Boost Asynchronous

Christophe Henry

Boost Asynchronous

Christophe Henry

Table of Contents

List of Tables

Introduction

Note: Asynchronous is not part of the Boost library. It is planed to be offered for Review at the beginning of 2016.

Asynchronous is first of all an architecture tool. It allows organizing a complete application into Thread Worlds, each world having the possibility to use the same or different threadpools for long-lasting tasks. The library provides an implementation of the Active Object pattern, extended to allow many Active Objects to live in the same World. It provides several Threadpools and many parallel algorithms making use of it. And most important of all, it allows simple, blocking-free asynchronous programming based on thread-safe callbacks.

This is particularly relevant for Designers who often have headaches bringing the notion of threads into class diagrams. These usually do not mix well. Asynchronous solves this problems: it allows representing a Thread World as a Component or Package, objects of this Component or Package living into the corresponding thread.

Why do we care? Herb Sutter wrote in an article [http://www.gotw.ca/publications/concurrency-ddj.htm] "The Free Lunch Is Over", meaning that developpers will be forced to learn to develop multi-threaded applications. The reason is that we now get our extra power in the form of more cores. The problem is: multithreading is hard! It's full of ugly beasts waiting hidden for our mistakes: races, deadlocks, crashes, all kinds of subtle timing-dependent bugs. Worse yet, these bugs are hard to find because they are never reproducible when we are looking for them, which leaves us with backtrace analysis, and this is when we are lucky enough to have a backtrace in the first place.

This is not even the only danger. CPUs are a magnitude faster than memory, I/O operations, network communications, which all stall our programms and degrade our performance, which means long sessions with coverage or analysis tools.

Trying to solve these problems with tools of the past (mutexes, programmer-managed threads) is a dead-end. It's just too hard. This is where Boost Asynchronous is helping. Let us forget what mutexes, atomics and races are!

There are existing solutions for asynchronous or parallel programming. To name a few:

- std/boost::async.

- Intel TBB.

- N3428.

TBB is a wonderful parallel library. But it's not asynchronous as one needs to wait for the end of a parallel call.

std::async will return a future. But what to do with it? Wait for it? This would be synchronous. Collect them and then wait for all? This would also be synchronous. Collect them, do something else, then check whether they are ready? This would be wasted opportunity for more calculations.

To solve these problems, NB3428 is an attempt at continuations. Let's have a quick look at code using futures and .then (taken from N3428):

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f) {return f.get().to_string();}); /
f2.get(); // just a "small get" at the end?
```

Saying that there is only a "small get" at the end is, for an application with real-time constraints, equivalent to saying at a lockfree conference something like "what is all the fuss about? Can't we just add a small lock at the end?". Just try it...

Worse yet, it clutters the code, makes it hard to debug and understand. The author, also being the author of Boost Meta State Machine sees no way to use this paradigm with state machines.

Asynchronous supports this programming model too, though it is advised to use it only for simple programs, quick prototyping unit tests, or as a step to the more powerful tools offered by the library. std::async can be replaced by boost::asynchronous::post_future:

```
auto pool = boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::multiqueue_threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<>>>(8); // create a
std::future<int> fu = boost::asynchronous::post_future(pool,
    []()
    {
        return 123;
    });
f1.get();
```

Instead of an ugly future.then, Asynchronous supports continuations as coded into the task itself. We will see later how to do it. For the moment, here is a quick example. Let's say we want to modify a vector in parallel, then reduce it, also in parallel, without having to write synchronization points:

```
std::future<int> fu = boost::asynchronous::post_future(pool, // pool as before
    [this]()
    {
        return boost::asynchronous::parallel_reduce(                          // reduce
            boost::asynchronous::parallel_for(std::move(this->m_data), // our da
                                    [](int& i)
                                    {
                                        i += 2;                              // trans
                                    }, 1024),                                // cutof
            [](int const& a, int const& b)                                   // reduce
            {
                return a + b;
            }, 1024);                                                        // reduce
    });
int res = fu.get();
```

But this is just the beginning. It is not really asynchronous. More important, Boost Asynchronous is a library which can play a great role in making a thread-correct architecture. To achieve this, it offers tools for asynchronous designs: ActiveObject, safe callbacks, threadpools, servants, proxies, queues, algorithms, etc.

Consider the following example showing us why we need an architecture tool:

```
struct Bad : public boost::signals::trackable
{
    int foo();
};
boost::shared_ptr<Bad> b;
future<int> f = async([b](){return b->foo()});
```

Now we have the ugly problem of not knowing in which thread Bad will be destroyed. And as it's pretty hard to have a thread-safe destructor, we find ourselves with a race condition in it.

Asynchronous programming has the advantage of allowing to design of code, which is nonblocking and single-threaded while still utilizing parallel hardware at full capacity. And all this while forgetting what a mutex is.

This brings us to a central point of Asynchronous: if we build a system with strict real-time constraints, there is no such thing as a small blocking get(). We need to be able to react to any event in the system in a timely manner. And we can't afford to have lots of functions potentially waiting too long everywhere in our code. Therefore, .then() is only good for an application of a few hundreds of lines. What about

using a timed_wait instead? Nope. This just limits the amount of time we waste waiting. Either we wait too long before handling an error or result, or we wait not enough and we poll. In any case, while waiting, our thread cannot react to other events and wastes time.

An image being more worth than thousand words, the following story will explain in a few minutes what Asynchronous is about. Consider some fast-food restaurant:
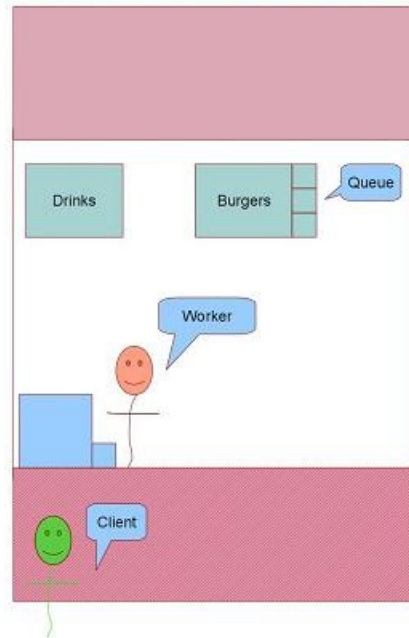
This restaurant has a single employee, Worker, who delivers burgers through a burger queue and drinks. A Customer comes. Then another, who waits until the first customer is served.
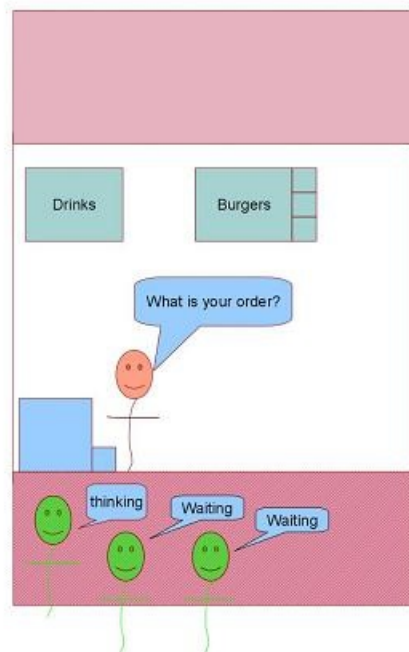
To keep customers happy by reducing waiting time, the restaurant owner hires a second employee:
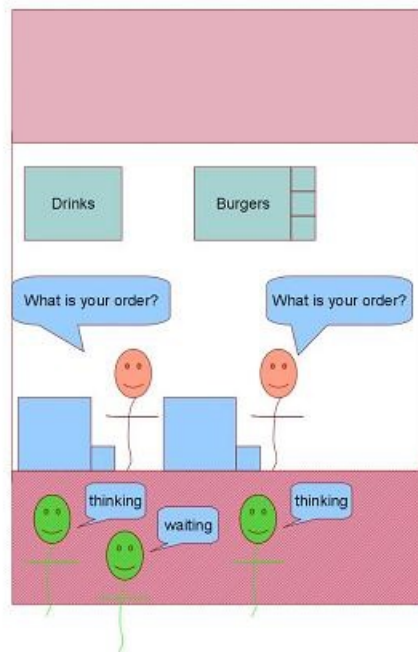
Unfortunately, this brings chaos in the restaurant. Sometimes, employes fight to get a burger to their own customer first:
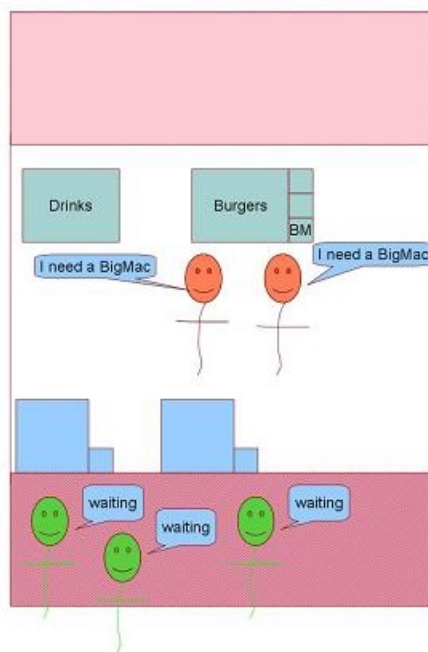
And sometimes, they stay in each other's way:
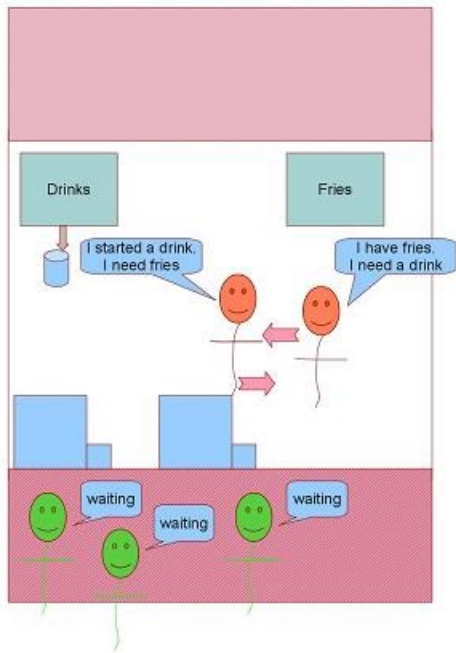
This clearly is a not an optimal solution. Not only the additional employee brings additional costs, but both employees now spend much more time waiting. It also is not a scalable solution if even more customers want to eat because it's lunch-time right now. Even worse, as they fight for resources and stay in each other's way, the restaurant now serves people less fast than before. Customers flee and the restaurant gets bankrupt. A sad story, isn't it? To avoid this, the owner decides to go asynchronous. He keeps a single worker, who runs in zero time from cash desk to cash desk:
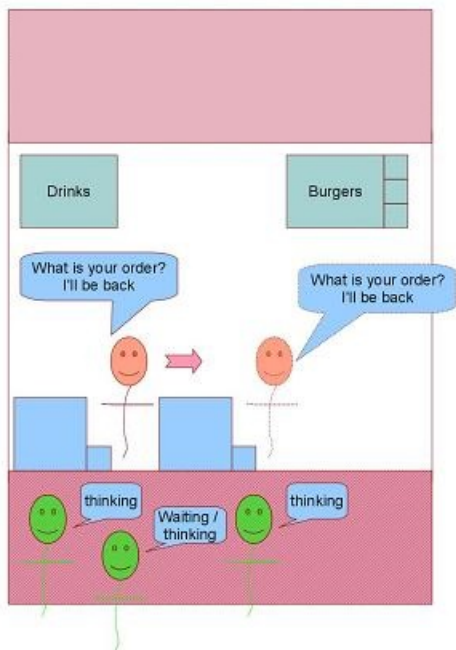
The worker never waits because it would increase customer's waiting time. Instead, he runs from cash desks to the burger queue, beverage machine using a self-made strategy:

- ask what the customer wants and keep an up-to-date information of the customer's state.

- if we have another customer at a desk, ask what he wants. For both customers, remember the state of the order (waiting for customer choice, getting food, getting drink, delivering, getting payment, etc.)

- as soon as some new state is detected (customer choice, burger in the queue, drink ready), handle it.

- priorities are defined: start the longest-lasting tasks first, serve angry-looking customers first, etc.

The following diagram shows us the busy and really really fast worker in action:
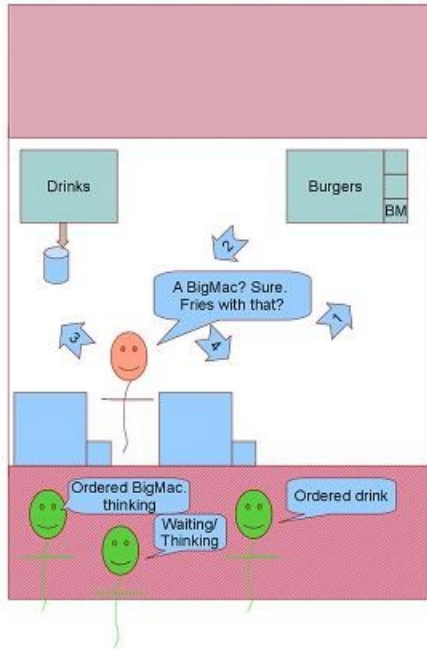
Of course the owner needs a worker who runs fast, and has a pretty good memory so he can remember what customers are waiting for.

This is what Asynchronous is for. A worker (thread) runs as long as there are waiting customers, following a precisely defined algorithm, and lots of state machines to manage the asynchronous behaviour. In case of customers, we could have a state machine: Waiting -> PickingMenu -> WaitingForFood -> Paying.

We also need some queues (Burger queue, Beverage glass positioning) and some Asynchronous Operation Processor (for example a threadpool made of workers in the kitchen), event of different types (Drinks delivery). Maybe we also want some work stealing (someone in the kitchen serving drinks as he has no more burger to prepare. He will be slower than the machine, but still bring some time gain).

To make this work, the worker must not block, never, ever. And whatever he's doing has to be as fast as possible, otherwise the whole process stalls.

Part I. Concepts

Table of Contents

Chapter 1. Related designs: std::async, Active Object, Proactor

std::async

What is wrong with it

The following code is a classical use of std::async as it can be found in articles, books, etc.

```
std::future<int> f = std::async([](){return 42;}); // executes asynchronously
int res = f.get(); // wait for result, block until ready
```

It looks simple, easy to use, and everybody can get it. The problem is, well, that it's not really asynchronous. True, our lambda will execute in another thread. Actually, it's not even guaranteed either. But then, what do we do with our future? Do we poll it? Or call get() as in the example? But then we will block, right? And if we block, are we still asynchronous? If we block, we cannot react to any event happening in our system any more, we are unresponsive for a while (are we back to the old times of freezing programs, the old time before threads?). We also probably miss some opportunities to fully use our hardware as we could be doing something more useful at the same time, as in our fast-food example. And diagnostics are looking bad too as we are blocked and cannot deliver any. What is left to us is polling. And if we get more and more futures, do we carry a bag of them with us at any time and check them from time to time? Do we need some functions to, at a given point, wait for all futures or any of them to be ready?

Wait, yes they exist, `wait_for_all` and `wait_for_any`...

And what about this example from an online documentation?

```
{
    std::async(std::launch::async, []{ f(); });
    std::async(std::launch::async, []{ g(); });
}
```

Every std::async returns you a future, a particularly mean one which blocks upon destruction. This means that the second line will not execute until f() completes. Now this is not only not asynchronous, it's also much slower than calling sequentially f and g while doing the same.

No, really, this does not look good. Do we have alternatives?

N3558 / N3650

Of course it did not go unnoticed that std::async has some limitations. And so do we see some tries to save it instead of giving it up. Usually, it goes around the lines of blocking, but later.

```
future<int> f1 = async([]() { return 123; });
future<string> f2 = f1.then([](future<int> f)
{
  return f.get().to_string(); // here .get() won't block
});
// and here?
string s= f2.get();
```

The idea is to make std::async more asynchronous (this already just sounds bad) by adding something (.then) to be called when the asynchronous action finishes. It still does not fly:

• at some point, we will have to block, thus ending our asynchronous behavior

- This works only for very small programs. Do we imagine a 500k lines program built that way?

And what about the suggestion of adding new keywords, async and await, as in N3650? Nope. First because, as await suggests, someone will need, at some point, to block waiting. Second because as we have no future, we also lose our polling option.

Active Object
Design
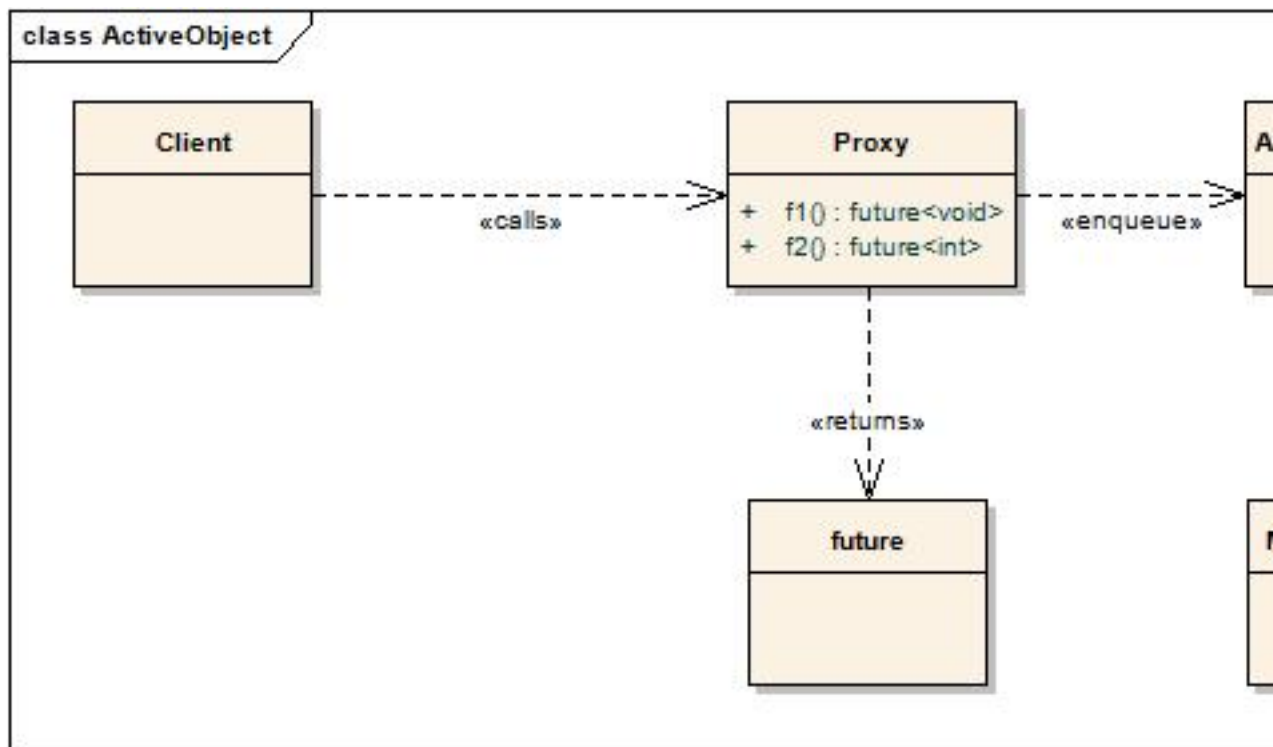
This simplified diagram shows a possible design variation of an Active Object pattern.

A thread-unsafe Servant is hidden behind a Proxy, which offers the same members as the Servant itself. This Proxy is called by clients and delivers a future object, which will, at some later point, contain the result of the corresponding member called on the servant. The Proxy packs a MethodRequest corresponding to a Servant call into the ActivationQueue. The Scheduler waits permanently for MethodRequests in the queue, dequeues them, and executes them. As only one scheduler waits for requests, it serializes access to the Servant, thus providing thread-safety.

However, this pattern presents some liabilities:

- Performance overhead: depending on the system, data moving and context switching can be a performance drain.

- Memory overhead: for every Servant, a thread has to be created, consuming resources.

- Usage: getting a future gets us back to the non-asynchronous behaviour we would like to avoid.

Proactor
Design
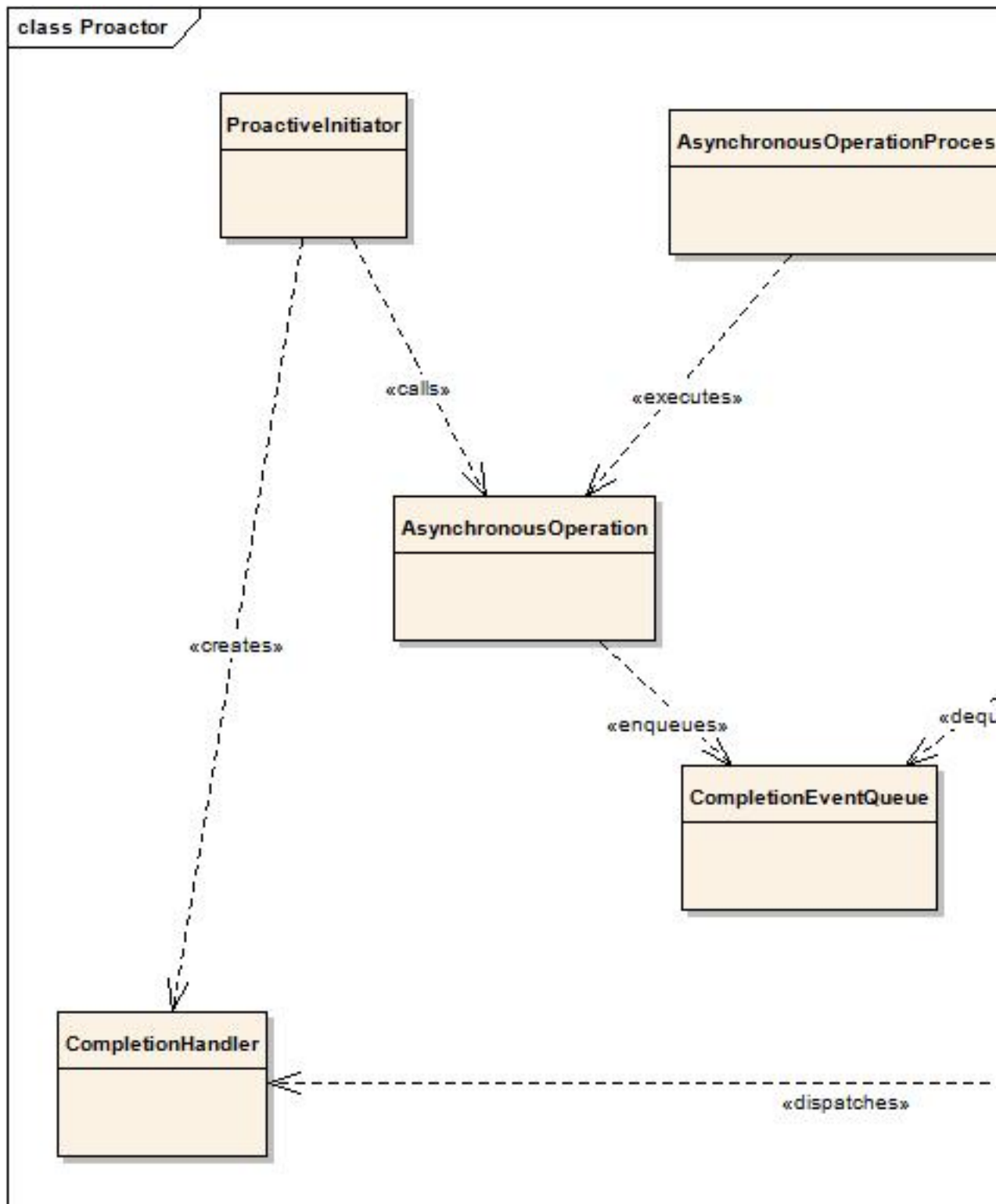
This is the design pattern behind Boost.Asio. See: Boost.Asio documentation [http://www.boost.org/doc/libs/1_57_0/doc/html/boost_asio/overview/core/async.html] for a full explanation. Boost Asynchronous is very similar. It supports enqueueing asynchronous operations and waiting for callbacks, offering extensions: safe callbacks, threadpools, proxies, etc.

Chapter 2. Features of Boost.Asynchronous

Thread world

Extending Active Objects with more servants within a thread context

A commonly cited drawback of Active Objects is that they are awfully expensive. A thread per object is really a waste of ressources. Boost.Asynchronous extends this concept by allowing an unlimited number of objects to live within a single thread context, thus amortizing the costs. It even provides a way for n Active Objects to share m threads while still being called single thread. This allows tuning thread usage.

As many objects are potentially living in a thread context, none should be allowed to process long-lasting tasks as it would reduce reactivity of the whole component. In this aspect, Asynchronous' philosophy is closer to a Proactor.

As long-lasting tasks do happen, Boost.Asynchronous provides several implementations of threadpools and the needed infrastructure to make it safe to post work to threadpools and get aynchronously a safe callback. It also provides safe mechanisms to shutdown Thread worlds and threadpools.

Better Architecture

We all learned in our design books that a software should be organized into layers. This is, however, easier said than done, single-threaded, but much worse when layers are having their own threads. Let's say, layer A is on top and basing itself on layer B. A creates B and keeps it alive as long as it lives itself. A and B are each composed of hundreds of classes / objects. Our standard communication is A => B, meaning A gives orders to B, which executes them. This is the theory. Unfortunately, B needs to give answers, usually delayed, to A. Unfortunately, A and B live in different threads. This means mutexes. Ouch. Now we are forced to check every class of A and protect it. Worse, the object of A getting an answer might have long been destroyed. Ouch again. What to do? We could keep the object of A alive in the callback of B. But then we have a dependency B -> A. Ouch again, bad design. We can also hide the dependency using some type erasure mechanism. We still have a logical one as B keeps its owner, A, alive. Then, we can use a weak_ptr so that B does not keep A alive. But when we lock, we do keep A alive. It's for a short time, but what if A is shutting down? It's lost, our layered design is broken.

Asynchronous is more that a library providing a better std::async or some parallel algorithms, it's first of all an architectural tool. In the above case, we will decide that every layer will live in its own thread(s), called schedulers in Asynchronous language. Deciding in which thread an object "lives" is a key point of a good design. Then the top layer, A, will make a request to B, aking a future as a result, or much better, providing a callback. Asynchronous offers a callback safe in two ways: thread-safe and checking the lifetime of the callback target. This callback is provided by `make_safe_callback`. This simple tool is a major help in making a safe and efficient design.

Shutting down

Shutting down a thread turns out to be harder in practice than expected, as shown by several posts of surprise on the Boost mailing lists when Boost.Thread tried to match the C++ Standard. Asynchronous hides all these ugly details. What users see is a scheduler proxy object, which can be shared by any number of objects, and running any number of threads, managed by a scheduler. The scheduler proxy object manages the lifetime of the scheduler.

When the last instance of the scheduler object is destroyed, the scheduler thread is stopped. When the last instance of a scheduler proxy is destroyed, the scheduler thread is joined. It's as simple as that. This makes threads shared objects.

Object lifetime

There are subtle bugs when living in a multithreaded world. Consider the following class:

```
struct Unsafe
{
  void foo()
  {
    m_mutex.lock();
    // call private member
    m_mutex.unlock();
  }
private:
  void foobar()
  {
    //we are already locked when called, do something while locked
  }
  boost::mutex m_mutex;
};
```

This is called a thread-safe interface pattern. Public members lock, private do not. Simple enough. Unfortunately, it doesn't fly.

First one has the risk of deadlock if a private member calls a public one while being called from another public member. If we forget to check one path of execution within a class implementation, we get a deadlock. We'll have to test every single path of execution to prove our code is correct. And this at every commit.

Usually, for any complex class, where there's a mutex, there is a race or a deadlock...

But even worse, the principle itself is not correct in C++. It supposes that a class can protect itself. Well, no, it can't. Why? One cannot protect the destructor. If the object (and the mutex) gets destroyed when a thread waits for it in foo(), we get a crash or an exception. We can mitigate this with the use of a shared_ptr, then we have no destructor call while someone waits for the mutex. Unfortunately, we still have a risk of a signal, callback, etc. all those things mixing badly with threads. And if we use too many shared_ptr's, we start having lifetime issues or leaks.

There are more lifetime issues, even without mutexes or threads. If you have ever used Boost.Asio, a common mistake and an easy one is when a callback is called in the proactor thread after an asynchronous operation, but the object called is long gone and the callback invalid. Asynchronous provides **trackable_servant** which makes sure that a callback is not called if the object which called the asynchronous operation is gone. It also prevents a task posted in a threadpool to be called if this condition occurs, which improves performance. Asynchronous also provides a safe callback for use as Boost.Asio or similar asynchronous libraries.

Servant Proxies

Asynchronous offers `servant_proxy`, which makes the outside world call members of a servant as if it was not living in an ActiveObject. It looks like a thread-safe interface, but safe from deadlock and race conditions.

Interrupting

Or how to catch back if you're drowning.

Let's say you posted so many tasks to your threadpool that all your cores are full, still, your application is slipping more and more behind plan. You need to give up some tasks to catch back a little.

Asynchronous can give us an interruptible cookie when we post a task to a scheduler, and we can use it to **stop a posted task**. If not running yet, the task will not start, if running, it will stop at the next interruption point, in the sense of the Boost.Thread documentation [http://www.boost.org/doc/libs/1_54_0/doc/html/thread/thread_management.html#thread.thread_management.tutorial.interruption]. Diagnostics will show that the task was interrupted.

Diagnostics

Finding out how good your software is doing is not an easy task. Developers are notoriously bad at it. You need to add lots of logging to find out which function call takes too long and becomes a bottleneck. Finding out the minimum required hardware to run your application is even harder.

Asynchronous design helps here too. By logging the required time and the frequency of tasks, it is easy to find out how many cores are needed. Bottlenecks can be found by logging what the Thread world is doing and how long. Finally, designing the asynchronous Thread world as state machines and logging state changes will allow a better understanding of your system and make visible potential for concurrency. Even for non-parallel algorithms, finding out, using a state machine, the earliest point a task can be thrown to a threadpool will give some low-hanging-fruit concurrency. Throw enough tasks to the threadpool and manage this with a state machine and you might use your cores with little effort. Parallelization can then be used later on by logging which tasks are worth parallelized.

Asynchronous offers **tools generating nice HTML outputs** for every schedulers, including waiting and execution times of tasks, histograms, etc.

Continuations

Callbacks are great when you have a complex flow of operations which require a state machine for management, however there are cases where callbacks are not an ideal solution. Either because your application would require a constant switching of context between single-threaded and parallel schedulers, or because the single-threaded scheduler might be busy, which would delay completion of the algorithm. A known example of this is a parallel fibonacci. In this case, one can register a **continuation**, which is to be executed upon completion of one or several tasks.

This mechanism is flexible so that you can use it with futures coming from another library, thus removing any need for a `wait_for_all(futures...)` or a `wait_for_any(futures...)`.

Want more power? What about extra machines?

What to do if your threadpools are using all of your cores but there simply are not enough cores for the job? Buy more cores? Unfortunately, the number of cores a single-machine can use is limited, unless you have unlimited money. A dual 6-core Xeon, 24 threads with hyperthreading will cost much more than 2 x 6-core i7, and will usually have a lesser clock frequency and an older architecture.

The solution could be: start with the i7, then if you need more power, add some more machines which will steal jobs from your threadpools using **TCP**. This can be done quite easily with Asynchronous.

Want to build your own hierarchical network of servers? It's hard to make it easier.

Parallel algorithms

The library also comes with **non-blocking algorithms** with iterators or ranges, partial support for TCP, which fit well in the asynchronous system, with more to come. If you want to contribute some more, be welcome. At the moment, the library offers:

- most STL algorithms

- parallel_for / parallel_for_each

- parallel_reduce

- parallel_extremum

- parallel_find_all

- parallel_invoke

- parallel_sort , parallel_quicksort

- parallel_scan

- parallelized Boost.Geometry algorithms for polygons (parallel_union, parallel_intersection, parallel_geometry_intersection_of_x, parallel_geometry_union_of_x)

Task Priority

Asynchronous offers this possibility for all schedulers at low performance cost. This means you not only have the possibility to influence task execution order in a threadpool but also in Active Objects.

This is achieved by posting a task to the queue with the corresponding priority. It is also possible to get it even more fine-grained by using a sequence of queues, etc.

Integrating with Boost.Asio

Asynchronous offers a Boost.Asio based **scheduler** allowing you to easily write a Servant using Asio, or an Asio based threadpool. An advantage is that you get safe callbacks and easily get your Asio application to scale. Writing a server has never been easier.

Asynchronous also uses Boost.Asio to provide a timer with callbacks.

Integrating with Qt

What about getting the power of Asynchronous within a Qt application? Use Asynchronous' threadpools, algorithms and other cool features easily.

Work Stealing

Work stealing is supported both within the threads of a threadpool but also between different threadpools. Please have a look at Asynchronous' composite scheduler.

Extending the library

Asynchronous has been written with the design goal of allowing anybody to extend the library. In particular, the authors are hoping to be offered the following extensions:

- More schedulers, threadpools

- Queues

- Parallel algorithms

- Integration with other libraries

Desig

servant2_proxy

«shared»

«shared»

any_shared_scheduler

«lock»

an

«shared»

any_shared_scheduler_proxy

«post(callback)»

actually, another
any_shared_scheduler_proxy

composite_threadpool_scheduler

1
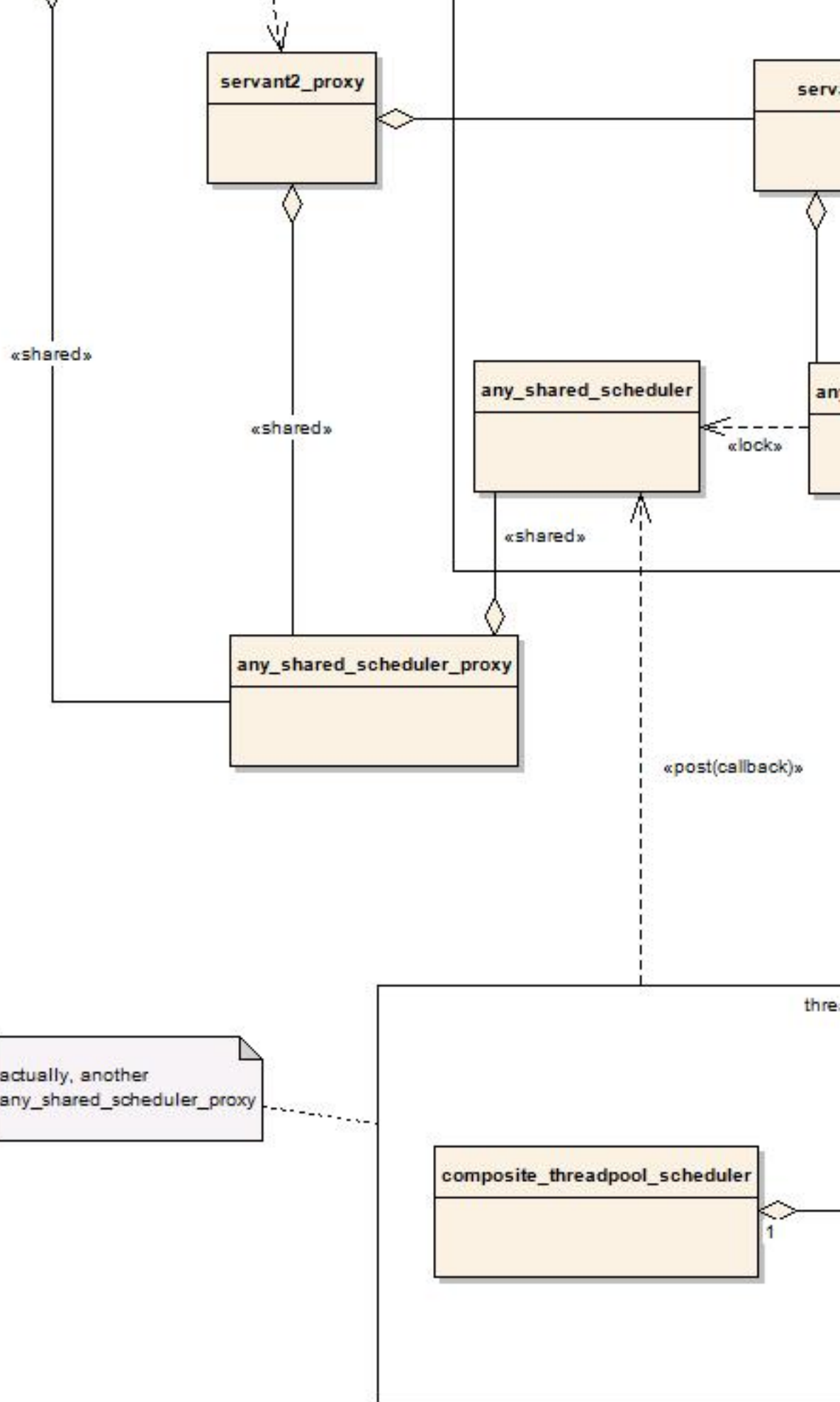
thre

serv

This diagram shows an overview of the design behind Asynchronous. One or more Servant objects live in a single-theaded world, communicating with the outside world only through one or several queues, from which the single-threaded scheduler pops tasks. Tasks are pushed by calling a member on a proxy object.

Like an Active Object, a client uses a proxy (a shared object type), which offers the same members as the real servant, with the same parameters, the only difference being the return type, a std::future<R>, with R being the return type of the servant's member. All calls to a servant from the client side are posted, which includes the servant constructor and destructor. When the last instance of a servant is destroyed, be it used inside the Thread world or outside, the servant destructor is posted.

any_shared_scheduler is the part of the Active Object scheduler living inside the Thread world. Servants do not hold it directly but hold an any_weak_scheduler instead. The library will use it to create a posted callback when a task executing in a worker threadpool is completed.

Shutting down a Thread world is done automatically by not needing it. It happens in the following order:

• While a servant proxy is alive, no shutdown

• When the last servant proxy goes out of scope, the servant destructor is posted.

• if jobs from servants are running in a threadpool, they get a chance to stop earlier by running into an interruption point or will not even start.

• threadpool(s) is (are) shut down.

• The Thread world scheduler is stopped and its thread terminates.

• The last instance of any_shared_scheduler_proxy goes out of scope with the last servant proxy and joins.

It is usually accepted that threads are orthogonal to an OO design and therefore are hard to manage as they don't belong to an object. Asynchronous comes close to this: threads are not directly used, but instead owned by a scheduler, in which one creates objects and tasks.

Part II. User Guide

Table of Contents

Chapter 3. Using Asynchronous

Definitions

Scheduler

Object having 0..n threads, executing jobs or callbacks. Stops owned threads when destroyed.

Thread World (also known as Appartment)

A "thread world" is a world defined by a (single threaded) scheduler and all the objects which have been created, are living and destroyed within this context. It is usually agreed on that objects and threads do not mix well. Class diagrams fail to display both as these are orthogonal concepts. Asynchronous solves this by organizing objects into worlds, each living within a thread. This way, life cycles issues and the question of thread access to objects are solved. It is similar to the Active Object pattern, but with n Objects living within a thread.

Weak Scheduler

A weak_ptr to a shared scheduler. Does not keep the Scheduler alive.

Trackable Servant

Object living in a (single-threaded) scheduler, starting tasks and handling callbacks.

Queue

Holds jobs for a scheduler to execute. Used as communication mean between Schedulers / Worlds

Servant Proxy

A thread-safe object looking like a Servant and serializing calls to it.

Scheduler Shared Proxy

Object holding a scheduler and interfacing with it. The last instance joins the threads of the scheduler.

Posting

Enqueueing a job into a Scheduler's queue.

Hello, asynchronous world

The following code shows a very basic usage (a complete example here [examples/ example_post_future.cpp]), this is not really asynchronous yet:

```
#include <boost/asynchronous/scheduler/threadpool_scheduler.hpp>
#include <boost/asynchronous/queue/lockfree_queue.hpp>
#include <boost/asynchronous/scheduler_shared_proxy.hpp>
#include <boost/asynchronous/post.hpp>
struct void_task
```

```
{
    void operator()()const
    {
        std::cout << "void_task called" << std::endl;
    }
};
struct int_task
{
    int operator()()const
    {
        std::cout << "int_task called" << std::endl;
        return 42;
    }
};

// create a threadpool scheduler with 3 threads and communicate with it using a
// we use auto as it is easier than boost::asynchronous::any_shared_scheduler_p
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::threadpool_scheduler<
                                boost::asynchronous::lockfree_queue<> >(3));
// post a simple task and wait for execution to complete
std::future<void> fuv = boost::asynchronous::post_future(scheduler, void_task()
fuv.get();
// post a simple task and wait for result
std::future<int> fui = boost::asynchronous::post_future(scheduler, int_task());
int res = fui.get();
```

Of course this works with C++11 lambdas:

```
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::threadpool_scheduler<
                                boost::asynchronous::lockfree_queue<> >(3));
// post a simple task and wait for execution to complete
std::future<void> fuv = boost::asynchronous::post_future(scheduler, [](){std::co
fuv.get();
// post a simple task and wait for result
std::future<int> fui = boost::asynchronous::post_future(scheduler, [](){std::co
int res = fui.get();
```

boost::asynchronous::post_future posts a piece of work to a threadpool scheduler with 3 threads and using a lockfree_queue. We get a std::future<the type of the task return type>.

This looks like much std::async, but we're just getting started. Let's move on to something more asynchronous.

A servant proxy

We now want to create a single-threaded scheduler, populate it with some servant(s), and exercise some members of the servant from an outside thread. We first need a servant:

```
struct Servant
{
    Servant(int data): m_data(data){}
    int doIt()const
    {
        std::cout << "Servant::doIt with m_data:" << m_data << std::endl;
        return 5;
```

```
    }
    void foo(int& i)const
    {
        std::cout << "Servant::foo with int:" << i << std::endl;
        i = 100;
    }
    void foobar(int i, char c)const
    {
        std::cout << "Servant::foobar with int:" << i << " and char:" << c <<std
    }
    int m_data;
};
```

We now create a proxy type to be used in other threads:

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    // forwarding constructor. Scheduler to servant_proxy, followed by arguments
    template <class Scheduler>
    ServantProxy(Scheduler s, int data):
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s, data)
    {}
    // the following members must be available "outside"
    // foo and foobar, just as a post (no interesting return value)
    BOOST_ASYNC_POST_MEMBER(foo)
    BOOST_ASYNC_POST_MEMBER(foobar)
    // for doIt, we'd like a future
    BOOST_ASYNC_FUTURE_MEMBER(doIt)
};
```

Let's use our newly defined proxy:

```
int something = 3;
{
    auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::single_thread_scheduler<
                            boost::asynchronous::lockfree_queue<> >);

    {
        // arguments (here 42) are forwarded to Servant's constructor
        ServantProxy proxy(scheduler,42);
        // post a call to foobar, arguments are forwarded.
        proxy.foobar(1,'a');
        // post a call to foo. To avoid races, the reference is ignored.
        proxy.foo(something);
        // post and get a future because we're interested in the result.
        std::future<int> fu = proxy.doIt();
        std::cout<< "future:" << fu.get() << std::endl;
    }// here, Servant's destructor is posted and waited for
}// scheduler is gone, its thread has been joined
std::cout<< "something:" << something << std::endl; // something was not changed
```

We can call members on the proxy, almost as if they were called on Servant. The library takes care of the posting and forwarding the arguments. When required, a future is returned. Stack unwinding works, and when the servant proxy goes out of scope, the servant destructor is posted. When the scheduler goes out of scope, its thread is stopped and joined. The queue is processed completely first. Of course, as many servants as desired can be created in this scheduler context. Please have a look at the complete example [examples/example_simple_servant.cpp].

Using a threadpool from within a servant

If you remember the principles of Asynchronous, blocking a single-thread scheduler is taboo as it blocks the thread doing all the management of a system. But what to do when one needs to execute long tasks? Asynchronous provides a whole set of threadpools. A servant posts something to a threadpool, provides a callback, then gets a result. Wait a minute. Callback? Is this not thread-unsafe? Why not threadpools with futures, like usual? Because in a perfectly asynchronous world, waiting for a future means blocking a servant scheduler. One would argue that it is possible not to block on the future, and instead ask if there is a result. But frankly, polling is not a nice solution either.

And what about thread-safety? Asynchronous takes care of this. A callback is never called from a threadpool, but instead posted back to the queue of the scheduler which posted the work. All the servant has to do, is to do nothing and wait until the callback is executed. Note that this is not the same as a blocking wait, the servant can still react to events.

Clearly, this brings some new challenges as the flow of control gets harder to follow. This is why a servant is often written using state machines. The (biased) author suggests to have a look at the Meta State Machine library [http://www.boost.org/doc/libs/1_59_0/libs/msm/doc/HTML/index.html] , which plays nicely with Asynchronous.

But what about the usual proactor issues (crashes) when the servant has long been destroyed when the callback is posted. Gone. Asynchronous `trackable_servant` post_callback ensures that a callback is not called if the servant is gone. Better even, if the servant has been destroyed, an unstarted posted task will not be executed.

What about another common issue? If one posts a task, say a lambda, which captures a shared_ptr to an object per value, and this object is a boost::signal? Then when the task object has been executed and is destroyed, one could have a race on the signal deregistration. But again no. Asynchronous ensures that a task created within a scheduler context gets destroyed in this context.

This is about the best protection one can get. What Asynchronous cannot protect from are self-made races within a task (if you post a task with a pointer to the servant, you're on your own and have to protect your servant). A good rule of thumb is to consider data passed to a task as moved or passed by value. To support this, Asynchronous does not copy tasks but moves them.

Armed with these protections, let's give a try to a threadpool, starting with the most basic one, `threadpool_scheduler` (more to come):

```
struct Servant : boost::asynchronous::trackable_servant<>
{
    Servant(boost::asynchronous::any_weak_scheduler<> scheduler)
        : boost::asynchronous::trackable_servant<>(scheduler,
                                        // threadpool with 3 threads and
                                        boost::asynchronous::create_shar
                                            new boost::asynchronous::thr
                                                boost::asynchronous:
    // call to this is posted and executes in our (safe) single-thread scheduler
    void start_async_work()
    {
        //ok, let's post some work and wait for an answer
        post_callback(
                    [](){std::cout << "Long Work" << std::endl;}, // work, do n
                    [/*this*/](boost::asynchronous::expected<void>){...}// call
        );
    }
};
```

We now have a servant, living in its own thread, which posts some long work to a three-thread-threadpool and gets a callback, but only if still alive. Similarly, the long work will be executed by the

threadpool only if Servant is alive by the time it starts. Everything else stays the same, one creates a proxy for the servant and posts calls to its members, so we'll skip it for conciseness, the complete example can be found here [examples/example_post_trackable_threadpool.cpp].

A servant using another servant proxy

Often, in a layered design, you'll need that a servant in a single-threaded scheduler calls a member of a servant living in another one. And you'll want to get a callback, not a future, because you absolutely refuse to block waiting for a future (and you'll be very right of course!). Ideally, except for main(), you won't want any of your objects to wait for a future. There is another servant_proxy macro for this, BOOST_ASYNC_UNSAFE_MEMBER(unsafe because you get no thread-safety from if and you'll take care of this yourself, or better, `trackable_servant` will take care of it for you, as follows):

```cpp
// Proxy for a basic servant
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s, int data):
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s, data)
    {}
    BOOST_ASYNC_UNSAFE_MEMBER(foo)
    BOOST_ASYNC_UNSAFE_MEMBER(foobar)
};

// Servant using the first one
struct Servant2 : boost::asynchronous::trackable_servant<>
{
    Servant2(boost::asynchronous::any_weak_scheduler<> scheduler,ServantProxy w
        :boost::asynchronous::trackable_servant<>(scheduler)
        ,m_worker(worker) // the proxy allowing access to Servant
    void doIt()
    {
        call_callback(m_worker.get_proxy(), // Servant's outer proxy, for post
                      m_worker.foo(), // what we want to call on Servant
                      // callback functor, when done.
                      [](boost::asynchronous::expected<int> result){...} );// e
    }
};
```

Call of `foo()` will be posted to `Servant`'s scheduler, and the callback lambda will be posted to `Servant2` when completed. All this thread-safe of course. Destruction is also safe. When `Servant2` goes out of scope, it will shutdown `Servant`'s scheduler, then will his scheduler be shutdown (provided no more object is living there), and all threads joined. The complete example [examples/example_two_simple_servants.cpp] shows a few more calls too.

Asynchronous offers a different syntax to achieve the same result. Which one you use is a matter of taste, both are equivalent. The second method is with BOOST_ASYNC_MEMBER_UNSAFE_CALLBACK(_LOG if you need logging). It takes a callback as argument, other arguments are forwarded. Combined with `make_safe_callback`, one gets the same effect (safe call) as above.

```cpp
// Proxy for a basic servant
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s, int data):
```

```
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s, data)
    {}
    BOOST_ASYNC_MEMBER_UNSAFE_CALLBACK(foo) // say, foo takes an int as argument
};

// Servant using the first one
struct Servant2 : boost::asynchronous::trackable_servant<>
{
    Servant2(boost::asynchronous::any_weak_scheduler<> scheduler,ServantProxy w
        :boost::asynchronous::trackable_servant<>(scheduler)
        ,m_worker(worker) // the proxy allowing access to Servant
    void doIt()
    {
        m_worker.foo(make_safe_callback([](boost::asynchronous::expected<void>
                                        {/* callback code*/}),
                     42 /* arguments of foo*/);
    }
};
```

Interrupting tasks

Let's imagine that a manager object (a state machine for example) posted some long-lasting work to a threadpool, but this long-lasting work really takes too long. As we are in an asynchronous world and non-blocking, the manager object realizes there is a problem and decides the task must be stopped otherwise the whole application starts failing some real-time constraints (how would we do if we were blocked, waiting for a future?). This is made possible by using another form of posting, getting a handle, on which one can require interruption. As Asynchronous does not kill threads, we'll use one of Boost.Thread predefined interruption points. Supposing we have well-behaved tasks, they will be interrupted at the next interruption point if they started, or if they did not start yet because they are waiting in a queue, then they will never start. In this example [examples/example_interrupt.cpp], we have very little to change but the post call. We use `interruptible_post_callback` instead of `post_callback`. We get an `any_interruptible object`, which offers a single `interrupt()` member.

```
struct Servant : boost::asynchronous::trackable_servant<>
{
    ... // as usual
    void start_async_work()
    {
        // start long interruptible tasks
        // we get an interruptible handler representing the task
        boost::asynchronous::any_interruptible interruptible =
        interruptible_post_callback(
                // interruptible task
                [](){
                    std::cout << "Long Work" << std::endl;
                    boost::this_thread::sleep(boost::posix_time::milliseconds(10
                // callback functor.
                [](boost::asynchronous::expected<void> ){std::cout << "Callback
        );
        // let the task start (not sure but likely)
        // if it had no time to start, well, then it will never.
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
        // actually, we changed our mind and want to interrupt the task
        interruptible.interrupt();
        // the callback will likely never be called as the task was interrupted
    }
};
```

Logging tasks

Developers are notoriously famous for being bad at guessing which part of their code is inefficient. This is bad in itself, but even worse for a control class like our post-callback servant as it reduces responsiveness. Knowing how long a posted tasks or a callback lasts is therefore very useful. Knowing how long take tasks executing in the threadpools is also essential to plan what hardware one needs for an application(4 cores? Or 100?). We need to know what our program is doing. Asynchronous provides logging per task to help there. Let's have a look at some code. It's also time to start using our template parameters for `trackable_servant`, in case you wondered why they are here.

```
// we will be using loggable jobs internally
typedef boost::asynchronous::any_loggable<std::chrono::high_resolution_clock> se
// the type of our log
typedef std::map<std::string,std::list<boost::asynchronous::diagnostic_item<std

// we log our scheduler and our threadpool scheduler (both use servant_job)
struct Servant : boost::asynchronous::trackable_servant<servant_job,servant_job
{
    Servant(boost::asynchronous::any_weak_scheduler<servant_job> scheduler) //s
        : boost::asynchronous::trackable_servant<servant_job,servant_job>(sched
                                        boost::asynchronous::create_share
                                            // threadpool with 3 threads
                                            // Furthermore, it logs poste
                                            new boost::asynchronous::thre
                                                //servant_job is our
                                                boost::asynchronous:
    void start_async_work()
    {
        post_callback(
                // task posted to threadpool
                [](){...}, // will return an int
                [](boost::asynchronous::expected<int> res){...},// callback func
                // the task / callback name for logging
                "int_async_work"
        );
    }
    // we happily provide a way for the outside world to know what our threadpo
    // get_worker is provided by trackable_servant and gives the proxy of our t
    diag_type get_diagnostics() const
    {
        return (*get_worker()).get_diagnostics();
    }
};
```

The proxy is also slightly different, using a _LOG macro and an argument representing the name of the task.

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s):
        boost::asynchronous::servant_proxy<ServantProxy,Servant,servant_job>(s)
    {}
    // the _LOG macros do the same as the others, but take an extra argument, t
    BOOST_ASYNC_FUTURE_MEMBER_LOG(start_async_work,"proxy::start_async_work")
    BOOST_ASYNC_FUTURE_MEMBER_LOG(get_diagnostics,"proxy::get_diagnostics")
};
```

We now can get diagnostics from both schedulers, the single-threaded and the threadpool (as external code has no access to it, we ask Servant to help us there through a get_diagnostics() member).

```
// create a scheduler with logging
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::single_thread_schedule
                                    boost::asynchronous::lockfree_queue<servant_
// create a Servant
ServantProxy proxy(scheduler);
...
// let's ask the single-threaded scheduler what it did.
diag_type single_thread_sched_diag = scheduler.get_diagnostics();
for (auto mit = single_thread_sched_diag.begin(); mit != single_thread_sched_di
{
     std::cout << "job type: " << (*mit).first << std::endl;
     for (auto jit = (*mit).second.begin(); jit != (*mit).second.end();++jit)
     {
          std::cout << "job waited in us: " << std::chrono::nanoseconds((*jit).
          std::cout << "job lasted in us: " << std::chrono::nanoseconds((*jit).
          std::cout << "job interrupted? "  << std::boolalpha << (*jit).is_inte
          std::cout << "job failed? "  << std::boolalpha << (*jit).is_failed()
     }
}
```

It goes similarly with the threapool scheduler, with the slight difference that we ask the Servant to deliver diagnostic information through a proxy member. The complete example [examples/example_log.cpp] shows all this, plus an interrupted job.

Generating HTML diagnostics

We just saw how to programmatically get diagnostics from schedulers. This is very useful, but nobody likes to do it manually, so the authors went the extra mile and provide an HTML formatter for convenience. The included example [examples/example_html_diagnostics.cpp] shows how to use it. In this example, we have a Servant, living in its own single-threaded scheduler called "Servant". It uses a threadpool call "Threadpool". When the Servant's foo() method is called, it executes a parallel_reduce(parallel_for(...)), or whatever you like. These operations are named accordingly. We also create a third scheduler, called "Formatter scheduler", which will be used by the formatter code. Yes, even this scheduler will be logged too. The example creates a Servant, calls foo() on the proxy, sleeps for a while (how long is passed to the example as argument), then generates a first output statistics [examples/in_progress.html]. Depending on the sleep time, the parallel work might or might not be finished, so this is an intermediate result.

We then wait for the tasks to finish, destroy the servant, so that its destructor is logged too, and we generate a final diagnostics [examples/final.html].

The HTML pages display the statistics for all schedulers, including the formatter. It shows with different colors the waiting times of tasks (called Scheduling time), the execution times, successful or failed separately, and the added total time for each task, with max min, average duration. One can also display the full list of tasks and even histograms. As this is a lot of information, it is possible to hide part of it using checkboxes.

One also gets the very useful information of how long are the different scheduler queues, which gives a very good indication of how busy the system is.

Queue container with priority

Sometimes, all jobs posted to a scheduler do not have the same priority. For threadpool schedulers, composite_threadpool_scheduler is an option. For a single-threaded scheduler,

Asynchronous does not provide a priority queue but a queue container, which itself contains any number of queues, of different types if needed. This has several advantages:

- Priority is defined simply by posting to the queue with the desired priority, so there is no need for expensive priority algorithms.

- Reduced contention if many threads of a threadpool post something to the queue of a single-threaded scheduler. If no priority is defined, one queue will be picked, according to a configurable policy, reducing contention on a single queue.

- It is possible to mix queues.

- It is possible to build a queue container of queue containers, etc.

Note: This applies to any scheduler. We'll start with single-threaded schedulers used by managing servants for simplicity, but it is possible to have composite schedulers using queue containers for finest granularity and least contention.

First, we need to create a single-threaded scheduler with several queues for our servant to live in, for example, one threadsafe list and three lockfree queues:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
                        boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::single_thread_schedule
                                boost::asynchronous::any_queue_containe
                    (boost::asynchronous::any_queue_container_config<boost:
                     boost::asynchronous::any_queue_container_config<boost:
                     ));
```

`any_queue_container` takes as constructor arguments a variadic sequence of `any_queue_container_config`, with a queue type as template argument, and in the constructor the number of objects of this queue (in the above example, one `threadsafe_list` and 3 `lockfree_queue` instances, then the parameters that these queues require in their constructor (100 is the capacity of the underlying `boost::lockfree_queue`). This means, that our `single_thread_scheduler` has 4 queues:

- a threadsafe_list at index 1

- lockfree queues at indexes 2,3,4

- >= 4 means the queue with the least priority.

- 0 means "any queue" and is the default

The scheduler will handle these queues as having priorities: as long as there are work items in the first queue, take them, if there are no, try in the second, etc. If all queues are empty, the thread gives up his time slice and sleeps until some work item arrives. If no priority is defined by posting, a queue will be chosen (by default randomly, but this can be configured with a policy). This has the advantage of reducing contention of the queue, even when not using priorities. The servant defines the priority of the tasks it provides. While this might seem surprising, it is a design choice to avoid that the coder using a servant proxy interface would have to think about it, as you will see in the second listing. To define a priority for a servant proxy, there is a second field in the macros:

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s):
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s)
    {}
    BOOST_ASYNC_SERVANT_POST_CTOR(3)
    BOOST_ASYNC_SERVANT_POST_DTOR(4)
```

```
        BOOST_ASYNC_FUTURE_MEMBER(start_async_work,1)
};
```

BOOST_ASYNC_FUTURE_MEMBER and other similar macros can be given an optional priority parameter, in this case 1, which is our threadsafe list. Notice how you can then define the priority of the posted servant constructor and destructor.

```
ServantProxy proxy(scheduler);
std::future<std::future<int>> fu = proxy.start_async_work();
```

Calling our proxy member stays unchanged because the macro defines the priority of the call.

We also have an extended version of `post_callback`, called by a servant posting work to a threadpool:

```
post_callback(
        [](){return 42;},// work
        [this](boost::asynchronous::expected<int> res){}// callback functor.
        ,"",
        2, // work prio
        2  // callback prio
);
```

Note the two added priority values: the first one for the task posted to the threadpool, the second for the priority of the callback posted back to the servant scheduler. The string is the log name of the task, which we choose to ignore here.

The priority is in any case an indication, the scheduler is free to ignore it if not supported. In the example [examples/example_queue_container.cpp], the single threaded scheduler will honor the request, but the threadpool has a normal queue and cannot honor the request, but a threadpool with an `any_queue_container` or a `composite_threadpool_scheduler` can. The same example [examples/example_queue_container_log.cpp] can be rewritten to also support logging.

`any_queue_container` has two template arguments. The first, the job type, is as always by default, a callable (`any_callable`) job. The second is the policy which Asynchronous uses to find the desired queue for a job. The default is `default_find_position`, which is as described above, 0 means any position, all other values map to a queue, priorities >= number of queues means last queue. Any position is by default random (`default_random_push_policy`), but you might pick `sequential_push_policy`, which keeps an atomic counter to post jobs to queues in a sequential order.

If you plan to build a queue container of queue containers, you'll probably want to provide your own policy.

Multiqueue Schedulers' priority

A multiqueue_... threadpool scheduler has a queue for each thread. This reduces contention, making these faster than single queue schedulers, like threadpool_scheduler. Furthermore, these schedulers support priority: the priority given in post_future or post_callback is the (1-based) position of the queue we want to post to. 0 means "any queue". A queue of priority 1 has a higher priority than a queue with priority 2, etc.

Each queue is serving one thread, but threads steal from each other's queue, according to the priority.

Threadpool Schedulers with several queues

A queue container has advantages (different queue types, priority for single threaded schedulers) but also disadvantages (takes jobs from one end of the queue, which means potential cache misses, more typing work). If you don't need different queue types for a threadpool but want to reduce contention, multiqueue schedulers are for you. A normal `threadpool_scheduler` has x threads and one

queue, serving them. A `multiqueue_threadpool_scheduler` has x threads and x queues, each serving a worker thread. Each thread looks for work in its queue. If it doesn't find any, it looks for work in the previous one, etc. until it finds one or inspected all the queues. As all threads steal from the previous queue, there is little contention. The construction of this threadpool is very similar to the simple `threadpool_scheduler`:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::create_shared_scheduler_proxy(
                // 4 threads and 4 lockfree queues of 10 capacity
                new boost::asynchronous::multiqueue_threadpool_scheduler<boost:
```

The first argument is the number of worker threads, which is at the same time the number of queues. As for every scheduler, if the queue constructor takes arguments, they come next and are forwarded to the queue.

This is the <u>advised</u> scheduler for standard cases as it offers lesser contention and task stealing between the queues it uses for task transfer.

Limitation: these schedulers cannot have 0 thread like their single-queue counterparts.

Composite Threadpool Scheduler

Usage

When a project becomes more complex, having a single threadpool for the whole application does not offer enough flexibility in load planning. It is pretty hard to avoid either oversubscription (more busy threads than available hardware threads) or undersubscription. One would need one big threadpool with exactly the number of threads available in the hardware. Unfortunately, if we have a hardware with, say 12 hardware threads, parallelizing some work using all 12 might be slowlier than using only 8. One would need different threadpools of different number of threads for the application. This, however, has the serious drawback that there is a risk that some threadpools will be in overload, while others are out of work unless we have work stealing between different threadpools.

The second issue is task priority. One can define priorities with several queues or a queue container, but this ensures that only highest priority tasks get executed if the system is coming close to overload. Ideally, it would be great if we could decide how much compute power we give to each task type.

This is what `composite_threadpool_scheduler` solves. This pool supports, like any other pool, the `any_shared_scheduler_proxy`concept so you can use it in place of the ones we used so far. The pool is composed of other pools (`any_shared_scheduler_proxy` pools). It implements work stealing between pools if a) the pools support it and b) the queue of a pool also does. For example, we can create the following worker pool made of 3 sub-pools:

```
// create a composite threadpool made of:
// a multiqueue_threadpool_scheduler, 1 thread, with a lockfree_queue of capaci
// This scheduler does not steal from other schedulers, but will lend its queue
boost::asynchronous::any_shared_scheduler_proxy<> tp = boost::asynchronous::cre
                new boost::asynchronous::multiqueue_threadpool_scheduler<boost::

// a stealing_multiqueue_threadpool_scheduler, 3 threads, each with a threadsaf
// this scheduler will steal from other schedulers if it can. In this case it w
boost::asynchronous::any_shared_scheduler_proxy<> tp2 = boost::asynchronous::cr
                    new boost::asynchronous::stealing_multiqueue_threadpool_sch

// a multiqueue_threadpool_scheduler, 4 threads, each with a lockfree_spsc_queu
// this is safe because there will be no stealing as the queue does not support
boost::asynchronous::any_shared_scheduler_proxy<> tp3 = boost::asynchronous::cr
                new boost::asynchronous::multiqueue_threadpool_scheduler<boost::
```

```
// create a composite pool made of the 3 previous ones
boost::asynchronous::any_shared_scheduler_proxy<> tp_worker =
                boost::make_shared<boost::asynchronous::composite_threadpool_sched
```

We can use this pool:

• As a big worker pool. In this case, the priority argument we use for posting refers to the (1-based) index of the subpool (post_callback(func1,func2,"task name",**1**,0);). "1" means post to the first pool. But another pool could steal the work.

• As a pool container, but different parts of the code will get to see only the subpools. For example, the pools tp, tp2 and tp3 can still be used independently as a worker pool. Calling composite_threadpool_scheduler<>::get_scheduler(std::size_t index_of_pool) will also give us the corresponding pool (1-based, as always).

Another example of why to use this pool is reusing threads allocated to an asio-based communication for helping other schedulers. Addng an asio scheduler to a composite pool will allow the threads of this scheduler to help (steal) other pools when no communication is currently happening.

Stealing is done with priority. A stealing pool first tries to steal from the first pool, then from the second, etc.

The following example [examples/example_composite_threadpool.cpp] shows a complete servant implementation, and the **ASIO section** will show how an ASIO pool can steal.

The threadpool schedulers we saw so far are not stealing from other pools. The single-queue schedulers are not stealing, and the multiqueue schedulers steal from the queues of other threads of the same pool. The scheduler-stealing schedulers usually indicate this by appending a `stealing_` to their name:

• `stealing_threadpool_scheduler` is a `threadpool_scheduler` which steals from other pools.

• `stealing_multiqueue_threadpool_scheduler` is a `multiqueue_threadpool scheduler` which steals from other pools.

• `asio_scheduler` steals.

The only difference with their not stealing equivalent is that they steal from other schedulers. To achieve this, they need a composite_scheduler to tell them from which schedulers they can steal.

Not all schedulers offer to be stolen from. A `single_thread_scheduler` does not as it would likely bring race conditions to active objects.

Another interesting usage will be when planning for extra machines to help a threadpool by processing some of the work: work can be stolen from a threadpool by a **tcp_server_scheduler** from which other machines can get it. Just pack both pools in a `composite_threadpool_scheduler` and you're ready to go.

Priority

A composite supports priority. The first pool passed in the constructor of the composite pool has priority 1, the second 2, etc. 0 means "any pool" and n where n > number of pools will me modulo-ed.

Posting to this scheduler using post_future or post_callback using a given priority will post to the according pool. If a pool supports stealing from other pools (stealing_... pools), it will try to steal from other pools, starting with the highest priority, but only if the to be stolen from pools supports it. For example, we try to post to the first pool, callback to any queue.

```
post_callback(
                [](){},// work
                [this](boost::asynchronous::expected<int>){},// callback functor
```

```
                    "", // task and callback name
                    1,  // work priority, highest
                    0   // callback anywhere
    );
```

More flexibility in dividing servants among threads

TODO example and code. We saw how to assign a servant or several servants to a single thread scheduler. We can also create schedulers and divide servants among them. This is very powerful but still has some constraints:

- We need to assign servants to schedulers while what we want is to assign them to threads. We also have to consider how many schedulers to create. This is not very flexible.

- If a servant is taking too long, it blocks all other servants living inside this thread context. This increases latency.

We can increase the flexibility and reduce latency by using a `multiple_thread_scheduler`. This scheduler takes as first argument a number of threads to use and a maximum number of client "worlds" (clients living logically in the same thread context). What it does, is to assign any of its threads to different client worlds, but only one thread can service a world at a time. This means that the thread safety of servants is preserved. At the same time, having any number of threads decreases latency because if a servant keeps its thread busy, it does not block other servants from being serviced. As we can choose the number of threads this scheduler will use, we achieve very fine granularity in planing our thread resources.

Another interesting characteristics of this scheduler is that its threads service its servants in order. If a thread serviced servant x, it next tries to service servant x+1. This makes for good pipelining capabilities as it increases the odds that task is koved from a pipeline stage to the next one by the same thread and will be hot in its cache.

Processor binding

TODO example and code.On many systems, it can improve performance to bind threads to a processor: better cache usage is likely as the OS does not move threads from core to core. Mostly for threadpools this is an option you might want to try.

Usage is very simple. One needs to call `processor_bind(core_index)` on a scheduler proxy. This function takes a single argument, the core to which the first thread of the pool will be bound. The second thread will be bound to core+1, etc.

asio_scheduler

Asynchronous supports the possibility to use Boost.Asio as a threadpool provider. This has several advantages:

- asio_scheduler is delivered with a way to access Asio's io_service from a servant object living inside the scheduler.

- asio_scheduler handles the necessary work for creating a pool of threads for multithreaded-multi-io_service communication.

- asio_scheduler threads implement work-stealing from other Asynchronous schedulers. This allows communication threads to help other threadpools when no I/O communication is happening. This helps reducing thread oversubscription.

- One has all the usual goodies of Asynchronous: safe callbacks, object tracking, servant proxies, etc.

Let's create a simple but powerful example to illustrate its usage. We want to create a TCP client, which connects several times to the same server, gets data from it (in our case, the Boost license will do), then checks if the data is coherent by comparing the results two-by-two. Of course, the client has to be perfectly asynchronous and never block. We also want to guarantee some threads for the communication and some for the calculation work. We also want to communication threads to "help" by stealing some work if necessary.

Let's start by creating a TCP client using Boost.Asio. A slightly modified version of the async TCP client from the Asio documentation will do. All we change is pass it a callback which it will call when the requested data is ready. We now pack it into an Asynchronous trackable servant:

```
// Objects of this type are made to live inside an asio_scheduler,
// they get their associated io_service object from Thread Local Storage
struct AsioCommunicationServant : boost::asynchronous::trackable_servant<>
{
    AsioCommunicationServant(boost::asynchronous::any_weak_scheduler<> scheduler
                             const std::string& server, const std::string& path
        : boost::asynchronous::trackable_servant<>(scheduler)
        , m_client(*boost::asynchronous::get_io_service<>(),server,path)
    {}
    void test(std::function<void(std::string)> cb)
    {
        // just forward call to asio asynchronous http client
        // the only change being the (safe) callback which will be called when
        m_client.request_content(cb);
    }
private:
    client m_client; //client is from Asio example
};
```

The main noteworthy thing to notice is the call to **boost::asynchronous::get_io_service<>()**, which, using thread-local-storage, gives us the io_service associated with this thread (one io_service per thread). This is needed by the Asio TCP client. Also noteworthy is the argument to test(), a callback when the data is available.

Wait a minute, is this not unsafe (called from an asio worker thread)? It is but it will be made safe in a minute.

We now need a proxy so that this communication servant can be safely used by others, as usual:

```
class AsioCommunicationServantProxy: public boost::asynchronous::servant_proxy<
{
public:
    // ctor arguments are forwarded to AsioCommunicationServant
    template <class Scheduler>
    AsioCommunicationServantProxy(Scheduler s,const std::string& server, const 
        boost::asynchronous::servant_proxy<AsioCommunicationServantProxy,AsioCo
    {}
    // we offer a single member for posting
    BOOST_ASYNC_POST_MEMBER(test)
};
```

A single member, test, is used in the proxy. The constructor takes the server and relative path to the desired page. We now need a manager object, which will trigger the communication, wait for data, check that the data is coherent:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
    Servant(boost::asynchronous::any_weak_scheduler<> scheduler,const std::stri
        : boost::asynchronous::trackable_servant<>(scheduler)
```

```
        , m_check_string_count(0)
    {
        // as worker we use a simple threadpool scheduler with 4 threads (0 wou
        auto worker_tp = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<boost::asynchr

        // for tcp communication we use an asio-based scheduler with 3 threads
        auto asio_workers = boost::asynchronous::create_shared_scheduler_proxy(

        // we create a composite pool whose only goal is to allow asio worker t
        m_pools = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::composite_threadpool_scheduler<> (

        set_worker(worker_tp);
        // we create one asynchronous communication manager in each thread
        m_asio_comm.push_back(AsioCommunicationServantProxy(asio_workers,server
        m_asio_comm.push_back(AsioCommunicationServantProxy(asio_workers,server
        m_asio_comm.push_back(AsioCommunicationServantProxy(asio_workers,server
    }
... //to be continued
```

We create 3 pools:

- A worker pool for calculations (page comparisons)

- An asio threadpool with 3 threads in which we create 3 communication objects.

- A composite pool which binds both pools together into one stealing unit. You could even set the
  worker pool to 0 thread, in which case the worker will get its work done when the asio threads have
  nothing to do. Only non- multiqueue schedulers support this. The worker pool is now made to be
  the worker pool of this object using set_worker().

We then create our communication objects inside the asio pool.

Note: asio pools can steal from other pools but not be stolen from. Let's move on to the most interesting
part:

```
void get_data()
{
    // provide this callback (executing in our thread) to all asio servants as
    std::function<void(std::string)> f =
...
    m_asio_comm[0].test(make_safe_callback(f));
    m_asio_comm[1].test(make_safe_callback(f));
    m_asio_comm[2].test(make_safe_callback(f));
}
```

We skip the body of f for the moment. f is a task which will be posted to each communication servant
so that they can do the same work:

- call the same http get on an asio servants

- at each callback, check if we got all three callbacks

- if yes, post some work to worker threadpool, compare the returned strings (should be all the same)

- if all strings equal as they should be, cout the page

All this will be doine in a single functor. This functor is passed to each communication servant, packed
into a make_safe_callback, which, as its name says, transforms the unsafe functor into one which

posts this callback functor to the manager thread and also tracks it to check if still alive at the time of the callback. By calling `test()`, we trigger the 3 communications, and f will be called 3 times. The body of f is:

```
std::function<void(std::string)> f =
                [this](std::string s)
                {
                    this->m_requested_data.push_back(s);
                    // poor man's state machine saying we got the result of our a
                    if (this->m_requested_data.size() == 3)
                    {
                        // ok, this has really been called for all servants, comp
                        // but it could be long, so we will post it to threadpool
                        std::cout << "got all tcp data, parallel check it's corre
                        std::string s1 = this->m_requested_data[0];
                        std::string s2 = this->m_requested_data[1];
                        std::string s3 = this->m_requested_data[2];
                        // this callback (executing in our thread) will be calle
                        auto cb1 = [this,s1](boost::asynchronous::expected<bool>
                        {
                            if (res.get())
                                ++this->m_check_string_count;
                            else
                                std::cout << "uh oh, the pages do not match, data
                            if (this->m_check_string_count ==2)
                            {
                                // we started 2 comparisons, so it was the last o
                                std::cout << "data has been confirmed, here it is
                                std::cout << s1;
                            }
                        };
                        auto cb2=cb1;
                        // post 2 string comparison tasks, provide callback where
                        this->post_callback([s1,s2](){return s1 == s2;},std::move
                        this->post_callback([s2,s3](){return s2 == s3;},std::move
                    }
                };
```

We start by checking if this is the third time this functor is called (this, the manager, is nicely serving as holder, kind of poor man's state machine counting to 3). If yes, we prepare a call to the worker pool to compare the 3 returned strings 2 by 2 (cb1, cb2). Again, simple state machine, if the callback is called twice, we are done comparing string 1 and 2, and 2 and 3, in which case the page is confirmed and cout'ed. The last 2 lines trigger the work and post to our worker pool (which is the threadpool scheduler, or, if stealing happens, the asio pool) two comparison tasks and the callbacks.

Our manager is now ready, we still need to create for it a proxy so that it can be called from the outside world asynchronously, then create it in its own thread, as usual:

```
class ServantProxy : public boost::asynchronous::servant_proxy<ServantProxy,Ser
{
public:
    template <class Scheduler>
    ServantProxy(Scheduler s,const std::string& server, const std::string& path
        boost::asynchronous::servant_proxy<ServantProxy,Servant>(s,server,path)
    {}
    // get_data is posted, no future, no callback
    BOOST_ASYNC_POST_MEMBER(get_data)
};
```

```
...
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                              new boost::asynchronous::single_thread_schedule
                                  boost::asynchronous::threadsafe_list<> >);
{
    ServantProxy proxy(scheduler,"www.boost.org","/LICENSE_1_0.txt");
    // call member, as if it was from Servant
    proxy.get_data();
    // if too short, no problem, we will simply give up the tcp requests
    // this is simply to simulate a main() doing nothing but waiting for a termi
    boost::this_thread::sleep(boost::posix_time::milliseconds(2000));
}
```

As usual, here the complete, ready-to-use example [examples/example_asio_http_client.cpp] and the
implementation of the Boost.Asio HTTP client [examples/asio/asio_http_async_client.hpp].

Timers

Very often, an Active Object servant acting as an asynchronous dispatcher will post tasks which have
to be done until a certain point in the future, or which will start only at a later point. State machines
also regularly make use of a "time" event.

For this we need a timer, but a safe one:

• The timer callback has to be posted to the Active Object thread to avoid races.

• The timer callback shall not be called if the servant making the request has been deleted (it can be
  an awfully long time until the callback).

Asynchronous itself has no timer, but Boost.Asio does, so the library provides a wrapper around it
and will allow us to create a timer using an asio::io_service running in its own thread or in an asio
threadpool, provided by the library.

Constructing a timer

One first needs an `asio_scheduler` with at least one thread:

```
boost::asynchronous::any_shared_scheduler_proxy<> asio_sched = boost::asynchron
```

The Servant living in its ActiveObject thread then creates a timer (as attribute to keep it alive) using
this scheduler and a timer value:

```
 boost::asynchronous::asio_deadline_timer_proxy m_timer (asio_sched,boost::posi
```

It can now start the timer using `trackable_servant` (its base class)`::async_wait`, passing it
a functor call when timer expires / is cancelled:

```
 async_wait(m_timer,
             [](const ::boost::system::error_code& err)
             {
                 std::cout << "timer expired? "<< std::boolalpha << (bool)err <<
             }
             );
```

Canceling or recreating the timer means destroying (and possibly recreating) the timer object:

```
 m_timer =  boost::asynchronous::asio_deadline_timer_proxy(get_worker(),boost::
```

Alternatively, asio_deadline_timer_proxy offers a reset(duration) member, which is more efficient than recreating a proxy. The following example [examples/example_asio_deadline_timer.cpp] displays a servant using an asio scheduler as a thread pool and creating there its timer object. Note how the timer is created using the worker scheduler of its owner.

Continuation tasks

A common limitation of threadpools is support for recursive tasks: tasks start other tasks, which start other tasks and wait for them to complete to do a merge of the part-results. Unfortunately, all threads in the threadpool will soon be busy waiting and no task will ever complete. One can achieve this with a controller object or state machine in a single-threaded scheduler waiting for callbacks, but for very small tasks, using callbacks might just be too expensive. In such cases, Asynchronous provides continuations: a task executes, does something then creates a continuation which will be excuted as soon as all child tasks complete.

General

The Hello World of recursive tasks is a parallel fibonacci. The naive algorithm creates a task calculating fib(n). For this it will start a fib(n-1) and fib(n-2) and block until both are done. These tasks will start more tasks, etc. until a cutoff number, at which point recursion stops and fibonacci is calculated serially. This approach has some problems: to avoid thread explosion, we would need fibers, which are not available in Boost at the time of this writing. Even with fibers, tasks would block, which means interrupting them is not possible, and a stack will have to be paid for both. Performance will also suffer. Furthermore, blocking simply isn't part of the asynchronous philosophy of the library. Let's have a look how callback continuation tasks let us implement a parallel fibonacci.

First of all, we need a serial fibonacci when n is less than the cutoff. This is a classical one:

```
 long serial_fib( long n ) {
    if( n<2 )
        return n;
    else
        return serial_fib(n-1)+serial_fib(n-2);
}
```

We now need a recursive-looking fibonacci task:

```
// our recursive fibonacci tasks. Needs to inherit continuation_task<value type
struct fib_task : public boost::asynchronous::continuation_task<long>
{
    fib_task(long n,long cutoff):n_(n),cutoff_(cutoff){}
    // called inside of threadpool
    void operator()()const
    {
        // the result of this task, will be either set directly if < cutoff, oth
        boost::asynchronous::continuation_result<long> task_res = this_task_res
        if (n_<cutoff_)
        {
            // n < cutoff => execute immediately
            task_res.set_value(serial_fib(n_));
        }
        else
        {
            // n>= cutoff, create 2 new tasks and when both are done, set our re
            boost::asynchronous::create_callback_continuation(
                    // called when subtasks are done, set result of the cal
                    [task_res](std::tuple<boost::asynchronous::expected<long
                    {
                        long r = std::get<0>(res).get() + std::get<1>(res).
```

```
                              task_res.set_value(r);
                          },
                          // recursive tasks
                          fib_task(n_-1,cutoff_),
                          fib_task(n_-2,cutoff_));
            }
        }
        long n_;
        long cutoff_;
};
```

Our task need to inherit `boost::asynchronous::continuation_task<R>` where R is the returned type. This class provides us with `this_task_result()` where we set the task result. This is done either immediately if n < cutoff (first if clause), or (else clause) using a continuation.

If n>= cutoff, we create a continuation. This is a sleeping task, which will get activated when all required tasks complete. In this case, we have two fibonacci sub tasks. The template argument is the return type of the continuation. We create two sub-tasks, for n-1 and n-2 and when they complete, the completion functor passed as first argument is called.

Note that `boost::asynchronous::create_continuation` is a variadic function, there can be any number of sub-tasks. The completion functor takes as single argument a tuple of `expected`, one for each subtask. The template argument of the future is the template argument of `boost::asynchronous::continuation_task` of each subtask. In this case, all are of type long, but it's not a requirement.

When this completion functor is called, we set our result to be result of first task + result of second task.

The main particularity of this solution is that a task does not block until sub-tasks complete but instead provides a functor to be called asynchronously as soon as subtasks complete.

All what we still need to do is create the first task. In the tradition of Asynchronous, we show it inside an asynchronous servant which posts the first task and waits for a callback, but the same is of course possible using `post_future`:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
...
    void calc_fibonacci(long n,long cutoff)
    {
        post_callback(
                // work
                [n,cutoff]()
                {
                    // a top-level continuation is the first one in a recursive ser
                    // Its result will be passed to callback
                    return boost::asynchronous::top_level_callback_continuation<lon
                },
                // callback with fibonacci result.
                [](boost::asynchronous::expected<long> res){...}// callback functor
            );
    }
};
```

We call `post_callback`, which, as usual, ensures that the callback is posted to the right thread and the servant lifetime is tracked. The posted task calls `boost::asynchronous::top_level_callback_continuation<task-return-type>` to create the first, top-level continuation, passing it a first fib_task. This is non-blocking, a special version of `post_callback` recognizes a continuation and will call its callback (with a `expected<task-return-type>`) only when the calculation is finished, not when

the "work" lambda returns. For this to work, **it is essential not to forget the return statement**. Without it, the compiler will unhappily remark that an `expected<void>` cannot be casted to an `expected<long>`, or worse if one expects an `expected<void>`, the callback would be called to early.

As usual, calling get() on the expected is non-blocking, one gets either the result or an exception if thrown by a task.

Please have a look at the complete example [examples/example_fibonacci.cpp].

Logging

What about logging? We don't want to give up this feature of course and would like to know how long all these fib_task took to complete. This is done through minor changes. As always we need a job:

```
typedef boost::asynchronous::any_loggable<std::chrono::high_resolution_clock> s
```

We give the logged name of the task in the constructor of fib_task, for example fib_task_xxx:

```
fib_task(long n,long cutoff)
        : boost::asynchronous::continuation_task<long>("fib_task_" + boost::lex
        ,n_(n),cutoff_(cutoff){}
```

And call `boost::asynchronous::create_continuation_job` instead of `boost::asynchronous::create_continuation`:

```
boost::asynchronous::create_callback_continuation_job<servant_job>(
                        [task_res](std::tuple<boost::asynchronous::expected<lon
                        {
                            long r = std::get<0>(res).get() + std::get<1>(res).
                            task_res.set_value(r);
                        },
                        fib_task(n_-1,cutoff_),
                        fib_task(n_-2,cutoff_)
);
```

Inside the servant we might optionally want the version of post_callback with name, and we need to use `top_level_continuation_job` instead of `top_level_continuation`:

```
post_callback(
            [n,cutoff]()
            {
                return boost::asynchronous::top_level_callback_continuation_
            },// work
            // the lambda calls Servant, just to show that all is safe, Serva
            [this](boost::asynchronous::expected<long> res){...},// callback
            "calc_fibonacci"
        );
```

The previous example has been rewritten with logs and a display of all tasks [examples/example_fibonacci_log.cpp] (beware, with higher fibonacci numbers, this can become a long list).

Limitation: in the current implementation, tasks are logged, but the continuation callback is not. If it might take long, one should post a (loggable) task.

Note: to improve performance, the last task passed to **create_callback_continuation(_job)** is not posted but executed directly so it will execute under the name of the task calling **create_callback_continuation(_job)**.

Important note about exception safety. The passed **expected** contains either a result or an exception. Calling get() will throw contained exceptions. You should catch it, in the continuation callback and

in the task itself. Asynchronous will handle the exception, but it cannot set the **continuation_result**, which will never be set and the callback part of post_callback never called. This simple example does not throw, so we save ourselves the cost, but more complicated algorithms should take care of this.

Creating a variable number of tasks for a continuation

It is sometimes not possible to know at compile-time the number of tasks or even the types of tasks used in the creation of a continuation. In this cases, Asynchronous provides more possibilities:

- Pack all subtasks of a same type into a std::vector, then pass it to create_callback_continuation or create_callback_continuation_job. In this case, we know that these subtasks all have the same type, so our continuation is called with a vector<expected<return_type>>:

```
struct sub_task : public boost::asynchronous::continuation_task<long>
{
    // some task with long as result type
};
struct main_task : public boost::asynchronous::continuation_task<long>
{
  void operator()()
  {
      boost::asynchronous::continuation_result<long> task_res = this_task_resul
      std::vector<sub_task> subs;
      subs.push_back(sub_task());
      subs.push_back(sub_task());
      subs.push_back(sub_task());

      boost::asynchronous::create_callback_continuation(
            [task_res](std::vector<boost::asynchronous::expected<long>> res)
            {
                long r = res[0].get() + res[1].get() + res[2].get();
                task_res.set_value(r);
            },
            std::move(subs));
    }
};
```

- If the subtasks have different type, but a common result type, we can pack them into a std::vector<boost::asynchronous::any_continuation_task<return_type>> instead, the rest of the code staying the same:

```
#include <boost/asynchronous/any_continuation_task.hpp>

struct sub_task : public boost::asynchronous::continuation_task<long>
{
    // some task with long as result type
};
struct main_task2 : public boost::asynchronous::continuation_task<long>
{
    void operator()()
    {
        boost::asynchronous::continuation_result<long> task_res = this_task_re
        std::vector<boost::asynchronous::any_continuation_task<long>> subs;
        subs.push_back(sub_task());
        subs.push_back(sub_task2());
        subs.push_back(sub_task3());
```

```
boost::asynchronous::create_callback_continuation(
    [task_res](std::vector<boost::asynchronous::expected<long>> res)
    {
        long r = res[0].get() + res[1].get() + res[2].get();
        task_res.set_value(r);
    },
    std::move(subs));
    }
};
```

- Of course, if we have continuations in the first place, returned by top_level_callback_continuation<task-return-type> or top_level_callback_continuation<task-return-type>, as all of Asynchronous' algorithms do, these can be packed into a vector as well:

```
struct main_task3 : public boost::asynchronous::continuation_task<long>
{
    void operator()()
    {
        boost::asynchronous::continuation_result<long> task_res = this_task_re
        std::vector<boost::asynchronous::detail::callback_continuation<long>>
        std::vector<long> data1(10000,1);
        std::vector<long> data2(10000,1);
        std::vector<long> data3(10000,1);
        subs.push_back(boost::asynchronous::parallel_reduce(std::move(data1),
                                                [](long const& a,
                                                {
                                                  return a + b;
                                                },1000));
        subs.push_back(boost::asynchronous::parallel_reduce(std::move(data2),
                                                [](long const& a,
                                                {
                                                  return a + b;
                                                },1000));
        subs.push_back(boost::asynchronous::parallel_reduce(std::move(data3),
                                                [](long const& a,
                                                {
                                                  return a + b;
                                                },1000));

        boost::asynchronous::create_callback_continuation(
                        [task_res](std::vector<boost::asynchronous::expected<l
                        {
                            long r = res[0].get() + res[1].get() + res[2].get(
                            task_res.set_value(r);
                        },
                        std::move(subs));
    }
};
```

Creating a continuation from a simple functor

For very simple tasks, it is in a post C++11 world annoying to have to write a functor class like our above sub_task. For such cases, Asynchronous provides a simple helper function:

```
auto make_lambda_continuation_wrapper(functor f, std::string const&
```
name="") where auto will be a continuation_task. We can replace our first case above by a more concise:

```
struct main_task4 : public boost::asynchronous::continuation_task<int>
{
    void operator()()
    {
        // 15, 22,5 are of type int
        boost::asynchronous::continuation_result<int> task_res = this_task_resu
        std::vector<boost::asynchronous::any_continuation_task<int>> subs;
        subs.push_back(boost::asynchronous::make_lambda_continuation_wrapper([]
        subs.push_back(boost::asynchronous::make_lambda_continuation_wrapper([]
        subs.push_back(boost::asynchronous::make_lambda_continuation_wrapper([]

        boost::asynchronous::create_callback_continuation(
                    [task_res](std::vector<boost::asynchronous::expected<in
                    {
                        int r = res[0].get() + res[1].get() + res[2].get();
                        task_res.set_value(r);
                    },
                    std::move(subs));
    }
};
```

Future-based continuations

The continuations shown above are the fastest offered by Asynchronous. Sometimes, however, we
are forced to use libraries returning us only a future. In this case, Asynchronous also offers "simple"
continuations, which are future-based. Consider the following trivial example. We consider we have
a task, called sub_task. We will simulate the future-returning library using post_future. We want
to divide our work between sub_task instances, getting a callback when all complete. We can create
a continuation using these futures:

```
// our main algo task. Needs to inherit continuation_task<value type returned b
struct main_task : public boost::asynchronous::continuation_task<long>
{
    void operator()()const
    {
        // the result of this task
        boost::asynchronous::continuation_result<long> task_res = this_task_res

        // we start calculation, then while doing this we see new tasks which c
        // when all are done, we will set the result
        // to post tasks, we need a scheduler
        boost::asynchronous::any_weak_scheduler<> weak_scheduler = boost::asyncl
        boost::asynchronous::any_shared_scheduler<> locked_scheduler = weak_sche
        if (!locked_scheduler.is_valid())
            // ok, we are shutting down, ok give up
            return;
        // simulate algo work
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
        // let's say we just found a subtask
        std::future<int> fu1 = boost::asynchronous::post_future(locked_schedule
        // simulate more algo work
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
        // let's say we just found a subtask
        std::future<int> fu2 = boost::asynchronous::post_future(locked_schedule
        // simulate algo work
        boost::this_thread::sleep(boost::posix_time::milliseconds(100));
        // let's say we just found a subtask
        std::future<int> fu3 = boost::asynchronous::post_future(locked_schedule
```

```
                    // our algo is now done, wrap all and return
            boost::asynchronous::create_continuation(
                        // called when subtasks are done, set our result
                        [task_res](std::tuple<std::future<int>,std::future<int>,std
                        {
                            try
                            {
                                long r = std::get<0>(res).get() + std::get<1>(res).
                                task_res.set_value(r);
                            }
                            catch(...)
                            {
                                task_res.set_exception(std::current_exception());
                            }
                        },
                        // future results of recursive tasks
                        std::move(fu1),std::move(fu2),std::move(fu3));
    }
    };
```

Please have a look at the complete example [examples/example_continuation_algo.cpp]

Our tasks starts by posting 3 instances of sub_task, each time getting a future. We then call **create_continuation(_job)**, passing it the futures. When all futures are ready (have a value or an exception), the callback is called, with 3 futures containing the result.

Advantage:

• can be used with any library returning a std::future

Drawbacks:

• lesser performance

• the thread calling **create_continuation(_job)** polls until all futures are set. If this thread is busy, the callback is delayed.

Important note: Like for the previous callback continuations, tasks and continuation callbacks should catch exceptions.

create_continuation(_job) has a wider interface. It can also take a vector of futures instead of a variadic version, for example:

```
// our main algo task. Needs to inherit continuation_task<value type returned by
struct main_task : public boost::asynchronous::continuation_task<long>
{
    void operator()()const
    {
        // the result of this task
        boost::asynchronous::continuation_result<long> task_res = this_task_res

        // we start calculation, then while doing this we see new tasks which ca
        // when all are done, we will set the result
        // to post tasks, we need a scheduler
        boost::asynchronous::any_weak_scheduler<> weak_scheduler = boost::asynch
        boost::asynchronous::any_shared_scheduler<> locked_scheduler = weak_sche
        if (!locked_scheduler.is_valid())
            // ok, we are shutting down, ok give up
```

```
        return;
// simulate algo work
std::vector<std::future<int> > fus;
boost::this_thread::sleep(boost::posix_time::milliseconds(100));
// let's say we just found a subtask
std::future<int> fu1 = boost::asynchronous::post_future(locked_scheduler
fus.emplace_back(std::move(fu1));
// simulate more algo work
boost::this_thread::sleep(boost::posix_time::milliseconds(100));
// let's say we just found a subtask
std::future<int> fu2 = boost::asynchronous::post_future(locked_scheduler
fus.emplace_back(std::move(fu2));
// simulate algo work
boost::this_thread::sleep(boost::posix_time::milliseconds(100));
// let's say we just found a subtask
std::future<int> fu3 = boost::asynchronous::post_future(locked_scheduler
fus.emplace_back(std::move(fu3));

// our algo is now done, wrap all and return
boost::asynchronous::create_continuation(
        // called when subtasks are done, set our result
        [task_res](std::vector<std::future<int>> res)
        {
            try
            {
                long r = res[0].get() + res[1].get() + res[2].get()
                task_res.set_value(r);
            }
            catch(...)
            {
                task_res.set_exception(std::current_exception());
            }
        },
        // future results of recursive tasks
        std::move(fus));
}
};
```

The drawback is that in this case, all futures must be of the same type. Please have a look at the complete example [examples/example_continuation_algo2.cpp]

Distributing work among machines

At the time of this writing, a core i7-3930K with 6 cores and 3.2 GHz will cost $560, so say $100 per core. Not a bad deal, so you buy it. Unfortunately, some time later you realize you need more power. Ok, there is no i7 with more cores and an Extreme Edition will be quite expensive for only a little more power so you decide to go for a Xeon. A 12-core E5-2697v2 2.7GHz will go for almost $3000 which means $250 per core, and for this you also have a lesser frequency. And if you need later even more power, well, it will become really expensive. Can Asynchronous help us use more power for cheap, and at best, with little work? It does, as you guess ;-)

Asynchronous provides a special pool, `tcp_server_scheduler`, which will behave like any other scheduler but will not execute work itself, waiting instead for clients to connect and steal some work. The client execute the work on behalf of the `tcp_server_scheduler` and sends it back the results.

For this to work, there is however a condition: jobs must be (boost) serializable to be transferred to the client. So does the returned value.

_____

Let's start with a simplest example [examples/example_tcp_server.cpp]:

```cpp
// notice how the worker pool has a different job type
struct Servant : boost::asynchronous::trackable_servant<boost::asynchronous::any
{
  Servant(boost::asynchronous::any_weak_scheduler<> scheduler)
       : boost::asynchronous::trackable_servant<boost::asynchronous::any_callab
  {
       // let's build our pool step by step. First we need a worker pool
       // possibly for us, and we want to share it with the tcp pool for its s
       boost::asynchronous::any_shared_scheduler_proxy<> workers = boost::async
                                                              boos

       // we use a tcp pool using the 3 worker threads we just built
       // our server will listen on "localhost" port 12345
       auto pool= boost::asynchronous::make_shared_scheduler_proxy<
                   boost::asynchronous::tcp_server_scheduler<
                         boost::asynchronous::lockfree_queue<boost::asynchro
                              (workers,"localhost",12345);
       // and this will be the worker pool for post_callback
       set_worker(pool);
  }
};
```

We start by creating a worker pool. The `tcp_server_scheduler` will delegate to this pool all its serialization / deserialization work. For maximum scalability we want this work to happen in more than one thread.

Note that our job type is no more a simple callable, it must be (de)serializable too (**boost::asynchronous::any_serializable**).

Then we need a `tcp_server_scheduler` listening on, in this case, localhost, port 12345. We now have a functioning worker pool and choose to use it as our worker pool so that we do not execute jobs ourselves (other configurations will be shown soon). Let's exercise our new pool. We first need a task to be executed remotely:

```cpp
struct dummy_tcp_task : public boost::asynchronous::serializable_task
{
    dummy_tcp_task(int d):boost::asynchronous::serializable_task("dummy_tcp_task
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & m_data;
    }
    int operator()()const
    {
        std::cout << "dummy_tcp_task operator(): " << m_data << std::endl;
        boost::this_thread::sleep(boost::posix_time::milliseconds(2000));
        std::cout << "dummy_tcp_task operator() finished" << std::endl;
        return m_data;
    }
    int m_data;
};
```

This is a minimum task, only sleeping. All it needs is a `serialize` member to play nice with Boost.Serialization and it must inherit `serializable_task`. Giving the task a name is essential as it will allow the client to deserialize it. Let's post to our TCP worker pool some of the tasks, wait for a client to pick them and use the results:

```cpp
// start long tasks in threadpool (first lambda) and callback in our thread
```

```
for (int i =0 ;i < 10 ; ++i)
{
    std::cout << "call post_callback with i: " << i << std::endl;
    post_callback(
            dummy_tcp_task(i),
            // the lambda calls Servant, just to show that all is safe, Servant
            [this](boost::asynchronous::expected<int> res){
                    try{
                            this->on_callback(res.get());
                    }
                    catch(std::exception& e)
                    {
                            std::cout << "got exception: " << e.what() << std::endl;
                            this->on_callback(0);
                    }
              }// callback functor.
    );
}
```

We post 10 tasks to the pool. For each task we will get, at some later undefined point (provided some clients are around), a result in form of a (ready) expected, possibly an exception if one was thrown by the task.

Notice it is safe to use `this` in the callback lambda as it will be only called if the servant still exists.

We still need a client to execute the task, this is pretty straightforward (we will extend it soon):

```
int main(int argc, char* argv[])
{
    std::string server_address = (argc>1) ? argv[1]:"localhost";
    std::string server_port = (argc>2) ? argv[2]:"12346";
    int threads = (argc>3) ? strtol(argv[3],0,0) : 4;
    cout << "Starting connecting to " << server_address << " port " << server_p

    auto scheduler = boost::asynchronous::make_shared_scheduler_proxy<boost::as
    {
        std::function<void(std::string const&,boost::asynchronous::tcp::server_
        executor=
        [](std::string const& task_name,boost::asynchronous::tcp::server_repons
            std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="dummy_tcp_task")
            {
                dummy_tcp_task t(0);
                boost::asynchronous::tcp::deserialize_and_call_task(t,resp,when_
            }
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <
                throw boost::asynchronous::tcp::transport_exception("unknown ta
            }
        };

        auto pool = boost::asynchronous::make_shared_scheduler_proxy<
                        boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::lockfree_queue<boost::asynchro
        boost::asynchronous::tcp::simple_tcp_client_proxy proxy(scheduler,pool,
                                                        0/*ms betwe
        std::future<std::future<void> > fu = proxy.run();
```

```
            std::future<void> fu_end = fu.get();
            fu_end.get();
        }
        return 0;
}
```

We start by taking as command-line arguments the server address and port and the number of threads the client will use to process stolen work from the server.

We create a single-threaded `asio_scheduler` for the communication (in our case, this is sufficient, your case might vary) to the server.

The client then defines an executor function. This function will be called when work is stolen by the client. As Asynchronous does not know what the work type is, we will need to "help" by creating an instance of the task using its name. Calling `deserialize_and_call_task` will, well, deserialize the task data into our dummy task, then call it. We also choose to return an exception is the task is not known to us.

Next, we need a pool of threads to execute the work. Usually, you will want more than one thread as we want to use all our cores.

The simplest client that Asynchronous offers is a `simple_tcp_client_proxy` proxy. We say simple, because it is only a client. Later on, we will see a more powerful tool. `simple_tcp_client_proxy` will require the asio pool for communication, the server address and port, our executor and a parameter telling it how often it should try to steal work from a server.

We are now done, the client will run until killed.

Let's sum up what we got in these few lines of code:

- a pool behaving like any other pool, which can be stolen from

- a server which does no work itself, but still scales well as serialization is using whatever threads it is given

- a trackable servant working with `post_callback`, like always

- a multithreaded client, which can be tuned precisely to use a given pool for the communication and another (or the same btw.) for work processing.

Interestingly, we have a very versatile client. It is possible to reuse the work processing and communication pools, within the same client application, for a different `simple_tcp_client_proxy` which would be connecting to another server.

The server is also quite flexible. It scales well and can handle as many clients as one wishes.

This is only the beginning of our distributed chapter.

A distributed, parallel Fibonacci

Lets's revisit our parallel Fibonacci example. We realize that with higher Fibonacci numbers, our CPU power doesn't suffice any more. We want to distribute it among several machines while our main machine still does some calculation work. To do this, we'll start with our previous example, and rewrite our Fibonacci task to make it distributable.

We remember that we first had to call `boost::asynchronous::top_level_continuation` in our post_callback to make Asynchronous aware of the later return value. The difference now is that even this one-liner lambda could be serialized and sent away, so we need to make it a `serializable_task`:

```
struct serializable_fib_task : public boost::asynchronous::serializable_task
```

```
{
    serializable_fib_task(long n,long cutoff):boost::asynchronous::serializable_
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & n_;
        ar & cutoff_;
    }
    auto operator()()const
        -> decltype(boost::asynchronous::top_level_continuation_log<long,boost:
                    (tcp_example::fib_task(long(0),long(0))))
    {
        auto cont =  boost::asynchronous::top_level_continuation_job<long,boost
                (tcp_example::fib_task(n_,cutoff_));
        return cont;
    }
    long n_;
    long cutoff_;
};
```

We need to make our task serializable and give it a name so that the client application can recognize
it. We also need a serialize member, as required by Boost.Serialization. And we need an operator() so
that the task can be executed. There is in C++11 an ugly decltype, but C++14 will solve this if your
compiler supports it. We also need a few changes in our Fibonacci task:

```
// our recursive fibonacci tasks. Needs to inherit continuation_task<value type
struct fib_task : public boost::asynchronous::continuation_task<long>
                , public boost::asynchronous::serializable_task
{
    fib_task(long n,long cutoff)
        :  boost::asynchronous::continuation_task<long>()
        , boost::asynchronous::serializable_task("serializable_sub_fib_task")
        ,n_(n),cutoff_(cutoff)
    {
    }
    template <class Archive>
    void save(Archive & ar, const unsigned int /*version*/)const
    {
        ar & n_;
        ar & cutoff_;
    }
    template <class Archive>
    void load(Archive & ar, const unsigned int /*version*/)
    {
        ar & n_;
        ar & cutoff_;
    }
    BOOST_SERIALIZATION_SPLIT_MEMBER()
    void operator()()const
    {
        // the result of this task, will be either set directly if < cutoff, ot
        boost::asynchronous::continuation_result<long> task_res = this_task_res
        if (n_<cutoff_)
        {
            // n < cutoff => execute ourselves
            task_res.set_value(serial_fib(n_));
        }
        else
```

```
                    {
                        // n>= cutoff, create 2 new tasks and when both are done, set our re
                        boost::asynchronous::create_callback_continuation_job<boost::asynchr
                                    // called when subtasks are done, set our result
                                    [task_res](std::tuple<std::future<long>,std::future<long
                                    {
                                        long r = std::get<0>(res).get() + std::get<1>(res).g
                                        task_res.set_value(r);
                                    },
                                    // recursive tasks
                                    fib_task(n_-1,cutoff_),
                                    fib_task(n_-2,cutoff_));
                    }
        }
        long n_;
        long cutoff_;
};
```

The few changes are highlighted. The task needs to be a serializable task with its own name in the constructor, and it needs serialization members. That's it, we're ready to distribute!

As we previously said, we will reuse our previous TCP example, using `serializable_fib_task` as the main posted task. This gives us this example [examples/example_tcp_server_fib.cpp].

But wait, we promised that our server would itself do some calculation work, and we use as worker pool only a `tcp_server_scheduler`. Right, let's do it now, throwing in a few more goodies. We need a worker pool, with as many threads as we are willing to offer:

```
// we need a pool where the tasks execute
auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                    boost::asynchronous::lockfree_queue<boost::asynchronous::any
```

This pool will get the fibonacci top-level task we will post, then, if our clients connect after we start, it will get the first sub-tasks.

To make it more interesting, let's offer our server to also be a job client. This way, we can build a cooperation network: the server offers fibonacci tasks, but also tries to steal some, thus increasing homogenous work distribution. We'll talk more about this in the next chapter.

```
// a client will steal jobs in this pool
auto cscheduler = boost::asynchronous::create_shared_scheduler_proxy(new boost:
// jobs we will support
std::function<void(std::string const&,boost::asynchronous::tcp::server_reponse,
                std::function<void(boost::asynchronous::tcp::client_request const&)>
        [](std::string const& task_name,boost::asynchronous::tcp::server_repons
            std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="serializable_sub_fib_task")
            {
                tcp_example::fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_callback_continua
            }
            else if (task_name=="serializable_fib_task")
            {
                tcp_example::serializable_fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_top_level_callbac
            }
            // else whatever functor we support
            else
```

```
              {
                  std::cout << "unknown task! Sorry, don't know: " << task_name <<
                  throw boost::asynchronous::tcp::transport_exception("unknown ta
              }
          };
boost::asynchronous::tcp::simple_tcp_client_proxy client_proxy(cscheduler,pool,
                                                    10/*ms between ca
```

Notice how we use our worker pool for job serialization / deserialization. Notice also how we check both possible stolen jobs.

We also introduce two new deserialization functions. boost::asynchronous::tcp::**deserialize_and_call_task** was used for normal tasks, we now have boost::asynchronous::tcp::**deserialize_and_call_top_level_callback_continuation_task** for our top-level continuation task, and boost::asynchronous::tcp::**deserialize_and_call_callback_continuation_task** for the continuation-sub-task.

We now need to build our TCP server, which we decide will get only one thread for task serialization. This ought to be enough, Fibonacci tasks have little data (2 long).

```
// we need a server
// we use a tcp pool using 1 worker
auto server_pool = boost::asynchronous::create_shared_scheduler_proxy(
                      new boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::lockfree_queue<> >(1));

auto tcp_server= boost::asynchronous::create_shared_scheduler_proxy(
                      new boost::asynchronous::tcp_server_scheduler<
                            boost::asynchronous::lockfree_queue<boost::asynchro
                            boost::asynchronous::any_callable,true>
                                (server_pool,own_server_address,(unsigned int)ow
```

We have a TCP server pool, as before, even a client to steal work ourselves, but how do we get ourselves this combined pool, which executes some work or gives some away?

Wait a minute, combined pool? Yes, a composite_threadpool_scheduler will do the trick. As we're at it, we create a servant to coordinate the work, as we now always do:

```
// we need a composite for stealing
auto composite = boost::asynchronous::create_shared_scheduler_proxy
                  (new boost::asynchronous::composite_threadpool_scheduler<boost:
                        (pool,tcp_server));

// a single-threaded world, where Servant will live.
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                            new boost::asynchronous::single_thread_schedule
                                boost::asynchronous::lockfree_queue<> >);
{
      ServantProxy proxy(scheduler,pool);
      // result of BOOST_ASYNC_FUTURE_MEMBER is a shared_future,
      // so we have a shared_future of a shared_future(result of start_async_wo
      std::future<std::future<long> > fu = proxy.calc_fibonacci(fibo_val,cutoff
      std::future<long> resfu = fu.get();
      long res = resfu.get();
}
```

Notice how we give only the worker "pool" to the servant. This means, the servant will post the top-level task to it, it will immediately be called and create 2 Fibonacci tasks, which will create each one 2 more, etc. until at some point a client connects and steals one, which will create 2 more, etc.

The client will not steal directly from this pool, it will steal from the `tcp_server` pool, which, as long as a client request comes, will steal from the worker pool, as they belong to the same composite. This will continue until the composite is destroyed, or the work is done. For the sake of the example, we do not give the composite as the Servant's worker pool but keep it alive until the end of calculation. Please have a look at the complete example [examples/example_tcp_server_fib2.cpp].

In this example, we start taking care of homogenous work distribution by packing a client and a server in the same application. But we need a bit more: our last client would steal work so fast, every 10ms that it would starve the server or other potential client applications, so we're going to tell it to only steal if the size of its work queues are under a certain amount, which we will empirically determine, according to our hardware, network speed, etc.

```
int main(int argc, char* argv[])
{
    std::string server_address = (argc>1) ? argv[1]:"localhost";
    std::string server_port = (argc>2) ? argv[2]:"12346";
    int threads = (argc>3) ? strtol(argv[3],0,0) : 4;
    // 1..n => check at regular time intervals if the queue is under the given s
    int job_getting_policy = (argc>4) ? strtol(argv[4],0,0):0;
    cout << "Starting connecting to " << server_address << " port " << server_po

    auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(
                new boost::asynchronous::asio_scheduler<>);
    {
        std::function<void(std::string const&,boost::asynchronous::tcp::server_
        executor=
        [](std::string const& task_name,boost::asynchronous::tcp::server_repons
           std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="serializable_fib_task")
            {
                tcp_example::serializable_fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_top_level_callbac
            }
            else if (task_name=="serializable_sub_fib_task")
            {
                tcp_example::fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_callback_continua
            }
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <<
                throw boost::asynchronous::tcp::transport_exception("unknown ta
            }
        };

        // guarded_deque supports queue size
        auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                        new boost::asynchronous::threadpool_scheduler<
                            boost::asynchronous::guarded_deque<boost::asynchron
        // more advanced policy
        // or simple_tcp_client_proxy<boost::asynchronous::tcp::queue_size_chec
        typename boost::asynchronous::tcp::get_correct_simple_tcp_client_proxy<
                        scheduler,pool,server_address,server_port,executor,
                        0/*ms between calls to server*/,
                        job_getting_policy /* number of jobs we try to keep in
        // run forever
        std::future<std::future<void> > fu = proxy.run();
```

```
        std::future<void> fu_end = fu.get();
        fu_end.get();
    }
    return 0;
}
```

The important new part is highlighted. `simple_tcp_client_proxy` gets an extra template argument, `queue_size_check_policy`, and a new constructor argument, the number of jobs in the queue, under which the client will try, every 10ms, to steal a job. Normally, that would be all, but g++ (up to 4.7 at least) is uncooperative and requires an extra level of indirection to get the desired client proxy. Otherwise, there is no change.

Notice that our standard lockfree queue offers no size() so we use a less efficient guarded_deque.

You will find in the complete example [examples/simple_tcp_client.cpp] a few other tasks which we will explain shortly.

Let's stop a minute to think about what we just did. We built, with little code, a complete framework for distributing tasks homogenously among machines, by reusing standard component offered by the library: threadpools, composite pools, clients, servers. If we really have client connecting or not is secondary, all what can happen is that calculating our Fibonacci number will last a little longer.

We also separate the task (Fibonacci) from the threadpool configuration, from the network configuration, and from the control of the task (Servant), leading us to highly reusable, extendable code.

In the next chapter, we will add a way to further distribute work among not only machines, but whole networks.

Example: a hierarchical network

We already distribute and parallelize work, so we can scale a great deal, but our current model is one server, many clients, which means a potentially high network load and a lesser scalability as more and more clients connect to a server. What we want is a client/server combo application where the client steals and executes jobs and a server component of the same application which steals jobs from the client on behalf of other clients. What we want is to achieve something like this:

deployment TCPHierarchical

«device»
MainJobServer

ServerApplication

TCP

TCP

«device»
ClientMachine1

ClientApplication

«d
Client

ClientServ
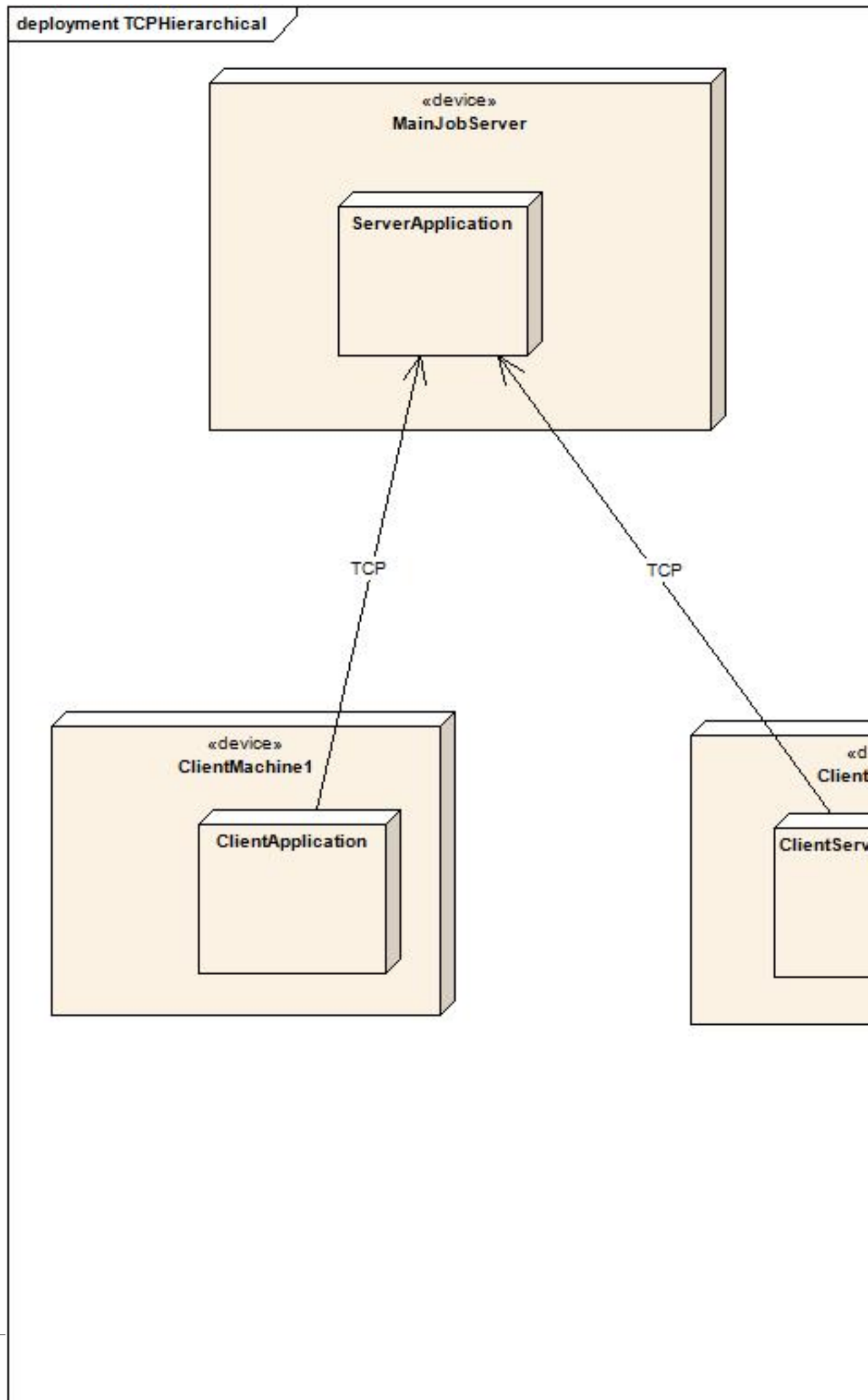
We have our server application, as seen until now, called interestingly ServerApplication on a machine called MainJobServer. This machine executes work and offers at the same time a steal-from capability. We also have a simple client called ClientApplication running on ClientMachine1, which steals jobs and executes them itself without further delegating. We have another client machine called ClientMachine2 on which ClientServerApplication runs. This applications has two parts, a client stealing jobs like ClientApplication and a server part stealing jobs from the client part upon request. For example, another simple ClientApplication running on ClientMachine2.1 connects to it and steals further jobs in case ClientMachine2 is not executing them fast enough, or if ClientMachine2 is only seen as a pass-through to move jobs execution to another network. Sounds scalable. How hard is it to build? Not so hard, because in fact, we already saw all we need to build this, so it's kind of a Lego game.

```
int main(int argc, char* argv[])
{
    std::string server_address = (argc>1) ? argv[1]:"localhost";
    std::string server_port = (argc>2) ? argv[2]:"12345";
    std::string own_server_address = (argc>3) ? argv[3]:"localhost";
    long own_server_port = (argc>4) ? strtol(argv[4],0,0):12346;
    int threads = (argc>5) ? strtol(argv[5],0,0) : 4;
    cout << "Starting connecting to " << server_address << " port " << server_p
         << " listening on " << own_server_address << " port " << own_server_po

// to be continued
```

We take as arguments the address and port of the server we are going to steal from, then our own address and port. We now need a client with its communication asio scheduler and its threadpool for job execution.

```
auto scheduler = boost::asynchronous::create_shared_scheduler_proxy(new boost::
    { //block start
        std::function<void(std::string const&,boost::asynchronous::tcp::server_
                          std::function<void(boost::asynchronous::tcp::client_
        [](std::string const& task_name,boost::asynchronous::tcp::server_repons
          std::function<void(boost::asynchronous::tcp::client_request const&)>
        {
            if (task_name=="serializable_fib_task")
            {
                tcp_example::serializable_fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_top_level_callba
            }
            else if (task_name=="serializable_sub_fib_task")
            {
                tcp_example::fib_task fib(0,0);
                boost::asynchronous::tcp::deserialize_and_call_callback_continu
            }
            // else whatever functor we support
            else
            {
                std::cout << "unknown task! Sorry, don't know: " << task_name <
                throw boost::asynchronous::tcp::transport_exception("unknown ta
            }
        };
        // create pools
        // we need a pool where the tasks execute
        auto pool = boost::asynchronous::create_shared_scheduler_proxy(
                    new boost::asynchronous::threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<boost::asynchro
        boost::asynchronous::tcp::simple_tcp_client_proxy client_proxy(schedule
                                                            10/*ms be
// to be continued
```

We now need a server to which more clients will connect, and a composite binding it to our worker pool:

```
// we need a server
// we use a tcp pool using 1 worker
auto server_pool = boost::asynchronous::create_shared_scheduler_proxy(
            new boost::asynchronous::threadpool_scheduler<
                    boost::asynchronous::lockfree_queue<> >(1));
auto tcp_server= boost::asynchronous::create_shared_scheduler_proxy(
            new boost::asynchronous::tcp_server_scheduler<
                    boost::asynchronous::lockfree_queue<boost::asynchro
                    boost::asynchronous::any_callable,true>
                        (server_pool,own_server_address,(unsigned int)ov
// we need a composite for stealing
auto composite = boost::asynchronous::create_shared_scheduler_proxy(new boos
                                                    (pool,tcp_se

std::future<std::future<void> > fu = client_proxy.run();
std::future<void> fu_end = fu.get();
fu_end.get();
} //end block

return 0;
} //end main
```

And we're done! The client part will steal jobs and execute them, while the server part, bound to the client pool, will steal on sub-client-demand. Please have a look at the complete code [examples/tcp_client_server.cpp].

Picking your archive

By default, Asynchronous uses a Boost Text archive (text_oarchive, text_iarchive), which is simple and efficient enough for our Fibonacci example, but inefficient for tasks holding more data.

Asynchronous supports any archive task, requires however a different job type for this. At the moment, we can use a `portable_binary_oarchive/portable_binary_iarchive` by selecting `any_bin_serializable` as job. If Boost supports more archive types, support is easy to add.

The previous Fibonacci server example has been rewritten [examples/example_tcp_server_fib2_bin.cpp] to use this capability. The client [examples/simple_tcp_client_bin_archive.cpp] has also been rewritten using this new job type.

Parallel Algorithms (Christophe Henry / Tobias Holl)

Asynchronous supports out of the box quite some asynchronous parallel algorithms, as well as interesting combination usages. These algorithms are callback-continuation-based. Some of these algorithms also support distributed calculations as long as the user-provided functors are (meaning they must be serializable).

What is the point of adding yet another set of parallel algorithms which can be found elsewhere? Because truly asynchronous algorithms are hard to find. By this we mean non-blocking. If one needs parallel algorithms, it's because they could need long to complete. And if they take long, we really do not want to block until it happens.

All of the algorithms are made for use in a worker threadpool. They represent the work part of a `post_callback;`

In the philosophy of Asynchronous, the programmer knows better the task size where he wants to start parallelizing, so all these algorithms take a cutoff. Work is cut into packets of this size.

All range algorithms also have a version taking a continuation as range argument. This allows to combine algorithms functional way, for example this (more to come):

```
return parallel_for(parallel_for(parallel_for(...)));
```

Asynchronous implements the following algorithms. It is also indicated whether the algorithm supports arguments passed as iterators, moved range, or continuation. Indicated is also whether the algorithm is distributable.

Table 3.1. Non-modifying Algorithms, in boost/asynchronous/algorithm

Name	Description	Header	Input Parameters	Distributable
parallel_all_of	checks if a predicate is true for all of the elements in a range	parallel_all_of.hpp	Iterators, continuation	No
parallel_any_of	checks if a predicate is true for any of the elements in a range	parallel_any_of.hpp	Iterators, continuation	No
parallel_none_of	checks if a predicate is true for none of the elements in a range	parallel_none_of.hpp	Iterators, continuation	No
parallel_for_each	aplpies a functor to a range of elements and accumulates the result into the functor	parallel_for_each.hpp	Iterators	No
parallel_for	applies a function to a range of elements	parallel_for.hpp	Iterators, moved range, continuation	Yes
parallel_count	returns the number of elements satisfying specific criteria	parallel_count.hpp	Iterators, moved range, continuation	Yes
parallel_count_if	returns the number of elements satisfying specific criteria	parallel_count_if.hpp	Iterators, moved range, continuation	No
parallel_equal	determines if two sets of elements are the same	parallel_equal.hpp	Iterators	No
parallel_mismatch	finds the first position where two ranges differ	parallel_mismatch.hpp	Iterators	No
parallel_find_all	finds all the elements satisfying specific criteria	parallel_find_all.hpp	Iterators, moved range, continuation	Yes
parallel_find_end	finds the last sequence of	parallel_find_end.hpp	Iterators, continuation	No

Name	Description	Header	Input Parameters	Distributable
	elements in a certain range			
parallel_find_first_of	searches for any one of a set of elements	parallel_find_first_of.hpp	Iterators, continuation	No
parallel_adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate)	parallel_adjacent_find.hpp	Iterators	No
parallel_lexicographical_compare	checks if one range is lexicographically less than another	parallel_lexicographical_compare.hpp	Iterators	No
parallel_search	searches for a range of elements	parallel_search.hpp	Iterators, continuation	No
parallel_search_n	searches for a number consecutive copies of an element in a range	parallel_search_n.hpp	Iterators, continuation	No
parallel_scan	does a custom scan over a range of elements	parallel_scan.hpp	Iterators, moved range, continuation	No
parallel_inclusive_scan	does an inclusive scan over a range of elements	parallel_inclusive_scan.hpp	Iterators, moved range, continuation	No
parallel_exclusive_scan	does an exclusive scan over a range of elements	parallel_exclusive_scan.hpp	Iterators, moved range, continuation	No

Table 3.2. Modifying Algorithms, in boost/asynchronous/algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_copy	copies a range of elements to a new location	parallel_copy.hpp	Iterators, moved range, continuation	No
parallel_copy_if	copies a the elements to a new location for which the given predicate is true.	parallel_copy_if.hpp	Iterators	No
parallel_move	moves a range of elements to a new location	parallel_move.hpp	Iterators, moved range, continuation	No
parallel_fill	assigns a range of elements a certain value	parallel_fill.hpp	Iterators, moved range, continuation	No
parallel_transform	applies a function to a range of elements	parallel_transform.hpp	Iterators	No

Name	Description	Header	Parameters taken	Distributable
parallel_generate	saves the result of a function in a range	parallel_generate.hpp	Iterators, moved range, continuation	No
parallel_remove_copy	copies a range of elements that are not equal to a specific value	parallel_remove_copy.hpp	Iterators	No
parallel_remove_copy_if	copies a range of elements omitting those that satisfy specific criteria	parallel_remove_copy.hpp	Iterators	No
parallel_replace	replaces all values with a specific value with another value	parallel_replace.hpp	Iterators, moved range, continuation	No
parallel_replace_if	replaces all values satisfying specific criteria with another value	parallel_replace.hpp	Iterators, moved range, continuation	No
parallel_reverse	reverses the order of elements in a range	parallel_reverse.hpp	Iterators, moved range, continuation	No
parallel_swap_ranges	swaps two ranges of elements	parallel_swap_ranges.hpp	Iterators	No
parallel_transform_inclusive_scan	does an inclusive scan over a range of elements after applying a function to each element	parallel_transform_inclusive_scan.hpp	Iterators	No
parallel_transform_exclusive_scan	does an exclusive scan over a range of elements after applying a function to each element	parallel_transform_exclusive_scan.hpp	Iterators	No

Table 3.3. Partitioning Operations, in boost/asynchronous/algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_is_partitioned	determines if the range is partitioned by the given predicate	parallel_is_partitioned.hpp	Iterators	No
parallel_partition	divides a range of elements into two groups	parallel_partition.hpp	Iterators, moved range, continuation	No
parallel_stable_partition	divides elements into two groups while preserving their relative order	parallel_stable_partition.hpp	Iterators, moved range, continuation	No
parallel_partition_copy	copies a range dividing the	parallel_partition_copy.hpp	Iterators	No

Name	Description	Header	Parameters taken	Distributable
	elements into two groups			

Table 3.4. Sorting Operations, in boost/asynchronous/algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_is_sorted	checks whether a range is sorted according to the given predicate	parallel_is_sorted.hpp	Iterators	No
parallel_is_reverse_sorted	checks whether a range is reverse sorted according to the given predicate	parallel_is_sorted.hpp	Iterators	No
parallel_sort	sorts a range according to the given predicate	parallel_sort.hpp	Iterators, moved range, continuation	Yes
parallel_sort_inplace	sorts a range according to the given predicate using inplace merge	parallel_sort_inplace.hpp	Iterators, moved range, continuation	No
parallel_spreadsort_inplace	sorts a range according to the given predicate using a Boost.Spreadsort algorithm and inplace merge	parallel_sort_inplace.hpp	Iterators, moved range, continuation	No
parallel_stable_sort_inplace	sorts a range of elements while preserving order between equal elements using inplace merge	parallel_sort_inplace.hpp	Iterators, moved range, continuation	No
parallel_spreadsort	sorts a range according to the given predicate using a Boost.Spreadsort algorithm	parallel_sort.hpp	Iterators, moved range, continuation	No
parallel_stable_sort	sorts a range of elements while preserving order between equal elements	parallel_stable_sort.hpp	Iterators, moved range, continuation	No
parallel_partial_sort	sorts the first N elements of a range	parallel_partial_sort.hpp	Iterators	No
parallel_quicksort	sorts a range according to the given predicate using a quicksort	parallel_quicksort.hpp	Iterators	No

Name	Description	Header	Parameters taken	Distributable
parallel_quick_spreadsort	sort a range according to the given predicate using a quicksort and a Boost.Spreadsort algorithm	parallel_quicksort.hpp	Iterators	No
parallel_nth_element	partially sorts the given range making sure that it is partitioned by the given element	parallel_nth_element.hpp	Iterators	No

Table 3.5. Numeric Algorithms in boost/asynchronous/algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_iota	fills a range with successive increments of the starting value	parallel_iota.hpp	Iterators, moved range	No
parallel_reduce	sums up a range of elements	parallel_reduce.hpp	Iterators, moved range, continuation	Yes
parallel_inner_product	computes the inner product of two ranges of elements	parallel_inner_product.hpp	Iterators, moved range, continuation	Yes
parallel_partial_sum	computes the partial sum of a range of elements	parallel_partial_sum.hpp	Iterators, moved range, continuation	No

Table 3.6. Algorithms Operating on Sorted Sequences in boost/asynchronous/ algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_merge	merges two sorted ranges	parallel_merge.hpp	Iterators	No

Table 3.7. Minimum/maximum operations in boost/asynchronous/algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_extremum	returns an extremum(smaller/ greater) of the given values according to a given predicate	parallel_extremum.hpp	Iterators, moved range, continuation	Yes

Table 3.8. Miscellaneous Algorithms in boost/asynchronous/algorithm

Name	Description	Header	Parameters taken	Distributable
parallel_invoke	invokes a variable number of operations in parallel	parallel_invoke.hpp	variadic sequence of functions	Yes
if_then_else	invokes algorithms based on the result of a function	if_then_else.hpp	if/then/else clauses	No

Table 3.9. (Boost) Geometry Algorithms in boost/asynchronous/algorithm/ geometry (compatible with boost geometry 1.58). Experimental and tested only with polygons.

Name	Description	Header	Parameters taken	Distributable
parallel_geometry_intersection_of	calculates the intersection of many geometries	parallel_geometry_intersection_of.hpp	iterators of a range, continuation	No
parallel_geometry_union_of_x	union of x many geometries which each other	parallel_geometry_union_of_x.hpp	iterators of a moved range, continuation	No
parallel_intersection	calculates the intersection of two geometries	parallel_intersection.hpp	two geometries	No
parallel_union	combines two geometries which each other	parallel_union.hpp	two geometries	No

Finding the best cutoff

The algorithms described above all make use of a cutoff, which is the number of elements where the algorithm should stop going parallel and execute sequentially. It is sometimes also named grain size in the literrature. Finding the best value can often make quite a big difference in execution time. Unfortunately, the best cutoff differs greatly between different processors or even machines. To make this task easier, Asynchronous provides a helper function, **find_best_cutoff**, which helps finding the best cutoff. For best results, it makes sense to use it at deployment time. **find_best_cutoff** can be found in boost/asynchronous/helpers.hpp.

```
template <class Func, class Scheduler>
std::tuple<std::size_t,std::vector<std::size_t>> find_best_cutoff(Scheduler s,
                                      std::size_t cutoff_begin,
                                      std::size_t cutoff_end,
                                      std::size_t steps,
                                      std::size_t retries,
                                      const std::string& task_na
                                      std::size_t prio=0 )
```

Return value: A tuple containing the best cutoff and a std::vector containing the elapsed times of this best cutoff.

Parameters:

• Scheduler s: the scheduler which will execute the algorithm we want to optimize

- Func f: a unary predicate which will be called for every cutoff value. It must have a signature of the form Unspecified-Continuation f (std::size_t cutoff); which will return a continuation, which is what algorithms described in the next sections will return.

- cutoff_begin, cutoff_end: the range of cutoffs to test

- steps: step between two possible cutoff values. This is needed because testing every possible cutoff would take very long.

- retries: how many times the same cutoff value will be used. Using retries will give us a better mean vakue for a given cutoff.

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Please have a look at this example finding the best cutoff for a parallel_sort [examples/example_cutoff_sort.cpp].

parallel_for

Applies a functor to every element of the range [beg,end) .

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_for(Iterator beg, Iterator end,Func func,long cutoff,const std::string
```

The parallel_for version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

The third argument is the predicate applied on each element of the algorithm.

The fourth argument is the cutoff, meaning in this case the max. number of elements of the input range in a task.

The optional fifth argument is the name of the tasks used for logging.

The optional sixth argument is the priority of the tasks in the pool.

The return value is a void continuation containing either nothing or an exception if one was thrown from one of the tasks.

Example:

```
struct Servant : boost::asynchronous::trackable_servant<>
{
    void start_async_work()
    {
        // start long tasks in threadpool (first lambda) and callback in our th
        post_callback(
                [this](){
                        return boost::asynchronous::parallel_for(this->m_data.be
                                                [](int& i)
                                                {
                                                    i += 2;
                                                },1500);
                    },// work
                // the lambda calls Servant, just to show that all is safe, Serv
                [](boost::asynchronous::expected<void> /*res*/){
                        ...
                }// callback functor.
```

```
                );
        }
        std::vector<int> m_data;
};

// same using post_future
std::future<void> fu = post_future(
                            [this](){
                             return boost::asynchronous::parallel_for(this->m_da
                                            [](int& i)
                                            {
                                              i += 2;
                            },1500);});
```

The most important parts are highlighted. Do not forget the return statement as we are returning a continuation and we do not want the lambda to be interpreted as a void lambda. The caller has responsibility of the input data, given in the form of iterators.

The code will do following:

- start tasks in the current worker pool of max 1500 elements of the input data

- add 2 to each element in parallel

- The parallel_for will return a continuation

- The callback lambda will be called when all tasks complete. The expected will be either set or contain an exception

- If post_future is used, a future<void> will be returned.

Please have a look at the complete example [examples/example_parallel_for.cpp].

The functor can either be called for every single element, or for a range of elements:

```
std::future<void> fu = post_future(
                            [this](){
                             return boost::asynchronous::parallel_for(this->m_da
                                            [](std::vector
                                            {
                                              for(;beg !=
                                              {
                                                *beg +=
                                              }
                            },1500);});
```

The second version takes a range per rvalue reference. This is signal given to Asynchronous that it must take ownership of the range. The return value is then a continuation of the given range type:

```
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_for(Range&& range,Func func,long cutoff,const std::string& task_name="
```

A post_callback / post_future will therefore get a expected<new range>, for example:

```
post_callback(
    [](){
    {
        std::vector<int> data;
        return boost::asynchronous::parallel_for(std::move(data),
                                          [](int& i)
```

```
                                                              {
                                                                  i += 2;
                                                              },1500);
        },
        ](boost::asynchronous::expected<std::vector<int>> ){}
);
```

In this case, the programmer does not need to ensure the container stays valid, Asynchronous takes care of it.

The third version of this algorithm takes a range continuation instead of a range as argument and will be invoked after the continuation is ready.

```
// version taking a continuation of a range as first argument
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<typename Range::return_type,
parallel_for(Range range,Func func,long cutoff,const std::string& task_name="",
```

This version allows chaining parallel calls. For example, it is now possible to write:

```
post_callback(
    [](){
    {
        std::vector<int> data;
        return parallel_for(parallel_for(parallel_for(
                                                            // executed first
                                                            std::move(data),
                                                            [](int& i)
                                                            {
                                                                i += 2;
                                                            },1500),
                                              // executed second
                                              [](int& i)
                                              {
                                                  i += 2;
                                              },1500),
                                  // executed third
                                  [](int& i)
                                  {
                                      i += 2;
                                  },1500);
    },
    ](boost::asynchronous::expected<std::vector<int>> ){} // callback
);
```

This code will be executed as follows:

- the most inner parallel_for (parallel execution)

- A kind of synchronization point will be done at this point until the parallel_for completes

- the middle parallel_for will be executed (parallel execution)

- A kind of synchronization point will be done at this point until the parallel_for completes

- the outer parallel_for will be executed (parallel execution)

- A kind of synchronization point will be done at this point until the parallel_for completes

- The callback will be called

With "kind of synchronization point", we mean there will be no blocking synchronization, it will just be waited until completion.

Finally, this algorithm has a distributed version. We need, as with our Fibonacci example, a serializable sub-task which will be created as often as required by our cutoff and which will handle a part of our range:

```
struct dummy_parallel_for_subtask : public boost::asynchronous::serializable_ta
{
    dummy_parallel_for_subtask(int d=0):boost::asynchronous::serializable_task(
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & m_data;
    }
    void operator()(int& i)const
    {
        i += m_data;
    }
    // some data, so we have something to serialize
    int m_data;
};
```

We also need a serializable top-level task, creating sub-tasks:

```
struct dummy_parallel_for_task : public boost::asynchronous::serializable_task
{
    dummy_parallel_for_task():boost::asynchronous::serializable_task("dummy_par
    template <class Archive>
    void serialize(Archive & ar, const unsigned int /*version*/)
    {
        ar & m_data;
    }
    auto operator()() -> decltype(boost::asynchronous::parallel_for<std::vector
                                      std::move(std::vector<int>()),
                                      dummy_parallel_for_subtask(2),
                                      10))
    {
        return boost::asynchronous::parallel_for
                <std::vector<int>,dummy_parallel_for_subtask,boost::asynchronou
            std::move(m_data),
            dummy_parallel_for_subtask(2),
            10);
    }
    std::vector<int> m_data;
};
```

We now post our top-level task inside a servant or use post_future:

```
post_callback(
                dummy_parallel_for_task(),
                // the lambda calls Servant, just to show that all is safe, Serv
                [this](boost::asynchronous::expected<std::vector<int>> res){
                  try
                  {
                      // do something
                  }
                  catch(std::exception& e)
                  {
```

```
                        std::cout << "got exception: " << e.what() << std::endl;
                   }
                }// end of callback functor.
    );
```

Please have a look at the complete server example [examples/example_parallel_for_tcp.cpp].

parallel_for_each

Applies a functor to every element of the range [beg,end). This functor can save data. It is merged at different steps with other instances of this functor. The algorithm returns the last merged instance.

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Func,Job>
parallel_all_of(Iterator begin, Iterator end,Func func,long cutoff,const std::s
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: A merged instance of a functor of type Func.

Parameters:

• begin, end: the range of elements to search

• func: a class / struct object with a:

  • void operator()(const Type& a)

  • void merge (Func const& f)

• cutoff: the maximum size of a sequential chunk

• task_name: the name displayed in the scheduler diagnostics

• prio: task priority

parallel_all_of

Checks if unary predicate p returns true for all elements in the range [begin, end).

```
// version for iterators
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_all_of(Iterator begin, Iterator end,Func func,long cutoff,const std::s

// version for ranges returned as continuations
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_all_of(Range range,Func func,long cutoff,const std::string& task_name=
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: true if unary predicate returns true for all elements in the range, false otherwise. Returns true if the range is empty.

Parameters:

• begin, end: the range of elements to search

Or a continuation, coming from another algorithm.

- func: unary predicate function. The signature of the function should be equivalent to the following:
  bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_any_of

Checks if unary predicate p returns true for at least one element in the range [begin, end).

```
// version for iterators
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_any_of(Iterator begin, Iterator end,Func func,long cutoff,const std::s

// version for ranges returned as continuations
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_any_of(Range range,Func func,long cutoff,const std::string& task_name=
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: true if unary predicate returns true for at least one element in the range, false otherwise. Returns false if the range is empty.

Parameters:

- begin, end: the range of elements to search

  Or a continuation, coming from another algorithm.

- func: unary predicate function. The signature of the function should be equivalent to the following:
  bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_none_of

Checks if unary predicate p returns true for no elements in the range [begin, end).

```
// version for iterators
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_none_of(Iterator begin, Iterator end,Func func,long cutoff,const std::

// version for ranges returned as continuations
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_none_of(Range range,Func func,long cutoff,const std::string& task_name=
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Return value</u>: true if unary predicate returns true for no elements in the range, false otherwise. Returns true if the range is empty.

<u>Parameters</u>:

- begin, end: the range of elements to search

  Or a continuation, coming from another algorithm.

- func: unary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_equal

Checks if two ranges are equal.

```
template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_equal(Iterator1 begin1, Iterator1 end1,Iterator2 begin2, long cutoff,co
```

```
template <class Iterator1,class Iterator2, class Func, class Job=BOOST_ASYNCHRON
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_equal(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Func func, long
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Return value</u>: If the length of the range [first1, end1) does not equal the length of the range beginning at begin2, returns false If the elements in the two ranges are equal, returns true. Otherwise returns false.

<u>Parameters</u>:

- begin1, end1: the first range

- begin2: the beginning of the second range

- func: binary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a,const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_mismatch

Returns the first mismatching pair of elements from two ranges: one defined by [begin1, end1) and another starting at begin2.

```
template <class Iterator1,class Iterator2, class Func, class Job=BOOST_ASYNCHRON
boost::asynchronous::detail::callback_continuation<std::pair<Iterator1,Iterator
```

```
parallel_mismatch(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Func func,

template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<std::pair<Iterator1,Iterator
parallel_mismatch(Iterator1 begin1, Iterator1 end1,Iterator2 begin2, long cutof
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: std::pair with iterators to the first two non-equivalent elements. If no mismatches are found when the comparison reaches end1 or end2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range.

Parameters:

• begin1, end1: the first range

• begin2: the beginning of the second range

• func: binary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a,const Type &b);

• cutoff: the maximum size of a sequential chunk

• task_name: the name displayed in the scheduler diagnostics

• prio: task priority

parallel_find_end

Searches for the last subsequence of elements [begin2, end2) in the range [begin1, end1). [begin2, end2) can be replaced (3rd form) by a continuation.

```
template <class Iterator1,class Iterator2, class Func, class Job=BOOST_ASYNCHRON
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_find_end(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Iterator2 en

template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_find_end(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Iterator2 en

template <class Iterator1,class Range,class Func, class Job=BOOST_ASYNCHRONOUS_
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_find_end(Iterator1 begin1, Iterator1 end1,Range range, long cutoff,con
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: Iterator to the beginning of last subsequence [begin2, end2) in range [begin1, end1). If [begin2, end2) is empty or if no such subsequence is found, end1 is returned.

Parameters:

• begin1, end1: the first range

• begin2, end2 / Range: the subsequence to look for.

• func: binary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a,const Type &b);

• cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_find_first_of

Searches the range [begin1, end1) for any of the elements in the range [begin2, end2). [begin2, end2) can be replaced (3rd form) by a continuation.

```
template <class Iterator1,class Iterator2, class Func, class Job=BOOST_ASYNCHRO
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_find_first_of(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Iterat

template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT_
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_find_first_of(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Iterat

template <class Iterator1,class Range,class Func, class Job=BOOST_ASYNCHRONOUS_
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_find_first_of(Iterator1 begin1, Iterator1 end1,Range range, long cutof
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: Iterator to the first element in the range [begin1, end1) that is equal to an element from the range [begin2; end2). If no such element is found, end1 is returned.

Parameters:

- begin1, end1: the first range

- begin2, end2 / Range: the subsequence to look for.

- func: binary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a,const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_adjacent_find

Searches the range [begin, end) for two consecutive identical elements.

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Iterator,Job>
parallel_adjacent_find(Iterator begin, Iterator end,Func func, long cutoff,cons

template <class Iterator, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Iterator,Job>
parallel_adjacent_find(Iterator begin, Iterator end, long cutoff,const std::str
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: an iterator to the first of the first pair of identical elements, that is, the first iterator it such that *it == *(it+1) for the second version or func(*it, *(it + 1)) != false for the first version. If no such elements are found, last is returned.

Parameters:

- begin, end: the range of elements to examine

- func: binary predicate function. The signature of the function should be equivalent to the following:
  bool pred(const Type &a,const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_lexicographical_compare

Checks if the first range [begin1, end1) is lexicographically less than the second range [begin2, end2).

```
template <class Iterator1,class Iterator2, class Func, class Job=BOOST_ASYNCHRO
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_lexicographical_compare(Iterator1 begin1, Iterator1 end1,Iterator2 beg

template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT_
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_lexicographical_compare(Iterator1 begin1, Iterator1 end1,Iterator2 beg
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: true if the first range is lexicographically less than the second.

Parameters:

- begin1, end1: the first range of elements to examine

- begin2, end2: the second range of elements to examine.

- func: binary predicate function. The signature of the function should be equivalent to the following:
  bool pred(const Type &a,const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_search

Searches for the first occurrence of the subsequence of elements [begin2, end2)in the range [begin1, end1 - (end2 - end1)).

```
template <class Iterator1,class Iterator2, class Func, class Job=BOOST_ASYNCHRO
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_search(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Iterator2 end

template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT_
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_search(Iterator1 begin1, Iterator1 end1,Iterator2 begin2,Iterator2 end

template <class Iterator1,class Range,class Func, class Job=BOOST_ASYNCHRONOUS_
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_search(Iterator1 begin1, Iterator1 end1,Range range, long cutoff,const
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: Iterator to the beginning of first subsequence [begin2, end2) in the range [begin1, end1 - (end2 - begin2)). If no such subsequence is found, end1 is returned. If [begin2, end2) is empty, begin1 is returned.

Parameters:

- begin1, end1: the first range

- begin2, end2 / Range: the subsequence to look for.

- func: binary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a,const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_search_n

Searches the range [begin, end) for the first sequence of count identical elements, each equal to the given value value.

```
template <class Iterator1, class Size, class T, class Func, class Job=BOOST_ASYN
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_search_n(Iterator1 begin1, Iterator1 end1, Size count, const T& value,

template <class Iterator1, class Size, class T, class Job=BOOST_ASYNCHRONOUS_DE
boost::asynchronous::detail::callback_continuation<Iterator1,Job>
parallel_search_n(Iterator1 begin1, Iterator1 end1, Size count, const T& value,
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: Iterator to the beginning of the found sequence in the range [begin1, end1). If no such sequence is found, end1 is returned.

Parameters:

- begin1, end1: the first range

- count: the length of the sequence to search for

- value: the value of the elements to search for

- func: binary predicate which returns true if the elements should be treated as equal. The signature of the function should be equivalent to the following: bool pred(const Type &a,const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_scan

Computes all partial reductions of a collection. For every output position, a reduction of the input up to that point is computed. This is the basic,

most flexible underlying implementation of parallel_exclusive_scan, parallel_inclusive_scan, parallel_transform_exclusive_scan, parallel_transform_inclusive_scan.

The operator, represented by the Combine function, must be associative.

The algorithm works by doing two passes on the sequence: the first pass uses the Reduce function, the second pass uses the result of Reduce in Combine. Scan will output the result.

```
// version for iterators
template <class Iterator, class OutIterator, class T, class Reduce, class Combi
boost::asynchronous::detail::callback_continuation<OutIterator,Job>
parallel_scan(Iterator beg, Iterator end, OutIterator out, T init, Reduce r, Co

// version for moved ranges
template <class Range, class OutRange, class T, class Reduce, class Combine, cla
boost::asynchronous::detail::callback_continuation<std::pair<Range,OutRange>,Jo
parallel_scan(Range&& range,OutRange&& out_range,T init,Reduce r, Combine c, Sc

// version for a single moved range (in/out)
template <class Range, class T, class Reduce, class Combine, class Scan, class
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_scan(Range&& range,T init,Reduce r, Combine c, Scan s, long cutoff,con

// version for continuation
template <class Range, class T, class Reduce, class Combine, class Scan, class
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_scan(Range range,T init,Reduce r, Combine c, Scan s, long cutoff,const
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to scan.

- out: the beginning of the output range.

- out_range: the output range.

- init: initial value, combined with the first scanned element.

- Reduce: binary predicate which returns an accumulated value. The signature of the function should be equivalent to the following: Ret reduce(Iterator,Iterator); The type Ret must be such that an object of type Iterator can be dereferenced and assigned a value of type Ret.

- Combine: binary predicate which combines two elements, like std::plus would do. The signature of the function should be equivalent to the following: Ret combine(const Type&,const Type&); The type Ret must be such that an object of type Iterator can be dereferenced and assigned a value of type Ret.

- Scan: assigns the result to the out range / iterator. The signature of the function should be equivalent to the following: void scan(Iterator beg, Iterator end, OutIterator out, T init).

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- Last version returns a Range of the type returned by continuation

parallel_inclusive_scan

Computes all partial reductions of a collection. For every output position, a reduction of the input up to that point is computed. The nth output value is a reduction over the first n input values.

```
// version for iterators
template <class Iterator, class OutIterator, class T, class Func, class Job=BOO
boost::asynchronous::detail::callback_continuation<OutIterator,Job>
parallel_inclusive_scan(Iterator beg, Iterator end, OutIterator out, T init, Fu

// version for moved ranges
template <class Range, class OutRange, class T, class Func, class Job=BOOST_ASY
boost::asynchronous::detail::callback_continuation<std::pair<Range,OutRange>,Jol
parallel_inclusive_scan(Range&& range,OutRange&& out_range, T init, Func f, long

// version for a single moved range (in/out)
template <class Range, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_inclusive_scan(Range&& range,T init,Func f, long cutoff,const std::str

// version for continuation
template <class Range, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_inclusive_scan(Range range,T init,Func f, long cutoff,const std::string
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to scan.

- out: the beginning of the output range.

- out_range: the output range.

- init: initial value, combined with the first scanned element.

- Func: binary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a, const Type &b); The type Ret must be such that an object of type OutIterator can be dereferenced and assigned a value of type Ret

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- Last version returns a Range of the type returned by continuation

parallel_exclusive_scan

Computes all partial reductions of a collection. For every output position, a reduction of the input up to that point is computed. The nth output value is a reduction over the first n- input values.

```
// version for iterators
template <class Iterator, class OutIterator, class T, class Func, class Job=BOO
boost::asynchronous::detail::callback_continuation<OutIterator,Job>
parallel_exclusive_scan(Iterator beg, Iterator end, OutIterator out, T init, Fu
```

```
// version for moved ranges
template <class Range, class OutRange, class T, class Func, class Job=BOOST_ASYI
boost::asynchronous::detail::callback_continuation<std::pair<Range,OutRange>,Jol
parallel_exclusive_scan(Range&& range,OutRange&& out_range, T init, Func f, long
```

```
// version for a single moved range (in/out)
template <class Range, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT'
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_exclusive_scan(Range&& range,T init,Func f, long cutoff,const std::str.
```

```
// version for continuation
template <class Range, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT'
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_exclusive_scan(Range range,T init,Func f, long cutoff,const std::string
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to scan.

- out: the beginning of the output range.

- out_range: the output range.

- init: initial value, combined with the first scanned element.

- Func: binary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a, const Type &b); The type Ret must be such that an object of type OutIterator can be dereferenced and assigned a value of type Ret

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- Last version returns a Range of the type returned by continuation

parallel_copy

Copies the elements in the range, defined by [begin, end), to another range beginning at result. The order of the elements that are not removed is preserved.

```
// version for iterators
template <class Iterator,class ResultIterator, class Job=BOOST_ASYNCHRONOUS_DEF,
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_copy(Iterator begin, Iterator end,ResultIterator result, long cutoff,c
```

```
// version for moved range
template <class Range,class ResultIterator, class Job=BOOST_ASYNCHRONOUS_DEFAULT'
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_copy(Range&& range, ResultIterator result, long cutoff,const std::strii
```

```
// version for continuation
template <class Range,class ResultIterator, class Job=BOOST_ASYNCHRONOUS_DEFAULT'
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_copy(Range range,ResultIterator out, long cutoff,const std::string& ta:
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to copy

- result: the beginning of the destination range.

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_copy_if

Copies copies the elements for which a predicate returns true. The order of the elements that are not removed is preserved.

```
template <class Iterator,class ResultIterator, class Func, class Job=BOOST_ASYNC
boost::asynchronous::detail::callback_continuation<ResultIterator,Job>
parallel_copy_if(Iterator begin, Iterator end,ResultIterator result,Func func,
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end: the range of elements to copy

- result: the beginning of the destination range.

- func: unary predicate which returns true for the required elements. The signature of the function should be equivalent to the following: bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_move

Moves the elements in the range [begin, end), to another range beginning at d_first. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

```
// version for iterators
template <class Iterator,class ResultIterator, class Job=BOOST_ASYNCHRONOUS_DEF
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_move(Iterator begin, Iterator end,ResultIterator result, long cutoff,c

// version for moved range
template <class Range,class ResultIterator, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_move(Range&& range, ResultIterator result, long cutoff,const std::stri

// version for continuation
template <class Range,class ResultIterator, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
```

```
parallel_move(Range range,ResultIterator out, long cutoff,const std::string& ta
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Parameters:</u>

- begin, end / range: the range of elements to move

- result: the beginning of the destination range.

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- 3rd version returns a Range of the type returned by continuation

parallel_fill

Assigns the given value to the elements in the range [begin, end).

```
// version for iterators
template <class Iterator, class Value, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_fill(Iterator begin, Iterator end,const Value& value, long cutoff,cons
```

```
// version for moved range
template <class Range, class Value, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_fill(Range&& range, const Value& value, long cutoff,const std::string&
```

```
// version for continuation
template <class Range, class Value, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_fill(Range range,const Value& value, long cutoff,const std::string& ta
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Parameters:</u>

- begin, end / range: the range of elements to fill

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- 3rd version returns a Range of the type returned by continuation

parallel_transform

Applies a given function to one or a variadic number of ranges and stores the result in another range.

```
// version for iterators, one range tranformed to another
template <class Iterator1, class ResultIterator, class Func, class Job=BOOST_AS
boost::asynchronous::detail::callback_continuation<ResultIterator,Job>
parallel_transform(Iterator1 begin1, Iterator1 end1, ResultIterator result, Fun
```

```
// version for iterators, two ranges tranformed to another
template <class Iterator1, class Iterator2, class ResultIterator, class Func, c
boost::asynchronous::detail::callback_continuation<ResultIterator,Job>
parallel_transform(Iterator1 begin1, Iterator1 end1, Iterator2 begin2, ResultIt

// version for any number of iterators (not with ICC)
template <class ResultIterator, class Func, class Job, class Iterator, class...
boost::asynchronous::detail::callback_continuation<ResultIterator,Job>
parallel_transform(ResultIterator result, Func func, Iterator begin, Iterator e
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Return value</u>: Result iterator to the element past the last element transformed.

<u>Parameters</u>:

- begin1, end1, begin2, end2, iterators: the range of input elements

- result: the beginning of the destination range.

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- 3rd version takes a variadic number of input ranges

- func in first version : unary predicate which returns a new value. The signature of the function should be equivalent to the following: Ret pred(const Type &a); The type Ret must be such that an object of type ResultIterator can be dereferenced and assigned a value of type Ret

- func in second version : binary predicate which returns a new value. The signature of the function should be equivalent to the following: Ret pred(const Type1 &a,const Type2 &b); The type Ret must be such that an object of type ResultIterator can be dereferenced and assigned a value of type Ret

- func in third version : n-ary predicate which returns a new value. The signature of the function should be equivalent to the following: Ret pred(const Type1 &a,const Type2 &b...,const TypeN &n); The type Ret must be such that an object of type ResultIterator can be dereferenced and assigned a value of type Ret

parallel_generate

Assigns each element in range [begin, end) a value generated by the given function object func.

```
// version for iterators
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_generate(Iterator begin, Iterator end,Func func, long cutoff,const std

// version for moved range
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_generate(Range&& range, Func func, long cutoff,const std::string& task_

// version for continuation
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
```

```
parallel_generate(Range range,Func func, long cutoff,const std::string& task_na
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to assign

- func: unary predicate which returns a new value. The signature of the function should be equivalent to the following: Ret pred(); The type Ret must be such that an object of type Iterator can be dereferenced and assigned a value of type Ret.

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- 3rd version returns a Range of the type returned by continuation

parallel_remove_copy / parallel_remove_copy_if

Copies elements from the range [begin, end), to another range beginning at out, omitting the elements which satisfy specific criteria. The first version ignores the elements that are equal to value, the second version ignores the elements for which predicate func returns true. Source and destination ranges cannot overlap.

```
template <class Iterator,class Iterator2, class Value, class Job=BOOST_ASYNCHRON
boost::asynchronous::detail::callback_continuation<Iterator2,Job>
parallel_remove_copy(Iterator begin, Iterator end,Iterator2 out,Value const& va

template <class Iterator,class Iterator2, class Func, class Job=BOOST_ASYNCHRON
boost::asynchronous::detail::callback_continuation<Iterator2,Job>
parallel_remove_copy_if(Iterator begin, Iterator end,Iterator2 out,Func func, l
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: Iterator to the element past the last element copied.

Parameters:

- begin, end: the range of elements to copy

- out: the beginning of the destination range.

- func: unary predicate which returns true for the required elements. The signature of the function should be equivalent to the following: bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_replace / parallel_replace_if

Replaces all elements satisfying specific criteria with new_value in the range [begin, end). The replace version replaces the elements that are equal to old_value, the replace_if version replaces elements for which predicate func returns true.

```
// version for iterators
template <class Iterator, class T, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_replace(Iterator begin, Iterator end, const T& new_value, const T& new_
template <class Iterator, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEF/
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_replace_if(Iterator begin, Iterator end,Func func, const T& new_value,

// version for moved range
template <class Range, class T, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_replace(Range&& range, const T& old_value, const T& new_value, long cut
template <class Range, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_replace_if(Range&& range, Func func, const T& new_value, long cutoff,co

// version for continuation
template <class Range, class T, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_replace(Range range, const T& old_value, const T& new_value, long cuto
template <class Range, class T, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_replace_if(Range range, Func func, const T& new_value, long cutoff,con
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to modify

- func: unary predicate which returns true if the element value should be replaced. The signature of the predicate function should be equivalent to the following: bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- continuation version returns a Range of the type returned by continuation

parallel_reverse

Reverses the order of the elements in the range [begin, end).

```
// version for iterators
template <class Iterator, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_reverse(Iterator begin, Iterator end, long cutoff,const std::string& ta

// version for moved range
template <class Range, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_reverse(Range&& range, long cutoff,const std::string& task_name="", st

// version for continuation
template <class Range, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
```

```
parallel_reverse(Range range, long cutoff,const std::string& task_name="", std:
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end: the range of elements to reverse

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- continuation version returns a Range of the type returned by continuation

parallel_swap_ranges

Exchanges elements between range [begin1, end1) and another range starting at begin2..

```
template <class Iterator1,class Iterator2, class Job=BOOST_ASYNCHRONOUS_DEFAULT_
boost::asynchronous::detail::callback_continuation<Iterator2,Job>
parallel_swap_ranges(Iterator1 begin1, Iterator1 end1,Iterator2 begin2, long cu
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: true if the range [begin, end) is empty or is partitioned by p. false otherwise.

Parameters:

- begin, end: the first range of elements to swap

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_transform_inclusive_scan

Computes all partial reductions of a collection. For every output position, a reduction of the input up to that point is computed. The nth output value is a reduction over the first n input values. This algorithm applies a transformation to the input element before scan. Fusing transform and scan avoids having to process two passes on the element and saves time.

```
// version for iterators
template <class Iterator, class OutIterator, class T, class Func, class Transfo
boost::asynchronous::detail::callback_continuation<OutIterator,Job>
parallel_transform_inclusive_scan(Iterator beg, Iterator end, OutIterator out, '
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to scan.

- out: the beginning of the output range.

- out_range: the output range.

- init: initial value, combined with the first scanned element.

- Func: binary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a, const Type &b); The type Ret must be such that an object of type OutIterator can be dereferenced and assigned a value of type Ret

- Transform: unary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a); The type Ret must be such that an object of type OutIterator can be dereferenced and assigned a value of type Ret

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- Last version returns a Range of the type returned by continuation

parallel_transform_exclusive_scan

Computes all partial reductions of a collection. For every output position, a reduction of the input up to that point is computed. The nth output value is a reduction over the first n-1 input values. This algorithm applies a transformation to the input element before scan. Fusing transform and scan avoids having to process two passes on the element and saves time.

```
// version for iterators
template <class Iterator, class OutIterator, class T, class Func, class Transfo
boost::asynchronous::detail::callback_continuation<OutIterator,Job>
parallel_transform_exclusive_scan(Iterator beg, Iterator end, OutIterator out, '
```

The algorithm requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end / range: the range of elements to scan.

- out: the beginning of the output range.

- out_range: the output range.

- init: initial value, combined with the first scanned element.

- Func: binary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a, const Type &b); The type Ret must be such that an object of type OutIterator can be dereferenced and assigned a value of type Ret

- Transform: unary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a); The type Ret must be such that an object of type OutIterator can be dereferenced and assigned a value of type Ret

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- Last version returns a Range of the type returned by continuation

parallel_is_partitioned

Returns true if all elements in the range [begin, end) that satisfy the predicate func appear before all elements that don't. Also returns true if [begin, end) is empty.

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_is_partitioned(Iterator begin, Iterator end,Func func,long cutoff,cons
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Return value</u>: true if unary predicate returns true for all elements in the range, false otherwise. Returns true if the range is empty.

<u>Parameters</u>:

- begin, end: the range of elements to search

  Or a continuation, coming from another algorithm.

- func: unary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_partition

Reorders the elements in the range [begin, end) in such a way that all elements for which the predicate func returns true precede the elements for which predicate func returns false

```
// version with iterators
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Iterator,Job>
parallel_partition(Iterator begin, Iterator end,Func func,const uint32_t thread_

// version with moved range
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<std::pair<Range,decltype(rang
parallel_partition(Range&& range,Func func,const uint32_t thread_num = boost::th

// version with continuation
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<std::pair<Unspecified-Range,U
parallel_partition(Range&& range,Func func,const uint32_t thread_num = boost::th
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

<u>Return value</u>: an iterator to the first element of the second group for the first version. A pair of a range and an iterator to the first element of the second group of this range for the second and third.

<u>Parameters</u>:

- begin, end: the range of elements to partition

Or a continuation, coming from another algorithm.

- func: unary predicate function. The signature of the function should be equivalent to the following: bool pred(const Type &a);

- thread_num: this algorithm being not a divide and conquer, it requires the number of threads which will be used. By default boost::thread::hardware_concurrency() is used.

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_stable_partition

Reorders the elements in the range [begin, end) in such a way that all elements for which the predicate func returns true precede the elements for which predicate func returns false. Relative order of the elements is preserved.

```
template <class Iterator, class Iterator2, class Func,class Job=BOOST_ASYNCHRON
boost::asynchronous::detail::callback_continuation<Iterator2,Job>
parallel_stable_partition(Iterator begin, Iterator end, Iterator2 out, Func fun

// version taking ownership of the container to be sorted
template <class Range, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<std::pair<Range,decltype(ra
parallel_stable_partition(Range&& range,Func func,long cutoff,const std::string

// version taking a continuation of a range as first argument
template <class Range, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<std::pair<Unspecified Range,
parallel_stable_partition(Range range,Func func,long cutoff,const std::string& 
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Returns:

- an iterator to the first element of the second group (first version)

- The partitioned range and an iterator to the first element of the second group (second version)

- The partitioned range returned from the continuation and an iterator to the first element of the second group (second version)

Parameters:

- begin, end: the range of elements to reorder.

  Or range: a moved range. Returns the sorted moved range.

  Or a continuation, coming from another algorithm. Returns the sorted range.

- func: unary predicate which returns true if the element should be ordered before other elements. The signature of the predicate function should be equivalent to the following: bool func(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

• prio: task priority

parallel_partition_copy

Copies the elements from the range [begin, end) to two output ranges in such a way that all elements for which the predicate func returns true are in the first range and all the others are in the second range. Relative order of the elements is preserved.

```
template <class Iterator, class OutputIt1, class OutputIt2, class Func,class Jol
boost::asynchronous::detail::callback_continuation<std::pair<OutputIt1, OutputIt
parallel_partition_copy(Iterator begin, Iterator end, OutputIt1 out_true, Output
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Returns: a pair of iterators to the first element of the first and second group

Parameters:

• begin, end: the range of elements to reorder.

• out_true: the beginning of the range for which func returns true.

• out_false: the beginning of the range for which func returns false.

• func: unary predicate which returns true if the element should be in the first range. The signature of the predicate function should be equivalent to the following: bool func(const Type &a);

• cutoff: the maximum size of a sequential chunk

• task_name: the name displayed in the scheduler diagnostics

• prio: task priority

parallel_is_sorted

Checks if the elements in range [first, last) are sorted in ascending order. It uses the given comparison function func to compare the elements.

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_is_sorted(Iterator begin, Iterator end,Func func,long cutoff,const std
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: true if the elements in the range are sorted in ascending order.

Parameters:

• begin, end: the range of elements to search

  Or a continuation, coming from another algorithm.

• func: binary predicate function. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

• cutoff: the maximum size of a sequential chunk

• task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_is_reverse_sorted

Checks if the elements in range [first, last) are sorted in descending order. It uses the given comparison function func to compare the elements.

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<bool,Job>
parallel_is_reverse_sorted(Iterator begin, Iterator end,Func func,long cutoff,c
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Return value: true if the elements in the range are sorted in descending order.

Parameters:

- begin, end: the range of elements to search

  Or a continuation, coming from another algorithm.

- func: binary predicate function. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_iota

Fills the range [begin, end) with sequentially increasing values, starting with value and repetitively evaluating ++value

```
// version for iterators
template <class Iterator, class T, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_iota(Iterator beg, Iterator end,T const& value,long cutoff,const std::s

// version taking a moved range
template <class Range, class T, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_iota(Range&& range,T const& value,long cutoff,const std::string& task_
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Returns: a void continuation for the first version, a continuation containing the new range for the second one.

Parameters:

- beg, end: the range of elements

  Or range: a moved range.

- T value: initial value to store, the expression ++value must be well-formed

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_reduce

Description: Sums up elements of a range using func.

```
template <class Iterator, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<decltype(func(...)),Job>
parallel_reduce(Iterator beg, Iterator end,Func func,long cutoff,const std::str

template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<decltype(func(...)),Job>
parallel_reduce(Range&& range,Func func,long cutoff,const std::string& task_name

// version taking a continuation of a range as first argument
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<decltype(func(...)),Job>
parallel_reduce(Range range,Func func,long cutoff,const std::string& task_name=
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- beg, end: the range of elements to sum

  Or range: a moved range.

  Or a continuation, coming from another algorithm.

- func: binary operation function object that will be applied. The signature of the function should be equivalent to the following: Ret fun(const Type &a, const Type &b);

  Alternatively, func might have a signature with a range: Ret fun(Iterator a, Iterator b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Returns: the return value type of calling func.

Example:

```
std::vector<int> data;
post_callback(
    [this]()
    {
        return boost::asynchronous::parallel_reduce(this->data.begin(),this->data
                                    [](int const& a, int const&
                                    {
                                        return a + b; // retur
                                    },
                                    1500);
    },
    ](boost::asynchronous::expected<int> ){} // callback gets an int
);
```

We also have a distributed version [examples/example_parallel_reduce_tcp.cpp] as an example, which strictly looks like the parallel_for version.

parallel_inner_product

Description: Computes inner product (i.e. sum of products) of the range [begin1, end1) and another range beginning at begin2.It uses op and red for these tasks respectively.

```
template <class Iterator1, class Iterator2, class BinaryOperation, class Reduce
boost::asynchronous::detail::callback_continuation<T,Job>
parallel_inner_product(Iterator1 begin1, Iterator1 end1, Iterator2 begin2, Binar

template <class Range1, class Range2, class BinaryOperation, class Reduce, class
boost::asynchronous::detail::callback_continuation<T,Job>
parallel_inner_product(Range1 && range1, Range2 && range2, BinaryOperation op, I

// version taking continuations of ranges as first and second argument
template <class Continuation1, class Continuation2, class BinaryOperation, class
boost::asynchronous::detail::callback_continuation<T,Job>
parallel_inner_product(Continuation1 && cont1, Continuation2 && cont2, BinaryOpe
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

• begin1, end1: the first range of elements

  Or range1: a moved range.

  Or a cont1, coming from another algorithm.

• begin2, end2: the second range of elements

  Or range2: a moved range.

  Or a cont2, coming from another algorithm.

• op: binary operation function object that will be applied. This "sum" function takes a value returned by op2 and the current value of the accumulator and produces a new value to be stored in the accumulator.

• red: binary operation function object that will be applied. This "product" function takes one value from each range and produces a new value. The signature of the function should be equivalent to the following: Ret fun(const Type1 &a, const Type2 &b);

• value: initial value of the sum of the products

• cutoff: the maximum size of a sequential chunk

• task_name: the name displayed in the scheduler diagnostics

• prio: task priority

Returns: the return value type of calling func.

parallel_partial_sum

Description: Computes the partial sums of the elements in the subranges of the range [begin, end) and writes them to the range beginning at out. It uses the given binary function func to sum up the elements.

```
// version for iterators
template <class Iterator, class OutIterator, class Func, class Job=BOOST_ASYNCHI
boost::asynchronous::detail::callback_continuation<OutIterator,Job>
parallel_partial_sum(Iterator beg, Iterator end, OutIterator out, Func func, lo

// version for moved ranges
template <class Range, class OutRange, class Func, class Job=BOOST_ASYNCHRONOUS_
boost::asynchronous::detail::callback_continuation<std::pair<Range,OutRange>,Jol
parallel_partial_sum(Range&& range,OutRange&& out_range,Func func,long cutoff,cc

// version for a single moved range (in/out) => will return the range as contin
template <class Range, class Func, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_partial_sum(Range&& range,Func func,long cutoff,const std::string& tas]

// version taking a continuation of a range as first argument
template <class Range, class OutIterator, class Func, class Job=BOOST_ASYNCHRON(
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_partial_sum(Range range,Func func,long cutoff,const std::string& task_:
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's
job to ensure this.

Parameters:

- begin, end: the range of elements to sum

  Or range: a moved range.

  Or a continuation, coming from another algorithm.

- out_range: a moved range which will be returned upon completion

- func: binary operation function object that will be applied. The signature of the function should be
  equivalent to the following: Ret fun(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Returns: Iterator to the element past the last element written in the first version, an iterator and the
passed moved range in the second, the passed range in the third, a range returned by the continuation
in the fourth.

parallel_merge

Description: Merges two sorted ranges [begin1, end1) and [begin2, end2) into one sorted range
beginning at out. It uses the given comparison function func to compare the elements. For equivalent
elements in the original two ranges, the elements from the first range (preserving their original order)
precede the elements from the second range (preserving their original order). The behavior is undefined
if the destination range overlaps either of the input ranges (the input ranges may overlap each other).

```
template <class Iterator1, class Iterator2, class OutIterator, class Func, clas
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_merge(Iterator1 begin1, Iterator1 end1, Iterator2 beg2, Iterator2 end2
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's
job to ensure this.

Parameters:

- begin1, end1: the first range of elements to merge.

- begin2, end2: the second range of elements to merge.

- out: the beginning of the destination range

- func: comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following: bool func(const Type1 &a, const Type2 &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Returns: A void continuation.

parallel_invoke

parallel_invoke invokes a variadic list of predicates in parallel and returns a (continuation of) tuple of expected containing the result of all of them. Functors have to be wrapped within a **boost::asynchronous::to_continuation_task**.

```
template <class Job, typename... Args>
boost::asynchronous::detail::callback_continuation<std::tuple<expected<return t
parallel_invoke(Args&&... args);
```

Parameters:

- args: functors to call.

Returns: an expected of tuple of expected containing the result of every called functor.

Example:

Of course, the futures can have exceptions if exceptions are thrown, as in the following example:

```
post_callback(
                []()
                {
                    return boost::asynchronous::parallel_invoke<boost::asynchron
                                boost::asynchronous::to_continuation_task(
                                boost::asynchronous::to_continuation_task(
                },// work
                // the lambda calls Servant, just to show that all is safe, Ser
                [this](boost::asynchronous::expected<std::tuple<asynchronous::ex
                {
                    try
                    {
                        auto t = res.get();
                        std::cout << "got result: " << (std::get<1>(t)).get() <
                        std::cout << "got exception?: " << (std::get<0>(t)).has_
                    }
                    catch(std::exception& e)
                    {
                        std::cout << "got exception: " << e.what() << std::endl
```

```
                  }
               }// callback functor.
);
```

Notice the use of **to_continuation_task** to convert the lambdas in continuations.

As always, the callback lambda will be called when all tasks complete and the futures are non-blocking.

Please have a look at the complete example [examples/example_parallel_invoke.cpp].

if_then_else

Description: Executes a then or an else clause passed as continuations depending on an if clause. If clause is a functor returning a bool. Then and Else clauses are functors returning a continuation. Typically, if_then_else is called after an algorithm returning a continuation for further processing.

```
template <class IfClause, class ThenClause, class ElseClause, class Job=BOOST_A
Unspecified-Continuation-Type
if_then_else((IfClause if_clause, ThenClause then_clause, ElseClause else_claus
```

Parameters:

- if_clause: unary predicate function which returns a bool. The signature of the function should be equivalent to the following: bool pred(const Type &a); where Type is returned by a previous algorithm

- then_clause: unary predicate function which returns a continuation. The signature of the function should be equivalent to the following: Unspecified-Continuation pred(const Type &a); where Type is returned by a previous algorithm.

- else_clause: unary predicate function which returns a continuation. The signature of the function should be equivalent to the following: Unspecified-Continuation pred(const Type &a); where Type is returned by a previous algorithm.

- task_name: the name displayed in the scheduler diagnostics

Returns: A continuation of the type returned by then_clause or else_clause. Both must return the same continuation type.

Example: the following code calls a parallel_for (1) based on the result of another algorithm, a parallel_for (4). Both then_clause (2) and else_clause (3) will return a continuation representing a different parallel_for. The if-clause (5) makes the decision.

```
post_callback(
            // task
            [this](){
                    // the outer algorithm (1). Called with a continuation retu
                    return boost::asynchronous::parallel_for(
                                boost::asynchronous::if_then_else(
                                    // if clause (5). Here, always true.
                                    [](std::vector<int> const&){return true
                                    // then clause. Will be called, as if-c.
                                    [](std::vector<int> res)
                                    {
                                        std::vector<unsigned int> new_resul
                                        // This algorithm (2) will be calle
                                        return boost::asynchronous::paralle
                                                std::move(new_re
                                            [](unsigned int
                                            {
```

```
                                                        const_cast<ur
                                            },1500);
                                },
                                // else clause. Will NOT be called, as
                                [](std::vector<int> res)
                                {
                                    std::vector<unsigned int> new_resul
                                    // This algorithm (3) would be call
                                    return boost::asynchronous::paralle
                                                std::move(new_re
                                                [](unsigned int
                                                {
                                                    const_cast<ur
                                                },1500);
                                }
                            )
                            // argument of if_then_else, a continuation
                            (
                                boost::asynchronous::parallel_for(
                                                    std::m
                                                    [](int
                                                    {
                                                        con
                                                    },1500
                            ),
                    [](unsigned int const& i)
                    {
                        const_cast<unsigned int&>(i) += 1;
                    },1500
                );
                },
        // callback functor
        [](boost::asynchronous::expected<std::vector<unsigned int>> res){
                        auto modified_vec = res.get();
                        auto it = modified_vec.begin();
                        BOOST_CHECK_MESSAGE(*it == 7,"data[0] is wrong: "+ std:
                        std::advance(it,100);
                        BOOST_CHECK_MESSAGE(*it == 7,"data[100] is wrong: "+ sto
                        std::advance(it,900);
                        BOOST_CHECK_MESSAGE(*it == 7,"data[1000] is wrong: "+ s
                        std::advance(it,8999);
                        BOOST_CHECK_MESSAGE(*it == 7,"data[9999] is wrong: "+ s
                        auto r = std::accumulate(modified_vec.begin(),modified_
                        BOOST_CHECK_MESSAGE((r == 70000),
                                    ("result of parallel_for after if/e
                                     ", should have been 70000"));
        }
    );
```

parallel_geometry_intersection_of_x

Calculate the intersection of any number of (Boost.Geometry) geometries. The free function intersection calculates the spatial set theoretic intersection of geometries.

```
template <class Iterator, class Range,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Iterator-Reference,Job>
parallel_geometry_intersection_of_x(Iterator beg, Iterator end,const std::string
```

```
template <class Range, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range-Begin-Reference,Job>
parallel_geometry_intersection_of_x(Range&& range,const std::string& task_name=

// version taking a continuation of a range as first argument
template <class Range, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_geometry_intersection_of_x(Range range,const std::string& task_name=""
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Returns: a geometry of the type referenced by the iterator / contained in the range

Parameters:

- beg, end: the range of input geometries

  Or range: a moved range.

  Or a continuation, coming from another algorithm.

- cutoff: the maximum size of a sequential chunk of geometries

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- overlay_cutoff: performance tuning. Cutoff used for the parallelization of the internal overlay algorithm

- partition_cutoff: performance tuning. Cutoff used for the parallelization of the internal partition algorithm

parallel_geometry_union_of_x

Combines any number of (Boost.Geometry) geometries which each other. he free function union calculates the spatial set theoretic union of geometries.

```
template <class Iterator, class Range,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_geometry_union_of_x(Iterator beg, Iterator end,const std::string& task_

template <class Range, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_geometry_union_of_x(Range&& range,const std::string& task_name="", std

// version taking a continuation of a range as first argument
template <class Range, class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Unspecified-Range,Job>
parallel_geometry_union_of_x(Range range,const std::string& task_name="", std::
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Returns: a sequence containing the resulting geometries

Parameters:

- beg, end: the range of geometries to combine

Or range: a moved range.

Or a continuation, coming from another algorithm.

- cutoff: the maximum size of a sequential chunk of geometries

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- overlay_cutoff: performance tuning. Cutoff used for the parallelization of the internal overlay algorithm

- partition_cutoff: performance tuning. Cutoff used for the parallelization of the internal partition algorithm

parallel_union

Combines two geometries which each other.

```
template <class Geometry1, class Geometry2, class Collection,class Job=BOOST_AS
boost::asynchronous::detail::callback_continuation<Collection,Job>
parallel_union(Geometry1 geometry1,Geometry2 geometry2,long overlay_cutoff,long
```

Returns: a geometry. Currently, Type of Geometry1= Type of Geometry2 = Type of Collection

Parameters:

- geometry1, geometry2: the geometries to combine

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- overlay_cutoff: performance tuning. Cutoff used for the parallelization of the internal overlay algorithm

- partition_cutoff: performance tuning. Cutoff used for the parallelization of the internal partition algorithm

parallel_intersection

Calculate the intersection of two geometries.

```
template <class Geometry1, class Geometry2, class Collection,class Job=BOOST_AS
boost::asynchronous::detail::callback_continuation<Collection,Job>
parallel_intersection(Geometry1 geometry1,Geometry2 geometry2,long overlay_cuto
```

Returns: a geometry. Currently, Type of Geometry1= Type of Geometry2 = Type of Collection

Parameters:

- geometry1, geometry2: the geometries to combine

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

- overlay_cutoff: performance tuning. Cutoff used for the parallelization of the internal overlay algorithm

- partition_cutoff: performance tuning. Cutoff used for the parallelization of the internal partition algorithm

parallel_find_all

Finds and copies into a returned container all elements of a range for which a predicate returns true.

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- beg, end: the range of elements to search

  Or range: a moved range.

  Or a continuation, coming from another algorithm.

- func: unary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool pred(const Type &a);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Returns: a container with the searched elements. Default is a std::vector for the iterator version, a range of the same type as the input range for the others.

```
template <class Iterator, class Func,
          class ReturnRange=std::vector<...>,
          class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<ReturnRange,Job>
parallel_find_all(Iterator beg, Iterator end,Func func,long cutoff,const std::st

template <class Range, class Func, class ReturnRange=Range, class Job=BOOST_ASYN
boost::asynchronous::detail::callback_continuation<ReturnRange,Job>
parallel_find_all(Range&& range,Func func,long cutoff,const std::string& task_na

// version taking a continuation of a range as first argument
template <class Range, class Func, class ReturnRange=typename Range::return_type
boost::asynchronous::detail::callback_continuation<ReturnRange,Job>
parallel_find_all(Range range,Func func,long cutoff,const std::string& task_name
```

Example:

```
std::vector<int> data;
post_callback(
    [this]()
    {
        return boost::asynchronous::parallel_find_all(this->data.begin(),this->da
                                      [](int i)
                                      {
                                          return (400 <= i) && (.
                                      },
                                      1500);
    },
    ](boost::asynchronous::expected<std::vector<int>> ){}
```

```
);
```

Please have a look at the complete example [examples/example_parallel_find_all.cpp].

parallel_extremum

parallel_extremum finds an extremum (min/max) of a range given by a predicate.

```
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<typename std::iterator_traits
parallel_extremum(Iterator beg, Iterator end,Func func,long cutoff,const std::st

template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
decltype(boost::asynchronous::parallel_reduce(...))
parallel_extremum(Range&& range,Func func,long cutoff,const std::string& task_na

// version taking a continuation of a range as first argument
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
decltype(boost::asynchronous::parallel_reduce(...))
parallel_extremum(Range range,Func func,long cutoff,const std::string& task_name
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- beg, end: the range of elements to search

  Or range: a moved range.

  Or a continuation, coming from another algorithm.

- func: binary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Please have a look at the complete example [examples/example_parallel_extremum.cpp].

parallel_count / parallel_count_if

parallel_count counts the elements of a range satisfying a predicate.

```
template <class Iterator, class T,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<long,Job>
parallel_count(Iterator beg, Iterator end,T const& value,long cutoff,const std:
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<long,Job>
parallel_count_if(Iterator beg, Iterator end,Func func,long cutoff,const std::st

template <class Range, class T,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<long,Job>
parallel_count(Range&& range,T const& value,long cutoff,const std::string& task_
template <class Range, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
parallel_count_if(Range&& range,Func func,long cutoff,const std::string& task_na
```

```
// version taking a continuation of a range as first argument
template <class Range, class T,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<long,Job>
parallel_count(Range range,T const& value,long cutoff,const std::string& task_na
template <class Range, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<long,Job>
parallel_count_if(Range range,Func func,long cutoff,const std::string& task_name
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

• beg, end: the range of elements

  Or range: a moved range.

  Or a continuation, coming from another algorithm.

• T value: the value to search for

• func: unary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool func(const Type &a);

• cutoff: the maximum size of a sequential chunk

• task_name: the name displayed in the scheduler diagnostics

• prio: task priority

Please have a look at the complete example [examples/example_parallel_count.cpp].

parallel_sort / parallel_stable_sort / parallel_spreadsort / parallel_sort_inplace / parallel_stable_sort_inplace / parallel_spreadsort_inplace

parallel_sort / parallel_stable_sort implement a parallel mergesort. parallel_spreadsort is a parallel version of Boost.Spreadsort if BOOST_ASYNCHRONOUS_USE_BOOST_SPREADSORT is defined. They all use a parallel mergesort. For the sequential part, parallel_sort uses std::sort, parallel_stable_sort uses std::stable_sort, parallel_spreadsort uses Boost.Spreadsort.

parallel_sort_inplace / parallel_stable_sort_inplace / parallel_spreadsort_inplace use an inplace merge to save memory, at the cost of a performance penalty. For the sequential part, parallel_sort_inplace uses std::sort, parallel_stable_sort_inplace uses std::stable_sort, parallel_spreadsort_inplace uses Boost.Spreadsort.

```
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_sort(Iterator beg, Iterator end,Func func,long cutoff,const std::string
parallel_stable_sort(Iterator beg, Iterator end,Func func,long cutoff,const std
parallel_spreadsort(Iterator beg, Iterator end,Func func,long cutoff,const std:
parallel_sort_inplace(Iterator beg, Iterator end,Func func,long cutoff,const sto
parallel_stable_sort_inplace(Iterator beg, Iterator end,Func func,long cutoff,co
parallel_spreadsort_inplace(Iterator beg, Iterator end,Func func,long cutoff,cor
```

```
// version taking ownership of the container to be sorted
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_sort(Range&& range,Func func,long cutoff,const std::string& task_name=
parallel_stable_sort(Range&& range,Func func,long cutoff,const std::string& tas
parallel_spreadsort(Range&& range,Func func,long cutoff,const std::string& task_
parallel_sort_inplace(Range&& range,Func func,long cutoff,const std::string& ta
parallel_stable_sort_inplace(Range&& range,Func func,long cutoff,const std::str
parallel_spreadsort_inplace(Range&& range,Func func,long cutoff,const std::stri

// version taking a continuation of a range as first argument
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<Range,Job>
parallel_sort(Range range,Func func,long cutoff,const std::string& task_name=""
parallel_stable_sort(Range range,Func func,long cutoff,const std::string& task_
parallel_spreadsort(Range range,Func func,long cutoff,const std::string& task_n
parallel_sort_inplace(Range range,Func func,long cutoff,const std::string& task_
parallel_stable_sort_inplace(Range range,Func func,long cutoff,const std::string
parallel_spreadsort_inplace(Range range,Func func,long cutoff,const std::string
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- beg, end: the range of elements. Returns nothing.

  Or range: a moved range. Returns the sorted moved range.

  Or a continuation, coming from another algorithm. Returns the sorted range.

- func: binary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Please have a look at the complete example [examples/example_parallel_count.cpp].

parallel_partial_sort

Rearranges elements such that the range [begin, middle) contains the sorted middle - begin smallest elements in the range [begin, end). The order of equal elements is not guaranteed to be preserved. The order of the remaining elements in the range [begin, end) is unspecified. It uses the given comparison function func.

```
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_partial_sort(Iterator begin, Iterator middle, Iterator end,Func func,l
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end: the range of elements.

- middle: until where the range will be sorted

- func: binary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_quicksort / parallel_quick_spreadsort

Sorts the range [begin,end) using quicksort. The order of the remaining elements in the range [begin, end) is unspecified. It uses the given comparison function func. parallel_quicksort will use std::sort to sort when the algorithm finishes partitioning. parallel_quick_spreadsort will use Boost.Spreadsort for this.

```
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_quicksort(Iterator begin, Iterator end,Func func,long cutoff,const std
```

```
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_quick_spreadsort(Iterator begin, Iterator end,Func func,long cutoff,con
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end: the range of elements.

- func: binary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

parallel_nth_element

nth_element is a partial sorting algorithm that rearranges elements in [begin, end) such that: The element pointed at by nth is changed to whatever element would occur in that position if [begin, end) was sorted. All of the elements before this new nth element are less than or equal to the elements after the new nth element. It uses the given comparison function func.

```
template <class Iterator, class Func,class Job=BOOST_ASYNCHRONOUS_DEFAULT_JOB>
boost::asynchronous::detail::callback_continuation<void,Job>
parallel_nth_element(Iterator begin, Iterator nth, Iterator end,Func func,long
```

The version taking iterators requires that the iterators stay valid until completion. It is the programmer's job to ensure this.

Parameters:

- begin, end: andom access iterators defining the range sort.

- nth: random access iterator defining the sort partition point

- func: binary predicate function which returns true for searched elements. The signature of the function should be equivalent to the following: bool func(const Type &a, const Type &b);

- cutoff: the maximum size of a sequential chunk

- thread_num: the number of threads in the pool executing the algorithm

- task_name: the name displayed in the scheduler diagnostics

- prio: task priority

Parallel containers

Any gain made by using a parallel algorithm can be reduced to nothing if the calling codes spends most of its time creating a std::vector. Interestingly, most parallel libraries provide parallel algorithms, but very few offer parallel data structures. This is unfortunate because a container can be parallelized with a great gain as long as the contained type either has a non-simple constructor / destructor or simply is big enough, as our tests show (see test/perf/perf_vector.cpp). Though memory allocating is not parallel, constructors can be made so. Reallocating and resizing, adding elements can also greatly benefit.

Asynchronous fills this gap by providing boost::asynchronous::vector. It can be used like std::vector by default.

However, it can also be used as a parallel, synchronous type if provided a threadpool. Apart from the construction, it looks and feels very much like a std::vector. In this case, it cannot be posted to its own threadpool without releasing it (see release_scheduler / set_scheduler) as it would create a cycle, and therefore a possible deadlock. It is defined in:

#include <boost/asynchronous/container/vector.hpp>

The vector supports the same constructors that std::vector, with as extra parameters, the threadpool for parallel work, and a cutoff. Optionally, a name used for logging and a threadpool priority can be given, for example:

```
struct LongOne;
boost::asynchronous::any_shared_scheduler_proxy<> pool =
                boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::multiqueue_threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<>
                    >>(tpsize,tasks);

boost::asynchronous::vector<LongOne> vec (pool,1024 /* cutoff */, /* std::vecto
                                         // optional, name for logging, prio
                                         , "vector", 1);
```

At this point, asynchronous::vector can be used like std::vector, with the difference that constructor, destructor, operator=, assign, clear, push_back, emplace_back, reserve, resize, erase, insert are executed in parallel in the given threadpool.

The vector adds a few members compared to std::vector:

- release_scheduler(): removes the threadpool from vector. At this point, the vector is no more parallel, but can live from within the pool.

- set_scheduler(): (re)sets scheduler, so that vector is again parallel. At this point, the vector cannot live from within the pool.

- long get_cutoff() const: returns the cutoff as given in constructor.

- std::string get_name() const: the logged name, as given in the constructor.

- std::size_t get_prio()const: the priority, as given in the constructor.

This example [examples/example_vector.cpp] displays some basic usage of vector.

Table 3.10. #include <boost/asynchronous/container/vector.hpp>

Public Member functions as in std::vector	Description	Parallel if threadpool?
(constructor)	constructs the vector	Yes
(destructor)	destructs the vector	Yes
operator=	assigns values to the container	Yes
assign	assigns values to the container	Yes
get_allocator	returns the associated allocator	No
at	access specified element with bounds checking	No
operator[]	access specified element	No
front	access the first element	No
back	access the last element	No
data	direct access to the underlying array	No
begin / cbegin	returns an iterator to the beginning	No
end / cend	returns an iterator to the end	No
rbegin / crbegin	returns a reverse iterator to the beginning	No
rend / crend	returns a reverse iterator to the end	No
empty	checks whether the container is empty	No
size	returns the number of elements	No
max_size	returns the maximum possible number of elements	No
reserve	reserves storage	Yes
capacity	returns the number of elements that can be held in currently allocated storage	Yes
shrink_to_fit	reduces memory usage by freeing unused memory	Yes
clear	clears the contents	Yes

Public Member functions as in std::vector	Description	Parallel if threadpool?
insert	inserts elements	Yes
emplace	constructs element in-place	Yes
erase	erases elements	Yes
push_back	adds an element to the end	Yes
emplace_back	constructs an element in-place at the end	Yes
pop_back	removes the last element	No
resize	changes the number of elements stored	Yes
swap	swaps the contents	No
operator==	lexicographically compares the values in the vector	Yes
operator!=	lexicographically compares the values in the vector	Yes
operator<	lexicographically compares the values in the vector	Yes
operator<=	lexicographically compares the values in the vector	Yes
operator>	lexicographically compares the values in the vector	Yes
operator>=	lexicographically compares the values in the vector	Yes

All these members have the same signature as std::vector. Only some constructors are new. First the standard ones:

```
vector();

explicit vector( const Allocator& alloc );

vector( size_type count,
        const T& value = T(),
        const Allocator& alloc = Allocator());

template< class InputIt >
vector( InputIt first, InputIt last,
        const Allocator& alloc = Allocator() );

vector( const vector& other );

vector( vector&& other )
```

```
vector( std::initializer_list<T> init,
        const Allocator& alloc = Allocator() );
```

There are variants taking a scheduler making them a servant with parallel capabilities:

```
explicit vector(boost::asynchronous::any_shared_scheduler_proxy<Job> scheduler,
                long cutoff,
                const std::string& task_name="",
                std::size_t prio=0,
                const Alloc& alloc = Alloc());

explicit vector( boost::asynchronous::any_shared_scheduler_proxy<Job> scheduler
                 long cutoff,
                 size_type count,
                 const std::string& task_name="",
                 std::size_t prio=0,
                 const Allocator& alloc = Allocator());

explicit vector( boost::asynchronous::any_shared_scheduler_proxy<Job> scheduler
                 long cutoff,
                 size_type count,
                 const T& value = T(),
                 const std::string& task_name="",
                 std::size_t prio=0,
                 const Allocator& alloc = Allocator());

        template< class InputIt >
        vector( boost::asynchronous::any_shared_scheduler_proxy<Job> scheduler
                long cutoff,
                InputIt first, InputIt last,
                const std::string& task_name="",
                std::size_t prio=0,
                const Allocator& alloc = Allocator() );

        vector( boost::asynchronous::any_shared_scheduler_proxy<Job> scheduler
                long cutoff,
                std::initializer_list<T> init,
                const std::string& task_name="",
                std::size_t prio=0,
                const Allocator& alloc = Allocator() );
```

Some new members have also been added to handle the new functionality:

Table 3.11. #include <boost/asynchronous/container/vector.hpp>

Public Member functions as in std::vector	Description	Parallel if threadpool?
void set_scheduler(any_shared_scheduler_proxy<Job>)	adds / replaces the scheduler pool	No
void release_scheduler()	Removes the scheduler pool. vector is now "standard"	No
long get_cutoff()const	returns cutoff	No
void set_cutoff(long)	sets cutoff	No

Public Member functions as in std::vector	Description	Parallel if threadpool?
std::string get_name() const	returns vector name, used in task names	No
void set_name(std::string const&)	sets vector name, used in task names	No
std::size_t get_prio()const	returns vector task priority in threadpool	No
void set_prio(std::size_t)	set vector task priority in threadpool	No

Chapter 4. Tips.

Which protections you get, which ones you don't.

Asynchronous is doing much to protect developers from some ugly beasts around:

- (visible) threads

- races

- deadlocks

- crashes at the end of an object lifetime

It also helps parallelizing and improve performance by not blocking. It also helps find out where bottlenecks and hidden possible performance gains are.

There are, however, things for which it cannot help:

- cycles in design

- C++ legal ways to work around the protections if one really wants.

- blocking on a future if one really wants.

- using "this" captured in a task lambda.

- writing a not clean task with pointers or references to data used in a servant.

No cycle, ever

This is one of the first things one learns in a design class. Cycles are evil. Everybody knows it. And yet, designs are often made without care in a too agile process, dependency within an application is not thought out carefully enough and cycles happen. What we do learn in these classes is that cycles make our code monolithic and not reusable. What we however do not learn is how bad, bad, bad this is in face of threads. It becomes impossible to follow the flow of information, resource usage, degradation of performance. But the worst of all, it becomes almost impossible to prevent deadlocks and resource leakage.

Using Asynchronous will help write clean layered architectures. But it will not replace carefully crafted designs, thinking before writing code and the experience which make a good designer. Asynchronous will not be able to prevent code having cycles in a design.

Fortunately, there is an easy solution: back to the basics, well-thought designs before coding, writing diagrams, using a real development process (hint: an agile "process" is not all this in the author's mind).

No "this" within a task.

A very easy way to see if you are paving the way to a race even using Asynchronous is to have a look at the captured variables of a lambda posted to a threadpool. If you find "this", it's probably bad, unless you really know that the single-thread code will do nothing. Apart from a simple application, this will not be true. By extension, pointers, references, or even shared smart pointers pointing to data living in a single-thread world is usually bad.

Experience shows that there are only two safe way to pass data to a posted task: copy for basic types or types having a trivial destructor and move for everything else. Keep to this rule and you will be safe.

On the other hand, "this" is okay in the capture list of a callback task as Asynchronous will only call it if the servant is still alive.

Chapter 5. Design examples

This section shows some examples of strongly simplified designs using Asynchronous.

A state machine managing a succession of tasks

This example will show how the library allows a solution more powerful than the proposed future.then() extension.

Futures returned by an asynchronous function like std::async have to be get()-ed by the caller before a next task can be started. To overcome this limitation, a solution is to add a then(some_functor) member to futures. The idea is to chain several tasks without having to get() the first future. While this provides some improvement, some serious limitations stay:

- A future still has to be waited for

- The then-functor is executed in one of the worker threads

- The then-functor has to be complete at the first call. By complete we mean that it should not use any data from the caller thread to avoid races.

All this makes from the then functor a fire-and-forget one and prevents reacting on changes happening between the first functor and the then functor.

A superior solution exists using Asynchronous schedulers. In this example [examples/ example_callback.cpp], we define a Manager object, which lives in his single thread scheduler. This Manager, a simplified state machine, starts a task when asked (calling start() on its proxy). Upon completing the first task, the Manager chooses to start or not the second part of the calculation (what would be done in future.then()). In our example, an event cancels the second part (calling cancel()) so that it never starts.

Notice in this example some important points:

- The time we choose to start or not the second part is done after the first part completes, not before, which is a noticeable improvement compared to future.then().

- We have no race although no mutex is used and at least three threads are implied (main / 2 std::thread / Manager's world / threadpool). All events (start / cancel /completion of first task / completion of second task) are done within the Manager thread world.

- A proxy object can be copied and the copies used safely in any number of threads. The proxy is copied, the thread world no. We use in our example 2 std::threads which both share the proxy (and share control of the thread world's lifecycle) with the main thread and access members of the servant, all safely. The last thread going will cause the thread world to shutdown.

- Thinking in general design, this is very powerful. Software is usually designed in components, which one tries to make reusable. This is usually difficult because of thread issues. This problem is now gone. The component delimited by one (or several) proxy is safe, completely reusable and its thread limits are well defined.

- We have in this example only one servant and its proxy. It would be easily possible to define any number of pair of those. In this case, the last proxy destroyed would shut down the thread world.

We can also write the same example using a real Boost.MSM state machine [examples/ example_callback_msm.cpp]

A layered application

A common design pattern for an application is organizing it into layers. We suppose we are having three layers:
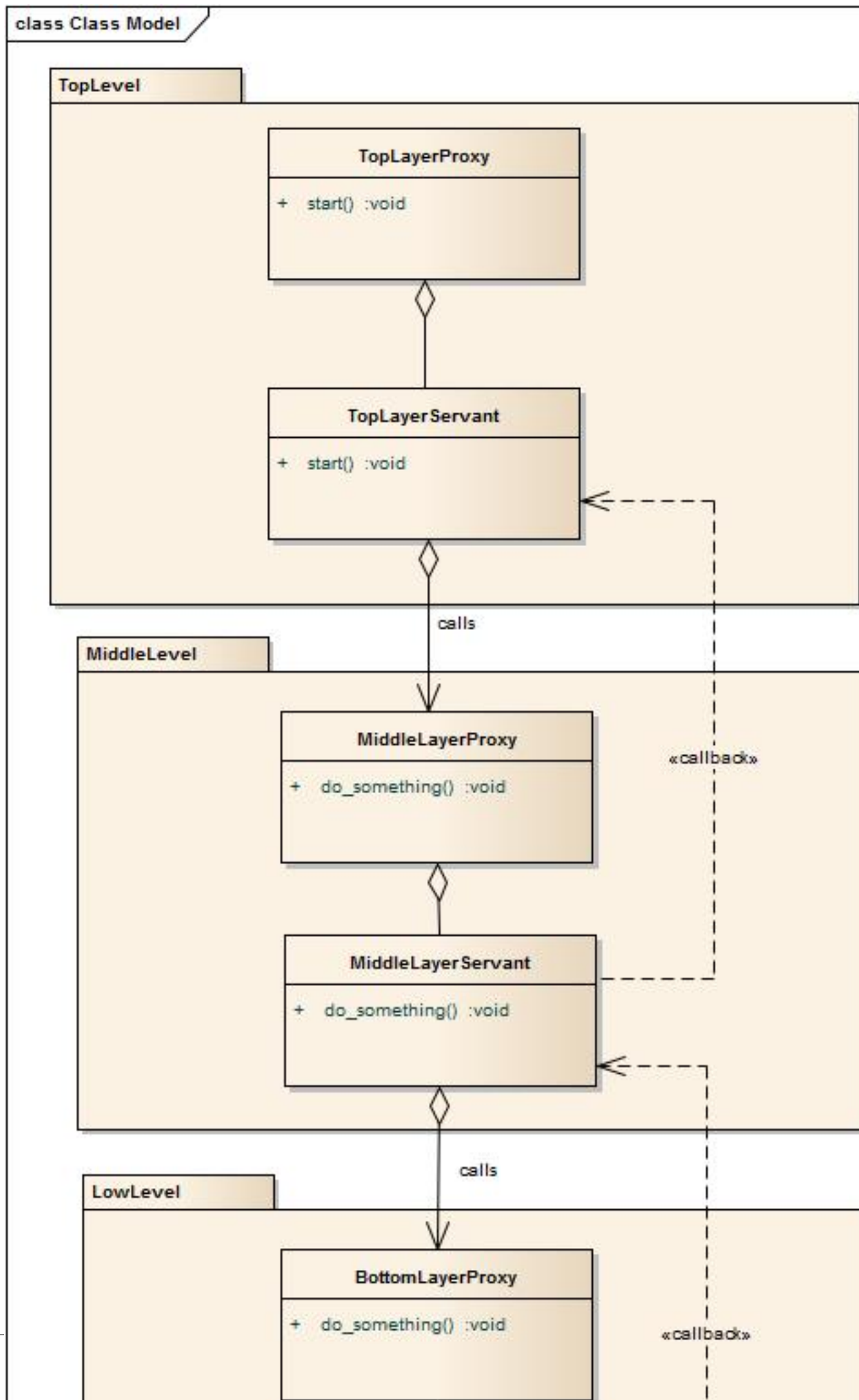
- TopLevel: what the user of the application is seeing

- MiddleLevel: some internal business logic

- LowLevel: communication layer

This is a common design in lots of books. A top level layer receives user commands , checks for their syntax, then delegates to a middle layer, composed of business logic, checking whether the application is in a state to execute the order. If it is, the low-level communication task is delegated to a low level layer.

Of course this example is strongly simplified. A layer can be made of hundreds of objects and thousands of lines of code.

What the books often ignore, however, are threads and lifecycle issues. Non-trivial applications are likely to be running many threads. This is where the problems start:

- Which layer is running into which thread?

- How do to avoid races if each layer is running his own thread(s). Usually, mutexes are used.

- How to handle callbacks from lower layers, as these are likely to be executed in the lower layer thread(s)

- Lifecycles. Usually, each layer has responsibility of keeping his lower layers alive. But how to handle destruction of higher-levels? Callbacks might already be under way and they will soon meet a destroyed mutex?

class Class Model

TopLevel

TopLayerProxy

+   start() :void

TopLayerServant

+   start() :void

calls

MiddleLevel

MiddleLayerProxy

+   do_something() :void

«callback»

MiddleLayerServant

+   do_something() :void

calls

LowLevel

BottomLayerProxy

+   do_something() :void

«callback»

Asychronous provides a solution to these problems:

- Each layer is living in its own thread world (or sharing one).

- Asynchronous guarantees that each layer will be destroyed in turn: LowLevel - MiddleLevel - TopLevel.

- Asynchronous provides proxies to serialize outside calls into a servant thread world.

- Asynchronous provides safe callbacks: each servant can use make_safe_calback, which guarantees execution in the servant thread if and only if the servant is still alive.

In this simplified example [examples/example_layers.cpp], each layer has its own thread world. Using the proxies provided by the library, each servant is protected from races through calls from their upper layer or the outside world (main). Servants are also protected from callbacks from their lower layer by make_safe_callback.

Boost.Meta State Machine and Asynchronous behind a Qt User Interface

We will now implement an application closer to a real-life case. The code is to be found under asynchronous/libs/asynchronous/doc/examples/msmplayer/ .

The Boost.MSM documentation introduces a CD player implementation. This CD player reacts to events coming from a hardware: opening of the player, adding a disc, playing a song, increasing volume, etc. This logic is implemented in the logic layer, which mostly delegates it to a state machine based on Boost.MSM. The file playerlogic.cpp implements this state machine, which is an extension of the ones found in the Boost.MSM documentation.

This logic is to be thought of as a reusable component, which must be thread-safe, living in a clearly defined thread context. Asychronous handles this: the PlayerLogic class is a servant, protected by a proxy, PlayerLogicProxy, which in its constructor creates the necessary single-threaded scheduler. At this point, we have a self-sufficient component.

Supposing that, like often in real-life, that the hardware is not ready while our software would be ready for testing, we decide to build a Qt application, acting as a hardware simulator. This also allows fast prototyping, early testing, writing of training material and concept-checking by the different stakeholders.

To achieve this, we write a QWidget-derived class named PlayerGui, a simple interface simulating the controls which will be offered by the real CD player. It implements IDisplay, the interface which the real CD player will provide.

The real hardware will also implement IHardware, an interface allowing control of the buttons and motors of the player. Our simple PlayerGui will also implement it for simplicity.

A Qt application is by definition asynchronous. Boost.Asynchronous provides qt_servant, allowing a Qt object to make us of the library features (safe callbacks, threadpools, etc.).

The application is straightforward: PlayerGui creates the logic of the application, which means constructing a PlayerLogicProxy object. After the construction, we have a usable, movable, perfectly asynchronous component, which means being based on an almost 0 time run-to-completion implemented by the state machine. The PlayerGui itself is also fully asynchronous: all actions it triggers are posted into the logic component, and therefore non-blocking. After the logic updates its internal states, it calls a provided safe callback, which will update the status of all buttons. So we now have an asynchronous, non blocking user interface delegating handling the hardware to an asynchronous, non blocking logic layer:

- PlayerGui(.h [examples/msmplayer/playergui.h].cpp [examples/msmplayer/playergui.cpp]): a QWidget providing a simple user interface with buttons like the real hardware would. It inherits qt_servant to get access to safe callbacks. It provides SafeDisplay, an implementation of IDisplay, interface which the real CD player will also implement, and SafeHardware, an implementation of IHardware, an interface which the real hardware will implement. The "Safe" part is that the callbacks being passed are resulting from make_safe_callback: they will be executed within the Qt thread, only if the QWidget is still alive.

- PlayerLogic(.h [examples/msmplayer/playerlogic.h].cpp [examples/msmplayer/playerlogic.cpp]): the entry point to our logic layer: a trackable_servant, hiden behind a servant_proxy. In our example, it will delegate all logic work to the state machine.

- StateMachine [examples/msmplayer/playerlogic.cpp]: a Boost.MSM state machine, implementing the whole CD player logic.

- IDisplay [examples/msmplayer/idisplay.h]: the user interface provided by the real player.

- IHardware [examples/msmplayer/ihardware.h]: the interface provided by the real hardware (buttons, motors, sensor, etc).

This example shows very important concepts of the Boost.MSM and Asynchronous libraries in actions:

- A state machine is based on run-to-completion: it is unstable while processing an event and should move as fast as possible back to a stable state.

- To achieve this, the actions should be short. Whatever takes long is posted to a thread pool. Our actions are costing only the cost of a transition and posting.

- Asynchronous provides the infrastructure needed by the state machine: pools, safe callbacks, protection from external threads.

- A logic component is behaving like a simple, safe, moveable object. All the application sees is a proxy object.

- A Qt Object can also make use of thread pools, safe callbacks.

- Our application is completely asynchronous: it never ever blocks. This is no small feat when we consider that we are controlling a "real" hardware with huge response times compared to the speed of a CPU.

Please have a closer look at the following implementation details:

- PlayerLogicProxy [examples/msmplayer/playerlogic.h] has a future-free interface. Its members take a callback. This is a sign there will be no blocking.

- The state machine in PlayerLogic's state machine uses post_callback for long tasks. In the callback, the next event processing will start.

- The UI (PlayerGui) is also non-blocking. We make use of callbacks (look at the make_safe_callback calls). We therefore have a very responsive UI.

- PlayerGui::actionsHandler will set the new state of all buttons each time the state machine updates its status.

Part III. Reference

Table of Contents

Chapter 6. Queues

Asynchronous provides a range of queues with different trade-offs. Use `lockfree_queue` as default for a quickstart with Asynchronous.

threadsafe_list

This queue is mostly the one presented in Anthony Williams' book, "C++ Concurrency In Action". It is made of a single linked list of nodes, with a mutex at each end of the queue to minimize contention. It is reasonably fast and of simple usage. It can be used in all configurations of pools.

Its constructor does not require any parameter forwarded from the scheduler.

Stealing: from the same queue end as pop. Will be implemented better (from the other end to reduce contention) in a future version.

Caution: crashes were noticed with gcc 4.8 while 4.7 and clang 3.3 seemed ok though the compiler might be the reason. For this reason, lockfree_queue is now the default queue.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class threadsafe_list;
```

lockfree_queue

This queue is a light wrapper around a `boost::lockfree::queue`, which gives lockfree behavior at the cost of an extra dynamic memory allocation. Please use this container as default when starting with Asynchronous.

The container is faster than a `threadsafe_list`, provided one manages to set the queue size to an optimum value. A too small size will cause expensive memory allocations, a too big size will significantly degrade performance.

Its constructor takes optionally a default size forwarded from the scheduler.

Stealing: from the same queue end as pop. Stealing from the other end is not supported by `boost::lockfree::queue`. It can be used in all configurations of pools.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class lockfree_queue;
```

lockfree_spsc_queue

This queue is a light wrapper around a `boost::lockfree::spsc_queue`, which gives lockfree behavior at the cost of an extra dynamic memory allocation.

Its constructor requires a default size forwarded from the scheduler.

Stealing: None. Stealing is not supported by `boost::lockfree::spsc_queue`. It can only be used Single-Producer / Single-Consumer, which reduces its typical usage to a queue of a `multiqueue_threadpool_scheduler` as consumer, with a `single_thread_scheduler` as producer.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class lockfree_spsc_queue;
```

lockfree_stack

This queue is a light wrapper around a `boost::lockfree::stack`, which gives lockfree behavior at the cost of an extra dynamic memory allocation. This container creates a task inversion as the last posted tasks will be executed first.

Its constructor requires a default size forwarded from the scheduler.

Stealing: from the same queue end as pop. Stealing from the other end is not supported by `boost::lockfree::stack`. It can be used in all configurations of pools.

Declaration:

```
template<class JOB = boost::asynchronous::any_callable>
class lockfree_stack;
```

Chapter 7. Schedulers

There is no perfect scheduler. In any case it's a question of trade-off. Here are the schedulers offered by Asynchronous.

single_thread_scheduler

The scheduler of choice for all servants which are not thread-safe. Serializes all calls to a single queue and executes them in order. Using `any_queue_container` as queue will however allow it to support task priority.

This scheduler does not steal from other queues or pools, and does not get stolen from to avoid races.

Declaration:

```
template<class Queue, class CPULoad>
class single_thread_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
          boost::asynchronous::single_thread_scheduler<
              boost::asynchronous::lockfree_queue<>>>();

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
        boost::asynchronous::single_thread_scheduler<
            boost::asynchronous::lockfree_queue<>>>(10); // size of queue
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<std::chrono::high_resolution_clock> s

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                              boost::asynchronous::single_thread_scheduler<
                                  boost::asynchronous::threadsafe_list<serva

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                              boost::asynchronous::single_thread_scheduler<
                                  boost::asynchronous::lockfree_queue<servan
```

Table 7.1. #include <boost/asynchronous/scheduler/ single_thread_scheduler.hpp>

Characteristics	
Number of threads	1
Can be stolen from?	No
Can steal from other threads in this pool?	N/A (only 1 thread)
Can steal from other threads in other pools?	No

multiple_thread_scheduler

The scheduler is an extended version of single_thread_scheduler, where all servants are operated by only one thread at a time, though not always the same one. It creates a n (servants) to m (threads) dependency. The advantages of this scheduler is that one long task will not block other servants, more flexibility in distributing threads among servants, and better cache behaviour (a thread tries to serve servants in order).

This scheduler does not steal from other queues or pools, and does not get stolen from to avoid races.

Declaration:

```
template<class Queue, class CPULoad>
class multiple_thread_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
            boost::asynchronous::multiple_thread_scheduler<
                boost::asynchronous::lockfree_queue<>>>(n,m); // n: max number of

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
            boost::asynchronous::multiple_thread_scheduler<
                boost::asynchronous::lockfree_queue<>>>(n,m,10); // n: max number
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<std::chrono::high_resolution_clock> s

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                                boost::asynchronous::single_thread_scheduler<
                                    boost::asynchronous::threadsafe_list<serva

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                                boost::asynchronous::single_thread_scheduler<
                                    boost::asynchronous::lockfree_queue<servan
```

Table 7.2. #include <boost/asynchronous/scheduler/ single_thread_scheduler.hpp>

Characteristics	
Number of threads	1..n
Can be stolen from?	No
Can steal from other threads in this pool?	No
Can steal from other threads in other pools?	No

threadpool_scheduler

The simplest and easiest threadpool using a single queue, though multiqueue behavior could be done using `any_queue_container`. The advantage is that it allows the pool to be given 0 thread and only be stolen from. The cost is a slight performance loss due to higher contention on the single queue.

This pool does not steal from other pool's queues.

Use this pool as default for a quickstart with Asynchronous.

Declaration:

```
template<class Queue,class CPULoad>
class threadpool_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::threadpool_scheduler<
                        boost::asynchronous::threadsafe_list<>>>(4); // 4 thre

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<>>>(4,10); // siz
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<std::chrono::high_resolution_clock> s

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::threadpool_scheduler<
                        boost::asynchronous::threadsafe_list<servant_job>>>(4

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<servant_job>>>(4,
```

Table 7.3. #include <boost/asynchronous/scheduler/threadpool_scheduler.hpp>

Characteristics	
Number of threads	**0**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	N/A (only 1 queue)
Can steal from other threads in other pools?	No

multiqueue_threadpool_scheduler

This is a `threadpool_scheduler` with multiple queues to reduce contention. On the other hand, this pool requires at least one thread.

This pool does not steal from other pool's queues though pool threads do steal from each other's queues.

Declaration:

```
template<class Queue,class FindPosition=boost::asynchronous::default_find_posit
class multiqueue_threadpool_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
```

```
boost::asynchronous::make_shared_scheduler_proxy<
                boost::asynchronous::multiqueue_threadpool_scheduler<
                        boost::asynchronous::threadsafe_list<>>>(4); // 4 thr

boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                boost::asynchronous::multiqueue_threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<>>>(4,10); // siz
```

Or, using logging:

```
typedef boost::asynchronous::any_loggable<std::chrono::high_resolution_clock> s

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                boost::asynchronous::multiqueue_threadpool_scheduler<
                        boost::asynchronous::threadsafe_list<servant_job>>>(4

boost::asynchronous::any_shared_scheduler_proxy<servant_job> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                boost::asynchronous::multiqueue_threadpool_scheduler<
                        boost::asynchronous::lockfree_queue<servant_job>>>(4,
```

Table 7.4. #include <boost/asynchronous/scheduler/ multiqueue_threadpool_scheduler.hpp>

Characteristics	
Number of threads	**1**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	Yes
Can steal from other threads in other pools?	No

stealing_threadpool_scheduler

This is a `threadpool_scheduler` with the added capability to steal from other pool's queues within a `composite_threadpool_scheduler`. Not used within a `composite_threadpool_scheduler`, it is a standard `threadpool_scheduler`.

Declaration:

```
template<class Queue,class CPULoad, bool /* InternalOnly */ = true >
class stealing_threadpool_scheduler;
```

Creation if used within a `composite_threadpool_scheduler`:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                boost::asynchronous::stealing_threadpool_scheduler<
                        boost::asynchronous::threadsafe_list<>>>(4); // 4 thr
```

However, if used stand-alone, which has little interest outside of unit tests, we need to add a template parameter to inform it:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                boost::asynchronous::stealing_threadpool_scheduler<
                        boost::asynchronous::threadsafe_list<>,true >>(4); //
```

Table 7.5. #include <boost/asynchronous/scheduler/ stealing_threadpool_scheduler.hpp>

Characteristics	
Number of threads	**0**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	N/A (only 1 queue)
Can steal from other threads in other pools?	Yes

stealing_multiqueue_threadpool_scheduler

This is a `multiqueue_threadpool_scheduler` with the added capability to steal from other pool's queues within a `composite_threadpool_scheduler` (of course, threads within this pool do steal from each other queues, with higher priority). Not used within a `composite_threadpool_scheduler`, it is a standard `multiqueue_threadpool_scheduler`.

Declaration:

```
template<class Queue,class FindPosition=boost::asynchronous::default_find_posit
class stealing_multiqueue_threadpool_scheduler;
```

Creation if used within a `composite_threadpool_scheduler`:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::stealing_multiqueue_threadpool_schedul
                        boost::asynchronous::threadsafe_list<>>>(4); // 4 thre
```

However, if used stand-alone, which has little interest outside of unit tests, we need to add a template parameter to inform it:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::stealing_multiqueue_threadpool_schedul
                        boost::asynchronous::threadsafe_list<>,boost::asynchro
```

Table 7.6. #include <boost/asynchronous/ stealing_multiqueue_threadpool_scheduler.hpp>

Characteristics	
Number of threads	**1**-n
Can be stolen from?	Yes
Can steal from other threads in this pool?	Yes
Can steal from other threads in other pools?	Yes

composite_threadpool_scheduler

This pool owns no thread by itself. Its job is to contain other pools, accessible by the priority given by posting, and share all queues of its subpools among them. Only the stealing_* pools and `asio_scheduler` will make use of this and steal from other pools though.

For creation we need to create other pool of stealing or not stealing, stolen from or not, schedulers. stealing_xxx pools will try to steal jobs from other pool of the same composite,

but only if these schedulers support this. Other threadpools will not steal but get stolen from. single_thread_scheduler will not steal or get stolen from.

```
// create a composite threadpool made of:
// a multiqueue_threadpool_scheduler, 0 thread
// This scheduler does not steal from other schedulers, but will lend its queue
auto tp = boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::threadpool_scheduler<boost::asynchrono

// a stealing_multiqueue_threadpool_scheduler, 3 threads, each with a threadsafe
// this scheduler will steal from other schedulers if it can. In this case it w
auto tp2 = boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::stealing_multiqueue_threadpool_schedule

// composite pool made of the previous 2
auto tp_worker = boost::asynchronous::make_shared_scheduler_proxy<boost::asynch
```

Declaration:

```
template<class Job = boost::asynchronous::any_callable,
         class FindPosition=boost::asynchronous::default_find_position< >,
         class Clock = std::chrono::high_resolution_clock  >
class composite_threadpool_scheduler;
```

**Table   7.7.   #include   <boost/asynchronous/scheduler/
composite_threadpool_scheduler.hpp>**

Characteristics	
Number of threads	0
Can be stolen from?	Yes
Can steal from other threads in this pool?	N/A
Can steal from other threads in other pools?	No

asio_scheduler

This pool brings the infrastructure and access to io_service for an integrated usage of Boost.Asio. Furthermore, if used withing a composite_threadpool_scheduler, it will steal jobs from other pool's queues.

Declaration:

```
template<class FindPosition=boost::asynchronous::default_find_position< boost::a
class asio_scheduler;
```

Creation:

```
boost::asynchronous::any_shared_scheduler_proxy<> scheduler =
    boost::asynchronous::make_shared_scheduler_proxy<
                    boost::asynchronous::asio_scheduler<>>(4); // 4 threads in p
```

Table 7.8. #include <boost/asynchronous/extensions/asio/asio_scheduler.hpp>

Characteristics	
Number of threads	**1**-n

Characteristics	
Can be stolen from?	No*
Can steal from other threads in this pool?	Yes
Can steal from other threads in other pools?	Yes

Chapter 8. Performance tests

asynchronous::vector

Test: libs/asynchonous/test/perf_vector.cpp.

Test processor Core i7-5960X / Xeon Phi 3120A.

Compiler: g++ 6.1, -O3, -std=c++14, link with libtbbmalloc_proxy.

The test uses a vector of 10000000 elements, each element containing a std::vector of 10 integers.

Table 8.1. Performance of asynchronous::vector members using 4 threads

Member	std::vector	asynchronous::vector	speedup asynchronous / std
Constructor	385ms	147ms	2.62
Copy	337ms	190ms	1.77
compare	81ms	43ms	1.88
clear	221ms	98ms	2.25
resize	394ms	276ms	1.43

Table 8.2. Performance of asynchronous::vector members using 8 threads

Member	std::vector	asynchronous::vector	speedup asynchronous / std
Constructor	380ms	90ms	4.22
Copy	306ms	120ms	2.55
compare	74ms	30ms	2.47
clear	176ms	54ms	3.26
resize	341ms	178ms	1.92

Table 8.3. Performance of asynchronous::vector members Xeon Phi 3120A 57 Cores / 228 Threads

Member	std::vector	asynchronous::vector	speedup asynchronous / std
Constructor	4175 ms	240 ms	17.4
Copy	5439 ms	389 ms	14
compare	4139 ms	43 ms	96
clear	2390 ms	39 ms	61.3
resize	5223 ms	222 ms	23.5

Sort

This test will compare asynchronous:parallel_sort with TBB 4.3 parallel_sort. 16 threads used.

Test: libs/asynchonous/test/perf/parallel_sort_future_v1.cpp. TBB test: libs/asynchonous/test/perf/tbb/tbb_parallel_sort.cpp

Test processor Core i7-5960X.

Compiler: g++ 6.1, -O3, -std=c++14, link with libtbbmalloc_proxy.

The same test will be done using TBB then asynchronous + std::sort, then asynchronous + boost::spreadsort.

Table 8.4. Sorting 200000000 uint32_t

Test	TBB	asynchronous	asynchronous + boost::spreadsort	asynchronous::parallel_sort2	asynchronous::parallel_sort2 + boost::spreadsort
test_random_elements_many_repeated	2416 ms	1133 ms	561 ms	1425 ms	928 ms
test_random_elements_few_repeated	2408 ms	1301 ms	1002 ms	1694 ms	1385 ms
test_random_elements_quite_repeated	2844 ms	1298 ms	1027 ms	1687 ms	1344 ms
test_sorted_elements	22.5 ms	16 ms	16 ms	16 ms	16 ms
test_reversed_sorted_elements	554 ms	47 ms	47 ms	48 ms	48 ms
test_equal_elements	26 ms	16 ms	16 ms	17 ms	17 ms

Table 8.5. Sorting 200000000 double

Test	TBB	asynchronous	asynchronous + boost::spreadsort	asynchronous::parallel_sort2	asynchronous::parallel_sort2 + boost::spreadsort
test_random_elements_many_repeated	2504 ms	1446 ms	1133 ms	2173 ms	1919 ms
test_random_elements_few_repeated	2600 ms	1714 ms	1266 ms	2406 ms	2044 ms
test_random_elements_quite_repeated	2602 ms	1715 ms	1309 ms	2448 ms	2037 ms
test_sorted_elements	34 ms	32 ms	32 ms	32 ms	34 ms
test_reversed_sorted_elements	644 ms	95 ms	94 ms	95 ms	95 ms
test_equal_elements	34 ms	33 ms	32 ms	33 ms	32 ms

Table 8.6. Sorting 200000000 std::string

Test	TBB	asynchronous	asynchronous + boost::spreadsort	asynchronous::parallel_sort2	asynchronous::parallel_sort2 + boost::spreadsort
test_random_elements_many_repeated	880 ms	924 ms	791 ms	889 ms	777 ms
test_random_elements_few_repeated	1033 ms	1069 ms	906 ms	1053 ms	967 ms
test_random_elements_quite_repeated	920 ms	1000 ms	838 ms	998 ms	1003 ms
test_sorted_elements	11 ms	16 ms	16 ms	16 ms	32 ms
test_reversed_sorted_elements	365 ms	28 ms	28 ms	29 ms	38 ms
test_equal_elements	12 ms	4 ms	3 ms	3 ms	4 ms

Table 8.7. Sorting 10000000 objects containing 10 longs

Test	TBB	asynchronous	asynchronous::parallel_sort2
test_random_elements_many_repeated	869 ms	1007 ms	204 ms
test_random_elements_few_repeated	803 ms	887 ms	226 ms

Test	TBB	asynchronous	asynchronous::parallel_sort2
test_random_elements_quite_repeated	810 ms	960 ms	175 ms
test_sorted_elements	22 ms	27 ms	2 ms
test_reversed_sorted_elements	338 ms	34 ms	3 ms
test_equal_elements	25 ms	23 ms	2 ms

parallel_scan

Test: libs/asynchonous/test/perf_scan.cpp.

Test processor Core i7-5960X / Xeon Phi 3120A.

Compiler: g++ 6.1, -O3, -std=c++14, link with libtbbmalloc_proxy.

The test will exercise 10 times a parallel_scan on vector of 1000000 elements.

Table 8.8. Performance of parallel_scan vs serial scan on a i7 / Xeon Phi Knight's Corner

Member	parallel_scan	sequential scan	speedup parallel / serial
i7-5960X (8 Cores / 16 Threads)	204 ms	812 ms	4
i7-5960X (8 Cores / 8 Threads)	305 ms	887 ms	2.9
Xeon Phi (57 Cores /228 Threads)	74 ms	423 ms	57
Xeon Phi (57 Cores /114 Threads)	143 ms	354 ms	25
Xeon Phi (57 Cores /10 Threads)	373 ms	350 ms	9.4

parallel_stable_partition

Test: libs/asynchonous/test/perf_scan.cpp.

Test processor Core i7-5960X / Xeon Phi 3120A.

Compiler: g++ 6.1, -O3, -std=c++11, link with libtbbmalloc_proxy.

The test will exercise a parallel_stable_partition on vector of 100000000 floats. As comparison, std::partition will be used.

Table 8.9. Partitioning 100000000 floats on Core i7-5960X 8 Cores / 8 Threads (16 Threads bring no added value)

Test	parallel	std::partition	speedup parallel / serial
test_random_elements_many_repeated	187 ms	720 ms	3.87
test_random_elements_few_repeated	171 ms	1113 ms	6.5
test_random_elements_quite_repeated	172 ms	555 ms	3.22
test_sorted_elements	176 ms	1139 ms	6.5
test_reversed_sorted_elements	180 ms	1125 ms	6.25

Test	parallel	std::partition	speedup parallel / serial
test_equal_elements	168 ms	1121 ms	6.7

Table 8.10. Partitioning 100000000 floats on Core i7-5960X 4 Cores / 4 Threads

Test	parallel	std::partition	speedup parallel / serial
test_random_elements_many_repeated	296 ms	720 ms	2.43
test_random_elements_few_repeated	301 ms	1113 ms	3.7
test_random_elements_quite_repeated	294 ms	555 ms	1.9
test_sorted_elements	287 ms	1139 ms	4
test_reversed_sorted_elements	288 ms	1125 ms	3.9
test_equal_elements	286 ms	1121 ms	3.9

Table 8.11. Partitioning 100000000 floats on Xeon Phi 3120A 57 Cores / 228 Threads

Test	parallel	std::partition	speedup parallel / serial
test_random_elements_many_repeated	88 ms	15944 ms	181
test_random_elements_few_repeated	80 ms	27186 ms	339
test_random_elements_quite_repeated	89 ms	16067 ms	180
test_sorted_elements	77 ms	26830 ms	348
test_reversed_sorted_elements	73 ms	27367 ms	374
test_equal_elements	82 ms	27464 ms	334

Table 8.12. Partitioning 100000000 floats on Xeon Phi 3120A 57 Cores / 114 Threads

Test	parallel	std::partition	speedup parallel / serial
test_random_elements_many_repeated	152 ms	15944 ms	104
test_random_elements_few_repeated	129 ms	27186 ms	210
test_random_elements_quite_repeated	153 ms	16067 ms	105
test_sorted_elements	104 ms	26830 ms	258
test_reversed_sorted_elements	110 ms	27367 ms	249
test_equal_elements	114 ms	27464 ms	241

Table 8.13. Partitioning 100000000 floats on Xeon Phi 3120A 57 Cores / 10 Threads

Test	parallel	std::partition	speedup parallel / serial
test_random_elements_many_repeated	1131 ms	15944 ms	14
test_random_elements_few_repeated	816 ms	27186 ms	33
test_random_elements_quite_repeated	1212 ms	16067 ms	13
test_sorted_elements	755 ms	26830 ms	35
test_reversed_sorted_elements	739 ms	27367 ms	37
test_equal_elements	798 ms	27464 ms	34

The Xeon Phi speedups are quie surprising. The implementation of std::partition seems inefficient on this platform.

parallel_for

Test: libs/asynchonous/test/parallel_for.cpp.

Test processor Core i7-5960X / Xeon Phi 3120A.

Compiler: g++ 6.1, -O3, -std=c++14, link with libtbbmalloc_proxy.

The test will exercise 10 times a parallel_for with a dummy operation on a vector of 100000000 elements.

Table 8.14. Performance of parallel_for on a i7 / Xeon Phi Knight's Corner

Member	parallel_for	speedup parallel / serial
i7-5960X (8 Cores / 16 Threads)	83 ms	5.9
i7-5960X (8 Cores / 8 Threads)	76 ms	6.4
i7-5960X (8 Cores / 4 Threads)	136 ms	3.6
i7-5960X (8 Cores / 1 Threads)	491 ms	-
Xeon Phi (57 Cores /228 Threads)	51 ms	42
Xeon Phi (57 Cores /114 Threads)	84 ms	25
Xeon Phi (57 Cores /10 Threads)	2124 ms	-

Chapter 9. Compiler, linker, settings

C++ 11

Asynchronous is C++11/14-only. Please check that your compiler has C++11 enabled (-std=c++0x or -std=c++11 in different versions of gcc). Usually, C++14 is recommended.

Supported compilers

Asynchronous is tested and ok with:

- gcc: >= 4.9

- clang: >= 3.5

- VS2015 with a limitation: BOOST_ASYNC_FUTURE/POST_MEMBER_1(or _2 or _3) as variadic macros are not supported

- Intel ICC >= 13.

Supported targets

Asynchronous has been tested on Linux and Windows PCs, Intel and AMD, with the above compilers, and with mingw.

Asynchronous being based on Boost.Thread, can also work on Intel Xeon Phi with a minor change: within Boost, all usage of boost::shared_ptr must be replaced by std::shared_ptr. Strongly recommended is linking with tbbmalloc_proxy for better performance.

Linking

Asynchronous is header-only, but requires Boost libraries which are not. One should link with: boost_system, boost_thread, boost_chrono and boost_date_time if logging is required

Compile-time switches

The following symbols will, when defined, influence the behaviour of the library:

- BOOST_ASYNCHRONOUS_DEFAULT_JOB replaces boost::asynchronous::any_callable by the required job type.

- BOOST_ASYNCHRONOUS_REQUIRE_ALL_ARGUMENTS: forces Asynchronous to only provide servant_proxy macros with all their arguments to avoid accidental forgetting. Precisely:

  - BOOST_ASYNC_FUTURE_MEMBER /BOOST_ASYNC_POST_MEMBER require priority

  - BOOST_ASYNC_FUTURE_MEMBER_LOG / BOOST_ASYNC_POST_MEMBER_LOG require task name and priority

  - make_safe_callback requires name and priority

  - make_lambda_continuation_wrapper requires task name

  - parallel algorithms require task name and priority

  - asynchronous::vector requires as last arguments name and priority

- BOOST_ASYNCHRONOUS_NO_SAVING_CPU_LOAD: overrides default of Asynchronous: schedulers will run at full speed. This can slightly increase speed, at the cost of high CPU load.

- BOOST_ASYNCHRONOUS_PRCTL_SUPPORT: Allows naming of threads if sys/prctl is supported (Linux).

- BOOST_ASYNCHRONOUS_USE_BOOST_SPREADSORT: in older Boost versions, Spreasort was not included. This switch will provide parallel_spreadsort, parallel_quick_spreadsort and parallel_spreadsort_inplace