Concurrent object pool
(was: Scoped thread-local storage)

Abstract
This paper proposes a concurrent object pool, designed as cache for parallel algorithms lacking a straightforward one-to-one mapping between input and output.

Tony Table

<table>
<thead>
<tr>
<th>Before</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 span&lt;Triangle&gt; input = ...;</td>
<td>span&lt;Triangle&gt; input = ...;</td>
</tr>
<tr>
<td>2 double max_area = ...;</td>
<td>double max_area = ...;</td>
</tr>
<tr>
<td>4 //split triangle mesh based on max triangle size</td>
<td>//split triangle mesh based on max triangle size</td>
</tr>
<tr>
<td>5 mutex m;</td>
<td>object_pool&lt;vector&lt;Triangle&gt;&gt; tmp;</td>
</tr>
<tr>
<td>6 deque&lt;vector&lt;Triangle&gt;&gt; tmp;</td>
<td>//process in parallel</td>
</tr>
<tr>
<td>7</td>
<td>for_each(execution::par, input.begin(), input.end(),</td>
</tr>
<tr>
<td>8 //process in parallel</td>
<td>[&amp;](const auto &amp; tria) {</td>
</tr>
<tr>
<td>9 for_each(execution::par, input.begin(), input.end(),</td>
<td>const lock_guard lock(m);</td>
</tr>
<tr>
<td>10 [&amp;](const auto &amp; tria) {</td>
<td>if(tmp.empty()) return {}; //need new object</td>
</tr>
<tr>
<td>11 //extract exclusive object</td>
<td>auto val(move(tmp.front()));</td>
</tr>
<tr>
<td>12 auto object[[&amp;] -&gt; vector&lt;Triangle&gt;] {</td>
<td>tmp.pop_front();</td>
</tr>
<tr>
<td>13 const lock_guard lock(m);</td>
<td>return val;</td>
</tr>
<tr>
<td>14 if(tmp.empty()) return {}; //need new object</td>
<td>();</td>
</tr>
<tr>
<td>15 auto val(move(tmp.front()));</td>
<td>auto &amp; object=*handle;</td>
</tr>
<tr>
<td>16 tmp.pop_front();</td>
<td>//generating unbounded output</td>
</tr>
<tr>
<td>17 return val;</td>
<td>for(const auto &amp; t : split(tria, max_area))</td>
</tr>
<tr>
<td>18 ();</td>
<td>object.emplace_back(t);</td>
</tr>
<tr>
<td>19</td>
<td>//make object available again</td>
</tr>
<tr>
<td>20 //generating unbounded output</td>
<td>const lock_guard lock(m);</td>
</tr>
<tr>
<td>21 for(const auto &amp; t : split(tria, max_area))</td>
<td>partial.emplace_back(std::move(object));</td>
</tr>
<tr>
<td>22 object.emplace_back(t);</td>
<td>}</td>
</tr>
<tr>
<td>23</td>
<td>//~handle() makes object available again</td>
</tr>
<tr>
<td>24 //make object available again</td>
<td></td>
</tr>
<tr>
<td>25 const lock_guard lock(m);</td>
<td></td>
</tr>
<tr>
<td>26 partial.emplace_back(std::move(object));</td>
<td></td>
</tr>
<tr>
<td>27 }</td>
<td></td>
</tr>
<tr>
<td>28 }</td>
<td></td>
</tr>
<tr>
<td>30 //post-process partial results sequentially</td>
<td>for(const auto &amp; t : tmp</td>
</tr>
<tr>
<td>31 for(const auto &amp; t : tmp</td>
<td>views::join)</td>
</tr>
<tr>
<td>32 process(t);</td>
<td></td>
</tr>
</tbody>
</table>

Revisions
R0: Initial version

R1: Redesign after SG1 review on 2023-06-13:
• Changed design to concurrent object pool.
• The design is no longer limited by what can be expressed with std::atomic.

¹ RISC Software GmbH, Softwarepark 32a, 4232 Hagenberg, Austria, michael.hava@risc-software.at
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 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 based on a concurrently accessible object pool. The proposed design does not require expensive locking for concurrent access, nor does it introduce global memory overhead.

Design Space
`std::object_pool` is a concurrency-safe, dynamically growing object pool. Conceptually it is similar to the following class, though implementations should use more efficient synchronization mechanisms than locking².

```cpp
template<default_initializable T, typename Allocator = allocator<T>>
class object_pool {
    mutable mutex mutex;
    mutable intrusive_list<T, Allocator> storage;
    class handle; // see below
    class snapshot; // see below
public:
    object_pool(Allocator allocator = Allocator{}) noexcept;
    ~object_pool() noexcept;
    [[nodiscard]]
    auto lease() const -> handle;
    [[nodiscard]]
    auto lease_all() const noexcept -> snapshot;
};
```

Given its intended usage as a (low-level) concurrency primitive, `std::object_pool` is neither copy- nor movable (as neither is possible in a lock-free manner) and offers allocator support.

We decided to require pooled objects to be default-constructible (doing value-initialization like containers) as the alternatives would require us to store an initialization function. Said function would have to be either a high-level function like `T()` - requiring `T` to be move-constructible -, or a low-level function like `void(T *)`.

Furthermore, this design sidesteps the unresolved issue of allocator-aware polymorphic function wrappers that lead to the removal of allocator support in `std::function ([P0302R1])`. We expect users to use wrappers like `std::optional` for non-default-constructible types or when a custom initialization logic is needed.

² Our reference implementation employs atomic operations (DWCAS specifically).
A pooled object is at any given point either directly managed by the pool, in which case it is available for future requests, or by a RAII-class granting exclusive access to it. There are two extraction operations:

- lease��得单一对象的所有权。在空池的情况下，将分配新的对象。
- lease_all获得所有当前可用的池对象的所有权。

这两个函数可以安全地并发调用，因此它们被标记为`const`，尽管它们会改变池。

**Handles**

std::object_pool::handle是RAII类，用于管理提取对象的独家访问。其接口如下:

```cpp
template<default_initializable T, typename Allocator>
class object_pool<T, Allocator>::handle {
    object_pool & owner;
    typename decltype(storage)::node_type object;
    public:
        handle() = delete;
        handle(const handle &) = delete;
        auto operator=(const handle &) -> handle & = delete;
        ~handle() noexcept;
        auto operator() const noexcept -> T &;
        auto operator->() const noexcept -> T *;
        auto get() const noexcept -> T *;
};
```

Handles被绑定到构建池，并在销毁时将管理的对象放回池中。访问对象是通过引用来的(*, ->)。

由于 handles可能永远不会过期，我们提出一种设计，使其无法移动，依靠**guaranteed copy-elision**。这种设计消除了需要在moved-from状态（或某种程度上）的可能的悬空。

**Snapshots**

std::object_pool::snapshot遵循std::object_pool::handle，但其管理多个对象，并提供迭代支持 - 允许从并行计算中获取的中间结果进行后处理，其接口相当自述。

```cpp
template<default_initializable T, typename Allocator>
class object_pool<T, Allocator>::snapshot {
    vector<handle> handles;
    public:
        snapshot() = delete;
        snapshot(const snapshot &) = delete;
        auto operator=(const snapshot &) -> snapshot & = delete;
        ~snapshot() noexcept;
        class iterator {
            ... 
            static_assert(forward_iterator<iterator>);
            auto begin() noexcept -> iterator;
            auto end() noexcept -> iterator;
        };
};
```

**Comparison to Established Practice**

在SG1审查R0时，我们被引导到几种靠近这个论文领域或接近的现有实现。我们不提供关于那些的详细技术审查，而是给出高概述的比较。

---

3 The name lease was chosen to indicate that the caller only temporarily gets access to the object.
Two of the designs (Folly, TBB) we analyzed emulate *thread-local* storage bound to a local object. Whilst such a design is easy to reason about, it becomes sub-optimal in environments with high thread counts and little reuse during a parallel operation. R0 of this paper proposed a similar design and switched to the current object pool design after evaluating SG1 feedback.

The other design (BDE) we compared to is also an object pool, albeit of a different design - offering some control on the pool's size and providing a customization point for resetting an object on release. The need to control the count of objects has never come up in our use-cases, but we reckon it would be trivial to add to our design. Customizing resetting behavior is something that although not directly supported by our design, can be replicated by looping over a snapshot - the benefit of this approach is increased flexibility.

**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.

**Proposed Wording**

Wording is relative to [N4958]. Additions are presented like *this*, removals like *this* and drafting notes like *this*.

```
#define __cpp_lib_object_pool YYYYMM //also in <object_pool>
```

[DRAFTING NOTE: Adjust the placeholder value as needed to denote the proposal's date of adoption.]

---

4 e.g. GPUs and dedicated accelerators
[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>[safercl]</td>
<td>Safe reclamation</td>
</tr>
<tr>
<td>[racefree]</td>
<td>Race-free storage</td>
</tr>
</tbody>
</table>

[racefree]

[DRAFTING NOTE: Add a new section in [thread]]

1 ??.?? Race-free storage
2 ??.??.1 General [racefree]
3 ??.??.2 Header <object_pool> synopsis [racefree.objectpool.syn]
4 namespace std {
5     // [racefree.objectpool.class], class template object_pool
6     template<class T, class Allocator = allocator<T>> class object_pool;
7 
8 namespace pmr {
9     template<class T>
10     using object_pool = std::object_pool<T, polymorphic_allocator<T>>;
11 }
12 }
13 ??.??.3 Class template object_pool [racefree.objectpool.class]
14 namespace std {
15     template<default_initializable T, class Allocator = allocator<T>>
16     class object_pool {
17     public:
18         // [racefree.objectpool.handle.class], class object_pool::handle
19         class handle;
20 
21         // [racefree.objectpool.snapshot.class], class object_pool::snapshot
22         class snapshot;
23 
24         // [racefree.objectpool.ctor], constructors, assignment, and destructor
25         object_pool(const Allocator& alloc = Allocator());
26         object_pool(const object_pool&) = delete;
27         object_pool& operator=(const object_pool&) = delete;
28         ~object_pool();
29 
30         // [racefree.objectpool.mod], modifiers
31         [[nodiscard]] handle lease() const;
32         [[nodiscard]] snapshot lease_all() const noexcept;
33     };
34 }
35
1 The object_pool class template is a concurrency-safe, dynamically growing object pool. Growing the pool does not invalidate pointers or references to existing objects.
2 Allocator shall be a cv-unqualified type that meets the C++17Allocator requirements [allocator.requirements.general].
3 Recommended practice: Implementations should avoid high synchronization overhead for concurrent access to storage.

1 ??.??.3.1 Constructors, and destructor [racefree.objectpool.ctor]
2 object_pool(const Allocator& alloc = Allocator());
3 Effects: Initializes the pool with alloc.
4 ~object_pool();
5 Effects: Releases all managed objects.

1 ??.??.3.2 Modifiers [racefree.objectpool.mod]
2 [[nodiscard]] handle lease() const;
3 Effects: If the pool contains no objects, creates a new value-initialized object obj. Otherwise, extracts an object obj from the pool.
4 Synchronization: Synchronizes with other access to the pool.
Returns: A handle whose owner is initialized with \*this and whose object is obj.

Throws: Any exception thrown when growing the pool.

\[
\text{[[nodiscard]] snapshot lease_all() const noexcept;}
\]

Let objs be all objects in the pool.

Postconditions: The pool contains no objects.

Synchronization: Synchronizes with other access to the pool.

Returns: A snapshot whose owner is initialized with \*this and whose objects is objs.

??? Class object_pool::handle [racefree.objectpool.handle.class]

namespace std {
    template<default_initializable T, class Allocator>
    class object_pool<T, Allocator>::handle {
        object_pool & owner;
        // exposition only
        T * object;
        // exposition only
    public:
        // [racefree.objectpool.handle.ctor], constructors, assignment, and destructor
        handle() = delete;
        handle(const handle&) = delete;
        handle& operator=(const handle&) = delete;
        ~handle();
        // [racefree.objectpool.handle.acc], accessors
        T& operator*() const noexcept;
        T* operator->() const noexcept;
        T* get() const noexcept;
    }
}

??? Class object_pool::snapshot [racefree.objectpool.snapshot.class]

namespace std {
    template<default_initializable T, class Allocator>
    class object_pool<T, Allocator>::snapshot {
        object_pool & owner;
        // exposition only
        vector<T *> objects;
        // exposition only
    public:
        using iterator       = implementation-defined;
        using const_iterator = implementation-defined;
        // [racefree.objectpool.snapshot.ctor], constructors, assignment, and destructor
        snapshot() = delete;
        snapshot(const snapshot&) = delete;
        snapshot& operator=(const snapshot&) = delete;
        ~snapshot();
        // [racefree.objectpool.snapshot.iter], iteration
        const_iterator begin() const noexcept;
        iterator begin() noexcept;
        const_iterator cbegin() const noexcept;
        iterator end() noexcept;
        const_iterator cend() const noexcept;
    }
}

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.

??? Class object_pool::snapshot lease_all [racefree.objectpool.snapshot.lease_all.class]

Use: Construction to initialize the snapshot.

1. The object_pool::snapshot class allows exclusive access to an object owned by the creating object_pool.
Acknowledgements
Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading.