Document	P3481R0
Date	2024-10-16
Reply To	Lewis Baker <lewissbaker@gmail.com> Lucian Radu Teodorescu <lucteo@lucteo.ro> Ruslan Arutyunyan <ruslan.arutyunyan@intel.com>
Audience	SG1, LEWG

Summarizing std::execution::bulk() issues

Abstract

When P2300R10 was merged into the working draft, a subsequent review of the github issue tracker discovered several outstanding issues relating to the use of the `bulk` algorithm that was in P2300R10 which were not addressed during the LEWG review. These issues mostly account for better expressing what `bulk` can and cannot do.

This paper tries to summarize these issues, give some ideas on the solution and promises to return back with a complete design.

Motivation

There are several shortcomings of the `bulk` as defined by [exec.bulk], that this paper tries to summarize:

- `bulk` shall allow default alternative implementations,
- `bulk` shall allow to call the functor concurrently,
- `bulk` shall allow cancellation,
- `bulk` shall have well defined exceptions behavior in non-sequential executions,
- `bulk`‘s functor shall be copyable,
- `bulk` shall take an execution policy paremeterallow specifying an execution policy,
- `bulk` shall be more suitable for the use case with many small iterations.

In [exec.bulk] we specify the behavior of `bulk`, in the absence of any customizations, as being equivalent to a C++ code. This has a few of drawbacks:

- Limits vendors ability to customize the behavior for improving performance (e.g., adaptive partitioning, vectorization, etc.)
- Doesn't allow out-of-order execution of the given functor.
- Doesn't allow cancellation of the `bulk` work.

Users of `bulk` expect that the given functor to be called concurrently. The current specification breaks this expectation.

Another reasonable expectation that is broken by `bulk` is to support cancellation of the work. If the environment of `bulk` has a cancellable token, and cancellation is signaled, the `bulk` implementation will simply ignore it (after the predecessor sender was completed).

In the default implementation of `bulk`, we don't have a need to copy the given functor `f`, as the code can sequentially access it by using a reference to it ([exec.bulk]). Considering customization, we don't specify what the `bulk` implementation can do with the given functor. This is especially problematic for implementation of `bulk` that rely on parallelism or may move computations to other execution contexts. By contrast, the `std::for_each` that takes an execution policy ([alg.foreach]) has a requirement that the given function needs to be `Cpp17CopyConstructible`.

Similarly, the current specification of `bulk` doesn't specify if the arguments passed to `f` are copyable. These arguments are produced by the input sender, and they need to be used in all the invocations to `f`, and in the invocation to `set_value`. If `f` and `set_value` are executed in different execution contexts, we may need to be able to copy the arguments.

The specification of `bulk` ([exec.bulk]) doesn't properly address exception propagation when we are not just forwarding the error from the previous sender. This spans on two levels:

1. In the default implementation, as soon as one exception is raised from calling `f`, it is forwarded to the error completion signal. But, if we want to make the default implementation customizable (see above issues), the same strategy cannot be applied. In a concurrent execution of `f`, detecting the first exception thrown may not make sense; also, we might have multiple exceptions thrown before we manage to complete with error.
2. The specification for customized versions of `bulk` doesn't mention what happens when `f` throws. We need to also specify the general expectations in this case, so that users of `bulk` know what can happen when they utilize this algorithm.

One shall note that, for customizations of `bulk` that execute `f` on different hardware, there may not be a way to forward the exceptions thrown by `f` to the receiver connected with the `bulk` sender.

Another omission from the specification is the execution policy allowed for the given functor `f`. Similar to `for_each`, users might expect that `bulk` also takes an execution policy as argument. This is needed for the following reasons:

- While not including an execution policy is fine for the default implementation of `bulk`, as specified, this is not sufficient if the customizations may choose different execution policies.
- A default execution policy isn't enough; there is hardware that supports both `par` and `par_unseq` execution policy semantics, while there is hardware that supports `par_unseq`. Standardizing the API without execution policies might occasionally give one or several hardware vendors an advantage and there is hardware that performs better with `par_unseq`.
- Not having the execution policy, may force the users to fall back to `for_each` for functions that only support `par` execution policy; this is an impediment to generic programming.

There is also a performance problem with the way `bulk` is defined for cases in which we have many iterations, and we would benefit from chunking together consecutive operations. Here is an example:

```
std::atomic<std::uint32_t> sum{0};
std::vector<std::uint32_t> data;
ex::sender auto s = ex::bulk(ex::just(), std::execution::par, 100'000,
  [&sum,&data](std::uint32_t idx) {
    sum.fetch_add(data[idx]);
  });
```

In this example, we are doing 100,000 atomic operations. A better implementation would allow us to have an atomic operation for each chunk of work, and allow each chunk of work to sum things locally; this may look like:

```
std::atomic<std::uint32_t> sum{0};
ex::sender auto s = ex::bulk_chunked(ex::just(), std::execution::par,
100'000,
  [&sum,&data](std::uint32_t begin, std::uint32_t end) {
    std::uint32_t partial_sum = 0;
    while (begin != end) {
      partial_sum += data[idx];
    }
    sum.fetch_add(partial_sum);
  });
```

Design discussions

Define `bulk` to match the following API:

```
// NEW: algorithm to be used as a basis operation
template<execution::sender Predecessor,
         typename ExecutionPolicy,
         std::integral Size,
         std::invocable<Size, Size, values-sent-by(Predecessor)...> Func
>
execution::sender auto bulk_chunked(Predecessor pred,
                                    ExecutionPolicy&& pol,
                                    Size size,
                                    Func f);

template<execution::sender Predecessor,
         typename ExecutionPolicy,
         std::integral Size,
         std::invocable<Size, values-sent-by(Predecessor)...> Func
>
execution::sender auto bulk(Predecessor pred,
                            ExecutionPolicy&& pol, // NEW
                            Size size,
                            Func f) {
    // Default implementation
    return bulk_chunked(
        std::forward<Predecessor>(pred),
        std::forward<ExecutionPolicy>(pol),
        size,
        [func=std::move(func)]<typename... Vs>(
            Size begin, Size end, Vs&... vs)
            noexcept(std::is_nothrow_invocable_v<Func, Size, Vs&&...>) {
            while (begin != end) {
                f(begin++, std::forward<Vs>(vs)...);
            }
        });
}
```

The `bulk_chunked` algorithm is a customization point returning a sender that describes a work with following properties:

- In the absence of any errors, call `f(b, e, args…)` zero or multiple times, such as, for every `i` in range [0, `size`), there is exactly one call to `f(b, e, args…)` with `i` ∈ [`b`, `e`). All the calls to `f` strongly happen before any call to `set_value`.
- If any call to `f` throws, then the sender completes on receiver `rcvr` with `set_error(std::move(rcvr), e)`, where `e` is one of the exceptions thrown, or is an exception derived from `std::runtime_error`. Before calling `set_error`, the

operation may call `f(b, e, args…)` zero or multiple times, such as, for every `i` in range [0, `size`), there is maximum one call to `f(b, e, args…)` with `i` ∈ [`b`, `e`).

The `bulk` algorithm is a customization point. The default implementation calls `bulk_chunked` as shown above.

Use chunked version as a basis operation

To address the performance problem described in the motivation, we propose to add a chunked version for `bulk`. This allows implementations to process the iteration space in chunks, and take advantage of the locality of the chunk processing. This is useful when publishing the effects of the functor may be expensive (the example from the motivation) and when the given functor cannot be inlined.

The implementation of `bulk_chunked` can use dynamically sized chunks that adapt to the workload at hand. For example, computing the results of a Mandelbrot fractal on a line is an unbalanced process. Some values can be computed very fast, while others take longer to compute.

Passing a range as parameters to the functor passed to `bulk_chunked` is similar to the way Intel TBB's `parallel_for` functions (see https://www.intel.com/content/www/us/en/docs/onetbb/developer-guide-api-reference/2021-6/parallel-for.html).

An implementation of `bulk` can be easily built on top of `bulk_chunked` without losing performance (as shown in the Proposal section).

Using execution policies

As discussed in the Motivation section, not having the possibility to specify execution policies for the `bulk` algorithm is a downside. If the default execution policy for regular algorithms is `seq`, for `bulk` (and `bulk_chunked`) it makes more sense for the default to be `par` or `par_unseq`. But, there are cases in which any of these two would be more appropriate. Thus, we cannot choose a default policy.

One criticism of this solution is that each invocation of `bulk` needs to contain an extra parameter that is typically verbose to type. But there is always an easy workaround: the user can define a thin wrapper on top of `bulk` / `bulk_chunked` that calls the algorithm with the default execution policy. Something like:

```
auto easy_bulk(auto prev, auto size, auto f) {
```

```
  return std::execution::bulk(std::execution::par, prev, size, f);
}
```

Another idea considered by the authors is to provide versions of the algorithms that have the execution policies already baked in. Something like:

```
auto bulk_seq(auto prev, auto size, auto f);
auto bulk_unseq(auto prev, auto size, auto f);
auto bulk_par(auto prev, auto size, auto f);
auto bulk_par_unseq(auto prev, auto size, auto f);

auto bulk_chunked_seq(auto prev, auto size, auto f);
auto bulk_chunked_unseq(auto prev, auto size, auto f);
auto bulk_chunked_par(auto prev, auto size, auto f);
auto bulk_chunked_par_unseq(auto prev, auto size, auto f);
```

We dropped this idea, as this isn't scalable. Moreover, the user can easily add these overloads in their codebase.

Relaxing the default implementation

As mentioned in the Motivation section, the way the un-customized `bulk` is defined by [exec.bulk] is too constrained and doesn't meet typical user's expectations. With this definition, the default `bulk` implementation needs to be serial.

We would want to relax this over-specification of the algorithm and specify the minimal properties that need to hold for the implementation. We would want to specify that the functor needs to be called for the entire range, but leave the details to the implementation. This allows the implementation to fine-tune the algorithm and make it more performant for the envisioned usage/hardware.

Such a relaxed specification should allow:

- Calling the given functor concurrently.
- Making decay copies of the values produced by the previous sender (if the values produced are copyable).
- Handling cancellation inside the `bulk` algorithm.

With this change we also require that the given functor be copy-constructible. This will make `bulk` require the same type of function as a `for_each` with an execution policy.

The derided direction is to specify that `bulk` can react to cancellation requests, but leave the details of how cancellation is implemented to be unspecified. One implementation might choose to check for cancellation for every iteration, while others may want to check for cancellation on

processing a batch of items. It shall be also possible for vendors to completely ignore cancellation requests (if, for example, supporting cancellation would actually slow down the main uses-cases that the vendor is optimizing for).
Checking the cancellation token every iteration may be expensive for certain cases. A cancellation check requires an acquire memory barrier (which may be too much for really small functors), and might prevent vectorization. Thus, implementations may want to check the cancellation every N iterations; N can be a statically known number, or can be dynamically determined.

In our proposal, `bulk` is implemented in terms of `bulk_chunked`. Thus, we define the minimal constraints on `bulk_chunked`.

Exception handling

While there are multiple solutions to specify which errors can be thrown by `bulk`, the most sensible ones seem to be:

1. pick an arbitrary exception thrown by one of the invocations of `f` (maybe using a atomic operations to select the first thrown exception),
2. reduce the exceptions to another exception that can carry one or more of the thrown exceptions using a user-defined reduction operation,
3. allow implementations to produce a new exception type (e.g., to represent failures outside of `f`, or to represent failure to transmit the original exception to the receiver)

One should note that option 2 can be seen as a particular case of option 3.

Also, option 2 seems to be more complex than option 1, without providing any palpable benefits to the users.

The third option is very useful when implementations may fail outside of the calls to the given functor. Also, there may be cases in which exceptions cannot be transported from the place they were thrown to the place they need to be reported.

Based on the experience with the existing parallel frameworks we incline to recommend the option one because

- In a parallel execution the common behavior is non-deterministic by nature. Thus, we can peak arbitrary exception to throw
- Catching any exception already indicates that something went wrong, thus a good implementation might initiate a cancellation mechanism to finish already failed work as soon as possible.

Other considerations

Dropping a predecessor

It is still an open question whether `bulk_chunked` whether `bulk`. The options are:

- No predecessor API
- Only predecessor API (status quo)
- Both with and without predecessor

The questions are:

- What are the user expectations?
- What are more common use-cases?
- In case of both, which one is the basis operation?
- In case of both, which one is customizable?
- etc.

There is too much room for a design for a moment. We didn't explore it well-enough, thus we are not ready to recommend any design.

Please find the example below, how both `bulk_chunked` both with and without predecessor might coexist:

```
template<execution::sender Predecessor,
         typename ExecutionPolicy,
         std::integral Size,
         std::invocable<Size, Size, values-sent-by(Predecessor)...> Func
>
execution::sender auto bulk_chunked_with_pred(Predecessor pred,
                                              ExecutionPolicy&& pol,
                                              Size size,
                                              Func f) {
  return let_value(pred, [func, size, pol] (auto&&... vals) {
    return bulk_chunked(pol, size, [&](auto b, auto e) {
      func(b, e, vals...);
    });
  });
}
```

Conclusion

We identified several significant issues with `std::execution:bulk` API, which are better to be fixed before C++26 timeframe. Some of the solutions are likely problematic to add later if

`bulk` is going to be released in the upcoming standard. At the same time, more exploration is required before proposing a good design for `bulk`.