Using tasks to simplify concurrency in modern C++

Christian Blume, Serato

October 26, 2017 at Pacific++

What is a task?

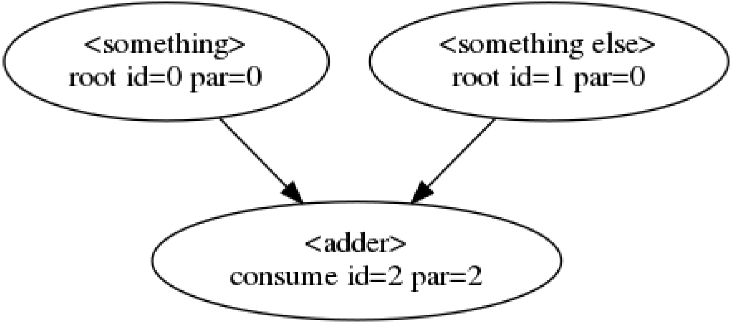
A unit of work that *may* be dependent on other tasks.

Examples

- Adding two numbers
- Computing a Fourier transformation
- Making a web request
- Scheduling a GPU job
- ...

In this talk, all tasks are organized in a directed acyclic graph.

A simple graph of tasks



- parents are at the top (<something>, <something else>)
- parents are consumed or waited for by child (<adder>)
- parents are independent with regard to child and can run in parallel

Task-based concurrency

- is concerned with units of work only, not threads
- is orthogonal to threads, i.e. independent of particular threads
- allows us to focus on getting stuff done (no thread handling)
- lets us organize our work into logical, possibly dependent tasks (parallel by design)

Disclaimer: Task-based concurrency does NOT help with data races.

Two concurrency problems

```
int x = 0;

void add_one() {
    x += 1;
}

void mult_two() {
    x *= 2;
}
```

We want to achieve two goals:

1. Repeatedly run `add_one` on a different thread
2. Repeatedly run `add_one` and then `mult_two`, both on separate threads

Three libraries are compared: standard lib, boost, transwarp

Problem 1: Run add_one only (*classic* approach)

All code that follows is in a single translation unit.
We're ignoring exception handling.

```
bool quit = false; // do we want to quit processing?
atomic_bool done{false}; // has the result been computed?
bool start = false; // can we start processing?
condition_variable cv; // to notify the worker thread
mutex m; // need this for cv and start
```

Please do NOT write code like this!

Problem 1: Run add_one only (*classic* approach)

```
void worker() { // run on a separate thread
    for (;;) {
        {
            unique_lock<mutex> lock(m);
            cv.wait(lock, [&]{ return start || quit; });
            if (quit) return; // the app wants to terminate
            start = false;
        }
        add_one();
        done = true; // signal that result is available
    }
}
```

Problem 1: Run add_one only (*classic* approach)

```
int main() {
    thread t{worker}; // launch the thread
    int count = 0;
    while (count++ < 3) {
        {
            lock_guard<mutex> lock(m);
            start = true; // we want to start calculating
        }
        cv.notify_one(); // notify to start calculating
        while (!done); // block until result is available (cv?)
        done = false;
        cout << "x = " << x << endl;
    } // end of while
    ...
```

Problem 1: Run add_one only (*classic* approach)

```
    ...
    {
        lock_guard<mutex> lock(m);
        quit = true; // the app is terminated
    }
    cv.notify_one();
    t.join();
}
```

Output:

```
x = 1
x = 2
x = 3
```

Problem 1: Run add_one only (std::async approach)

```
int main() {
    int count = 0;
    while (count++ < 3) {
        // first param is launch policy
        auto future = async(launch::async, add_one);
        future.wait(); // wait until result becomes available
        cout << "x = " << x << endl;
    }
}
```

Problem: No control over the thread that is used to run the operation.
Worst case: Every iteration launches a new thread :(

Problem 1: Run add_one only (*boost* approach)

```
int main() {
    boost::basic_thread_pool executor{4}; // pool with 4 threads
    int count = 0;
    while (count++ < 3) {
        auto future = boost::async(executor, add_one);
        future.wait();
        cout << "x = " << x << endl;
    }
}
```

The custom executor gives us control over where our tasks are executed.
A boost executor can be any class that implements:

```
void submit(boost::function<void()> functor);
```

Problem 1: Run add_one only (*transwarp* approach)

```
int main() {
    tw::parallel executor{4}; // thread pool with 4 threads
    // first param is the task type (root, consume, wait)
    auto task = tw::make_task(tw::root, add_one);
    int count = 0;
    while (count++ < 3) {
        task->schedule(executor);
        task->get_future().wait();
        cout << "x = " << x << endl;
    }
}
```

A transwarp executor must implement this interface:

```
string get_name() const;
void execute(const function<void()>& functor,
             const shared_ptr<tw::node>& node);
```

Problem 2: Run add_one and mult_two

Given these functions:

```
int x = 0;

void add_one() {
    x += 1;
}

void mult_two() {
    x *= 2;
}
```

We want to repeatedly run `add_one` and then `mult_two`, both on separate threads

Problem 2: Run `add_one` and `mult_two` (*classic* approach)

Not shown to prevent brain injuries.

Essentially:

- Another set of locks, mutexes, and condition variables
- Another worker function to wait for `add_one` and then run `mult_two`
- Another thread that runs the second worker function

Output:

```
x = 2
x = 6
x = 14
```

Problem 2: Run add_one and mult_two (*boost* approach)

```
int main() {
    boost::basic_thread_pool executor{4};
    int count = 0;
    while (count++ < 3) {
        auto future1 = boost::async(executor, add_one);
        // make use of a continuation
        auto future2 = future1.then(executor,
                              [](boost::future<void>) { mult_two(); });
        future2.wait();
        cout << "x = " << x << endl;
    }
}
```

This cannot be done with current standard C++ but *may* be coming with C++20 (concurrency ts).

Problem 2: Run add_one and mult_two (*transwarp* approach)

```
int main() {
    tw::parallel executor{4};
    // build the task graph upfront
    auto task1 = tw::make_task(tw::root, add_one);
    auto task2 = tw::make_task(tw::wait, mult_two, task1);
    int count = 0;
    while (count++ < 3) {
        // schedule all tasks in the right order
        task2->schedule_all(executor);
        task2->get_future().wait();
        cout << "x = " << x << endl;
    }
}
```

What we've seen so far

- Solved two very simple concurrency problems
- Four approaches: classic, std::async, boost, transwarp

classic approach:

- Involves locks, mutexes, and condition variables
- Complicated and error-prone, explicit thread-handling
- Threads are bound to functions we want to calculate
- Hard to extend when dependencies change

Task-based approaches

std::async approach:

- Already makes code much simpler
- Gives up control of which thread is used to run the operation
- Cannot be used to model dependencies (Problem 2 cannot be tackled)

boost approach:

- Solves the shortcomings of the current C++ standard
- Has support for executors to control where operations are run
- Allows to model dependencies via future continuations

transwarp approach:

- Similar to boost except that tasks can be scheduled multiple times
- The task graph is built upfront as a model of the operations

Other libraries for task-parallelism

HPX (High Performance ParalleX)

- Very similar interface to Boost with regard to futures
- Neither HPX nor Boost provide task graphs for multiple invocations

TBB (Threading Building Blocks)

- tasks can be chained similar to continuations, also for multiple invocations
- API seems somewhat harder to use than HPX or Boost

Stlab

- Provides Boost-like future support for one-shot graphs
- Introduces the concept of *channels* for multiple invocations

A more evolved problem

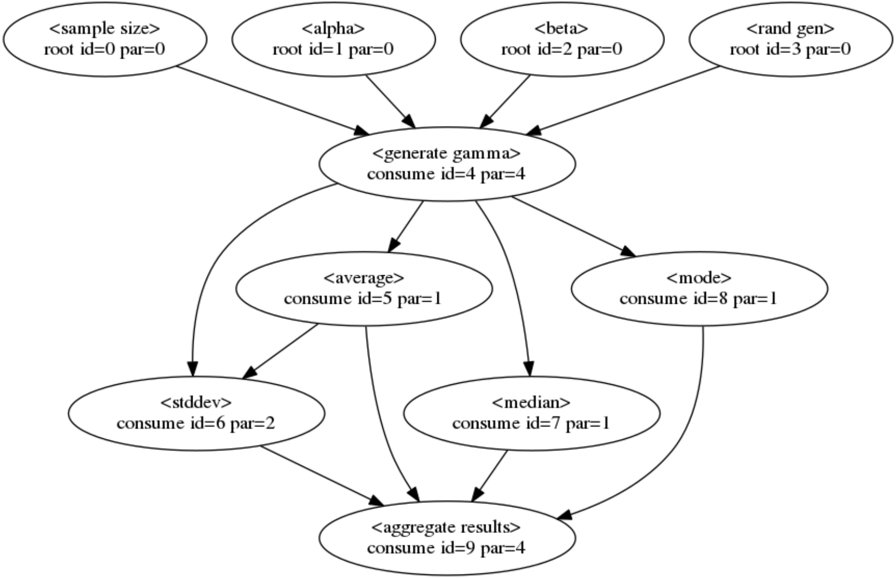
- Draw random values from a gamma distribution
- Compute average, standard deviation, mode, and median
- Goal: Compute stuff in parallel in a task-based fashion

The standard deviation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

depends on the average $\overline{x}$ itself!

Task graph (courtesy of transwarp)



Prerequisites

```
using data_t = shared_ptr<vector<double>>;

data_t generate_gamma(size_t sample_size, double alpha,
                      double beta, shared_ptr<mt19937> gen);

double average(data_t data);

double stddev(data_t data, double average);

double median(data_t data);

int mode(data_t data);

// result is a simple struct holding the results
result aggregate(double avg, double std, double med, int mode);
```

Solving it with Boost

```
double alpha = 1;
double beta = 1;
boost::basic_thread_pool exec{4};

double count = 1;
while (count < 4) {
    // compute_graph is where the magic happens
    auto future = compute_graph(exec, sample_size, alpha, beta);
    const result res = future.get();
    cout << res << endl;
    // Changing input
    alpha += count;
    beta += count;
    ++count;
}
```

Solving it with Boost (cont.)

```
template<typename Executor>
boost::future<result> compute_graph(Executor& exec,
          size_t sample_size, double alpha, double beta) {
// wrapping parameters into ready futures
auto gen_fut =
   boost::make_ready_future(make_shared<mt19937>(1));
auto size_fut = boost::make_ready_future(sample_size);
auto alpha_fut = boost::make_ready_future(alpha);
auto beta_fut = boost::make_ready_future(beta);
// creating a future of futures
auto para_fut = boost::when_all(move(gen_fut), move(size_fut),
   move(alpha_fut), move(beta_fut));
...
```

Solving it with Boost (cont.)

```
// Generate the data from a gamma distribution
auto data_fut = para_fut.then(exec, [](decltype(para_fut) f) {
    auto p = f.get(); // p is a tuple of futures
    shared_ptr<mt19937> gen = get<0>(p).get();
    size_t sample_size = get<1>(p).get();
    double alpha = get<2>(p).get();
    double beta = get<3>(p).get();
    return generate_gamma(sample_size, alpha, beta, gen);
}).share();
// Compute the average
auto avg_fut = data_fut.then(exec,
   [](boost::shared_future<data_t> f) {
    return average(f.get());
}).share();
...
```

Solving it with Boost (cont.)

```
// future of futures that waits for completion of parents
auto d_a_fut = boost::when_all(data_fut, avg_fut);
// Compute the stddev using data and average
auto stddev_fut = d_a_fut.then(exec, [](decltype(d_a_fut) f) {
    auto data_avg = f.get(); // data_avg is a tuple of futures
    data_t data = get<0>(data_avg).get();
    double avg = get<1>(data_avg).get();
    return stddev(data, avg);
});
// Compute the median
auto median_fut = data_fut.then(exec,
   [](boost::shared_future<data_t> f) {
    return median(f.get());
});
...
```

Solving it with Boost (cont.)

```
// Compute the mode
auto mode_fut = data_fut.then(exec,
    [](boost::shared_future<data_t> f) {
     return mode(f.get());
});
// Wait for all to finish
auto agg_wait_fut = boost::when_all(avg_fut, move(stddev_fut),
    move(median_fut), move(mode_fut));
// Create aggregated result
return agg_wait_fut.then(exec, [](decltype(agg_wait_fut) f){
    auto r = f.get(); // r is a tuple of futures
    return aggregate(get<0>(r).get(), get<1>(r).get(),
                     get<2>(r).get(), get<3>(r).get());
});
} // compute_graph
```

Solving it with transwarp

```
double alpha = 1;
double beta = 1;
// Creating the task graph upfront
auto task = build_graph(sample_size, alpha, beta);
tw::parallel exec{4};
double count = 1;
while (count < 4) {
    task->schedule_all(exec); // schedule all in the right order
    const result res = task->get_future().get();
    cout << res << endl;
    // Changing input
    alpha += count;
    beta += count;
    ++count;
}
```

Solving it with transwarp (cont.)

```
shared_ptr<tw::task<result>> build_graph(size_t sample_size,
                                  double& alpha, double& beta) {
auto gen = make_shared<mt19937>(1);
// Creating parameter tasks at the top of the graph
auto gen_task = tw::make_task(tw::root, [gen] { return gen; });
auto size_task = tw::make_task(tw::root,
                        [sample_size] { return sample_size; });
auto alpha_task = tw::make_task(tw::root,
                               [&alpha] { return alpha; });
auto beta_task = tw::make_task(tw::root,
                               [&beta] { return beta; });
// The data task consumes the parameter tasks
auto data_task = tw::make_task(tw::consume, generate_gamma,
                 size_task, alpha_task, beta_task, gen_task);
...
```

Solving it with transwarp (cont.)

```
// Create tasks for the different measures
auto avg_task = tw::make_task(tw::consume, average, data_task);
auto stddev_task = tw::make_task(tw::consume, stddev,
                                 data_task, avg_task);
auto median_task = tw::make_task(tw::consume, median,
                                 data_task);
auto mode_task = tw::make_task(tw::consume, mode, data_task);
// Create the final aggregate task
shared_ptr<tw::task<result>> t = tw::make_task(tw::consume,
   aggregate, avg_task, stddev_task, median_task, mode_task);
return t;
} // build_graph
```

transwarp executor interface

```
class executor {
public:

virtual ~executor() = default;

virtual std::string get_name() const = 0;

virtual void execute(const std::function<void()>& functor,
                     const std::shared_ptr<tw::node>& node) = 0;
};
```

- functor is the function to be run
- node is a container carrying meta-data of the current task

transwarp sequential executor

```cpp
class sequential : public tw::executor {
public:

std::string get_name() const override {
    return "transwarp::sequential";
}

void execute(const std::function<void()>& functor,
             const std::shared_ptr<tw::node>&) override {
    functor(); // run on same thread as the call to schedule()
}

};
```

transwarp parallel executor

```
class parallel : public tw::executor {
public:

explicit parallel(std::size_t n_threads)
: pool_(n_threads)
{}

void execute(const std::function<void()>& functor,
             const std::shared_ptr<tw::node>&) override {
    pool_.push(functor); // push into the thread pool
}

private:
    tw::detail::thread_pool pool_; // locks to push/pop functors
};
```

A lock-free single-thread executor

```
void execute(/* ... */) override {
    q_.push(functor); // push into lock-free queue
}
void worker() { // is run on the thread
    for (;;) {
        std::function<void()> functor;
        bool success = false;
        while (!success && !done_) success = q_.pop(functor);
        if (!success && done_) break;
        functor(); // run the functor
    }
}
std::atomic_bool done_(false);
boost::lockfree::spsc_queue<std::function<void()>> q_(1000);
std::thread thread_;
```

Conclusions

We've seen

- two very simple concurrency problems
- a real-world case study from statistics
- how to define custom executors

Take away

- task-based approaches greatly simplify concurrent code
- task-based concurrency is orthogonal to threads
- futures in current standard are pretty weak (hopefully better in C++20)
- Boost futures allow for continuations with custom executors
- transwarp provides a task-graph for multiple invocations
- executors can be a powerful tool to schedule tasks

Thank You

transwarp - future work

- Add new methods to `task`: `wait()`, `get()`
- Create function to create a *ready* task with given value
- Create custom `packaged_task` for better scheduling performance
- Improve error messages, in particular when parent tasks don't match the child task
- ...

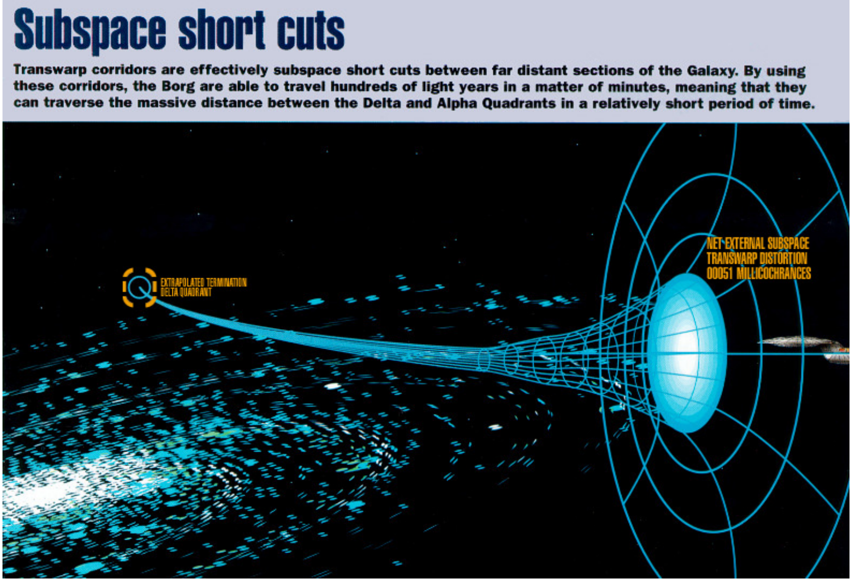
I am accepting pull requests!

`https://github.com/bloomen/transwarp`

schedule_impl(tw::executor* executor)

```
weak_ptr<task_impl> self = this->shared_from_this();
auto futures = tw::detail::get_futures(parents_);
auto pack_task =
    make_shared<packaged_task<result_type()>>(bind(
    &task_impl::evaluate, node_->get_id(), move(self),
    move(futures)));
future_ = pack_task->get_future();
if (executor_) {
     executor_->execute([pack_task] { (*pack_task)(); }, node_);
} else if (executor) {
     executor->execute([pack_task] { (*pack_task)(); }, node_);
} else {
     (*pack_task)();
}
```

transwarp



Subspace short cuts

Transwarp corridors are effectively subspace short cuts between far distant sections of the Galaxy. By using these corridors, the Borg are able to travel hundreds of light years in a matter of minutes, meaning that they can traverse the massive distance between the Delta and Alpha Quadrants in a relatively short period of time.