Document number: P1017R0
Date: 20180506
Project: Programming Language C++, SG1
Authors:
    Hartmut Kaiser
    John Biddiscombe
Emails:
    hartmut.kaiser@gmail.com
    biddisco@cscs.ch
Reply to: hartmut.kaiser@gmail.com

1 Executors should be variadic

This paper responds to the proposed resolution of a GitHub issue Executors:Issue9 posted 2 years ago relating to the currently discussed executor design paper P0443.

1.1 Introduction

The proposed executor interfaces (see P0443) are to be the fundamental bases for implementing parallelism for today's and - more importantly - any future application. Thus it is of utmost importance to come up with a design that is as flexible as possible and does not impose any limitations on possible future use cases.

1.2 Rationale

This paper attempts to provide a rationale for changing the current executor design (as outlined in P0443) such that all execution functions exposed by the executors should be variadic instead of them accepting nullary functions. The required specification changes are minimal and do not change any of the semantics proposed by P0443.

The following sections focus on various aspects and provide arguments in favor of exposing variadic execution interfaces for all the proposed types of executors. In brief, the current proposal disrupts consistency, inhibits usability, and prevents runtime information from being passed to the executor.

1.2.1 Consistency with `async` and `thread`

Well-designed language standards need to be uniform and non-contradictory. The definitions of interfaces pertaining to similar functionalities should be as consistent as possible. In C++17 we have three API's related to scheduling and launching of tasks: `async`, `thread`, and `invoke` (the latter exposing a synchronous execution interface). All of these functions are specified to accept a variadic set of arguments. Requiring all executors to utilize only nullary function objects contradicts this fundamental requirement of standardization. We claim that the executor interfaces should be specified to be variadic - if nothing else - at least to ensure the consistency of the standard.

1.2.2 Current design inhibits usability

In our opinion, the arguments presented by the authors of P0443 defending the use of nullary function interfaces are not compelling given the cost of usability. This section attempts to address some of these concerns and provides alternative design choices that would allow for equivalent functionalities as proposed by P0443, but without the limitation of requiring for the execution interfaces to be limited to using nullary functions. (for all quotes in this section, please see Executors:Issue9)

1.2.2.1 Claim: variadics introduce complexity

> "Separate, variadic bulk analogues of async & invoke make sense to me, but I think they are out of scope for this minimal proposal."

The proposed P0443 is everything but minimal at this point. The specification of variadic execution interfaces would add a very small amount of additional wording without any change in exposed semantics. Moreover, the specification of variadic interface would simplify the specification of some of the APIs. For instance, as we will show below, there is no need for special purpose arguments to be provided to `bulk_execute` and `bulk_twoway_execute` (like the shared factories) if the API was specified to be variadic. Removing those would simplify the specification and streamline the overall design.

1.2.2.2 Claim: we need bulk interfaces

> "Moreover, I think it's important to reserve additional parameter slots ... for parameters which carry semantics that executors will actually need to customize."

Also:

> "Finally, I'm concerned variadic parameters would preclude an important additional functionality that bulk executors will want to provide. We envision that a future proposal will want to introduce the ability to create hierarchical executors to support things like GPUs and executors that are composed of a composition of other, more primitive executors."

These quotes refer to special arguments as proposed for the bulk interfaces, namely factory functions for the generated results etc. We argue that not only such very specific arguments (like the proposed factory functions) are limiting the generality of the proposed APIs (as not all bulk executors might have a need for factory functions or similar), they also unnecessarily leak implementation details of a limited type of executors to the general users of those interfaces.

Over-specifying the meaning of certain arguments to be passed to the executors inhibit the user's ability to express general use cases, especially for those we don't anticipate today.

The special arguments today - and any possible special argument that might be required in the future - can alternatively easily passed to an underlying executor through a modular layered design of wrapping executors.

Here is a small example demonstrating this. Let's assume, we have a two-way-executor (for simplicity we'll show only the implementation of a blocking bulk execute implementation, the same technique can be applied to all other executor APIs). We also assume that it exposes functionality which is callable through a *variadic* API:

```
struct two_way_executor
{
    template <typename F, typename ...Ts>
    void bulk_twoway_execute(F && f, size_t n, Ts &&... ts)
    {
        for (size_t i = 0; i != n; ++i)
        {
            invoke(f, i, ts...);
        }
    }
};
```

Now it is trivial to achieve the functionality that is relying on factory functions for shared data by defining another executor that exposes functionality similar to `bulk_twoway_execute` as specified by P0443 without any special arguments, completely relying on a variadic bulk execution interface:

```
template <typename Executor, typename RF, typename SF>
struct shared_data_executor
```

```
{
    template <typename F, typename ...Ts>
    auto bulk_twoway_execute(F && f, size_t n, Ts &&... ts)
    {
        // implement as needed using the factory functions, while relying on
        // wrapped executor to do the work
        auto shared_results = rf(n);
        auto shared_data = sf();

        // PLEASE NOTE: from the standpoint of 'f' there is no discernible
        // difference to a direct implementation as mandated by P00443
        return exec.bulk_twoway_execute(forward<F>(f), n,
            ref(shared_results), ref(shared_data),
            forward<Ts>(ts)...);
    }

    Executor exec;
    RF rf;
    SF sf;
};
```

We invoke almost as before:

```
size_t shape = 100;
auto f = [](size_t, auto& shared_results, auto& shared_data) { ... };
auto result_factory = [](size_t n) {...};
auto shared_factory = []() {...};

auto shared_data_exec = shared_data_executor{
    two_way_executor{}, move(result_factory), move(shared_factory)
};

shared_data_exec.bulk_twoway_execute(f, shape, ...);
```

Please note that from the standpoint of the scheduled function `f` there is no discernable difference to a direct implementation of the executor as mandated by P0443 (i.e. compared to using the nullary interfaces).

We do not even need the currently proposed special arguments for the bulk interfaces, more-so by applying the described implementation techniques we can clearly avoid adding more special arguments in the future. Thus the provided argumentation that we cannot have variadics today because that would prohibit adding other special arguments in the future is invalid.

By making the executor interfaces variadic we open the path for seamless future extensions to the executor interface semantics without having to break backwards compatibility.

1.2.2.3 Claim: we have no motivating use case

> "It's also not clear that there is any reason that an executor author would want to customize how variadic parameters get forwarded."

We don't know of any use case for an executor to customize variadic parameter forwarding either. We will however provide a valid and very important use case for an executor having to be able to access (read) those parameters for performance reasons. We will elaborate on this use case in this section.

1.2.3 An executor can always package arguments into a nullary object if required

In general, a user shouldn't have to perform tasks like parameter forwarding that can be applied mechanically and in a general purpose fashion. In discussions the authors of P0443 have explained that the idea would be to leave the mundane task of wrapping arguments into nullary function objects to higher level control structures like (a future version of) `async` or `invoke`, thus the executors wouldn't have to do that.

We believe, however, that instead of relying on future convenience interfaces to perform the task of packaging arguments, executors can always perform this task themselves if an underlying execution context requires this. Not all executors may even have to perform that task to begin with.

The discussed executor interfaces are fundamental to today's and all future parallelism library features and many users (and not only library writers) will use those directly in order to implement their own functionalities. These users would have to repeatedly write code packaging the arguments of their functions into nullary objects.

From the standpoint of library developers, the enforcement of nullary function interfaces will cause unnecessary implementation complexity on things like the parallel algorithms. Why sprinkle parameter packing/unpacking all over the place where executors are (directly) used?

Making the executor interfaces nullary requires the use of a lambda which is error prone (arguments must be forwarded into the capture, must be moved/forwarded out of the capture when calling the function, and the lambda would have to be mutable), for instance:

```
template <typename F, typename ... Args>
future<result_of_t<F(Args...)>> foo(F && f, Args && ... args)
{
    return my_exec.twoway_execute(
        [f = forward<F>(f), ...args = forward<Args>(args)]() mutable
        {
            return invoke(forward<F>(f), forward<Args>(args)...);
        });
}
```

Alternatively we would have to introduce a lightweight `bind` helper that moves its arguments, e.g. something like:

```
template <typename F, typename ... Args>
future<result_of_t<F(Args...)>> foo(F && f, Args && ... args)
{
    return my_exec.twoway_execute(
        deferred_call(forward<F>(f), forward<Args>(args)...)
    );
}
```

All of this increases implementation complexity.

At the same time, with a variadic interface the same function implementation would just look like:

```
template <typename F, typename ... Args>
future<result_of_t<F(Args...)>> foo(F && f, Args && ... args)
{
    return my_exec.twoway_execute(forward<F>(f), forward<Args>(args)...);
}
```

For these reasons, we propose that making the interfaces variadic adds a small amount of specification complexity, but massively reduces implementation complexity of all code that uses executors. The executors themselves would not necessarily become more complex (`twoway_execute` in the simplest blocking case would call `invoke` and could use `thread` if non-blocking, both of which are already variadic).

1.2.4 Executors can't see original arguments (possible performance problem)

On conventional architectures, execution contexts will frequently consist of thread pools that are managed by some runtime framework that is responsible for scheduling of work/tasks as well as other resource management functions. To optimize performance, users will wish to pass parameters via executors to influence task placement, priorities or other arbitrary properties. The schedulers that must perform the underlying execution of functions/tasks do not yet (or possibly never will) have a well-defined or standardized API and it is therefore not yet known what the nature of the parameters required for scheduling might be.

Earlier proposals such as P0113 suggested that simple customization of executors (for priority) could be used as follows

```
priority_scheduler sched;
auto low = sched.get_executor(0);
auto high = sched.get_executor(9);
...
async(low, []{ cout << "1\n"; });
async(high, []{ cout << "9\n"; });
```

and the executor can then forward the `priority` on to the underlying scheduler during the launching process (this assumes that the executor and scheduler have been developed cooperatively and therefore the executor is aware of the scheduler capabilities beforehand). This approach is well suited to the concept of a model of executors that are lightweight, copyable and do not introduce the possibility of races or exceptions since the executor state is `const` and all tasks created with the executor inherit the same properties. In this example, all tasks launched using the `low` priority executor are treated identically, but differently from all tasks launched with the `high` priority executor.

There are use cases that require a more complex type of scheduling that might depend on the *parameters/arguments of the task* as well as the task itself. When the argument values are known at compile time, then the same approach as above may be used, an executor may be constructed with those values (or some function derived from them) and the executor is then used directly. If the parameters/arguments are not known at compile time, then some property/value derived from them must be computed at run time and passed through to the scheduler via the executor interface. This breaks the rule that all tasks launched with the executor are treated equally and requires the use of a more complex executor and interface. An example of this is now presented as a motivating example for a variadic executor implementation.

1.2.5 Task dispatch using argument introspection

Consider a task that operates on two items `matrix1` and `matrix2` that have had their memory allocated on a dual socket node with separate memory controllers handlng the two NUMA domains. The task may make a large number of memory accesses to the matrices which may not fit in cache memory and it is therefore desirable to schedule it to run on one of the cores assigned to the same memory controller/domain as the matrix allocation - if the matrices are the result of other calculations that have taken place within other tasks, then determing at compile time which domain the memory is assigned to for each task may prove difficult or impossible. One solution might be to provide two thread pools, one for each NUMA domain, and assign tasks to each via an executor created for each pool - however, this solution shifts the problem to selecting which executor to use and may, in addition, reduce efficiency if task stealing between pools is not possible.

We solve the problem in HPX by using a `guided_pool_executor` (so named because it is an executor that places tasks on a thread pool with hints to guide task placement) - the thread pool may span multiple NUMA domains and the scheduler responsible for managing tasks on the pool is capable of placing tasks on one or other of the NUMA domains on request.

Assuming our example task has the following signature

```
auto compute(Matrix && m1, Matrix && m2)
{
```

```
    return memory_intensive_calculation(move(m1), move(m2));
}
```

HPX provides a custom implementation of `when_all(...).then(...)` for futures named `dataflow(...)` that simplifies slightly the creation of continuations for multiple futures - using this construct we can write

```
hpx::future<Matrix> future_m1 = ...
hpx::future<Matrix> future_m2 = ...
...
hpx::dataflow(guided_executor,
    [](hpx::future<Matrix> && m1, hpx::future<int> && m2)
    {
        return compute(m1.get(), m2.get());
    },
    move(future_m1), move(future_m2)
);
```

The executor is defined as follows

```
// NOTE: the default thread pool is being used in this example
guided_pool_executor<matrix_hint_type> guided_executor("default");
```

and the `matrix_hint_type` type that is used to customize the executor is defined in this example as

```
template <>
struct pool_numa_hint<example_tag>
{
  template <typename Matrix>
  int operator()(Matrix const& matrix1, Matrix const& matrix2) const
  {
      int dom1 = matrix1.get_numa();
      int dom2 = matrix2.get_numa();
      if (dom1 == dom2) return dom2; // trivial example
      return dom1;
  }
};
using matrix_hint_type = pool_numa_hint<example_tag>;
```

What makes this construction powerful is that, when the `dataflow` continuation is passed to the executor using a variadic signature, the two matrix futures are available to the `guided_pool_executor`, it delays passing them through to the scheduler, instead waiting for them to become ready, then it extracts the values from the futures, calls the numa hint function to find which numa domain is preferred for the matrix objects, then passes the arguments on to an internal executor that in turn wraps the task into a bound nullary function for the scheduler to handle.

This example uses a continuation because it highlights the problem that the matrix futures could have been generated by functions that are far removed from the code responsible for the compute task and the programmer can delay the decision of which NUMA domain to assign the task to until runtime when the objects can be queried before scheduling. In the example shown, we are able to pass NUMA information to the scheduler for tasks that benefit from it because we have codeveloped the schedulers and executors to support this feature. Other hint types can be supported, it core/socket/domain/device/other identifiers were needed to improve performance.

Using bound functions for the executor interface hides the arguments from the executor and prevents this kind of introspection from taking place.

1.2.6 Existing implementation experience

HPX is a library that implements many - if not all - of the existing and proposed standard interfaces related to concurrency and parallelism (such as the C++17 parallel algorithms and the Concurrency TS - ISO/IEC TS 19571:2016). There we have implemented an earlier version of P0443 with the extension of making the interfaces variadic. All of the APIs exposed by HPX that require scheduling new tasks are implementedusing this variadic executor API (such as the implementation of task_block, all of the parallel algorithms, etc.). These constructs have been used by many users over the last years and have proven to be stable, performant, and cross platform. No serious issues have been reported.

We pose this proves that the currently standardized standard library features as well as many of the currently features proposed related to parallelism and concurrency can be implemented with good success on top of variadic executor interfaces.

2 Conclusion

Overall, based on the discussion above we propose changing the currently discussed executor design as described in P0443 to support for all execution functions to be variadic.

3 Acknowledgments

The authors thank Augustin K-ballo Berge and Adrian Serio for helpful comments and suggestions.