vollmann engineering gmbh
>

Finally Executors for C++

A Base Concurrency Building Block

parallel Heidelberg 2018
March 2018

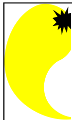
Detlef Vollmann
vollmann engineering gmbh

vollmann engineering gmbh
>

Finally Executors for C++
A Base Concurrency Building Block

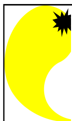
Detlef Vollmann
vollmann engineering gmbh
Luzern, Switzerland

dv@vollmann.ch
http://www.vollmann.ch/



Part 0

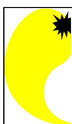
Prelude



Kona Compromise

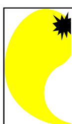
WG21 resolves that for this revision of the C++ standard (aka "C++0x") the scope of concurrency extensions shall be constrained as follows:

- Include a memory model, atomic operations, threads, locks, condition variables, and asynchronous future values.
- Exclude thread pools, task launching, and reader-writer locks.



Part 1

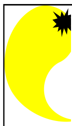
The Mission



Motivation: async

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

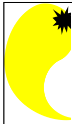
- No concurrency
- No real control over execution agent
  - `launch::async` and `launch::deferred` insufficient



Motivation: Pipelines

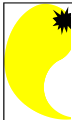
```
pipeline::plan restaurant(
  orders
  | pipeline::parallel(chef, 3)
  | pipeline::parallel(waiter, 4)
  | end);

thread_pool pool;

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

- Executors as building blocks for higher level abstractions



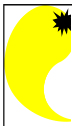
Motivation: Parallel STL

- Parallelism TS provides `std::par` execution policy
  - to run algorithms in parallel
- Requires a mechanism to create parallel execution agents



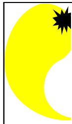
Executor Requirements

- Run tasks
- Control some lifetime aspects



Part 2

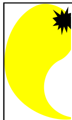
All Beginning is ... Easy



Original Executor Interface

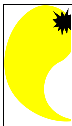
```
class executor{
public:
  virtual ~executor();
  virtual void add(function<void()> closure) = 0;
  virtual size_t
          uninitiated_task_count() const = 0;
};
```

- (Not quite the original interface.)



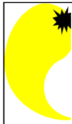
Default Executor

```
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`



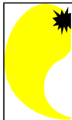
async

```
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"; });
```

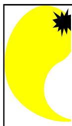
- Could probably also use `default_executor`
  - without breaking any existing code
  - but still blocks on future destructor



async

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

- General way to launch a task on a specific executor

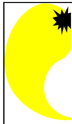


Motivation: Pipelines

```
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

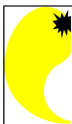
 Copyright ©2010-2018, Detlef Vollmann



Mission Accomplished

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

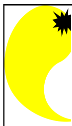
 Copyright ©2010-2018, Detlef Vollmann



Part 3

The Real Discussion Begins

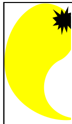
 Copyright ©2010-2018, Detlef Vollmann



Abstract Base Class

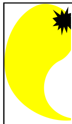
```
virtual void add(function<void()> closure) = 0;
```

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



Part 4

More Requirements

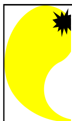


Layers

User programs
Application

Library components
Containers
async
.then
ASIO
FlowGraph
Parallel Algorithms
pipeline
Event Loop

Building blocks
allocator
executor



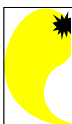
.then

- 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?

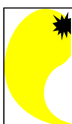
 Copyright ©2010-2018, Detlef Vollmann



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

 Copyright ©2010-2018, Detlef Vollmann

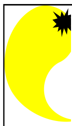


Data Concentrator

```
RTExecutor rtExec(80);

pipeline::plan process{
  wrap(rtExec, input1) + input2
       | validate
       | store};
process.run(pool);
```

- Pipeline as concentrator
  - Two producers, one filter, one consumer
  - One producer has higher priority

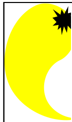
 Copyright ©2010-2018, Detlef Vollmann



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

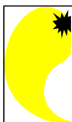
 Copyright ©2010-2018, Detlef Vollmann



`wrap()`

- `wrap` and `get_associated_executor()` from P0113 "Executors and Asynchronous Operations" seems to fit the bill
- It's a static type facility, so type of executor is available
- Independent from executor, so no overhead for executor implementers
  - part of Networking TS

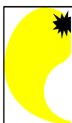
 Copyright ©2010-2018, Detlef Vollmann



Part 5

New Proposals

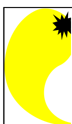
 Copyright ©2010-2018, Detlef Vollmann



Executors and Async Ops

- ASIO based P0113 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` (fixed size)
  - `loop_executor`

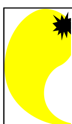
 Copyright ©2010-2018, Detlef Vollmann



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

 Copyright ©2010-2018, Detlef Vollmann

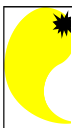


Executors (R6)

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

- As template based concept
  - with an interface for type erasing abstract base class
- P0008

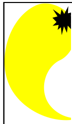
 Copyright ©2010-2018, Detlef Vollmann



Executor Traits

- "An Interface for Abstracting Execution" (P0058)
- Required interface as traits
- Executor semantics
  - concurrent
  - parallel
  - weakly parallel
- Future type
- Task starting
- Bulk task starting

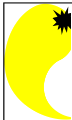
 Copyright ©2010-2018, Detlef Vollmann



Executor Traits

- P0058 is very specific for parallel algorithms
- Not a proposal for a specific executor interface
- Traits allow for implementation that's not provided by the executor
  - bulk interface
  - future based interface

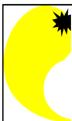
 Copyright ©2010-2018, Detlef Vollmann



Part 6

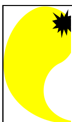
Status Quo 2016

 Copyright ©2010-2018, Detlef Vollmann



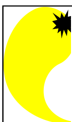
Proposal Status

- 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
- ASIO based proposal part of Networking TS
- Traits (P0058) proposal discussed several times, no vote
- ASIO customization points (P0285) not discussed yet



Part 7

Rethinking



Layers

User programs

Application

Library components

Containers

`async`

`.then`

ASIO

`FlowGraph`

Parallel Algorithms

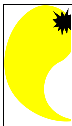
`pipeline`

Event Loop

Building blocks

`allocator`

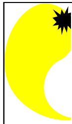
`executor`



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

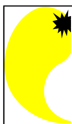
 Copyright ©2010-2018, Detlef Vollmann



Part 8

Still Something Else

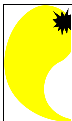
 Copyright ©2010-2018, Detlef Vollmann



Asynchronicity

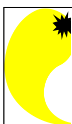
- Blocking wastes resources
- No blocking waits for external events
  - I/O, network, signals, timer, ...
- Asynchronous calls means concurrency
  - sometimes preemptive
- No standard support for asynchronous functions yet
  - Boost ASIO pre-standardized as networking TS
  - resumable functions, `.then`, coroutines, ...

 Copyright ©2010-2018, Detlef Vollmann



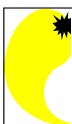
Coroutines

- Coroutines are an important part of asynchronicity
- ASIO works together with coroutines
  - with explicit interface
- The coroutine await/yield approach doesn't seem to mix well with executors



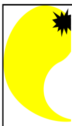
ASIO Without Coroutines

```
void start() { // start async read;
  socket.async_read_some(net::buffer(data),
    [] (size_t length) { handleRead(length); });
}

void handleRead(size_t length) {
  // start async write
  net::async_write(socket,
    net::buffer(data),
    [] () { handleWrite(); });
}

void handleWrite() { // start async read
  socket.async_read_some(net::buffer(data),
    [] (size_t length) { handleRead(length); });
}
```

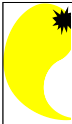
ASIO With Coroutines

```
awaitable<void> echo(tcp::socket socket
                     , await_context ctx) {
  size_t length;
  char data[128];
  while (true) {
    length = co_await socket.async_read_some(
                        net::buffer(data), ctx);
    co_await async_write(socket
                 , net::buffer(data, length)
                 , ctx);
  }
}
```



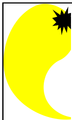
Part 9

Restart



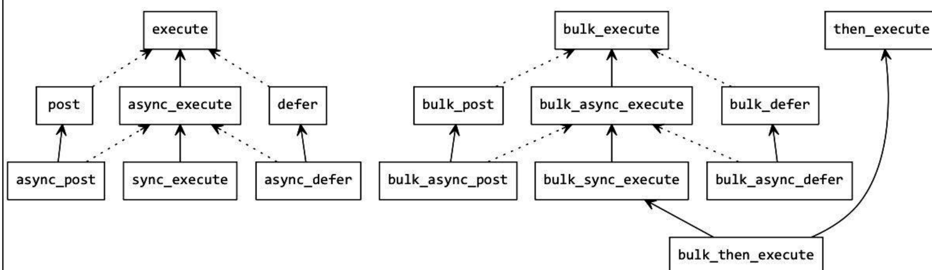
Joint Proposal

- "A Unified Executors Proposal for C++" (P0443 R0 for Issaquah Nov 2016)
- With authors from all previous proposals
  - with 16 different execution functions
- "A Proposal to Simplify the Unified Executors Design" (P0688 R0 for Toronto Jun 2017)
- After more work accepted by SG1 in Albuquerque Nov 2017
  - heavy discussions in LEWG
- Still open issues

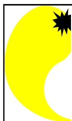


New Executor Interface

- Execution functions from P0443 R1 (Feb 2017)



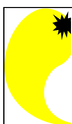
- Too many for SG1



New Executor Interface

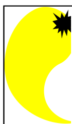
- Execution functions from P0443 R5 (Mar 2018)

```
void execute(F);
Future twoway_execute(F);
Future then_execute(F, Future);
void bulk_execute(F, size_t, PF);
Future bulk_twoway_execute(F, sixe_t, RF, PF);
Future bulk_then_execute(F, size_t, Future,
                         RF, PF);
```

Executor Properties

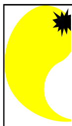
- Executors have properties
  - depending on the available syntactic interface
  - depending on the semantics of the interface
  - to provide extra information
- Direction
  - `oneway`, `twoway`, `then`
- Cardinality
  - `single`, `bulk`
- Blocking
  - `never_blocking`, `possibly_blocking`, `always_blocking`
- Continuation, more work, progress, new thread, allocator
- User defined properties are possible



`require/prefer`

- `require()` and `prefer()` to get specific properties

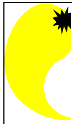
```
auto newExec1 = require(oldExec,
                        oneway,
                        single,
                        never_blocking);

auto newExec2 = prefer(newExec1,
                       outstanding_work);
```



One-Way Executions

- Two-way execution functions return a `future`
  - possibly not `std::future`
- One-way execution functions don't return any handle
  - this is still being discussed

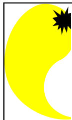
 Copyright ©2010-2018, Detlef Vollmann



Part 10

Finally a Base

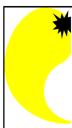
 Copyright ©2010-2018, Detlef Vollmann



Demo

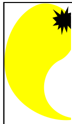
- Some real code

 Copyright ©2010-2018, Detlef Vollmann



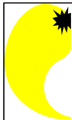
Presentation Downloads

- The slides and source code will be available at http://www.vollmann.ch/de/presentations/



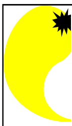
References

- Joint proposal:
  P0443R5, Jared Hoberock, Michael Garland,
  Chris Kohlhoff, Chris Mysen,
  Carter Edwards, Gordon Brown
  "A Unified Executors Proposal for C++"
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0443r5.html
- Implementation and latest version of proposal
  git://github.com/executors/issaquah_2016.git



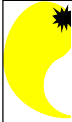
References

- P0008R0, Chris Mysen,
  "C++ Executors"
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0008r0.pdf
- "C++ extensions for Networking (N4588)"
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/n4588.pdf



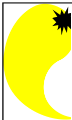
References

- P0058R1, Jared Hoberock, Michael Garland, Olivier Giroux
  "An Interface for Abstracting Execution"
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0058r1.pdf
- P0285R0, Christopher Kohlhoff
  "Using customization points to unify executors"
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0285r0.html



References

- P0113R0, Christopher Kohlhoff
  "Executors and Asynchronous Operations"
  http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2015/p0113r0.html



Questions

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