P0772R1: Execution Agent Local Storage

Document number: P0772R1 (RAP)
Date: 2018-05-07
Authors: Nat Goodspeed <nat@lindenlab.com>
Michael Wong <michael@codeplay.com>
Paul McKenney <paulmck@linux.vnet.ibm.com>
Jared Hoberock <jhoberock@nvidia.com>
Carter Edwards <hedwards@nvidia.com>
Tony Tye <Tony.Tye@amd.com>
Alex Voicu <avoicu@amd.com >
Gordon Brown <gordon@codeplay.com>
Mark Hoemmen <mark.hoemmen@gmail.com >
Audience: SG1

Reply-to: nat@lindenlab.com

1 Changelog

R0: initial version discussed in ABQ
R1: adjust for feedback from Heterogeneous C++ Google Group and ABQ
"Right of Refusal": exceptions as only mechanism for communicating refusal probably not viable
"Execution Agent Nesting" – recognize that most EAs are based on underlying EAs. Example: Fiber running on a std::thread. If code running on such an EA requests TLS (and it's not able to provide TLS), it shouldn't necessarily fall back to the underlying EA by default.
"Do not limit reuse of TLS" when lifetimes don't overlap.
Difference between thread local and task local.
Is initialization/construction involved on lightweight EA.

2 Abstract

The Concurrency Study Group has long recognized that we have yet to address the interaction between thread-local storage and execution agents finer grained than `std::threads`. This paper is an attempt to explore that space.

In brief:

- This paper proposes that the C++ runtime track a "current execution agent."
- Certain kinds of execution agents are based on underlying execution agents. For example, a fiber depends on an underlying thread in just this way. Therefore, the notion of "current execution agent" is nested. At each layer there is a current agent; at any given step in program execution, there is an innermost current agent.
- In ordinary C++ code, the innermost current agent must be determined *dynamically*. Just as a library function cannot know the nature of the calling application, so it cannot statically determine on what

execution agent (or even on what kind of execution agent) it is running.[1]

- This paper proposes a facility by which a function can request storage local to the innermost current execution agent. Importantly, not every kind of agent will grant that request: an implementer must have the option to refuse.
- The lifespan of that storage, and the circumstances under which it is initialized and destroyed, are implementation-defined.

3 Non-goal

It is our belief that it would be a mistake to alter the current semantics of the `thread_local` keyword. Much code has been written, and will be written before any such change could be published, that would break in that case. More unfortunate still, since the effects would be strongly timing-dependent, many of the resulting bugs would be subtle and intermittent.

That implies that adapting code to use the general execution agent local storage rather than specifically `thread_local` storage requires explicit editing and recompilation.

If there is disagreement on this point, please surface it.

4 Deliberate Omission

This paper does not yet attempt to present a concrete API. If this direction proves to be worth pursuing, the authors hope that the supporting API can be evolved collaboratively.

5 The Problem

Thread-local storage (TLS) has a long history in C++, having been implemented previously by many vendors following GNU TLS. Since C++11, we have added `thread_local` that duplicates much of GNU TLS functionality with a distinct keyword, so as not to collide with GNU name practice. Its various use cases are explored in P0072 and P0097.

TLS made some sense when there might be up to 16 CPUs served by 128 GB of DRAM, such that each thread could have 8 GB of footprint, and reasonable stack space.

The problem is that TLS in its current form did not account for the explosion in massive parallelism, especially in the form of GPUs, where the ratio of GPU streaming units to DRAM can be as much as 5,000-16,000 units to 16 GB. In such cases, not only is each execution agent's stack space extremely limited, any increase in startup latency would also be undesirable, especially if we were to represent each streaming unit as an execution agent per P0072. Finally, P0072, P0097 and P0108 all point out additional problems with TLS.

6 State of the Art

TLS has implementation experience in many languages including pthread, Windows, C, C++, Lisp, D, Java, C#/.Net, Objective C/Pascal, Perl, Python and Ruby. Even OpenMP had TLS called `threadprivate`, which was effectively implemented using the platform TLS once it became available. While some wish to ban TLS, its usefulness remains in many code bases, especially the Linux kernel. As such, outright banning it

[1] Certain C++ statements may be compiled for one specific kind of execution agent. In that case, of course, the nature of that agent can be statically known.

seems out of the question.

The authors definitely do not want to alter the existing semantics of `thread_local`. Rather, they adopt a new facility allowing an application to associate local data with each execution agent, which may be much lighter weight than `std::thread`.

7 Recommended Solution for New Code

For new code, there are a couple of reasonable approaches to providing storage that persists between calls:

- The affected function(s) can be member(s) of a class. Each instance of that class provides storage that persists between calls to member functions.
- Alternatively, each affected function can be written to accept an object it can use for storage that persists between calls, and callers can be required to pass such an object.

These two alternatives are effectively equivalent: the implicit `this` pointer passed to each member function provides the storage for use between calls.
If we were working with a brand-new language without any nontrivial code base, that might suffice.

8 Example Use Cases

Consider a class intended to measure runtime performance at various levels of granularity. The designed usage is to declare a stack instance near the top of each function of interest. Its constructor captures a string label and the start time; its destructor subtracts start time from current time.
To model the hierarchy of levels of granularity, the constructor also links `*this` as the new head of a LIFO list. The destructor unlinks it.
In a single-threaded program, it would be reasonable to use a class static pointer member as the anchor for that linked list.
In a program aware of C++11 threads, the anchor for that linked list would be a `thread_local` pointer.
Mark Hoemmen suggested two additional uses cases:

- use as a simple scratch space with per thread semantics, or
- persistent state to communicate across agents

For finer-grained execution agents, to what should that declaration be changed?

Consider a thread-parallel algorithm that requires some fixed amount of "scratch" memory space per parallel loop iteration. The algorithm really only needs as much scratch space as the number of concurrently executing threads, which could be much less than the number of parallel loop iterations. With thread-local storage, each thread could have "its own scratch space" that it may freely use. Users would then not need to write thread pool implementations.

Consider an agent-based (not the same as execution agents) concurrent application, where agents may send or receive messages to each other. If different threads could access each other's thread-local storage, then agents could use thread-local storage to send or receive messages without needing to copy data. Compare to "zero-copy" implementations of the Message Passing Interface (MPI) distributed-memory programming model that rely on shared-memory hardware.

9 Desired Behavior

Consider a library function `func()` that requires some storage that persists from one call to the next.

Suppose the default thread, the one running `main()`, launches a `std::thread t`. Suppose further that thread `t` launches fibers `f1`, `f2` and `f3`.

When `func()` is called a second time by the default thread, it should obtain the same storage as the previous call from the default thread.

When `func()` is called a second time by fiber `f1`, it should obtain the same storage as the previous call from `f1`. This storage should be distinct from that belonging to the default thread.

When `func()` is called a second time by fiber `f3`, it should obtain the same storage as the previous call from `f3`. This storage should be distinct from that belonging to fiber `f1` or the default thread.

In other words:

When `func()` is called by the default thread, the innermost current agent is the default thread. The storage obtained by `func()` here is thread-local, in the conventional sense.

When `func()` is called by fiber `f3`, although the current thread is `t`, the innermost current agent is fiber `f3`. The storage obtained by `func()` in that case must be fiber-local to remain distinct from the storage local to `f1`.

We expect that if thread `t` calls `func()` *before* launching any fibers, `func()` will obtain thread-local storage. That storage will remain distinct from the storage obtained when `func()` is called by `f1` or `f3`.

This is why the innermost current execution agent must be determined dynamically.

It should be possible to build a given program with dynamic innermost current execution agent tracking disabled. In that case, *every* request for execution agent local storage will be refused.

10 Static versus Dynamic Knowledge

The still-to-be-designed API proposed in this paper is intended to address use cases in which the innermost current execution agent cannot be known statically – or, put differently, in which statically engaging facilities for one specific kind of execution agent would prohibit that code from being called on any other kinds of execution agents.

Consider the case of maintaining a library whose API is cast in concrete. New back-end requirements mandate persisting information from one call to the next. Because the API cannot be changed, you find yourself contemplating `thread_local`, the only existing facility for agent-specific storage. But you are anxiously aware that your library code might be called by all kinds of different execution agents, and you do want to support that kind of usage. How should you address your need for storage that persists from one call to the next, in a way that doesn't prohibit lightweight execution agents?

This paper targets such use cases.

It could be argued that when a given C++ function is compiled for a specific target execution agent – when the generated code cannot be run on any other kind of execution agent – existing static facilities (with distinct names) suffice. On the other hand, in such a case the compiler could be taught to recognize use of the API requested in this paper, and emit code specific to the appropriate execution agent.

But this paper is not primarily focused on the case in which the execution agent can be statically known.

11 Right of Refusal

An implementer of, say, GPU execution agents, or SIMD lanes, might refuse to provide storage local to an execution agent instance.

This argues against introducing a declaration keyword such as `agent_local`. The API by which this storage is requested must permit the implementation not to support local storage at all.

However, when the innermost current execution agent refuses to support local storage, it is important that the API not fall back to the next outer layer of execution agents. Storage private to the innermost current execution agent is semantically different from storage shared with other execution agents at that innermost layer; transparently substituting the latter for the former would be error-prone at best.

(It is plausible to imagine a separate request API that *does* support fallback, if that's really acceptable to the requesting code.)

12 Storage on Demand

The cost of constructing execution agent local objects constitutes another argument against a declaration keyword. One of the present difficulties with `thread_local` is the implication that all such declarations must be initialized upon launch of any new thread, and destroyed upon thread termination. It has been pointed out that even a thread that does not access a given `thread_local` declaration must still pay for its construction and destruction.

Instead, the request API should take the form of a function call or class constructor (though refusal, in the latter case, would seem to require an exception). Only storage explicitly requested would be constructed. Only that storage would be destroyed.

We might consider something like the `boost::thread_specific_ptr` API.

We further consider the idea of a standardized allocator type that accepts a size argument.

If all three of these conditions hold:

- an instance of that allocator is declared in class scope
- the declaration is marked agent-local, by whatever means we decide
- its constructor argument is constexpr

then the constructor argument to the allocator is added to the total agent-local storage to be preallocated for that class, for an execution context that requires preallocation.

It is up to the code to pass that allocator instance to any standard library type (e.g. std::vector) that may allocate dynamic storage. It is not sufficient to mark a declaration of a std::vector as agent-local, without passing an instance of the special allocator: that would provide in agent-local storage a std::vector instance whose dynamic storage is nonetheless obtained by operator new().

Significant questions still remain in terms of how does it get the object?

(1) Specially tagged class member or

(2) passed in to the operator().

Carter prefers approach (2) because it requires less machinery to implement especially when using lambdas. Another orthogonal design question is: what kind of memory resource? Stack-based allocation is the lightest and most performant, but least flexible.

13 Implementation-Defined Lifespan

It is important to permit the implementer of a given category of execution agents to determine (and document) the circumstances under which execution agent local objects are destroyed and reconstructed.

For threads and fibers, it may be acceptable to destroy local objects on termination of the agent.

An implementer of very lightweight agents might reasonably take the position that the storage provided by the API will not be shared by any other currently-running agent – but that storage might be "inherited" from a previously-terminated agent, and thus contain non-initial values.

We also need to clarify the differences between work and execution agent in the lifecycle of a callable with agent-local storage requirements. Carter Edwards defines it as:

- construction (lambda capture)
- submission to an execution context
    - Request net amount of agent-local-storage at submission?
    - Might not be fulfillable request, should fail now if cannot
    - Might influence scheduling
- waiting in the execution context to be executed
- State transition waiting->executing
    - Execution agent is attached / associated; permissible to have happened earlier
    - Must all of this agent's agent-local-storage be ready / pre-allocated before execution?
    - May pre-reallocate the agent-local storage that net amount of requested storage is available before execution begins? (for some environments it may be advantageous to pre-allocate while others will prefer deferred allocation)
- being executed on an execution resource by an execution agent
- State transition to executing->complete
    - Agent-local-storage can be released
    - Execution agent can be detached
- complete
- destruction

This life cycle is important to preserve "agent local storage" across swap out/in, and the work item with it.

14 Storage Sharing

Calling the API to request execution agent local storage should either provide that storage, or refuse. Supposing that storage is provided, is it permitted to pass a pointer or a reference to such an object to some other execution agent at the same layer?

One is tempted to answer "of course" – but perhaps this, too, should be an open question.

If the restriction is implementation-defined, how would we prohibit passing a pointer across execution agent

instances?

If the restriction applies to all implementations, enforcement would seem to depend on the details of the API. A class resembling a container (e.g. `agent_local<T>`) could refuse to support `operator&`. But with a class resembling a smart pointer (e.g. `agent_local_ptr<T>`), that would be harder: consider `(&*ptr)`.

Related question: should there be an API by which code running on a given execution agent could retrieve storage belonging to some *other* execution agent?

The answer to the second question is less clear. That presupposes some sort of agent ID; not all providers of execution agents will want to guarantee distinct agent IDs. (On the other hand, perhaps any execution agent implementation willing to provide agent-local storage would also be willing to provide distinct agent IDs.)

More thorny: are all execution agent IDs of the same type? Or should the ID type itself be distinct for each execution agent implementation? Is it permitted for code running on (e.g.) a fiber to request storage local to (e.g.) some other thread? If that distinction isn't indicated by the type of the ID, how should it be indicated? Must we also add an execution agent implementation ID?

If there is a mechanism by which one agent can retrieve local storage for some other agent, it should at least be easier for an implementation to decide whether to support or refuse such access.

15 Migration

Consider a fiber `f1` running on thread `t1`. In some hypothetical future, it might be possible to migrate a fiber from one underlying thread to another. What happens if fiber `f1` is migrated to thread `t2`?

We submit that in that case, fiber `f1` must continue to find its original fiber-local storage even though it is now running on a different underlying execution agent.

16 API Speculation

Discussions of this idea to date have surfaced a number of interesting questions, some of which are touched on below. This section explains why this paper does not yet propose a concrete API.

- A declaration API analogous to `boost::thread_specific_ptr`, on the face of it, would seem to require an exception to indicate refusal to provide execution agent local storage. Should such a class have an API more like `optional<thread_specific_ptr>`? Or perhaps a `valid()` method which must be checked before any attempt to dereference?
- Should the request API literally require a factory-function call, perhaps returning a `pair<pointer_type, bool>`?
- A smart pointer type can be initialized to `nullptr`; if the caller wants to `reset()` it to something other than `nullptr`, the caller must explicitly construct the referenced object. That still implies that at a time defined by the execution agent implementation, any non-`nullptr` value belonging to an agent must be `deleted`. Must this facility support a custom deleter function?
- What about non-pointer types? Should the API look more like `agent_local<T>`? What restrictions, if any, would we want to place on `T`? Must it be intrinsic? Must it be pointer-sized? Must it be trivially constructible and destructible?
- Should the restrictions on `T` be determined by the API, or more specifically by each execution agent implementation? A given call to the request API might engage different execution agent implementations at runtime – therefore if the restrictions vary by execution agent implementation, `T`

cannot be validated at compile time. Apparently that means that the implementation would refuse to provide storage for any type it cannot support. In that case, should there be an error code of some sort to allow a caller to distinguish between "execution agent implementation does not support agent-local storage at all" and "execution agent implementation does not support the requested type"?

- If we allow non-pointer `T`, any attempt to access that object must necessarily instantiate it – or throw an exception. (This could be a convenience API for people willing to risk an exception.) That must include binding a reference to the instance. Do we implicitly instantiate every `agent_local` declaration within a given scope? Or do we try to preserve lazy instantiation, but require front-end support to intercept binding a reference?

Given the open questions, the authors presently favor an API specific to pointers – but would be delighted to accept concrete solutions to these issues.

17 Implementation Speculation

None of this section should be taken as definitive. It is merely a thought experiment to demonstrate that the proposed functionality should be implementable. The authors do not claim that the thought experiment is in any way optimal.

In the present C++ execution model, a kernel thread – whether implicitly launched by the operating system, or explicitly by application code – is the execution agent underlying all other agents.

`static` objects provide process-local storage, as it were.

`thread_local` objects support the next inner layer of execution agents.

It seems plausible that a linked list of objects representing execution agent layers could be anchored with a `thread_local` pointer.

It is important to have an extensible underlying mechanism permitting new implementations of execution agents and their associated storage support (if any). Perhaps the Standard might specify an implementer-facing base class from which an implementation must derive its own representation of its execution agent layer. That base class could define virtual methods to request execution agent local storage. Then the application-facing storage request function could simply invoke the virtual method on the current stack top.

One could imagine an implementer-facing template function such as: "If my class is already on the stack of execution agent layers, return its pointer. Otherwise, construct an instance and link it as the new stack top." An execution agent implementation could engage this function when launching a new agent, which might or might not be the first at that layer.

Once an application has launched fibers within a given thread, all subsequent code on that thread is logically running on one of those fibers. On the other hand, running code on SIMD lanes does not mean that all subsequent code continues running on SIMD lanes. If a SIMD implementation were to push a new execution agent layer representation, it must be able to pop it. On the third hand, it might be unnecessary for a SIMD execution agent implementation to do either: perhaps the correct behavior for SIMD can be determined at compile time.

Naturally, context switching at a given layer from one execution agent to another must update that layer's representation object in such a way that subsequent storage requests will obtain storage associated with the new current agent.

However, we need not intercept context-switching between `std::threads`: anchoring the layer stack with a `thread_local` pointer implicitly takes care of that.

18 Questions and Straw polls

We remain unsure whether to introduce a keyword for agent-local variable declarations. We want some way to constrain the set of agent-local declarations that the runtime is required to support. Using class scope is an attempt to address that problem. Please suggest better ways.

Should we introduce an agent-shared annotation as well as agent-local? On GPUs there can be storage that is only accessible by a single execution agent that may be desirable to use due to performance.

Would it be UB to access this storage from a different execution agent to allow an implementation to use special hardware if available?

When someone wants to be able to execute existing code on a newer execution agent -- but that code uses thread_local or static storage -- what's WG21's recommendation for replacing thread_local? Rewrite the entire call chain?

Should we have agent-shared between all execution agents or separate ALS into local and static variants to support dynamic behaviour.

How is the amount of agent-local-storage is decided, and how it would be accessed in the execution agent. For example, does the user specify a keyword on the variables they want to have agent-local-storage, and then access the storage by that name? Or is there a single allocation that can be specified and a language intrinsic to access that single allocation?

How can the compiler limit which declarations it has to consider to determine the size of agent-local-storage for a specific kernel. In the presence of indirect function calls, and dynamically loaded libraries, how would that work? A function pointer could be passed into the callable which may be defined in another dynamically loaded library, which declares agent-local-storage.

How the compiler will decide on how to address the agent-local-storage. If it is defined in multiple functions that can be called by disjoint kernels, then would the placement in the agent-local-storage area need to agree amongst all the kernels? If so then that may mean the compiler effectively has to allocate space for all agent-local-variables whether they are used by a kernel or not in order to get a consistent layout.

Many current implementations of kernel compilers rely on seeing the full set of sources of a kernel, and may even perform full inlining which can avoid issues of sharing the code of a function between kernels. But I am unclear if the restrictions these impose are acceptable here.

19 Abbreviation

The authors hope that we collectively refrain from using “ALS” to describe this proposed feature. We would

also avoid “agent-specific storage” due to unfortunate initials. Perhaps “per-execution agent local storage,” i.e. PEALS? EALS? Or perhaps XLS – though the latter is already overloaded.

20 Acknowledgements

This paper is based on discussions between the authors and JF Bastien. Thanks to the discussion held in the Heterogeneous C++ Google Group.

21 References

boost::thread_specific_ptr	www.boost.org/doc/libs/release/doc/html/thread/thread_local_storage.html
P0072	Light-Weight Execution Agents http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0072r1.pdf
P0097	Use Cases for Thread-Local Storage http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0097r0.html
P0108	Skeleton Proposal for Thread-Local Storage (TLS) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0108r1.html
XLS	https://en.wikipedia.org/wiki/XLS
Heterogeneous C++ Google Group	https://groups.google.com/forum/#!forum/hetero-c