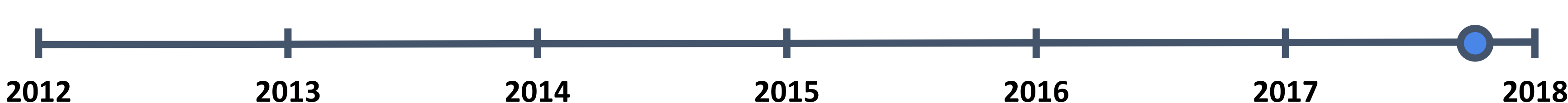
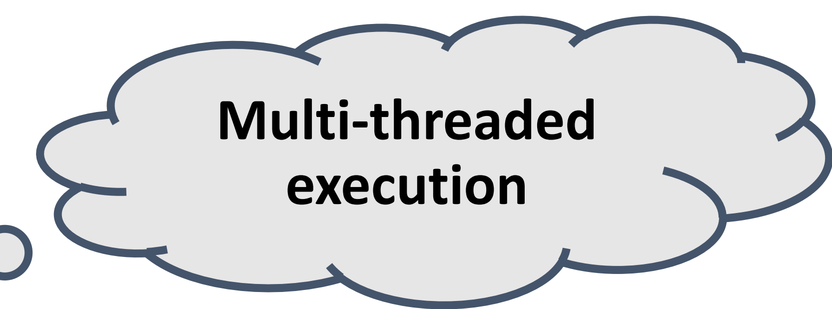
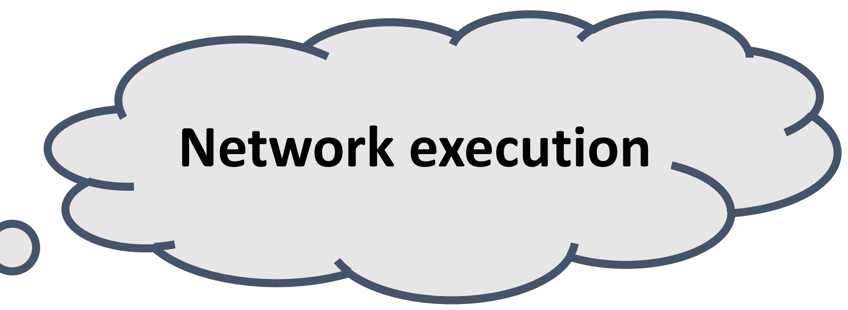
- With the feedback from SG1 the proposal is close to forming a technical specification for executors

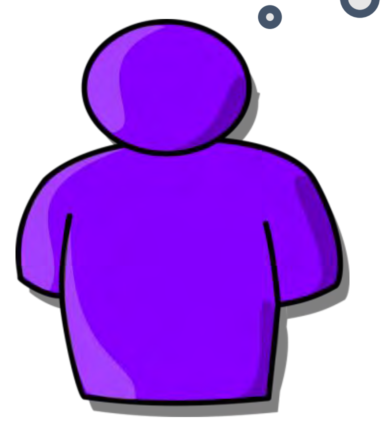


What is execution?

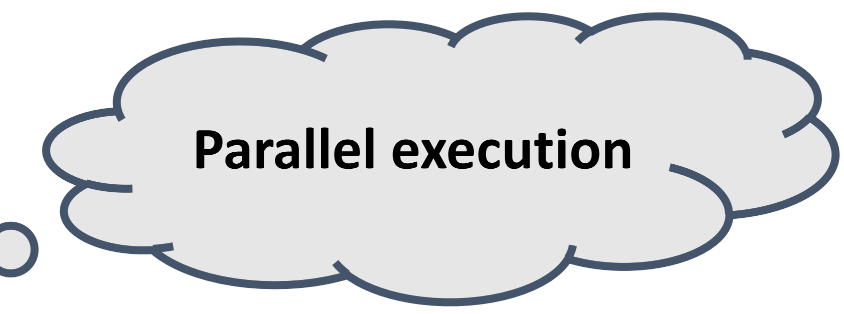


- Thread pools (fixed sized, dynamic)
- std::async()
- Launch policies
- Work that can execution as if on a std::thread

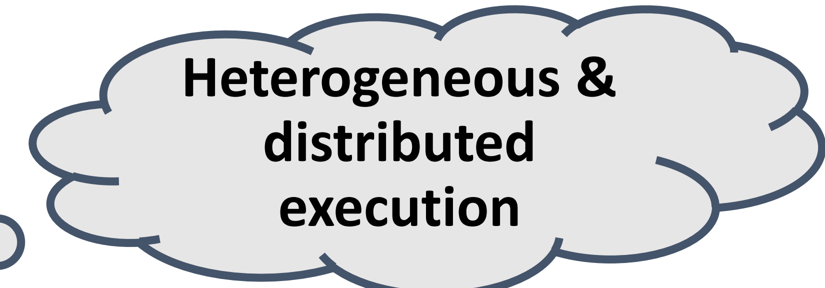




- Network devices
- Boost.asio / networking TS
- One way communication
- Work tracking



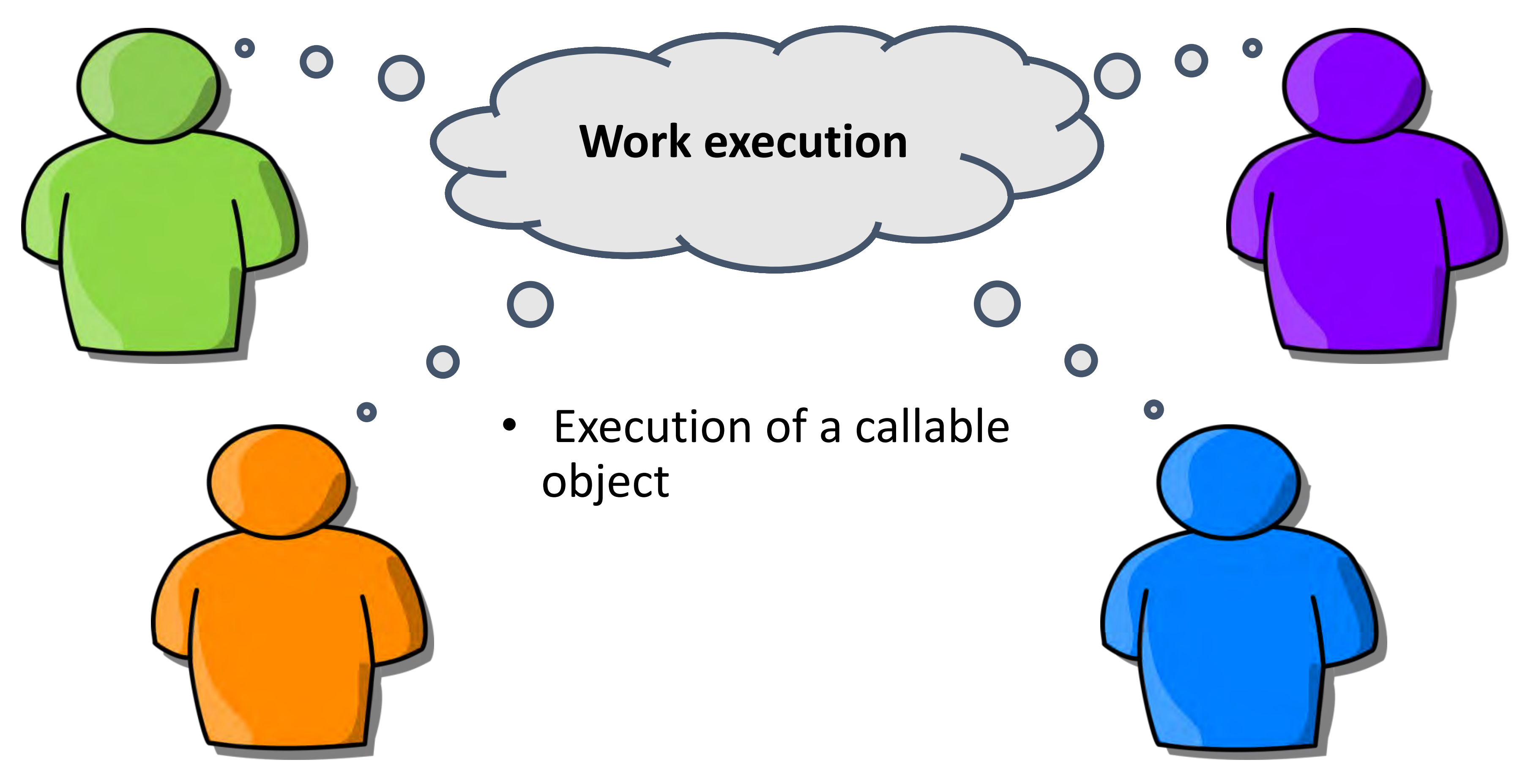
- Parallel / vectorized algorithms
- Execution policies
- Bulk execution of threads
- Channel for returning a result

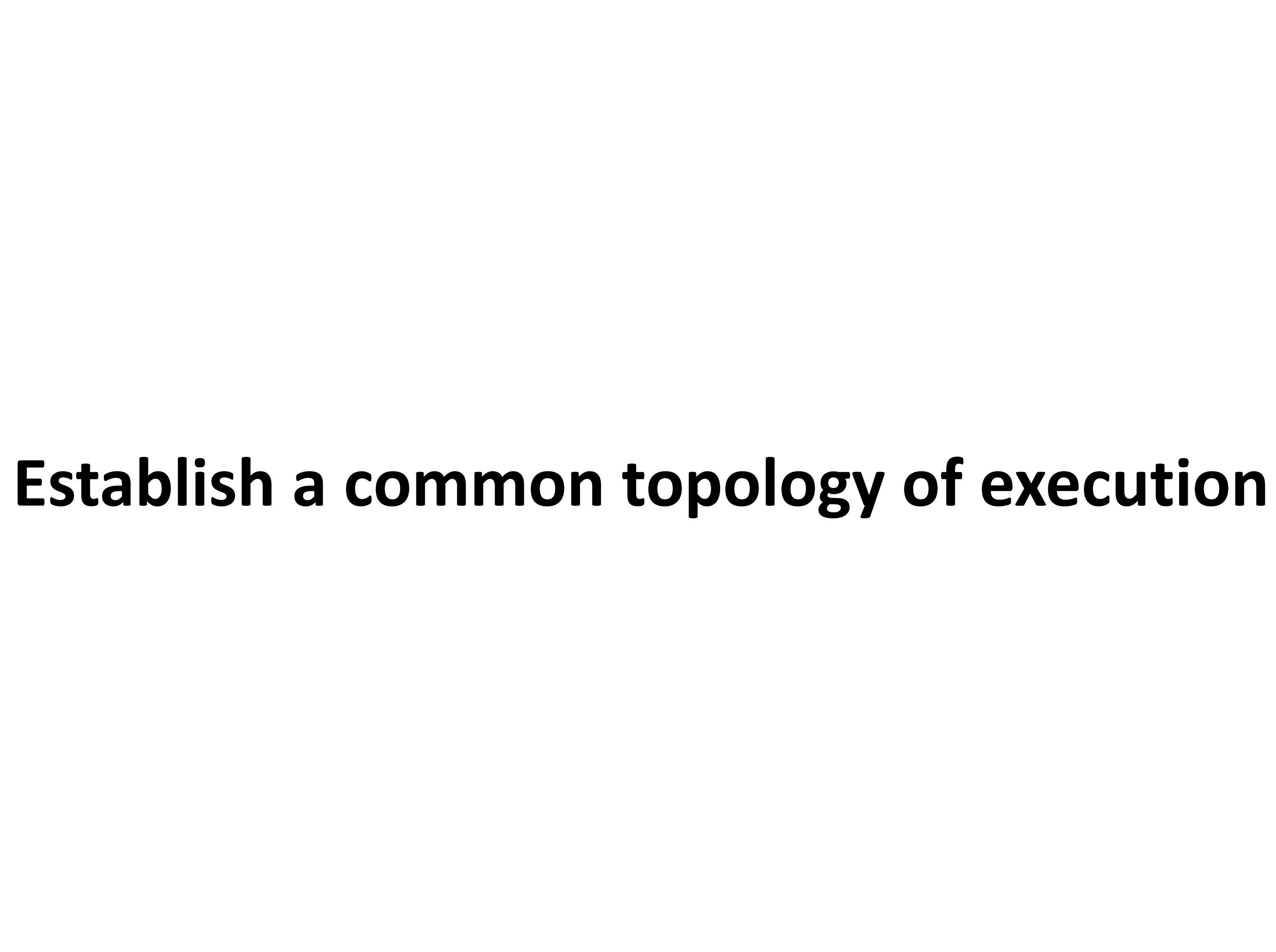


- Managed execution contexts
- Discrete non-CPU architectures
- Task graphs
- Bulk execution of threads

How do we make sure everyone gets what they want?

What do these all have in Common?


Work execution
• Execution of a callable object

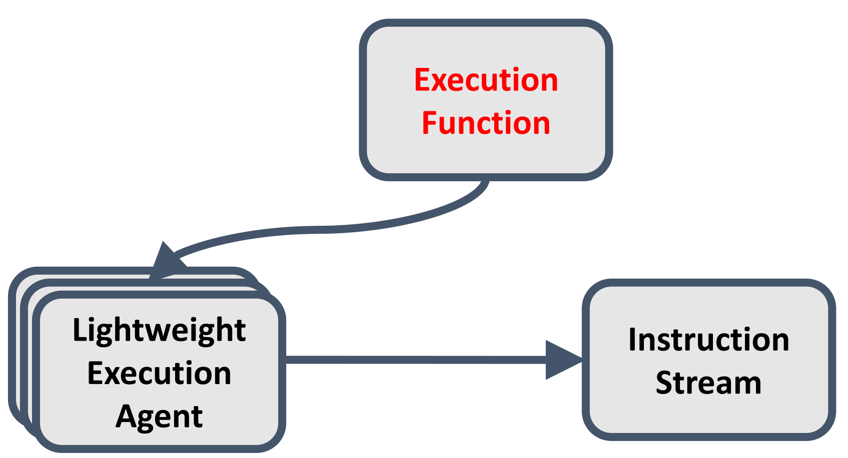

Establish a common topology of execution

- An instruction stream is a callable object that is to be executed

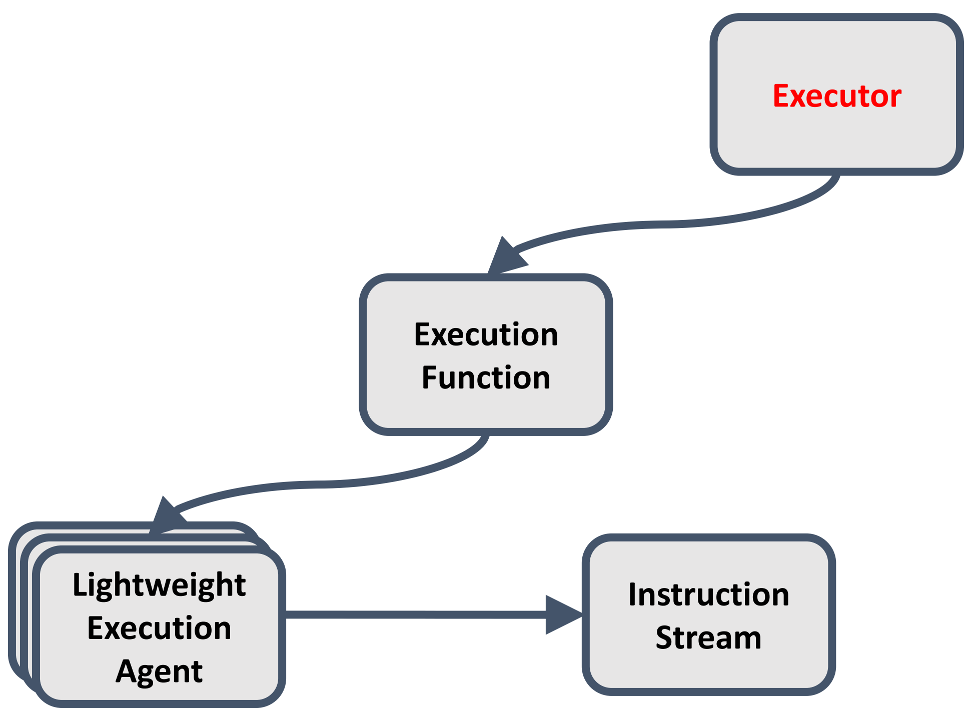
Instruction Stream

- A light-weight execution agent is a single thread of execution executing the instruction stream

- An execution function is a function which executes an instruction stream on one or more light-weight execution agents with a particular set of properties

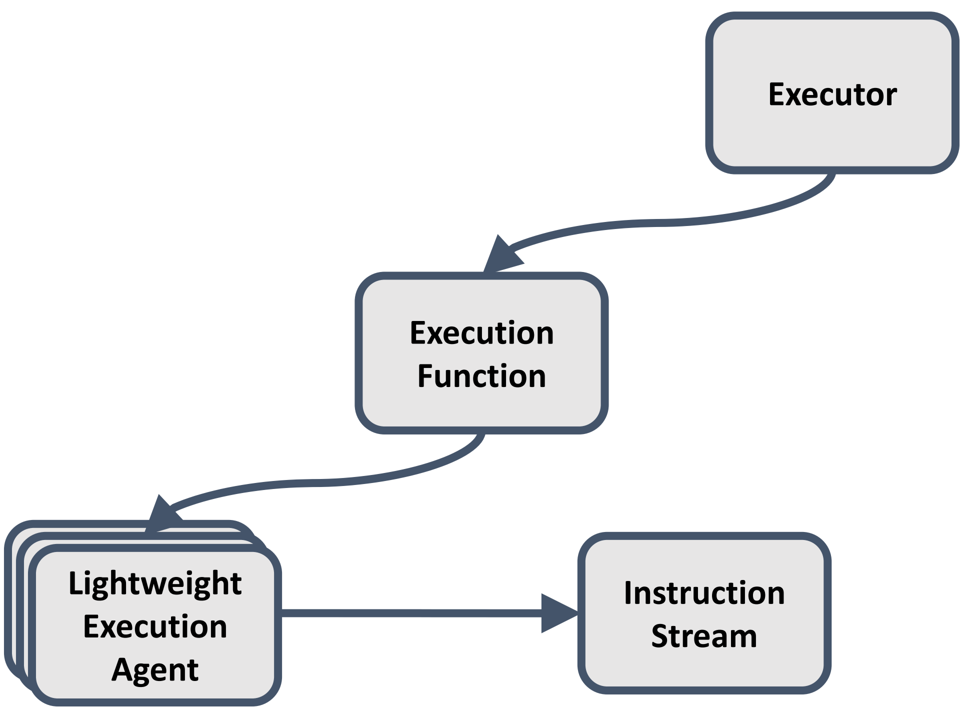


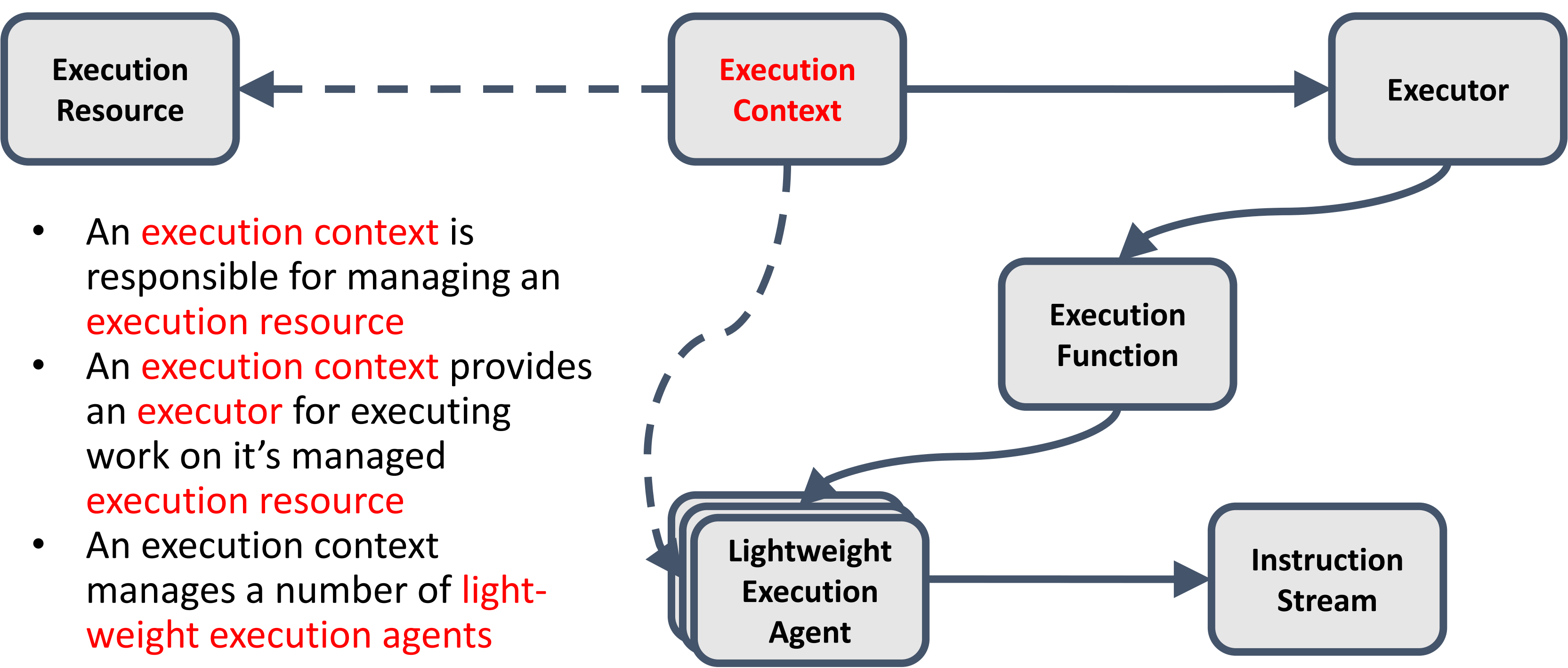
- An executor is an interface that describes where, when and how to execute work
- An executor can spawn one or more light-weight execution agents each executing the same instruction stream via execution functions



Execution Resource

- An execution resource is the hardware abstraction which is executing the work
- Examples of an execution resource are a CPU thread pool, GPU context, network device





- An execution context is responsible for managing an execution resource
- An execution context provides an executor for executing work on it's managed execution resource
- An execution context manages a number of light-weight execution agents

```
{
    execution_context execContext;
    auto exec = execContext.executor();
    exec.execute([&](){ func(); });
}
```

```
{
    execution_context execContext;
    auto exec = execContext.executor();
    exec.execute([&](){ func(); });
}
```

```
{
    execution_context execContext;
    auto exec = execContext.executor();
    exec.execute([&](){ func(); });
}
```

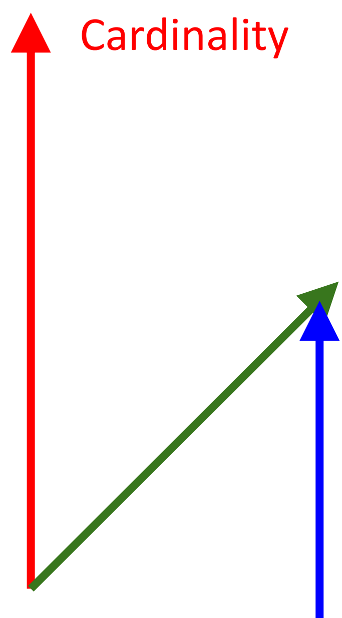
```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

```
{
  execution_context execContext;
  auto exec = execContext.executor();
  exec.execute([&](){ func(); });
}
```

Establish the bifurcations of execution

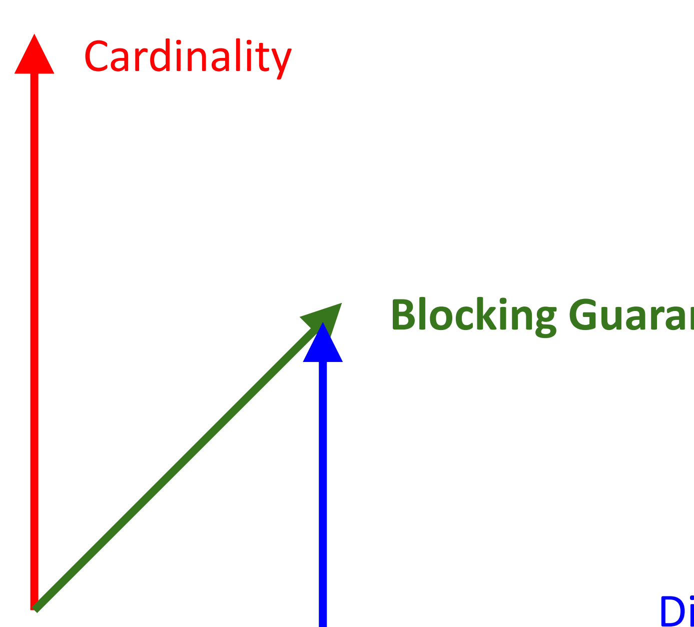
Cardinality

- An executor's cardinality reflects whether an execution launches a single thread of execution or multiple threads of execution
    - Single cardinality
    - Bulk cardinality



Directionality

- An executor's directionality reflects whether an execution does or does not provides a channel by which to synchronise or return a result or exception
  - One-way directionality
  - Two-way directionality



- An executor's blocking guarantee reflects whether an execution blocks or does not block the caller thread until execution is complete
  - Possibly-blocking guarantee
  - Always-blocking guarantee
  - Never-blocking guarantee

Establish the execution functions

	One-way	Two-way
Single	`execute()`	`twoway_execute()`
Bulk	`bulk_execute()`	`bulk_twoway_execute()`

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  twoway_executor exec;
  auto fut =
exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  twoway_executor exec;
  auto fut =
exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  bulk_executor exec;
  exec.bulk_execute([&](size_t index,
    auto &s){
    func(i, s);
  }, shape, sharedFactory);
}
```

Bulk One-way

```
{
  oneway_executor exec;
  exec.execute([&]() {
    func();
  });
}
```

Single One-way

```
{
  twoway_executor exec;
  auto fut =
exec.twoway_execute([&](){
    return func();
  });
}
```

Single Two-way

```
{
  bulk_executor exec;
  exec.bulk_execute([&](size_t index,
    auto &s){
    func(i, s);
  }, shape, sharedFactory);
}
```

Bulk One-way

```
{
  bulk_twoway_executor exec;
  auto fut =
exec.bulk_twoway_execute(
    [&](size_t index, auto &r, auto
&s){
    func(i, r, s);
  }, shape, resultFactory,
sharedFactory);
}
```

Bulk Two-way

Establish the properties of execution

Property	Description
single	Executes an instruction stream exactly once
bulk	Executes an instruction stream a number of times
oneway	Does not return a future
twoway	Returns a future for the return value or exception and synchronisation
possibly_blocking	May or may not block the caller thread on execution completion
always_blocking	Always blocks the caller thread on execution completion
never_blocking	Never blocks the caller thread on execution completion

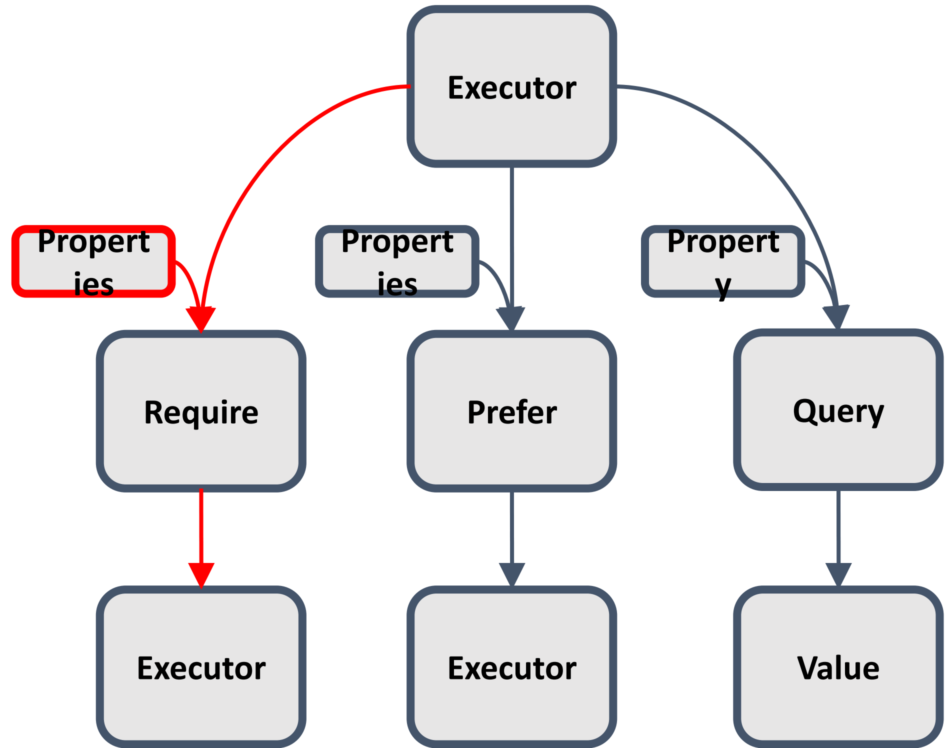
Property	Description
single	Executes an instruction stream exactly once
bulk	Executes an instruction stream a number of times
oneway	Does not return a future
twoway	Returns a future for the return value or exception and synchronisation
possibly_blocking	May or may not block the caller thread on execution completion
always_blocking	Always blocks the caller thread on execution completion
never_blocking	Never blocks the caller thread on execution completion

Properties which modify the interface

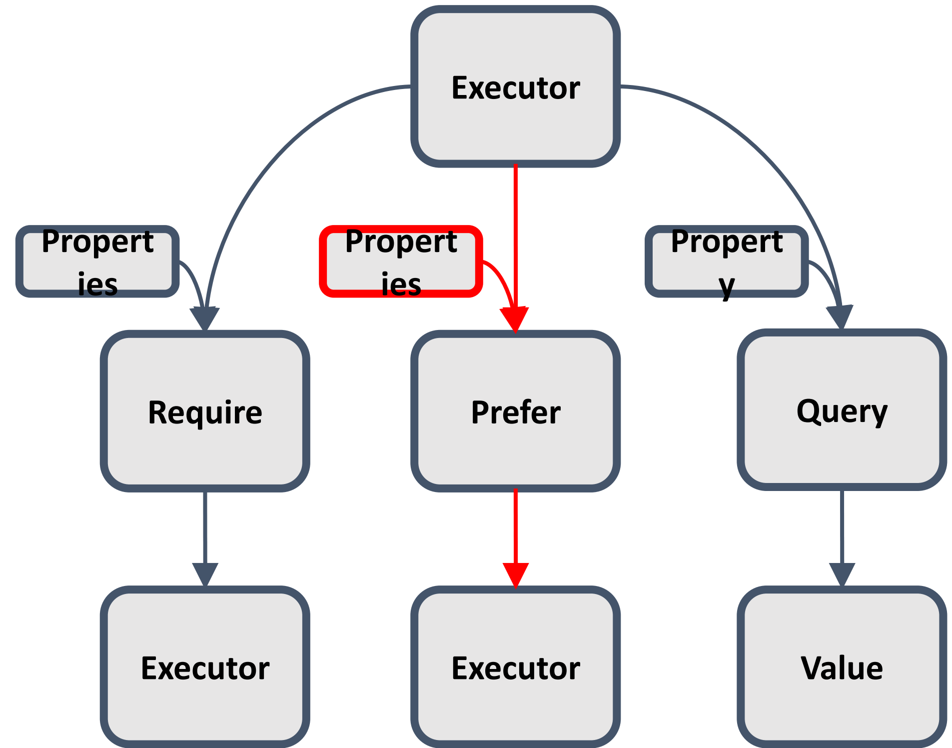
Properties	Description
Thread mapping semantics	Specifies the way in which the instruction stream is mapped to threads of execution
Bulk execution guarantees	Specifies the guarantees between threads of execution within a bulk execution
Caller forward progress guarantees	Specifies the forward progress guarantees between the threads of execution and the caller thread
Continuation	Specifies whether the instruction stream should be executed as a continuation
Future work submission	Specifies whether or not the execution context should expect future work to be submitted
Allocator	Specifies the allocator to use when allocating memory for the instruction stream

Establish how executors could be customised

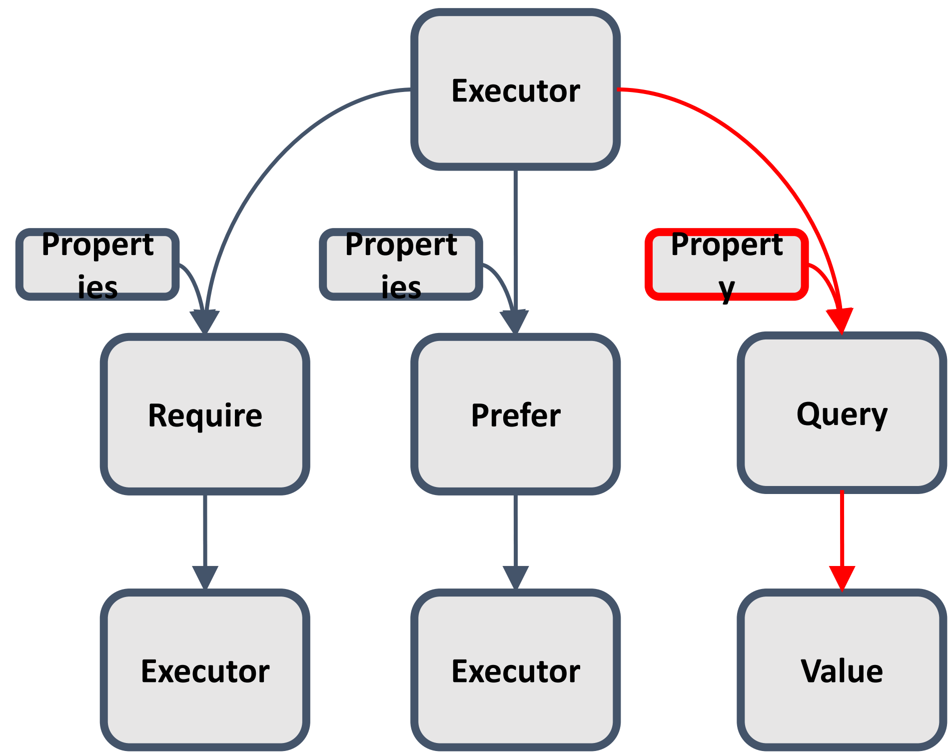
- Performing a require returns an executor that will have the requested properties
  - If the properties are already supported the original executor is returned
  - If the properties are not supported this will result in a compile-time error



- Performing a prefer returns an executor that may have the requested properties
  - If the properties are already supported the same executor is returned
  - If the properties are not supported the executor will simply return the original executor



- Performing a query returns the current value of a specific property
  - In many cases this value will be a boolean type



```
oneway_executor exec;

auto newExec = require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
oneway_executor exec;

auto newExec = require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
possibly_blocking_executor exec;

auto newExec = prefer(exec, never_blocking);

newExec.execute([&]() {
  func();
});
```

Prefer

```
oneway_executor exec;

auto newExec = require(exec, twoway);

auto fut = newExec.twoway_execute([&]() {
  return func();
});
```

Require

```
possibly_blocking_executor exec;

auto newExec = prefer(exec, never_blocking);

newExec.execute([&]() {
  func();
});
```

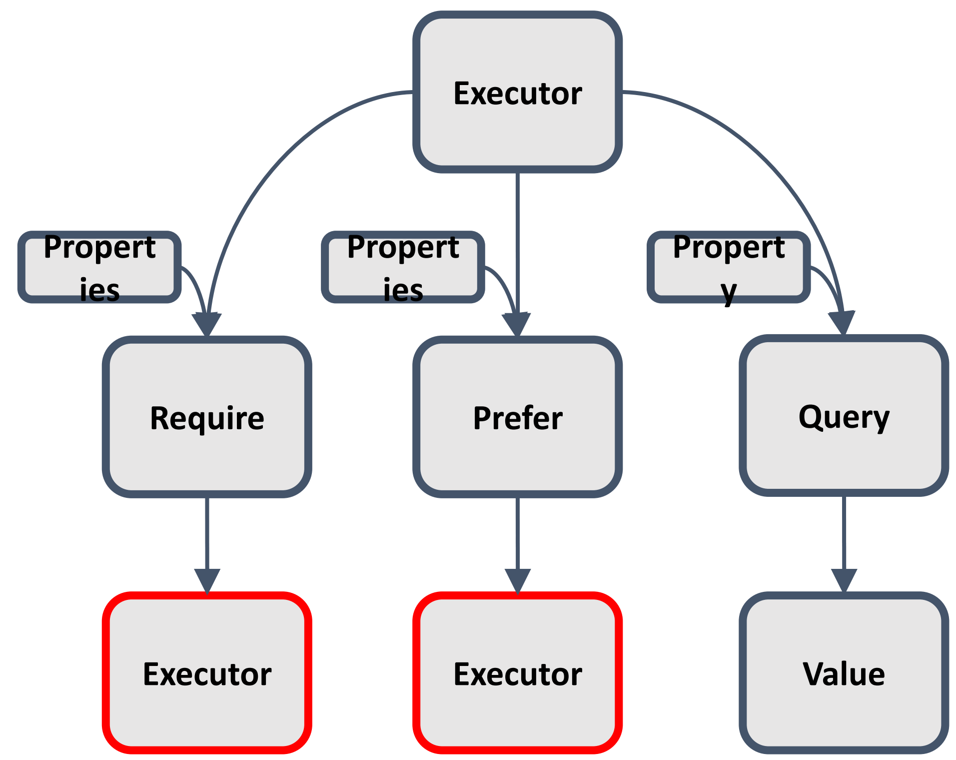
Prefer

```
possibly_blocking_executor exec;

auto newExec = prefer(exec, never_blocking);

auto isNeverBlocking = query(newExec, never_blocking);
```

Query

Polymorphic Executor

- The executor returned by require and prefer can be:
    - A static typed executor such as one_way_executor.
    - A dynamically typed executor wrapped in the polymorphic executor.



```
class my_scheduler {

  twoway_executor exe

};

{

  my_scheduler myScheduler;

  auto newExec = prefer(myScheduler.exec, never_blocking);

  auto fut = newExec.twoway_execute([&](){
    return func();
  });

}
```

Statically typed executors means executors cannot be stored generically

Statically typed executors means result of prefer must be known at compile time

```
class my_scheduler {

  executor exec;

};

{

  my_scheduler myScheduler;

  myScheduler.exec = prefer(myScheduler.exec, never_blocking);

  auto fut = myScheduler.exec.twoway_execute([&](){
    return func();
  });

}
```

Implementing an executor

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {









  }
};
```

Naive Implementation

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

  }
};
```

Naive Implementation

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread newThread([=](){






    });


  }
};
```

Naive Implementation

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread newThread([=](){
      try {


      } catch (...) {

      }
    });


  }
};
```

Naive Implementation

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread newThread([=](){
      try {
        auto result = func();
        promise.set_value(result);
      } catch (...) {
        promise.set_exception(std::current_exception());
      }
    });


  }
};
```

Naive Implementation

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    std::thread newThread([=](){
      try {
        auto result = func();
        promise.set_value(result);
      } catch (...) {
        promise.set_exception(std::current_exception());
      }
    }).detach();

    return fut;
  }
};
```

Naive Implementation

```
class my_executor {

  template <typename Function>
  std::future<std::invoke_result_t<std::decay_t<Function>>>
  twoway_execute(Function &&func) {

    using return_type = invoke_result_t<std::decay_t<Function>>;
    std::promise<return_type> promise;
    auto fut = promise.get_future();

    this->spawn_thread([=](){
      try {
        auto result = func();
        promise.set_value(result);
      } catch (...) {
        promise.set_exception(std::current_exception());
      }
    }).detach();

    return fut;
  }
};
```

Naive Implementation

Implementing std::async()

```
template <typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Function &&func, Args &&...args) {






}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {




}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

  auto fut = requiredExec.twoway_execute([&](){

  });


}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

  auto fut = requiredExec.twoway_execute([&](){
    return func(std::forward<Args>(args)...);
  });


}
```

```
template <typename Executor, typename Function, typename… Args>
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args...>>>
async(Executor exec, Function &&func, Args &&...args) {

  auto requiredExec = require(exec, single, twoway, never_blocking);

  auto fut = requiredExec.twoway_execute([&](){
    return func(std::forward<Args>(args)...);
  });

  return fut;
}
```

Using an executor

```
{

  int input = 10;

  auto fut = std::async([=](int m) {
    int factorial = 1;
    for (int i = 1; i <= m; ++i) {
      factorial *= i;
    }
    return factorial;
  }, input);

  auto result = fut.get();

}
```

```
{

  twoway_executor exec;
  int input = 10;

  auto fut = std::async(exec, [=](int m) {
    int factorial = 1;
    for (int i = 1; i <= m; ++i) {
      factorial *= i;
    }
    return factorial;
  }, input);

  auto result = fut.get();

}
```

Using a different executor

```
{

  gpu_executor exec;
  int input = 10;

  auto fut = std::async(exec, [=](int m) {
    int factorial = 1;
    for (int i = 1; i <= m; ++i) {
      factorial *= i;
    }
    return factorial;
  }, input);

  auto result = fut.get();

}
```

Use the Proper Abstraction

- Cores
- HW Threads
- Vectors
- Offload
- Heterogeneous
- Cloud
- Caches
- NUMA

- Tasks, C++11/14/14
- Tasks, C++11/14/17
- SIMD, Parallelism TS2
- OpenCL or SYCL
- OpenCL or SYCL
- OpenCL or SYCL
- Context, executors
- Context, executors

If you have to remember 2 things

- Expose more parallelism
- Increase Locality of reference

Codeplay

Standards bodies	Research	Open source	Presentations	Company
• HSA Foundation: Chair of software group, spec editor of runtime and debugging • Khronos: chair & spec editor of SYCL. Contributors to OpenCL, Safety Critical, Vulkan • ISO C++: Chair of Low Latency, Embedded WG; Editor of SG1 Concurrency TS • EEMBC: members	• Members of EU research consortiums: PEPPHER, LPGPU, LPGPU2, CARP • Sponsorship of PhDs and EngDs for heterogeneous programming: HSA, FPGAs, ray-tracing • Collaborations with academics • Members of HiPEAC	• HSA LLDB Debugger • SPIR-V tools • RenderScript debugger in AOSP • LLDB for Qualcomm Hexagon • TensorFlow for OpenCL • C++ 17 Parallel STL for SYCL • VisionCpp: C++ performance-portable programming model for vision	• Building an LLVM back-end • Creating an SPMD Vectorizer for OpenCL with LLVM • Challenges of Mixed-Width Vector Code Gen & Scheduling in LLVM • C++ on Accelerators: Supporting Single-Source SYCL and HSA • LLDB Tutorial: Adding debugger support for your target	• Based in Edinburgh, Scotland • 57 staff, mostly engineering • License and customize technologies for semiconductor companies • ComputeAorta and ComputeCpp: implementations of OpenCL, Vulkan and SYCL • 15+ years of experience in heterogeneous systems tools

2001 - 2003

VectorC for x86
Our VectorC technology was chosen and actively used for Computer Vision

First showing of VectorC{VU}

Delivered VectorC{VU} to the National Center for Supercomputing

VectorC{EE} released
An optimising C/C++ compiler for PlayStation®2 Emotion Engine (MIPS)

2005 - 2006

Ageia chooses Codeplay for PhysX
Codeplay is chosen by Ageia to provide a compiler for the PhysX processor.

Codeplay joins the Khronos Group

2007 - 2011

Sieve C++ Programming System released
Aimed at helping developers to parallelise C++ code, evaluated by numerous researchers

Offload released for Sony PlayStation®3

OffloadCL technology developed

Codeplay joins the PEPPHER project

2013

New R&D Division
Codeplay forms a new R&D division to develop innovative new standards and products

Becomes specification editor of the SYCL standard

2014

LLDB Machine Interface Driver released

Codeplay joins the CARP project

Codeplay shows technology to accelerate Renderscript on OpenCL using SPIR

2015

Chair of HSA System Runtime working group

Development of tools supporting the Vulkan API

2016

Open-Source HSA Debugger release

Releases partial OpenCL support (via SYCL) for Eigen Tensors to power TensorFlow

ComputeAorta 1.0 release

ComputeCpp Community Edition beta release
First public edition of Codeplay's SYCL technology

Codeplay build the software platforms that deliver massive performance

What our ComputeCpp users say about us

Benoit Steiner – Google TensorFlow engineer

"We at Google have been working closely with Luke and his Codeplay colleagues on this project for almost 12 months now. Codeplay's contribution to this effort has been tremendous, so we felt that we should let them take the lead when it comes down to communicating updates related to OpenCL. … we are planning to merge the work that has been done so far… we want to put together a comprehensive test infrastructure"

ONERA

"We work with royalty-free SYCL because it is hardware vendor agnostic, single-source C++ programming model without platform specific keywords. This will allow us to easily work with any heterogeneous processor solutions using OpenCL to develop our complex algorithms and ensure future compatibility"

Hartmut Kaiser -HPX

"My team and I are working with Codeplay's ComputeCpp for almost a year now and they have resolved every issue in a timely manner, while demonstrating that this technology can work with the most complex C++ template code. I am happy to say that the combination of Codeplay's SYCL implementation with our HPX runtime system has turned out to be a very capable basis for Building a Heterogeneous Computing Model for the C++ Standard using high-level abstractions."

WIGNER Research Centre for Physics

It was a great pleasure this week for us, that Codeplay released the ComputeCpp project for the wider audience. We've been waiting for this moment and keeping our colleagues and students in constant rally and excitement. We'd like to build on this opportunity to increase the awareness of this technology by providing sample codes and talks to potential users. We're going to give a lecture series on modern scientific programming providing field specific examples."

Further information

- OpenCL https://www.khronos.org/opencl/
- OpenVX https://www.khronos.org/openvx/
- HSA http://www.hsafoundation.com/
- SYCL http://sycl.tech
- OpenCV http://opencv.org/
- Halide http://halide-lang.org/
- VisionCpp https://github.com/codeplaysoftware/visioncpp

Community Edition

Available now for free!

Visit:

computecpp.codeplay.com