Title:	Executors Design Document
Document Number:	P0761R0
Authors:	Jared Hoberock, jhoberock@nvidia.com Michael Garland, mgarland@nvidia.com Chris Kohlhoff, chris@kohlhoff.com Chris Mysen, mysen@google.com Carter Edwards, hcedwar@sandia.gov Gordon Brown, gordon@codeplay.com Michael Wong, michael@codeplay.com
Date:	2017-07-31
Audience:	SG1 - Concurrency and Parallelism
Reply-to:	sg1-exec@googlegroups.com
Abstract:	This paper is a companion to P0443 and describes the executors programming model it specifies. This paper is directed toward readers who want to understand in detail the mechanics of P0443's programming model, and the rationale underpinning the choices of that model's design.

1 Introduction

Execution is a fundamental concern of C++ programmers. Every piece of every program executes somehow and somewhere. For example, the iterations of a `for` loop execute in sequence on the current thread, while a parallel algorithm may execute in parallel on a pool of threads. A C++ program's performance depends critically on the way its execution is mapped onto underlying execution resources. Naturally, the ability to reason about and control execution is crucial to the needs of performance-conscious programmers.

In general, there is no standard and ubiquitous way for a C++ programmer to control execution, but there should be. Instead, programmers control execution through diverse and non-uniform facilities which are often coupled to low-level platform details. This lack of common interface is an obstacle to programmers that wish to target execution-creating facitilies because each must be targeted idiosyncratically. For example, consider the obstacles a programmer must overcome when targeting a simple function at one of many facilities for creating execution:

```
void parallel_for(int facility, int n, function<void(int)> f) {
  if(facility == OPENMP) {
    #pragma omp parallel for
    for(int i = 0; i < n; ++i) {
      f(i);
    }
  }
  else if(facility == GPU) {
    parallel_for_gpu_kernel<<<n>>>(f);
  }
  else if(facility == THREAD_POOL) {
    global_thread_pool_variable.submit(n, f);
  }
}
```

Complexity. The first obstacle highlighted by this example is that each facility's unique interface necessitates an entirely different implementation. As the library introduces new facilities, each introduction intrudes upon `parallel_for`'s implementation. Moreover, the problem worsens as the library introduces new functions.

While the maintainance burden of a single simple function like `parallel_for` might be manageable, consider the maintainance complexity of the cross product of a set of parallel algorithms with a set of facilities.

Synchronization. Execution created through different facilities has different synchronization properties. For example, an OpenMP parallel for loop is synchronous because the spawning thread blocks until the loop is complete due to an implicit barrier at the end of the parallel region by default. In contrast, the execution of GPU kernels is typically asynchronous; kernel launches return immediately and the launching thread continues its execution without waiting for the kernel's execution to complete. Work submitted to a thread pool may or may not block the submitting thread. Correct code must account for these synchronization differences or suffer data races. To minimize the possibility of races, these differences should be exposed by library interfaces.

Non-Expressivity. Our `parallel_for` example restricts its client to a few simple modes of execution through the use of a single integer choosing which facility to use. These modes are so restrictive that even simple generalizations are out of reach. For example, suppose the programmer wants to supply their own thread pool rather than use the global thread pool, or perhaps the global pool augmented with some notion or priority or affinity? Similarly, perhaps the programmer may wish to target a specific GPU or collection of GPUs rather than some GPU implied by the surrounding environment. The vocabulary of `parallel_for`'s interface is not rich enough to express these subtleties.

This example illustrates the kinds of problems we propose to solve with **executors**, which we envision as a standard way to create execution in C++. There has already been considerable work within the C++ Standards Committee to standardize a model of executors. Google's proposal interfaced executors to thread pools and was briefly incorporated into a draft of the Concurrency TS [1, 2, 11–14]. Next, Chris Kohlhoff's proposal focused on asynchronous processing central to the requirements of the Networking TS [10]. Finally, NVIDIA's proposal focused on bulk execution for the parallel algorithms of the Parallelism TS [3–5]. A unification of this work [6, 7] specifies the most current version [7] of the executor model this paper describes. Our goal in this document is to outline our vision for programming with executors in C++ and explain how we believe our design achieves this vision.

2 Terminology

We envision executors as an abstraction of diverse underlying facilities responsible for implementing execution. This abstraction will introduce a uniform interface for creating execution which does not currently exist in C++. Before exploring this vision, it will be useful to define some terminology for the major concepts involved in our programming model: execution resources, execution contexts, execution functions, execution agents and executors.

An **execution resource** is an instance of a hardware and/or software facility capable of executing a callable function object. Different resources may offer a broad array of functionality and semantics and exhibit different performance characteristics of interest to the performance-conscious programmer. For example, an implementation might expose different processor cores, with potentially non-uniform access to memory, as separate resources to enable programmers to reason about locality.

Typical examples of an execution resource can range from SIMD vector units accessible in a single thread to an entire runtime managing a large collection of threads.

A program may require creating execution on multiple different kinds of execution resources, and these resources may have significantly different capabilities. For example, callable function objects invoked on a `std::thread` have the repertoire of a Standard C++ program, including access to the facilities of the operating system, file system, network, and similar. By contrast, GPUs do not create standard threads of execution, and the callable function objects they execute may have limited access to these facilities. Moreover, functions executed by GPUs typically require special identification by the programmer and the addresses of these functions are incompatible with those of standard functions. Because execution resources impart different freedoms and restrictions to the execution they create, and these differences are visible to the programmer, we say that they are **heterogeneous**.

Our proposal does not currently specify a programming model for dealing with heterogeneous execution resources. Instead, the execution resources representable by our proposal are implicitly **homogeneous** and execute Standard C++ functions. We envision that an extension of our basic executors model will deal with heterogeneity by exposing execution resource **architecture**. However, the introduction of a notion of concrete architecture into C++ would be a departure from C++'s abstract machine. Because such a departure will likely prove controversial, we think a design for heterogeneous resources is an open question for future work.

An **execution context** is a program object that represents a specific collection of execution resources and the **execution agents** that exist within those resources. In our model, execution agents are units of execution, and a 1-to-1 mapping exists between an execution agent and an invocation of a callable function object. An agent is bound[1] to an execution context, and hence to one or more of the resources that the context represents.

Typical examples of an execution context are a thread pool or a distributed or heterogeneous runtime.

An **execution agent** is a unit of execution of a specific execution context that is mapped to a single invocation of a callable function on an execution resource. An execution agent can too have different semantics which are derived from the execution context.

Typical examples of an execution agent are a CPU thread or GPU execution unit.

An **executor** is an object associated with a specific execution context. It provides one or more **execution functions** for creating execution agents from a callable function object. The execution agents created are bound to the executor's context, and hence to one or more of the resources that context represents.

Executors themselves are the primary concern of our design.

3 Using Executors

We expect that the vast majority of programmers will interact with executors indirectly by composing them with functions that create execution on behalf of a client.

3.1 Using Executors with the Standard Library

Some functions, like a new executor-aware overload of `std::async`, will receive executors as parameters directly:

```
// get an executor through some means
my_executor_type my_executor = ...

// launch an async using my executor
auto future = std::async(my_executor, [] {
  std::cout << "Hello world, from a new execution agent!" << std::endl;
});
```

This use of `std::async` has semantics similar to legacy uses of `std::async`, but there are at least two important differences. First, instead of creating a new (internal) `std::thread`, this overload of `std::async` uses `my_executor` to create an **execution agent** to execute the lambda function. In this programming model, execution agents act as units of execution, and every use of an executor to create execution creates

[1]An execution agent is bound to an execution context and thus is restricted to execute only on the associated specific collection of execution resources. For example, if a context includes multiple threads then the agent may execute on any of those threads, or migrate among those threads.

one or more execution agents. Secondly, the type of future object returned by this overload of `std::async` depends on the type of `my_executor`. We will discuss executor-defined future types in a later section.

Other functions will receive executors indirectly. For example, algorithms will receive executors via execution policies:

```
// get an executor through some means
my_executor_type my_executor = ...

// execute a parallel for_each "on" my executor
std::for_each(std::execution::par.on(my_executor), data.begin(), data.end(), func);
```

In this example, the expression `par.on(my_executor)` creates a parallel execution policy whose associated executor is `my_executor`. When `std::for_each` creates execution it will use the executor associated with this execution policy to create multiple execution agents to invoke `func` in parallel.

3.2 Using Executors with the Networking TS

The Networking TS provides numerous asynchronous operations, which are operations that allow callers to perform network-related activities without blocking the initiating thread. Whenever an asynchronous operation is initiated, the caller supplies a completion handler – a function object to be invoked when the operation completes and passed the results of the operation. The Networking TS uses executors to determine when, where and how a completion handler should be invoked. Every completion handler has an associated executor, and a conforming asynchronous operation queries the completion handler to obtain this executor object.

By default, a completion handler is associated to the system executor. This means that when the user writes:

```
// obtain an acceptor (a listening socket) through some means
tcp::acceptor my_acceptor = ...

// perform an asynchronous operation to accept a new connection
acceptor.async_accept(
    [](std::error_code ec, tcp::socket new_connection)
    {
      ...
    }
  );
```

the user places no constraints on when and where the completion handler (a lambda in this example) will be invoked. (In practice, other things will constrain the invocation to specific threads. For example, if the user is only running the `std::experimental::net::io_context` object from a single thread then invocation will occur only on that thread.)

Instead, if the user wants the completion handler to be invoked using a particular set of rules, they may specify an associated executor using the `std::experimental::net::bind_executor` function:

```
// obtain an acceptor (a listening socket) through some means
tcp::acceptor my_acceptor = ...

// obtain an executor for a specific thread pool
auto my_thread_pool_executor = ...

// perform an asynchronous operation to accept a new connection
acceptor.async_accept(
    std::experimental::net::bind_executor(my_thread_pool_executor,
      [](std::error_code ec, tcp::socket new_connection)
      {
        ...
      }
    )
  );
```

The example above runs the completion handler on a specific thread pool. Other common reasons for an associated executor include guaranteeing non-concurrency for a group of completion handlers (by using a `std::experimental::net::strand` executor), or to embellish invocation with logging or tracing.

The `std::experimental::net::bind_executor` function is a convenience function for specifying the associated executor. For user-defined completion handler types, the association may also be established by providing a nested `executor_type` typedef and `get_executor` member function, or by specializing the `std::experimental::net::associated_executor` trait.

The vast majority of Networking TS users are expected to be pure consumers of asynchronous operations, as illustrated above. However, more advanced uses may require the development of custom asynchronous operations. In this case the library user will write code to interact with the associated executor directly. This interaction will adhere to the following pattern.

To begin, an asynchronous operation will obtain the associated executor from the completion handler by calling the `get_associated_executor` function:

```
auto ex = std::experimental::net::get_associated_executor(completion_handler)
```

To inform the executor that there is a pending operation, the asynchronous operation creates an `executor_work_guard` object and keeps a copy of it until the operation is complete. This object automatically calls the executor's `on_work_started` and `on_work_finished` member functions.

```
auto work = std::experimental::net::make_work_guard(ex);
```

If an asynchronous operation completes immediately (that is, within the asynchronous operation's initiating function), the completion handler is scheduled for invocation using a never-blocking executor created from the original executor `ex`:

```
auto never_blocking_ex = std::execution::require(ex, std::execution::never_blocking);

never_blocking_ex.execute(
    [h = std::move(completion_handler), my_result]() mutable
    {
      h(my_result);
    }
  );
```

A never-blocking executor is used above to ensure that operations that always complete immediately do not lead to deadlock or stack exhaustion. On the other hand, if the operation completes later then the asynchronous operation invokes the completion handler using a possibly-blocking executor:

```
auto possibly_blocking_ex = std::execution::require(ex, std::execution::possibly_blocking);

possibly_blocking_ex.execute(
    [h = std::move(completion_handler), my_result]() mutable
    {
      h(my_result);
    }
  );
```

This allows the result to be delivered to the application code with minimal latency.

Finally, if an asynchronous operation consists of multiple intermediate steps, these steps may be scheduled using the defer execution function: these steps may be scheduled using an executor which may execute as continuations of the calling thread:

```
auto continuation_ex = std::execution::prefer(ex, std::execution::continuation);

// asynchronous operation consists of multiple steps
continuation_ex.execute(my_intermediate_complete_handler);
```

This informs the executor of the relationship between the intermediate completion handlers, and allows it to optimize the scheduling and invocation accordingly.

3.3 Using Executors with Application-Level Libraries

When composing executors with functions which use them to create execution agents, we use the following convention. When a function uses an executor to create a single agent, the first parameter is an executor. When a function uses an executor to create multiple agents at once, the first parameter is an execution policy whose associated executor is the implied executor to use. The rationale is that the requirements imposed by execution policies include ordering among agents organized into a group executing as a unit. Logically, requirements that only apply to agents executing as a group are nonsensical to functions which only create a single agent.

3.3.1 Executors Associated with Execution Policies

For example, the library interface for a numerical solver of systems of linear equations might parameterize a `solve` function like this:

```
template<class ExecutionPolicy>
void solve(ExecutionPolicy policy, const matrix& A, vector& x, const vector& b) {
  // invert the matrix using the policy
  matrix A_inverse = invert(policy, A);

  // multiply b by A's inverse to solve for x
  x = multiply(A_inverse, b);
}
```

By organizing `solve`'s implementation around an execution policy, it is insulated from the details of creating execution. This frees the implementer to apply their expertise to the application domain – namely, numerical linear algebra – rather than orthogonal problems introduced by execution. Simultaneously, this organization decouples `solve` from any particular kind of execution. Because an execution policy is exposed in `solve`'s interface and forwarded along through its calls to lower-level functions, `solve`'s client is in complete control over its execution.

This is a powerful way to compose libraries. For example, a client of `solve` may initially choose to execute `solve` sequentially:

```
solve(std::execution::seq, A, x, b);
```

Later, the client may introduce parallelism as an optimization:

```
solve(std::execution::par, A, x, b);
```

A further optimization might locate `solve`'s execution nearer to the memory it accesses in order to avoid the performance hazards of non-uniform access. Associating an executor with affinity for particular processor cores could constrain `solve` to execute on those cores:

```
executor_with_affinity exec = ...
```

```
solve(std::execution::par.on(exec), A, s, b);
```

In the meantime, the efficiency of `solve` or the quality of its output may have been improved through the use of a more sophisticated algorithm. Composing libraries around execution policies and executors allows these two programming activities to proceed independently.

3.3.2 Executors for Coarse-Grained Tasks

A similar argument holds for application library interfaces that consume executors directly in order to create execution agents one by one. For example, consider a library function that executes some long-running task. To avoid requiring clients to wait for its completion, this function immediately returns a future object corresponding to its completion:

```
template<class Executor>
std::execution::executor_future_t<Executor,void>
long_running_task(const Executor& exec) {
  // first, start two subtasks asynchronously
  auto future1 = subtask1(exec);
  auto future2 = subtask2(exec);

  // finally, start subtask3 when the first two are complete
  return subtask3(exec, future1, future2);
}
```

Consider `long_running_task`'s interface. Because the ordering requirements imposed by an execution policy are irrelevant to `long_running_task`'s semantics, it is parameterized by an executor instead of an execution policy. The type of future object returned by `long_running_task` is given by the type trait `std::execution::executor_future_t`, which names the type of future returned when asynchronous execution is created by the type of executor used as its template parameter. The implementation forwards along the executor similarly to our previous example. First, the executor is passed to calls to two independent, asynchronous subtasks. Then, the two futures corresponding to these subtasks along with the executor are used to call the third subtask. Its asynchronous result becomes the overall resulting future object.

3.4 Obtaining Executors

So far, we have not addressed the issue of actually obtaining an executor to use. We believe that there will be many different sources of executors.

Executors from contexts. Though our proposal makes no such requirement, we believe many execution contexts will provide methods to create executors bound to them. For example, our proposal defines `static_thread_pool`, which is an execution context representing a simple, manually-sized thread pool. Clients may receive an executor which creates work on a `static_thread_pool` by calling its `.executor` method:

```
// create a thread pool with 4 threads
static_thread_pool pool(4);

// get an executor from the thread pool
auto exec = pool.executor();

// use the executor on some long-running task
auto task1 = long_running_task(exec);
```

Executors from policies. Another standard source of executors will be the standard execution policies, which will each have a similar `.executor` method:

```
// get par's associated executor
auto par_exec = std::execution::par.executor();

// use the executor on some long-running task
auto task2 = long_running_task(par_exec);
```

System executors. We may also decide to provide access to implied "system" executors used by various Standard Library functions. For example, the legacy overload `std::async(func)` could be redefined in terms of executors in a way that also preserves its semantics. If the implied executor used by the legacy overload `std::async(func)` were made available, programmers porting their existing codes to our proposed new overload `std::async(exec, func)` could target executors in a way that preserved the original program's behavior.

Executor adaptors. Still other executors may be "fancy" and adapt some other type of base executor. For example, consider a hypothetical logging executor which prints output to a log when the base executor is used to create execution:

```
// get an executor from a thread pool
auto exec = pool.executor();

// wrap the thread pool's executor in a logging_executor
logging_executor<decltype(exec)> logging_exec(exec);

// use the logging executor in a parallel sort
std::sort(std::execution::par.on(logging_exec), my_data.begin(), my_data.end());
```

We do not believe this is an exhaustive list of executor sources. Like other adaptable, user-defined types ubiquitous to C++, sources of executors will be diverse.

4 Building Control Structures

The previous section's examples illustrate that for the vast majority of programmers, executors will be opaque objects that merely act as abstract representations of places where execution happens. The mechanics of direct interaction with executors to create execution are irrelevant to this audience. However, these mechanics are relevant to the small audience of programmers implementing **control structures**. By control structure, we mean any function which uses an executor, directly or indirectly, to create execution. For example, `std::async`, the parallel algorithms library, `solve`, and `long_running_task` are all examples of control structures because they use a client's executor to create execution agents. In particular, our proposal adds executor support to the following control structures from the Standard Library and technical specifications.

Table 2: The control structures we propose to introduce.

Standard Library	Concurrency TS	Parallelism TS	Networking TS
`invoke`	`future::then`	`define_task_block`	`post`
`async`	`shared_future::then`	`define_task_block-_restore_thread`	`dispatch`
parallel algorithms		`task_block::run`	`defer`
			asynchronous operations

Standard Library	Concurrency TS	Parallelism TS	Networking TS
			`strand<>` (N.B. although an executor itself, a `strand` acts as a control structure over other executors in order to guarantee non-concurrent execution)

4.1 Fundamental Interactions with Executors via Execution Functions

Some control structures (e.g., `solve`) will simply forward the executor to other lower-level control structures (`invert` and `multiply`) to create execution. However, at some low level of the call stack, one or more control structures must actually interact with the executor at a fundamental level. `std::async` is an illustrative example. Consider a possible implementation:

```
template<class Executor, class Future, class... Args>
execution::executor_future_t<Executor,auto>
async(const Executor& exec, Function&& f, Args&&... args) {
  // bind together f with its arguments
  auto g = bind(forward<Function>(f), forward<Args>(args)...);

  // introduce single-agent, two-way execution requirements
  auto new_exec = execution::require(exec, execution::single, execution::twoway);

  // implement with execution function twoway_execute
  return new_exec.twoway_execute(g);
}
```

The implementation proceeds in three steps. First, we package `f` along with its arguments into a nullary function, `g`. Next, using `execution::require`, we introduce requirements for single-agent, two-way executor **properties**. This step uses `exec` to produce a new executor, `new_exec` which encapsulates these requirements. Finally, we call an **execution function**. Execution functions are the fundamental executor interactions which create execution. In this case, that execution functions is `.twoway_execute`, which creates a single execution agent to invoke a nullary function. The agent's execution is asynchronous, and `.twoway_execute` returns a future corresponding to its completion.

Before describing the precise semantics of execution functions in detail, we will first describe executor properties which affect the way they behave.

4.1.1 Executor Properties

Executor properties are objects associated with an executor. Through calls to `execution::require` and `execution::prefer`, users may either strongly or weakly associate a property with a given executor. Such reassociations may transform the executor's type in the process. For example, in our example implementation of `std::async`, we use `execution::require` to strongly require the `single` and `twoway` properties from `exec`. This operation produces `new_exec`, whose type may be different from the type of the original executor, `exec`.

4.1.1.1 Standard Properties

Our design includes eight sets of properties we have identified as necessary to supporting the immediate needs of the Standard Library and other technical specifications. Two of these sets describe the directionality and cardinality of executor member functions which create execution. When a user requests these properties, they are implicitly requesting an executor which provides the execution functions implied by the request.

Directionality. Some execution functions return a future object corresponding to the eventual completion of the created execution. Other execution functions allow clients to "fire-and-forget" their execution and return `void`. We refer to fire-and-forgetful execution functions as "one-way" while those that return a future are "two-way"[2]. Two-way execution functions allow executors to participate directly in synchronization rather than require inefficient synchronization out-of-band. On the other hand, when synchronization is not required, one-way execution functions avoid the cost of a future object.

The directionality properties are `execution::oneway`, `execution::twoway`, and `execution::then`. An executor with the `execution::oneway` property has either or both of the one-way execution functions: `.execute()` or `.bulk_execute()`. An executor with the `execution::twoway` property has either or both of the two-way execution functions: `.twoway_execute()` or `.bulk_twoway_execute()`. An executor with the `execution::then` property has either or both of the `then_` execution functions: `then_execute()` or `bulk_then_execute()`. Because a single executor type can have one or more of these member functions all at once, these properties are not mutually exclusive.

Cardinality. Cardinality describes how many execution agents the use of an execution function creates, whether it be a single agent or multiple agents. We include bulk agent creation in our design to enable executors to amortize the cost of execution agent creation over multiple agents. By the same token, support for single-agent creation enables executors to apply optimizations to the important special case of a single agent.

There are two cardinality properties: `execution::single` and `execution::bulk`. An executor with the `execution::single` property has at least one execution function which creates a single execution agent from a single call: `.execute()`, `twoway_execute()`, or `then_execute()`. Likewise, an executor with the `execution::bulk` property has at least one execution function which creates multiple execution agents in bulk from a single call: `.bulk_execute()`, `bulk_twoway_execute()`, or `.bulk_then_execute`(). Like the directionality properties, the cardinality properties are not mutually exclusive, because it is possible for a single executor type to have both kinds of execution functions.

Unlike the directionality and cardinality properties, which imply the existence of certain execution functions, other properties modify the behavior of those execution functions. Moreover, those properties modify the behavior of *all* of an executor's execution functions.

Blocking. An executor's execution functions may or may not block their client's execution pending the completion of the execution they create. Depending on the relationship between client and executed task, blocking guarantees may be critical to either program correctness or performance. An executor may guarantee that its execution functions never block, possibly block, or always block their clients.

There are three mutually-exclusive blocking properties : `execution::never_blocking`, `execution::possibly-_blocking`, and `execution::always_blocking`. The blocking properties guarantee the blocking behavior of all of an executor's execution functions. For example, when `.execute(task)` is called on an executor whose blocking property is `execution::never_blocking`, then the forward progress of the calling thread will never be blocked pending the completion of the execution agent created by the call. The same guarantee holds for every other execution function of that executor. The net effect is that the blocking behavior of execution functions is completely a property of the executor type. However, that property can be changed at will by transforming the executor into a different type through a call to `require()`.

Continuations. There are two mutually-exclusive properties to indicate that a task submitted to an executor represents a continuation of the calling thread: `execution::continuation` and `execution::not_continuation`. A client may use the `execution::continuation` property to indicate

[2]We think that the names "one-way" and "two-way" should be improved.

that a program may execute more efficiently when tasks are executed as continuations of the client's calling thread.

Future task submission. There are two mutually-exclusive properties to indicate the likelihood of additional task submission in the future. The `execution::outstanding_work` property indicates to an executor that additional task submission is likely. Likewise, the `execution::not_outstanding_work` property indicates that no outstanding work remains.

Bulk forward progress guarantees. There are three mutually-exclusive properties which describe the forward progress guarantees of execution agents created in bulk. These describe the forward progress of an agent with respect to the other agents created in the same submission. These are `execution::bulk_sequenced_execution`, `execution::bulk_parallel_execution`, and `execution::bulk_unsequenced_execution`, and they correspond to the three standard execution policies.

Thread execution mapping guarantees. There are two mutually-exclusive properties for describing the way in which execution agents are mapped onto threads. `thread_execution_mapping` guarantees that execution agents are mapped onto threads of execution, while `new_thread_execution_mapping` extends that guarantee by guaranteeing that each execution agent will be individually executed on a newly-created thread of execution. These guarantees may be used by the client to reason about the availability and sharing of thread-local storage over an execution agent's lifetime.

Allocators. A final property, `allocator`, associates an allocator with an executor. A client may use this property to suggest the use of a preferred allocator when allocating storage necessary to create execution. Of the properties we have described, `allocator(alloc)` is the only one which takes an additional parameter; namely, the desired allocator to use.

The properties of execution created by fundamental executor interactions vary along three dimensions we have identified as critical to an interaction's correctness and efficiency. The combination of these properties determines the customization point's semantics and name, which is assembled by concatenating a prefix, infix, and suffix.

4.1.1.2 User-Defined Properties

In addition to the standard properties enumerated by the previous section, our design also allows user-defined properties. A programmer may introduce a user-defined executor property by defining a property type and specializing either the `require` or `prefer` customization points. When `execution::require` (respectively, `execution::prefer`) is used with an executor and the user's property, the user's specialization of `require` (`prefer`) may introduce the property to the executor.

As an example, consider the task of adding logging to an executor. We wish to note every time the executor creates work through `.execute` by printing a message. First, we create a user-defined property and a "fancy" executor adaptor which wraps another executor, printing a message each time the wrapped executor creates work:

```
// a user-defined property for logging
struct logging { bool on; };

// an adaptor executor which introduces logging
template<class Ex>
struct logging_executor
{
  bool on;
  Ex wrapped;

  auto context() const { return wrapped.context(); }
  bool operator==(const logging_executor& other) { return wrapped == other.wrapped; }
  bool operator!=(const logging_executor& other) { return wrapped != other.wrapped; }
  template<class Function>
  void execute(Function f)
  {
    if(on) std::cout << ".execute() called" << std::endl;
    wrapped.execute(f);
  }

  // intercept require & prefer requests for logging
  logging_executor require(logging l) const { return { l.on, wrapped }; }
  logging_executor prefer(logging l) const { return { l.on, wrapped }; }

  // forward other kinds of properties to the wrapped executor
  template<class Property>
  auto require(const Property& p) const
    -> logging_executor<execution::require_member_result_t<Ex, Property>>
  {
    return { on, wrapped.require(p) };
  }

  template<class Property>
  auto prefer(const Property& p) const
    -> logging_executor<execution::prefer_member_result_t<Ex, Property>>
  {
    return { on, wrapped.prefer(p) };
  }
};
```

In addition to the typical executor member functions, our `logging_executor` also provides implementations of `.require` and `.prefer`. The first of these are overloads which intercept our special `logging` property, and their implementations return a copy of the `logging_executor` with logging enabled or disabled, as indicated by the state of the `logging` parameter. The last two members of `logging_executor` intercept “foreign” executor properties and forward them to the wrapped executor. Their result is a new `logging_executor` which wraps the type of executor returned by `wrapped.require(p)`, or `wrapped.prefer(p)`, respectively. This type is given by the type traits `execution::require_member_result_t` and `execution::prefer_member_result_t`.

The final task is to provide a free function overload of `require` to introduce a `logging_executor` when the provided executor does not have the `logging` property:

```
template<class Ex>
std::enable_if_t<!execution::has_require_member_v<Ex, logging>, logging_executor<Ex>>
require(Ex ex, logging l)
{
  return { l.on, std::move(ex) };
}
```

This overload of `require` is only enabled when the given executor type cannot natively introduce logging through a call to `.require(logging)`. Note that because our `logging_executor` does provide such a member function, this overload of `require` is disabled for `logging_executors`. This policy prevents redundantly nested instantiations of the form `logging_executor<logging_executor<...>>`.

4.1.2 Execution Functions

Once a user has introduced any requirements or preferences, they use the resulting executor's **execution functions** to actually create execution. There are six of these, resulting from the cross product of the cardinality and directionality properties:

Name	Cardinality	Directionality
`execute`	single	oneway
`twoway_execute`	single	twoway
`then_execute`	single	then
`bulk_execute`	bulk	oneway
`bulk_twoway_execute`	bulk	twoway
`bulk_then_execute`	bulk	then

In a concrete context, the type of the executor, and therefore its suite of member functions, is known a priori. In such cases, execution functions may be called safely without the use of `execution::require`:

```
void concrete_context(const my_oneway_single_executor& ex)
{
  auto task = ...;
  ex.execute(task);
}
```

In a generic context, the programmer should use `execution::require` to ensure that the necessary execution function is available.

```
template<class Executor>
void generic_context(const Executor& ex)
{
  auto task = ...;

  // ensure .execute() is available with execution::require()
  execution::require(ex, execution::single, execution::oneway).execute(task);
}
```

In any case, each execution function has a unique semantic meaning corresponding to a particular use case.

4.1.2.1 Single-Cardinality Execution Functions

First we describe the general semantics of single-cardinality execution functions. For example, `.twoway_execute`:

```
template<class Function>
executor_future_t<Executor, std::invoke_result_t<std::decay_t<Function>>
twoway_execute(Function&& f) const;
```

In the descriptions that follow, let `Executor` be the type of `*this`; that is, the type of the executor. The only parameter of `.twoway_execute` is a callable object encapsulating the task of the created execution. The member function is `const` because executors act as shallow-`const` "views" of execution contexts. Creating execution does not mutate the view. Single-agent execution functions receive the callable as a forwarding reference. Single-agent, two-way customization points return the result of the callable object through a future as shown above. One-way customization points return `void`.

For `.then_execute`, the second parameter is a future which is the predecessor dependency for the execution:

```
template<class Function, class Future>
```

```
executor_future_t<Executor,std::invoke_result_t<std::decay_t<Function>,U&>>
then_execute(Function&& f, Future& predecessor_future) const;
```

Let `U` be the type of `Future`'s result object. The callable object `f` is invoked with a reference to the result object of the predecessor future (or without a parameter if a `void` future). By design, this is inconsistent with the interface of the Concurrency TS's `std::experimental::v1::future::then` which invokes its continuation with a copy of the predecessor future. Our design avoids the composability issues of `std::experimental::v1::future::then` [9] and is consistent with `.bulk_then_execute`, discussed below. Note that the type of `Future` is allowed to differ from `executor_future_t<Executor,U>`, enabling interoperability between executors and foreign or new future types.

Note that execution functions do not receive a parameter pack of arguments for `f`. This is a deliberate design intended to embue all customization point parameters with a semantic meaning which may be exploited by the executor. Generic parameters for `f` would have no special meaning to the execution function. We expect most clients to manipulate executors through higher-level control structures which are better positioned to provide conveniences like variadic parameter packing. Otherwise, a client may use `std::bind` if an appropriate control structure is unavailable.

4.1.2.2 Bulk-Cardinality Execution Functions

Bulk-cardinality execution functions create a group of execution agents as a unit, and each of these execution agents calls an individual invocation of the given callable function object. The forward progress ordering guarantees of these invocations are given by `std::execution::executor_bulk_forward_progress_guarantee_t`. Because they create multiple agents, bulk execution functions introduce ownership and lifetime issues avoided by single-cardinality customization points and they include additional parameters to address these issues. For example, consider `.bulk_twoway_execute`:

```
template<class Function, class Factory1, class Factory2>
executor_future_t<Executor,std::invoke_result_t<Factory1>>
bulk_twoway_execute(Function f, executor_shape_t<Executor> shape,
                    Factory1 result_factory, Factory2 shared_parameter_factory) const;
```

Bulk results. The first difference is that `.bulk_twoway_execute` returns the result of a **factory** rather than the result of `f`. Because bulk customization points create a group of execution agents which invoke multiple invocations of `f`, the result of execution is ambiguous. For example, all results of `f` could be collected into a container and returned, or a single individual result could be selected and returned. Our design requires the client to explicitly disambiguate the result via a factory. The `result_factory` is simply a callable object that is invoked before the group of execution agents begins invoking `f`, and the result of this factory is passed as a parameter to the invocations of `f`, which may arbitrarily mutate the result as a side effect. Any result of `f` itself is discarded.

Pass-by-value. Next, note that `f` is passed by value, rather than via forwarding reference. In general, it is impossible to elect a single agent to own `f` during execution because the group of agents may not be executing concurrently with each other or with the client. Instead, each agent owns a copy of `f`. One consequence of this policy is that move-only callables must be passed by a proxy such as `std::reference_wrapper`.

Shape. The first new parameter is `shape`, which describes the index space of the group of created execution agents. Each agent in the group is assigned a unique point in this index space and the agent receives it as a parameter to `f`. The type of this index is `executor_index_t<Executor>`. Currently, our proposal requires `executor_shape_t` (and hence `executor_index_t`) to be an integral type, but we envision generalizing this to support higher-dimensional index spaces.

Factories. The next two parameters are factories. The first is the `result_factory`, which we have already discussed. The second factory creates a shared parameter for `f`. Like the result, the shared parameter is constructed before the group of agents begins execution and it is passed as a parameter to `f`. Unlike the result, the shared parameter is discarded. Its purpose is to act as a temporary data structure shared by all execution agents during the computation. Examples are `std::barrier` or `std::atomic` objects. If the client desires to retain the shared parameter, it may be incorporated into the result during the execution of `f`.

The result and shared parameter are passed indirectly via factories instead of directly as objects because we believe this is the most general-purpose and efficient scheme to pass parameters to newly-created groups of execution agents [8]. First, factories allow non-movable types to be parameters, including concurrency primitives like `std::barrier` and `std::atomic`. Next, some important types are not efficient to copy, especially containers used as scratchpads. Finally, the location of results and shared parameters will be important to a parallel algorithm's efficiency. We envision associating allocators with individual factories to provide control[3].

The user function receives the shared state objects via bare references rather than an alternative channel such as `std::shared_ptr` because the lifetime of these shared objects is bound to the entire group of agents which share them. Because the sharing relationship is structured and identified beforehand, this enables optimizations that would be impossible for `shared_ptr`. For example, the way `shared_ptr` allows sharers to join and leave its group of sharers in an unstructured fashion necessitates dynamic storage and reference counting. By contrast, the structure enforced by bulk customization permits more efficient storage and sharing schemes.

Bulk continuations. Like `.then_execute`, `.bulk_then_execute` introduces a predecessor future upon which the bulk continuation depends:

```
template<class Function, class Future, class Factory1, class Factory2>
executor_future_t<Executor,std::invoke_result_t<Factory1>>
bulk_then_execute(Function f, executor_shape_t<Executor> shape,
                  Future& predecessor_future,
                  Factory1 result_factory, Factory2 shared_factory) const;
```

If the predecessor future's result object is not `void`, a reference to the predecessor object is passed to `f`'s invocation. Like the result and shared parameter, we pass the predecessor object by reference because no single agent in the group is its owner. The predecessor is collectively owned by the entire group of agents. As a consequence, `f` must carefully synchronize access to the predecessor object to avoid creating data races.

Parameter order. In any case, `f` is invoked with parameters provided in the same order as the corresponding parameters of the customization point. The agent index is always the first parameter, followed by the parameters emanating from `predecessor_future`, `result_factory`, and `shared_factory`.

4.2 Customization Points Adapt An Executor's Native Functionality

Our `std::async` implementation example did not interact with the incoming executor directly through a member function. Our design allows the user to interpose the `execution::require` and `execution::prefer` customization points between control structures and executors to create a uniform interface to target. Recall that we have identified a set of six execution functions and we expect that this set may grow in the future. Since it would be too burdensome for every type of executor to natively support this entire growing set of possible interactions, our design allows executors to select a subset for native support.

At the same time, for any given executor, control structures need access to the largest possible set of fundamental interactions. Control structures gain access to the entire set[4] of execution functions via adaptation. When an executor natively supports the execution function requested through `execution::require`, `execution::require` acts like the identity function and returns the executor unchanged. When the requested execution function is unavailable, the executor's native execution functions are adapted to fulfill the requested requirement.

[3]This envisioned allocator support is why we refer to these callable objects as "factories" rather than simply "functions" or "callable objects".

[4]In certain cases, some interactions are impossible because their requirements are inherently incompatible with a particular executor's provided functionality. For example, a requirement for never-blocking execution from an executor which always executes "inline".

As a simple example, consider a possible adaptation performed by `execution::require(ex, execution::always_blocking)` when `ex` does not natively guarantee always-blocking execution:

```
template<class Executor>
struct always_blocking_adaptor
{
  Executor wrapped;

  template<class Function>
  executor_future_t<Executor, std::invoke_result_t<std::decay_t<Function>>
  twoway_execute(Function&& f) const
  {
    // create twoway execution through the wrapped executor
    auto future = wrapped.twoway_execute(std::forward<Function>(f));

    // make the resulting future ready
    // note that this always blocks the caller
    future.wait();

    // return the future
    return future;
  }

  ...
};
```

In this case, `execution::require(ex, execution::always_blocking)` can return a copy of `ex` wrapped inside `always_blocking_adaptor` if `ex` does not natively provide always-blocking execution. Even if `ex` does not natively provide always-blocking execution, its client may use `ex` as if it does.

Property preservation. There are limits to the kinds of adaptations that `execution::require` and `execution::prefer` may apply, and these limits preserve executor properties. The rationale is that a customization point should not introduce surprising behavior when adapting an executor's native functionality. During adaptation, the basic rule we apply is that only the properties requested through a call to either `execution::require` or `execution::prefer` may be changed. The resulting executor must retain all the other properties of the original executor which were not named by the call.

5 Implementing Executors

A programmer implements an executor by defining a type which satisfies the requirements of the executor interface. The simplest possible example is an executor which always creates execution "inline":

```
struct inline_executor {
  bool operator==(const inline_executor&) const noexcept {
    return true;
  }

  bool operator!=(const inline_executor&) const noexcept {
    return false;
  }

  const inline_executor& context() const noexcept {
    return *this;
  }
  inline_executor require(execution::always_blocking) const noexcept
  {
    return *this;
  }

  template<class Function>
  void execute(Function&& f) const noexcept {
    std::forward<Function>(f)();
  }
};
```

First, all executor types must be `CopyConstructible`, which our `inline_executor` implicitly satisfies. Other requirements are satisfied by explicitly defining various member types and functions for introspection, property requests, and execution agent creation.

5.1 Introspection

Clients introspect executors at runtime through functions and at compile time through executor-specific type traits.

5.1.1 Functions

Executor identity. All executors are required to be `EqualityComparable` in order for clients to reason about their identity. If two executors are equivalent, then they may be used interchangably to produce the same side effects. For example, because `inline_executor::execute` simply invokes its function inline, all instances of `inline_executor` produce the same side effects and are therefore equivalent.

As another example, consider an executor type which creates execution agents by submitting to a thread pool. Suppose two executors of this type submit to the same underlying thread pool. These executors are equivalent because they both produce the same side effect of submitting to a common thread pool. However, if one of these executors were to change its underlying thread pool, they would become nonequivalent.

As a final example, consider a prioritizing executor type which submits work with an associated priority to a queue. The queue executes work in order of priority, and when two tasks have equivalent priority they are executed in non-deterministic order. Suppose two executors of this type submit to the same underlying queue, but with different priorities. These two executors are nonequivalent, because even though they both submit to a common underlying queue, they do so with different priority. Therefore, these executors produce different side effects and cannot be used interchangeably.

Execution context access. Next, all executors are required to provide access to their associated execution context via a member function named `.context`. The single type requirement for execution context types is `EqualityComparable`. However, we envision that these requirements will be refined in specializations as future proposals introduce additional requirements for their specific use cases. The `NetworkingExecutionContext` concept to be specified by the Networking TS already provides one example of refinement.

In non-generic programming contexts where the concrete types of executors and their associated contexts are known in advance, clients may use execution context identity to reason about underlying execution resources in order to make choices about where to create execution agents. In generic programming contexts such as templates the concrete type of execution context will not be known. However, the programmer may still manipulate execution contexts semi-generically through specializations which apply to concrete contexts.

Recall `inline_executor`. Because it is such a simple executor, it serves as its own execution context. Its `.context()` function simply returns a reference to itself. In general, more sophisticated executors will return some other object. Consider a `thread_pool_executor`:

```
class thread_pool_executor {
  private:
    mutable thread_pool& pool_;

  public:
    thread_pool_executor(thread_pool& pool) : pool_(pool) {}

    bool operator==(const thread_pool& other) const {
      return pool_ == other.pool_;
    }

    bool operator!=(const thread_pool& other) const {
      return pool_ != other.pool_;
    }

    const thread_pool& context() const {
      return pool_;
    }

    ...

    template<class Function>
    void execute(Function&& f) const {
      pool_.submit(std::forward<Function>(f));
    }
};
```

In this example, an executor which creates execution agents by submitting to a thread pool returns a reference to that thread pool from `.context`.

Our design allows programmers to reason about the identities of executors and execution contexts separately because the side effects they create may be distinct. For example, perfoming comparisons on `thread_pool_executor` objects yields no additional information than could be gained by comparing their execution contexts directly. The same is true for `inline_executor`. However, consider our prioritizing executor example whose execution context is a queue. When two prioritizing executors have different priorities, they are nonequivalent even if they both have equivalent execution contexts.

5.1.2 Type Traits

Executor-specific type traits advertise semantics of cross-cutting guarantees and also identify associated types. Executor type traits are provided in the `execution` namespace and are prefixed with `executor_`. Unless otherwise indicated, when an executor type does not proactively define a member type with the corresponding name (sans prefix), the value of these traits have a default. This default conveys semantics that make the fewest assumptions about the executor's behavior.

Execution context type. `executor_context` simply names the type of an executor's execution context by decaying the result of its member function `.context`. This default cannot be overriden by a member type because `.context`'s result is authoritative.

Associated `Future` type. `executor_future` names the type of an executor's associated future type, which is the type of object returned by asynchronous, two-way customization points. The type is determined by the result of `execution::async_execute`, which must satisfy the requirements of the `Future` concept[5].

[5]For now, the only type which satisfies `Future` is `std::experimental::future`, specified by the Concurrency TS. We expect the requirements of `Future` to be elaborated by a separate proposal.

Otherwise, the type is `std::future`. All of an executor's two-way asynchronous customization points must return the same type of future.

Executor shape type. When an executor creates a group of execution agents in bulk, the index space of those agents is described by a *shape*. Our current proposal is limited to one-dimensional shapes representable by an integral type, but we envision generalization to multiple dimensions. The type of an executor's shape is given by `executor_shape`, and its default value is `std::size_t`.

Executor index type. Execution agents within a group are uniquely identified within their group's index space by an *index*. In addition to sharing the dimensionality of the shape, these indices have a lexicographic ordering. Like `executor_shape`, the type of an executor's index is given by `executor_index`, and its default value is `std::size_t`.

5.2 Property Requests via `.require` and `.prefer`

Executors may optionally implement the member functions `.require` or `.prefer` to receive property requests from clients. In the case of our `inline_executor` example, `.require` can receive requests for always-blocking execution, which all `inline_executors` natively provide. The result of this function is simply a copy of the executor.

5.3 Execution Agent Creation via Execution Functions

Executors expose their native support for execution agent creation through **execution functions** which are executor member functions. In this section, we describe the suite of execution functions we have identified as key to the needs of the Standard Library and TSes we have chosen to target. In the discussion that follows, let `Executor` be the type of the executor whose execution function is being described.

5.3.1 Two-Way Bulk-Agent Functions

We begin by discussing the execution functions which create groups of execution agents in bulk, because the corresponding single-agent functions are each a functionally special case.

5.3.1.1 `bulk_then_execute`

```
template<class Future, class Function, class ResultFactory, class SharedFactory>
executor_future_t<Executor, std::invoke_result_t<std::decay_t<Function>,
                  decltype(std::declval<Future>().get())&>>
bulk_then_execute(Function&& func, Future& pred,
                  executor_shape_t<Executor> shape,
                  ResultFactory result_factory,
                  SharedFactory shared_factory) const;
```

`bulk_then_execute` creates a group of execution agents of shape `shape` and these agents begin execution after `pred` becomes ready. `bulk_then_execute` returns a future that can be used to wait for execution to complete, and this future contains the result of `result_factory`. Each created execution agent calls `std::forward<Function>(func)(i, r, s)`, where `i` is of type `executor_index_t<Executor>`, `r` is a function object returned from `return_factory` and `s` is a shared object returned from `shared_factory`.

`bulk_then_execute` is the most general execution function we have identified because it may be used to implement any other execution function without having to go out-of-band through channels not made explicit through the execution function's interface. Explicitly elaborating this information through the interface is critical because it enables the executor author to participate in optimizations which would not be possible had that information been discarded through backchannels.

For example, suppose the only available execution function was `bulk_twoway_execute`. It would be possible to implement `.bulk_then_execute`'s functionality by making a call to `bulk_twoway_execute` inside a continuation created by `predecessor_future.then`:

```
predecessor_future.then([=] {
  return exec.bulk_twoway_execute(exec, f, shape, result_factory, shared_factory).get();
});
```

Note that this implementation creates `1 + shape` execution agents: one agent created by `then` along with `shape` agents created by `bulk_twoway_execute`. Depending on the relative cost of agents created by `then` and `bulk_twoway_execute`, the overhead of introducing that extra agent may be significant. Moreover, because the `then` operation occurs separately from `bulk_twoway_execute`, the continuation is invisible to `exec` and this precludes `exec`'s participation in scheduling. Because we wish to allow executors to abstract sophisticated task-scheduling runtimes, this shortcoming is unacceptable.

5.3.1.2 `bulk_twoway_execute`

```
template<lass Function, class ResultFactory,
         class SharedFactory>
executor_future_t<std::invoke_result_t<std::decay_t<ResultFactory>>>
bulk_twoway_execute(Function&& func,
                    executor_shape_t<Executor> shape,
                    ResultFactory result_factory,
                    SharedFactory shared_factory) const;
```

`bulk_twoway_execute` creates a group of execution agents of shape `shape` and returns a future that can be used to wait for execution to complete. This future contains the result of `result_factory`. Each created execution agent calls `std::forward<Function>(func)(i, r, s)`, where `i` is of type `executor_index_t<Executor>`, `r` is a function object returned from `return_factory` and `s` is a shared object returned from `shared_factory`.

Like `bulk_then_execute`, `bulk_twoway_execute` returns a future corresponding to the result of the asynchronous group of execution agents it creates. Due to these similarities, `bulk_twoway_execute` is functionally equivalent to calling `bulk_then_execute` with a ready `void` future:

```
future<void> no_predecessor = make_ready_future();
exec.bulk_then_execute(func, shape, no_predecessor, result_factory,
                       shared_factory);
```

We include `bulk_twoway_execute` because the equivalent path through `bulk_then_execute` via a ready future may incur overhead. The cost of the future itself may be significant, especially if any sort of dynamically-allocated asynchronous state is associated with that future. Alternatively, the act of scheduling itself may be a source of overhead, especially if it requires any sort of graph analysis performed by a dynamic runtime. Providing executors the opportunity to specialize for cases where it is known at compile time that no dependency exists avoids both hazards.

Moreover, common types of executor may not naturally create execution in terms of continuations on futures as expected by `bulk_then_execute`. `bulk_async_execute` is a better match for these cases because it does not require accommodating a predecessor dependency.

5.3.2 Two-Way Single-Agent Functions

Conceptually, single-agent execution functions are special cases of their bulk counterparts. However, we expect single-agent creation to be an important special case; in fact, many existing applications employ executors solely for single-agent execution. Explicit support for single-agent submission allows executor implementations to optimize for this important use case at compile time.

In particular, an important use for single-agent execution functions is the ability to submit move-only function objects. Move-only function objects allow us to submit tasks that contain relatively heavyweight, move-only resources, such as files, sockets or `std::promise` objects. It is true that bulk execution functions can support move-only function objects, but only if additional care is taken to ensure the function object remains valid until all execution agents complete. This would probably be achieved by placing the function object in reference counted storage, and represents a significant overhead that is not required in the single-agent case.

Alternatively, a bulk execution function could optimise for the single agent case by performing a runtime test for a `shape` of `1`: when this condition is detected, a non-reference-counted implementation would be selected. However, for the many applications that desire efficient single-agent execution, a bulk-only interface results in additional complexity and code (bloat). Furthermore, if custom executors are developed for these applications, a bulk-only approach means that development effort must be expended on the multi-agent case even if that runtime branch is never used. And, finally, we lose the ability to perform a compile time test to determine whether an executor "natively" supports single-agent execution (e.g. the `has_executor_member` trait included in this proposal).

5.3.2.1 `then_execute`

```
template<class Function, class Future>
executor_future_t<Executor, std::invoke_result_t<std::decay_t<Function>,
                  decltype(std::declval<Future>().get())&>>
then_execute(Function&& func, Future& pred) const;
```

`then_execute` creates a single execution agent and this agent begins execution after `pred` becomes ready. `then_execute` returns a future that can be used to wait for execution to complete, and this future contains the result of `func`. The created execution agent calls `std::forward<Function>(func)()`.

`then_execute` may be implemented by using `bulk_then_execute` to create a group with a single agent:

```
using result_t = std::invoke_result_t<Function>;
using predecessor_t = decltype(predecessor_future.get());

// create a factory to return an object of the appropriate type
// XXX instead of default construction, this really needs to return some sort
//     of storage for an unintialized result and then the lambda below would
//     placement new it
auto result_factory = []{ return result_t(); };

// pass func as a shared parameter to account for move-only functions
auto shared_factory = [func = std::forward<Function>(func)]{ return func; };

// create a lambda for the "group" of agents to invoke
auto g = [](executor_index_t<Executor> ignored_index,
            predecessor_t& predecessor,
            result_t& result,
            Function& func) {
  // invoke func with the predecessor as an argument and assign the result
  result = func(predecessor);
};

return exec.bulk_then_execute(g,
  executor_shape_t<Executor>{1}, // create a single agent group
  pred,
  result_factory,
  shared_factory
);
```

The sample implementation passes both the function to invoke and its result indirectly via factories. The result of these factories are shared across the group of agents created by `bulk_then_execute`. However, this group has only one agent and no sharing actually occurs. The cost of this unnecessary sharing may be significant and can be avoided if an executor natively provides `then_execute`.

5.3.2.2 `twoway_execute`

```
template<class Function>
executor_future_t<Executor, std::invoke_result_t<std::decay_t<Function>>>
twoway_execute(Function&& func) const;
```

`twoway_execute` creates a single execution agent and returns a future that can be used to wait for execution to complete. This future contains the result of `func`. The created execution agent calls `std::forward<Function>(func)()`.

`twoway_execute` may be implemented by using `then_execute` with a ready `void` future:

```
std::experimental::future<void> ready_future = std::experimental::make_ready_future();
return exec.then_execute(std::forward<Function>(f), ready_future);
```

Alternatively, `bulk_twoway_execute` could be used, analogously to the use of `bulk_then_execute` in the example implementation of `then_execute`.

The cost of a superfluous immediately-ready future object could be significant compared to the cost of agent creation. For example, the future object's implementation could contain data structures required for inter-thread synchronization. In this case, these data structures are wasteful and never need to be used because the future is ready-made.

On the other hand, once a suitable `Future` concept allows for user-definable futures, the initial future need not be `std::experimental::future`. Instead, a hypothetical `always_ready_future` could be an efficient substitute as it would not require addressing the problem of synchronization:

```
always_ready_future<void> ready_future;
return exec.then_execute(std::forward<Function>(f), ready_future);
```

However, to fully exploit such efficiency, `then_execute` may need to recognize this case and take special action for `always_ready_future`.

Because of the opportunity for efficient specialization of a common use case, and to avoid requiring executors to explicitly support continuations with `then_execute`, including `twoway_execute` as an execution function is worthwhile.

5.3.3 One-Way Bulk-Agent Functions

5.3.3.1 `bulk_execute`

```
template<class Function, class SharedFactory>
void bulk_execute(const Executor& exec, Function&& func,
                  executor_shape_t<Executor> shape,
                  SharedFactory shared_factory) const;
```

`bulk_execute` creates a group of execution agents of shape `shape` and does not return a result. Each created execution agent calls `std::forward<Function>(func)(i, s)`, where `i` is of type `executor_index_t<Executor>` and `s` is a shared object returned from `shared_factory`.

`bulk_execute` is equivalent to `execute` except that it creates a single execution agent rather than a group of execution agents. Consider an example implementation:

```
// create shared object
auto shared = shared_factory;

// Iterate over the shape
for (int i = 0; i < shape; i++) {

    // construct index
    executor_shape_t<Executor> index{i};

    // create a lambda for the function for `execute`
    auto g = [=, &shared](Function& func) {
      func(index, shared);
    };

    exec.execute(g);
}
```

We include `bulk_execute` because the equivalent path through `execute` via a for loop at the point of submission would incur overhead and would not be able to guarantee correct forward progress guarantees between each execution agent created by `execute`.

5.3.4 One-Way Single-Agent Functions

5.3.4.1 `execute`

```
template<class Function>
void execute(Function&& func) const;
```

`execute` asynchronously creates a single execution agent and does not return a result. The created execution agent calls `std::forward<Function>(func)()`.

`execute` is equivalent to `bulk_execute` except that it creates a group of execution agents rather than a single execution agent. This means that an executor that provides `bulk_execute` could be adapted to provide the semantic guarantees of `execute` if it were absent, by creating a group with a single execution agent. Consider an example implementation:

```
// pass func as a shared parameter to account for move-only functions
auto shared_factory = [func = std::forward<Function>(f)]{ return func; };

// create a lambda for the "group" of agents to invoke
auto g = [](executor_index_t<Executor> ignored_index, Function& func) {
  result = func();
};

exec.bulk_then_execute(g,
  executor_shape_t<Executor>{1}, // create a single agent group
  shared_factory
);
```

As with our example implementation of `then_execute` described earlier, this group has only one agent and no sharing actually occurs. The cost of this unnecessary sharing may be significant and can be avoided by an executor providing `execute`.

6 Future Work

We conclude with a survey of future work. Some of this work is in scope for P0443 and should be done before the design is considered complete. Other work is explicitly out of scope, and should be pursued independently using our design as a foundation.

6.1 Open Design Issues

Much of our design for executors is well-established. However, some aspects of the design remain the subject of ongoing discussion.

Relationship with Thread Local Storage. By design, our executors model provides no explicit support for creating thread-local storage. Instead, our design provides programmers with tools to reason about the relationship of programmer-defined `thread_local` variables and execution agents created by executors. For example, the executor properties `thread_execution_mapping` and `new_thread_execution_mapping` describe how execution agents are mapped onto threads, and consequently how the lifetimes of those agents relate to the lifetimes of `thread_local` variables. It is unclear whether these tools are sufficient or if more fine-grained control over thread local storage is warranted.

Forward Progress Guarantees and Boost Blocking. Our executor programming model prescribes a way for bulk executors to advertise the forward progress guarantees of execution agents created in bulk. This guarantee describes an agent's forward progress with respect to other agents within the same group as that agent. However, our model prescribes no analogous way for advertising any guarantee of forward progress between a single execution agent and the client thread which requested the creation of that agent. Similarly, our programming model does not describe how executors would make such guarantees. Incorporating a model of *boost blocking* into our design could be one option.

6.2 Envisioned Extensions

we conclude with a brief survey of future work extending our proposal. Some of this work has already begun and there are others which we believe ought to be investigated.

`Future` Concept. Our proposal depends upon the ability of executors to create future objects whose types differ from `std::future`. Such user-defined `std::future`-like objects will allow interoperation with resources whose asynchronous execution is undesirable or impossible to track through standard means. For example, scheduling runtimes maintain internal data structures to track the dependency relationships between different tasks. The reification of these data structures can be achieved much more efficiently than by pairing a `std::promise` with a `std::future`. As another example, some "inline" executors will create execution immediately in their calling thread. Because no interthread communication is necessary, inline executors' asynchronous results do not require expensive dynamic allocation or synchronization primitives of full-fledged `std::future` objects. We envision that a separate effort will propose a `Future` concept which would introduce requirements for these user-defined `std::future`-like types.

Thread Pool Variations. Our proposal specifies a single thread pool type, `static_thread_pool`, which represents a simple thread pool which assumes that the creator knows the correct thread count for the use case. As a result, it assumes a pre-determined sizing and does not automatically resize itself and has no default size.

There exist heuristics for right-sizing a thread pool (both statically determined like `2*std::thread::hardware-_concurrency()`, as well as dynamically adjusted), but these are considered to be out of scope of this proposal as a reasonable size pool is specific to the application and hardware.

We recognize that alternative approaches serving other use cases exist and anticipate additional thread pool proposals. In particular, we are aware of a separate effort which will propose an additional thread pool type, `dynamic_thread_pool`, and we expect this type of thread pool to be both dynamically and automatically resizable.

Execution Resources. Our executors model describes execution agents as bound to *execution resources*, which we imagine as the physical hardware and software facilities upon which execution happens. However, our design does not incorporate a programming model for execution resources. We expect that future proposals will extend our work by describing a programming model for programming tasks such as enumerating the resources of a system and querying the underlying resources of a particular execution context.

Heterogeneity. Contemporary execution resources are heterogeneous. CPU cores, lightweight CPU cores, SIMD units, GPU cores, operating system runtimes, embedded runtimes, and database runtimes are examples. Heterogeneity of resources often manifests in non-standard C++ programming models as programmer-visible versioned functions and disjoint memory spaces. Therefore, the ability for standard executors to target heterogeneous execution resources depends on a standard treatment of heterogeneity in general.

The issues raised by heterogeneity impact the entire scope of a heterogeneous C++ program, not just the space spanned by executors. Therefore, a comprehensive solution to these issues requires a holistic approach. Moreover, the relationship between heterogeneous execution and executors may require technology that is out of scope of a library-only solution such as our executors model. This technology might include heterogeneous compilation and linking, just-in-time compilation, reflection, serialization, and others. A separate effort should characterize the programming problems posed by heterogeneity and suggest solutions.

Bulk Execution Extensions. Our current proposal's model of bulk execution is flat and one-dimensional. Each bulk execution function creates a single group of execution agents, and the indices of those agents are integers. We envision extending this simple model to allow executors to organize agents into hierarchical groups and assign them multidimensional indices. Because multidimensional indices are relevant to many high-performance computing domains, some types of execution resources natively generate them. Moreover, hierarchical organizations of agents naturally model the kinds of execution created by multicore CPUs, GPUs, and collections of these.

The organization of such a hierarchy would induce groups of groups (of groups..., etc.) of execution agents and would introduce a different piece of shared state for each non-terminal node of this hierarchy. The interface to such an execution function would look like:

```
template<class Function, class ResultFactory, class... SharedFactories>
execution::executor_future_t<Executor, std::invoke_result_t<ResultFactory()>>
bulk_twoway_execute(Function f, executor_shape_t<Executor> shape,
                    ResultFactory result_factory, SharedFactories... shared_factories);
```

In this interface, the `shape` parameter simultaneously describes the hierarchy of groups created by this execution function as well as the multidimensional shape of each of these groups. Instead of receiving a single factory to create the shared state for a single group, the interface receives a different factory for each level of the hierarchy. Each group's shared parameter originates from the corresponding factory in this variadic list.

Transactional Memory. SG5 Transactional Memory is studying how proposed TM constructs in the Transactional Memory TS can be integrated with executors. As TM constructs are compound statements of the form `atomic_noexcept | atomic_commit | atomic_cancel {<compound-statement> }` and `synchronized {<compound-statement> }`, it seems they can also apply with executors.

7 Acknowledgements

We appreciate the feedback on this document provided by Billy O'Neal and Torvald Riegel.

References

[1] Austern, M., Lawrence, C., Carruth, C., Gustafsson, N., Mysen, C. and Yasskin, J. 2013. Executors and schedulers, revision 1. (Mar. 2013).

[2] Austern, M., Lawrence, C., Carruth, C., Mysen, C. and Yasskin, J. 2012. A preliminary proposal for work executors. (Feb. 2012).

[3] Hoberock, J., Garland, M. and Giroux, O. 2015. An Interface for Abstracting Execution. (Sep. 2015).

[4] Hoberock, J., Garland, M. and Giroux, O. 2015. Parallel Algorithms Need Executors. (Apr. 2015).

[5] Hoberock, J., Garland, M., Giroux, O. and Kaiser, H. 2016. An Interface for Abstracting Execution. (Feb. 2016).

[6] Hoberock, J., Garland, M., Kohlhoff, C., Mysen, C. and Edwards, C. 2016. A Unified Executors Proposal for C++. (Oct. 2016).

[7] Hoberock, J., Garland, M., Kohlhoff, C., Mysen, C., Edwards, C. and Brown, G. 2017. A Unified Executors Proposal for C++. (Jan. 2017).

[8] Kaiser, H. and Hoberock, J. 2016. Execution interfaces should be variadic.

[9] Kaiser, H. and Hoberock, J. 2016. then_execute differs from Concurrency TS V1 semantics.

[10] Kohlhoff, C. 2014. Executors and Asynchronous Operations. (May 2014).

[11] Mysen, C. 2015. C++ Executors. (Sep. 2015).

[12] Mysen, C. 2015. Executors and schedulers, revision 5. (Apr. 2015).

[13] Mysen, C. and Gustafsson, N. 2013. Executors and schedulers, revision 2. (Aug. 2013).

[14] Mysen, C., Gustafsson, N., Austern, M. and Yasskin, J. 2013. Executors and schedulers, revision 3. (Oct. 2013).