Document number: P2774R1
Date: 2023-09-30
Project: Programming Language C++
Audience: SG1
Reply-to: Michael Florian Hava[1] <mfh.cpp@gmail.com>

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

Before

```cpp
span<Triangle> input = …;
double max_area = …;

//split triangle mesh based on max triangle size
mutex m;
deque<vector<Triangle>> tmp;

//process in parallel
for_each(execution::par, input.begin(), input.end(),
  [&](const auto & tria) {
    //extract exclusive object
    auto object{[&] -> vector<Triangle> {
      const lock_guard lock{m};
      if(tmp.empty()) return {}; //need new object
      auto val{move(tmp.front())};
      tmp.pop_front();
      return val;
    }()};

    //generating unbounded output
    for(const auto & t : split(tria, max_area))
      object.emplace_back(t);

    //make object available again
    const lock_guard lock{m};
    partial.emplace_back(std::move(object));
  }
);

//post-process partial results sequentially
for(const auto & t : tmp | views::join)
  process(t);
```

Proposed

```cpp
span<Triangle> input = …;
double max_area = …;

//split triangle mesh based on max triangle size

object_pool<vector<Triangle>> tmp;

//process in parallel
for_each(execution::par, input.begin(), input.end(),
  [&](const auto & tria) {
    //get handle to exclusive object
    auto handle{tmp.lease()};





    auto & object{*handle};

    //generating unbounded output
    for(const auto & t : split(tria, max_area))
      object.emplace_back(t);



    //~handle() makes object available again
  }
);

//post-process partial results sequentially
for(const auto & t : tmp.lease_all() | views::join)
  process(t);
```

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 locking[2].

```
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(const object_pool &) =delete;
  auto operator=(const object_pool &) -> object_pool & =delete;
  ~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`[3] obtains ownership of a single object. In case of an empty pool, a new objects is allocated.
- `lease_all` obtains ownership of all objects currently available in the pool.

Both of these functions can safely be called concurrently, therefore they are marked `const`, even though they mutate the pool.

Handles

`std::object_pool::handle` is a RAII-class that manages exclusive access to an extracted object. Its interface is as follows:

```
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 are tied to the constructing pool and pass the managed object back to it on destruction. Access to the object is granted via the dereference operators (∗, −>) and `get`.

As handles may never outlive the respective pool we propose a design that makes them immovable, relying on *guaranteed copy-elision*. This design removes the need for a *moved-from state* and (somewhat) limits the potential for dangling.

Snapshots

The design of `std::object_pool::snapshot` follows `std::object_pool::handle` but instead manages multiple objects and provides iteration support - enabling post-processing of partial results from parallel computations with any STL-style algorithm. Its interface is rather self-explanatory:

```
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

During the SG1 review of R0 we've been pointed to several existing implementations in/close to the domain of this paper. We don't provide extensive technical reviews of those, but give a high-level comparison to our proposed design:

[3] The name `lease` was chosen to indicate that the caller only temporarily gets access to the object.

	This paper	BDE ObjectPool	Folly ThreadLocal	TBB enumerable_thread_specific
Design	object pool	object pool	thread-local storage emulation	thread-local storage emulation
Copyable	❌	❌	❌	✓
Moveable	❌	❌	✓	✓
Allocator support	✓	✓	❌	✓
Iterator support	✓	❌	✓	✓
Explicit size management	❌	`increaseCapacity` `reserveCapacity`	❌	❌
Object initialization	*value-initialization*	`void(*)(void *, allocator *);`	`T*(*)();`	`T(*)();`
Object recycling	automatic	`releaseObject`	automatic	automatic

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[4]. 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**.

[version.syn]

```
#define __cpp_lib_object_pool YYYYMML //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]

	Subclause	Header
…	…	…
[saferecl]	Safe reclamation	`<rcu>`, `<hazard_pointer>`
[racefree]	Race-free storage	`<object_pool>`

[racefree]

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

??.?? Race-free storage **[racefree]**

??.??.1 General **[racefree.general]**

1 ??.?? describes components that provide race-free storage in multithreaded environments.

??.??. 2 Header <object_pool> synopsis **[objectpool.syn]**

```
namespace std {
  // [racefree.objectpool.class], class template object_pool
  template<class T, class Allocator = allocator<T>> class object_pool;

  namespace pmr {
    template<class T>
      using object_pool = std::object_pool<T, polymorphic_allocator<T>>;
  }
}
```

??.??.3 Class template object_pool **[racefree.objectpool.class]**

```
namespace std {
  template<default_initializable T, class Allocator = allocator<T>>
  class object_pool {
  public:
    // [racefree.objectpool.handle.class], class object_pool::handle
    class handle;

    // [racefree.objectpool.snapshot.class], class object_pool::snapshot
    class snapshot;

    // [racefree.objectpool.ctor], constructors, assignment, and destructor
    object_pool(const Allocator& alloc = Allocator());
    object_pool(const object_pool&) =delete;

    object_pool& operator=(const object_pool&) =delete;

    ~object_pool();

    // [racefree.objectpool.mod], modifiers
    [[nodiscard]] handle lease() const;
    [[nodiscard]] snapshot lease_all() const noexcept;
  };
}
```

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 *Cpp17Allocator* requirements (***[allocator.requirements.general]***).

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

??.??.3.1 Constructors, and destructor **[racefree.objectpool.ctor]**

`object_pool(const Allocator& alloc = Allocator());`

1 *Effects*: Initializes the pool with `alloc`.

`~object_pool();`

2 *Effects*: Releases all managed objects.

??.??.3.2 Modifiers **[racefree.objectpool.mod]**

`[[nodiscard]] handle lease() const;`

1 *Effects*: If the pool contains no objects, creates a new value-initialzed object `obj`. Otherwise, extracts an object `obj` from the pool.

2 *Synchronization*: Synchronizes with other access to the pool.

3 *Returns*: A handle whose *owner* is initialized with `*this` and whose *object* is `obj`.

4 *Throws*: Any exception thrown when growing the pool.

```
[[nodiscard]] snapshot lease_all() const noexcept;
```

5 Let `objs` be all objects in the pool.

6 *Postconditions*: The pool contains no objects.

7 *Synchronization*: Synchronizes with other access to the pool.

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

??.??.4 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;
  };
}
```

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

??.??.4.1 Constructors, and destructor [racefree.objectpool.handle.ctor]

```
~handle();
```

1 *Synchronization*: Synchronizes with other accesses to the pool of *owner*.

2 *Postconditions*: *object* is available in the pool of *owner*.

??.??.4.2 Accessors [racefree.objectpool.handle.acc]

```
T& operator*() const noexcept;
```

1 *Returns*: Equivalent to: `*`*object*.

```
T* operator->() const noexcept;
T* get() const noexcept;
```

2 *Returns*: Equivalent to: *object*.

??.??.5 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;

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

1 The `object_pool::snapshot` class allows exclusive access to multiple objects owned by the creating `object_pool`.

2 `object_pool::snapshot::iterator` meets the forward iterator requirements (***[forward.iterators]***) with value type T.

3 `object_pool::snapshot::const_iterator` meets the requirements of a constant iterator and those of a forward iterator with value type T.

??.??.5.1 Constructors, and destructor [racefree.objectpool.snapshot.ctor]

```
~snapshot();
```

1 *Synchronization*: Synchronizes with other accesses to the pool of *`owner`*.

2 *Postconditions*: *`objects`* are available in the pool of *`owner`*.

??.??.5.2 Iteration [racefree.objectpool.snapshot.iter]

```
const_iterator begin() const noexcept;
iterator begin() noexcept;
const_iterator cbegin() const noexcept;
```

1 *Returns*: An iterator referring to the start of *`objects`*.

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

2 *Returns*: An iterator representing the past-the-end of *`objects`*.

Acknowledgements

Thanks to RISC Software GmbH for supporting this work. Thanks to Peter Kulczycki for proof reading.