Document No.	P1316R0
Date	2018-10-08
Reply To	Lewis Baker <lbaker@fb.com>
Audience	SG1, LEWG

A `when_all()` operator for coroutines

Abstract

This paper proposes to introducing two new standard library functions for introducing structured concurrency by allowing a coroutine to concurrently `co_await` multiple `Awaitable`[1] objects.

The first function is a variadic `when_all()` which allows you to concurrently await a number of `Awaitable` arguments and obtain a `std::tuple` of their individual resulting values.

The second function is a variadic `when_all_ready()` which also allows you to concurrently await a number of `Awaitable` arguments but instead of producing a std::tuple of the resulting values, produces a `std::tuple` of result objects that allow the caller to inspect the success/failure of individual operations. This allows code to better handle partial failure cases at the cost of a slightly more cumbersome interface.

```
task<A> get_a();
task<B> get_b();

task<> when_all_example()
{
  // Concurrently co_await get_a() and get_b() tasks.
  auto [a, b] = co_await std::when_all(get_a(), get_b());
  // Use a and b results ...

  // Also concurrently co_await get_a() and get_b() but allow
  // checking for partial failure.
  auto [aa, bb] = co_await std::when_all_ready(get_a(), get_b());
```

[1] See P1288R0 for a definition of the Awaitable concept https://wg21.link/P1288R0

```
  if (aa.has_value()) { use(aa.value()); }
  if (bb.has_value()) { use(bb.value()); }
}
```

Motivation

The Coroutines TS (N4760) introduces the ability to write asynchronous functions that can suspend and later resume execution. The paper P1056R0 proposes adding a new `task<T>` type as the standard coroutine type.

One feature of the `task<T>` type is that it is inherently sequential in nature - it cannot be used to introduce concurrency in the asynchronous operations. This design is intentional as it allows the `task<T>` implementation to avoid requiring any synchronisation overhead in the case where concurrency is not needed. However, this means that we need to provide some other abstraction to allow a coroutine to introduce concurrency.

Consider a typical situation where a function needs to perform multiple requests to a database or service and then later combine their results once both results are available.

Example: A naive implementation may look something like this

```
// This makes a request to a database/service to get a friends list.
task<std::vector<int>> get_friends(int id);

task<std::vector<int>> get_common_friends_slow(int id1, int id2)
{
  auto friends1 = co_await get_friends(id1);
  auto friends2 = co_await get_friends(id2);
  co_return set_intersection(friends1, friends2);
}
```

The disadvantage of this approach is that the second request to the database is not made until the first request has completed, meaning that the overall latency is at least to two round-trips to the database. Ideally, to minimise latency, we would like to execute the two database requests concurrently.

The `when_all()` function can help with this!

```
task<std::vector<int>> get_common_friends_fast(int id1, int id2)
{
  auto [friends1, friends2] = co_await std::when_all(
    get_friends(id1), get_friends(id2));
  co_return set_intersection(friends1, friends2);
}
```

In this example, we first call get_friends(id1) and get_friends(id2) to create, but not yet start, the tasks for

In this example, the `get_friends(id1)` task first starts executing and will send off the a request and then suspend waiting for the response. Once it suspends, the `when_all()` operator then immediately starts executing `get_friends(id2)` which then also sends off a request and suspends waiting for the response. Once both responses arrive and both tasks resume and run to completion the `when_all()` operator will resume execution of the awaiting `get_common_friends_fast()` coroutine. This means that the two requests are now being executed concurrently and we have limited the operation to a single round-trip.

Task-based Parallel Algorithms

The when_all() operator can also be used to implement task-based parallelism for running data-parallel algorithms on a thread-pool.

For example, by using the `when_all()` operator in conjunction with the work-stealing `static_thread_pool` executor from cppcoro it becomes straight-forward to implement parallel algorithms.

Example: A recursive parallel accumulate algorithm.

```
template<typename Iter, typename Sentinel, typename T,
         typename BinaryOp, typename Scheduler>
task<T> parallel_accumulate(Iter begin, Sentinel end, T init,
                            BinaryOp op, Scheduler& scheduler,
                            bool defer = false)
{
  if (defer) {
    co_await scheduler.schedule();
  }

  auto count = std::distance(begin, end);
  if (count < 512) {
    // Below some threshold just run the single-threaded version.
    co_return std::accumulate(begin, end, init, op);
  } else {
    // Divide range into two halves
    auto mid = begin + (count / 2);
    // Use when_all() to execute both halves concurrently.
    // Pass 'defer=true' to first call to schedule it onto the
    // executor. Pass 'defer=false' to second call to execute it
    // inline on the current thread.
    auto [left, right] = co_await when_all(
      parallel_accumulate(begin, mid, init, op, scheduler, true),
      parallel_accumulate(mid + 1, end, *mid, op, scheduler,
false));

    co_return op(left, right);
  }
}
```

```
void parallel_sum_example()
{
  cppcoro::static_thread_pool tp;

  std::vector<double> values = ...;

  // Use sync_wait() from P1171 to block on async result.
  double result = sync_wait(parallel_accumulate(
    values.begin(), values.end(), 0.0, std::plus<>{}, tp));

  std::cout << result << std::endl;
}
```

API Synopsis

```
// <experimental/coroutine>
namespace std::experimental
{
  // Something we can put in a tuple in place of void.
  struct [[maybe_unused]] __unit {};

  template<typename T>
  using __void_to_unit_t = conditional_t<is_void_v<T>, __unit, T>;

  // This is the same decay operation used in sync_wait() proposal.
  template<typename T>
  using __rvalue_decay_t = conditional_t<
```

```
    is_lvalue_reference_v<T>, T, remove_cvref_t<T>>;

  template<typename T>
  using __tuple_decay_t = __void_to_unit_t<__rvalue_decay_t<T>>;

  template<Awaitable... Awaitables>
  auto when_all(Awaitables&&... awaitables)
    -> AwaitableOf<std::tuple<
         __tuple_decay_t<await_result_t<Awaitables>>...>>;

  template<typename T>
  class when_all_result
  {
  public:
    when_all_result(when_all_result&& other) noexcept;
    ~when_all_result();
    when_all_result& operator=(when_all_result&& other) noexcept;

    // Result is move only
    when_all_result(const when_all_result& other) = delete;
    when_all_result& operator=(const when_all_result&) = delete;

    // Result either has a value or an exception.
    bool has_value() const noexcept;

    // Obtain the exception without throwing.
    std::exception_ptr exception() const noexcept;

    // Returns a reference to the value or rethrows the exception.
    add_rvalue_reference_t<T> value() &&;
    add_lvalue_reference_t<T> value() &;

  };

  template<Awaitable... Awaitables>
  auto when_all_ready(Awaitables&&... awaitables)
    -> AwaitableOf<std::tuple<
         when_all_result<await_result_t<Awaitables>>...>>;
}
```

Design Discussion

The semantics of when_all()

When you call when_all() with arguments a1, a2, ... , aN where N > 1, the function allocates N new coroutine frames by calling an unspecified coroutine function N times, once with each argument. This coroutine function is initially suspended.

For example, an individual when_all coroutine will look something like this:

```
template<typename _Awaitable>
auto __make_when_all_task(_Awaitable awaitable)
  -> __when_all_task<await_result_t<_Awaitable>>
{
  co_yield co_await static_cast<_Awaitable&&>(awaitable);
}
```

The when_all() function then returns a new Awaitable that holds all of these ‘when_all task’ objects. When the returned Awaitable is co_awaited, its operator co_await() will resume the first coroutine which will then execute the ‘co_await awaitable’ expression. When the first coroutine suspends or runs to completion and execution returns to the when_all operator it then resumes the second coroutine. And so on until all of the coroutines have been started.

Each of these coroutines then has some logic in its final_suspend() to decrement the atomic counter of outstanding work. When this counter hits zero it resumes the coroutine that was awaiting the when_all awaitable object. This then calls await_resume() on the awaitable object which then builds a tuple of the results or rethrows an exception.

Why do we need both when_all() and when_all_ready()?

The main difference between the two functions revolves around how exceptions are handled.

The when_all() function provides an interface that returns a tuple of all of the values returned by the co_await expressions. This makes it easy to obtain the individual values with a structured binding. However, if any of the component co_await operations complete with an exception then the operation as a whole completes with an exception and any partially successful results are discarded. It is also not possible to determine which operation threw the exception in this case.

The when_all_ready() function is intended for the use cases where you still want to execute operations concurrently and wait for them all to complete before continuing, but where you want to be able to handle partial failures/successes or where you want to be able to determine which operation failed. The interface of when_all_ready() is slightly more cumbersome to use but this is because it gives you more information.

How do you handle reporting multiple failures of concurrent operations?

Whenever we have multiple concurrent operations, each of which could fail independently, we need to come up with a strategy for handling partial failure or for reporting multiple failures.

The proposed behaviour for when_all() in the case when more than one of the input operations fail with an exception is to simply pick one of the exceptions and rethrow that exception as the result of the when_all() operation.

If it is important to know which operations failed or to handle partial failures then the when_all_ready() operator can be used and the individual results can be subsequently queried for success or failure.

What sort of transactional guarantees do I have that either all operations will start or none will start?

When launching concurrent computation it can sometimes be important for correctness that either all operations start executing or that none of them do. For example, one task may start executing and then later wait for a signal from the second task. If we start executing the first task and then fail to start the second task then the first task will never finish executing.

In order for the `when_all()` operator to be able to concurrently wait for each of the awaitables it will need to allocate a new coroutine frame for each awaitable. Each coroutine-frame will be responsible for executing the `co_await` expression for one of the input awaitables. As this may need to allocate memory on the heap, it's possible that the creation of the coroutine-frame could fail.

To provide the guarantee that either all operations will be co_awaited or none of them will be, the when_all() operator typically needs to allocate all of the coroutine frames when the function is first called. This allows it to guarantee that it can launch all of the operations when the returned awaitable is actually co_awaited.

Why is the order that the operations are launched important?

The when_all() and when_all_ready() functions explicitly define the order that the co_await expressions are evaluated to be in order from the first argument to the last argument.

This ordering can be important for correctness in some cases. For example, we may use when_all() to send multiple requests in a pipelined fashion down a single message channel that will process the messages in-order. It may be important to send these requests all at once but the ordering may be significant.

Another example may be that we want to pipeline multiple sends on a socket (I realise there are more efficient ways of doing this - imagine there is some buffering going on underneath).

```
task<> socket_example(connection c)
{
  // These need to be launched in-order.
  auto [send1, send2, send3] = co_await when_all(
    c.send(get_header()),
    c.send(get_body()),
    c.send(calculate_checksum()));
}
```

Another example is using when_all() to first launch some asynchronous work and then enter the event loop of some I/O context.

```
task<> my_application(io_context& io);

int main()
{
  io_context io;
  sync_wait(when_all(
    [&]() -> task<>
    {
      // Launch the application.
      co_await my_application(io);
      // Stop the event loop when app task stops.
      io.stop();
    }(),
    [&]() -> task<>
    {
      // This task will be run second after the application has
      // launched and will enter the event loop.
      io.run();
      co_return;
    }()));
}
```

This technique allows us to launch an asynchronous operation and enter an event loop and then cleanly shutdown without needing to launch any threads. However, this relies on the tasks being awaited in a specific order.

How does this proposal relate to the `when_all()` function from the Concurrency TS?

The Concurrency TS also defines a std::experimental::when_all() function which accepts a variadic number of std::experimental::future<T> objects.

Currently, there would be no conflicts between the when_all() function proposed by this paper since the overload proposed here is constrained to Awaitable types and std::experimental::future<T> does not currently satisfy the Awaitable concept.

If we do choose to extend std::experimental::future<T> to implement an operator co_await() in future then we will need to either find a way to differentiate the two overloads or find a way to merge them.

Wording

Wording has not yet been drafted for this proposal.

Appendix - Reference Implementation

This proposed reference implementation is based on the implementation from cppcoro[2] with some modifications to make use of symmetric-transfer so that the launching of the last operation is done with a tail-call and resuming of the awaiting coroutine when the last operation completes is also done with a tail-call.

An online version is available here: https://godbolt.org/z/omE2dZ

Note, this implementation depends on the presence of the `await_result_t<T>` and `awaiter_type_t<T>` type traits from P1288R0.

[2] https://github.com/lewissbaker/cppcoro