The Continuing Future of C++ Concurrency

Anthony Williams

Just Software Solutions Ltd
http://www.justsoftwaresolutions.co.uk

8th June 2016
The Continuing Future of C++ Concurrency

- C++14
- C++17
- Technical Specifications:
  - Concurrency
  - Parallelism
  - Transactional Memory
Concurrency in C++14
Only one new concurrency feature:

- `std::shared_timed_mutex`
- `std::shared_lock<>`
Multiple threads may hold a shared lock

OR

One thread may hold an exclusive lock
std::map<std::string, std::string> table;
std::shared_timed_mutex m;
std::string find_entry(std::string s) {
    std::shared_lock<
        std::shared_timed_mutex> guard(m);
    auto it = table.find(s);
    if (it == table.end())
        throw std::runtime_error("Not found");
    return it->second;
}
std::shared_timed_mutex: exclusive locks

std::map<std::string, std::string> table;
std::shared_timed_mutex m;

void add_entry(
    std::string key, std::string value){
    std::lock_guard<std::shared_timed_mutex> guard(m);
    table.insert(std::make_pair(key, value));
}
The **timed** part of `std::shared_timed_mutex`

```cpp
std::shared_timed_mutex m;
void foo() {
    std::shared_lock<
        std::shared_timed_mutex> sl(m, std::chrono::seconds(1));
    if(!sl.owns_lock())
        return;
    do_foo();
}
```
std::shared_timed_mutex performance

- Not always an optimization: profile, profile, profile
- The `std::shared_timed_mutex` itself is a point of contention
Concurrency in C++17
New in C++17

Two new concurrency features:

- `std::shared_mutex` (non-timed)
- Variadic `std::lock_guard<>`

Plus: the Parallelism TS v1 has been merged, so there are parallel versions of most STL algorithms.
`std::shared_mutex` omits the lock-with-timeout operations from `std::shared_timed_mutex`. It is simpler and faster on some platforms.
Variadic std::lock_guard

In C++11 and C++14, std::lock_guard can only be used with a single mutex.

In C++17, you can use std::lock_guard to lock multiple mutexes in one go, using the same mechanism as std::lock() to avoid deadlock.

std::lock_guard<std::mutex, std::mutex> guard(m1, m2);
Technical Specification for C++
Extensions for Concurrency
Concurrent Futures and Waiters

- Continuations for futures
- Waiting for one or all of a set of futures
- Latches and Barriers
- Atomic Smart Pointers
Concurrency TS v2: Proposals Under Consideration

- Executors and Schedulers
- Distributed Counters
- Concurrent Unordered Containers
- Concurrent Queues
- Safe concurrent stream access
- Resumable functions and coroutines
- Pipelines
Concurreny TS namespace

The concurrency TS provides functions and classes in the `std::experimental` namespace.

In the slides I’ll use `stdexp` instead, as it’s shorter.

namespace stdexp=std::experimental;
A continuation is a new task to run when a future becomes ready

Continuations are added with the new `then` member function

Continuation functions must take a `std::future` as the only parameter

The source future is no longer `valid()`

Only one continuation can be added
Continuations and `stdexp::future`

```cpp
int find_the_answer();
std::string process_result(
    stdexp::future<int>);
auto f = stdexp::async(find_the_answer);
auto f2 = f.then(process_result);
```
int fail(){
    throw std::runtime_error("failed");
}
void next(stdexp::future<int> f){
    f.get();
}
void foo(){
    auto f=stdexp::async(fail).then(next);
    f.get();
}
Using lambdas to wrap plain functions

```cpp
int find_the_answer();
std::string process_result(int);

auto f = stdexp::async(find_the_answer);
auto f2 = f.then([](stdexp::future<int> f) {
    return process_result(f.get());
});
```
Continuations and `stdexp::shared_future`

- Continuations work with `stdexp::shared_future` as well
- The continuation function must take a `stdexp::shared_future`
- The source future remains `valid()`
- Multiple continuations can be added
int find_the_answer();
void next1(stdexp::shared_future<int>);
int next2(stdexp::shared_future<int>);

auto fi=stdexp::async(find_the_answer).share();
auto f2=fi.then(next1);
auto f2=fi.then(next2);
when\_any waits for the first future in the supplied set to be ready. It has two overloads:

```cpp
template<typename ... Futures>
stdexp::future<stdexp::when\_any result<
std::tuple<Futures...>>>
when\_any(Futures... futures);

template<typename Iterator>
stdexp::future<stdexp::when\_any result<
std::vector<
  std::iterator\_traits<Iterator>::value\_type>>>
when\_any(Iterator begin, Iterator end);
```
**when\_any**

**when\_any** is ideal for:

- Waiting for speculative tasks
- Waiting for first results before doing further processing

```cpp
auto f1=stdexp::async(foo);
auto f2=stdexp::async(bar);
auto f3=when\_any(
    std::move(f1),std::move(f2));
f3.then(baz);
```
when_all waits for all futures in the supplied set to be ready. It has two overloads:

```cpp
template<typename ... Futures>
stdexp::future<std::tuple<Futures...>>
when_all(Futures... futures);

template<typename Iterator>
stdexp::future<std::vector<
    std::iterator_traits<Iterator>::value_type>>
when_all(Iterator begin, Iterator end);
```
**when_all**

**when_all** is ideal for waiting for all subtasks before continuing. Better than calling `wait()` on each in turn:

```cpp
auto f1=stdexp::async(subtask1);
auto f2=stdexp::async(subtask2);
auto f3=stdexp::async(subtask3);
auto results=when_all(
    std::move(f1), std::move(f2),
    std::move(f3)).get();
```
Small improvements

The TS also has a couple of small improvements to the `stdexp::future` interface:

- `make_ready_future()` — creates a `stdexp::future` that is *ready*, holding the supplied value
- `make_exceptional_future()` — creates a `stdexp::future` that is *ready*, holding the supplied exception
- `is_ready()` member function — returns whether or not the future is *ready*
A **Latch** is a single-use count-down synchronization mechanism: once **Count** threads have decremented the latch it is permanently signalled.

A **Barrier** is a reusable count-down synchronization mechanism: once **Count** threads have decremented the barrier, it is reset.
std::shared_ptr<T> and std::weak_ptr<T> are not bitwise-copyable, so you can’t have std::atomic<std::shared_ptr<T> > or std::atomic<std::weak_ptr<T> >.

The TS provides stdexp::atomic_shared_ptr<T> and stdexp::atomic_weak_ptr<T> instead.
Concurrency TS: Proposals Under Consideration
Executors and Schedulers

- An executor schedules tasks for execution
- Different executors have different scheduling properties
  e.g. Thread Pools, Serial executor
Distributed counters improve performance by reducing contention on a global counter.

- Counts can be buffered locally to a function or a thread
- Updates of the global count can be via push from each thread or pull from the reader
The current proposal is for a `concurrent_unordered_value_map`.

- No references can be obtained to the stored elements
- Many functions return `optional<mapped_type>`
- As well as simple queries like `find` there are also member functions `reduce` and `for_each"
A concurrent queue is a vital means of inter-thread communication.

- Queues may or may not be lock-free
- May be fixed-size or unlimited
- May be **closed** to prevent additional elements being pushed
- You can obtain a “push handle” or “pop handle” for writing or reading
- Input and output iterators are supported
Safe concurrent stream access

The standard streams provide limited thread safety — output may be interleaved

```cpp
void thread_1()
{
    std::cout<<10<<20<<30;
}
void thread_2()
{
    std::cout<<40<<50<<60;
}

output may be
104050206030
We need a way to group output from several inserts:

```cpp
basic_ostream_buffer<char>
```

```cpp
void thread_1(){
    basic_ostream_buffer<char> buf(std::cout);
    buf<<10<<20<<30;
} // buf destroyed
// contents written to std::cout
```
Coroutines expose a “pull” interface for callback-style implementations.

Resumable functions automatically generate async calls from code that waits on futures.

Both provide alternative ways of structuring code that does asynchronous operations.
The pipeline proposal is a way of wrapping concurrent queues and tasks:

```cpp
queue<InputType> source;
queue<OutputType> sink;
pipeline::from(source) |
    pipeline::parallel(foo, num_threads) |
    bar | baz | sink;
```
Further proposals

There are more proposals not covered here.

Technical Specification for C++ Extensions for Parallelism
Parallelism TS

Parallelism TS v1 (merged to C++17):
- Parallel algorithms
- Mapreduce
- Lightweight Execution Agents
- SIMD and Vector algorithms

Parallelism TS v2:
- Task Blocks
The v1 TS (and thus C++17) provides a new set of overloads of the standard library algorithms with an execution policy parameter:

template<typename ExecutionPolicy, typename Iterator, typename Function>
void for_each(
    ExecutionPolicy&& policy, 
    Iterator begin, Iterator end, 
    Function f);

The **execution policy** may be:

- **std::sequential** — sequential execution on the calling thread
- **std::par** — indeterminately sequenced execution on unspecified threads
- **std::par_vec** — unsequenced execution on unspecified threads
execution_policy objects may be used to pass the desired sequencing as a parameter:

```cpp
void outer(execution_policy policy){
    sort(policy, data.begin(), data.end());
}
void foo(){
    outer(std::par);
}
```
Supported algorithms

The vast majority of the C++ standard algorithms are parallelized, and a few more besides:

adjacent_difference adjacent_find all_of any_of copy copy_if copy_n count count_if equal exclusive_scan fill fill_n find find_end find_first_of find_if find_if_not for_each for_each_n generate generate_n includes inclusive_scan inner_product inplace_merge is_heap is_heap_until is_partitioned is_sorted is_sorted_until lexicographical_compare max_element merge min_element minmax_element mismatch move none_of nth_element partial_sort partial_sort_copy partition partition_copy reduce remove remove_copy remove_copy_if remove_if replace replace_copy replace_copy_if replace_if reverse reverse_copy_copy rotate rotate_copy_copy search search_n set_difference set_intersection set_symmetric_difference set_union sort stable_partition stable_sort swap_ranges transform transform_exclusive_scan transform_inclusive_scan transform_reduce uninitialized_copy uninitialized_copy_n uninitialized_fill uninitialized_fill_n unique unique_copy
Task blocks allow for managing hierarchies of tasks:

- Nested task blocks within an outer task block can run in parallel
- All nested task blocks created within a task region are complete when the region exits
- Task blocks can be nested
Transactional Memory for C++
Two basic types of “transaction” blocks: *synchronized* blocks and *atomic* blocks

- **Synchronized** blocks introduced with the `synchronized` keyword
- **Atomic** blocks introduced with one of `atomic_noexcept`, `atomic_commit` or `atomic_cancel`
**Synchronized** blocks behave as if they lock a global mutex.

```java
int i;
void foo() {
    synchronized {
        ++i;
    }
}
```
Atomic blocks

**Atomic** execute atomically and not concurrently with any synchronized blocks.

```cpp
int i;
void bar(){
    atomic_noexcept {
        ++i;
        ++i;
    }
}
```
Atomic blocks may be concurrent

**Atomic** may execute concurrently if no conflicts

```c
int i, j;
void bar()
{
    atomic_noexcept
    { ++i; }
}
void baz()
{
    atomic_noexcept
    { ++j; }
}
```
The **atomic** blocks differ in their behaviour with exceptions:

- **atomic_noexcept** — escaping exceptions cause undefined behaviour
- **atomic_commit** — escaping exceptions commit the transaction
- **atomic_cancel** — escaping exceptions roll back the transaction, but must be **transaction safe**
Questions?
just::thread provides a complete implementation of the C++14 thread library and the C++ Concurrency TS.

Just::Thread Pro gives you actors, concurrent hash maps, concurrent queues and synchronized values.
C++ Concurrency in Action:
Practical Multithreading

http://stdthread.com/book