The Continuing Future of C++ Concurrency

Anthony Williams

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

12th April 2014

The Continuing Future of C++ Concurrency

- C++14
- Technical Specifications prior to C++17:
  - Concurrency
  - Parallelism
  - Transactional Memory

Concurrency in C++14

New in C++14

Only one new concurrency feature:

- `std::shared_timed_mutex`
- `std::shared_lock<>`

C++14: std::shared_timed_mutex

Multiple threads may hold a shared lock

OR

One thread may hold an exclusive lock

Shared look-up table: reading

```
std::map<std::string,std::string> table;
std::mutex m;

std::string find_entry(std::string s){
  std::lock_guard<std::mutex> guard(m);
  auto it=table.find(s);
  if(it==table.end())
    throw std::runtime_error("Not found");
  return it->second;
}
```

Shared look-up table: updating

```
std::map<std::string,std::string> table;
std::mutex m;

void add_entry(
  std::string key,std::string value){
  std::lock_guard<std::mutex> guard(m);
  table.insert(std::make_pair(key,value));
}
```

std::shared_timed_mutex: shared locks

```
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));
}
```

std::shared_timed_mutex: condition variables

```
std::shared_timed_mutex mut;
std::condition_variable_any cv;

bool ready_to_proceed();

void get_data() {
  std::shared_lock<
    std::shared_timed_mutex> sl(mut);
  cv.wait(sl,ready_to_proceed);
}
```

The **timed** part of `std::shared_timed_mutex`

```
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: more timeouts

```
std::shared_lock<
  std::shared_timed_mutex> sl(
    m,
    std::chrono::steady_clock::now()+
    std::chrono::milliseconds(100));

std::unique_lock<
  std::shared_timed_mutex> ul(
    m,std::chrono::milliseconds(100));
```

std::shared_timed_mutex performance

- Not always an optimization:
  profile, profile, profile
- The `std::shared_timed_mutex` itself is a point of contention

C++14: `std::async` unchanged

Futures returned from `std::async` still block in their destructor if not deferred.

C++14: `std::async` unchanged

This code is still safe:

```
#include <future>
#include <iostream>
void write_message(
  std::string const& message) {
    std::cout<<message;
}
int main() {
  std::string s="hello world\n";
  auto f=std::async([&s]{write_message(s);});
  // oops no wait
}
```

Technical Specification for C++ Extensions for Concurrency

Concurrency TS: Accepted Proposals

Only two accepted proposals:

- Executors and Schedulers
- Continuations for `std::future`

Concurrency TS: Proposals Under Consideration

- Latches and Barriers
- Task groups and regions
- Distributed Counters
- Concurrent Unordered Containers
- Concurrent Queues
- Safe concurrent stream access
- Resumable functions and coroutines
- Pipelines

Executors and Schedulers

- An executor schedules tasks for execution
- `executor` is an abstract base class
- Derived executors have different scheduling properties

Executors

Scheduling a task is done with the `virtual` member function `add`:

```
void add(std::function<void()>)
```

Using executors

```
void schedule_tasks(
  executor& ex){
  ex.add(task1);
  ex.add(task2);
  ex.add([]{ do_something();});
}
```

Supplied executors

The TS includes several executor classes:

- `inline_executor` — `add` runs the task before it returns
- `thread_pool` — runs the tasks on a fixed number of threads
- `serial_executor` — ensures tasks are run in FIFO order on another executor
- `loop_executor` — queues tasks until a “run tasks” function is called manually

loop_executor

loop_executor has three member functions for running tasks:

- try_run_one_closure() — run a task if there is one queued
- run_queued_closures() — run all tasks currently queued
- loop() — run tasks until told to stop

The make_loop_exit() member function interrupts loop() and run_queued_closures() between tasks

loop_executor

```
loop_executor ex;

void thread_1(){
  ex.add(taskA);
  ex.add(taskB);
  ex.add([]{ex.make_loop_exit();});
}
void thread_2(){
  ex.loop();
}
```

The `scheduled_executor` interface

The `scheduled_executor` is derived from `executor`, and adds two new functions to the `executor` interface:

- `add_at(system_time,func)` — schedule the task as soon after `system_time` as possible
- `add_after(delay,func)` — `add_at(system_clock::now()+delay,func)`

The `thread_pool` executor

- The `thread_pool` executor is the only example of a `scheduled_executor` in the TS.
- It provides a fixed-size thread pool.

```
thread_pool ex(
  std::thread::hardware_concurrency());
```

- Dependencies between tasks will potentially deadlock

Executors and `std::async`

There is a new overload of `std::async`:

```
template<class F, class... Args>
future<typename result_of<
    typename decay<F>::type(
        typename decay<Args>::type...)
    >::type>
async(executor &ex,
  F&& f, Args&&... args);
```

Executors and `std::async`

Key differences from normal `std::async`:

- The task is scheduled with `ex.add()` rather than on its own thread
- The resultant future **does not wait in its destructor**

Executors and `std::async`

This code is **NOT** safe:

```
void write_message(
  std::string const& message) {
    std::cout<<message;
}
void foo(executor& ex) {
  std::string s="hello world\n";
  auto f=std::async(ex,[&s]{write_message(s);});
  // oops no wait
}
```

Continuations and `std::future`

- 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 `std::future`

```
int find_the_answer();
std::string process_result(
  std::future<int>);
auto f=std::async(find_the_answer);
auto f2=f.then(process_result);
```

Exceptions and continuations

```
int fail(){
  throw std::runtime_error("failed");
}
void next(std::future<int> f){
  f.get();
}
void foo(){
  auto f=std::async(fail).then(next);
  f.get();
}
```

Using lambdas to wrap plain functions

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

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

Continuations and std::shared_future

- Continuations work with std::shared_future as well
- The continuation function must take a std::shared_future
- The source future remains valid()
- Multiple continuations can be added

std::shared_future continuations

```
int find_the_answer();
void next1(std::shared_future<int>);
unsigned next2(std::shared_future<int>)

auto fi=std::async(find_the_answer).
  share();
fi.then(next1);
fi.then(next2);
```

Scheduling continuations

By default, the continuation inherits the scheduling properties of the parent future:

- `std::promise` or `std::packaged_task` => `std::async(continuation)`
- `std::async(func)` => `std::async(continuation)`
- `std::async(policy,func)` => `std::async(policy,continuation)`
- `std::async(executor,func)` => `std::async(executor,continuation)`

Custom scheduling for continuations

You can specify the scheduling manually:

```
void continuations(
  std::future<int> f,executor& ex){
  auto f2=f.then(
    std::launch::deferred,foo);
  auto f3=f2.then(
    std::launch::async,bar);
  auto f4=f3.then(ex,baz);
  f4.wait();
}
```

Waiting for the first future to be ready

`when_any` waits for the first future in the supplied set to be ready. It has two overloads:

```
template<typename ... Futures>
std::future<std::tuple<Futures...> >
when_any(Futures... futures);
template<typename Iterator>
std::future<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

```
auto f1=std::async(foo);
auto f2=std::async(bar);
auto f3=when_any(
  std::move(f1),std::move(f2));
f3.then(baz);
```

Waiting for all futures to be ready

`when_all` waits for all futures in the supplied set to be ready. It has two overloads:

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

```
auto f1=std::async(subtask1);
auto f2=std::async(subtask2);
auto f3=std::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 `std::future` interface:

- `make_ready_future()` — creates a `std::future` that is **ready**, holding the supplied value
- `ready()` member function — returns whether or not the future is **ready**
- `unwrap()` member function — converts a `std::future<std::future<T> >` into a `std::future<T>`

Concurrency TS:
Proposals Under Consideration

Latches and Barriers

- 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.

Task groups and regions

Task groups or regions allow for managing hierarchies of tasks:

- Tasks within a task region can run in parallel
- All tasks created within a task region are complete when the region exits
- Task regions can be nested

Distributed Counters

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

Concurrent Unordered Containers

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`

Concurrent Queues

A concurrent queue is a vital means of inter-thread communication.

- Queues may or may not be lock-free
- May be fixed-size of 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

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

output may be

```
104050206030
```

Safe concurrent stream access

We need a way to group output from several inserts: `basic_ostream_buffer<char>`

```
void thread_1(){
  basic_ostream_buffer<char> buf(
    std::cout);
  buf<<10<<20<<30;
} // buf destroyed
  // contents written to std::cout
```

Resumable functions and coroutines

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.

Pipelines

The pipeline proposal is a way of wrapping concurrent queues and tasks:

```
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

Already accepted:

- Parallel algorithms

Still under discussion:

- Mapreduce
- Lightweight Execution Agents
- SIMD and Vector algorithms

Parallel Algorithms

This TS 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);
```

Execution Policies

The **execution policy** may be:

- **parallel::seq** — sequential execution on the calling thread
- **parallel::par** — indeterminately sequenced execution on unspecified threads
- **parallel::vec** — unsequenced execution on unspecified threads

execution_policy objects

execution_policy objects may be used to pass the desired sequencing as a parameter:

```
void outer(execution_policy policy){
  sort(policy,data.begin(),data.end());
}
void foo(){
  outer(parallel::par);
}
```

Supported algorithms

The vast majority of the C++ standard algorithms are parallelized:

`adjacent_find` `all_of` `any_of` `copy_if` `copy_n` **`copy`** `count_if` **`count`** `equal` `exclusive_scan` `fill_n` `fill` `find_end` `find_first_of` `find_if_not` `find_if` **`find`** `for_each_n` **`for_each`** `generate_n` `generate` `includes` `inclusive_scan` `inplace_merge` `is_heap` `is_partitioned` `is_sorted_until` `is_sorted` `lexicographical_compare` `max_element` **`merge`** `min_element` `minmax_element` `mismatch` `move` `none_of` `nth_element` `partial_sort_copy` `partial_sort` `partition_copy` `partition` **`reduce`** `remove_copy_if` `remove_copy` `remove_if` `remove` `replace_copy_if` `replace_copy` `replace` `reverse_copy` `reverse` `rotate_copy` `rotate` `search_n` `search` `set_difference` `set_intersection` `set_symmetric_difference` `set_union` **`sort`** `stable_partition` `stable_sort` `swap_ranges` **`transform`** `uninitialized_copy_n` `uninitialized_copy` `uninitialized_fill_n` `uninitialized_fill` `unique_copy` `unique`

Parallelism TS:
Proposals Under Consideration

Parallelism TS: Proposals Under Consideration

- Map-reduce:
  A policy-based framework from transforming a set of input values and combining the results
- Vector and SIMD computation:
  Better support for vector computations than `parallel::vec`
- Lightweight Execution Agents:
  How do SIMD and GPGPU tasks map to thread-local storage and thread IDs?

Transactional Memory for C++

Transactional Memory

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

Synchronized blocks behave as if they lock a global mutex.

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

Atomic blocks

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

```
int i;
void bar(){
  atomic_noexcept {
    ++i;
  }
}
```

Atomic blocks may be concurrent

Atomic may execute concurrently if no conflicts

```
int i,j;
void bar(){
  atomic_noexcept { ++i; }
}
void baz(){
  atomic_noexcept { ++j; }
}
```

Atomic blocks and exceptions

The **atomic** blocks differ in their behaviour with exceptions:

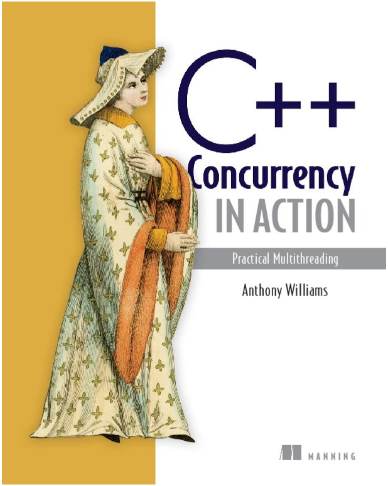
- `atomic_noexcept` — escaping exceptions cause undefined behaviour
- `atomic_commit` — escaping exceptions commit the transaction
- `atomic_cancel` — escaping exceptions roll back the transaction, but must be **transaction safe**

Questions?

Just::Thread

`just::thread` provides a complete implementation of the C++11 thread library for MSVC and g++ on Windows, and g++ for Linux and MacOSX. C++14 support currently in testing.

My Book



C++ Concurrency in Action:
Practical Multithreading

http://stdthread.com/book