Suggestions for `bulk_execute` | P1933

Jared Hoberock

2019-10-07

Abstract. In order to standardize executors on schedule for C++23, bulk execution must provide an appropriate eager substrate for our envisioned programming model for lazy execution. This paper compares two possible approaches that would allow `bulk_execute` to be such a substrate.

Comparison Summary

The proposed `submit(sender, callback)` requires the invocation of a scalar function (i.e. `callback`) after the work represented by `sender` completes execution. The purpose of this callback is to effect a transition to another execution context where downstream work that depends on the sender will be executed.

P0443R10's current specification of `bulk_execute` has no generic mechanism which could invoke a single scalar function following the bulk group of execution agents' completion:

```
template<class Function, class SharedFactory>
void bulk_execute(Function f, size_t n, SharedFactory shared_factory);
```

However, `bulk_execute` could be augmented to incorporate an explicit callback:

```
template<class Function, class SharedFactory, class Callback>
void bulk_execute(Function f, size_t n, SharedFactory shared_factory, Callback callback);
```

Alternatively, some executors could optionally return a *successor executor* from functions like `execute` and `bulk_execute`:

```
// submit f for execution and receive an executor in return
auto after_f_ex = ex.bulk_execute(f, n);

// my_callback is invoked only after f completes
after_f_ex.execute(my_callback);
```

This paper compares the two and suggests that the latter approach is technically superior.

Problem Definition

P0443R10 specifies `bulk_execute(f, n, shared_factory)`'s semantics (paraphrased):

> Let `s = shared_factory()`. Invokes `f(i, s)` for each value of `i` in `[0, n)`.

The semantics provide no in-band notification of the completion of the invocations of `f`, nor do they provide a mechanism for invoking a callback function, which is required by the senders programming model. At first glance, it may seem straightforward for a client to introduce an out-of-band augmentation of `f` that would elect one of its invocations to invoke a callback once all invocations complete. However, because the execution agents created by `bulk_execute` may have weak forward progress, there is no generic and efficient mechanism which could perform the necessary synchronization.

Instead, we require some explicit, in-band means of sequencing a scalar function invocation to follow the bulk function invocation.

Possibility 1: Introduce a Callback

One straightforward solution suggested by P1660 is simply to introduce a callback into `bulk_execute`'s parameters.

A possible signature for `bulk_execute` could be:

```
template<class Function, class SharedFactory, class Callback>
void bulk_execute(Function f, size_t n, SharedFactory shared_factory, Callback callback);
```

`callback` would be invoked after `f`'s invocations are complete. We believe this is implementable on any platform that could implement P0443R10's specification of `bulk_execute`, because the destructor of the shared state returned by `shared_factory` must also be invoked after `f`'s invocations are complete. If an executor is able to sequence that destructor invocation after the bulk invocation, we reason that it ought to be possible to do so for an arbitrary function as well.

Incorporating an explicit callback, or finalizing function, would have a nice symmetry with the shared factory, which acts as an initialization function. Moreover, the shared state variable could be passed off to the callback in cases where the value of the shared variable is interesting to downstream tasks.

Caveats

While introducing a callback into `bulk_execute` is workable, we have identified important caveats.

A Callback hides dependency information. We desire for executors to take full advantage of scalable systems that exploit visibility of task dependency relationships to create efficient schedules. The introduction of a callback hides dependencies from such systems:

```
ex.bulk_execute(f, ...,
  [=]{
    ex.bulk_execute(after_f, ...);
  });
```

In this example, an efficient scheduling runtime abstracted by `ex` cannot see that `after_f`'s invocation depends on the completion of `f`, because the lambda passed as the callback to the outer `bulk_execute` is opaque. This hidden dependency defeats the ability of the runtime to create efficient schedules.

Implementation strategies of Callbacks on GPU runtimes are limited. The most straightforward implementation of a Callback on GPU runtimes would be scheduled on a so-called "host callback". In this context, a host callback is a scalar function invocation executed on a CPU thread after preceding GPU work completes. The CUDA runtime places important restrictions on the code executed within such a callback. Critically, it is an error for host callbacks to make CUDA API calls. This limitation defeats the purpose of the callback, which is to create chained, dependent work. Worse yet, because Callbacks are opaque functions, there is no mechanism that can enforce this restriction.

No-op Callbacks waste resources. Some Callbacks will have no interesting work to perform. However, because Callbacks are opaque functions, this case is undetectable. Callbacks following scalar function invocations may be inlined or elided by the compiler because they may be scheduled on the same execution agent as the scalar function. Such a composition scheme is unavailable to Callbacks following bulk function invocation. In the bulk case, a no-op Callback must be scheduled and executed, wasting resources.

Possibility 2: Successor Executors

An optional result returned from `execute` and `bulk_execute` could be used as a token for chaining dependent work. Moreover, using an executor as the token would avoid the introduction of a new concept into the programming model. In this model, execution agents created by the resulting *successor executor* do not begin execution until the execution agents created from the originating function complete:

```
// ex returns a successor
auto after_f_ex = ex.execute(f);

// g is guaranteed to begin after f completes
after_f_ex.execute(g);
```

A successor executor ensures that a scalar callback executes after bulk execution:

```
auto successor_ex = ex.bulk_execute(...);
successor_ex.execute(my_callback);
```

Because successor executors provide an in-band channel for eager submission of dependent tasks, they enable executors to exploit dependency relationships for efficiency.

The purpose of a successor executor is to make explicit the ordering of tasks submitted through an executor. However, an implicit task still hides within `bulk_execute`'s current specification. Namely, the destructor of the shared object created by the shared factory must be invoked after the bulk work completes. If successor executors are adopted, we suggest eliminating this implicit join point by removing shared factories. Fortunately, efficient implementations of shared state we are familiar with may be efficiently reintroduced by a client via inspection of the executor's properties.

The possible signatures of `bulk_execute` become:

```
// one-way, fire-and-forget
template<class Function>
void bulk_execute(Function f, size_t shape);

// two-way,
template<class Function>
Executor bulk_execute(Function f, size_t shape);
```

In such a model, all data flow would be described either out-of-band or in-band via higher-level senders. The lower-level executor programming model would be concerned only with control flow.

Caveats

Successor executors also come with caveats.

Successors pessimize fire-and-forget tasks. Some use cases of executors do not require dependent tasks. In such cases, a successor returned from `execute` or `bulk_execute` will be discarded and any associated resources associated will have been wasted. Avoiding the initial resource allocation would allow the initial fire-and-forget task to execute more efficiently.

Adapting a fire-and-forget executor may be inefficient. We suggest that returning a successor executor be optional. However, we still require the ability to chain dependent work from tasks created via fire-and-forget executors. In such cases, we imagine that an adaptation could be applied to track the completion of the predecessor task. Some kind of reference counting scheme seems necessary to ensure that all copies of the successor executor can correctly track the predecessor task's completion.

Comparison of Trade-offs

	Callbacks	Successors
Usage	`ex.bulk_execute(f, n, factory, callback);`	`auto after = ex.bulk_execute(f,n); after.execute(callback);`
Example `submit(bulk_sender, cb)` implementation	`bulk_sender.executor_ .bulk_execute(..., cb);`	`auto after = bulk_sender.executor_ .bulk_execute(...); after.execute(cb);`
Optional?	Required by `bulk_execute`	Yes
Caveats	Hidden dependencies Limited usefulness on GPUs Inefficient no-ops	Pessimized fire-and-forget Inefficient adaptations

Summary

Though both approaches have important caveats, the way that a Callback hides dependent work from the executor seems to be the critical concern. Additionally, Successors' caveats seem tractable. Pessimized fire-and-forget could be avoided by naming the Successor-returning variants differently, possibly via a customization point. For these reasons, we prefer Successors over Callbacks in order to integrate `bulk_execute` with Senders.

Acknowledgements

Thanks to Michael Garland, David Hollman, Chris Kohlhoff, Eric Niebler, and Kirk Shoop for feedback.