Executors review: concepts breakout group report.

document: P2204R0
date: 2020:08:14
audience:
  - Library Evolution Working Group
author:
  - name: Guy Davidson (chair)
    email: <guy.cpp.wg21@gmail.com>
  - name: Ben Craig (minutes)
    email: ben.craig@gmail.com
  - name: Robert Leahy
    email: rleahy@rleahy.ca
  - name: Michał Dominiak
    email: griwes@griwes.info
  - name: Alexey Kukanov
    email: alexey.kukanov@intel.com
  - name: Hartmut Kaiser
    email: hartmut.kaiser@gmail.com
  - name: Daisy Hollman (author)
    email: dshollm@sandia.gov
  - name: Jared Hoberock (author)
    email: jhoberock@nvidia.com
  - name: Gordon Brown (author)
    email: gordon@codeplay.com

Abstract

This is the report of the Executors review group 3: Concepts, of
paper [P0443R13].

Over four one hour meetings we reviewed sections 2.1, 2.2.2, 2.2.9 and 2.3.

Recommended actions were largely editorial:

1. Migrate the inheritance of receiver_invocation_error from the header synopsis to the class synopsis

2. Italicise unspecified items

3. Withdraw hyphens from exposition-only types unless there is precedent

4. Provide examples of user code that relies on concepts

5. Provide an annex of new identifiers being introduced

The findings below are a summary of the minutes, also attached for reference.

We found a few issues with the paper. Some have to do with a general lack of examples and the fact that the paper isn't finished. Due to the seeming generality of the mechanism and lack of example guidance and discussion, it is difficult to program against it without running into subtle impedance mismatches in interfaces. Clearer guidance on "how you should use this" as opposed to just concept definitions would be very welcome.

As expected, the paper authors were extremely responsive during the review, and have already taken a number of issues under advisement.

We are looking forward to the next iteration of the paper.

Potential Polls

POLL: Move away from combined traits class in favor of single element traits

POLL: Move away from namespace "execution", in favor of something shorter

POLL: Remove executor_shape, as we won't be able to change it after release, and it provides nothing other than a verbose size_t

Findings

2.1.1 General

The first question was about the nature of a thread pool. This was described as an execution context, in the same way that a GPU runtime is also an execution context. It is a long lived resource. An executor is a handle to the execution context, and an execution agent is what the executor allocates: a function invoking some place with some properties.

Secondly, the lifetime of an execution agent has its own paragraph, and a question was raised about why this is warranted. This is because some of the properties refer to the lifetime of the agent. Sometimes each execution agent gets its own thread, or a unique thread that was already present. During the drafting of P0443 it was insisted that this be stated.

Sometimes the agent might have a binding to a particular resource or create resources. Describing lifetimes of the things dependent on the execution agent is assisted by this definition. The execution context is long-lived, while the agent is created to execute a particular function. It isn't reused. An agent could reuse a given resource, but each agent is unique.

2.1.2 Header <execution> synopsis

The long namespace, execution, is a cause for concern.

It was observed that all of the customisation points look for member functions before free functions.

Nine concepts are defined. There was discussion about whether they need to be named or if they could be exposition-only. However, using SFINAE is very inconvenient in terms of implementation. Concepts may appear in interfaces: the executor concept definitely will. Algorithms designed around them will need to use these concepts to compose a chain of work, where you want to ensure compatibility along the chain. They are necessary for overload sets for senders and receivers. Additionally, this allows non-standard libraries to have a common vocabulary amongst each other. Even if they weren't concepts, they would need to be named requirements.

The relationship between sender and receiver was discussed at length. To determine that something is a sender, you need a receiver too. If you just have the sender in isolation and can't test against a receiver there needs to be a way to advertise that something is a sender. Without that, sender_traits can't be used. Having said that, the issue of whether or not traits classes are discouraged in modern C++ was raised.

It is important to standardise these properties because some of these are likely to be used by the standard library. For example, the parallel STL will need bulk_execute, while the networking TS will need others. Combinations of properties must make sense: not every executor needs to be concerned with the properties outlined.

The memory allocation properties gave reviewers pause. std::execution::allocator is an instance, while std::allocator is a template. However, it enables statements like
auto ex_with_alloc = require(ex, execution::allocator(alloc));
which is seen as valuable.

There was discussion about executor_shape. In general, it is a hierarchy of nested multiple dimensions. There was consideration of having the shape as an interval; some said that bulk_execute should take a range rather than a single integer, but platform APIs do better with a single integer. Having bulk_execute produce a contiguous set of indices allows it to be used more generically.

2.2.2 Invocable archetype

The idea of a type that cannot be created provoked discussion. No precedent could be recalled for such a thing. Perhaps deleted constructors would be appropriate to prevent the creation of an instance of such a class.

An author remarked that it would make GPU executors impossible if it were an empty type, illustrating his point with the following code:

```
// with invocable_archetype
struct inline_executor
{
  template<class F>
  void execute(F f)
  {
    ...
  }
};


// with some other solution
struct inline_executor
{
  using executor_tag = ...;

  template<class F>
  void execute(F f)
  {
    ...
  }
};
struct gpu_executor
{
  void execute(invocable_archetype);

  template<class F>
    requires gpu_function<F>
  void execute(F);
};
```

The author remarked that it would be fine to create instances of invocable_archetype as long as you can't invoke operator().

2.2.9 Concepts executor and executor_of

It was observed that copy_constructible<E> was implied by is_nothrow_copy_constructible_v<E> in the executor-of-impl expository concept. Also, the execution::execute requirement may need to be std::forward(f) instead of (F&&)f.

In the executor_of concept, executor<E> is checked in addition to the expository concept. This is because this forces a check of both F and invocable_archetype. There was uncertainty if the latter was necessary: it may be for subsumption. However, it means that all execute extension points need to be able to take an invocable_archetype, which means that execution::execute will usually need to be a template or provide specific (unused) execution::invocable_archetype. This makes genericity easier but specificity harder. Perhaps executor<E> could be defined in terms of executor_of<E, invocable_archetype>.

Clarification is required on defining swap for executor. This raised issues of thread safety.

In the context of the execution::execute requirements table, two authors discussed the insufficiency of block's definition. Within the operational semantics, block may refer to concurrent forward progress or parallel forward progress, but it is not clear which. It was observed that the table used to have many more rows, and that the requirements may be better presented as a paragraph. If mandates are required in such tables then the section can be rearranged appropriately.

The synchronizes-with semantic should be with the invocation of cf, not f.

The issue of exception propagation was thorny. There were a lot of compromises in this space. The outcome was to increase the burden on the executor implementer, rather than the person that passes a function to the executor.

2.3 Executor type traits

These types are intended to be specialised by users. There was author dissatisfaction with this part of the paper, which is quite old. Non-rectangular examples of shape would be useful here. Shape seems like a poor choice of name: perhaps extent, rank or dimensions would be more appropriate. A good suggestion turned out to be coordinate.

executor_shape_t describes N-dimensional cubes whose "lower-left" corner is at the origin. The name "shape" was chosen because it describes the shape of an iteration space. Conceptually, the elements of executor_shape_t represent the upper bound of a set of perfectly-nested loops. Initially, executor_shape_t is fixed to be size_t and can only represent a one-dimensional iteration space (i.e. a single for loop). We envision relaxing that constraint to describe higher-dimensional spaces nested arbitrarily deeply. In such a regime,

we could imagine defining a concept for such types, and it would require that a shape be either an integral type, or a tuple-like type of shapes.

Minutes

Many thanks to Ben Craig for taking minutes throughout the process. This is a combined presentation of the minutes for all the sessions.

2.1.1 General

Robert: Was a paper about bulk execute customization point in the last mailing.

Jared: SG1 hasn't looked at it yet, suggesting we not look at it yet.

Ben: What is a thread pool

Jared: An execution context. A GPU runtime is also an execution context. A long lived resource.

Jared: Executor is a handle to the execution context.

Jared: Execution agent is a thing the executor allocates. A function invoking some place with some properties. Useful for saying what the extension points do.

Guy: Lifetime of execution agent? Why does that get it's own paragraph? Why do we need to define it?

Jared: Some of the properties refer to the lifetime of the agent. Sometimes each execution agent gets it's own thread. Sometimes each execution agent gets a unique thread, but was already present. At some point, someone insisted we state this.

Gordon: Motivation is there because sometimes, it might have a binding to a particular resource or create resources. This helps us describe the lifetimes of the things that depend on the execution agent. The lifetime may not mean much on it's own, but it is important for certain contexts.

Alexey: The context is long lived, the agent is something that you create when you need to execute a particular function. It's not reused, doesn't have to execute more than one function.

Gordon: It could map to reusable resources, and one agent could reuse a given resource, but each agent is unique.

Robert: Last paragraph doesn't discuss timeliness, just that it is cleaned up at some point after it is invoked.

Jared: Lifetime on the beginning of the lifetime could be substantially deferred as well, while waiting for some resource to become available.

2.1.2 Header <execution> synopsis

Ben: note the long namespace execution, may be bikeshed

Jared: policies already exist there

ACTION
Ben: synopsis of stdexcept.syn does the wording here different. Would just be `struct receiver_invocation_error;`, and the inheritance would be shown in the class synopsis instead of the header synopsis.

ACTION
Ben: Italicize your unspecifieds

Ben: These customization points are all in terms of free functions?

Jared: All of these look for a member function, then a free function

Ben: What are requirements of connect_result_t?

Jared: Covered in connect. Probably won't enforce that S and R are senders and receivers.

Guy: are the hyphens in the exposition only types ok?

Ben: Look for precedent for that, if none, please change it.

Ben: Can't change concepts, do we need these to be named concepts? can they be exposition only.

Jared: using sfinae is very inconvenient in terms of implementation. Need for concepts, may appear in interfaces. Executor will definitely appear in interfaces.

Gordon: Algorithms designed around this will need to use these concepts to compose a chain of work, where you need to make sure what you are passing down the chain is the right kind of work, and you want to make sure things are compatible.

Gordon: senders and receivers that you are composing where you have a tuple of types that you want to be able to have different error types and dispatch based off of that. Probably necessary so that you can have an overload set for sender and receiver.

Jared: Aside from overloads and decorating standard libraries like the sender combinator library. Another case is for non-standard libraries to have a common vocabulary between libraries. Even if they weren't concepts, they would need to be named requirements.

Ben: changing named requirements are technically breaking changes too.

Ben: recommend you have examples with user code caring about a concept.

Ben: is the sender concept a disjunction, where it is x || y || z? Those are rough on subsumption and build throughput.

Ben: aren't traits classes discouraged? Shouldn't we be asking one question at a time rather than bundle them?

Alexey: not sure why sender_base is needed. The concept should be defined in terms of what the type can do, why would we need sender_base to opt in to this?

Jared: To figure out if something is a sender, you really need a receiver too. If you just have the sender in isolation and can't test against a receiver, we need some way to advertise that something is a sender. Without that, we can't use sender_traits.

Ben: Do these properties have state?

Jared: They have members, but they are stateless. You can say blocking.never. Shouldn't be any non-static data members.

Alexey: Can these properties be considered universal, or are they executor specific? Not all executors will have these properties, right? Why is it important to standardize these properties.

Jared: Some of these will be used by the standard library. The parallel STL will need bulk. Need to make sure combinations of properties make sense. Some of these are needed by the networking TS. Not every executor needs to concern themselves with these specific properties.

Ben: std::allocator is a template, std::execution::allocator is an instance?

Guy: gave me pause too

Jared: Want to be able to craft things like this...
auto ex_with_alloc = require(ex, execution::allocator(alloc));

Guy: is there an annex or appendix of all new identifiers to aid in bikeshedding? Can help focus the efforts of those that care deeply about the names.

Ben: So executor_shape_t is currently size_t? A shape is a size?

Jared: In the generalization, it's a hierarchically nested multiple dimensions

Alexey: did you consider having the shape as an interval, so that you can do something besides 0-n, but also, 1-n?

Jared: Some said that bulk execute should take a range rather than a single integer. Platform APIs do better with a single integer though. By assuming that bulk execute produces a contiguous set of indices lets us simplify things and use bulk execute more generically.

2.2.2 Invocable archetype

Ben: any precedent for a type that can't be created. Why not an empty type that would be pointless to create? Can it be exposition only?

Jared: If we want all executors to be able to invoke this, it would make some kinds of executors impossible. Namely GPU ones

Alexey: Can we borrow from the definition of declval

From Jared Hoberock to Everyone: 01:28 PM

```
// with invocable_archetype
struct inline_executor
{
  template<class F>
  void execute(F f)
  {
    ...
  }
};


// with some other solution
struct inline_executor
{
  using executor_tag = ...;

  template<class F>
```

```
  void execute(F f)
  {
    ...
  }
};
struct gpu_executor
{
  void execute(invocable_archetype);

  template<class F>
    requires gpu_function<F>
  void execute(F);
};
```

Need to constrain the execute function in some way.

Jared: would be fine to create instances of this, so long as you can't actually call the call operator function.

Ben: alternatively, add some =deletes to constructors or destructors so that the core language makes it impossible to create instances of the class.

2.2.9 executor and executor_of

Jared: Maybe remove copy_constructible<E> and let is_nothrow_copy_constructible to handle it

execution::execute requirement may need to be std::forward instead of (F&&)f

Guy: italicize executor-of-impl

Jared: (can't do that easily)

Alexey: Why check executor in executor_of

Jared: Want to check against both the provided F and for invocable_archetype

Alexey: do we need to check against invocable_archetype

Jared: Not sure. May just need that for subsumption.

Ben: That means that all execute extension points need to be able to take an invocable_archetype. Means that exection::execute will usually need to be a template, or provide specific (unused) execution::invocable_archetype.

Alexey: THis makes the generic thing easy, but the specific thing harder.

Alexey: Why not have just executor<E> defined in terms of executor_of<E, invocable_archetype>

Jared: Not sure, Eric is going to know this better.

Alexey: I didn't see a swap defined for executor.

Jared: I'm not sure if it was an explicit choice whether this should be a requirement of the concept or not. If an executor is moveable, it seems like it should be swappable? Not sure if that hits no throw swappable.

Ben: Swappable is allowed to throw.

Alexey: do you mean same instance can be used from multiple threads, or different instances. Currently say's "executor's type's"

Jared: Should probably be referring to executor objects.

Alexey: swap being thread safe is weird

Ben: Destructor being thread safe is _really_ weird

Alexey: So maybe this does refer to the type.

Ben: vector doesn't have to specify that distinct vectors are thread safe from each other.

Daisy: This is another place where we don't define block thoroughly enough. There's a lot of variants of not block. This might be concurrent forward progress.

Gordon: May need some other sync point for parallel forward progress, or it might just be concurrent forward progress.

Daisy: We don't say anything about sychronizing with the completion of f

Alexy: why is this a table and not a paragraph?

Daisy: Because the table used to be a lot more rows.

Guy: several semantics in that single element.

Gordon: This was particularly useful for bulk executors at the time.

Daisy: Something about tables have a certain level of power that don't require them to have mandates, etc in them. If they want those things, then we can rearrange.

Alexy: What happens first, what happens next with synchronizes with?

Daisy: it means that relaxed atomic operations need to be visible on another thread of execution. Strongest form of memory barrier. Things that happen before the call to execute are visible to f.

Robert: There is no invocation of f, there is an invocation of cf.

Ben: Does anything else threading related use DECAY_COPY?

Daisy: algorithms capture the predicate with decay_copy

Robert: decay-copy, not decay_copy

Robert: Does that exception statement include the copy of the object?

Daisy: Intended to involve the invocation, and not the copy. Probably needs tweaking.

Gordon: exceptions should be able to be thrown be execute itself. Queue may be full.

Alexey: Inline exector would need to do something extra to not propatage an exception?

Daisy: Yes, just a try / catch block though. Should be an example of the inline executor in the paper.

Alexey: Why wouldn't you say that any exception thrown is undefined behavior? That would let the inline executor be simpler.

Daisy: This is an area that was fought out a lot, lots of compromises in this space. Want to put more burden on the executor implementer, rather than the person that passes a function to the executor. We don't want UB because that would make it hard to just use stdlib functions that happen to throw exceptions.

Alexey: I get why you don't want UB, but my concern is the "shall-not propagate". Not sure why that is considered better. terminate, exception list, and implementation defined thing may just be better.

Daisy: Takes getting used to this generic design. Every call to execute, if we don't say this is noexcept, every call to execute in generic code would have to assume that execute could throw. The answers would be different if we were doing a concrete facility.

Robert: At that point, if exceptions can come from the invocables, then implementers could no longer assume that execute exceptions are really bad.

Gordon: Still leaves a window for other execution contexts to provide back channels for handling exceptions in more specialized uses.

Daisy: Things are a little backwards in this level of generic facility. Implementation defined means less flexibility in this case.

Alexey: Why not say that the callable is unconditionally noexcept?

Daisy: Let's you assume that any exception that comes out of execute means that work couldn't be scheduled. CPO was noexcept at one point, but people said that that was too constraining.

Daisy: Some of these error cases can be handled at the sender receiver level, but not the executor level.

Robert: Example implementation of the inline executor is broken because it doesn't decay-copy and propagates exceptions. Won't compile executing because of rvalue invocation.

2.3 Executor type traits

Daisy: These are largely for bulk

Gordon: Bulk is still in here, but in a slightly reduced form

Daisy: cuda has a shape, a 3d thing, and a block index that has an x,y,z. Seemed too domain specific. Only issue is that if you claim that it is integral, it is integral for all time.

Ben: Are these types intended to be specialized by users?

Daisy: Yes. Also, using detection idiom

Gordon: This is an older part of the paper.

Gordon: Maybe shape and index could be a more concrete type in the future.

Daisy: Agreed that was the intent, but not sure that this is possible here. Should talk with Jared if these still need to be here. Need to ask LEWG if this can be generalized in the future.

Gordon: should these be concepts? You'd have to know that an index is appropriate.

Robert: Seems that shape is trying to pick something out of a T. If you can specialize it, then the specialization wouldn't have the static assert.

Daisy: Non-intrusive specialization can be non-integral? That would make this a completely open concept.

Robert: agreed. This currently feels really weird.

Ben: We've changed traits several times before, both in core language and library.

Ben: an int doesn't seem like a shape

Gordon: expresses iteration shape

Robert: is it describing shape or dimensionality? Could this describe a triangle?

Daisy: Potentially. No intention to have shape imply rectangular.

Alexey: I think that shape means set of coordinates that describes a rectangle or vertices. Wasn't interested in describing abstract multi dimension rectangle.

Daisy: would have to assume that all dimensions have a uniform flattening. Sometimes a row major flattening is much more performant, and sometimes a column major flattening is better.

Gordon: Maybe any shape type or index type should be convertible to an integral type.

Daisy: Also has subsumptions and concrete examples in mind that can do more interesting things.

Robert: Why do we care about these, and not just about the addability and subtractability of the bounds?

Daisy: Maybe we should be using iterator terminology on this. Many HPC folks were uncomfortable with range formulation.

Robert: Then we should get rid of these, and the default should be iota_range

Daisy: That has come up before. FB isn't too keen on bulk execution in general. They think that is a higher level algorithm than this paper wants to make it.

Ben: Would like to see non-rectangular examples

Ben: Still confused on shape. How does size_t represent a torus, or a double helix?

Gordon: History was that it was going to cover rectangular contiguous things, and maybe triangular

Ben: Maybe extent, or rank, or dimension

Guy: Look at md_span?

Guy: Was the issue the name, or something deeper?

Gordon: I think there were issues with whether it was rectangular, or something more complicated, like a stride. We could request clarification or a name change.

Summary review meeting

Ben: Potential polls:

POLLS: Move away from combined traits class in favor of single element traits

POLLS: Move away from namespace "execution", in favor of something shorter

POLLS: remove executor_shape, as we won't be able to change it after release, and it provides nothing other than a verbose size_t

Jared: We may be able to get to a point where scheduler concept isn't necessary. Don't think that it adds enough to keep it. Scheduler and executor are close enough to each other to be made identical, and things would be better if they were the same.

Jared: Necessary at the moment because sender / receiver has a different view of error handling that can't be implemented on all executors. So we built it into schedulers. If we taught executors about error channels and other channels, possibly through properties, then we may not need scheduler. Not currently possible.

Gordon: If we could get sender / receiver through properties, then we could join the two. That would dramatically simplify things. But we spent a lot of time trying to get there.

Jared: A forcing function from LEWG may get there.

Jared: Last poll. From my prototyping and experimenting, it would be better if the two were combined into a single type. Having them separate causes more problems than they solve. Name like coordinate would describe the ideas. Would also avoid the problems with the name shape and index. We normally use the term index for one dimensional things. I'd be in favor of making that change or not mentioning it at all. Also considered making an executor exposing it as a property instead of a type trait. Might be worth excising it from version 0.

Ben: What about the size_t criticism?

Robert: The paper contains some static_asserts related to size_t'ness.

Gordon: A requirement for the type to be integral.

Jared: Intent was for these to be integral for the first release, and relax later. If we can't relax later, then we need a different solution.

Gordon: Could make it more difficult to write code generic for different executors unless you always used auto.

Jared: The intent was to provide a name to the type and to make the type customizable. We haven't fleshed out the requirements beyond integral, which makes it a placeholder at the moment.

Jared: Whatever we do with shape should be taken in consideration with bulk_execute, because that's the consumer of it.

Jared: sender_traits is really inconvenient to use. Two of the members are templates themselves, so we may be able to enhance the ergonomics.

Jared: Dealing with sender_traits and defining senders are the most painful parts of this paper. Requires a lot of metaprogramming.

<END>