An Interface for Abstracting Execution | P0058R1

Jared Hoberock Michael Garland Olivier Giroux
{jhoberock,mgarland,ogiroux}@nvidia.com
Hartmut Kaiser hartmut.kaiser@gmail.com

2016-02-12

1 Introduction

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. A suitable abstraction encapsulating the details of how work is created across such diverse platforms would be of significant value to parallel algorithm implementations. Furthermore, other constructs that expose parallelism to the programmer—including `async` and `task_block`—would benefit from a common abstraction for launching work. We believe that a suitably defined executor interface provides just such a facility.

Execution policies, which are the organizing principle of the current Parallelism TS, provide a means of specifying *what* parallel work can be created, but do not currently 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[1], we describe an executor interface that can efficiently abstract the details of execution across a range of platforms. It provides the features required to implement parallel algorithms and control structures such as `task_block`. Furthermore, this executor interface is designed to compose with the execution policies introduced in Version 1 of the Parallelism TS. 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

[1] This paper updates P0058R0. New sections are highlighted with green titles.

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.

Enable convenient launch mechanisms. 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 executor-based overloads of control structures such as `task_block` and `std::async`, examples 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 Implementing Control Structures with Executors

The remainder of this paper outlines a featureful, concrete executor facility that both satisfies the minimum requirements for supporting the algorithms of the Parallelism TS, and also provides a foundation for creating powerful control structures based on abstracted, modular execution. Before providing the detailed design, we examine some motivating examples of control structures that use our executor interface.

3.1 Implementing `async`

One of the the simplest possible applications of our `executor_traits` interface could be found in a hypothetical implementation of `std::async`. The following implementation sketch demonstrates that `std::async` may be implemented with a call to a single executor operation:

```
template<class Function, class... Args>
future<result_of_t<Function(Args...)>>
  async(Function&& f, Args&&... args)
{
  using executor_type = ...
  executor_type ex;

  return executor_traits<executor_type>::async_execute(
    ex,
    bind(forward<Function>(f), forward<Args>(args)...)
  );
}
```

This implementation assumes the existence of a special `executor_type` which delivers `std::async`'s idiosyncratic future blocking behavior. However, the implementation could also be generalized to allow an overload of `async` which composes with a user-supplied executor taken as a parameter:

```
template<class Executor, class Function, class... Args>
typename 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)...)
  );
}
```

Such an overload would address use cases where the blocking destructor behavior of futures returned by the current `std::async` is undesirable. The blocking behavior of the `future` type returned by this new, hypothetical overload would simply be a property of the executor's associated `future` type.

3.2 Implementing `for_each_n`

The initial motivation for the interface we describe in this paper 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.

3.3 Implementing `task_block`

Though the initial motivation for our `executor_traits` design was to support the parallel algorithms of the Parallelism TS, we believe that it is sufficiently flexible enough to implement other execution control structures. Below is a sketch of a possible implementation of the salient member functions of N4411's `task_block`.

```
class task_block
{
  public:
    template<class F>
    void run(F&& f)
    {
      // asynchronously execute f on a new agent
      auto fut = executor_traits<executor_type>::async_execute(exec, std::forward<F>(f));

      // record the outstanding future in our list
      futures.emplace_back(move(fut));
    }

    void wait()
    {
      // wait for all outstanding futures
      for(auto& f : futures)
      {
        future_traits<future_type>::wait(f);
      }

      futures.clear();
    }
```

```
  private:
    using executor_type = ...
    using future_type = typename executor_traits<executor_type>::template future<void>;

    executor_type exec;
    vector<future_type> futures;
};
```

In this example, the implementation of `task_block` provides its own custom type of executor to provide the concurrency guarantees detailed in N4411. Additionally, the type of future it collects is also potentially custom and depends on the executor. Instead of managing execution agent lifetimes directly, `task_block`'s `.run()` and `.wait()` functions simply defer agent creation to its member executor. Since agent creation is delegated to the executor, it presents an interesting opportunity for relaxing `task_block`'s semantics in a future revision. If the default execution agent semantics of N4411 are undesirable for particular use cases, a future revision of `task_block` could potentially parameterize these semantics by accepting a user executor.

3.4 Extending P0155R0 to support execution policies and executors

Enabling passing an optional execution policy to `define_task_block` gives the user control over the amount of parallelism employed by the created `task_block`. In the following example the use of an explicit `par` execution policy makes the user's intention explicit:

```
template <typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    define_task_block(
        par,                // parallel_execution_policy
        [&](task_block<>& tb) {
            if (n->left)
                tb.run([&] { left = traverse(n->left, compute); });
            if (n->right)
                tb.run([&] { right = traverse(n->right, compute); });
        });

    return compute(n) + left + right;
}
```

Please note, that this addition to PR0115R0 would require to turn the actual `task_block` type as passed to the lambda into a template. Here is the corresponding changed interface such an implementation would expose:

```
namespace std {
  namespace experimental {
  namespace parallel {
  inline namespace v2 {

    class task_canceled_exception;

    template <typename ExPolicy = v1::parallel_execution_policy>
      class task_block;
```

```
    template <typename F>
      void define_task_block(F&& f);
    template <typename F>
      void define_task_block_restore_thread(F&& f);

    // new: overloads taking an additional execution policy argument
    template <typename ExPolicy, typename F>
      void define_task_block(ExPolicy&& policy, F&& f);
    template <typename ExPolicy, typename F>
      void define_task_block_restore_thread(ExPolicy&& policy, F&& f);
  }
  }
  }

}
```

This change also enables defining at runtime what execution policy to use (by passing an instance of a generic `v1::execution_policy`). This is benefitial in many contexts, for instance debugging (by dynamically setting the execution policy to `seq`).

Often, we want to be able to not only define an execution policy to use by default for all spawned tasks inside the task block, but in addition to customize the execution context for one of the tasks executed by `task_block::run`. Adding an optionally passed executor instance to that function enables this use case:

```
template <typename Func>
int traverse(node *n, Func&& compute)
{
    int left = 0, right = 0;

    define_task_block(
        par,                // parallel_execution_policy
        [&](auto& tb) {
            if (n->left)
            {
                // use explicitly specified executor to run this task
                tb.run(my_executor(), [&] { left = traverse(n->left, compute); });
            }
            if (n->right)
            {
                // use the executor associated with the par execution policy
                tb.run([&] { right = traverse(n->right, compute); });
            }
        });

    return compute(n) + left + right;
}
```

A corresponding template `task_block` would look like this:

```
template <typename ExPolicy = v1::parallel_execution_policy>
class task_block {
private:
    // Private members and friends (for exposition only)
```

```
    template <typename F>
      friend void define_task_block(F&& f);
    template <typename F>
      friend void define_task_block_restore_thread(F&& f);

    // new: overloads taking an additional execution policy argument
    template <typename ExPolicy, typename F>
      friend void define_task_block(ExPolicy&& policy, F&& f);
    template <typename ExPolicy, typename F>
      friend void define_task_block_restore_thread(ExPolicy&& policy, F&& f);

    task_block(_unspecified_);
    ~task_block();

public:
    task_block(const task_block&) = delete;
    task_block& operator=(const task_block&) = delete;
    task_block* operator&() const = delete;

    template <typename F>
      void run(F&& f);

    // new: overload taking an additional executor argument
    template <typename Executor, typename F>
      void run(Executor& ex, F&& f);

    void wait();

    // new: expose underlying execution policy.
    ExPolicy& policy();
    const ExPolicy& policy() const;
};
```

3.5 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 futures via `future_traits`

Futures represent a handle to the completion of an asynchronous task. Their interface allows clients to wait for, and get, the result of asynchronous computations. In addition to acting as one-way channels through which asynchronously executing agents communicate their results to receiver clients, futures alternatively allow redirection of results to a successor *continutation* task. By connecting continuations to futures in this way, clients may use futures as the basic building blocks for assembling sophisticated, asynchronous control flows. Our proposed `executor_traits` interface uses futures to return the results of asynchronously executing agents.

The C++ Standard Library provides two future templates: `std::future<T>`, and `std::shared_future<T>`. The former provides a single client with a handle to a asynchronous result, while the latter accomodates multiple clients at once. As parallel programming proliferates through the C++ ecosystem, we envision that other types of futures will appear beyond the two types specified by the C++ Standard Library. These will include user-defined future types – especially those associated with novel types of executors. We wish to accomodate this diversity by describing a uniform, standardizable protocol for library implementors to interact with and compose different future types in a uniform way.

One solution for uniform interaction with different future types is through our proposed `future_traits` interface, which follows:

```
template<class Future>
class future_traits
{
  public:
    // the type of the future
    using future_type = Future;

    // the type of future value
    using value_type = ...

    // allows rebinding value_type
    template<class U>
    using rebind = ...

    // the type of future returned by share()
    using shared_future_type = ...

    // waits for the future to become ready
    static void wait(future_type& fut);

    // waits for the future to become ready and returns its value
    static value_type get(future_type& fut);

    // makes a void future which is immediately ready
    static rebind<void> make_ready();

    // makes a future<T> from constructor arguments which is immediately ready
    template<class T, class... Args>
    static rebind<T> make_ready(Args&&... args);

    // attach continuation f to receive the result of fut
    // result of the continuation is communicated through the returned future
    template<class Function>
    rebind<...> then(future_type& fut, Function&& f);

    // casts future<T> to future<U>
    // note: some kinds of casts can be performed without creating a continuation
    template<class U>
    static rebind<U> cast(future_type& fut);

    // shares the given future
    static shared_future_type share(future_type& fut);
};
```

This presentation of `future_traits` matches the functionality present in our prototype implementation. We have included the operations and associated type aliases which seem most fundamental to a future concept. One operation which we include for which there is not yet a standard is the `cast()` operation. While implementing our prototype of `executor_traits`, described in the next section, we often found it necessary to cast one type of future (e.g., `future<T>`) to a related type (e.g., `future<void>`). In general, it would be necessary to create a continuation to perform a conversion from a future's result to some other type. However, there are frequently-occurring cases where a cast may be performed logically where a continuation is not

required. Such use cases include discarding the value of a `future<T>` by converting it into a `future<void>` or converting an empty type into a different empty type. We wish to allow future implementors to target and accelerate such cases through the `cast()` operation when such optimizations are available.

Additional operations beyond which we describe here are certainly possible. For example, an operation which would create an exceptional future, or an operation which would query whether the given future was ready would be useful. If there exists broad support for our `future_traits` proposal, we expect commitee feedback to guide the design of a complete interface.

4.2 Uniform manipulation of executors via `executor_traits`

Executors are modular components for requisitioning execution agents. For library components to be freely composable with diverse executors, including user-defined executors, all executors must be capable of supporting a common interface. This common interface is defined by the `executor_traits` template. This represents the *minimal* interface all control structures can rely on. The definition of this interface has been chosen so that `executor_traits` can ensure that the entire interface can be supported for all executor types, even when the concrete type implements only a subset of the interface. When the concrete executor type does not implement a particular executor operation, `executor_traits` *synthesizes* this operation's implementation.

There are three executor operations which create execution agents: `execute()`, `async_execute()`, and `then_execute()`.

4.2.1 Execution Agent Rules

Execution agent creation is subject to two rules:

1. Only concrete executor operations create execution agents. Consequently, synthesized executor operations only cause agents to be created by calling concrete executor operations, either directly or indirectly.
2. The method for choosing the thread or threads on which these agents execute is a property of the concrete executor type.

One implication of these rules is that only concrete executor operations are permitted to create threads. Because they control the mapping onto threads, it is the prerogative of concrete executors to decide whether execution can occur on the thread which invoked the operation. This ensures that operations synthesized by `executor_traits` cannot spuriously introduce threads into a program, nor eagerly execute agents on the calling thread, unless the specific concrete executor is explicitly built to do these things.

Execution agent synchronization is subject to two rules:

3. Execution agents created by `execute()` synchronize with the thread which invoked `execute()`.
4. If `async_execute()` (`then_execute()`) has a result, it must satisfy the `Future` concept. Execution agents created by `async_execute()` (`then_execute()`) synchronize with the thread which calls `.wait()` on the future returned by `async_execute()` (`then_execute()`).

Rule 4. implies that if `async_execute()` and `then_execute()` have no result, then the execution agents they create synchronize with nothing.

4.2.2 `executor_traits` Synopsis

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 calls to operations with "execute" in their name
    using execution_category = ...

    // the type of index passed to functions invoked by agents created by
    // the multi-agent creation functions
    using index_type = ...

    // the type of the shape parameter passed to the multi-agent creation functions
    using shape_type = ...

    // the type of future returned by asynchronous functions
    template<class T>
    using future = ...

    // the type of future returned by share_future
    template<class T>
    using shared_future = ...

    // the type of container used to return multiple results
    // from multi-agent operations
    template<class T>
    using container = ...

    // creates an immediately ready future containing a T contructed from
    // the given constructor arguments
    template<class T, class... Args>
    static future<T>
      make_ready_future(executor_type& ex, Args&&... args);

    // converts a (possibly foreign) some_future<U> to this executor's future<T>
    template<class T, class Future>
    static future<T>
      future_cast(executor_type& ex, Future& fut);

    // creates a shared_future<T> from a some_future<T>
    static shared_future<...>
      share_future(executor_type& ex, Future& fut);

    // creates multiple shared_futures<T> from a some_future<T>
    static container<shared_future<...>>
      share_future(executor_type& ex, Future& fut,
                   shape_type shape);
```

```
// creates multiple shared_futures<T> from a some_future<T> and returns them
// through the result of the given Factory
template<class Factory>
static result_of_t<Factory(shape_type)>
  share_future(executor_type& ex, Future& fut,
               Factory factory,
               shape_type shape);

// returns the largest shape the executor can accomodate in a single operation
template<class Function>
static shape_type max_shape(const executor_type& ex, const Function& f);

// returns a future to a tuple-like type containing the values of
// the given futures. The result becomes ready when all the given futures are ready
template<class... Futures>
static future<...>
  when_all(executor_type& ex, Futures&&... futures);

// single-agent when_all_execute_and_select
// invokes the function when all the input futures are ready
// the values of the input futures are passed through to the result future as a tuple.
// the caller may select which values to passthrough
template<size_t... Indices, class Function, class TupleOfFutures>
static future<...>
  when_all_execute_and_select(executor_type& ex, Function&& f,
                              TupleOfFutures&& futures);

// multi-agent when_all_execute_and_select
template<size_t... Indices, class Function, class TupleOfFutures, class... Factories>
static future<...>
  when_all_execute_and_select(executor_type& ex, Function f,
                              shape_type shape,
                              TupleOfFutures&& futures,
                              Factories... factories);

// single-agent then_execute
// asynchronously invokes f(value) when the input future's value is ready
// returns the result f(value) via future
template<class Function, class Future>
static future<...>
  then_execute(executor_type& ex, Function&& f, Future& fut);

// multi-agent then_execute returning user-specified container
// asynchronously invokes f(idx, value, shared_args...) when the input future's value is ready
// returns the results of f(idx, value, shared_args...) via future<some_container>
template<class Function, class Future, class Factory, class... Factories>
static future<result_of_t<Factory(shape_type)>>
  then_execute(executor_type& ex, Function f,
               Factory result_factory,
               shape_type shape,
               Future& fut,
               Factories... factories);

// multi-agent then_execute returning default container
```

```
// asynchronously invokes f(idx, value) when the input future's value is ready
// returns the results of f(idx, value, shared_args...) via future<container<...>>
template<class Function, class Future, class... Factories>
static future<container<...>>
  then_execute(executor_type& ex, Function f,
               shape_type shape,
               Future& fut,
               Factories... factories);

// single-agent async_execute
// asynchronously invokes f()
// returns the result of f()'s via future
template<class Function>
static future<result_of_t<Function()>>
  async_execute(executor_type& ex, Function&& f);

// multi-agent async_execute returning user-specified container
// asynchronously invokes f(idx, shared_args...)
// returns the results of f(idx, shared_args...) via future<some_container>
template<class Container, class Function, class Factory, class... Factories>
static future<result_of_t<Factory(shape_type)>
  async_execute(executor_type& ex, Function f,
                Factory result_factory,
                shape_type shape,
                Factories... factories);

// multi-agent async_execute returning default container
// asynchronously invokes f(idx, shared_args...)
// returns the results of f(idx, shared_args...) via future<container<...>>
template<class Function, class... Factories>
static future<container<...>>
  async_execute(executor_type& ex, Function f,
                shape_type shape,
                Factories... factories);

// single-agent execute
// synchronously invokes f()
// returns the result of f()
template<class Function>
static result_of_t<Function()>
  execute(executor_type& ex, Function&& f);

// multi-agent execute returning user-specified container
// synchronously invokes f(idx, shared_args...)
// returns the results of f(idx, shared_args...) via Container
template<class Function, class Factory, class... Factories>
static result_of_t<Factory(shape_type)>
  execute(executor_type& ex, Function f,
          Factory result_factory,
          shape_type shape,
          Factories... factories);

// multi-agent execute returning default container
// synchronously invokes f(idx, shared_args...)
```

```
    // returns the results of f(idx, shared_args...) via container<...>
    template<class Function, class... Factories>
    static container<...>
      execute(executor_type& ex, Function f,
              shape_type shape,
              Factories... factories);
};

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

4.3 Groups of Execution Agents

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

```
executor_traits<my_executor_type>::execute(my_executor, [](size_t i)
{
  // execute 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 similar functions in the interface exist to create groups of invocations asynchronously. Each of these `executor_traits` operations also provides an overload which omits the extra parameter and creates a single execution agent. This single-agent mode of operation corresponds to a special case of the general multi-agent mode and closely matches the executor model proposed by Mysen (N4143).

4.4 Execution Hierarchies

We often reason about parallel execution as a generalization of sequential `for` loops. Indeed, parallel programming systems often expose themselves to programmers as annotations which may be applied to normal `for` loops, transforming them into parallel loops. Such programming systems expose the simplest kind of collective execution: a single "flat" group of execution agents.

In fact, collections of execution agents, and the underlying platform architectures which physically execute them, often have hierarchical structure. An instance of this kind of execution hierarchy is a simple nesting of `for` loops:

```
for(size_t i = 0; i < m; ++i)
{
  for(size_t j = 0; j < n; ++j)
  {
    f(i,j);
  }
}
```

In our terminology, this nested `for` loop example creates a hierarchical grouping of execution agents: one group of sequential execution agents of size m is created by the outer `for` loop, and m groups of sequential

execution agents of size n is created by the inner loop. Each iteration of the outer `for` loop executes in sequence, while an entire inner loop must execute before the next inner loop in sequential is able to begin.

A parallel programmer could use a programming system such as OpenMP to transform the sequential execution of these nested `for` loops by applying annotations:

```
#pragma omp parallel
for(size_t i = 0; i < m; ++i)
{
  #pragma omp simd
  for(size_t j = 0; j < n; ++j)
  {
    f(i,j);
  }
}
```

In this example, the execution agents created by the iterations of the two loops differ: the outer loop iterations execute in parallel, while the inner loop iterations execution in SIMD fashion. This hierarchical style of programming is popular among parallel programmers because it provides a close mapping to the execution exposed by the topology of the physical hardware. In order to best take advantage of parallel platforms, we should endeavour to provide programmers with a uniform programming model for such hierarchies.

Our executor programming model is one such solution to this problem. In our programming model, executors may advertise their category of execution as *nested*, creating groups of both *outer* and *inner* execution agents. For each execution agent created in the outer group, the executor also creates a group of inner execution agents. The execution semantics of the outer group dicates the ordering semantics of the groups of inner agents relative to one other. Within each group of inner execution agents, ordering semantics are given as normal.

One application of nested executors used in the current implementation of our proposal is to easily create new types of executors via composition. For example, our implementation of `parallel_executor` defines itself in terms of special kinds of *executor adaptors*. First, we create an executor which nests sequential execution agents inside concurrent execution agents. Our implementation of `parallel_executor` is simply the "flattening" of this hierchical executor.

The resulting implementation of `parallel_executor` is a single `using` declaration:

```
using parallel_executor = flattened_executor<
  nested_executor<
    concurrent_executor,
    sequential_executor
  >
>;
```

4.5 Shared Parameters

Groups of execution agents often require communicating among themselves. It makes sense to mediate this 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, the multi-agent `executor_traits` operations receive optional `Factory` arguments. The job of these factories is to construct these parameters which are shared across the group of agents. Because locality is so important to efficient communication, we envision associating with each factory an

optional allocator which executors may use to allocate storage for these shared parameters. We use the nomenclature `Factory` for these parameters to distinguish them from normal `Function`s which do not have an associated allocator. In our implementation, the client must pass either zero factories, or exactly one factory per level of the executor's execution hierarchy.

When an executor client passes a factory to a multi-agent `executor_traits` operation, the factory is invoked once per group of execution agents created before the agents begin execution. The object created by the factory is passed by reference to each execution agent as an additional argument to the task to execute. In this way, each execution agent sees the same shared object, and with proper synchronization, the execution agents may communicate through it.

4.6 Multidimensionality

When programming multidimensional applications, one often must often generate points in a multidimensional index space. A straightforward way to produce multidimensional indices in a sequential application is to use loop variables generated by nested `for` loops:

```
for(int i = 0; i < m; ++i)
{
  for(int j = 0; j < n; ++i)
  {
    auto idx = int2(i,j);
  }
}
```

These `for` loops create indices `idx` which enumerate the points spanned by a $M \times N$ rectangular lattice originating at the origin.

We abstract this application of sequential `for` loops within our executor programming model by generalizing the agent indices produced by executors to multiple dimensions. By the same token, the parameter specifying the number of execution agents to create is also allowed to be multidimensional. Respectively, these are the `index_type` and `shape_type` typedefs associated with each executor.

The difference between the multidimensional `for` loop example described in this section and the hierarchical `for` loop example described in the section on execution hierarchies is that the agents created in the multidimensional example all belong to a single group. By contrast, the agents created by the hierarchical example belong to separate groups: one outer group, and many inner groups, where the execution semantics of the outer and inner groups are allowed to differ. Note that depending on the definition of the executor, hierarchical execution agents can in general be indexed multidimensionally. In other words, hierarchy and multidimensionality are orthogonal features. A given executor can choose to support one, both, or neither.

4.7 Fire-and-Forget

Some creators of work may be uninterested in the work's outcome: they simply with to "fire off" some work and then "forget" about it. In these cases, creators of work do not require synchronizing with the work's completion. For example, a deferred resource deallocation scheme might present clients with an asynchronous interface. Behind the interface, an executor might create asynchronous agents to execute the destructors of resources deallocated through the interface. Requiring a future for each deallocation request would burden the executor with the wasteful construction of superfluous and potentially expensive synchronization primitives.

Our proposal allows executors to advertise their asynchronous operations as "fire-and-forget" by allowing them to return `void` from such operations. Instead of returning an instance of a type that fulfills the `Future` concept, `async_execute()` and `then_execute()` would return `void`. For such executors, the associated `future<T>` template always yields `void` when instantiated for any given `T`. Of course, it is always possible for

the client to synchronize with work created this way by introducing their own synchronization primitives into the program.

4.8 Associated Typedefs and Templates

`executor_traits` exposes a number of associated member types and templates which appear in the signatures of its member functions. If the executor for which `executor_traits` has been instantiated has a member typedef (template) with a corresponding name, these associated typedefs (templtes) are simply an alias for that type (template). Otherwise, they are assigned a default.

4.8.1 `executor_type`

`executor_type` simply names the type of executor for which `executor_traits` has been instantiated.

4.8.2 `execution_category`

`execution_category` advertises the guaranteed semantics of execution agents created by calls to multi-agent operations. For example, an `execution_category` of `sequential_execution_tag` guarantees that execution agents created in a group by a call to a function such as `execute(exec, f, n)` all execute in sequential order. By default, `execution_category` should be `parallel_execution_tag`. We discuss execution categories in detail in a later section.

4.8.3 `index_type` and `shape_type`

`index_type` and `shape_type` describe both the domain of multi-agent execution groups and the indices of individual agents therein, as discussed previously. By default, both of these types are `size_t`.

4.8.4 `future<T>`

The `future<T>` template parameterizes the types of futures which the `executor_type` produces as the result of asynchronous operations. Executors may opt to define their own types of future when it is desirable to do so for the reasons discussed in previous sections. By default, this template is `std::future<T>`.

4.8.5 `shared_future<T>`

The `shared_future<T>` template parameterizes the types of futures which the `executor_type` produces as a result of the `share_future` function. By default, this template is `future_traits<future<T>>::shared_future_type`.

4.8.6 `container<T>`

The `container<T>` template parameterizes the collections of results produced by multi-agent operations. For example, a call to a function such as `execute(exec, f, n)` will produce a result for all the invocations of `f(idx)`. In order to communicate all results to the caller of `.execute()`, we collect them within a container. By default, instantiations of this template are an implementation-defined random access containers. However, an executor might choose to expose a custom container type if the default choice was undesirable. For example, an executor whose agents executed in a NUMA machine might choose to locate results in a memory location local to those execution agents in order to enjoy the benefits of memory locality.

For the purposes of `executor_traits`, a container is any type which may be constructed from a single `shape_type` argument and whose elements may be assigned via `cont[idx] = x`, where `idx` is an instance of

`index_type`. In this way, there is a convenient correspondence between the `shape_type` and `index_type` which describe both multi-agent execution groups and containers collecting their results.

4.9 `executor_traits` Operations

`executor_traits`'s interface includes several member functions for creating execution agents and other associated operations. Every member function `foo` of `executor_traits`, `executor_traits::foo(exec, args...)` is implemented with `exec.foo(args...)`, if such a call is well-formed. Otherwise, `executor_traits` provides provides a default implementation through a lowering onto other functions in the interface. This section describes `executor_traits`'s member functions in detail.

4.9.1 `make_ready_future`

`make_ready_future` generalizes the C++ Concurrency TS's operation of the same name to interoperate with executors. It allows the client of an executor to create a new future in an immediately ready state. We have generalized the interface to invoke a general constructor, rather than simply a copy constructor. This accomodates executors abstracting NUMA architectures, where copies may be relatively expensive.

4.9.2 `future_cast`

`future_cast` generalizes our previously described `future_traits::cast` operation to interoperate with executors. With it, clients may convert a future of any type (even futures associated with a foreign executor) to a type of future associated with the given executor. It also provides a way for different executors' futures to interoperate. In general, `future_cast` must create a continuation (e.g. via `executor_traits::then_execute()`) to perform the conversion operation. However, there are cases where conversions may be performed "for free" without introducing additional asynchronous tasks. We include this operation because we found ourselves frequently requiring such casts within our current prototype's implementation of `executor_traits`. By exposing this operation as an optimization and customization point, it allows executor authors to accelerate this frequently-occurring use case.

4.9.3 `share_future`

Like `future_cast`, `share_future` generalizes our previously described `future_traits::share` operation to interoperate with executors. It is useful when implementing dynamically-sized "fan-out" operations and when constructing important kinds of compositions of executors. For example, an adaptor executor may adapt a collection of underlying executors and present the entire collection as a single logical executor. To implement the `then_execute` operation, the wrapping executor must communicate the incoming future dependency to each of the N underlying executors. The most straightforward way to accomplish this is to share the incoming future N times and call `then_execute()` on each underlying executor. Because the degree of sharing may be arbitrarily large, we provide overloads of the `share_future` operation which return a container of `shared_future<T>`. These overloads capture a frequent use case and allow the executor author to accelerate it.

4.9.4 `max_shape`

The `max_shape` function reports the largest multi-agent group which may be possibly created by a call to an `executor_traits` operation. It is directly analogous to `allocator_traits::max_size`, which reports a similar limit for memory allocation. Because the semantics guaranteed by executors will be limited by available resources (e.g. executors which make guarantees regarding concurrency), it makes sense to provide a means to query these limits. Note that requests for multi-agent groups whose shapes fit within `max_shape` may still fail to execute due to resource unavailability.

4.9.5 `when_all`

The `when_all` function introduces a join operation on a collection of futures by producing a future whose value is a tuple of the input futures' values. The result future becomes ready only when all of the input futures become ready. Our proposed version of `when_all` is similar to the one presented by the Concurrency TS, with one key difference: instead of producing `future<tuple<future<T1>,future<T2>,..., future<TN>>>`, our proposed version produces `future<tuple<T1,T2,...,TN>>`. In other words, our proposed version of `when_all` "unwraps" the input futures' values when tupling. To deal with `void` futures, the corresponding element of the tuple is simply omitted. If all input futures are `void`, our proposed version of `when_all` produces a `void` future instead of introducing a tuple. We believe this formulation promotes better composability because it allows continuation functions to consume the result of `when_all` directly, rather than be required to traffic in futures. In a future C++ ecosystem where we expect a diversity of future types to exist, it would be burdensome to require user lambdas to accomodate all types of futures. Instead, our alternate formulation allows them to operate on the values of function parameters directly.

4.9.6 `when_all_execute_and_select`

The `when_all_execute_and_select` function is an aggressively-parameterized general purpose function which acts as a sort of Swiss Army knife which may be employed to implement any other `executor_traits` operation efficiently. When implementing our current `executor_traits` prototype, without this function, we found it impossible to provide default implementations of many operations without introducing spurious continuations. For example, this issue presents itself when adapting an executor whose native operation creates single-agent tasks to the multi-agent interface; and vice versa.

`when_all_execute_and_select` executes a given function when all of its input future parameters become ready. The given function receives the values of the input futures as separate parameters. In this sense, it is a sort of fusion of `when_all` and `then_execute`. Rather than return the result of the function via a future, the values of the input parameters are forwarded into a future tuple returned by the function. If the caller wishes to retain the result of the invoked function, she may do so by storing a value to one of the input parameters. Because the input parameters get forwarded to the result, the effect is a return by output parameter. Note that `when_all_execute_and_select`'s client may introduce storage for an output parameter with `make_ready_future`.

If the caller of `when_all_execute_and_select` is uninterested in receiving some of the input parameters, it may discard them. The interface accomodates this use case by providing non-type integer template parameters which select a subset of interesting parameters by index. This subset of selected values is returned in the tuple; the remainder is discarded.

4.9.7 `then_execute`

The `then_execute` function allows an executor client to asynchronously enqueue a task as a continuation dependent on the completion of a predecessor task. Like the Concurrency TS, `executor_traits` uses a future to specify the predecessor task. When the predecessor future becomes ready, the continuation may be executed. When the continuation function is invoked, it receives as an argument an index identifying the invocation's execution agent. In the single-agent case, the index is omitted. The continuation function also receives the predecessor future's value as its second argument. Unlike the Concurrency TS, the continuation function consumes a value rather than a value wrapped in a future. We believe consuming values provides better composability and avoids burdening execution agents with exception handling. In our experiments, we have found handling exceptions in the continuation to be awkward or impossible for multi-agent groups. However, this alternate scheme suggests we should design a first class scheme for exception handling as future work.

Any result of the continuation function is returned through the future returned by `then_execute`. If the continuation has no result, `then_execute`'s result is `future<void>`, where `future` is the executor's associated

`future` template. If the continuation has a result of type `T`, then `then_execute`'s result is `future<T>` in the single-agent case. In the multi-agent case, the results of all the individual invocations are collected in a container.

The type of container returned by `then_execute` may be controlled by the caller. If the caller supplies a factory to construct the result via the `result_factory` parameter, then `then_execute`'s result is `future<result_of_t<Factory(shape_type)>>`; otherwise, it is `future<container<T>>`. In any case, execution agent `idx`'s result is located at index `idx` in the resulting container.

4.9.8 `async_execute`

The behavior of `async_execute` is identical to `then_execute`, except that `async_execute` omits the predecessor future which acts as the desired task's dependency. Instead, `async_execute` allows the client to signal that a task is immediately ready for asynchronous execution. In the single-agent case, the function invoked by execution agents created by this operation receives no parameters. In the multi-agent case, the function receives one parameter: the index of the invocation's execution agent. `async_execute` returns results from function invocations identically to `then_execute`.

4.9.9 `execute`

The behavior of the `execute` operation is identical to `async_execute`, except that `execute` is entirely synchronous: the caller of `execute` synchronizes with all execution agents created by the operation. Accordingly, the results of function invocations executed by agents created by `execute` are returned directly to the caller, rather than indirectly through a `future`.

4.10 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`.

- `concurrent_execution_tag` - The exact semantics of this execution category are to be determined. The basic idea is that the function invocations executed by a group of concurrent execution agents are permitted to block each others' forward progress. This guarantee allows concurrent execution agents to communicate and synchronize.
- `nested_execution_tag<OuterExecutionCategory,InnerExecutionCategory>` - The exact semantics of this execution category are to be determined. This category indicates that execution agents execute in a nested organization. The semantics of the *outer* group of execution agents are given by `OuterExecutionCategory`. With each execution agent in the outer group is associated a group of execution agents with semantics given by `InnerExecutionCategory`.

4.10.1 Categories without Corresponding Standard Execution Policies

4.10.1.1 `concurrent_execution_tag` 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 3.3 deadlocks when an individual `spawn()` call fails. To represent this category of execution, we introduce a concurrent execution category `concurrent_execution_tag` as well as a `concurrent_executor` class. We anticipate the semantics of this category of execution to be determined pending ongoing standardization efforts, especially the foundational work of Riegel's Light-Weight Execution Agents proposal (N4439).

Note that unlike other execution categories, `concurrent_execution_tag` currently has no corresponding execution policy. This is because it is doubtful that the semantics of the entire suite of parallel algorithms would allow concurrent element access functions. In principle, one could partition the set of parallel algorithms into those which could support concurrent execution and which could not. As yet, we have not attempted to devise a scheme for categorizing algorithms in such a fashion.

4.10.1.2 `nested_execution_tag` `nested_execution_tag` encodes the category of execution agent hierarchies. It is a template with two parameters: `OuterExecutionCategory` and `InnerExecutionCategory`. These categories name the execution categories of agents in the outer group, and inner groups, respectively. Hiearchies deeper than two levels may be described by nesting `nested_execution_tag` as the `InnerExecutionCategory`.

4.11 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.
- `concurrent_executor` - creates groups of execution agents which execute concurrently.

4.11.1 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.

4.12 A Dynamic Polymorphic Executor

The executor interface we define permits control structures to accept executors as template parameters. The resulting static polymorphism is valuable in avoiding unnecessary overhead and keeping the cost of abstraction near zero. On the other hand, there may be cases where executors must be passed across binary interfaces. To support such use cases, we propose the following `executor` class to act as a polymorphic container for all executor types:

```
class executor
{
  public:
    using execution_category = erased_type;

    template<class Executor>
```

```
    executor(Executor&& exec);

    template<class Function>
    result_of_t<Function()> execute(Function&& f);

    template<class Function, class... Factories>
    vector<result_of_t<Function(size_t, result_of_t<Factories()>...)>>
    execute(Function f, size_t n, Factories... factories);

    template<class Function>
    future<result_of_t<Function()>> async_execute(Function&& f);

    template<class Function, class... Factories>
    future<result_of_t<Function(size_t, result_of_t<Factories()>...)>>
    async_execute(Function f, size_t n, Factories... factories);

    // the rest of the Executor interface follows
    ...
};
```

An `executor` object may be constructed from an object of any type fulfilling the `Executor` concept. For every `executor_traits` function, `executor` exposes a corresponding member function whose implementation manipulates the underlying contained executor. This is possible via typical type erasure techniques.

For example, the following sketch of `executor` demonstrates a possible implementation of `executor::async_execute()`:

```
class executor
{
  public:
    template<class Executor>
    dynamic_executor(Executor&& exec)
      : executor_ptr(make_unique<abstract_executor>(forward<Executor>(exec))
    {}

    template<class Function>
    future<result_of_t<Function()>> async_execute(Function&& f)
    {
      // async_execute a future to any
      future<any> any_fut = executor_ptr->async_execute([]
      {
        // call f() and erase the type of its result
        return any(f());
      });

      using result_type = result_of_t<Function()>;

      // cast the any future to the right type
      return future_traits<future<any>>::template cast<result_type>(any_fut);
    }

  private:
    struct abstract_executor
    {
      virtual ~abstract_executor(){};
```

```
      virtual future<any> async_execute(function<any()>) = 0;
    };

    template<class Executor>
    struct concrete_executor : public abstract_executor
    {
      Executor exec;

      virtual future<any> async_execute(function<any()> f)
      {
        return executor_traits<Executor>::async_execute(exec, f);
      }
    };

    unique_ptr<abstract_executor> executor_ptr;
};
```

The general strategy is to transport all parameters that would be represented by template parameters in a function template interface through type erasing containers such as `any` and `function`. Similar constructions yield the rest of the interface.

4.13 Standard Control Structures

The `executor_traits` interface we have described is a *customization* interface for executors: authors of executors target it by implementing a subset of `executor_traits` functionality as concrete executor member functions. This interface has been chosen so that `executor_traits` can ensure that the entire interface can be supported for all executor types, even when the concrete executor supports only a subset of the interface.

Because its interface is complete, it is sufficient for a client to interface with an executor via `executor_traits` when implementing any computation whose execution is parameterized by that executor. However, working at such a low level of abstraction is often undesirable. Instead, the Standard Library should provide higher level abstract *control structures*, parameterized by executors, for implementing common operations. As a whole, this set of control structures forms an executor *usability* interface.

Examples of existing control structures are the functions `std::async`, `std::invoke`, `std::future::then`, as well as the entire collection of parallel algorithms. These control structures should be extended with new overloads specifying the executor to be used when creating work.

The syntactic convention we propose is that:

- Constructs that create a single agent accept as their first parameter an executor.
- Constructs that create one or more agents accept an execution policy.

We describe a mechanism for composing execution policies with executors in the next section.

4.14 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;
};
```

4.15 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`.

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 participating in ongoing discussions concerning executors.

7 Appendix: Future Work

We have described what we believe is a set of executor functionality necessary to interoperate with and implement the algorithms of the Parallelism TS and other execution control structures in emerging proposals. A full-featured executor design would likely include additional functionality, both for convenience and to broaden its scope. In this appendix, we identify features which may be valuable in a complete design and thus may warrant future work.

7.1 Continuation Interface

Our proposal's handling of continuations in `future_traits::then` and `executor_traits::then_execute` differ from the Concurrency TS. In the Concurrency TS, continutation functions consume parameters wrapped in futures as arguments rather than raw values. We believe that this additional layer of indirection hinders composability because it requires the authors of continuation functions to operate at the level of futures, rather than directly on the function's parameters of interest.

For example, suppose a program uses a variety of different executors. If the programmer wishes to write a continuation function that may interoperate with all these executors' futures, the programmer must choose from a few strategies which may be undesirable:

1. Write separate continuation functions parameterized on each different executor's future type.
2. Write a single continuation functions as function templates whose template parameter is `Future`.
3. Write a single continuation function which is called by a wrapper function whose job is to unwrap the predecessor future and pass its value to the actual continuation.

This scheme of receiving continuation arguments as futures also introduces difficult questions about how the continuation function is allowed to interact with the predecessor future. The continuation scheme described by the Concurrency TS may create a situation where functions which receive a future as a parameter must modulate its behavior based on whether or not it is acting as a continuation. For example, is the continuation function allowed to call a nested `.then()` on the predecessor future? If the answer is no, what is the best way for the future's interface to signal that `.then()` is unavailable? If the answer is yes, are nested continuations a use case that should be enabled?

The benefit of receiving predecessor arguments indirectly through futures is that it does not require additional APIs for exception handling in the presence of continuations. If the predecessor future contains an exception, the exception may be handled directly inside the continuation function. In a C++ ecosystem of only two types of futures (`std::future` and `std::shared_future`) and single-agent continuations, this seems like a reasonable convention.

However, an ecosystem of diverse execution platforms along with multi-agent continuations complicates this convention. For example, consider an executor which abstracts the execution resources of a distributed cluster of machines. On such a platform, a future may abstract the details necessary for tracking the lifetime of execution agents executing on remote machines across a network. In such an environment, where efficient shared memory may be unavailable, it may be awkward to present execution agents responsible for executing continuations with the future corresponding to the predecessor task. This is because the predecessor future may be stored in a remote memory. When implementing our prototype of the functionality proposed in this document, we encountered exactly this issue.

Multi-agent continuations present another issue for handling exceptions. Suppose the continuation executed by a multi-agent execution group receives an exceptional future as a parameter. If the execution agents are executing in parallel, it will be difficult to decide how to handle the exception: which agent(s) should be elected as the handler(s)? If the execution agents are not able to communicate and synchronize, there may be no way to decide this question. At worst, a single "leader" agent may be required to handle the exceptions by default. If the exception is a large `exception_list`, this strategy is likely to be inefficient.

For these reasons, we believe a more powerful design would "lift" the problem of exception handling "above" the level of execution agents. This alternative design would introduce exception handling as a first class primitive into the `executor_traits` interface. For example, it may be possible to separate the problem of exception handling from the `then_execute` function. In this scheme, `then_execute`'s exceptional behavior simply would forward any exception to the future it returns instead of executing its continuation.

For first class exception handling support, this alternative design could introduce a variant to `then_execute` which could receive as parameters both the continuation function and an exception handler separately. Instead of receiving a future as a parameter, the exception handler function could receive an `exception_ptr` or `exception_list`. This primitive could be an overload of `try_execute`, or a function with a different name, say, `then_try_execute`. `then_try_execute` would either execute the given continuation, or in the exceptional case, the given handler. Alternatively, support for exception handling could be exposed as a primitive entirely separate from `then_execute`. We anticipate exploring these alternatives as future work.

7.2 Obtaining executors

Before using executors to launch work, a program must obtain one or more executor objects to use. For some executor types, directly constructing an executor object of the desired type is the most natural mode of use. The simplest example of this case is the `sequential_executor` type. Objects of this type are quite likely to be stateless, and the implementation of `execute()` is a simple loop. Creating an object of type `sequential_executor` and then using it directly or passing it to an algorithm, is a simple and efficient way of using such executors.

In other cases, it may be desirable to have a means of obtaining suitable executors from a query interface. This seems most likely for executors that are meant to correspond to actual hardware resources. For example, a program might wish to enumerate a set of executors corresponding to the available cores of the processor on which the calling thread is running. However, most such use cases also seem to be platform specific. It is an open question whether a query interface applicable to all target platforms can be defined and whether it would prove sufficiently useful to include in a standard executor facility.

7.3 Executor Introspection

Some executor use cases require providing an execution agent with the ability to identify the executor or execution resource on which it is currently executing. Some proposals such as P0113R0 (Kohlkoff) refer to this resource as the execution agent's *execution context*. Executor introspection is useful when an execution agent created by an executor itself wishes to create additional work. Informing the execution agent of the executor from whence it came is useful for making informed decisions about the efficient use of execution resources within that agent.

While it is true that a function to be executed could simply capture the executor as part of its closure, this assumes that the author of the function has access to the executor at the point at which it is defined. This is in general not the case.

Kohlkoff's proposal provides a mechanism based on thread local storage for querying the ambient current executor within an execution agent. It seems unlikely that a mechanism based on thread local storage would be universally desirable or efficient.

Instead, we envision a protocol by which a function invoked through an executor operation could receive a copy of the executor as an optional first parameter. A copy of the executor, rather than reference, would be required for at least two reasons:

1. During asynchronous operations, the executor's lifetime may have ended by the time the function is invoked.
2. Executor operations that create multiple agents would create data races when accessing the same executor.

More sophisticated and convenient mechanisms which track the current executor could be built on top of this basic protocol.

7.4 Asynchronous Progress Guarantees

Our proposal describes a mechanism by which executors advertise the progress guarantees they make about groups of agents via a nested typedef `execution_category`. These guarantees describe an execution agent's progress with respect to the other agents within the same group as that agent. However, our proposal does not currently describe the progress relationship between a thread which launches an asynchronous executor operation and the group of agents created by that operation. It may be similarly useful to use the type system to describe this relationship as well.

8 Appendix: Design Notes

In this appendix, we provide additional insight into the rationale of various aspects of the design of our executor programming model.

8.1 Implementing `executor_traits` and Executors

`executor_traits`' interface is both featureful and designed specifically to make defining new types of executors as straightforward as possible. When a user-defined executor defines a native operation via a member function, the correspondingly-named `executor_traits` function simply forwards to that member function. When a native operation does not exist, a default implementation is applied via a lowering onto other operations within the `executor_traits` and `future_traits` interfaces. To avoid endless recursion, implementors of `executor_traits` should designate particular operations as terminal. In our implementation, these terminal operations are the `execute` functions. For any particular operation, there may be many possible lowerings which satisfy the operation's requirements, and different lowerings may have performance characteristics which vary from platform to platform. For this reason, we do not prescribe a particular lowering for each operation and instead suggest it be implementation-defined. If performance or other characteristics of a particular lowering is desirable, an executor author may implement that lowering directly within the executor's corresponding member function.

This `executor_traits` protocol makes defining new user-defined executors straightforward, yet permits codes to interact with user-defined executors as if they supported the complete executor interface natively. In fact, in our current implementation, user-defined executors need not define any native operations at all. Yet, these "empty" executors still remain compatible with `executor_traits` because `executor_traits` "fills in" the missing features. In a final specification, we anticipate introducing minimal requirements on executor authors. For example, to avoid confusing unrelated types for executors, we may choose to require executors to define at least one member function corresponding to an `executor_traits` operation.

8.2 Factories

In our initial design, executor clients passed shared parameter values directly to `executor_traits` operations rather than indirectly through a factory. It was `executor_traits`' job to make copies of these values for each group of execution agents. We realized that these copies often would be relatively expensive, especially on NUMA architectures. Moreover, on distributed, cluster-based systems, copies might be impossible. The `Factory`-based interface nicely avoids these problems while enabling customizable memory allocation.

User control over the construction of the result of these operations is also desirable, for the same reasons. When constructing an object to contain multiple results, the factory is invoked with the `shape` parameter given to the executor operation.

8.3 Multidimensionality

When we reason about an executor's "shape", we are describing the index space of each bulk task it creates. For example, suppose we have an executor abstracting a thread pool with a bounded number of k threads at its disposal. This executor advertises that the execution agents it creates during a multi-agent bulk task are mutually concurrent. Since this executor has access to only k threads, this means the maximum number of agents in a multi-agent bulk task it could ever create would be no more than k. Otherwise, the executor could not satisfy the concurrency guarantee it makes of its agents. In this example, the "shape" of these multi-agent bulk tasks is one-dimensional, and can be encoded with a `size_t`. However, if it is convenient to some application domain, the author of the executor could choose to impart dimensionality on these bulk tasks by introducing a different organization. For example, the author could partition the thread pool's k threads into a rectangular domain, and assign agents two-dimensional indices, where each 2D index corresponds to a lattice point within the rectangle. The only difference between the two examples is simply how agents are labeled: the underlying thread pool implementation could remain unchanged.

An alternative design would simply require that executors index the execution agents they create one-dimensionally. If multidimensionality is required by a client of the executor, the client could layer multidimensionality on top by reindexing the one-dimensional indices produced by the executor. Though it is simpler, the drawback of such a design is that reindexing can be an expensive operation relative to the cost of each agent's task. For example, lifting one-dimensional indices into higher dimensions often requires relatively expensive integer operations such as division and modulus. By the same token, projecting higher-dimensional indices into lower dimensions often requires multiplication and addition. Given that many executors will be able to produce multi-dimensional indices efficiently and directly, it seems unwise to bake an unavoidable inefficiency into the programming model at a very low level. Moreover, many client applications of executors will require reasoning about multidimensional domains. In the worst case, the simpler design would introduce multiple occurrances of reindexing: a natively multi-dimensional executor's indices would first be projected to 1D, and then lifted back into multiple dimensions by the client. We avoid these inefficiencies by designing explicitly for multidimensionality at the level of the executor interface.

To our knowledge, there is no precedent for multidimensional indexing within the existing C++ Standard Library, though N4512 proposes `std::bounds` and `std::index` as instances of this kind of functionality. Our requirements for these types are simple: for the purposes of `executor_traits`, a type may be used as a shape if it is an unsigned integral type or if it is a tuple-like type whose elements are themselves shapes. Index types have similar requirements, though in the terminal case, index types are permitted to be signed rather than unsigned. Note that these definitions are recursive, permitting hierarchical shapes and indices. Observe further that these definitions include both `std::bounds` and `std::index` as valid shape and index types, respectively. From these simple definitions, executors may manipulate and reason about shape and index types uniformly without requiring that all executors use a particular shape or index type.

8.4 Executor Operation Parameter Ordering

The order of parameters of the `executor_traits` functions `execute`, `async_execute`, and `then_execute` was chosen to conform to the following conventions:

1. Multi-agent overloads should append a `shape` parameter to the single-agent overload.
2. The parameter order of the user function to be invoked should match the order of the parameters to the `executor_traits` function.
3. The execution agent index, if it exists, should be the first parameter of the user function.

`then_execute` disrupts these conventions due to its `Future` parameter. The single-agent version of `then_execute` receives the `Future` as its second parameter:

```
template<class Function, class T>
future<result_of_t<Function(T)>>
  then_execute(Function&& f, future<T>& fut);
```

If the multi-agent `then_execute` overload extended this function by appending the shape parameter, the signature would look like this:

```
template<class Future, class T, class... Factories>
future<result_of_t<Function(T&,index_type,result_of_t<Factories()>&...)>>
  then_execute(Function f, future<T>& fut, shape_type shape, Factories... factories);
```

The signature of the user function `f` would need to be `f(T&, index_type, result_of_t<Factories>&...)` to match this parameter order. However, this violates 3.

The more important conventions are 2. and 3, so the form of `then_execute` we propose violates convention 1.:

```
template<class Future, class T, class... Factories>
future<result_of_t<Function(index_type,T&,result_of_t<Factories()>&...)>>
  then_execute(Function f, shape_type shape, future<T>& fut, Factories... factories);
```

With corresponding user function signature:

```
f(index_type, T&, result_of_t<Factories()>&...)
```

Return type factories further disrupt these conventions. We insert the `result_factory` between `f` and `shape` to position `result_factory` to the left of the function parameters, because it corresponds to the result of the user function.

9 Appendix: Changelog

0. P0058R1
    0. Rework introductory paragraph of Section 4.2.
    1. Define rules for execution agent creation in Section 4.2.1.
    2. Define rules for execution agent synchronization in Section 4.2.1.
    3. Added “Fire-and-Forget” Section and removed corresponding section from future work.
    4. Added section describing the dynamic polymorphic `executor` container.
    5. Added `shared_future_type` and `share` to `future_traits` synopsis.

6. Added `shared_future` and `share_future` to `executor_traits` synopsis.
7. Added descriptions of `executor_traits::shared_future` and `executor_traits::share_future`.
8. Added section Standard Control Structures arguing for a usability interface for executors and a convention for composing executors with control structures.
9. Added Future Work section Executor Introspection.
10. Added Future Work section Asynchronous Progress Guarantees.

1. P0058R0

    0. Added changelog section.
    1. Added `future_traits` sketch along with description and removed corresponding section from future work.
    2. Added section describing multidimensionality and removed corresponding section from future work.
    3. Added `task_block` implementation sketch.
    4. Added `concurrent_execution_tag` and `concurrent_executor` and removed corresponding section from future work.
    5. Describe in detail the behavior of each `executor_traits` function.
    6. `executor_traits` functions now return the results of function invocations.
    7. Added exception handling to future work section.
    8. Added section on obtaining executors to future work section.
    9. Added shared parameters section and removed the corresponding section from future work.
    10. Added section on continuation interface to future work section.
    11. Added section on execution hierarchies and removed the corresponding section from future work.
    12. Added design notes appendix.

2. N4406

    1. Initial Document

10 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.
4. N4406 - Execution Policies Need Executors, J. Hoberock et al., 2015.
5. N4411 - Task Block (formerly Task Region) R4, P. Halpern et al., 2015.
6. N4439 - Light-Weight Execution Agents, T. Riegel, 2015.
7. N4501 - Working Draft, Technical Specification for C++ Extensions for ConcurrencyA. Laksberg, 2015.
8. N4512 - Multidimensional bounds, offset and array_view, revision 7, L. Mendakiewicz & H. Sutter, 2015.
9. P0113 - Executors and Asynchronous Operations, Revision 2, C. Kohlkoff, 2015.