vollmann engineering gmbh

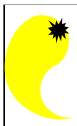
Executors for C++ - A Long Story

Struggling for a Base Concurrency Building Block

Detlef Vollmann

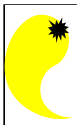
vollmann engineering gmbh, Luzern, Switzerland

Executors for C++, CppCon, September 2015



Part 1

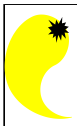
The mission



Motivation: `async`

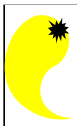
```
std::async([](){ std::cout << "Hello "; });
std::async([](){ std::cout << "World!\n"; });
```

- No concurrency
- No real control over execution agent
  - `launch::async` and `launch::deferred` insufficient



Async Problem

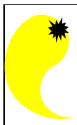
- A `future` destructor normally doesn't block
- But blocks if the `future` is created by `async`
- Lifetime control is important in C++
- For `async` there's no other mechanism to control its lifetime
- Problem is different behaviour of `~fututre()`



Motivation: Pipelines

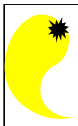
```
pipeline::plan restaurant(
  orders
  | pipeline::parallel(chef, 3)
  | pipeline::parallel(waiter, 4)
  | end);

thread_pool pool;

pipeline::execution work(restaurant.run(&pool));
```

- Executors as building blocks for higher level abstractions



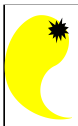
Executor Requirements

- Run tasks
- Control some lifetime aspects



Part 2

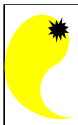
All beginning is ... easy



Original Executor Interface

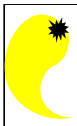
```
class executor{
public:
  virtual ~executor();
  virtual void add(function<void()> closure) = 0;
  virtual size_t
          uninitiated_task_count() const = 0;
};
```

- (Not really the original interface.)



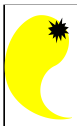
Default Executor

```
shared_ptr<executor> default_executor();
void set_default_executor(
         shared_ptr<executor> executor);
```



Concrete Executors

- `thread_pool`
- `serial_executor`
- `loop_executor`
- `inline_executor`
- `thread_executor`



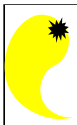
async

```
async(launch::executor,
      [](){ std::cout << "Hello!\n"; });
```

- Uses `default_executor`
  - we need just a little bit more to shutdown the `default_executor`

```
async([](){ std::cout << "Hello!\n"; });
```

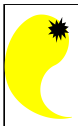
- Could probably also use `default_executor`
  - without breaking any existing code
  - but still blocks on future destructor



async

```
thread_pool myPool;
async(myPool,
      [](){ std::cout << "Hello!\n"; });
```

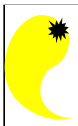
- General way to launch a task on an executor



Motivation: Pipelines

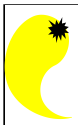
```
pipeline::plan restaurant(
  orders
  | pipeline::parallel(chef, 3)
  | pipeline::parallel(waiter, 4)
  | end);

thread_pool pool;

pipeline::execution work(restaurant.run(&pool));
```

- This can easily be implemented based on the initial proposal



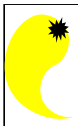
Mission Accomplished

- `async` problem solved
  - Just some more detail work
- Accepted February 2014 by Concurrency SG into Concurrency TS



Part 3

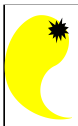
The real discussion begins



Abstract Base Class

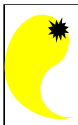
```
virtual void add(function<void()> closure) = 0;
```

- No template concept
- Not part of the type
  - Not really important for functions
  - Important for structures
- Can cross binary interfaces
- Sometimes simply too costly



Part 4

More requirements

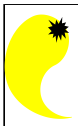


.then

- Proposed continuation `.then` also allows for an executor:

```
auto f = std::async([](){
           std::cout << "Hello "; });
f.then(myPool,
       [](){ std::cout << "World\n"; });
```

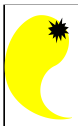
- Without executor, how does `.then` know on which executor to run best?



Data Concentrator

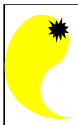
```
RTExecutor rtExec0(0);
RTExecutor rtExec80(80);

ConcentratorT dataConcentrator{
    wrap(rtExec80, ReadDev(1, in1))
    , wrap(rtExec0, ReadDev(2, in2))
    , wrap(rtExec0, StoreData(out))
};
dataConcentrator.run();
```

- Concentrator like pipeline
  - Two producers, one consumer
  - One producer has higher priority



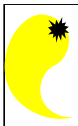
ASIO

- ASynchronous Input/Output
- Wants to run continuation on thread where OS I/O returns
- Wants to run concurrently or co-operative
- Wants to avoid overhead of futures
- Wants to run on user-defined executors
  - with support for system specific asynchronous events
  - signals/interrupts, timers, mailboxes, ...
- Grown out of lot of experience
  - ASIO specific



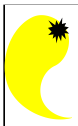
Part 5

New proposals



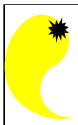
Executors and Async Ops

- ASIO based N4046 by Chris Kohlhoff
- `executor` and `execution_context`
  - `executor` is a light-weight handle
  - `execution_context` actually holds the threads and tasks
  - `execution_context` can be used to wait on everything to shut down.
- Proposed concrete executors:
  - `system_executor` (like `thread_executor`)
  - `strand` (like `serial_executor`
  - `thread_pool`
  - `loop_executor`



Customization Points

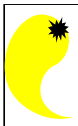
- Continuation token
  - direct continuation on same thread
  - synchronization mechanism
  - concurrency mechanism
- Execution interface
  - `dispatch()`
  - `post()`
  - `defer()`
- `get_associated_executor()`
  - generally required to use
  - allows for arbitrary info from task to executor



Executors and Schedulers, R5

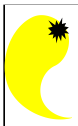
```
class executor{
public:
  template<class Func> void spawn(Func&& func);
};
```

- As template based concept
  - with an interface for type erasing abstract base class
- `task_wrapper` to associate an executor with a task



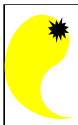
Executor Traits

- N4406 "Integrating Executors with Parallel Algorithm Execution"
- Required interface as traits
- Executor semantics
  - concurrent
  - parallel
  - weakly parallel
- Future type
- Task starting
- Bulk task starting



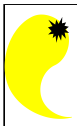
Executor Traits

- N4406 is very specific for parallel algorithms
- Not a proposal for a specific executor interfaces
- Traits allow for implementation that's not provided by the executor
  - bulk interface
  - future based interface



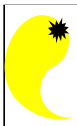
Part 6

Status quo



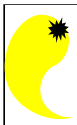
Proposal Status

- Original (modified) Google proposal accepted into Concurrency TS February 2014 (Issaquah)
- ASIO based proposal presented June 2014 in Rapperswil, tentatively accepted as new base:
  - remove N3785 from TS: SF-F-N-A-SA 6-7-5-2-0
  - More work on N4046 for TS: 10-8-0-0-0
  - Apply N4046 to TS without significant changes: 4-2-3-5-2
- R4 of the Google proposal was presented at SG1 meeting September 2014 in Redmond
  - (Re-)Start with Chris Mysen's proposal? SF-F-N-A-SA 9-5-4-0-2
- Traits proposal discussed May 2015 in Lenexa, no vote



Part 7

Rethinking



Layers

User programs

Application

Library components

Containers

`async`

`.then`

ASIO

`FlowGraph`

Parallel Algorithms

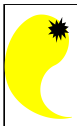
`pipeline`

Event Loop

Building blocks

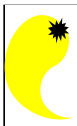
`allocator`

`executor`



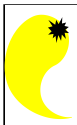
More Info = Better ...

- There's a lot of information about a task that may be useful for an executor implementation
  - relationship to spawning task
  - long/short running
  - blocking/non-blocking
  - repetitions
  - priority
  - information return
  - ...
- All very specific to some executors/domains
- Possibly nothing of them needs to be directly in the executor interface
- But there must exist mechanisms for information transfer
  - only some of them need to be known by intermediate mechanisms



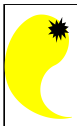
Part 8

Still something else



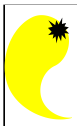
Coroutines

- Coroutines are an important part of asynchronicity
- ASIO works together with coroutines
- The current async/resume approach doesn't seem to mix well with executors
- This needs to change



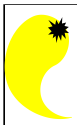
Part 9

Where to go?



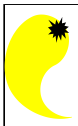
Summary

- Executors are an important building block for concurrency
- They should be low level
- The final interface is still very controversial
- Missing experience



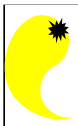
Still Open

- What is a task?
- What information does it carry?
- How?
- What is the right interface for the middle layer?
- What do we need to standardize?



Outlook

- ASIO based proposal very complex
- Provides three customization points
  - blocking hints/requirements, continuation token, associated executor
- Chris Mysens's proposal provides only one of them
  - associated executor
- Coroutines currently not co-operative



Questions

- ??????????????????????????????????