Restore shared state to `bulk_execute` | P1993R1

Jared Hoberock | jhoberock@nvidia.com

2020-01-13

Abstract. In P1933, we argued that in order to integrate bulk execution with a senders and receivers-based programming model, `bulk_execute`'s (P0443) factory which produces a shared state should be eliminated. In retrospect, we realize that our rationale was flawed. This paper recommends that shared state be restored via an incoming sender parameter, and that state forwarded through the outgoing result sender.

Executive Summary

P0443R11's formulation of `bulk_execute` removed its shared factory parameter as we suggested in P1933.

In retrospect, we recognize that this was the wrong decision for P0443, because:

- Correct use of shared variables may require inefficient out-of-band dynamic allocation.
- Efficient use of shared variables may require out-of-band and non-standard executor-specific optimizations.
- `bulk_execute`'s resulting `sender_of<void>` is only useful for signaling work completion and cannot communicate a result.

Restoring a sender to `bulk_execute`'s interface brings shared variables and results in-band, enabling uniform use of `bulk_execute` and executor-specific optimizations. Additionally, forwarding the shared state through `bulk_execute`'s resulting `sender` allows the communication of a result value.

The existing `bulk_execute` interface of P0443R12:

```
template<class F>
sender_of<void> auto bulk_execute(F f, size_t n);
```

Proposed:

```
template<class F, sender S>
sender_of<__sender_value_t<S>> auto bulk_execute(F f, size_t n, S incoming);
```

The incoming `sender`'s value(s) are forwarded through the resulting `sender`.

Astute readers will recognize this formulation of `bulk_execute` as the moral equivalent of P0443's erstwhile futures-based `bulk_then_execute`.

Changelog

R1

- Replaced use of factory parameters with a single incoming sender based on feedback from Lee Howes.
- Clarified code examples.

R0

- Initial version, which argued for a factory parameter.

Shared variables

Shared variables represent transient shared state required by parallel algorithm implementations. Such state may include temporary scratchpads of partial results or objects necessary to synchronize execution agents. This state is discarded once the group of execution agents completes.

Past versions of P0443 passed shared variables indirectly via invocable factories rather than directly as parameters because

- Important types of shared state are not movable (e.g. synchronization primitives such as `std::atomic`) and cannot be passed as parameters.
- Envisioned extensions of `bulk_execute` to agent hierarchies assigns separate instances of shared state per group.

In P0443R12, any shared state used during a `bulk_execute` must be communicated out-of-band, as discussed below.

Analysis of rationale for eliminating shared factories

In P1933, we explored the idea of a "successor executor":

> The purpose of a successor executor is to make explicit the ordering of tasks submitted through an executor.

Such an executor would be the result of a call to `bulk_execute` and act as an ordering primitive allowing clients to sequence work created by calls to `execute` or `bulk_execute` by creating dependent work via the resulting successor. However, successor executors were not adopted by P0443R11. Instead, P0433R11's `bulk_execute` returns a sender and fulfills an equivalent role.

We anticipated that successor executors would create problems for shared factories and suggested eliminating them:

> 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.

Unlike unadopted successor executors, the senders returned by `bulk_execute` explicitly represent in-band join points suitable for ending the lifetime of the shared state. In other words, P0443R11 `bulk_execute`'s adoption of senders does not suffer from problems created by shared state.

Finally, we reasoned that the functionality of shared factories could be replicated efficiently and generically by clients of `bulk_execute`:

> Fortunately, efficient implementations of shared state we are familiar with may be efficiently reintroduced by a client via inspection of the executor's properties.

We now realize that reasoning was flawed, for reasons discussed below.

`bulk_execute`'s P0443R12 interface is insufficient

The following code examples demonstrate the use of shared variables with different types of P0443R12 bulk executors.

```
// inline executor
my_inline_executor ex;

int share_me = ...;

// capturing a stack variable by reference is safe because bulk_execute always blocks
bulk_execute(ex, [&share_me](size_t i) {
  foo(share_me);
}, n);
```

```
// new thread executor
my_new_thread_executor ex;

// heap-allocating a shared_ptr is necessary because bulk_execute never blocks
auto share_me = std::make_shared<int>(...);

// capture the shared_ptr by value
bulk_execute(new_ex, [share_me](size_t i) {
  foo(*share_me);
}, n);


// CUDA executor
my_cuda_executor ex;

// heap-allocating a unique_ptr via a special CUDA-specific allocator is necessary because
// * my_cuda_executor never blocks and
// * the GPU cannot normally access system memory
auto share_me = my_make_cuda_unique<int>(...);

// capture the raw pointer by value
int* raw_ptr = share_me.get();
bulk_execute(ex, [raw_ptr](size_t i) {
  foo(*raw_ptr);
}, n);

// arrange to destroy the unique_ptr after the bulk_execute completes using an out-of-band interface
ex.and_then_destroy_ptr_out_of_band(std::move(share_me));

// Alternatively:

bulk_execute(ex, [](size_t i) {
  // use out-of-band, CUDA-specific means to create a statically-allocated shared variable
  __shared__ int share_me;
  foo(share_me);
}, n);
```

Note that in all examples, both the surrounding code and the body of the lambda differs based on how the shared variable is allocated. This complicates correct and efficient use of shared variables in generic code.

After proposed

The following code examples demonstrate how the introduction of a `sender` enables the uniform use of shared variables and enables optimizations unavailable to `bulk_execute`'s P0443R12 interface.

```
// inline executor
struct my_inline_executor {
  ...
  template<class F, sender S>
  sender auto bulk_execute(F f, size_t n, S incoming) const {
    // statically-allocate the shared variable on the stack
    auto share_me = __sender_as_function(incoming)();

    // invoke f
    for(size_t i = 0; i < n; ++i) {
      f(i, share_me);
    }
```

```
    return just(move(share_me));
  }
};

// new thread executor
struct my_new_thread_executor {
  template<class F>
  void execute(F&& f) const {
    std::thread(decay_copy(f)).detach();
  }

  template<class F, sender S>
  sender auto bulk_execute(F f, size_t n, S incoming) const {
    if(n > 0) {
      // create a sender for the result
      __my_sender<__sender_value_t<S>> result;

      // get access to the result sender's storage so the
      // result can be written when it is ready
      __sender_value_t<S>* result_storage = result.get_ptr();

      execute([=]{
        // statically-allocate shared state on agent 0's stack
        auto share_me = __sender_as_function(incoming)();
        std::latch latch(n - 1);

        // create additional agents
        for(size_t i = 1; i < n; ++i) {
          // capture shared state by reference
          execute([f, &share_me, &b](size_t i) {
            // invoke f in additional agents
            f(i, share_me);

            // signal agent i's completion
            latch.count_down();
          }
        }

        // invoke f in agent 0
        f(0, share_me);

        // wait for additional agents to complete
        latch.wait();

        // move the state to the result sender
        *result_storage = move(share_me);
      });
    }

    return result;
  }
};
```

Importantly, this optimization is unavailable to P0443R12's `bulk_execute` because the `latch` required to manage the shared variable's life on the stack cannot be introduced by a client of `bulk_execute`'s P0443R12 interface.

```
// CUDA executor
```

```
struct my_cuda_executor {
  ...
  template<class F, class S>
  sender auto bulk_execute(F f, size_t n, sender S incoming) const {
    // create a sender for the result
    __cuda_sender<__sender_value_t<S>> result;

    // get access to the result sender's storage so the
    // kernel can write the result when it is ready
    __sender_value_t<S>* result_storage = result.get_ptr();

    my_cuda_executor_kernel<<<1,n>>>([=] __device__
    {
      // statically-allocate the shared variable in on-chip memory
      __shared__ __sender_value_t<S> share_me;

      // agent 0 puts the incoming state in __shared__ memory
      if(threadIdx.x == 0) {
        share_me = __sender_as_function(incoming)();
      }

      // all agents wait for the shared state to be ready
      __syncthreads();

      // all agents call f
      f(threadIdx.x, share_me);

      // all agents wait for f to complete
      __syncthreads();

      // end the lifetime of the shared state in agent 0
      if(threadIdx.x == 0) {
        // move the shared value to the result's storage
        *result_storage = move(share_me);

        // destroy the locally-stored copy
        destroy(share_me);
      }
    });

    return result;
  }
}
```

The addition of the `sender` parameter allows uniform usage of `bulk_execute` with shared state:

```
template<class E, T>
void generic_function(E ex, size_t n, T shared) {
  ex.bulk_execute([](size_t i, auto& share_me) {
    foo(share_me);
  }, n, just(shared));
}
```

Note that the orthogonal details of the sender returned by `bulk_execute` in these code examples is omitted for brevity.

Returning a result from `bulk_execute`

In prior revisions of P0443, `bulk_execute` was a one-way fire-and-forget operation. Any communication of work completion or results required out-of-band side effects on memory or non-standard extensions to the executor interface. With P0443R11's introduction of a `sender` result, `bulk_execute` became two-way. This enhancement allows communication of the bulk task's completion, but still requires side-effects to communicate a value, even though the `sender` is the obvious channel through which to communicate a result.

We should seize this enhancement opportunity by allowing the `sender` returned by `bulk_execute` to transport a value representing the result of the bulk task.

What is the result of an invocation of `bulk_execute`? It cannot simply be the result of the user's function, because each agent produces one of these. In principle, it could be some sort of reduction of these results. However, such a design would further complicate `bulk_execute` and place additional burden on the executor implementation. Moreover, a reduction would not be general enough to cover all use cases, anyway.

The simplest and most general thing to do is to forward the result of the incoming sender to the output, which allows the programmer to apply any sort of transformation they desire by attaching additional senders. This also allows the client to programmatically coordinate the agents created by `bulk_execute` to produce any result desired. The complete interface becomes:

```
template<class F, sender S>
sender_of<__sender_value_t<S>> auto bulk_execute(F f, size_t n, S incoming);
```

For example, consider a pedagogical integer sum reduction:

```
template<class E>
sender_of<int> auto compute_sum(E ex, vector<int>& values) {
  sender auto reduction = bulk_execute(ex,
    [&, final_agent = make_shared<void>()](size_t i, int& result, atomic<int>& partial) {
      // all agents contribute to the partial result
      partial += values[i];

      // the last agent out assigns the result
      if(final_agent.unique()) {
        result = partial;
      }
    },
    values.size(),
    __function_as_sender([]{ return make_pair(0, atomic<int>{}); })
  );

  // compose with a utility combinator which selects the first element
  // of the resulting pair, which is the resulting sum
  return reduction | __select_first;
}
```

This example is inefficient, but correct in general. Optimizations of `compute_sum` may be introduced generically by inspecting the `bulk_guarantee` property of the executor and introducing the appropriate kind of synchronization and shared state.

Finally, note that this proposed interface for `bulk_execute` is simply the restoration of P0443's old `bulk_then_execute` operation, with factories replaced by a single incoming sender and result `future` also replaced by a `sender`.

Summary

P1933's rationale to eliminate `bulk_execute`'s factory-based shared state was flawed and irrelevant to the design of `bulk_execute` which eventually arrived in P0443R11. Because of this, shared state must be restored via a sender to ensure that `bulk_execute` can serve as a low-level foundation for building parallel algorithm implementations. Our

proposed interface for `bulk_execute` is more complex than found in P0443R11, but the advantages of uniform usage, optimization opportunities, and composition with sender combinators are worth the added complexity. Moreover, the resulting sender becomes much more useful and will allow convenient higher-level interfaces to be built on top (e.g. see P1897) to hide the lower-level complexity of packaging incoming shared state and results as senders.

Wording

Wording for this proposal will be introduced in the next revision of this paper as a diff to P0443R12 and submitted to the post-Prague mailing.