Document number: P2079R0

Date: 2020-01-12

Audience: SG1 (Concurrency and Parallelism)

Authors:

- Ruslan Arutyunyan <Ruslan.Arutyunyan@intel.com>
- Michael Voss <MichaelJ.Voss@intel.com>

Shared execution engine for executors

Motivation

`static_thread_pool` is a convenient way to create an execution context in place where you need it. However, including it as the sole, standard way to obtain execution resources on the host may promote bad practice that will inadvertently lead to oversubscription and poor composability. In this section, we outline some of the weaknesses of `static_thread_pool` and describe characteristics that an alternative execution context should possess to avoid these weaknesses. We are not arguing for the removal of `static_thread_pool` from P0443 but instead to complement it with at least one additional choice.

The first issue with `static_thread_pool` is that it can easily lead to oversubscription. Real-world applications are complicated and, in general, link with many third party libraries. Any shared object (.so) as well as an application itself may create its own `static_thread_pool`. Without alternatives in the standard, this might in fact seem like the only portable choice. However when there are many `static_thread_pool` instances, the end application will likely, inadvertently request more threads than the number of physical cores available in the hardware, oversubscribing the hardware. For compute-intense workloads, oversubscription often leads to poor performance.

The second problem is that even if an alternative context is specified, there is no clean way to composably construct software to use it. Imagine that an application wants to use OpenMP as the main parallel context and wants the third party libraries that it calls to also use OpenMP for parallelism. How does an application share this choice with a third party library? An obvious solution is for the third party library to support executors through its interfaces, but even so, the current proposals for executors would require applications to pass executors or execution contexts through the function parameters. But it may not always be possible or convenient to pass the execution context through the call stack. The executors are not the option since querying the context is optional but (as it will be shown below library needs to get properties of the execution context and it needs them at run-time (if it's .so). Of course it would be possible to support only `static_thread_pool` instances, but as we've already argued, `static_thread_pool` should not be the only choice. And even if it were the only choice. The idea is to support the arbitrary execution context. Thus, we need a standard interface for it: either virtual interface for execution context or polymorphic wrapper over it.

Let's consider an example when there are many shared libraries, and where some of them support executors and some of them do not. Imagine that we have an application that uses a library like Threading Building Blocks (TBB) that (in the future) provides support for executors and another math library that does not support executors but uses TBB internally; an example of such a library is the Intel Math Kernel Library (Intel MKL). Assume that part of TBB's support for executors includes a way to specify the executor for algorithms like `parallel_for`. If the application wants to change the executor for all `parallel_for` algorithms (to add affinity, priority, etc.), it can easily do this for its direct uses of `parallel_for`. But it can only change the executors used for the `parallel_for` algorithms in the math library if that library provides an interface to do so. The use of executors in the main application breaks composability, unless there is a standard way to communicate choices to nested uses. There should be a mechanism for a nested level to acquire a default execution context or the execution context it should use in its current level.

Finally, if we expect that there will be alternative execution contexts beyond `static_thread_pool`, additional execution properties are required so that nested parallelism can make proper run-time decisions

about the use of an execution context. As a motivating example, again imagine that we have application that uses TBB algorithms at the outer application level and a math library like MKL that uses TBB algorithms internally. By default, the TBB algorithms use a work-stealing task scheduler that typically leads to good nested composability. When using defaults, both the application and the MKL library algorithms can use TBB algorithms without being aware of each other and expect good performance. But what if a developer changes the default context for TBB algorithms to one that does not supported nested parallelism well? For example, they might choose to use OpenMP to execute TBB `parallel_for` algorithms (OpenMP has known composability issues but is often preferred in HPC due to its lower static scheduling overheads). If the application and library use the same non-default execution context for TBB algorithms, and the execution context does not support nested parallelism well, the library developer needs to know the context of outside execution at run-time to make a correct decision about whether to use nested parallelism at all. Knowing about the outer execution context may be necessary to avoid oversubscription, dead (live) locks. etc. Even if an execution context correctly supports nesting, it may do so inefficiently and so knowing the context may be critical from performance perspective.

Proposed Direction

To solve the oversubscription problem we propose to introduce `shared_execution_engine`. It should have shared state and provide interfaces through free functions to get executors, schedulers and other useful objects. It should be lazy initialized on the first call to any function that relies on the default execution context getter. This instance should be created implicitly and have `std::thread::hardware_concurrency` execution agents underneath. A getter of a `shared_execution_engine` will always return the same instance. Semantically, it's a singleton. Once it's created it stays alive as long as the process exists.

To simplify the communication of execution contexts to allow better composability with libraries, we also need a mechanism to substitute the default execution context. As was described in motivation section, passing the execution context through the function parameters is not convenient and, furthermore, not always possible. We therefore propose to introduce a global execution context and an API to set and get it. Using these APIs, functions that don't accept executors via parameters may just use the global getter to read the default execution context and run the algorithm using it. Semantically, this is very similar to `get/set_new_handler`. If there is a default context, there must also be either a virtual interface for execution contexts or a polymorphic wrapper over any execution context. It would be also great to have an execution context concept to assist in the implementation of custom execution contexts.

Finally, if there is a default execution context that we can get through a free function, additional properties may be necessary to get best performance when using. In particular, we may need to get information about the enclosing context of the current execution and we expect them to be available at run-time. We are considering three possible properties:

- `nesting_t`: As we wrote in the motivation section, one very useful property is the know about an executor's support for nesting parallel execution. We need this knowledge at run-time, so can we call (or not) another level of parallelization. Nested property types might include `nesting_t::supported` and `nesting_t::unsupported`.
- `outer_bulk_guarantee_t`: A property that is similar to the execution policy semantics in algorithms, with nested property types like `outer_bulk_guarantee_t::sequenced`, `outer_bulk_guarantee_t::unsequenced`, `outer_bulk_guarantee_t::parallel` and `outer_bulk_guarantee_t::parallel_and_unsequenced`. A library may want to know from the information of outer level it was called. For example, the library may want to create parallelism and if it knows that there is no parallelism in the outer level and so call executors APIs without restrictions. On the other hand, if the outer level already is `outer_bulk_guarantee_t::parallel` it might choose differently, for example:
    - If the execution context supports nesting and library has enough amount of the work for parallelization it may create additional parallel tasks,
    - Otherwise, if nesting is supported but the work is not enough for the parallelization, the library may call the algorithm sequentially, or vectorize it.

- `nesting_level_t`: A property that provides the current nesting level. Sometimes it important to know if execution is at the outermost level or contained within other parallel levels. We might propose to introduce something like `nesting_level_t::outermost_level` but the nesting level might be defined more generally.

Next steps

We are ready to prepare the API for the instances that we described in this paper that solve the problems listed above.

We are seeking feedback to see if standard committee agrees that the problems in this paper are important and we should spend more time preparing complete solutions to them.