Parallel Algorithms Need Executors | N4406

Jared Hoberock Michael Garland Olivier Giroux
{jhoberock, mgarland, ogiroux}@nvidia.com

2015-04-10

1 Introduction

The existing Version 1 of the Parallelism TS exposes parallel execution to the programmer in the form of standard algorithms that accept execution policies. A companion executor facility would both provide a suitable substrate for implementing these algorithms in a standard way and provide a mechanism for exercising programmatic control over where parallel work should be executed.

The algorithms and execution policies specified by the Parallelism TS are designed to permit implementation on the broadest range of platforms. In addition to preemptive thread pools common on some platforms, implementations of these algorithms may want to take advantage of a number of mechanisms for parallel execution, including cooperative fibers, GPU threads, and SIMD vector units, among others. This diversity of possible execution resources strongly suggests that a suitable abstraction encapsulating the details of how work is created across diverse platforms would be of significant value to parallel algorithm implementations. We believe that suitably defined executors provide just such a facility.

As we observed in paper N3724 (Section 2.1.1), the execution policies of the current Parallelism TS provide a means of specifying *what* parallel work can be created, but do not address the question of *where* this work should be executed. Executors can and should be the basis for exercising this level of control over execution, where that is desired by the programmer.

In this paper, we describe a minimal set of features required for an executor facility to support the implementation of parallel algorithms and compose with execution policies. We also sketch one possible concrete design supporting these features. We propose that an executor facility based on this design be added to Version 2 of the Parallelism TS in order to provide both a means of implementing parallel algorithms and a means of controlling their execution.

2 Summary of Proposed Functionality

An executor is an object responsible for creating execution agents on which work is performed, thus abstracting the (potentially platform-specific) mechanisms for launching work. To accommodate the goals of the Parallelism TS, whose algorithms aim to support the broadest range of possible platforms, the requirements that all executors are expected to fulfill should be small. They should also be consistent with a broad range of execution semantics, including preemptive threads, cooperative fibers, GPU threads, and SIMD vector units, among others.

The following points enumerate what we believe are the minimal requirements for an executor facility that would support and interoperate with the existing Parallelism TS.

Uniform API. A program may have access to a large number of standard, vendor-specific, or other third-party executor types. While various executor types may provide custom functionality in whatever way seems

best to them, they must support a uniform interface for the core functionality required of all executors. We outline an `executor_traits` mechanism as a concrete design that satisfies this requirement.

Compose with execution policies. Execution policies are the cornerstone of the parallel algorithm design found in the Parallelism TS. We describe below how we believe executor support can be incorporated into execution policies.

Advertised agent semantics. Executors should advertise the kind of execution agent they create. For example, it should be possible to distinguish between an executor that creates sequential agents and another that creates parallel agents. We introduce a notion of *execution categories* to categorize the kinds of execution agents executors create, and which further clarifies the connection between executors and execution policies.

Bulk agent creation. High-performance implementations of the parallel algorithms we aim to support may often need to create a large number of execution agents on platforms that provide a large number of parallel execution resources. To avoid introducing unacceptable overhead on such platforms, executors should provide an interface where a single invocation can cause a large number of agents to be created. Consolidating the creation of many agents in a single call also simplifies the implementation of more sophisticated executors that attempt to find the best schedule for executing and placing the work they are asked to create. The functionality we sketch below provides a bulk interface, creating an arbitrary number of agents in a single call and identifying the set of agents created with an integral range.

Standard executor types. There should be some, presumably small, set of standard executor types that are always guaranteed to be available to any program. We detail what we believe to be the minimal set of executor types necessary to support the existing Parallelism TS.

Convenient launch mechanism. Most algorithm implementations will want a convenient way of launching work with executors, rather than using the executor interface directly. The most natural way of addressing this requirement is to introduce an executor-based overload of `std::async`, an example of which we sketch in subsequent sections.

With the additions we propose in this paper, the following use cases of the parallel algorithms library are possible:

```
using namespace std::experimental::parallel::v2;

std::vector<int> data = ...

// legacy standard sequential sort
std::sort(data.begin(), data.end());

// explicitly sequential sort
sort(seq, data.begin(), data.end());

// permitting parallel execution
sort(par, data.begin(), data.end());

// permitting vectorized execution as well
sort(par_vec, data.begin(), data.end());

// NEW: permitting parallel execution in the current thread only
sort(this_thread::par, data.begin(), data.end());

// NEW: permitting vector execution in the current thread only
sort(this_thread::vec, data.begin(), data.end());

// NEW: permitting parallel execution on a user-defined executor
my_executor my_exec = ...;
std::sort(par.on(my_exec), data.begin(), data.end());
```

3 Motivating Examples

The remainder of this paper outlines a concrete executor facility that satisfies the minimum requirements we have described in the previous section. Before providing the detailed design, we examine some motivating examples that use our executor interface.

3.1 Implementing `for_each_n`

The initial motivation for the interface we describe is to support the implementation of algorithms in the Parallelism TS. As an example, the following code example demonstrates a possible implementation of `for_each_n` for random access iterators using the executor interface we define below.

```
template<class ExecutionPolicy, class InputIterator, class Function>
Iterator for_each_n(random_access_iterator_tag,
                    ExecutionPolicy&& policy, InputIterator first, Size n, Function f)
{
  using executor_type = typename decay_t<ExecutionPolicy>::executor_type;

  executor_traits<executor_type>::execute(policy.executor(), [=](auto idx)
  {
    f(first[idx]);
  },
  n
  );

  return first + n;
}
```

The design we suggest associates an executor with each execution policy. The implementation above uses the executor associated with the policy provided by the caller to create all its execution agents. Because `for_each_n` manipulates all executors it encounters uniformly via `executor_traits`, the implementation is valid for any execution policy. This avoids the burden of implementing a different version of the algorithm for each type of execution policy and permits user-defined execution policy types, leading to a substantial reduction in total code complexity for the library.

3.2 Composing higher-level user-defined codes with executors

The following code example is presented as a motivating example of paper N4143:

```
template<typename Exec, typename Completion>
void start_processing(Exec& exec, inputs1& input, outputs1& output, Completion&& comp)
{
  latch l(input.size(), forward<Completion>(comp));

  for(auto inp : input)
  {
    spawn(exec,
      [&inp]
      {
        // process inp
      },
```

```
      [&l]
      {
        l.arrive();
      }
    );
  }

  l.wait();
}
```

This code can be dramatically simplified in a framework where executors interoperate cleanly with execution policies and parallel algorithms. The equivalent code using our design is:

```
template<typename Exec, typename Completion>
void start_processing(Exec& exec, inputs1& input, outputs1& output, Completion&& comp)
{
  transform(par.on(exec), input.begin(), input.end(), output.begin(), [](auto& inp)
  {
    // process inp
  });

  forward<Completion>(comp)();
}
```

Because `transform`'s invocation synchronizes with its caller, there is no longer a need to introduce a low-level `latch` object into the high-level `start_processing` code. Moreover, the code avoids the potentially high overhead of launching individual tasks within a sequential `for` loop, replacing that with an abstract parallel algorithm.

4 Proposed functionality in detail

In this section, we outline a concrete design of an executor facility that meets the requirements laid out above and provides the interface used in the preceding motivating examples. We focus specifically on the aspects of a design that address the requirements laid out at the beginning of this paper. A complete design would also certainly provide functionality beyond the minimal set proposed in this section. We survey some possible directions for additional functionality in an appendix.

4.1 Uniform manipulation of executors via `executor_traits`

Executors are modular components for requisitioning execution agents. During parallel algorithm execution, execution policies generate execution agents by requesting their creation from an associated executor. Rather than focusing on asynchronous task queueing, our complementary treatment of executors casts them as modular components for invoking functions over the points of an index space. We believe that executors may be conceived of as allocators for execution agents and our interface's design reflects this analogy. The process of requesting agents from an executor is mediated via the `executor_traits` API, which is analogous to the interaction between containers and `allocator_traits`.

A sketch of our proposed `executor_traits` interface follows:

```
template<class Executor>
class executor_traits
```

```
{
  public:
    // the type of the executor
    using executor_type = Executor;

    // the category of agents created by the bulk-form execute() & async_execute()
    // executor_type::execution_category if it exists;
    // otherwise parallel_execution_tag
    using execution_category = ...

    // the type of index passed to the function by the bulk-form execute() & async_execute()
    // Executor::index_type if it exists;
    // otherwise std::size_t
    using index_type = ...

    // the type of bulk-form execute() & async_execute()'s shape parameter
    // Executor::shape_type if it exists;
    // otherwise std::size_t
    using shape_type = ...

    // the type of future returned by async_execute()
    // Executor::future<T> if it exists;
    // otherwise std::future<T>
    template<class T>
    using future = ...

    // returns the largest shape async_execute(ex, f, shape) can accomodate
    // Executor::max_shape(f) if it exists;
    // otherwise the largest value representable by shape_type
    template<class Function>
    static shape_type max_shape(const executor_type& ex, const Function& f);

    // singleton form of asynchronous execution agent creation
    // asynchronously creates a single function invocation f()
    // returns f()'s result through a future
    // calls ex.async_execute(ex, f) if it exists;
    // otherwise async_execute(ex, f, shape_type{1})
    template<class Function>
    static future<result_of_t<Function()>> async_execute(executor_type& ex, Function f);

    // like singleton form of async_execute(), but synchronizes with the caller
    template<class Function>
    static result_of_t<Function()> execute(executor_type& ex, Function f);

    // bulk form of asynchronous execution agent creation
    // asynchronously creates a group of function invocations f(i)
    // whose ordering is given by execution_category
    // i takes on all values in the index space implied by shape
    // all exceptions thrown by invocations of f(i) are reported in a manner consistent
    // with parallel algorithm execution through the returned future.
    template<class Function>
    static future<void> async_execute(executor_type& ex, Function f, shape_type shape);

    // like bulk form of async_execute(), but synchronizes with the caller
```

```
    template<class Function>
    static void execute(executor_type& ex, Function f, shape_type shape);
};

// inherits from true_type if T satisfies the Executor concept implied by executor_traits;
// otherwise false_type
template<class T>
struct is_executor;
```

With `executor_traits`, clients manipulate all types of executors uniformly:

```
executor_traits<my_executor_t>::execute(my_executor, [](size_t i){ // perform task i }, n);
```

This call synchronously creates a *group* of invocations of the given function, where each individual invocation within the group is identified by a unique integer `i` in [`0`,`n`). Other functions in the interface exist to create groups of invocations asynchronously and support the special case of creating a singleton group, resulting in four different combinations.

Though this interface appears to require executor authors to implement four different basic operations, there is really only one requirement: `async_execute()`. In practice, the other operations may be defined in terms of this single basic primitive. However, some executors will naturally specialize all four operations for maximum efficiency.

For maximum implementation flexibility, `executor_traits` does not require executors to implement a particular exception reporting mechanism. Executors may choose whether or not to report exceptions, and if so, in what manner they are communicated back to the caller. However, we expect many executors to report exceptions in a manner consistent with the behavior of execution policies described by the Parallelism TS, where multiple exceptions are collected into an `exception_list`. This list would be reported through `async_execute()`'s returned `future`, or thrown directly by `execute()`.

4.2 Execution categories

Execution categories categorize execution agents by the order in which they execute function invocations with respect to neighboring function invocations within a group. Execution categories are represented in the C++ type system by execution category tags, which are empty structs similar to iterator categories. Unlike execution policies, which encapsulate state (such as an executor) describing how and where execution agents are created, and may make guarantees over which threads are allowed to execute function invocations, execution categories are stateless and focused only on describing the ordering of groups of function invocations. A partial order on execution categories exists; one may be *weaker* or *stronger* than another. A library component (such as an executor) which advertises an execution category which is not weaker than another may be substituted for the other without violating guarantees placed on the ordering of function invocations.

The minimum set of execution categories necessary to describe the ordering semantics of the Parallelism TS are:

- `sequential_execution_tag` - Function invocations executed by a group of sequential execution agents execute in sequential order.
- `parallel_execution_tag` - Function invocations executed by a group of parallel execution agents execute in unordered fashion. Any such invocations executing in the same thread are indeterminately sequenced with respect to each other. `parallel_execution_tag` is weaker than `sequential_execution_tag`.
- `vector_execution_tag` - Function invocations executed by a group of vector execution agents are permitted to execute in unordered fashion when executed in different threads, and unsequenced with respect to one another when executed in the same thread. `vector_execution_tag` is weaker than `parallel_execution_tag`.

4.3 Standard executor types

In addition to other executor types under consideration in various proposals, we propose the following standard executor types. Each type corresponds to the execution policy implied by its name:

- `sequential_executor` - creates groups of sequential execution agents which execute in the calling thread. The sequential order is given by the lexicographical order of indices in the index space.
- `parallel_executor` - creates groups of parallel execution agents which execute in either the calling thread, threads implicitly created by the executor, or both.
- `this_thread::parallel_executor` - creates groups of parallel execution agents which execute in the calling thread.
- `vector_executor` - creates groups of vector execution agents which execute in either the calling thread, threads implicitly created by the executor, or both.
- `this_thread::vector_executor` - creates groups of vector execution agents which execute in the calling thread.

4.3.1 Example `this_thread::vector_executor` implementation

The following code example demonstrates a possible implementation of `this_thread::vector_executor` using `#pragma simd`:

```
namespace this_thread
{

class vector_executor
{
  public:
    using execution_category = vector_execution_tag;

    template<class Function, class T>
    void execute(Function f, size_t n)
    {
      #pragma simd
      for(size_t i = 0; i < n; ++i)
      {
        f(i);
      }
    }

    template<class Function, class T>
    std::future<void> async_execute(Function f, size_t n)
    {
      return async(launch::deferred, [=]
      {
        this->execute(f, n);
      });
    }
};

}
```

In our experiments with Clang and the Intel compiler, we found that the performance of codes written using `this_thread::vector_executor` is identical to an equivalent SIMD for loop.

4.4 Execution policy support for executors

We accomplish interoperation between execution policies and executors by associating an executor object with each execution policy object. During algorithm execution, execution agents may be created by the execution policy's associated executor. The following code presents additional members to the Parallelism TS v1's execution policy types which enable interoperation with executors.

```
// rebind the type of executor used by an execution policy
// the execution category of Executor shall not be weaker than that of ExecutionPolicy
template<class ExecutionPolicy, class Executor>
struct rebind_executor;

// add the following members to each execution policy defined in the Parallelism TS
class library-defined-execution-policy-type
{
  public:
    // the category of the execution agents created by this execution policy
    using execution_category = ...

    // the type of the executor associated with this execution policy
    using executor_type = ...

    // constructor with executor
    library-defined-execution-policy-type(const executor_type& ex = executor_type{});

    // returns the executor associated with this execution policy
    executor_type& executor();
    const executor_type& executor() const;

    // returns an execution policy p with the same execution_category as this one,
    // such that p.executor() == ex
    // executor_traits<Executor>::execution_category may not be weaker than
    // this execution policy's execution_category
    template<class Executor>
    typename rebind_executor<ExecutionPolicy,Executor>::type
      on(const Executor& ex) const;
};
```

4.5 Additional `this_thread`-specific execution policies

For convenient access to what we anticipate will be a common use case, we propose augmenting the existing suite of execution policies with two additional policies which permit algorithm execution within the invoking thread only:

```
namespace this_thread
{

class parallel_execution_policy;
constexpr parallel_execution_policy par{};

class vector_execution_policy;
constexpr vector_execution_policy vec{};
```

```
}
```

Our executor model allows us to distinguish between parallel and vector execution policies which restrict execution to the current thread and those which do not. For this reason, we suggest restoring the original name of `parallel_vector_execution_policy` and `par_vec` to `vector_execution_policy` and `vec`, respectively. With this restoration, the vector policies' nomenclature would be consistent with `par` and `this_thread::par`.

4.6 `std::async` executor overload

Finally, to address use cases where the blocking destructor behavior of futures returned by `std::async` is undesirable, we propose an overload for `async` which takes an executor as an argument. The blocking behavior of the `future` type returned by the overload is simply a property of its type. The following is a possible implementation:

```
template<class Executor, class Function, class... Args>
executor_traits<Executor>::template future<result_of_t<Function(Args...)>
  async(Executor& ex, Function&& f, Args&&... args)
{
  return executor_traits<Executor>::async_execute(
    ex,
    bind(forward<Function>(f), forward<Args>(args)...)
  );
}
```

5 Relationship with existing executor proposals

This proposal is primarily motivated by the desire to support algorithms in the Parallelism TS and to enable programmer control of work placement during parallel algorithm execution. Accordingly, the functionality we have described focuses only on the most basic features which enable interoperation between execution policies and executors. At first glance, this proposal may seem very different from the preexisting executor proposals N4143 (Mysen) and N4242 (Kohlkoff), which more closely resemble abstractions of work queues. However, we believe our proposal is largely complementary to these existing proposals.

The functionality described in N4143 and N4242, or indeed other alternatives, may be layered on top of our design. Either or both could form the basis for a more complete executor facility built upon the minimal foundations we have outlined. Indeed, we believe that our abstract `Executor` concept as well as our customizable `executor_traits` offers a plausible way for different types of executors to coexist within the same programming model.

5.1 Comparisons with paper N4143 (Mysen)

Our `async_execute()` function corresponds to N4143's `spawn()`, but returns a future by default. In the appendix, we discuss a scheme to generalize the result type.

An analogue to our synchronous `execute()` function is not present in N4143. We include it because synchronization is critical. Achieving synchronization via the introduction of a promise/future pair via side effects may be expensive, difficult, or impossible for some types of executors.

For example, the `vector_executor` example of 3.4.1 implemented with a SIMD for loop naturally synchronizes with the caller as a consequence of its loop-based implementation. Introducing asynchrony requires the additional step of calling `std::async`. If `execute()` was not a basic executor operation, the only way to achieve synchronization with the `vector_executor` would be to call `wait()` on the future returned from

`async_execute()`. The cost of calling a superfluous `std::async` would likely dominate the performance of many applications of `vector_executor`.

Like our proposed `executor_traits`, N4143 also provides a uniform means for generic code to manipulate executors through a type erasing executor wrapper called `executor`. We view `executor_traits` and `executor` as complementary and serving different use cases: when the underlying executor type is known statically, `executor_traits` is useful. In other cases, when it is inconvenient to statically track an executor's type (for example, when passing it through a binary API), `executor` may be used.

5.2 Comparisons with paper N4242 (Kohlkoff)

Our `async_execute()` function corresponds most closely with N4242's `post()`. Unlike `post()`, our `async_execute()` allows the executor to eagerly execute the given function, if that is the behavior of the executor. For example, our `sequential_executor` class must execute the function in the calling thread.

Another difference is that N4242 proposes three different basic executor operations. By contrast, our proposal suggests two: one synchronous, and one asynchronous. Our proposal could accomodate the additional semantics of N4242 by introducing different types of executors with the corresponding semantics.

The nuances of task dispatch semantics are critical to the use cases encountered in libraries such as Asio. On the other hand, we need not require all executors to support them. One way to harmonize these proposals would be to introduce a refinement of the `Executor` concept we propose catered to the use cases motivating N4242. For the sake of argument, suppose this refinement was called `Scheduler`. In this model, all `Scheduler`s would be `Executor`s, but not all `Executor`s would be `Scheduler`s.

6 Acknowledgments

Thanks to Jaydeep Marathe and Vinod Grover for feedback on the paper and Chris Mysen for fielding questions about N4143.

7 Appendix: Future Work

We have described what we believe is the minimal set of executor functionality necessary to interoperate with and implement the algorithms of the Parallelism TS. A full-featured executor design would likely include additional functionality, both for convenience and to broaden its scope. In this appendix, we identify features that lie outside the minimal requirements for an executor facility, but which may be valuable in a complete design and thus may warrant future work.

7.1 Concurrency

Efficient implementations of parallel algorithms must be able to create and reason about physically concurrent execution agents. Some implementations will require groups of concurrent agents to synchronize and communicate amongst themselves. To serve this use case, this category of executor must be able to guarantee successful creation (or indicate failure) of an entire group of concurrent agents. To appreciate why this is important, realize that the first code example of section 3.2 deadlocks when an individual `spawn()` call fails. To represent this category of execution, we anticipate the addition of a concurrent execution category as well as a `concurrent_executor` class.

7.2 Shared parameters to `executor_traits::execute()`

Groups of agents will often require communication. It makes sense to mediate the communication via special shared parameters set aside for this purpose. For example, parallel agents might communicate via a shared `atomic<T>` parameter, while concurrent agents might use a shared `barrier` object to synchronize their communication. Even sequential agents may need to receive messages from their ancestors. To enable these use cases, we anticipate overloads of `execute()` and `async_execute()` with an additional parameter to be shared across the group of agents.

7.3 Hierarchical execution categories

Our initial design categorizes execution semantics with a set of "flat" categories without internal structure. In general, the categories of execution we encounter in practice are hierarchical. On CPUs, an outer parallel loop is often composed with an inner vector loop. Similarly, GPU kernels may naturally be described via an outer-parallel, inner-concurrent construction. Parallelism across nodes in a cluster of machines adds yet another execution layer to the hierarchy. Capturing these nested execution categories within an executor design would allow for very efficient mappings of computation onto execution resources.

7.4 `executor_traits::then_execute()`

A natural extension of our design would allow executors to create asynchronous continuations dependent on a predecessor task in the style of `future::then()`. It is unclear if a hypothetical `executor_traits::then_execute()` function, whose execution would depend on the completion of a preceding task, would be a more basic primitive than `executor_traits::async_execute()`. At first glance, it seems difficult to implement `then_execute()` generically in terms of `async_execute()` because the executor needs to be directly involved in the details of scheduling a dependent. However, `async_execute()` could be implemented in terms of `then_execute()` if `executor_traits` could create an immediately ready future. The drawback of requiring `then_execute()` as a basic operation is the burden placed on executor authors to support dependent scheduling.

7.5 `Future` concept

If different executors are allowed to define executor-specific types of futures they return, it is necessary to describe the requirements on those types with some sort of concept. A `Future` concept would also allow different types of futures to interoperate and compose, for example, with functions such as `when_all`.

7.6 Fire-and-forget

Our proposed design does not attempt to incorporate a "fire-and-forget" mode of asynchronous execution which would not need to synchronize with the completion of tasks submitted to the executor. One way to support fire-and-forget as well as fire-and-remember executors within the same programming model would be to generalize the type `future<T>` of the asynchronous return result. Such a model would employ the same mechanism by which allocators customize the `pointer` type returned by the `allocate()` function. For fire-and-forget executors, this type could be `void`. The same scheme could also accomodate executors providing completion tokens instead of full-fledged futures.

7.7 Return values from bulk-style `execute()` and `async_execute()`

When implementing parallel algorithms such as `reduce` whose implementations are most naturally defined in terms of a series of stages, it is necessary to communicate collections of partial results from one stage to the

next. When these implementations must be asynchronous, it becomes awkward or impossible to manage the lifetime of these results. One remedy is to manage these collections of results through the future returned by `async_execute`. In this model, the bulk forms of `execute()` and `async_execute()` would return a container, and a future container, respectively.

7.8 Multidimensional indices

The choice of name for `shape_type` versus `size_type` reflects our intention for executors to generate indices from multidimensional index spaces. Future work will describe the requirements on multidimensional indices to interoperate with executors.

8 References

1. N3724 - A Parallel Algorithms Library, J. Hoberock et al. 2013.
2. N4143 - Executors and schedulers, revision 4. C. Mysen, et al. 2014.
3. N4242 - Executors and Asynchronous Operations, Revision 1. C. Kohlkoff, 2014.