared lock. Similarly, a thread with an exclusive lock on a thread disallows other threads from gaining any lock on the mutex until it has been unlocked.
`std::timed_shared_mutex`	Same as a `std::shared_mutex` but offers timeout based exclusive and shared locking.

Semaphores

- Locks are another type of synchronization primitive.
- The most basic lock is a semaphore.
- Semaphores are used to allow multiple threads to access the same shared resource.
- Accessors are dictated by a count which decrements with every acquisition of the semaphore and blocks any thread trying to acquire the semaphore when the count reaches zero.
- This is often called a counting semaphore which is also the case in C++ (`std::counting_semaphore`).
- C++ also has a specialization type which allows are single accessor called a binary semaphore (`std::binary_semaphore`).

```cpp
#include <chrono>
#include <iostream>
#include <semaphore>
#include <thread>

using namespace std::literals;

auto toMain     = std::binary_semaphore{ 0 };
auto fromMain   = std::binary_semaphore{ 0 };

auto work = []()
{
    fromMain.acquire();

    std::cout << "[thread]: Got signal" << std::endl;
    std::this_thread::sleep_for(3s);
    std::cout << "[thread]: Sent signal" << std::endl;

    toMain.release();
};

auto main() -> int
{
    auto th = std::thread{ work };

    std::cout << "[Main]: Sent signal" << std::endl;
    fromMain.release();
    toMain.acquire();
    std::cout << "[Main]: Got signal" << std::endl;

    th.join();

    return 0;
}
```

Locks Types

- The most common lock types in C++ are wrapper types that bind the locking lifetime of a `std::mutex` to the lifetime of the lock type.
- These lock types make the use of mutex RAII compliant, improving the safety of concurrent access to shared memory.
- The draw back to locks is there a bit of additional overhead in the creating and destruction of locks.

```cpp
#include <chrono>
#include <iostream>
#include <map>
#include <mutex>
#include <sstream>
#include <thread>
#include <vector>

using namespace std::literals;

auto mx  = std::mutex{};
auto map = std::map<int, long long>{};

auto job = [](auto job_id)
{
    std::this_thread::sleep_for(150ms);
    auto ss = std::stringstream{};
    ss << std::this_thread::get_id();
    auto thread_id = std::stoll(ss.str());

    /// Acquire a lock on mx that lasts for this scope
    {
        auto lk = std::lock_guard{ mx };
        map.insert({ job_id, thread_id });
    }

    std::this_thread::sleep_for(150ms);
};

auto main() -> int
{
    auto thr_count { std::thread::hardware_concurrency() };
    auto pool = std::vector<std::thread>(thr_count);

    /// Queue jobs
    for (auto i { 0u }; i < thr_count; ++i)
        pool.emplace_back(job, i);

    std::this_thread::sleep_for(200ms);

    /// Join all job threads
    for (auto& th : pool)
        if (th.joinable())
            th.join();

    std::cout << "{ ";
    for (auto i { map.size() }; auto& [k, v] : map)
        std::cout << k << ": " << v << (i-- ? ", " : "");
    std::cout << " }" << std::endl;

    return 0;
}
```

Lock Types

Mutex Type	Description
std::lock_guard	The most basic kind of mutex locking wrapper. It binds the locking lifetime of a mutex to the lifetime of the lock. It takes a template type parameter of the mutex type and a mutex as a constructor argument. It can also adopt the ownership of a mutex by passing a second constructor argument std::adopt_lock which does not lock the mutex but ensuring the calling thread will unlock it. std::lock_guard is non-copyable.
std::scoped_lock	A lock for acquiring ownership of zero or more mutexes for the duration of a scope block. When constructed and given ownership of multiple mutexes, the locking and unlocking of mutexes uses a deadlock avoidance algorithm.
std::unique_lock	Used to acquire an exclusive lock on a mutex with deferred, time-constrained, recursive and transfer semantics for locking. It is non-copyable but is moveable.
std::shared_lock	Used to gain shared access to a mutex with similar semantics to std::unique_lock. Used for locking a std::shared_lock in a shared ownership model.

Latches

- A `std::latch` is a count-down based synchronization primitive.
- The starting count of the latch is set at construction, and it cannot be reset, incrementing or changed after construction.
- It is a single use, non-copyable barrier type that is used to create a synchronization point between threads.
- Any threads must arrive at a latch and wait until all other threads arrive at the latch which is indicated by the count going to zero.

```cpp
#include <chrono>
#include <iostream>
#include <latch>
#include <syncstream>
#include <thread>
#include <vector>

using namespace std::literals;

auto thr_count  = std::thread::hardware_concurrency();
auto done       = std::latch{ thr_count };
auto cleanup    = std::latch{ 1 };

auto job = [](auto job_id)
{
    std::this_thread::sleep_for(2s);
    std::osyncstream(std::cout) << "Job " << job_id << " done.\n";
    done.count_down();
    cleanup.wait();
    std::osyncstream(std::cout) << "Job " << job_id << " cleaned up.\n";
};

auto main() -> int
{
    auto pool = std::vector<std::thread>(thr_count);

    std::cout << "Starting jobs...\n";
    for (auto i { 0u }; i < thr_count; ++i)
        pool.emplace_back(job, i);

    done.wait();
    std::cout << "All jobs done.\n";
    std::this_thread::sleep_for(200ms);
    std::cout << "\nStarting cleanup...\n";
    cleanup.count_down();
    std::this_thread::sleep_for(200ms);

    for (auto& th : pool)
        if (th.joinable())
            th.join();
    std::cout << "All jobs cleaned up.\n";

    return 0;
}
```

Barriers

- A std::barrier is a more general version of a std::latch.
- The lifetime of a barrier consists of one-or-more phases.
- The first phase is the synchronization phase which; similar to a latch, uses a count to make threads wait until all other threads arrive.
- Along with a count, std::barrier constructor can take an optional function object that will be run once all threads arrive at a barrier.
- After all threads arrive (and the optional call to the completion function), the threads are released. The barrier can later be reused to resynchronize threads.
- Threads can decrement the overall count of the barrier on arrival.
- A std::barrier object is non-copyable.

```cpp
#include <chrono>
#include <iostream>
#include <barrier>
#include <syncstream>
#include <string>
#include <thread>
#include <vector>

using namespace std::literals;

auto thr_count  = std::thread::hardware_concurrency();

auto on_completion = []() noexcept
{
    static auto message = "All jobs done.\nWorkers are at lunch before cleaning
up...\n"s;
    std::osyncstream(std::cout) << message;
    std::this_thread::sleep_for(3s);
    message = "All cleaned up.\n"s;
};

auto barrier = std::barrier{ thr_count, on_completion };

auto job = [](auto job_id)
{
    std::this_thread::sleep_for(2s);
    std::osyncstream(std::cout) << "Job " << job_id << " done.\n";
    barrier.arrive_and_wait();
    std::osyncstream(std::cout) << "Job " << job_id << " cleaned up.\n";
    barrier.arrive_and_wait();
};

auto main() -> int
{
    auto pool = std::vector<std::thread>(thr_count);

    std::cout << "Starting jobs...\n";
    for (auto i { 0u }; i < thr_count; ++i)
        pool.emplace_back(job, i);

    std::this_thread::sleep_for(200ms);

    for (auto& th : pool)
        if (th.joinable())
            th.join();

    return 0;
}
```

Async

Futures and Promises

- A std::promise<T> object stores are value or exception that is retrieved by a std::future<T> object.
- This represent a relationship of a function or task promising a value that will be available in the future.
- The std::future<T> object of a promised value must be obtained directly by the corresponding std::promise object.
- The caller of a function with a promised value can query, wait for and extract the value from the future but this can potentially block until the future is ready.
- The promised value is communicated via a shared memory space with only one input and one output endpoint.
- A std::future<T> cannot be copied but is can be shared using std::future<T>::share to create a std::shared_future<T> object which can be copied and thus the value can be access by multiple futures.

```cpp
#include <chrono>
#include <future>
#include <iostream>
#include <thread>
#include <utility>

using namespace std::literals;

auto job = [](std::promise<int>&& p, auto a, auto b)
{
    std::this_thread::sleep_for(3s);
    auto r = a + b;
    p.set_value(r);
    std::this_thread::sleep_for(3s);
};


auto main() -> int
{
    auto p = std::promise<int>{};
    auto f = p.get_future();

    auto th = std::thread(job, std::move(p), 4, 5);

    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "Waiting for job...\n";
    auto r          = f.get();
    auto finish     = std::chrono::high_resolution_clock::now();
    auto duration   = std::chrono::duration_cast<
                          std::chrono::milliseconds
                      >(finish - start).count();

    std::cout << "Result: " << r << std::endl;
    std::cout << "Took: " << duration << " ms" << std::endl;

    th.join();

    return 0;
}
```

Async

- In C++ you can create an asynchronously running function using std::async.
- This takes a std::launch policy, a function and any arguments that need to be forwarded to the function at invocation.
- The standard only supports two launch policies, std::launch::async which launches the job on another thread, and std::launch::deferred which will run the asynchronous function lazily, deferring its execution until the value is requested.
- A std::future object is return from std::async which can be queried, waited on or can be used to extract the future value.

```
template<std::random_access_iterator I, std::sentinel_for<I> S, std::movable A>
auto parallel_sum(I first, S last, A init) -> A
{
    auto middle = first + ((last - first) / 2);

    /// Launch async sum on last half of the values
    auto future = std::async(
        std::launch::async,
        parallel_sum<I, S, A>,
        middle,
        last,
        A{}
    );

    /// Sum first half of the range locally.
    auto result = parallel_sum(first, middle, init);

    /// Obtain the future and sum with the result
    return result + future.get();
}
```

Packaged Tasks

- The final way to create a future value is using a `std::packaged_task<R(Args...)>` object.
- This is used to wrap a function that will execute on another thread such that the functions return value will be captured as a promise and retrievable from a future.
- A `std::packaged_task<R(Args...)>` is move only.

```cpp
#include <chrono>
#include <future>
#include <iostream>
#include <thread>
#include <utility>

using namespace std::literals;

auto job = [](auto a, auto b)
{
    std::this_thread::sleep_for(150ms);
    auto r = a + b;
    std::this_thread::sleep_for(150ms);
    return r;
};

auto main() -> int
{
    auto pkg    = std::packaged_task<int(int, int)>{ job };
    auto f      = pkg.get_future();

    auto th = std::thread(std::move(pkg), 4, 5);

    auto start = std::chrono::high_resolution_clock::now();
    std::cout << "Waiting for job...\n";
    auto r          = f.get();
    auto finish     = std::chrono::high_resolution_clock::now();
    auto duration   = std::chrono::duration_cast<
                          std::chrono::milliseconds
                      >(finish - start).count();

    std::cout << "Result: " << r << std::endl;
    std::cout << "Took: " << duration << " ms" << std::endl;

    th.join();

    return 0;
}
```

Discussion

- Any questions?
- Need help?
- Open discussion.
- Concerns?

Thank You

Tyler Swann

https://github.com/MonashDeepNeuron/HPP