Concurrency, Parallelism and Coroutines

Anthony Williams

Just Software Solutions Ltd
https://www.justsoftwaresolutions.co.uk

29th April 2017

Concurrency, Parallelism and Coroutines

- Parallelism in C++17
- The Coroutines TS
- The Concurrency TS
- Coroutines and Parallel algorithms
- Executors

Aside: TS namespace

The TS's provides functions and classes in the `std::experimental` namespace.

In the slides I'll use `stdexp` instead, as it's shorter.

```
namespace stdexp=std::experimental;
```

Parallelism in C++17

Parallelism in C++17

C++17 provides a new set of overloads of the standard library algorithms with an **execution policy** parameter:

```
template<typename ExecutionPolicy,
  typename Iterator,
  typename Function>
void for_each(
  ExecutionPolicy&& policy,
  Iterator begin,Iterator end,
  Function f);
```

Execution Policies

The **execution policy** may be:

`std::execution::seq`
: Sequential execution on the calling thread

`std::execution::par`
: Indeterminately sequenced execution on unspecified threads

`std::execution::par_unseq`
: Unsequenced execution on unspecified threads

Plus any implementation-defined policies.

Supported algorithms

The vast majority of the C++ standard algorithms are parallelized:

`adjacent_find` `all_of` `any_of` `copy_if` `copy_n` **`copy`** `count_if` **`count`** `equal` `exclusive_scan` `fill_n` `fill` `find_end` `find_first_of` `find_if_not` `find_if` **`find`** `for_each_n` **`for_each`** `generate_n` `generate` `includes` `inclusive_scan` `inplace_merge` `is_heap` `is_heap_until` `is_partitioned` `is_sorted_until` `is_sorted` `lexicographical_compare` `max_element` **`merge`** `min_element` `minmax_element` `mismatch` `move` `none_of` **`nth_element`** `partial_sort_copy` `partial_sort` `partition_copy` `partition` **`reduce`** `remove_copy_if` `remove_copy` `remove_if` `remove` `replace_copy_if` `replace_copy` `replace` `replace_if` `reverse_copy` `reverse` `rotate_copy` `rotate` `search_n` `search` `set_difference` `set_intersection` `set_symmetric_difference` `set_union` **`sort`** `stable_partition` `stable_sort` `swap_ranges` **`transform`** `transform_inclusive_scan` `transform_exclusive_scan` **`transform_reduce`** `uninitialized_copy_n` `uninitialized_copy` `uninitialized_fill_n` `uninitialized_fill` `unique_copy` `unique`

Using Parallel algorithms

Just add an execution policy:

```
std::sort(std::execution::par,
          range.begin(),range.end());
```

It is up to you to ensure thread safety.

Thread Safety for Parallel Algorithms

`std::execution::seq`

No additional thread-safety requirements

`std::execution::par`

Applying operations on separate objects must be thread-safe

`std::execution::par_unseq`

Operations must be thread-safe and not need any synchronization; may be interleaved, and may switch threads.

Parallel Algorithms and Exceptions

Throwing an exception in a parallel algorithm will call `std::terminate`.

This applies for all 3 standard execution policies (even `std::execution::seq`).

Implementation provided extension policies may provide different behaviour.

Parallelism made easy!

"Just" add `std::execution::par` as the first parameter to standard algorithm calls.

```
std::sort(std::execution::par,v.begin(),v.end());
std::transform(
  std::execution::par,
  v.begin(),v.end(),v2.begin(),process);
```

However, as with all optimizations: **measure**. Parallelism has overhead, and some things are not worth parallelizing.

Technical Specification for C++ Extensions for Coroutines

What is a Coroutine?

A **coroutine** is a function that can be **suspended** mid execution and **resumed** at a later time.

Resuming a coroutine continues from the suspension point; local variables have their values from the original call.

Stackful vs Stackless coroutines

Stackful coroutines
: The entire call stack is saved

Stackless coroutines
: Only the locals for the current function are saved

The Coroutines TS only provides **stackless** coroutines.

Advantages of Stackless Coroutines

- Everything is localized
- Minimal memory allocation — can have millions of in-flight coroutines
- Whole coroutine overhead can be eliminated by the compiler — Gor's "disappearing coroutines"

Disadvantages of Stackless Coroutines

- Can only suspend coroutines — using `co_await` means the current function must be a coroutine
- Can only suspend current function — suspension returns to caller rather than suspending caller too

`co_`keywords make coroutines

A coroutine is a function that:

- contains at least one expression using one of the `co_await`, `co_yield`, or `co_return` keywords, and
- returns a type with corresponding **coroutine promise**.

`co_keywords`

`co_return` *some-value*

Return a final value from the coroutine

`co_await` *some-awaitable*

Suspend this coroutine if the **awaitable** expression is not **ready**

`co_yield` *some-value*

Return an intermediate value from the coroutine; the coroutine can be reentered at the next statement.

Promises and Awaitables

A **coroutine promise** type is a class that handles creating the return value object from a coroutine, and suspending the coroutine.

An **awaitable** type is something that a coroutine can wait for with `co_await`.

Often, **awaitable**s will have corresponding **coroutine promise**s, so you can return them from a coroutine.

Simple Coroutines

```
future<int> simple_return(){
  co_return 42;
}

generator<int> make_10_ints(){
  for(int i=0;i<10;++i)
  {
    co_yield i;
  }
}
```

Waiting for others

```
future<remote_data>
async_get_data(key_type key);

future<data> retrieve_data(
  key_type key){
  auto rem_data=
    co_await async_get_data(key);
  co_return process(rem_data);
}
```

Consuming generators

```
generator<int> make_10_ints();

void not_a_coroutine(){
  for(auto& x:make_10_ints()){
    do_stuff(x);
  }
}
```

Coroutines and parallel algorithms

Stackless coroutines work best if **everything** is a coroutine.

Implementations can use a custom execution policy to make parallel algorithms coroutines.

```
auto f=std::for_each(
  parallel_as_coroutine,
  v.begin(),v.end(),do_stuff);
co_await f;
```

Technical Specification for C++ Extensions for Concurrency

Concurrency TS v1

- Continuations for futures
- Waiting for one or all of a set of futures
- Latches and Barriers
- Atomic Smart Pointers

Continuations and `stdexp::future`

- A continuation is a new task to run when a future becomes ready
- Continuations are added with the new `then` member function
- Continuation functions must take a `stdexp::future` as the only parameter
- The source future is no longer `valid()`
- Only one continuation can be added

Continuations and `stdexp::future`

```
stdexp::future<int> find_the_answer();
std::string process_result(
  stdexp::future<int>);

auto f=find_the_answer();
auto f2=f.then(process_result);
```

Exceptions and continuations

```
stdexp::future<int> fail(){
  return stdexp::make_exceptional_future(
    std::runtime_error("failed"));
}
void next(stdexp::future<int> f){
  f.get();
}
void foo(){
  auto f=fail().then(next);
  f.get();
}
```

Wrapping plain function continuations: lambdas

```
stdexp::future<int> find_the_answer();
std::string process_result(int);

auto f=find_the_answer();
auto f2=f.then(
  [](stdexp::future<int> f){
    return process_result(f.get());
  });
```

Wrapping plain function continuations: `unwrapped`

```
template<typename F>
auto unwrapped(F f){
    return [f=std::move(f)](auto fut){
                return f(fut.get());
            };
}
```

```
stdexp::future<int> find_the_answer();
std::string process_result(int);
```

```
auto f=find_the_answer();
auto f2=f.then(unwrapped(process_result));
```

No `stdexp::async` — simple substitute

```
template<typename Func>
auto spawn_async(Func func){
  stdexp::promise<
    decltype(std::declval<Func>()())> p;
  auto res=p.get_future();
  std::thread t(
    [p=std::move(p),f=std::move(func)]()
      mutable{
      p.set_value_at_thread_exit(f());
    });
  t.detach();
  return res;
}
```

Continuations and `stdexp::shared_future`

- Continuations work with `stdexp::shared_future` as well
- The continuation function must take a `stdexp::shared_future`
- The source future remains `valid()`
- Multiple continuations can be added

stdexp::shared_future continuations

```
stdexp::future<int> find_the_answer();
void next1(stdexp::shared_future<int>);
int next2(stdexp::shared_future<int>);

auto fi=find_the_answer().share();
auto f2=fi.then(next1);
auto f3=fi.then(next2);
```

Waiting for the first future to be ready

`when_any` waits for the first future in the supplied set to be ready. It has two overloads:

```
template<typename ... Futures>
stdexp::future<stdexp::when_any_result<
std::tuple<Futures...>>>
when_any(Futures... futures);

template<typename Iterator>
stdexp::future<stdexp::when_any_result<
std::vector<
    std::iterator_traits<Iterator>::
      value_type>>>
when_any(Iterator begin,Iterator end);
```

when_any

`when_any` is ideal for:

- Waiting for speculative tasks
- Waiting for first results before doing further processing

```
auto f1=foo();
auto f2=bar();
auto f3=when_any(
  std::move(f1),std::move(f2));
f3.then(baz);
```

Waiting for all futures to be ready

`when_all` waits for all futures in the supplied set to be ready. It has two overloads:

```
template<typename ... Futures>
stdexp::future<std::tuple<Futures...>>
when_all(Futures... futures);

template<typename Iterator>
stdexp::future<std::vector<
    std::iterator_traits<Iterator>::
      value_type>>
when_all(Iterator begin,Iterator end);
```

`when_all`

`when_all` is ideal for waiting for all subtasks before continuing. Better than calling `wait()` on each in turn:

```
auto f1=spawn_async(subtask1);
auto f2=spawn_async(subtask2);
auto f3=spawn_async(subtask3);
auto results=when_all(
  std::move(f1),std::move(f2),
  std::move(f3)).get();
```

Coroutines and Continuations

Combining `stdexp::future` and coroutines

Futures ideally suited for coroutines:

- They hold a value
- You can wait on them
- They can represent asynchronous tasks
- You can create a future that holds a value

Returning `stdexp::future` from coroutines

To return a future, you need to specialize `stdexp::coroutine_traits` to provide a `promise_type`:

```
template <typename T>
struct coroutine_future_promise;

template <typename T, typename... Args>
struct stdexp::coroutine_traits<
stdexp::future<T>, Args...> {
  using promise_type=coroutine_future_promise<T>;
};
```

Returning `stdexp::future` from coroutines — coroutine promise

3 parts to it:

- Creating the return object
- Specifying whether to suspend before/after coroutine execution
- Setting the value

Waiting for futures in a coroutine

Waiting requires:

- Suspending the coroutine
- Telling the runtime how to resume the coroutine
- Obtaining the value when the coroutine is resumed

We have to tell the compiler how to do this for futures.

Waiting for futures — specifying the **awaiter**

Overload `operator co_await` to return an **awaiter** that provides customizations for our type:

```
template<typename T>
struct future_awaiter;

template<typename T>
auto operator co_await(stdexp::future<T>& f) {
    return future_awaiter<T>{f};
}
```

Future unwrapping and coroutines

If futures work with coroutines, you can use a coroutine as a continuation:

```
stdexp::future<result> my_coroutine(
    stdexp::future<data> x){
      auto res=co_await do_stuff(x.get());
      co_return res;
}

stdexp::future<result> foo(){
  auto f=spawn_async(make_data);
  return f.then(my_coroutine);
}
```

Coroutines and Parallel Algorithms

Parallel algorithms and blocking

For parallelism, we care about **processor utilization**.

Blocking operations hurt:

- They complicate scheduling
- They occupy a thread
- They force a context switch

Parallel Algorithms and blocking: Coroutines to the rescue

Coroutines allow us to turn blocking operations into non-blocking ones:

- `co_await` suspends current coroutine
- Coroutine can be automatically resumed **when the waited-for thing is ready**
- Current thread can process another task

Parallel Algorithms and stackless coroutines

If the suspension is in a nested call, a **stackless** coroutine wait just moves the blocking up a layer.

`f()` ⇒ `g()` ⇒ `h()`

If `h()` uses `co_await` to wait for a result, execution resumes in `g()`, which will then need to wait (and block) for the result of `h()`, and so on.

Parallel Algorithms and stackless coroutines

Solution: **Everything in the call stack must be a coroutine**

```
future<low_result> h(){
  co_return process(co_await get_data());
}
future<mid_result> g(){
  co_return process(co_await h());
}
future<result> f(){
  co_return process(co_await g());
}
```

Parallel Algorithms and stackless coroutines

Parallel algorithms with coroutines can then look like this:

```
future<result> parallel_func(data_type data){
  auto divided_data=
    co_await parallel_divide(data);
  auto res1=
    co_await parallel_func(divided_data.first);
  auto res2=
    co_await parallel_func(divided_data.second);
  auto final_result=
    co_await parallel_combine(res1,res2);
  co_return final_result;
}
```

Executors

Grand Unified Executors

19 executors papers going back to 2012

Much discussion

Added to Concurrency TS working draft and then removed

What is an executor?

This is the core issue: different use cases lead to different approaches

Fundamental answer: something that controls the execution of tasks.

Tasks?

- What kind of tasks?
- Where should they run?
- What relationships are there between tasks?
- Can tasks synchronize with each other?
- Can they run concurrently?
- Can they run interleaved?
- Can they migrate between threads?
- Can they spawn new tasks?
- Can they wait for each other?

Other questions

- Are executors copyable?
- Are they composable?
- Can you get an executor from a task handle?
- Can you get the executor for the currently-running task?

We want **All The Things**!

We want an executor **framework** that allows **all possible answers** to these questions.

Individual executors will provide **specific answers** to the questions.

Current Proposal

P0433R1: A Unified Executors Proposal for C++

MANY customization points

Basic executor

The basic requirements are simple. Executors must:

- be **CopyConstructible**,
- be **EqualityComparable**,
- provide a `context()` member function, and
- provide an `execute(f)` member function or `execute(e,f)` free function.

The framework can build everything else from there.

Execution Semantics

Three basic functions for executing tasks with an executor:

`execute(e,f)`
: Execute `f` with `e`. May or may not block current task.

`post(e,f)`
: Queue `f` for execution with `e` ASAP, without blocking current task.

`defer(e,f)`
: If currently running a task on `e`, queue `f` for execution with `e` **after current task has finished**. Otherwise, same as `post(e,f)`.

Returning values

`sync_execute(e,f)`
: Execute `f` with `e`. Blocks until `f` completes, returns result of invoking `f`.

`async_post(e,f)`
: Execute `f` with `e` like `post(e,f)`. Returns a `future` which will hold the return value of `f`.

`async_defer(e,f)`
: Execute `f` with `e` like `defer(e,f)`. Returns a `future` which will hold the return value of `f`.

Bulk execution and more

There are also functions to allow queuing multiple functions at once, and for scheduling continuations on executors.

Executors, Parallel Algorithms and Continuations

Implementations can provide an `ExecutionPolicy` for executors. e.g.

```
std::sort(on_executor(e),
  v.begin(),v.end());
```

It's also natural to do the same for continuations:

```
f.then(on_executor(e),my_func);
```

Availability

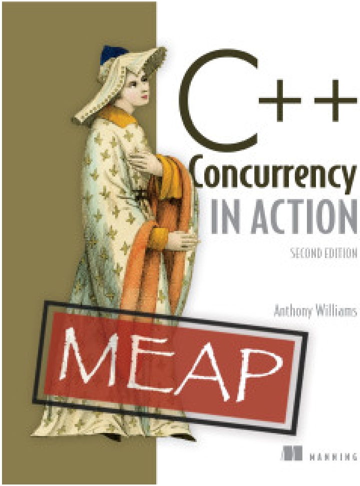
No shipping implementations provide all of these.

Visual Studio 2015 implements the coroutines TS.
Clang has a coroutines TS implementation in the works.

HPX provides parallel algorithms and futures with continuations from the Concurrency TS, as well as some executor support (but not the same as P0433R1).

Just::Thread Pro provides the Concurrency TS for gcc/clang/Visual Studio. Next version will have coroutines integration and parallel algorithms.

My Book



C++ Concurrency in Action:
Practical Multithreading,
Second Edition

Covers C++17 and the Concurrency TS

Early Access Edition now available

`http://stdthread.com/book`

Just::Thread Pro

`just::thread` Pro provides an actor framework, a concurrent hash map, a concurrent queue, synchronized values and a complete implementation of the C++ Concurrency TS, including a lock-free implementation of `atomic_shared_ptr`.

`http://stdthread.co.uk`

Questions?