A Better `bulk_schedule`

P2224r0

Michael Garland (mgarland@nvidia.com) Lee Howes (lwh@fb.com)
Jared Hoberock (jhoberock@nvidia.com)

September 14, 2020

1 Introduction

The paper P2181r0 ("Correcting the Design of Bulk Execution") introduces two fundamental interfaces for bulk execution:

- `bulk_execute`: an interface for eager work submission, and
- `bulk_schedule`: an interface for lazy work submission.

After extensive discussion, we have concluded that both the implementation and usability of `bulk_schedule` would be improved by a different formulation of its interface.

This paper summarizes our improved design for the `bulk_schedule` interface. We propose that a specification for an interface of this form be used in the next revision of P2181, while leaving its specification of `bulk_execute` unchanged, with the aim of correcting the specification of bulk execution in a future revision of P0443.

2 Interface

The originally proposed interface for `bulk_schedule`, which was presented to SG1 in August 2020, looked like this:

```
// Interface proposed in P2181r0
template<executor E, sender P>
sender auto bulk_schedule(E ex,
                          executor_shape_t<E> shape,
                          P&& prologue);
```

The returned object was a sender representing the initiation of the computation to be performed in each of the agents in the index domain given by `shape`. The caller was then responsible for constructing from this returned object another sender representing the actual computation to be performed in each agent. This interface left unspecified how a subsequent computation to be performed after the bulk section was complete could be attached to these senders, although a possible `bulk_join` operation was discussed in the SG1 meeting.

We propose to replace this interface with one of the following form:

```
// Interface proposed in this paper
template<scheduler E, invocable F, class... Ts>
sender_of<Ts...> auto bulk_schedule(sender_of<Ts...> auto&& prologue,
                                    E ex,
                                    executor_shape_t<E> shape,
                                    F&& factory);
```

The returned object is a sender representing the entire computation of the bulk section. It is analogous to the envisioned result of `bulk_join` in the old interface.

The invocable `factory` is responsible for constructing a sender that represents the computation to be performed in each agent of the bulk launch. The signature for this is a sender-factory and should be of the form:

```
auto factory(sender_of<executor_shape_t<E>, Ts&...>) -> sender_of<void>
```

The factory is called with a single parameter: a sender representing the initiation of each agent. This sender delivers to its receiver both an agent index and the values (if any) provided by `prologue`. The factory must return a `sender_of<void>` representing the entire computation to be performed by each agent.

The argument to the factory corresponds to the object returned by `bulk_schedule` in P2181r0 and its returned object corresponds to the sender constructed by operations applied subsequently by the caller of `bulk_schedule`. Thus, where the interface in P2181r0 would be used like so:

```
// P2181r0 approach to perform A in each of N agents, to be followed by B
// once the bulk section is complete.
auto S = bulk_schedule(ex, N, prologue) | ..A.. | bulk_join() | ..B..;
```

our new interface would be used like so:

```
// A better approach to specify the same computation
auto S = bulk_schedule(prologue, ex, N,
                       [](auto begin) { return begin | ..A..; })
           | ..B..;
```

3 Rationale

In this section, we review the most salient reasons to prefer our new design of the `bulk_schedule` interface.

As discussed during the presentation of P2181r0 to SG1, its design for `bulk_schedule` would have required a separate `bulk_join` operator for chaining dependent work after the bulk section. Our improved interface eliminates the need for this additional (as yet unspecified) operator.

Significant challenges arise in the implementation of P2181r0 due to the separation of `bulk_schedule` and `bulk_join`, which nevertheless must carefully coordinate their operation. A complete implementation of the interface described in this paper is substantially simpler.

The specification of `bulk_schedule` in P2181r0 required a new `many_receiver_of` concept. Another variant of `bulk_schedule`, explored in P2209r0, required both new `many_sender` and `many_receiver` concepts. In contrast, our improved interface requires no new concepts beyond those present in P0443.

The approach described in P2209r0 introduced a new `set_next` receiver protocol. This results in the need for new "bulk" variants of combinators such as `bulk_transform`. Our proposal requires no such new algorithms and composes cleanly with existing sender-based combinators such as `transform`.

4 Discussion

The interface proposed in P2181 used the `executor` concept, since it defines the operation of `bulk_schedule` in terms of `bulk_execute`. The interface proposed here uses the `scheduler` concept for consistency with the `schedule` interface. Since every executor is by definition a scheduler, as per P0443r13, this does not exclude any code that would have been valid under the prior definition. The definition of the `scheduler` concept in P0443 will need to be updated appropriately to include `bulk_schedule`.

We have also altered the order of arguments to `bulk_schedule`, placing the `prologue` argument first rather than last. Placing it last left room for convenience overloads making it optional, but this is no longer possible with the factory function being the last parameter. Leaving the executor/scheduler first makes the syntax consistent with method calls such as `ex.bulk_schedule(...)`; whereas, placing the `prologue` first makes the interface consistent with the range-style `operator|` chaining syntax. We have chosen the latter form of consistency in this paper.