Light-Weight Execution Agents

Document number: N3874
Date: 2014-01-20
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 is an initial attempt at defining 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-weigth variant among several kinds of EAs. This work is based on or has been motivated by prior discussions in SG1. It also is at an early stage, so I will conclude by discussing some of the open questions in Section 3.

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:

- N3724, "A Parallel Algorithms Library", provides execution modes that are essentially sequential, parallel, or allow both parallel and SIMD.
- N3409, "Strict Fork-Join Parallelism", allows for spawning parallel tasks.
- N3722, "Resumable functions", 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.
- N3734, "Vector Programming: A proposal for WG21", presents language constructs for SIMD loops and functions. Independent iterations of such loops can be considered to be execution agents that execute in lockstep.

- N3731, "Executors and schedulers, revision 2", 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).

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 Forward progress

Next, I will discuss four classes of forward progress guarantees that EAs can provide.

Concurrent execution This class provides the same guarantees as threads: EAs should eventually make progress if they are not blocked. Threads are blocked when they use features of the implementation that make their progress depend on the progress and execution of other EAs (e.g., by blocking on a mutex).[1] If this is not the case, the implementation's scheduler should eventually let them make forward progress, for each execution step they attempt—independently of what other EAs are doing.

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").

Note that sometimes, this progress guarantee is summarized as being able to synchronize. However, this is not an accurate description because it really is about forward progress and not synchronization in general. While nonblocking synchronization is allowed for EAs providing weaker guarantees (e.g., see parallel execution below), only concurrent execution allows for some kinds of blocking synchronization.

Parallel execution Parallel execution is weaker than concurrent execution in terms of forward progress. Specifically, one possible definition would be that such EAs cannot expect other parallel EAs to make progress concurrently.

This definition captures 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. To give the implementation full flexbility regarding resource usage, this does not reveal how many resources are used (i.e., like in case of a bounded thread pool that

[1] This should cover spin-waiting. When treating a, for example, spin lock as a black box, blocking using the spin lock is well-defined. When looking at the internals of the spin lock, the guarantee will make sure that the EA keeps spin-waiting.

exposes its specific bound to users). This is easy to implement because we just need to execute all such EAs eventually, in some order and interleaving.

However, this definition does not allow typical uses of critical sections inside of parallel EAs (e.g., to synchronize access to shared state), because then an EA might wait for another EA that is not guaranteed to make progress concurrently.[2]

A stronger variant of the former definition can be obtained by additionally guaranteeing that a parallel EA will make progress eventually once it has started to execute; in other words, once it starts, it is similar to a concurrent EA. This does allow typical uses of critical sections because as soon as EAs start and might acquire a mutex, they are guaranteed to finish execution and will not block other EAs indefinitely.[3] This is easy to implement with a typical thread pool—however, it cannot be implemented with, for example, certain kinds of work-stealing schedulers if work-stealing is allowed to happen during critical sections.[4]

SIMD execution This class attempts to model the guarantees of code that uses 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 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.

This guarantee disallows the use of typical forms of critical sections because we cannot expect to execute the critical sections in EAs one after the other.[5]

Unlike concurrent and parallel execution, SIMD execution is also special in that it not just gives a progress guarantee but can also give a safety guarantee: At least as specified by N3734, iterations of the same SIMD loop will be virtually executed in sequential order. However, it could be argued that this is a property of the mechanism using SIMD EAs (e.g., N3734), rather than a property of SIMD EAs.

While the progress guarantee is satisfied by an implementation that uses several concurrent threads to execute each EA, the safety guarantee is not. Also note that lock-free synchronization is allowed in SIMD EAs, whereas obstruction-free synchronization is unlikely to succeed due to EAs executing in lockstep being likely to interfere with each other.

[2] Specifically, 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).

[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 owns the mutex before it got started.

[4] 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.

[5] 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.

Parallel+SIMD execution This last class tries to allow for both parallel and SIMD execution, at the choice of the implementation (as in N3724). This means that its forward progress guarantees will not be stronger than parallel or SIMD in isolation (in other words, it's the weakest guarantee).

Whether this class actually needs to exist is debatable and depends on how parallel and SIMD execution are defined in detail. Both the stronger variant of parallel execution as well as the possible safety guarantee of SIMD execution aren't possible anymore because they are not provided by the respective other class. However, it may be that without these stronger parts, both classes are actually indistinguishable from each other because they disallow the same code from being executed.[6]

2.2 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 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.

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).

[6] Specifically, it might be that they disallow the same kinds of blocking code, but for different reasons: If we cannot assume that other EAs make progress, then EAs cannot block on each other. Likewise, if EAs cannot make progress independently, then they must not make other EAs block.

[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).

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.

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.

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).

3 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 first 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 very 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

[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.

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.

Do EAs inherit properties from the spawing EA? Regarding thread-specific state, I have already discussed this in Section 2.2. However, this also affects forward progress guarantees. For example, consider two concurrent EAs, each spawning a group of parallel EAs: Can a parallel EA from one group rely on at least one parallel EA from the other group to make progress concurrently? The definition for parallel execution given above does not provide this guarantee.

Nonetheless, it might be natural to assume that it is provided, especially in cases where the concurrent EA becomes part of the parallel EAs or blocks for the parallel EAs to finish their tasks. Thus, do EAs need always inherit some forward progress properties, or is this rather controlled by the specific mechanism used to spawn EAs (e.g., if parallel EAs are spawned and yet the concurrent EA continues to execute)?

A similar question is whether weak forward progress guarantees should be restricted to a specific context, for example, a single parallel loop. This should answer the above question, but it could also constrain the implementation regarding how many resources to use for execution.

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).