Document Number: P1019R2
Date: 2019-01-21
Audience: SG1, LEWG
Project: Programming Language C++
Reply-to: Jared Hoberock
NVIDIA Corporation
jhoberock@nvidia.com

1 Integrating Executors with Parallel Algorithms

Execution policies allow programmers to impose requirements on how parallel algorithms are allowed to execute. For example, `std::execution::par` allows algorithms to execute in parallel, possibly on multiple implementation-created threads:

```
using namespace std::execution;

// execute a parallel reduction using implementation-defined resources
auto sum = std::reduce(par, data.begin(), data.end());
```

However, the parallel algorithms currently provide no mechanism for controlling *where* execution is allowed to occur. For example, there is currently no mechanism for a programmer to communicate that a parallel algorithm should execute by submitting work to an externally-provided thread pool.

P0443 proposes *executors* as a general mechanism for programmatic control over execution-creating library interfaces, and others (N4406, P0761) have suggested a `policy.on(executor)` syntax for composing with parallel algorithms. With the introduction of executors and `.on()`, our previous example becomes flexible:

```
using namespace std::execution;

// execute a parallel reduction using implementation-defined resources
auto sum = std::reduce(par, data.begin(), data.end());

// NEW: execute a parallel reduction on a user-created thread pool with 4 threads
static_thread_pool pool(4);
sum = std::reduce(par.on(pool.executor()), data.begin(), data.end());

// NEW: execute a parallel reduction on a user-defined executor
my_executor_type my_executor;
sum = std::reduce(par.on(my_executor), data.begin(), data.end());
```

This paper is a sketch of the likely additions necessary to incorporate P0443's executor model into parallel algorithms.

1.1 Proposal summary

1. For each standard, named policy, introduce a `.on(ex)` member function to create a new execution policy with an adaptation of `ex` bound as its associated executor.
2. For policy types resulting from `.on(ex)` expressions, introduce a `.executor()` member function to return the policy's associated executor.
3. For each standard policy, introduce a `::execution_requirement` member variable to advertise an execution policy's bulk execution requirement.
4. Refactor existing wording to describe execution policy guarantees in terms of the guarantees provided by their `::execution_requirement`.
5. Introduce a new execution mapping property value, `execution::mapping.this_thread`, to capture `seq`'s requirement to execute on the calling thread.

These additions should merge with the Working Paper at the same time as executors. Depending on timing, an earlier shipping vehicle may also be possible; for example, a new Parallelism Technical Specification.

1.2 Why are both execution policies and executors required?

Some committee members have asked why both execution policies and executors are necessary for parallel algorithm execution. The implication being that an executor alone should be sufficient for describing how a parallel algorithm should execute. In fact, executors express just one dimension of algorithm execution. Both execution policies and executors are necessary.

Policies organize algorithm parameters. Many parameters may influence the performance of a parallel algorithm's implementation. The physical resource upon which the algorithm execute, represented by an executor, is just one of these. Another example would be a memory allocator to use for temporary memory allocation during algorithm execution. Some of these parameters may even be algorithm-specific tuning parameters which influence the performance of high performance implementations. Because these parameters could be arbitrarily detailed or specific, and there may be many of them, it makes sense to encapsulate them inside a single user-facing execution policy. Doing so keeps algorithm function signatures organized and allows users to focus on only those parameters relevant to their use cases.

Some policies may have no executor. When execution policies were originally proposed, one potential use case was to act as channels for exposing vendor-specific, high-performance algorithm implementations. One example given was a high-performance sorting algorithm implemented by an FPGA. This would be an example of a fixed-function execution policy compatible with only a single algorithm. Such an execution policy could have no executor because the underlying resource, a fixed-function FPGA, would not be able to host general purpose execution agents. In order to accomodate such platforms, we need to allow for execution policies without an associated executor.

Moreover, we suggest that standard policies such as `execution::par` should not have an associated executor. This is to allow standard library implementors maximum freedom in their parallel algorithm implementations. For example, a high-performance implementation might require an implementation that eschewed an executor. Of course, an executor may always be introduced at an algorithm call site with an expression involving a standard policy (e.g., `execution::par.on(ex)`), but the policy resulting from that expression would be different from the original policy and would have a different type.

1.3 Does `.on(executor)` provide a hint or a requirement?

An associated executor acts as a requirement to invoke element access functions on execution agents created by that executor. We believe that acting as a requirement best reflects programmer intent. By writing `.on(executor)`, the programmer has made explicit and visible statement of intent about a particular execution resource on which they would like execution to happen. It seems programmer-hostile for a parallel algorithm to then ignore that statement and schedule execution elsewhere. Consider the consequences of ignoring that intention on program validity, which may depend on the execution context on which an element access function is invoked. For example, an element access function executing on a fiber may access a more constrained set of synchronization primitives than it could while executing on a full-fledged thread. This means that writing correct programs demands that parallel algorithms interpret associated executors as strong requirements.

1.4 Why not receive executors as additional, optional parameters to parallel algorithms?

Some committee members have asked why we propose to package an associated executor within an execution policy rather than introduce an additional executor parameter into the function signatures of parallel algorithms. Their reasoning is that execution policies such as `execution::par` currently act as permissions on algorithm execution. By contrast, our proposed associated executor acts as a new requirement on execution. Our reasoning for encapsulating associated executors within a single execution policy parameter is because we expect the set of additional parameters affecting algorithm execution to be open-ended.

For example, execution policies in the Thrust algorithm library encapsulate a user-specified allocator used for temporary memory allocations during parallel algorithm execution. Rather than introduce an additional allocator parameter into every parallel algorithm signature, Thrust incorporates the temporary allocation policy into its execution policy objects. Still other optional parameters drive the behavior of individual algorithms, such as tuning policies governing the performance of specific algorithm implementations such as radix sort.

An alternative design which would proliferate these kinds of optional parameters would make parallel algorithms difficult to use, document, and maintain. However, we believe that isolating these parameters within general purpose execution policy parameters set aside for this purpose will scale. It is true that this proposed direction would expand the scope of execution policies from mere permissions to a collection of permissions, requirements, preferences, and hints. However, we believe the meaning of the word "policy" is abstract enough to encompass all of these different possibilities, and will not lead to confusion.

1.5 Implementation Experience

The `.on()` syntax is inspired by a long-standing feature within Thrust which provides the means to specify that parallel algorithm execution should occur "on" a particular CUDA stream. We have elaborated and generalized that syntax within Agency to support the P0443 executor model. After years of deployment experience, we are convinced that this syntax provides an clear and concise expression of the programmer's intent to create a binding between an algorithm's execution ordering requirements (the policy) with a particular resource (the executor).

1.6 Proposed additions to execution policies

The most significant changes will be to "modularize" portions of **[algorithms.parallel.exec]** and **[algorithms.parallel.exceptions]**. They are currently "hard-coded" to describe only `sequenced_policy`, `parallel_policy`, and `parallel_unsequenced_policy`. They should instead make general statements in terms of the associated executors of execution policies.

1.6.1 Definitions

1. The presence of a member function `.executor()` indicates that an execution policy has an *associated executor*, and the value returned by that function is a copy of the associated executor.

[*Editorial note:* We should find a way to word this such that the associated executor exists iff `.executor()` exists. –*end note*]

2. During parallel algorithm execution, execution policies invoke element access functions on execution agents created by their associated executor, if an associated executor exists. Otherwise, element access functions are invoked on execution policy-defined threads.

3. Execution policies have a *bulk execution requirement*, described by the member variable `execution_requirement`, if present. If absent, the execution policy's bulk execution requirement is `execution::bulk_guarantee.unsequenced`.

4. The bulk execution guarantee of an execution policy's associated executor satisfies the policy's bulk execution requirement.

[*Editorial note:* The intention of these paragraphs is to allow refactoring **[algorithms.parallel.exec]** p4, p5, and p6 to state them in terms of the execution policy's `execution_requirement`. The idea would be to eliminate p4, p5, & p6 in favor of something like these paragraphs. –*end note*]

1.6.2 New execution policy members

We propose one new static member variable and two new member functions for each standard execution policy type. `::execution_requirement` allows observation of an execution policy's requirements on parallel algorithm execution. `.executor()` allows observation of an execution policy type's associated executor. `.on()` allows a client to "rebind" a policy's associated executor. Rather than mutating the existing policy, a new policy is returned as a result. The given executor must be able to guarantee the original policy's execution requirement via an `execution::require` expression. Otherwise, the call to `.on()` will not compile.

1.6.2.1 Advertising the bulk execution requirement

For each standard execution policy type, introduce a `static constexpr` member `::execution_requirement`:

Execution policy type	Type of `::execution_requirement`
`sequenced_policy`	`execution::bulk_guarantee_t::sequenced_t`
`parallel_policy`	`execution::bulk_guarantee_t::parallel_t`
`parallel_unsequenced_policy`	`execution::bulk_guarantee_t::unsequenced_t`

1.6.2.2 Observing the associated executor

Introduce a member function `sequenced_policy::executor()`:

```
see-below executor() const;
```

Returns: A copy `ex` of this execution policy's associated executor. `can_require_v<decltype(ex), execution::bulk_t, decltype(execution_requirement), execution::mapping_t::this_thread_t>` is true.

For each of the other standard execution policy types, introduce a member function `.executor()`:

```
see-below executor() const;
```

Returns: A copy `ex` of this execution policy's associated executor. `can_require_v<decltype(ex), execution::bulk_t, decltype(execution_requirement), execution::mapping_t::thread_t>` is true.

1.6.2.3 Rebinding the associated executor

For each standard, named execution policy type, introduce a member function `.on()`:

```
template<class OtherExecutor>
see-below on(const OtherExecutor& ex) const;
```

Returns: An execution policy object `policy` identical to `*this` with the exception that the returned execution policy's associated executor is a copy of `ex`. [*Editorial note:* This wording is probably not quite correct. Its use of "identical" is inspired by P0443's specification of `require` expressions. *–end note*] `is_execution_policy<decltype(policy)>::value` is true.

Remarks: This function shall not participate in overload resolution unless `execution::can_require_v<OtherExecutor, execution::bulk, decltype(execution_requirement)>` is true.

1.6.2.4 Introducing `execution::mapping.this_thread`

Supplement `execution::mapping_t` with an additional nested property type:

Nested Property Type	Nested Property Object Name	Requirements
`mapping_t::this_thread_t`	`mapping::this_thread`	Each execution agent created by the executor is mapped onto the thread on which the originating execution function was invoked.

1.6.3 Notes

1. This proposal associates an executor with execution policies returned by standard policies' `.on()` functions. However, we do not propose a general requirement for *all* execution policies to have an associated executor because future user-defined execution policy types may not have an associated executor. For example, an execution policy targeting an ASIC specialized for accelerated sorting need not have an associated executor capable of executing arbitrary user-defined functions.
2. The return type of `.executor()` and `.on()` is not concretely specified in order to allow us to decide what those types should be later, if the standards committee decides to do so.
3. `.on()` does not require e.g. `can_require_v<Execution, twoway_t>` to give parallel algorithm authors implementation freedom to choose which bulk execution functions they use.
4. Rebinding an executor may have applications beyond execution policies, and it is possible that a general purpose mechanism for executor rebinding will emerge. We envision `.on()` to be syntactic sugar for such a general purpose mechanism, should one eventually exist.
5. The execution policy returned by e.g. `execution::par.on(my_exec)` is not guaranteed to have a `.on()` member because we intend for `.on()` to be a syntactic convenience provided by the standard named policies. It is unclear that `.on()` should be a requirement for all `ExecutionPolicy` types; for example, user-defined `ExecutionPolicy` types.
6. For convenience, `.on()` could be extended in the future to receive types which are not executors, but which do have an associated executor; for example, thread pools.
7. Whether or not `.on()` eagerly adapts the given executor via a `execution::require` call is an implementation detail.
8. With the appropriate executor exception handling properties, [**algorithms.parallel.exceptions**] could be reformulated in terms of general statements about the properties of the execution policy's associated executor. See P0797 for possible approaches to executor exception handling properties.

1.7 Possible Implementation

```
namespace execution {

template<class ExecutionRequirement, class Executor>
class __basic_execution_policy
{
  public:
    static_assert(execution::can_require_v<Executor, execution::bulk_t, ExecutionRequirement>::value);

    static constexpr ExecutionRequirement execution_requirement{};

    Executor executor() const
    {
      return executor_;
    }

  private:
    template<class ER, class E>
    friend __basic_execution_policy<ER, E> __make_basic_execution_policy(const E& ex)
    {
      return __basic_execution_policy<ER, E>{ex};
    }

    Executor executor_;
};

class parallel_policy
{
  private:
    __parallel_executor executor_;

    parallel_policy(const __parallel_executor& ex) : executor_(ex) {}

  public:
    static constexpr execution_requirement = execution::bulk_guarantee.parallel;

    parallel_policy() = default;

    template<class OtherExecutor,
             class = enable_if_t<
               execution::can_require_v<
                 execution::bulk_t, decltype(execution_requirement)
            >>>
    __basic_execution_policy<decltype(execution_requirement), OtherExecutor>
      on(const OtherExecutor& ex) const
    {
      return __make_basic_execution_policy(ex);
    }
};

} // end execution

template<class ExecutionRequirement, class Executor>
struct is_execution_policy<execution::__basic_execution_policy<ExecutionRequirement,Executor>> : true_type {};

template<>
struct is_execution_policy<execution::parallel_policy> : true_type {};

// parallel_policy specialization
```

```
template<class P, class I, class F>
void for_each(const execution::parallel_policy& policy, I first, I last, F f)
{
  // do something parallel_policy-specific
  ...
}

// general purpose implementation for policies with an associated executor
template<class P, class I, class F,
         class = enable_if_t<
           is_execution_policy_v<decay_t<P>>
        >>
void for_each(P&& policy, I first, I last, F f)
{
  auto ex_ = execution::require(policy.executor(),
    execution::bulk,
    policy.execution_requirement,
    execution::bulk_oneway,
    execution::blocking.always
  );

  try
  {
    // try to execute via the executor
    ex.bulk_execute(
      [=](auto idx, auto&, auto&)
      {
        f(first[idx]);
      },
      distance(first, last),
      []{ return ignore; },
      []{ return ignore; }
    );
  }
  catch(bad_alloc)
  {
    try
    {
      // try again on this thread
      for_each(first, last, f);
    }
    catch(...)
    {
      terminate();
    }
  }
}
```

1.8 Proposed Wording

1.8.1 Changes to [execpol.general]

Introduce these paragraphs:

> The presence of a member function `.executor()` indicates that an execution policy has an *associated executor*, and the value returned by that function is a copy of the associated executor.
>
> Execution policies have a *bulk execution requirement*, described by the member variable `execution_requirement`, if present. If absent, the execution policy's bulk execution requirement is `execution::bulk_guarantee.unsequenced`.

The bulk execution guarantee of an execution policy's associated executor satisfies the policy's bulk execution requirement.

1.8.2 Changes to [execpol.seq]

Replace `execution::sequenced_policy` class definition:

```
class execution::sequenced_policy { unspecified };
```

with

```
class execution::sequenced_policy
{
public:
  constexpr static auto execution_requirement = execution::bulk_guarantee.sequenced;

  template<class OtherExecutor>
    see below on(const OtherExecutor& ex) const;
};
```

Introduce a member function `sequenced_policy::on()`:

```
template<class OtherExecutor>
  see below on(const OtherExecutor& ex) const;
```

Returns: An execution policy object `policy` identical to `*this` with the exception that the returned execution policy's associated executor is a copy of `ex`. `execution::is_execution_policy<decltype(policy)>::value` is true.

Remarks: This function shall not participate in overload resolution unless `execution::can_require_v<OtherExecutor, execution::bulk, decltype(execution_requirement)>` is true.

1.8.3 Changes to [execpol.par]

Replace `execution::parallel_policy` class definition:

```
class execution::parallel_policy { unspecified };
```

with

```
class execution::parallel_policy
{
public:
  constexpr static auto execution_requirement = execution::bulk_guarantee.parallel;

  template<class OtherExecutor>
    see below on(const OtherExecutor& ex) const;
};
```

Introduce a member function `sequenced_policy::on()`:

```
template<class OtherExecutor>
  see below on(const OtherExecutor& ex) const;
```

Returns: An execution policy object `policy` identical to `*this` with the exception that the returned execution policy's associated executor is a copy of `ex`. `execution::is_execution_policy<decltype(policy)>::value` is true.

Remarks: This function shall not participate in overload resolution unless `execution::can_require_v<OtherExecutor, execution::bulk, decltype(execution_requirement)>` is true.

1.8.4 Changes to [execpol.parunseq]

Replace `execution::parallel_unsequenced_policy` class definition:

```
class execution::parallel_unsequenced_policy { unspecified };
```

with

```
class execution::parallel_unsequenced_policy
{
public:
  constexpr static auto execution_requirement = execution::bulk_guarantee.unsequenced;

  template<class OtherExecutor>
    see below on(const OtherExecutor& ex) const;
};
```

Introduce a member function `sequenced_policy::on()`:

```
template<class OtherExecutor>
  see below on(const OtherExecutor& ex) const;
```

Returns: An execution policy object `policy` identical to `*this` with the exception that the returned execution policy's associated executor is a copy of `ex`. `execution::is_execution_policy<decltype(policy)>::value` is true.

1.8.5 Changes to [algorithms.parallel.exec]

Replace **[algorithms.parallel.exec]** p4, p5, and p6 with the following paragraph:

4. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::sequenced_policy` all occur in the calling thread of execution. [*Note:* The invocations are not interleaved; see 6.8.1. –*end note*]

5. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_policy` are permitted to execute in either the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by thread (33.3.2) provide concurrent forward progress guarantees (6.8.2.2), then a thread of execution implicitly created by the library will provide parallel forward progress guarantees; otherwise, the provided forward progress guarantee is implementation-defined. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other. [*Note:* It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks. –*end note*]

   [*Example:*

   ```
   int a[] = {0,1};
   std::vector<int> v;
   std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
     v.push_back(i*2+1); // incorrect: data race
   });
   ```

   The program above has a data race because of the unsynchronized access to the container `v`.–*end example*]

   [*Example:*

   ```
   std::atomic<int> x{0};
   int a[] = {1,2};
   std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
     x.fetch_add(1, std::memory_order::relaxed);
     // spin wait for another iteration to change the value of x
     while (x.load(std::memory_order::relaxed) == 1) { } // incorrect: assumes execution order
   });
   ```

   The above example depends on the order of execution of the iterations, and will not terminate if both iterations are executed sequentially on the same thread of execution. –*end example*]

   [*Example:*

   ```
   int x = 0;
   std::mutex m;
   int a[] = {1,2};
   std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
     std::lock_guard<mutex> guard(m);
     ++x;
   });
   ```

The above example synchronizes access to the object `x` ensuring that it is incremented correctly. *–end example*]

6. The invocations of element access functions in parallel algorithms invoked with an execution policy of type `execution::parallel_unsequenced_policy` are permitted to execute in an unordered fashion in unspecified threads of execution, and unsequenced with respect to one another within each thread of execution. These threads of execution are either the invoking thread of execution or threads of execution implicitly created by the library; the latter will provide weakly parallel forward progress guarantees. [*Note:* This means that multiple function object invocations may be interleaved on a single thread of execution, which overrides the usual guarantee from 6.8.1 that function executions do not interleave with one another. *–end note*] Since `execution::parallel_unsequenced_policy` allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. Thus, the synchronization with `execution::parallel_unsequenced_policy` is restricted as follows: A standard library function is *vectorization-unsafe* if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, and if it is not a memory allocation or deallocation function. Vectorization-unsafe standard library functions may not be invoked by user code called from `execution::parallel_unsequenced_policy` algorithms. [*Note:* Implementations must ensure that internal synchronization inside standard library functions does not prevent forward progress when those functions are executed by threads of execution with weakly parallel forward progress guarantees. *–end note*] [*Example*:

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&](int) {
  std::lock_guard<mutex> guard(m); // incorrect: lock_guard constructor calls m.lock()
  ++x;
});
```

The above program may result in two consecutive calls to `m.lock()` on the same thread of execution (which may deadlock), because the applications of the function object are not guaranteed to run on different threads of execution. *–end example*] [*Note:* The semantics of the `execution::parallel_policy` or the `execution::parallel_unsequenced_policy` invocation allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation due to lack of resources. *–end note*]

4. During parallel algorithm execution, execution policies invoke element access functions on execution agents created by their associated executor, if an associated executor exists. Otherwise, element access functions are invoked on execution policy-defined threads. [*Note:* It is the caller's responsibility to ensure that the invocation does not introduce data races or deadlocks. *–end note*]

[*Example:*

```
int a[] = {0,1};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  v.push_back(i*2+1); // incorrect: data race
});
```

The program above has a data race because of the unsynchronized access to the container `v`.*–end example*]

[*Example:*

```
std::atomic<int> x{0};
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
  x.fetch_add(1, std::memory_order::relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order::relaxed) == 1) { } // incorrect: assumes execution order
});
```

The above example depends on the order of execution of the iterations, and will not terminate if both iterations are executed sequentially on the same thread of execution. *–end example*]

[*Example:*

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&](int) {
```

```
  std::lock_guard<mutex> guard(m);
  ++x;
});
```

The above example synchronizes access to the object `x` ensuring that it is incremented correctly. *–end example*]

[*Example:*

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&](int) {
  std::lock_guard<mutex> guard(m); // incorrect: lock_guard constructor calls m.lock()
  ++x;
});
```

The above program may result in two consecutive calls to `m.lock()` on the same thread of execution (which may deadlock), because the applications of the function object are not guaranteed to run on different threads of execution. *–end example*] [*Note:* The semantics of the `execution::parallel_policy` or the `execution::parallel_unsequenced_policy` invocation allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation due to lack of resources. *–end note*]

1.8.6 Additions to Executor Properties

Add a row to P0443's `execution::mapping_t` table:

Nested Property Type	Nested Property Object Name	Requirements
`mapping_t::this_thread_t`	`mapping::this_thread`	Each execution agent created by the executor is mapped onto the thread on which the originating execution function was invoked.

1.9 Acknowledgements

Thanks to Michael Garland, Carter Edwards, Olivier Giroux, and Bryce Lelbach for reviewing this paper and suggesting improvements, as well as SG1 and LEWG for their in-person feedback.

1.10 Changelog

1.10.1 R2

- In response to SG1 feedback received at the San Diego meeting, updated specification to require that element access functions be invoked on execution agents created by the associated executor, if one exists.
- Introduced explanation for why associated executors impose requirements rather than act as a hint.
- Introduced explanation for why we are proposing a `.on()` syntax for execution policies rather than augmenting parallel algorithms with parameters in addition to the policy parameter.

1.10.2 R1

- Introduced explanation for why both execution policies and executors are necessary.
- Removed `.executor()` from standard policy types based on feedback from Rapperswil and Bellevue.

1.10.3 R0

- Initial paper.