Light-Weight Execution Agents

Revision 2

Document number:	N4016
Revises:	N3874 (see Section 5 for revision history)
Date:	2014-05-23

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

1 Introduction

The standard currently uses the notion of threads of execution to allow for concurrent or parallel execution. It provides thread objects as a way to create and join threads of execution. This is a portable abstraction for threads as offered by many operating systems (e.g., POSIX Threads), which I will refer to as *OS threads* from now on.

OS threads are a specific mechanism, and they come with a set of quasi-standard features such as support for thread-local storage (i.e., objects with thread storage duration, §3.7.2). They also offer relatively strong forward progress guarantees, in particular that an unblocked thread should eventually make forward progress (§1.10). Users of threads rely on these properties when, for example, building concurrent applications: If two threads are in a producer–consumer relationship, the producer needs to make forward progress or the consumer will be blocked as well.

However, we do not need full-featured OS threads for all use cases. For example, when a program needs to crunch 1000 disjoint groups of numbers, it can do so in parallel but it does not need to run 1000 OS threads for this; if the groups are in fact independent, then working on one group does not need the results from work on another group, so we do not need concurrent execution. Instead, for the implementation, the number of OS threads to be used in this example is a performance decision.

While full-featured OS threads can easily exploit thread-level parallelism offered by the hardware, they also can be costly:

- Space overhead of a thread's stack and other thread-local data.
- Construction/destruction of thread-local storage.

- Ensuring concurrent execution between all threads, at least eventually, which results in context switch costs and having to schedule all threads.

The implementation/OS has some leeway and can try to avoid some of these overheads, but this can be difficult in practice. For example, an OS scheduler typically cannot detect whether one OS thread spin-waits for another OS thread, so the scheduler faces a trade-off between, say, trying to avoid frequent context switches and letting one thread wait more than necessary; even if the threads use OS mechanisms to wait, it might—depending on the mechanism and the threads' synchronization—not be visible to the scheduler which other thread is being waited for. Similarly, I suspect that many programmers expect the default scheduler to run OS threads in a more or less round-robin fashion.

Thus, if a program does not actually need concurrent execution (as in the parallel number-crunching example above) or other features of full OS threads, then using OS threads will lead to unnecessary runtime and space overheads.

In turn, if we want to use other lighter-weight implementations of concurrent/parallel execution, then those cannot provide full-featured OS threads. For example, a bounded thread pool is not a valid implementation of full OS threads (and thus the threads abstraction in the standard) because it does not guarantee concurrent execution (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).

This paper defines semantics of light-weight *execution agents* (EAs), which can be used to run more than one thread of execution but provide weaker forward progress guarantees and/or fewer features than OS threads (see Section 2). From this perspective, threads would be a rather heavy-weight variant among several kinds of EAs. I will give an overview of other SG1 proposals that incorporate concepts similar to light-weight EAs in Section 3, and conclude in Section 4.

2 Light-weight execution agents

In the standard, EAs are defined in §30.2.5.1: "An execution agent is an entity such as a thread that may perform work in parallel with other execution agents". They are used to specify lock ownership, although the term "threads" is used in this context as well.

Several existing proposals to SG1 incorporate execution agents (even though this term is not used) that are seemingly weaker than threads (see Section 3 for details): Even though the programming abstractions presented in these proposals are different, the conceptual EAs provided by them are often similar, and differ from threads in (1) the forward progress guarantees they provide and (2) how they handle other thread features, notably thread-local data. While different programming abstractions or interfaces can be useful, I believe that it would be beneficial to at least unify the execution concepts being used across these proposals, where possible. This would make the parallelism and concurrency support in the standard easier to grasp for programmers, and it would probably also ease implementing these proposals because they can then be put on top of one common base for shared usage of computing resources.

2.1 Basic forward progress definitions

At the Issaquah meeting, a few people were concerned about the squishy definition of progress that's currently used by the standard. 1.10p2 states that "Implementations should ensure that all unblocked threads eventually make progress."

"Progress" is not explicitly defined but is likely to be understood as 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).

The implementation has no control over whether the program itself is blocking, so it can only guarantee that threads are actually executed. Thus, 1.10p2 should probably rather state that implementations should[1] ensure that all threads are eventually allowed to make (implementation) steps, independently of other threads. This clarifies that the guarantee is about providing each thread with the compute resources it needs, and that one thread's actions cannot prevent another thread from making implementation steps. In other words, what the implementation should provide is a thread scheduler that never starves any thread, and is thus fair (at least to some degree).

If we need a more detailed definition of what an implementation step is, it would probably be best to base this on 1.10p24, which states that any thread will eventually either terminate, make a call to a library I/O function, access a volatile object, or use atomics or other synchronization. We can define a step as an execution of a sequence of elements that are not on this list followed by an item that is on this list.

Defining the underlying progress guarantee provided by the implementation in this way has the following advantages:

- It avoids defining the thread schedulers properties based on what it means for a thread to be blocked.
- Issues like blocking file I/O are separated from what the implementation has to guarantee; they might not be under control of the implementation, but could—if needed—still be modeled similar to blocking on another thread.
- It should be easier to explain this based on OS thread concepts like the scheduling of threads (e.g., a round-robin scheduler is fair).

[1] This definition uses "should" instead of "will" (as in the wording of §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, for general-purpose implementations, this should be a strong guarantee, I believe (i.e., "will").

2.2 Classes of forward progress guarantees

Next, I will discuss four classes of forward progress guarantees that EAs can provide. Three of them are either equal to the progress guarantee provided for threads by an implementation or weakened variants of it. The fourth class is a combination of a progress guarantee with an additional safety guarantee.

Concurrent execution This class provides the same guarantees as threads: EAs in this class will eventually be allowed by the scheduler to make all implementation steps they need to execute, independently of what other EAs (including threads) 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. They can block on the other EA and still make progress in the form of observable behavior of the abstract machine, unless the blocking relationship is cyclic.

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.

Parallel EAs cannot be expected to make implementation steps if they have not yet made any step; once they have performed their first step, they will eventually be allowed by the scheduler to make all implementation steps they need to execute, independently of what other EAs are doing. In other words, a parallel EA will behave like a thread, but only after performing its first step.

This easily maps onto a thread-pool-based implementation; to give it full flexibility regarding resource usage, this does not reveal how many resources are used (i.e., like in case of a bounded thread pool that exposes its specific bound to users). This is easy to implement because we just need to execute all parallel EAs eventually, 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.[2]

This progress guarantee is stronger than 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 EAs start and might acquire a mutex, they are guaranteed to eventually be able to finish execution, and thus they will not block other EAs indefinitely.[3]

Weakly parallel execution Weakly parallel EAs cannot be expected to make implementation steps concurrently with other EAs. Thus, they get weaker progress guarantees

[2] 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, a deadlock will arise.

[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.

than parallel EAs in that there is no upgrade to concurrent execution once a weakly parallel EA has started executing.

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.[5]

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.

Strict SIMD execution This class attempts to model the guarantees of code that uses *only* SIMD instructions to execute several EAs running the same code (e.g., independent iterations of a SIMD loop as in N3734). To allow such an implementation, SIMD EAs must not expect to make forward progress independently of other SIMD EAs in the same context (e.g., in the same SIMD loop). In other words, they execute in lockstep with each other, and the granularity of this is implementation defined.

Strict SIMD EAs are a combination of weakly parallel execution and a safety guarantee: They cannot be *generally* expected to make implementation steps concurrently with other EAs, yet they are guaranteed to execute in a specific interleaving derived from the sequenced-before relation of the common code executed by a group of SIMD EAs (see N3734 for details). Compared to weakly parallel EAs, in some use cases the additional safety guarantee allows for a more effective use of SIMD hardware on some architectures.

This guarantee means that the use of typical forms of critical sections is not possible because we cannot expect to execute the critical sections in each SIMD EA one after the other.[6] Like for weakly parallel EAs, nonblocking synchronization is possible.

2.3 Thread-specific state and features

Besides forward progress guarantees, we also need to consider how light-weight EAs relate to threads and features of threads, in particular state associated with particular

[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.

[6] However, as for parallel executions, some uses of mutexes might still be allowed; this indicates that specifying the progress guarantees is a more precise way to specify EAs than by trying to disallow the use of certain features (e.g., mutexes) altogehter.

thread instances. While programmers often will not need these particular features, we need to at least define the level of compatibility with existing thread-based code.

The following does not give concrete recommendations for how to deal with thread-specific features but rather describes the issues, which also affect many proposals that incorporate EA-like constructs (see Section 3).

Thread-local storage In N3556, "Thread-Local Storage in X-Parallel Computation", Pablo Halpern presents a classification of how different parallel execution models treat thread-local storage (TLS). The discussion in this paper applies in a very similar way to EAs; however, it could be argued that some of the concerns are tied to programming abstractions that spawn nested parallelism, whereas EAs could also be created in different ways.[7]

N3556 also mentions "x-local" storage, which would be distinct from TLS and scoped to instances of parallel tasks, for example. From the EA perspective, this seems to be the right approach. Nonetheless, I think that providing EA-local storage might not be ideal because, like TLS, it requires programmers to link the semantics of this state to specific execution mechanisms like EAs or threads. Instead, it might be beneficial to let programmers request a local storage mechanism by describing the intent behind using local storage. For example, programmers currently use TLS as both (1) storage that will not be accessed by multiple threads unless explicitly shared and (2) storage that will likely have good data locality when accessed from this thread. In other words, the use of TLS can be motivated by both wanting certain semantics (e.g., no concurrent accesses to it) and performance considerations (e.g., concurrent accesse being unlikely to avoid cache misses). TLS is an implementation mechanism that can be used for that, but other implementations are possible as well (e.g., per-workgroup storage on GPUs).

Emulating `this_thread::get_id()` This function returns an ID for the current thread of execution. However, a light-weight EA may not be a thread, so either we need to return some distinct value for an imaginary thread[8] or we need to handle this similarly to TLS, as discussed in N3556.

The discussion in Issaquah was inconclusive on this point: Some people felt having a unique ID would be valuable and creating them would not cause a lot of overhead, whereas others though that creating unique IDs could be costly in terms of performance.

Lock ownership The standard already specifies lock acquisition semantics in terms of lock ownership of EAs, and notes that other EAs than threads may exist (see §30.2.5). Thus, we do not need to define any association to any existing threads as N3556 does for TLS.

[7] For example, when exposing parallel execution opportunities via a parallel loop, then what the loop does is often related to the code that started the loop and thus spawned parallel EAs; in contrast, when spawning concurrent EAs (e.g., in an actor model), these often may not have a immediate relation to the EA that spawned them (i.e., similar to how threads are used today).

[8] The standard does not provide a way to look up a thread object based on the ID, so we do not need to create this imaginary thread.

However, implementations might have to be changed if they rely on having threads as the only possible EA (e.g., if a mutex stores an OS thread ID to designate the lock holder). Implementations of parallel execution that associate one thread with an EA throughout the whole lifetime of the EA 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.

2.4 Spawning and blocking on EAs

As presented so far, EAs are purely a concept used to specify execution properties. Beyond that, one could add ways to directly create instances of EAs, similar to how threads can be created. There are advantages and disadvantages of doing so (see Section 4), and a few people at the Issaquah meeting were interested in seeing such interfaces.

I believe that it is likely that there will not be a single interface that works best for all kinds of EAs and use cases. Nonetheless, such interfaces will have to deal with similar conceptual aspects. In what follows, I will use a simple exemplary interface to discuss the semantics of spawning and blocking on EAs.

Consider two functions, `spawn` and `block`. The former creates another EA, taking a function object as argument (which specifies what to execute). Spawning an EA also potentially starts execution of it, but does not otherwise affect the spawning EA, which keeps executing. All spawned EAs are implicitly associated with the spawning EA.

A call to `block` makes the calling EA block for completion of all EAs that it has spawned. I believe that it would be beneficial to define the exact semantics of `block` depending on the classes of the spawning and spawned EAs.

First, if the spawning EA runs under a stronger forward progress guarantee than at least one EA in the group of EAs it has spawned, then the spawning EA will boost the progress guarantee for at least one of the spawned EAs it is still waiting on throughout the whole time it is blocked.

For example, if a concurrent EA is blocked on a group of parallel EAs, it is guaranteed to eventually start all of them. If a parallel EA spawned weakly parallel EAs, then once the parallel EA has been started, one of the weakly parallel EAs will always eventually make implementation steps; in this case, the boost might switch back and forth between different weakly parallel EAs, which is in line with the progress guarantees for those.

This kind of boosting of progress guarantees makes sure that the guarantees of an EA are not broken when it blocks on EAs with weaker progress guarantees.[9] This also takes care of any progress bootstrap problem if we assume that each program starts with a single thread being active. Furthermore, it should be straight-forward to implement when EAs are based on threads.

[9] Note that this does not cover blocking as caused by, for example, trying to acquire a lock that is held by another weaker-progress EA. This would be similar to avoiding priority inversion, which can be useful, but comes at an implementation complexity and performance cost.

Besides affecting forward progress, the properties of the spawning EA could also determine how `block` affects thread-specific state: For example, if a thread (i.e., an EA with strong guarantees) blocks on a parallel EA, `block` should return on the same OS thread as it was called on. It would also be possible to require the programmer to use different variants of `block` to get different semantics (e.g., similar to N3832's `task_region` vs. `task_region_final`), but having `block` choose automatically based on the executing EA seems more intuitive and providing better composability and code reuse. Furthermore, this would also enable a combined `spawn` and `block` function that can be used to neatly upgrade to a stronger EA (e.g., a thread) for a while for regions of code that needs stronger guarantees (e.g., continuity of thread-local storage).

3 Light-weight EAs in other proposals

N3724 "A Parallel Algorithms Library" This provides execution modes that correspond to sequential execution (`std::seq`), or parallel (`std::par`) or weakly parallel (`std::vec`) execution as defined in Section 2.2.[10]

N3832 "Task Region" This provides means to execute parallel work using a fork/join approach. Based on the discussion of queue-based scheduling in this paper, it seems that the forward progress guarantee is parallel execution as defined in Section 2.2; however, weakly parallel execution is not clearly disallowed.

N3872 "A Primer on Scheduling Fork-Join Parallelism with Work Stealing" This compares different kinds of work stealing. Child stealing can be used to implement parallel execution as defined in Section 2.2, whereas continuation stealing can only be used to implement weakly parallel execution if using a bounded number of threads or if not all kinds of blocking can be reliably detected by the implementation.

N3722 "Resumable functions" This proposes language constructs that allow programmers to write code as if they had EAs that execute concurrently but do not provide all features that threads provide. According to feedback received in Issaquah, the forward progress guarantee is intended to be concurrent execution.[11]

N3734 "Vector Programming: A proposal for WG21" This presents language constructs for SIMD loops and functions. Independent iterations of such loops can be considered to be execution agents that execute in lockstep. Strict SIMD execution in Section 2.2 is intended to cover these semantics.

[10] This statement is based on feedback received in Issaquah.

[11] The paper advises against holding a lock across `await` calls because it assumes that lock implementations may be based on threads or thread IDs; if locks were tied to EAs, they should be ready to use.

N3731 "Executors and schedulers, revision 2" This proposes library interfaces to create EAs with different safety and liveness properties (e.g., where tasks created by one executor run one after the other).

4 Conclusion and open questions

I believe that it is important to provide light-weight EAs or to at least thoroughly define the semantics if no direct access to them is provided. The number of existing proposals to SG1 that relax execution guarantees compared to OS threads indicates that there is a real need for light-weight EAs.

Nonetheless, this paper is just a step towards that, so there are many open questions, of course. Some of them are outlined in what follows. Others are not discussed in this paper at all, but are important: For example, how to make the use of EAs efficient in terms of resource usage, and how to do so with a portable and abstract interface that does not rely on the programmer tightly controlling the specific computing resources that are being used (e.g., the number of OS threads).

Programming abstraction or conceptual entity? As presented so far, EAs are purely a concept used to specify execution properties. Beyond that, one could add ways to directly create instances of all or the most important kinds of EAs, similar to how threads can be created.

One advantage of doing so would be to give programmers full access to the shared resource usage facilities that an implementation would likely do internally anyway (e.g., balancing out the number of OS threads used across the program, independently of which parallel or concurrent abstraction was used to spawn EAs). However, finding a portable, stable, yet powerful interface for that might be difficult. The Executors proposals is about a similar direction, and currently proposes a few specific factories for EAs instead of covering all useful combinations of EA semantics (e.g., forward progress, TLS handling, ...) and performance properties (e.g., how many and which resources to use for execution).

Some of the EAs might be better exposed through specialized interfaces. SIMD execution can be such a case when the implementation requires custom code generation for such EAs, or to convey the context of the SIMD EAs (e.g., other iterations in a SIMD loop).

Even proposals that do not require customly generated code might be better served with a specialized interface. For example, `cilk_spawn` uses a language construct to make spawning parallel EAs look like a function call, yet the return value of this call might not be available until an implicitly associated or explicit `cilk_sync`. Resumable functions also use a function-call–like language construct to provide virtual concurrent EAs without using OS threads, but the mechanism used for returning values is based on futures.

Legacy code: Support `std::thread` or OS threads? To provide support for legacy code that assumes threads and not lighter-weight EAs, implementations at least need to define guarantees for uses of `std::thread`. In most implementations, `std::thread` is probably implemented by just using OS threads, but this may be difficult when implementing some of the ideas in N3556. Thus, should implementations strive for compatibility features for OS threads? While this may remain an implementation-defined choice, I think this needs to be considered.

How do we expose a thread compatibility mode? One way to do it is by giving guarantees such as the stronger ones in N3556, which would make compatibility with threads a part of the EA semantics. Another way would be to provide an interface to bind an EA to a thread at runtime, thus effectively transforming it into a stronger EA for a while (e.g., via bind/unbind calls, or with the combined `spawn` and `block` functions discussed in Section 2.4).

5 Revision history

Changes between N3874 and N4016:

- Added underlying progress definition based on implementation steps and scheduler guarantees (Section 2.1).
- Refined forward progress classes (see Section 2.2): 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.
- A few changes in Section 2.3.
- Added discussion of semantics of spawning and blocking on EAs (Section 2.4).
- Added a more detailed overview of related proposals (Section 3).