Doc No: P0335R1
Date: 2018-10-07
Audience: SG1 (parallelism & concurrency)
Authors: Pablo Halpern, Intel Corp.
phalpern@halpernwightsoftware.com

Context Tokens for Parallel Algorithms

Contents

1 Abstract

This paper proposes a mechanism whereby the *element-access function* of a parallel algorithm (i.e., the function or lambda that is invoked in parallel) can invoke specific operations provided by the execution policy. The proposed mechanism takes the form of a *cookie* or *token* passed from the algorithm to the function. The type of this token can be different for each execution policy. Acceptance and use of this token is entirely optional, allowing simple use cases to remain simple and backwards-compatible with the existing parallel algorithms library. This model of using tokens is similar to the *execution agents* in the Agency library.

This paper is at the conceptual stage – no formal wording is provided.

2 Changes from R0

Formal wording has been removed. This paper is more conceptual now, focused on what should go into the context token and what algorithms would benefit from it.

3 Motivation

Consider the following use of the parallel `for_loop` algorithm with the `vector_policy` described in the Parallelism TS v2:

```
for_loop(execution::vec, 0, N, [&](int i){
    ++ordered_update(histagram[A[i]]);
});
```

The `ordered_update` function is specifically tied to the `vec` execution policy, yet there is no syntactic connection between them. Replacing `vec` with `par` would render this code incorrect – undefined behavior. It would be better that such a substitution render the code ill-formed, but the library syntax does not give us a good way to express such a *syntactic* restriction.

Although the `vector_policy` is the first demonstrated example of this kind of problem, the problem will not remain limited to the `for_loop`, nor to the `vector _policy`. For example, a future enhancement of the parallel execution policy might provide support for critical sections or thread-local storage in a way that is safer and/or more efficient than the direct use of `mutex` and `thread_local`. As we move towards combining execution policies with executors, it might also be desirable to query the executor.

4 Proposal overview

This proposal is targeted at a version of the parallelism TS.

What is proposed is that each parallel algorithm may pass a special token that communicates execution context to the element-access functions passed to the algorithm by the user. Policy-specific operations and queries, rather than being free functions, would be member functions of the token object. The above example would be rewritten as follows:

```
for_loop(execution::vec, 0, N, [&](auto context, int i){
    ++context.ordered_update(histagram[A[i]]);
});
```

The token can carry information about the execution policy, the iteration of the loop, etc.. The type of this context token is defined as a nested type within the execution policy:

```
struct vector_policy {
    struct context_token { ... };
    ...
};
```

Note that if the execution policy is modified by an executor (e.g., using the `.on(executor)` syntax), then the context token is likely to have a different type than if the execution policy is used directly. If the token is not needed, it can simply be omitted from the argument list for the element-access function:

```
for_each(execution::vec, 0, N, [&](int i){
    A[i] = A[i + 1] + 10;
});
```

Thus, existing uses of the parallel algorithms are unaffected by the context token. This flexibility is enabled through the use of metaprogramming to invoke the element access function either with or without extra initial argument. If the invocable object has overloads both with or without the extra argument, the overload with the extra argument is preferred.

5 What is a context token?

5.1 General

A context token is an *object* encapsulating state about the current context of an element execution function, including general information such as the type of execution agent and policy-specific information such as the current worker ID. The class of the context token might also have typedefs for such things as the executor type and member functions (both static and non-static) for interacting with the context, e.g., for getting the current executor or SIMD lane or for executing a critical region in a way that is appropriate to the execution agent.

5.2 A context token is *not* an execution policy

An execution policy such as `parallel` or `unseq` is certainly part of the context in which the element access functions are run, but it does not encapsulate *invocation-specific* context such as the current thread-pool worker.

5.3 A context token is *not* an executor

An executor is an object on which to enqueue work. A context token is created *after* the executor has launched the work. As with the execution policy, the executor does not encapsulate information about the specific invocation of the launched task. The executor might be available through the context token for the purpose of enqueuing more work.

5.4 What goes into a context token?

It might make sense to define a `ContextToken` concept. The concept would define a (probably small) number of types and operations that would apply any context token type. There might also be *optional* members of the type (which necessarily cannot be part of the concept). These optional members would have the same meaning for contexts that support them, but would not be defined for other contexts.

In addition, each context token type is likely to have context-specific types and operations that would be usable only be element access functions that are tuned to a specific context type. For example, a kernel that is intended to run on a GPU might access the current warp from the context token. Such a kernel could not run on a CPU thread pool, whose context token would not be able to provide a warp. This lack of portability is often acceptable for highly-tuned code.

5.4.1 Some common members of context tokens

Types

- Execution policy type

- Executor type
- Exception support (`true_type` and `false_type`? Enumeration?)
- Native memory allocator
- Mutex type (optional)

Member functions

- `get_executor()`
- `get_worker_id()` (optional - worker ID for thread pools, lane number for SIMD, etc.)
- `get_iteration()` (optional - iteration number of a loop)
- `critical_section()` (optional - run a lambda protected from other parallel executions in the same executor)
- `barrier()` (optional)
- Access to worker-local storage (optional)
- Some way to perform reductions?
- Quit with an error or exit a speculative execution (e.g., cooperative cancelation of parallel iterations) (optional)

5.4.2 Some possible members of executor- and policy-specific context tokens

vec policy context member functions

- `ordered_update()`
- `vec_off()`
- `vec_lane()`

thread-pool context member functions

- `get_cpu_id()`
- Etc.

6 How does a context token become available to executing code?

6.1 Optional extra arguments to element access functions

The user-supplied callbacks to the parallel STL algorithms are called element access functions. With some metaprogramming the algorithm could determine if the element access function accepts a context token argument and, if so, generates and passes the token. Examples of element access functions with and without context-token arguments are shown in the proposal overview (section 4), above. The context would be passed by value, since parallel executions would need different context tokens. However, pointers, could be used to make them small and efficient to pass around.

For this mechanism to be adopted, we must decide which element access functions can participate and which cannot. The possibilities are

1. All element access functions, including comparison functors, etc.. Passing an optional context token to each of these would be a lot of work for the algorithm implementers. Also, it could not be threaded through things like fancy iterators.

2. Element access functions named `Function` in the standard. There are only a handful of algorithms to which this applies, including `foreach` in the standard and `for_loop` in the Parallelism TS. This was the proposed categorization in R0 of this paper.

3. An even narrower range of element access functions limited to just the overloads of `for_loop`. The thinking was that this is a very low-level algorithm. However, it means that no other algorithms would have access to the context, not even `foreach`.

4. All of the element access functions named `Function` as well as a select few others, such as the `UnaryOperation` functor in the `transform` algorithm. This is my current favorite.

6.2 Optional argument to the task queued on an executor

This would untie the idea from algorithms, to a degree. The same metaprogramming principle applies as for algorithms, where the argument is provided only if asked for. Conversely, we could simplify the spec by making the context token *always* passed into the task, at the cost of making it slightly more complicated to author tasks.

6.3 Optional extra argument to a task block

The argument to `define_task_block` has many of the qualities of an element access function. It would make sense to pass an executor to `define_task_block` and for `define_task_block` to pass a context token to the block itself. Similarly, the `run_task` member of the task block could pass a context token to the spawned task.

6.4 Agent-local storage

This is the most obvious out-of-band mechanism for making the context token available. Unfortunately, it can result in a catch-22 whereby the task cannot determine how to get agent-local storage without already having a context token to query. Some forms of agent-local storage might be expensive and some executors would not be able to provide agent-local storage at all.

6.5 Yet-to-be invented out-of-band mechanism

Pablo Halpern is working with Alisdair Merideth and Andrew Sutton to try to define a quasi out-of-band syntax and implementation mechanism for environmental attributes of which a context token would be a prime example (allocators being another example). Unfortunately, this work is still in the “I have an idea” phase, so it is not clear whether it will affect SG1 decision making in the near term.

7 References

Agency *Quick Start Guide*, a low-level library for abstracting parallel execution, Jared Hoberock, 2015-12-03

N4755 *Parallelism TS v2 working draft*, Jared Hoberock, editor, 2018-06-24