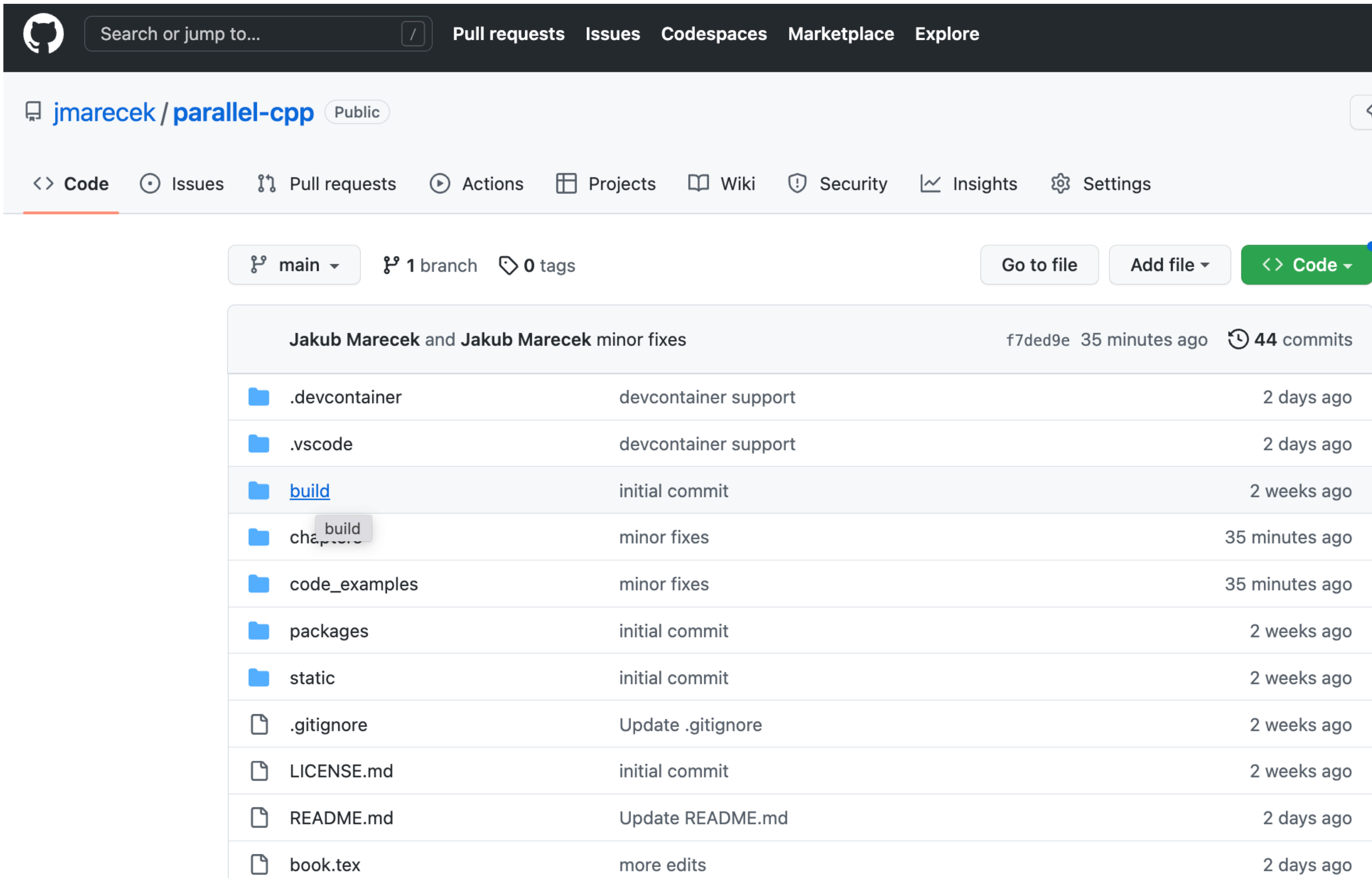
Paralelní a distribuované výpočty (B4B36PDV)

Jakub Mareček
jakub.marecek@fel.cvut.cz

Artificial Intelligence Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

What comes next?

What comes next?



What comes next?

- Structuring code
  - Thread
  - Jthread
  - Coroutines

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

Structuring code

Threads

- C++11 had a very basic support for threads, in terms of std::thread of header thread.
- The thread starts running once the constructor is called.
- The object is not CopyConstructible nor CopyAssignable.

The challenge in C++11 threads:

- one needs to call join or detach prior to the destructor being called. If neither was called, the program was std::aborted.
- Prior to calling either, one needs to check whether the thread is joinable().
- At the same time, it is almost impossible to handle exceptions while being able to call join correctly.
- The use of the C++11 thread is thus considered harmful and we will present only two short examples.

Structuring code

Threads

- The use of the C++11 thread is thus considered harmful and we will present only two short examples.

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

using namespace std::this_thread;
using namespace std::chrono_literals;

void A() {
    std::cout << "a";
    sleep_for(5s);
    std::cout << "A";
}

int main() {
    std::thread t(A);
    t.join();
}
```

Structuring code

Threads

An example of the use of a C++11 thread.

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

using namespace std::this_thread;
using namespace std::chrono_literals;

void A() {
    std::cout << "a";
    sleep_for(5s);
    std::cout << "A";
}

void B() {
    std::cout << "b";
    sleep_for(1s);
    std::cout << "B";
}

void C() {
    std::cout << "c";
    std::thread t(A);
    t.detach();
    std::thread u(B);
    u.join();
    std::cout << "C";
}

int main() {
    C();
    std::thread t(B);
    t.join();
    A();
}
```

Structuring code

C++20 jthread

- C++20 adds a new class jthread (``joining threads''), which does not require a call to join or detach. Instead, the destructor waits for completion of the code (``joins'') automatically.

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

void Hello();

int main(int argc, char* argv[]) {
    std::vector<std::jthread> threads;
    for (int cnt=0; cnt < 10; cnt++) {
        threads.push_back(std::jthread(Hello));
    }
    return 0;
}

void Hello() {
  using namespace std::chrono_literals;
  std::this_thread::sleep_for(2s);
}
```

This is an example of the ``resource acquisition is initialization'' idiom.
In RAII, the resource allocation is tied to an object's lifetime and is hence a class invariant.
In a constructor, one allocates the resources.
In a destructor, one releases the resources.
There is no risk of a resource leak.

- Notice that the example would very like result in abnormal program termination, if we changed jthread to thread. (Why?)

Structuring code

C++20 jthread

- When we use standard output, it is prudent to wrap it in a syncstream:

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

void Foobar(int cnt);

int main(int argc, char* argv[]) {
    std::vector<std::jthread> threads;
    for (int cnt=0; cnt < 10; cnt++) {
        threads.push_back(std::jthread(Foobar, cnt));
    }
    std::osyncstream(std::cout) << "Main thread" << std::endl;
    return 0;
}

void Foobar(int cnt) {
    std::osyncstream(std::cout) << "Thread " << cnt << std::endl;
}
```

Structuring code

C++20 jthread

- Rather commonly, one uses the lambda function to define the thread.

(This is the [](.)

```
#include <iostream>
#include <syncstream>
#include <thread>
using namespace std::chrono_literals;

int main() {

    auto t1 = std::jthread([](){
        std::osyncstream(std::cout) << "Another thread" << std::endl;
        std::this_thread::sleep_for(1s);
    });

    std::this_thread::sleep_for(2s);
    std::osyncstream(std::cout) << "Main thread" << std::endl;
}
```

Structuring code

C++20 jthread

- When we pass the first argument of type std::stop_token token, we request the thread to stop its execution by calling request_stop() on the jthread object:

```cpp
#include <iostream>
#include <syncstream>
#include <thread>
using namespace std::chrono_literals;

int main() {
    auto t1 = std::jthread([](std::stop_token token){
        while (!token.stop_requested()) {
            std::osyncstream(std::cout) << "A thread";
            std::this_thread::sleep_for(1s);
        }
        std::osyncstream(std::cout) << "Stop requested";
    });

    std::this_thread::sleep_for(2s);

    std::osyncstream(std::cout) << "Main thread";
    t1.request_stop();
}
```

Structuring code

C++20 jthread

```cpp
#include <iostream>
#include <syncstream>
#include <atomic>
#include <thread>
using namespace std::chrono_literals;

int main() {

    auto t = std::jthread([](std::stop_token token) {
        std::osyncstream(std::cout) << "Thread " << std::this_thread::get_id() << std::endl;
        std::atomic<bool> flag = false;
        std::stop_callback callback(token, [&flag]{
            std::osyncstream(std::cout) << "Stop requested" << std::endl;
            flag = true;
        });
        while (!flag) {
            std::this_thread::sleep_for(1s);
        }
    });

    std::osyncstream(std::cout) << "Main thread" << std::endl;
    std::this_thread::sleep_for(3s);
    t.request_stop(); // runs all callbacks!
}
```

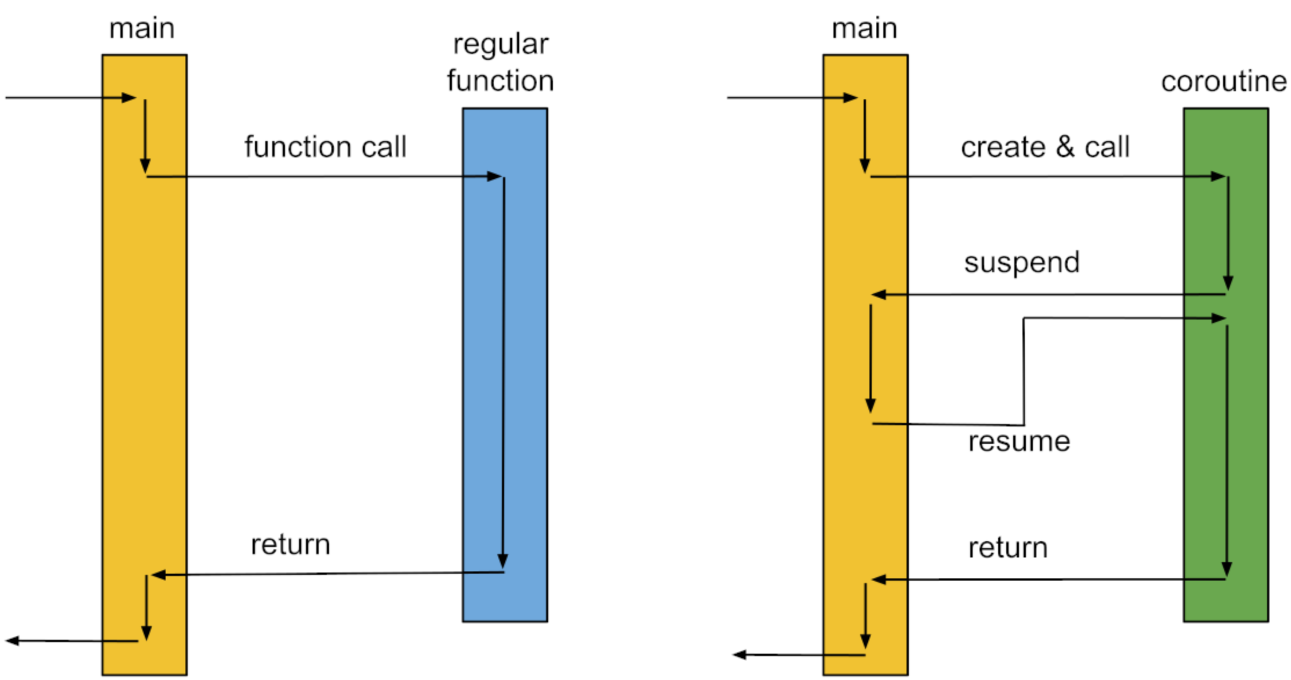
One can define std::stop_callback object inside the thread, whose constructor takes the stop token and a function.

The function gets executed, when the thread is requested to stop via the std::stop_token.

Structuring code

Coroutines

- Within a particular thread, one may utilize multiple **coroutines**, which can be seen as subroutines that can run in multiple steps, but sometimes can serve as a light-weight alternative to hardware threads.



https://blog.eiler.eu/posts/20210512/images/coroutines.png

Structuring code

Coroutines

- Coroutines can be called, can return when completed, but also can suspend themselves, yielding control and partial results, and be resumed by another co-routine.
- Typical uses involve generators and factories and various other concepts within ``lazy evaluation'', as well as event-driven architectures within cooperative multi-tasking.
- That is: two coroutines within one thread never run in parallel, but one can have the runs of two or more coroutines interleaved. We can suspend a co-routine in one thread and resume it within another thread.
- As it turns out, the ``context switch'' with user-level threads has a similar cost to a function call or suspending a coroutine (**co_yield**). Indeed, coroutines are typically implemented with user-level threads, which leads to cheaper context-switch compared with hardware threads. Within the user-level threads, one can distinguish stackful and stackless versions, where coroutine state is saved on the heap (as in C++).

Structuring code

Coroutines

In C++23 or 26, we hope to see some standard syntax for defining coroutines (cf. P2502), such as:

An example of the use of coroutines, which currently does not compile in GCC 12.2.

```
#include <coroutine>
#include <generator>
#include <iostream>
#include <syncstream>

std::generator<int> work() {
    for (int i = 0; i < 10; i++) {
        co_yield i;
    }
}

int main() {
    for (int i : work()) {
        std::osyncstream(std::cout) << ch << '\n';
    }
}
```

Structuring code

Coroutines

In terms of using the coroutine, there are three new keywords:

- **co_await** awaiter suspends computation and block the co-routine until the computation is resumed by another co-routine calling ``resume'' method of the present coroutine. In the process, it tests whether it is possible to suspend the computation using an awaiter such as std::suspend_always (or an awaitable object, more generally, as discussed below) and, if so, saves all local variables to a heap-allocated handle.
- **co_yield** yields a value and suspends computation as above, and
- **co_return** returns a value. (There is no notion of an optional return type in-built.)

Structuring code

Coroutines

A difficulty in using coroutines is the fact that **the coroutine may live longer than the scope it has been called from.** It is hence not advisable to pass by reference, except perhaps std::ref or std::cref.

One can either pass by value or pass, e.g., std::unique_ptr:

```
std::generator<char> split-by-value(std::string s) {
    for (char ch : ps) {
        co_yield ch;
    }
}

std::generator<char> split-by-uniqueptr(std::unique_ptr<std::string> ps) {
    for (char ch : *ps) {
        co_yield ch;
    }
}
```

Structuring code

Coroutines

Unfortunately, defining the coroutine in C++20 take some more effort.

In particular, it requires:

- defining the behaviour of the coroutine, which is known as a **promise** (different from std::promise), and requires one returns the type used to access the state of the coroutine on the heap, which is known as the handle,
- defining how to store the state of the coroutine on the heap, using template class **std::coroutine_handle** parametrized by the promise

Clearly, one needs to declare one, define the other, and then return to declare the first one. We will see how to do this later.

Optionally, we can also define an **awaiter**, which controls suspension and resumption behaviour.

Structuring code

Coroutines

First, we need to be able to define a promise class, which defines the behaviour of the coroutine by implementing methods:

- coroutine **get_return_object()** is called to inialize the coroutine and create the coroutine handle
- std::suspend_always **initial_suspend()**, suggests whether the coroutine starts right after initialization
  std::suspend_always **final_suspend()** noexcept, which can be rather formulaic std::suspend_always()
- void **return_void()** or void **return_value**(const auto& value), which is called upon reaching the end of the coroutine and upon reaching **co_return**. The latter (return_value) often just stores the result locally.
- void **unhandled_exception()**, which can be rather formulaic std::terminate(), or can save the exception via std::current_exception().

Structuring code

Coroutines

The promise class is instantiated for each instance of the coroutine, and its methods are called as follows:

```
{
  co_await promise.initial_suspend();
  try {
    ...
  }
  catch (...) {
    promise.unhandled_exception();
  }
  // finally
  co_await promise.final_suspend();
}
```

Structuring code

Coroutines

```
struct promise;

struct coroutine : std::coroutine_handle<promise> {
    using promise_type = struct promise;
};

struct promise {
    coroutine get_return_object() { return {coroutine::from_promise(*this)}; }
    std::suspend_always initial_suspend() noexcept { return {}; }
    std::suspend_always final_suspend() noexcept { return {}; }
    void return_void() {}
    void unhandled_exception() {}
};

int main() {
    coroutine h = [](int i) -> coroutine {
        std::osyncstream(std::cout) << i;
        co_return;
    }(0);
    h.resume();
    h.destroy();
}
```

Structuring code

Awaiters

Finally, let us consider awaiters, which can be called when a coroutine is suspended or resumed.

Key methods of an awaiter include:

- await_ready() is called immediately before suspension of a coroutine. If it returns true, the coroutine will not be suspended.
- await_suspend(handler) is called immediately after the suspension of the coroutine. The handler of type std::coroutine_handle can be used to pass the state of the coroutine (e.g., to another thread).
- await_resume() is called when the coroutine is resumed after a successful suspension. If it returns a value, this will be returned by the co_await routine.

The awaiters we have seen so far (std::suspend_never() and std::suspend_always()) returned boolean constants in await_ready()

Structuring code

Awaiters

By defining await_transform() in the promise type, the compiler will use co_await promise.await_transform(<expr>) instead of any call of co_await <expr> in the coroutine.

```
struct suspend_always
{
  constexpr bool await_ready() const noexcept { return false; }
  constexpr void await_suspend(coroutine_handle<>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};

struct suspend_never
{
  constexpr bool await_ready() const noexcept { return true; }
  constexpr void await_suspend(coroutine_handle<>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};
```

Structuring code

Our Own Generator

```
Generator myCoroutine() {
    int x = 0;
    while (true) {
        co_yield x++;
    }
}
```

Structuring code

Our Own Generator

```
// The caller-level type
struct Generator {
    // The coroutine level type
    struct promise_type {
        using Handle = std::coroutine_handle<promise_type>;

        Generator get_return_object() {
            return Generator{Handle::from_promise(*this)};
        }
        std::suspend_always initial_suspend() { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        std::suspend_always yield_value(int value) {
            current_value = value;
            return {};
        }
        void unhandled_exception() { }
        int current_value;
    };
```

Structuring code

Our Own Generator

```
explicit Generator(promise_type::Handle coro) : coro_(coro) {}
// Make move-only
Generator(const Generator&) = delete;
Generator& operator=(const Generator&) = delete;
Generator(Generator&& t) noexcept : coro_(t.coro_) { t.coro_ = {}; }
Generator& operator=(Generator&& t) noexcept {
    if (this == &t) return *this;
    if (coro_) coro_.destroy();
    coro_ = t.coro_;
    t.coro_ = {};
    return *this;
}

int get_next() {
    coro_.resume();
    return coro_.promise().current_value;
}
```

Structuring code

Our Own Message-Passing

```cpp
class Event {
 public:

    Event() = default;

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

    class Awaiter;
    Awaiter operator co_await() const noexcept;

    void notify() noexcept;

 private:
    friend class Awaiter;
    mutable std::atomic<void*> suspendedWaiter{nullptr};
    mutable std::atomic<bool> notified{false};

};
```

Structuring code

Our Own Message-Passing

```
class Event::Awaiter {
 public:
  Awaiter(const Event& eve): event(eve) {}

  bool await_ready() const;
  bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
  void await_resume() noexcept {}

 private:
    friend class Event;

    const Event& event;
    std::coroutine_handle<> coroutineHandle;
};
```

Structuring code

Our Own Message-Passing

```
struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};
```

What comes next?

- Structuring code
  - Thread
  - Jthread
  - Coroutines

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge

Synchronisation Primitives

Atomic Variables

Since C++11, there is an excellent support for atomic variables in header <atomic>. The primary template can be instantiated with types that are TriviallyCopyable, CopyConstructible, and CopyAssignable.

```
#include <iostream>
#include <atomic>
#include <thread>

std::atomic<int> i(0);

int main() {
    auto t1 = std::jthread([](){
        int j;
        do { j = i; }
        while (j == 0);
        std::cout << j << std::endl;
    });
    auto t2 = std::jthread([](){
        i = 1;
    });
    return 0;
}
```

Synchronisation Primitives

Atomic Variables

Since C++11, there is an excellent support for atomic variables in header <atomic>. The primary template can be instantiated with types that are TriviallyCopyable, CopyConstructible, and CopyAssignable.

```
#include <iostream>
#include <atomic>
#include <thread>

std::atomic<int> i(0);

int main() {
    auto t1 = std::jthread([](){
        int j;
        do { j = i.load(std::memory_order_relaxed); }
        while (j == 0);
        std::cout << j << std::endl;
    });
    auto t2 = std::jthread([](){
        i.store(1, std::memory_order_relaxed);
    });
    return 0;
}
```

Synchronisation Primitives

Atomic Variables

```
#include <atomic>
#include <stack>

template<typename T>
struct Node {
  T data;
  Node* next;
  Node(const T& data) : data(data), next(nullptr) {}
};

template<typename T> class stack {
    std::atomic<Node<T>*> head;
    public:
    void push(const T& data) {
        Node<T>* new_node = new Node<T>(data);
        new_node->next = head.load(std::memory_order_relaxed);
        while(!head.compare_exchange_weak(new_node->next,
        ↪   new_node, std::memory_order_release,
        ↪   std::memory_order_relaxed));
    }
};

int main() {
    std::stack<int> s; s.push(1); s.push(2); s.push(3);
}
```

Synchronisation Primitives

Barrier

- Since C++20, there is support for barriers in header <barrier>.
- The constructor takes an integer value, which is the number of threads that the barrier is expected to block.
- **arrive_and_wait():** blocking wait until the number of threads arrive at the same spot
- **arrive_and_drop():** decrements the initial expected count for all uses by one, as if one thread could never reach the barrier subsequently. This can be very useful in error management.

```cpp
#include <barrier>
#include <syncstream>
#include <iostream>
#include <vector>
#include <thread>
#include <algorithm>
#include <random>

int main() {
    std::barrier b(5);
    std::vector<std::jthread> ts;
    std::generate_n(std::back_inserter(ts), 5, [&b]{
        return std::jthread([&b]{
            std::mt19937 gen(std::hash<std::thread::id>{}(std::this_threa
            std::bernoulli_distribution d(0.3);
            int cnt = 1;
            while (true) {
                std::osyncstream(std::cout) << std::this_thread::get_id() << "/" << cnt << std::endl;
                std::this_thread::yield();
                if (d(gen)) {
                    b.arrive_and_drop();
                    return;
                } else {
                    b.arrive_and_wait();
                }
                cnt++;
            }
        });
    });
}
```

Synchronisation Primitives

Barrier

More complicated uses of barriers may use the template parameter CompletionFunction and have a callable executed whenever the barrier is hit (reaches zero):

```
    std::barrier b(4,[id = 1]() mutable noexcept {
        std::osyncstream(std::cout) << id << " OK" << std::endl;
        id++;
    });
    std::vector<std::jthread> runners;

    std::generate_n(std::back_inserter(runners), 4, [&b]{
        return std::jthread([&b]{
            std::osyncstream(std::cout) << std::this_thread::get_id() << "/1" << std::endl;
            std::this_thread::yield();
            b.arrive_and_wait();
            std::osyncstream(std::cout) << std::this_thread::get_id() << "/2" << std::endl;
            std::this_thread::yield();
            b.arrive_and_wait();
        });
    });
    runners.clear();
    std::osyncstream(std::cout) << std::endl;
}
```

Synchronisation Primitives

Mutexes and Locks

- Standard Template Library in header <mutex> provides multiple mutexes (of type BasicLockable that implement lock and unlock methods): **mutex**, recursive_mutex, timed_mutex, recursive_timed_mutex, and unique_lock.

- A good practice for the use of mutexes is to lock them via the RAII idiom. Since C++11, this is available as std::unique_lock and std::lock_guard, and since C++17 scoped_lock in header <mutex>.

- Crucially, using **scoped**_lock provides the ability to lock multiple mutexes at once, avoiding deadlock.

- One may hence advise to use one or more **mutex** with a **scoped_lock** on top.

Synchronisation Primitives

Mutexes and Locks

```
int main() {
    using namespace std::chrono_literals;
    struct Shared {
        int value;
        std::mutex mux;
    };
    Shared shared{0,{}};
    auto t1 = std::jthread([&shared]{
        std::this_thread::sleep_for(1s);
        for (int i = 0; i < 10; i++) {
            std::this_thread::yield();
            {
                std::unique_lock lock(shared.mux);
                shared.value += 10;
            } // mutex unlocks!
            std::this_thread::sleep_for(1s);
        }
    });
}
```

Synchronisation Primitives

Mutexes and Locks

```
int main() {
    using namespace std::chrono_literals;
    struct Shared {
        int value;
        std::mutex mux;
    };
    Shared shared{0,{}};
    auto t = std::jthread([&shared]{
        std::this_thread::sleep_for(1s);
        for (int i = 0; i < 10; i++) {
            {
               std::unique_lock lock(shared.mux);
               shared.value += 1;
            }
            std::this_thread::sleep_for(1s);
        }
    });
    auto observer = std::jthread([&shared]{
        while (true) {
            {
            std::unique_lock lock(shared.mux);
            std::cout << shared.value << std::endl;
            if (shared.value == 10)
                break;
            }
            std::this_thread::sleep_for(1s);
        }
    });
}
```

Algorithms in the Standard Template Library

For each

Since C++17, there is an excellent Parallel Standard Template Library in header <algorithm>.

The most useful algorithm from the Standard Template Library (STL) in terms of parallel programming is surely for each. As in the serial version of STL, the callable within for each is permitted to change the state of elements, if the underlying range is mutable, but cannot invalidate iterators.

```cpp
struct Custom {
    void expensive_operation() {
        // ...
    }
};

std::vector<Custom> data(10);

std::for_each(std::execution::par_unseq,
    data.begin(), data.end(),
    [](Custom& el) {
        el.expensive_operation();
    });
```

Algorithms in the Standard Template Library

Reduce

- Similarly useful is the reduce operation (also known as fold, accumulate, aggregate, compress, or inject).
- In Map Reduce, one applies an associative operation to each piece of data to obtain a partial result, and then obtains the final result by applying the same associative operation to the partial results.
- The binary-tree reduction makes it possible to utilize O(log(n)) rounds of computation on n processors.

```cpp
std::vector<int> data{1, 2, 3, 4, 5};

auto sum = std::reduce(data.begin(), data.end(), 0);
// sum == 15

sum = std::reduce(std::execution::par_unseq,
    data.begin(), data.end(), 0);
// sum == 15

auto product = std::reduce(data.begin(), data.end(), 1,
    std::multiplies<>{});
// product == 120

product = std::reduce(std::execution::par_unseq,
    data.begin(), data.end(), 1, std::multiplies<>{});
// product == 120
```

Algorithms in the Standard Template Library

Merge

Finally, in implementing parallel sorting algorithms, we will utilize the parallel merge operation:

```
std::vector<int> data1{1, 2, 3, 4, 5, 6};
std::vector<int> data2{3, 4, 5, 6, 7, 8};

std::vector<int> out(data1.size()+data2.size(), 0);
std::merge(std::execution::par_unseq,
    data1.begin(), data1.end(),
    data2.begin(), data2.end(),
    out.begin());
// out == {1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 8}
```

What we have seen?

- Structuring code
  - Thread
  - Jthread
  - Coroutines

- Atomic variables
- Mutexes and locks
- Barrier

- For each
- Reduce
- Merge