# History of Executor Properties

Jared Hoberock (jhoberock@nvidia.com)

2020-01-13

P2033

# Goals

Describe the history of P0443 as it relates to **executor properties**

Provide rationale for the design of P1393’s general purpose properties system

# Prehistoric Executors

N3378 - A preliminary proposal for work executors (Google), February 2012

N4046 - Executors and Asynchronous Operations (Kohlhoff), May 2014

N4406 - An Interface for Abstracting Execution (Nvidia), April 2015

Focused on different use cases

# Google Proposal

Executors derive from abstract base class

Type erasure

Methods possibly block

Separate methods for timed execution

`thread_pool` is-an executor

```
class executor
{
public:
  virtual void add(function<void()> closure) = 0;

  virtual void add_at(time_point abs_time,
                      function<void()> closure) = 0;

  virtual void add_after(duration rel_time,
                         function<void()> closure) = 0;

  ...
};

class thread_pool : public executor { ... };
```

# Kohlhoff Proposal

Three fundamental operations

Differ in how they are allowed to block the caller

Distinction between executors and execution contexts

```
class my_executor
{
public:
  template<class Function, class Allocator>
  void dispatch(Function&& f, const Allocator& alloc);

  template<class Function, class Allocator>
  void post(Function&& f, const Allocator& alloc);

  template<class Function, class Allocator>
  void defer(Function&& f, const Allocator& alloc);

  execution_context& context() noexcept;

  ...
};
```

# Nvidia Proposal

Emphasized bulk execution

`executor_traits`-based adaptation

Provided channels to results

Distinguished between async and blocking ops

```
template<class Executor>
struct executor_traits
{
  template<class Function>
  static future<auto> async_execute(Executor& ex,
                                    Function f);
  template<class Function>
  static future<auto> async_execute(Executor& ex,
                                    Function f,
                                    shape_type shape);

  template<class Function>
  static auto execute(Executor& ex,
                      Function f);
  template<class Function>
  static auto execute(Executor& ex,
                      Function f,
                      shape_type shape);

  ...
};
```

# “Unify, please.”

SG1, Kona, October 2015

# Unification

Began regular teleconferences

Year of discussion

Identified additional use cases

Sent proposal to Issaquah, November 2016

# P0443R0 - A Unified Executors Proposal

Defined seven executor “categories”

- `OneWayExecutor`
- `HostBasedOneWayExecutor`
- `NonBlockingOneWayExecutor`
- ...

Fifteen execution functions exposed as Niebler-style customization points

Customization points adapt when native functionality is missing

Mandatory exposure of execution contexts via `.context()`

# Execution Functions

Name encodes characteristics

- Blocking
- Directionality
- Cardinality

| | Blocking | Directionality | Cardinality |
|---|---|---|---|
| `execute` | possibly | one-way | single |
| `async_defer` | never | two-way | single |
| `bulk_sync_execute` | always | two-way | bulk |
| ... | ... | ... | ... |

# Cross-Cutting Concerns in P0443R0

- Invariants preserved by adaptations applied by customization points
- Exposed via type traits
- Enables compile-time decisions

Type Traits

- blocking behavior
- execution agent mapping
- execution function detection
- bulk execution semantics
- associated types
    - context
    - future
    - index
    - shape

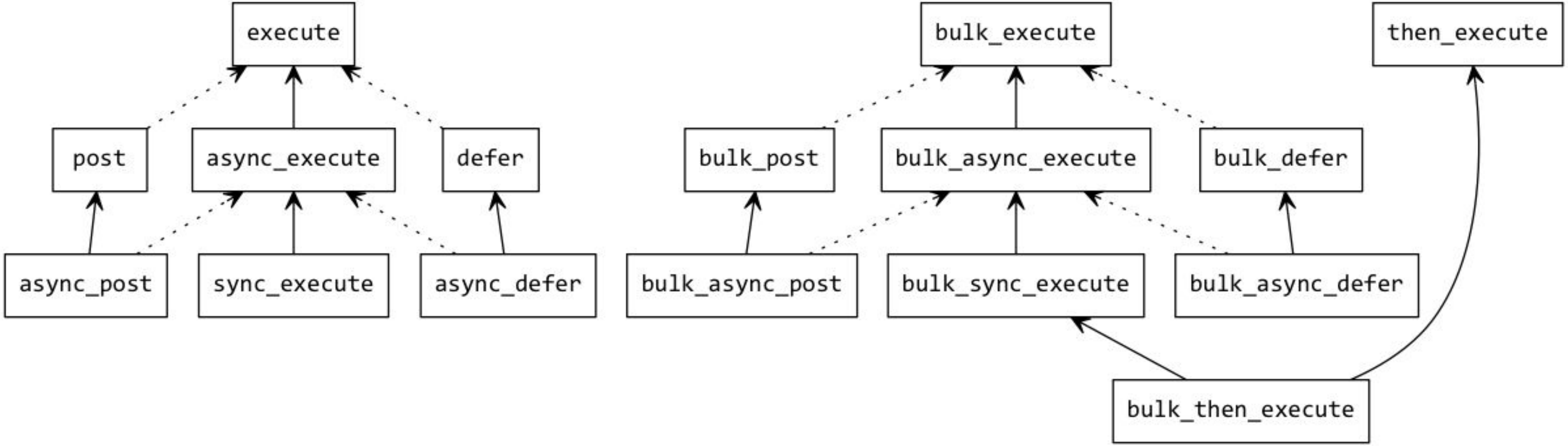
# Kitchen Sink

| | OneWay | HostBasedOneWay | NonBlockingOneWay | TwoWay | NonBlockingTwoWay | BulkOneWay | BulkTwoWay |
|---|---|---|---|---|---|---|---|
| .execute(f) -> void | 🟩 | 🟩 | 🟩 | | | 🟩 | 🟩 |
| .execute(f,alloc) -> void | | 🟩 | 🟩 | | | | |
| .post(f) -> void | | | 🟩 | | | | |
| .post(f,alloc) -> void | | | 🟩 | | | | |
| .defer(f) -> void | | | 🟩 | | | | |
| .defer(f,alloc) -> void | | | 🟩 | | | | |
| .async_post(f) -> future | | | | | 🟩 | | |
| .async_defer(f) -> future | | | | | 🟩 | | |
| .sync_execute(f) -> result | | | | 🟩 | 🟩 | 🟩 | 🟩 |
| .async_execute(f) -> future | | | | 🟩 | 🟩 | 🟩 | 🟩 |
| .then_execute(f,fut) -> future | | | | | | | |
| .bulk_execute(f,n,sf) -> void | | | | | | 🟩 | |
| .bulk_sync_execute(f,n,rf,sf) -> result | | | | | | | 🟩 |
| .bulk_async_execute(f,n,rf,sf) -> future | | | | | | | 🟩 |
| .bulk_then_execute(f,n,fut,rf,sf) -> future | | | | | | | 🟩 |

# Revised Kitchen Sink

| | OneWay | NonBlockingOneWay | TwoWay | BulkTwoWay |
|---|---|---|---|---|
| `.execute(f) -> void` | 🟩 | 🟩 | | |
| `.post(f) -> void` | | 🟩 | | |
| `.defer(f) -> void` | | 🟩 | | |
| `.async_post(f) -> future` | | | | |
| `.async_defer(f) -> future` | | | | |
| `.sync_execute(f) -> result` | | | 🟩 | |
| `.async_execute(f) -> future` | | | 🟩 | |
| `.then_execute(f,fut) -> future` | | | | |
| `.bulk_execute(f,n,sf) -> void` | | | | |
| `.bulk_sync_execute(f,n,sf) -> result` | | | | 🟩 |
| `.bulk_async_execute(f,n,sf) -> future` | | | | 🟩 |
| `.bulk_then_execute(f,n,fut,rf,sf) -> future` | | | | 🟩 |
| `.bulk_post(f,n,sf) -> void` | | | | |
| `.bulk_defer(f,n,sf) -> void` | | | | |
| `.bulk_async_post(f,n,sf) -> future` | | | | |
| `.bulk_async_defer(f,n,sf) -> future` | | | | |

# Adaptations Visualized



“This picture looks terrifying.”

SG1, Kona, February 2017

These functions simply create execution

So where is the complexity coming from?

# Factors of Complexity

```
execute
post
defer
async_post
async_defer
sync_execute
async_execute
then_execute
```

```
bulk_execute
bulk_sync_execute
bulk_async_execute
bulk_then_execute
bulk_post
bulk_defer
bulk_async_post
bulk_async_defer
```

# Factors of Complexity

**execute**

**post**

**defer**

async_post

async_defer

sync_execute

async_execute

then_execute

bulk_execute

bulk_sync_execute

bulk_async_execute

bulk_then_execute

bulk_post

bulk_defer

bulk_async_post

bulk_async_defer

Optional allocator argument

# Factors of Complexity

execute

**post**

defer

**async_post**

async_defer

**sync_execute**

async_execute

then_execute

bulk_execute

**bulk_sync_execute**

bulk_async_execute

bulk_then_execute

**bulk_post**

bulk_defer

**bulk_async_post**

bulk_async_defer

Blocking guarantee

# Factors of Complexity

execute
post
**defer**
async_post
**async_defer**
sync_execute
async_execute
then_execute

bulk_execute
bulk_sync_execute
bulk_async_execute
bulk_then_execute
bulk_post
**bulk_defer**
bulk_async_post
**bulk_async_defer**

Prefer continuation

# Factors of Complexity

**execute**<br>
**post**<br>
**defer**<br>
async_post<br>
async_defer<br>
sync_execute<br>
async_execute<br>
then_execute

**bulk_execute**<br>
bulk_sync_execute<br>
bulk_async_execute<br>
bulk_then_execute<br>
**bulk_post**<br>
**bulk_defer**<br>
bulk_async_post<br>
bulk_async_defer

One-way

# Factors of Complexity

execute
post
defer
**async_post**
**async_defer**
**sync_execute**
**async_execute**
**then_execute**

bulk_execute
**bulk_sync_execute**
**bulk_async_execute**
**bulk_then_execute**
bulk_post
bulk_defer
**bulk_async_post**
**bulk_async_defer**

Two-way

# Factors of Complexity

**execute**
**post**
**defer**
**async_post**
**async_defer**
**sync_execute**
**async_execute**
**then_execute**

bulk_execute
bulk_sync_execute
bulk_async_execute
bulk_then_execute
bulk_post
bulk_defer
bulk_async_post
bulk_async_defer

Single

# Factors of Complexity

```
execute
post
defer
async_post
async_defer
sync_execute
async_execute
then_execute
```

```
bulk_execute
bulk_sync_execute
bulk_async_execute
bulk_then_execute
bulk_post
bulk_defer
bulk_async_post
bulk_async_defer
```

Bulk

# Cross Product

Factors multiply combinatorially

Envisioned extensions aren’t yet represented

- Delayed execution
- Prioritized execution
- ...

Separate execution functions for each combination will not scale

Need to communicate ancillary execution properties separately from functions

# Factored Representation

P0688 proposed refactoring based on **properties**

See 5/16/17 sg1-exec thread “Executor simplification proposal”

# P0443R2: Factored Representation

## Functions

```
execute

bulk_execute

twoway_execute

then_execute

bulk_twoway_execute

bulk_then_execute
```

×

## Properties

```
never_blocking

always_blocking

possibly_blocking

continuation

not_continuation

...
```

# Result of Property-Based Factorization

Before

```
execute(ex, f)

bulk_execute(ex, f, s, sf)

sync_execute(ex, f)

defer(ex, f)

...
```

16 Customization Point Objs

After

```
require(ex, single, oneway).execute(f)

require(ex, bulk, oneway).bulk_execute(f, s, sf)

require(ex, single, twoway, always_blocking).twoway_execute(f)

prefer(require(ex, single, oneway), continuation).execute(f)

...
```

2 Customization Point Objs + 6 Member Functions + Many Properties

“P0443 is our preferred direction for executors.”

SG1, Albuquerque, November 2017

# P0443RX Follow-Ups

Introduced `query()`

Polished ergonomics

Clarified semantics

Introduced additional properties

Reduced scope to one-way in anticipation of Senders

Polymorphic executors were an important consideration for many design choices

# Additional Use Cases for Properties

Associated executors for various types

- Execution contexts
- Execution policies
- Tasks

“Arbitrary knobs” for execution policies

Array access behaviors

Allocator locality

See David’s Kona presentation on P1393

# Summary

P0443 is committed to supporting a diversity of use cases efficiently

Extensions to executors need to scale

Separating cross-cutting concerns from execution functions seems scalable

Reifying cross-cutting concerns as properties has been a productive organizing principle