Scoped thread-local storage

Abstract
This paper proposes a library facility for scoped thread-local storage, designed for parallel algorithms when there is no one-to-one mapping between input and output.

Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>Proposed</th>
</tr>
</thead>
</table>
| ```cpp
span<Triangle> input = ...;
double max_area = ...;
mutex m;
unordered_map<thread::id, vector<Triangle>> tmp;
```                                                                 | ```cpp
span<Triangle> input = ...;
double max_area = ...;
tls<vector<Triangle>> tmp;
```                                                                 |
| ```cpp
//process in parallel
for_each(execution::par, input.begin(), input.end(),
    [&](const auto & tria) {
        //get thread-specific storage
        auto & ref = tmp.local();
        //generating unbounded output
        for(const auto & t : split(tria, max_area))
            ref.emplace_back(t);
    });
```                                                                 | ```cpp
```                                                                 |
| ```cpp
//post-process results sequentially
for(const auto & tria : tmp | views::join)
process(tria);
```                                                                 | ```cpp
```                                                                 |
| ```cpp
//clear content for future parallel processing
tmp.clear();
```                                                                 | ```cpp
```                                                                 |

Revisions
R0: Initial version

Motivation
C++17 introduced parallel algorithms to the standard library. The design of said algorithms embodies the popular fork-join model of parallelization. Combining this structured parallelization style with the functional aspects of the “STL” was a perfect match for querying (e.g. `std::find`), in-place transformations (e.g. `std::sort`), and one-to-one transformations (e.g. `std::transform`).

One class of algorithms the standard library never supported (apart from “abusing” `std::for_each`) were one-to-many transformations. Applying the fork-join model to these

---

1 RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
algorithms proves to be difficult as their unbounded nature doesn’t lend itself easily to aggregating the results in a singular target object without overt locking.

If no singular result object is needed, the issue of locking could be sidestepped by the usage of thread_local variables - but such an approach has extensive hidden costs for all threads and transforms a local issue into a global problem.

We propose an alternative approach providing lazily initialized, thread-specific storage bound to a local object. The proposed std::tls does not require expensive locking for concurrent write access, nor does it increase the static memory set of a thread.

## Design Space

A std::tls<T> composes a concurrent storage for thread-specific instances of T and a concurrency-safe initialization function for storage entries. Conceptually it is similar to the following class:

```cpp
template<typename T, typename Allocator = allocator<T>>
class tls {
    mutex m;
    unordered_map<thread::id, T, hash<thread::id>, key_equal<thread::id>, Allocator> storage;
    //NOTE: at the moment none of the standard polymorphic function wrappers has allocator support.
    unmovable_function<Allocator, T() const> init_func;

public:
    // (1) constructors
    tls(Allocator alloc = Allocator{}) noexcept requires is_default_constructible_v<T>;
    tls(T value, Allocator alloc = Allocator{}) requires is_copy_constructible_v<T>;
    tls(auto func, Allocator alloc = Allocator{}) requires is_convertible_v<T, invoke_result_t<decltype(func)>>;

    // (2) not copy- nor moveable
    tls(const tls&) = delete;
    auto operator=(const tls&) -> tls& = delete;

    // (3) modifiers
    [[nodiscard]]
    auto local() -> tuple<T&, bool>; //thread-safe!
    void clear() noexcept;

    // (4) iteration support
    class iterator { … };
    static_assert(forward_iterator<iterator>);
    auto begin() -> iterator;
    auto end() -> iterator;
};
```

Implementations should use more efficient synchronization mechanisms than locking - our reference implementation employs atomic operations to represent storage similarly to a concurrent hash map with separate chaining.

## Constructors

A std::tls provides several constructors to specify the initial value a storage entry should be initialized with. Internally, initialization is handled via an initialization function:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Initialization function</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls();</td>
<td>[] { return T{}; }</td>
</tr>
<tr>
<td>tls(T value);</td>
<td>[value{move(value)}] { return value; }</td>
</tr>
<tr>
<td>tls(auto func);</td>
<td>func</td>
</tr>
</tbody>
</table>

Based on our usage experience with similar classes, we propose that std::tls should be neither copy- nor movable. Our design rationale for this is as follows: We consider copying a thread-local storage to be semantically ill-formed. Whilst moving the storage entries would be possible, it was never necessary for us. Furthermore, it introduces additional implementation overhead in the form of a moved-from state. Such a state would either have to be equivalent to the empty state or have
to be checked against on every call to `local`. Depending on the selected implementation strategy for storage this introduces considerable complexity and may require allocating on move.

Additionally, the proposed `std::tls` internally wraps a *type-erased polymorphic function wrapper*. There are unresolved technical issues concerning allocator support in this category of classes, that led to its removal from `std::function` ([P0302]). Recent classes in this category (`std::move_only_function` and `std::copyable_function`) don't provide allocator support either.

### Modifiers

`std::tls` offers two modifiers: `local` and `clear`. The former acts as a getter and retrieves the storage entry of the current thread. If no such element exists, it's constructed using the initialization function. In addition to a reference of the storage entry a `bool` flag is returned that indicates whether the element was constructed during this call. This diverges from the established practice\(^2\) that provides said `bool` flag as an optional output parameter (employing overloading). Given the recent spotlight on C++ safety, we consider the `tuple` to be superior as it prevents the usage of potentially uninitialized parameters.

`clear` removes all previously created storage entries - a cleared `std::tls` is considered semantically equivalent to a newly constructed one.

### Iteration support

Contrary to established practice we don't propose reduction operations to combine the thread-local values into a final result. Instead we propose to add iteration support, enabling users to post-process computation results with any STL-style algorithm of their choice.

### Impact on the Standard

This proposal is a pure library addition.

### Implementation Experience

The proposed design has been implemented at [https://github.com/MFHava/P2774](https://github.com/MFHava/P2774).

### Proposed Wording

Wording is relative to [N4944]. Additions are presented like this, removals like this and drafting notes like this

```cpp
#define __cpp_lib_tls YYYYMM //also in <tls>
[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]
```

### [thread.general], extend Table [tab:thread.summary]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>[futures]</td>
<td>Futures</td>
</tr>
<tr>
<td>&lt;future&gt;</td>
<td></td>
</tr>
<tr>
<td>[thread.storage]</td>
<td>Thread-local storage</td>
</tr>
<tr>
<td>&lt;tls&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Thread-local storage

General

Thread-local storage provides scoped thread-local storage. Every thread is lazily assigned an associated storage entry. The lifetime of storage entries is decoupled from the lifetime of the respective thread and instead bound to the containing object.

Note: The storage can outlive the respective thread or be destroyed before the thread of execution has ended.

Header <tls> synopsis

```cpp
namespace std {
    // [thread.storage.tls.class], class template tls
    template<class T, class Allocator = allocator<T>> class tls;

    namespace std {
        template<class T, class Allocator = allocator<T>>
        class tls {
            public:
                using iterator = implementation-defined;
                using const_iterator = implementation-defined;

                // [thread.storage.tls.ctor], constructors, and destructor
                explicit tls(const Allocator & allocator = Allocator());
                explicit tls(const T & value, const Allocator & allocator = Allocator());
                explicit tls(T && value, const Allocator & allocator = Allocator());
                template<class Func>
                explicit tls(Func init, const Allocator & allocator = Allocator());

                tls(const tls&) = delete;
                tls& operator=(const tls&) = delete;
                ~tls();

                // [thread.storage.tls.mod], modifiers
                [[nodiscard]] tuple<T&, bool> local();
                void clear() noexcept;

                // [thread.storage.tls.iter], iteration
                const_iterator begin() const noexcept;
                iterator begin() noexcept;
                const_iterator cbegin() const noexcept;
                iterator cbegin() noexcept;

                const_iterator end() const noexcept;
                iterator end() noexcept;
                const_iterator cend() const noexcept;
                iterator cend() noexcept;
        };
    }
}
```

The tls class template provides thread-specific storage of type T with a custom initialization function.

T shall be a cv-unqualified type that meets the Cpp17Destructible requirements ([tab:cpp17.destructible]).

Allocator shall be a cv-unqualified type that meets the Cpp17Allocator requirements ([allocator.requirements.general]) and can be safely used concurrently.

tls::iterator meets the forward iterator requirements ([forward.iterators]) with value type T.

tls::const_iterator meets the requirements of a constant iterator and those of a forward iterator with value type T.

Recommended practice: Implementations should avoid high synchronization overhead for concurrent access to storage.

Constructors, and destructor

```cpp
explicit tls(const Allocator & allocator = Allocator());

Constraints: is default constructible v<T> is true.
```

```cpp
explicit tls(const T & value, const Allocator & allocator = Allocator());

Constraints: is copy constructible v<T> is true.
```

```cpp
tls([] { return T(); }, allocator);
```

```cpp
tls({= [](return value; }, allocator);
```
explicit tls(T && value, const Allocator & allocator = Allocator());

Constraints: is_copy_constructible_v<T> is true.

Effects: As if by:

```cpp
    tls(value = std::move(value)) { return value; }, allocator);
```