Light-Weight Execution Agents

Revision 4

Document number:	N4439
Revises:	N4156 (see Section 7 for revision history)
Date:	2015-04-10

Author:	Torvald Riegel
Reply-to:	Torvald Riegel <triegel@redhat.com>

1 Introduction

N4016 makes a case for adding execution modes for threads of execution that are less capable than `std::thread`; doing so either allows for avoiding runtime overheads of `std::thread` or enables semantically different ways to execute such as SIMD loops.

This paper defines the semantics of these lighter-weight modes of execution by defining certain kinds of *execution agents* (EAs). An EA is a mechanism that executes a particular *thread of execution*[1].

Execution context was suggested as a different name instead of EA; this might make it clearer that the intent is to model execution properties and not primarily to describe the hardware or software resources used for execution. To provide some consistency in the revisions of this paper, I will stick to using the term EA and will happily delegate the choice of a final name to the committee.

The standard seems not quite clear regarding whether a thread of execution is a static (e.g., a function) or dynamic entity (i.e., an instance of an execution of single flow of control); in this paper, I will assume the latter. Given that, there is a one-to-one correspondence between an EA and a thread of execution. `std::thread` is a simple way to create and manage a certain type of EA. The entity that is a running thread (i.e., what was created by `std::thread`) is an EA. See N4321 for a more detailed explanation of this terminology (and of the existing terminology in the standard).

The standard also seems to not explicitly state what execution agent executes `main`. I will assume that it executes on an execution agent that is created implicitly by the implementation and has equivalent semantics to execution agents created by `std::thread`.

[1] A thread of execution is defined in the standard as "a single flow of control within a program" (see §1.10p1). *sequenced-before* is defined exactly for a thread of execution (see §1.9p13).

At the Rapperswil meeting, there was very strong consensus that forward progress requirements are essential to specifying EAs. Therefore, I will first present refined progress requirements (Section 2), then discusses `std::thread`-specific features such as thread-local storage (Section 3), sketch how to represent EA requirements in the type system (Section 4), and propose wording for version 2 of the Parallelism TS (Section 5).

2 Forward progress requirements

The requirements described in what follows are requirements for an implementation of the particular kinds of execution agents; in turn, programmers can treat them as the guarantees the implementation gives to the EAs they use in a program.

The requirements are deliberately specified on an abstract level of "making progress" and *not* by explaining them based on current implementation techniques such as mappings to OS threads. This helps separate implementation artifacts from the requirements we really need to specify in a standard, and thus should help avoid misunderstandings; for example, OS threads on a host CPU are often quite different from threads running on a GPU. There certainly is a performance aspect of EAs, but this should be separate from the semantics I am focusing on here (e.g., exposed through Executors).

2.1 Basic forward progress definitions

Currently, the standard requires that "implementations should ensure that all unblocked threads eventually make progress" (§1.10p2). While "progress" is not explicitly defined, it likely means events of observable behavior of the abstract machine, or termination of the abstract machine. Such behavior will be produced by the implementation taking steps that eventually lead to observable behavior (and thus correspond to steps of execution of the abstract machine). The term "unblocked" is also not defined, and is the trickier issue because there are different causes that prevent progress: (1) no execution of implementation steps vs. (2) a thread that is blocked due to program logic. The latter can happen both when the implementation does not execute steps (e.g., when a blocking file I/O function has been called, or a lock uses a blocking facility of the OS) and when it does execute steps (e.g., busy-waiting using atomics until another thread produces a value).

I propose to replace this with the following requirements. First, we define what "progress" of a thread of execution is:

> The execution of threads of execution consists of *steps*: A step ends with either (1) termination of the thread of execution, (2) access to or modification of a volatile object, or (3) a synchronization or atomic operation; the remainder of a step consists of other things executed by the implementation that are not in the previous list.
>
> Note: A thread of execution makes progress when its steps are executed.

These steps could also be summarized as observable actions in the sense of being either observable by other threads or observable in terms of observable behavior of the abstract machine.

Next, we need to cover calls to library I/O functions, which may block in an OS kernel when waiting for data to arrive, for example. This blocking is internal to the application, and as I mentioned previously, we need to distinguish between blocking due to, for example, lack of input data and the implementation not allowing the thread of execution to make progress. To solve this, we can model such blocking functions as *conceptual* busy-waiting loops:

> Calls to library I/O functions and other functions that block (i.e., that cannot return to the caller unless a certain, potentially external condition is met) can be conceptually expressed using steps as defined previously; each blocking operation can be replaced by a busy-waiting loop that polls for the condition to be met (e.g., using observable behavior of the abstract machine).

Note that this is purely a model to enable the specification of progress; implementations are not required to busy-wait nor required to not do that. It allows us to separate the progress allowed by the implementation (i.e., whether a thread of execution is allowed to make steps) from whether particular blocking operations such as I/O make progress in the sense of achieving what the program was meant to do. The implementation has control over the former, but not necessarily over the latter.

If blocking operations want to specify which condition they may wait for, then they can express this with busy-waiting loops.[2] The progress requirements defined below do *not* require such specifications but only rely on the fact that such a specification would be possible. Therefore, we do not need to add these specifications to the standard before being able to define progress as described here.

2.2 Execution agent progress requirements

Next, I will discuss three classes of forward progress requirements that EAs can fall into.

Concurrent execution This class provides the same progress guarantees as `std::thread`: EAs in this class will eventually be allowed by the scheduler to make all steps they need to execute, independently of what other EAs (including those created by `std::thread`) are doing.

Two EAs in this class will run concurrently with respect to each other, and can depend on the other EA to make implementation steps concurrently, even if they block on the other EA:

> The implementation should ensure that a concurrent execution agent will eventually be allowed to execute all steps its thread of execution consists

[2] For example, a non-lock-free atomic operation can be expressed as performing the (sequential code for the) operation in a critical section protected by a global lock; in turn, lock acquisition can be expressed as a spinlock.

of, independently of which steps other execution agents might or might not execute.

The execution agents created by `std::thread` are concurrent execution agents.

Note: To eventually fulfill this requirement means that this will happen in an unspecified but finite amount of time.

Thus, this can effectively replace the existing progress requirement (§1.10p2); it clarifies that the requirement is about providing each concurrent execution agent (e.g., one created using `std::thread`) with the compute resources it needs. In other words, what the implementation should provide for `std::thread` is a thread scheduler that never starves any thread, and is thus fair (at least to some degree).

The progress requirement is specified as "should" instead of "shall" (as in §1.10) because there might be implementations based on OS schedulers that cannot give these properties (e.g., in a hard-real-time environment). Nonetheless, general-purpose implementations should strive to fulfill this requirement.

Parallel execution Parallel execution is weaker than concurrent execution in terms of forward progress. In particular, we need to capture the notion that one would like to let programs define lots of parallel tasks, yet use a bounded set of resources (e.g., CPU cores) to execute those tasks:

A parallel EA cannot be expected to be allowed by the implementation to execute steps if it has not yet executed any step; once it has, the implementation should ensure that it will eventually be allowed to execute all steps its thread of execution consists of, independently of which steps other execution agents might or might not execute.

Note: This effectively makes the same progress requirements as for a concurrent execution agent once the parallel execution agent is started, but does not specify a requirement for when to start the parallel execution agent; the latter will typically be specified by the entity that creates the parallel execution agent.

In other words, a parallel EA will behave like a `std::thread` in terms of progress, but only after performing its first step. Note that though having no requirement on when to start a parallel EA might seem odd at first, this is completely taken care of by the notion of boost-blocking (see Section 2.3).

This progress requirement is stronger than for other forms of parallel execution (see below); the main advantage of it is that it allows typical uses of critical sections because as soon as a parallel EA starts and might acquire a mutex, it is guaranteed to eventually be able to finish execution, and thus it will not block other EAs indefinitely.[3]

[3]This still does not allow other uses of mutexes, for example an EA using a mutex to wait for the finished execution of another EA that already owned the mutex before it got started.

Weakly parallel execution Weakly parallel EAs cannot be expected to make steps concurrently with other EAs. Thus, they get weaker progress guarantees than parallel EAs in that there is no upgrade to concurrent execution once a weakly parallel EA has started executing:

> The implementation does not need to ensure that a weakly parallel EA is allowed to execute steps independently of which steps other EAs might or might not execute.

As a result, unlike for parallel EAs, typical uses of critical sections are not possible in weakly parallel EAs because then an EA might wait for another EA that is not guaranteed to be able to make implementation steps concurrently.[4] However, using nonblocking synchronization (e.g., using lock-free atomics) is possible because it does not need any specific guarantees from the scheduler (i.e., it just finishes a step).[5]

Also note that like for parallel EAs, boost-blocking takes care of what might appear to be a too weak progress requirement (see Section 2.3).

Implementation examples There are many possible implementations that would satisfy the progress requirements for the three classes of EAs; I will highlight just a few to illustrate the possibilities.

Concurrent EAs can be easily implemented by mapping each EA to one OS thread and using a round-robin OS thread scheduler. An unbounded thread pool that eventually adds a new OS thread to the pool if some EAs did not run yet is also a valid implementation. However, a bounded thread pool is not a correct implementation because it does not guarantee concurrent execution if there are more concurrent EAs than OS threads in the pool (e.g., the pool's threads might all be taken by consumers waiting for a producer that hasn't been started because the pool's threads are all being used).

Nonetheless, if an implementation that uses a non-preemptive thread scheduler is aware of all points of blocking—more precisely, points where one EA depends on another EA's progress—and it either preempts or starts new threads in case of blocking, then this can be a valid implementation of concurrent EAs. However, this can be nontrivial to implement; the, for example, compiler could have to instrument all loops containing atomic loads or read-modify-write ops and volatile loads (because these end steps), as well as calls to (OS) library functions that might block.

Parallel EAs, on the other hand, can be implemented based on a bounded thread pool; because the progress requirement does not require a certain amount of resources (e.g., OS threads) to be used, the implementation has full flexibility regarding resource usage. This is easy to implement because we just need to execute all parallel EAs eventually,

[4] Specifically, consider cases like when parallel EAs use the same mutex to protect critical sections. Other cases would still be allowed, such as when an EA uses a critical section that it will never block on (e.g., because nobody else uses the same mutex).

[5] Note that while the success of obstruction-free synchronization depends on whether operations are scheduled in such a way that there is no obstruction eventually, this can be emulated by the code executing the obstruction-free operations (e.g., using randomized exponential back-off). Blocking synchronization is different in that it cannot tolerate another EA not making any progress.

in some order and interleaving. However, it cannot be implemented with, for example, certain kinds of work-stealing schedulers if work-stealing is allowed to happen during critical sections.[6]

Weakly parallel EAs can be implemented in several ways. They can be run concurrently on threads from a thread pool, or in some interleaving on a single thread. Even implementations that use SIMD instructions fall into the latter category (i.e., using a mix of SIMD-parallel execution and serialized execution of code that cannot be vectorized). Work-stealing implementations are also able to provide weakly parallel EAs.

The progress requirements for parallel and weakly parallel EAs match the progress guarantees that are essentially in effect when using `parallel_execution_policy` and `parallel_vector_execution_policy`, respectively, of the Parallelism TS (N4104).

2.3 Boosting progress

While concurrent EAs are guaranteed to just execute eventually, the requirements for parallel and weakly parallel EAs do not include having to actually start execution of such EAs.

As an example, consider a simple program that contains a parallel loop; the calling EA of such a loop spawns a number of parallel EAs and waits until all of them have finished. The whole program starts executing as one `std::thread`, so the EA that calls the parallel loop construct is a concurrent EA. Thus, the program is already guaranteed to execute eventually up to the invocation of the parallel loop. However, then we have a bootstrap problem because the concurrent EA waits for the completion of parallel EAs, which have weaker progress requirements.

What we need is a way to specify that the stronger progress requirements for the concurrent EA should eventually lead to finishing execution of the parallel EAs spawned by the parallel loop. To do that, let us define a notion of *boost-blocking*:

> If an execution agent P uses boost-blocking to block on the completion of a set S of execution agents, and if P is subject to a stronger forward progress requirement than at least one execution agent in S, then throughout the whole time of P being boost-blocked on S, P will boost the progress requirement of at least one execution agent in S to Ps stronger requirement. Specifically, P is free to select which execution agent in S to boost and for which amount of time (i.e., the boost is not permament and in place for the rest of the lifetime of the boosted execution agent); as long as P is boost-blocked, it has to eventually boost an execution agent in S. Once an execution agent in S finishes execution, it is removed from S. Once S is empty, P stops being blocked.

[6] Consider a scheduler that immediately executes a spawned parallel task instead of finishing the spawning task (and keeps using a single OS thread): If the former blocks on a mutex acquired by the latter, then the OS thread used for the two EAs will get deadlocked; if the scheduler isn't aware of all blocking relationships nor promotes parallel EAs to concurrent EAs after a while (i.e., adds OS threads to the pool), a deadlock will arise.

Note: An execution agent thus can have an effectively stronger forward progress requirement for a certain amount of time, due to second agent being boost-blocked on it. In turn, this may allow this agent to itself boost the progress of a third agent that this agent is boost-blocked on.

Note: If all execution agents in S finish execution (e.g., they do not use blocking synchronization incorrectly), then Ps progress requirement will not be weakened by executing the boost-blocking operation.

Note: This does not remove any constraints regarding blocking synchronization for parallel or weakly parallel execution agents because P is not guaranteed to boost the particular execution agent whose too-weak progress requirement is preventing overall progress.

In the parallel loop example, this means that the calling EA's progress just helps one of the parallel EAs to make progress (e.g., to get a parallel EA started). If all of them terminate, they will eventually all be allowed by the implementation to execute all their steps. Note that, however, this does not mean that the concurrent EA will make all the parallel EAs start concurrently; the concurrent EA is free to boost just one parallel EA until this EA has terminated, and then boost the next one. This way, if none of the parallel EAs blocks for another parallel EA to be started, then the whole loop is guaranteed to be executed eventually.

This is straight-forward to implement when concurrent EAs are based on OS threads: Ahe calling EA just participates in executing iterations of the loop that are not processed yet; any OS threads available in a program-wide thread pool can be used to execute other iterations of the loop.

When a group of weakly parallel EAs are boosted by a concurrent or parallel EA, this works similarly to the previous example in that one of the weakly parallel EAs will always eventually make steps. This allows vectorized execution, for example: The concurrent or parallel EA can use SIMD operations to execute steps from all weakly parallel EAs in a lockstep execution fashion.

Boost-blocking is a property that operations such as the parallel loop from our example would explicitly guarantee, including specifying the group of EAs that the boost-blocking applies to. In the Parallelism TS, for example, all algorithms executed with a parallel or parallel-vector execution policy would guarantee boost-blocking for the group of parallel or weakly-parallel EAs that an algorithm spawns. Boost-blocking is not intended to be the default form of blocking; for example, `std::mutex` would not guarantee boost-blocking as this would increase the implementation complexity and performance cost.

Examples Let us look at two more complex examples. First, if we have two concurrent EAs that both execute one parallel loop spawning parallel EAs, then each concurrent EA boost-blocks on *its own* group of parallel EAs. If one of the parallel EAs of the first concurrent EA produces an intermediate value that a parallel EA of the second concurrent EA needs, then the second parallel EA can block on the first EA without needing boost-blocking to ensure progress (e.g., it can use a condition variable). This

works because boost-blocking reduces this case to essentially the second concurrent EA waiting for the first concurrent EA.

As a second example, consider an implementation that schedules several threads of execution nonpreemptively on fewer OS threads than there are threads of execution. This means having weakly parallel EAs because one of the EAs cannot expect to make steps without another EA potentially having to preempt voluntarily (i.e., making the OS thread it runs on available). We can envision a facility that spawns an entire network of such EAs that do message passing between each other (e.g., each EA represents one actor in an actor-based model). After being spawned, the network may make progress but this is not guaranteed because these are all weakly parallel EAs. To get an actual output from the network, we can use boost-blocking in two ways:

- Let a concurrent or started parallel EA boost-block on the whole network (i.e., the group of all weakly parallel EAs it is comprised of).
- If the network uses pull-based communication, let a concurrent or started parallel EA boost-block on exactly the EAs that produce the final output of the network, *and* employ pull or receive–message operations that use boost-blocking. This will establish transitive boost-blocking towards all the EAs whose input is needed to produce the output.

3 `std::thread`-specific state and features

Besides forward progress, we also need to consider how light-weight EAs relate to features of EAs created by `std::thread`, notably thread-local storage (TLS) and the value returned by `this_thread::get_id`. While programmers often will not need these features, we need to at least define the level of compatibility with existing code based on `std::thread`.

There are several choices for whether, and how, an EA can relate to these features. We can express these by specifying to which extent a particular EA can be conceptually considered to be running on top of another EA created by `std::thread` (in what follows, this is abbreviated as "being associated with a `std::thread`"). Ranging from the weakest requirement to the strongest, these are:

(1) Not associated with `std::thread`. In an EA for which only this requirement applies, accesses to TLS or calls to `this_thread::get_id` result in undefined behavior.

(2) Always associated with some `std::thread`. There is an associated `std::thread`, but which instance this is can change at any time during the execution of the EA; however, there will be no change during a *single* access to TLS, a call to `this_thread::get_id`, or the execution of external, non-C++ code.[7] This requires

[7] We could also say that changes can happen in all `std::` code and certain language constructs; however, this might not make it easier to use because, for example, calls to customized operators can be easy to overlook.

TLS to be used as if it where relaxed atomic accesses with respect to the change of the associated thread; it would be safe to make a plain load or store to TLS, but trying to apply a plain read–modify–write operation (e.g., `threadlocal++;`) is not guaranteed to work.

(3) Association with `std::thread` only changes in certain cases. Compared to (2), this restricts the places where the associated `std::thread` can change to explicitly specified cases, for example calls to certain `std::` functions or execution of language constructs. Depending on which these cases are, only a few or many `std::` functions might have to be included in the set (e.g., when the associated `std::thread` can change in every parallelism construct, then every potentially parallelized `std::` function is in the set).

(4) Associated `std::thread` is stable. The associated `std::thread` will not change during the lifetime of the EA.

(5) Associated `std::thread` is stable and has same lifetime as EA. This is exactly what code using `std::thread` would get. The difference to (4) is that constructors and destructors of TLS are guaranteed to be executed at the start and termination of the EA.

Additionally, we can distinguish whether TLS is potentially shared between several EAs, which can be useful if TLS is used as a mechanism to maintain likely-local state (in the sense of likely being accessed by just a few EAs), for example for caching. Thus, if we consider the above requirements to include that any associated `std::thread` is only associated with one EA at a particular point in time, then we can add two additional requirements:

(3s) Association with `std::thread` only changes in certain cases; shared. Like (3) except that an associated `std::thread` can be associated with other EAs concurrently.

(4s) Associated `std::thread` is stable; shared. Like (4) except that an associated `std::thread` can be associated with other EAs concurrently.

Note that for (5), a shared TLS implementation is probably not useful because it would require the affected EAs to have the exact same lifetimes. For (2), a shared TLS implementation does not make much difference in practice because the associated TLS can change anytime, so any value or pointer obtained through it could be accessible to another EA at the next moment anyway.

Consequences for implementations Option (1) is the easiest to implement. In contrast, option (5) restricts the implementation most in that each EA really has to have it's separate TLS space and constructors and destructors have to run. Because TLS is program-wide currently (although a compiler may be able to optimize this), short-lived EAs might face additional construction overhead.

Option (4) is somewhat better in that constructors/destructors do not need to be run on EA start/finish, so an implementation can implement such EAs on top of a `std::thread` thread pool, for example. Nonetheless, TLS and `this_thread::get_id` must not change across the EAs lifetime, so an implementation has to either (a) use exactly one underlying `std::thread` for such an EA (which can conflict with certain implementations of work stealing), change memory mappings, or introduce an indirection, and thus runtime overhead, for TLS (depending on the actual TLS implementation).

Options (2) and (3) should give implementations enough leeway in terms of which underlying `std::thread` is used while still allowing to run external or legacy code that might use TLS with little impact (unless, for example, there are several callbacks into the code and it expects to be executed by a single `std::thread`). However, code generation by the compiler is probably affected because accesses to TLS do not have the exact semantics as sequential code or TLS with options (4) and (5): TLS can change concurrently, so the compiler may have to prevent reloading of TLS values or may have to reload in other cases. Option (3) is a little less constraining for the compiler if all standard library functions, or at least those that can actually change the underlying `std::thread` in the particular implementation, are opaque to the compiler.

Options (3s) and (4s) are probably a little easier to implement than (3) and (4) because whether an EA has its own isolated TLS becomes basically a quality-of-implementation issue.

Storing a value for `this_thread::get_id` can be implemented using TLS; when TLS is shared (options (3s) and (4s)), one might still want to have distinct IDs for each EA (`this_thread::get_id` is currently specified to return an ID for a thread of execution, not just for a `std::thread`[8]). If so, we need to create unique IDs; we discussed this in Issaquah, and while some people felt that having a unique ID would be valuable and creating them would not cause a lot of runtime overhead, others though that creating unique IDs could be costly in terms of performance (e.g., on highly-parallel hardware with a large number of hardware threads).

It might also be interesting to reconsider whether, for an associated `std::thread`, there is still a relation to an underlying OS thread. Not establishing such a relation could make an implementation of light-weight EAs simpler because then it can implement the standard's TLS as required yet does not have to make the OS thread's TLS mechanism work.

Potential replacements for TLS I think that it would be valuable to investigate abstractions that can replace TLS usage, especially to make options (1) to (3) more widely applicable. Also, this could provide abstractions that are better suited to the particular use case than current `std::thread`-specific TLS; often, TLS is used as a mechanism but does not necessarily fulfill the intent perfectly.

When considering just a single EA, TLS is basically a special allocator combined with a very efficient way to obtain an EA-specific root pointer. Where TLS differs from that

[8]The standard does not provide a way to look up a `std::thread` object based on an ID, so we do not need to create a `std::thread`.

is that if you allocate a TLS data item (e.g., by adding a `thread_local` variable), that this automatically allocates one EA-specific data item for *every* past and future EA in the program; furthermore, that in each EA one can access this data item through a shared key (e.g., the `thread_local` variable).

I do not think that programmers need *exactly that* in the majority of use cases; usually, you need a TLS data item just for a subset of all EAs (e.g., all the EAs a subsequent parallel loop will spawn). Considering programs with several parallelized parts and systems that can run thousands of EAs potentially, this can make a difference.

Two major TLS use cases seem to be (a) global state that is accessed by just one thread and (b) locality of access. If TLS is used for the former, then allocating the data in the particular EA is fine, if it is possible to pass the pointer to the data along with other arguments to functions called by this EA, for example. If this data is expected to have the same lifetime as the EA, the implementation could even use a very efficient, local allocator for that.

If TLS is used for locality of access, then the motivation is primarily to try to reduce interference from other EAs, for example to reduce cache misses or access local memory. As an example, consider caches for sets of worker EAs on a NUMA system: A thread-safe cache could be costly if accessed from all EAs in the program due to lack of locality of memory accesses, but if all EAs running on one NUMA node would share a cache, the synchronization overheads would be small; giving each EA its own cache would remove the synchronization overheads completely, but could increase memory usage. In such a situation, it might be best to let an EA access the cache that is closest to the CPU core it is actually executing on; the instance of the cache might even change when the EA migrates to executing on another CPU core. This could be provided through a special allocator, for example one that allocates from per-workgroup memory on a GPU.

To summarize, I think that while TLS is important for compatibility with legacy code, specialized mechanisms should be better suited to address the actual intent behind using the TLS mechanism.

Lock ownership The standard already specifies lock acquisition semantics in terms of lock ownership of EAs, and notes that other EAs than those created by `std::thread` may exist (see §30.2.5). Thus, we do not need to define any association to any existing threads as for TLS.

However, implementations might have to be changed if they rely on having OS threads or `std::thread` as the only possible EA (e.g., if a mutex stores an OS thread ID to designate the lock owner). For options (4) and (5) above, implementations might be able to use existing thread-ID-based lock implementations. Implementations of weakly parallel execution such as SIMD execution could not use those, but weakly parallel EAs do not support blocking synchronization in general either; thus, perhaps the actual impact on implementations would be small.

4 A simple interface to schedule execution agents

So far, we have discussed EAs as purely a concept used to specify execution properties. Beyond that, one could add ways to directly create instances of EAs. Here is a sketch of how to make the requirements part of the type system, defining the different requirements as tag types:

```
// Forward progress requirements, ordered by strength
struct progress_weakly_parallel_tag {};
struct progress_parallel_tag : public progress_weakly_parallel_tag {};
struct progress_concurrent_tag : public progress_parallel_tag {};

// TLS requirements, ordered by strength
//   (Please ignore for a second that the following would only work with
//    virtual inheritance, whose overhead we want to avoid though.)
// Options (1) to (4):
struct tls_none_tag {};
struct tls_exists_tag : public tls_none_tag {};
struct tls_associated_tag : public tls_exists_tag {};
struct tls_stable_tag : public tls_associated_tag {};
// Option (3s) and (4s):
struct tls_associated_shared_tag : public tls_exists_tag {};
struct tls_stable_shared_tag : public tls_associated_shared_tag {};
// Option (5):
struct tls_thread_tag : public tls_stable_tag, public tls_stable_shared_tag {};
```

Next, we can define concrete EA types and traits. In this example, there is a parallel EA and an EA that is like those created by `std::thread`:

```
// Properties of an EA
template<class EA> struct ea_traits;

// A parallel EA type without TLS support
struct ea_lean_parallel {};
template<> struct ea_traits<ea_lean_parallel> {
  typedef progress_parallel_tag progress;
  typedef tls_none_tag tls;
};

// An EA type that has the properties of EAs created by std::thread
struct ea_stdthread {};
template<> struct ea_traits<ea_stdthread> {
  typedef progress_concurrent_tag progress;
  typedef tls_thread_tag tls;
};
```

If we have interfaces that spawn EAs (e.g., something similar to executors), then we can use the traits to select the most efficient implementation at compile time. We can also use them to disallow trying to create kinds of EAs that are not supported by the particular interface. For example, a strictly bounded thread pool cannot guarantee

concurrent execution of all EAs in general, so it might just refuse spawning concurrent EAs altogether:

```
// Executor-like example: Bounded thread pool can't launch concurrent EAs
struct bounded_threadpool {
  template<class Function>
  void spawn(Function f, progress_parallel_tag) { /* Can do it. */ }
  template<class Function>
  // Can't create concurrent EAs:
  void spawn(Function f, progress_concurrent_tag) = delete;
  template<class EA, class Function>
  void spawn(EA&, Function f) {
    spawn(std::forward<Function>(f), typename ea_traits<EA>::progress{});
  }
};
```

Considering executors in general, I think it would be beneficial to parametrize them or their member functions by an EA type. Although one could argue that, for example, a client of a bounded thread pool should be well aware which properties EAs created by it have, making the requirement explicit helps catch mistakes and is less error-prone in generic code.

More importantly, it separates semantic constraints for the execution of a particular piece of code from all the other reasons the programmer might have to use a specific executor—in particular, performance reasons.

For example, a programmer might want to use a thread pool with a fixed number of OS threads primarily because those threads may have been bound to the cores on a particular CPU socket, not because the programmer was thinking about bounded vs. not bounded and the consequences for progress. The thread pool implementation even could add additional OS threads to the pool if the program requests it to schedule more concurrent EAs. Another example would be a GPU executor: Its purpose would be to run EAs on the GPU, not just to run a particular type of EA.

We could specify one concrete executor for each combination of execution requirements (e.g., concurrent GPU, parallel GPU, parallel-no-TLS GPU, ... executors) but I do not think this would be easier to consume for readers of the standard. Also, if we want to combine, chain, or merge executors, then having a larger number of concrete executors makes the implementation more complex too.

As a last example for simple interfaces, let us consider a facility to upgrade the currently executing EA to stronger EA requirements:

```
// Spawns and boost-blocks on just one EA
template<class EA, class Function> void upgrade_ea(EA& ea, Function f);

// Just need stable TLS, but no constraints on progress
struct ea_stable_tls {};
template<> struct ea_traits<ea_stable_tls> {
  typedef progress_weakly_parallel_tag progress;
  typedef tls_stable_tag tls;
};
```

```
// ... code that does not use TLS ...
upgrade_ea(ea_stable_tls, legacy_code_that_needs_tls);
// ... code that does not use TLS ...
```

`upgrade_ea` spawns a new EA and waits for its termination. It will give the new EA requirements that are not weaker than neither the current EA nor the supplied EA type. Note that for the progress requirement, this could as well be achieved by specifying that `upgrade_ea` boost-blocks on the completion of the spawned EA.

5 Suggested wording for version 2 of the Parallelism TS

There was consensus in Urbana that the next step regarding the forward progress definitions would be to apply them to other proposals, notably the Parallelism TS. Thus, in what follows, I will propose wording for the next version of this TS, relative to N4352.

To avoid having to use and define execution agents, the proposed wording simply associates forward progress requirements directly with threads of execution. I still think that using EAs as a concept makes sense, but it's not strictly necessary for the uses in the current Parallelism TS.

First, we need to add the base definitions around progress. Those are not specific to the Parallelism TS but would rather replace the existing requirements in the standard; nonetheless, we need to refer to them, so just add a subsection titled "Forward progress" before Section 4.1.2, with the following content. The definitions are essentially the same as discussed previously in this paper except associating progress with threads of execution directly. Also, a few explanatory notes have been added:

> [Note: The definitions and requirements in this section are supposed to replace the forward progress requirements in the current C++ standard. — end note]
>
> The execution of threads of execution consists of *steps*: A step ends with either (1) termination of the thread of execution, (2) access to or modification of a volatile object, or (3) a synchronization or atomic operation; the remainder of a step consists of other things executed by the implementation that are not in the previous list. [Note: A thread of execution makes progress when its steps are executed. — end note]
>
> Calls to library I/O functions and other functions that block (i.e., that cannot return to the caller unless a certain, potentially external condition is met) can be conceptually expressed using steps as defined previously; each blocking operation can be replaced by a busy-waiting loop that polls for the condition to be met (e.g., using observable behavior of the abstract machine).
>
> [Note: It is not necessary to provide such specifications of blocking. The previous paragraph only states that it is always possible to do so, which allows separating blocking due to program logic from blocking due to how an implementation executes the abstract machine. — end note]

The implementation should ensure that a thread of execution providing *concurrent forward progress guarantees* will eventually be allowed to execute all its steps, independently of which steps other threads of executions might or might not execute. [Note: To eventually fulfill this requirement means that this will happen in an unspecified but finite amount of time. — end note]

[Note: The threads of execution created by `std::thread` and the implementation-created thread of execution that executes `main` provide concurrent forward progress guarantees. — end note]

A thread of execution providing *parallel forward progress guarantees* cannot be expected to be allowed by the implementation to execute steps if it has not yet executed any step; once it has, the implementation should ensure that it will eventually be allowed to execute all of its steps, independently of which steps other threads of execution might or might not execute.

[Note: This effectively makes the same progress requirements as for a concurrent forward progress guarantees once the parallel-forward-progress thread of execution is started, but does not specify a requirement for when to start this thread of execution; the latter will typically be specified by the entity that creates this thread of execution. — end note]

For a thread of execution providing *weakly parallel forward progress guarantees*, the implementation does not need to ensure that this thread of execution is allowed to execute steps independently of which steps other threads of execution might or might not execute.

[Note: Concurrent forward progress guarantees are stronger than parallel, which in turn are stronger than weakly parallel guarantees. For example, some kinds of synchronization between threads of execution may only make forward progress if the respective threads of execution provide parallel forward progress guarantees, but fail to make progress under weakly parallel guarantees. — end note]

If a thread of execution P uses *boost-blocking* to block on the completion of a set S of threads of execution, and if P is providing a stronger forward progress guarantee than at least one thread of execution in S, then throughout the whole time of P being boost-blocked on S, P will boost the progress requirement of at least one thread of execution in S to P's stronger guarantee. Specifically, P is free to select which thread of execution in S to boost and for which amount of time (i.e., the boost is not permament and in place for the rest of the lifetime of the boosted thread of execution); as long as P is boost-blocked, it has to eventually boost a thread of execution in S. Once a thread of execution in S finishes execution, it is removed from S. Once S is empty, P stops being blocked.

[Note: A thread of execution thus can temporarily provide an effectively stronger forward progress guarantee for a certain amount of time, due to a second thread of execution being boost-blocked on it. In turn, this may allow

the former thread of execution to itself boost the progress of a third thread of execution that it is itself boost-blocked on. — end note]

[Note: If all threads of execution in S finish executing (e.g., they terminate and do not use blocking synchronization incorrectly), then P's progress guarantee will not be weakened by executing the boost-blocking operation. — end note]

[Note: This does not remove any constraints regarding blocking synchronization for threads of execution providing parallel or weakly parallel forward progress guarantees because P is not required to boost a particular thread of execution whose too-weak progress guarantee is preventing overall progress. — end note]

With these foundations in place, we can then adapt Section 4.1.2 as follows. Generally, to avoid misinterpretations, we should replace all occurrences of "thread" with "thread of execution" (see N4231 for background).

In the third paragraph, specify that threads of execution supplied by the library provide parallel forward progress guarantees:

> The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `parallel_execution_policy` are permitted to execute in an unordered fashion in either the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution; the latter will provide parallel forward progress guarantees. Any such invocations executing in the same thread of execution are indeterminately sequenced with respect to each other. [Note: It is the caller's responsibility to ensure correctness, for example that the invocation does not introduce data races or deadlocks. — end note]

Similarly, specify weakly parallel guarantees in the fourth paragraph. Also, clarify that affected threads of execution are the invoking thread or managed by the library, and that blocking synchronization is the problematic case, not all kinds of synchronization in general:

> The invocations of element access functions in parallel algorithms invoked with an execution policy of type `parallel_vector_execution_policy` are permitted to execute in an unordered fashion in unspecified threads of execution, and unsequenced with respect to one another within each thread of execution. These threads of execution are either the invoking thread of execution or implicitly created by the library; the latter will provide weakly parallel forward progress guarantees. [Note: This means that multiple function object invocations may be interleaved on a single thread of execution. — end note]
>
> [Note: This overrides the usual guarantee from the C++ standard, Section 1.9 [intro.execution] that function executions do not interleave with one another. — end note]

> Since `parallel_vector_execution_policy` allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. Thus, code with `parallel_vector_execution_policy` is restricted as follows:
>
> A standard library function is vectorization-unsafe if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, and if it is not a memory allocation or deallocation function. Vectorization-unsafe standard library functions may not be invoked by user code called from `parallel_vector_execution_policy` algorithms.
>
> [Note: Implementations must ensure that internal synchronization inside standard library routines does not prevent forward progress when those routines are executed by threads of execution with weakly parallel forward progress guarantees. This can be achieve by either avoiding synchronization incompatible with those guarantees, or by executing these routines differently. — end note]

I would have liked to make the two paragraphs before the last note of this more precise; however, we need to use some categorization of synchronization that the current standard gives us, and it doesn't make a distinction between blocking and general synchronization.

The changes to the last note attempt to clarify what I understand as the actual takeaway: The deadlock mentioned is a clash between blocking synchronization and weak progress, and implementations can either avoid such blocking or make sure that progress is stronger (e.g., by not "vectorizing" calls to such routines).

Finally, we need to ensure that threads of execution implicitly created by the library do not weaken the forward progress of the threads of execution invoking the library. Add a new paragraph before paragraph 6:

> If an invocation of a parallel algorithm uses threads of execution implicitly created by the library, then the invoking thread of execution will boost-block on the completion of these library-managed threads of execution. [Note: In boost-blocking in this context, a thread of execution created by the library is considered to have finished execution as soon as it has finished the execution of the particular element access function that the invoking thread of execution logically depends on.]

5.1 What if we consider just forward progress?

It may seem surprising that the changes proposed previously do not significantly change the wording regarding which threads can be used to execute a parallel algorithm. This is not because the forward progress definitions would be insufficient, but because I did not want to change the semantics of TLS and lock ownership.

If we do not rule out TLS accesses and want to allow typical implementations, we need to cover executions in which element access functions access both the invoking thread's

TLS and TLS of threads managed by the library. Thus, this is a TLS visibility/lifetime issue, not a limitation of the forward progress model.

This is furthermore problematic because allowing functions to be executed in indeterminately ordered fashion on the invoking thread is not just needed to allow implementations to do exactly that, but also to—indirectly—create a forward progress constraint: If the order is not specified, we cannot expect another element access function to start executing eventually (which covers the weaker guarantee of parallel EAs compared to concurrent EAs). The same problem exists in the specification of the parallel-vector policy. Thus, this cannot be used to describe an execution policy that would never execute on the invoking thread (e.g., because it really wants to promise to execute the algorithm on a capable accelerator) — because disallowing execution on the invoking thread would remove the progress constraint.

Finally, basing the specification solely on which threads of execution are used binds thread identity and TLS to progress guarantees: We cannot specify a parallel-vector policy that provides weakly parallel progress for the element access functions but gives each concurrent invocation of such a function its own, non-shared TLS. This is because if this would be the case (i.e., visible to the program through different thread identities), a program could infer that there are separate threads, and thus also assume the default progress guarantees for threads of execution specified by the current standard (i.e., concurrent progress).

In contrast, if we were to ignore or disallow TLS and lock ownership in the specification, or would specify these aspects by assigning TLS support levels to execution agents (see Section 3), we could simply let each element access function run on one conceptual execution agent managed by the implementation.

In such a scenario, we would just need to say the following for the third paragraph (now using execution agents as discussed in this paper)...:

> The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `parallel_execution_policy` will be executed using parallel execution agents. [Note: It is the caller's responsibility to ensure correctness, for example that the invocation does not introduce data races or deadlocks. — end note]

...and the fourth paragraph:

> The invocations of element access functions in parallel algorithms invoked with an execution policy of type `parallel_vector_execution_policy` will be executed using weakly parallel execution agents.
>
> Since certain, for example certain uses of mutexes, risk deadlock when run by weakly-parallel execution agents, synchronization with `parallel_vector_execution_policy` is restricted as follows:
>
> A standard library function is vectorization-unsafe if it is specified to synchronize with another function invocation, or another function invocation is

specified to synchronize with it, and if it is not a memory allocation or deallocation function. Vectorization-unsafe standard library functions may not be invoked by user code called from `parallel_vector_execution_policy` algorithms.

[Note: Implementations must ensure that internal synchronization inside standard library routines does not prevent forward progress when those routines are executed by weakly parallel execution agents. This can be achieve by either avoiding synchronization incompatible with those agents, or by executing these routines differently. — end note]

This may not constitute a large improvement when just considering the current Parallelism TS, but should improve clarity when also considering a future Concurrency TS, the executor proposals, and the Networking TS — all of which will have to specify forward progress requirements and constraints.

6 Acknowledgements

I thank Jonathan Wakely for his input on Section 4 and review of earlier revisions of this paper. Jared Hoberock proposed to represent progress requirements in the type system.

7 Revision history

Changes between N4439 and N4156:

- Added Section 5 (suggested wording for version 2 of the Parallelism TS).
- Clarified that this paper assumes that a thread of execution is a dynamic entity.
- Clarified that `main` is supposed to be executed by an EA semantically equivalent to those created by `std::thread`.
- Mention "execution context" as another naming alternative to execution agent.
- Clarified that boost-blocking only applies to the set of execution agents that have not yet finished execution. Clarified that the boosting is only temporary and not persistent for the rest of the lifetime of the boosted EA. Added a note that boost-blocking can boost progress transitively.
- Slightly expanded the note on the definition of parallel EAs regarding not having a requirement when the EA gets started, and that this typically would be specified by the entity that launches such an EA.
- Applied non-controversial definition changes suggested in Urbana.

Changes between N4156 and N4016:

- Added Section 4 on interfaces and EA types.
- Revised discussion of thread-specific features; added levels of potential TLS requirements.
- Added "boost-blocking" definition, refined other progress definitions. Added more examples.
- Removed section on strict SIMD execution.
- Removed discussion why lighter-weight-than-OS-thread EAs are useful.
- Removed overview of other proposals containing light-weight EAs.
- Removed open questions: Settle on EAs being primarily a conceptual entity, remove thread compatibility mode and OS thread `std::thread` question.

Changes between N3874 and N4016:

- Added underlying progress definition based on implementation steps and scheduler guarantees.
- Refined forward progress classes: Base definitions on implementation steps instead of blocking; merge SIMD+Parallel with the weaker variant of Parallel and rename to Weakly Parallel; refine safety guarantee description of SIMD and rename to Strict SIMD.
- Added discussion of semantics of spawning and blocking on EAs.
- Added a more detailed overview of related proposals.