Core C++ 2024

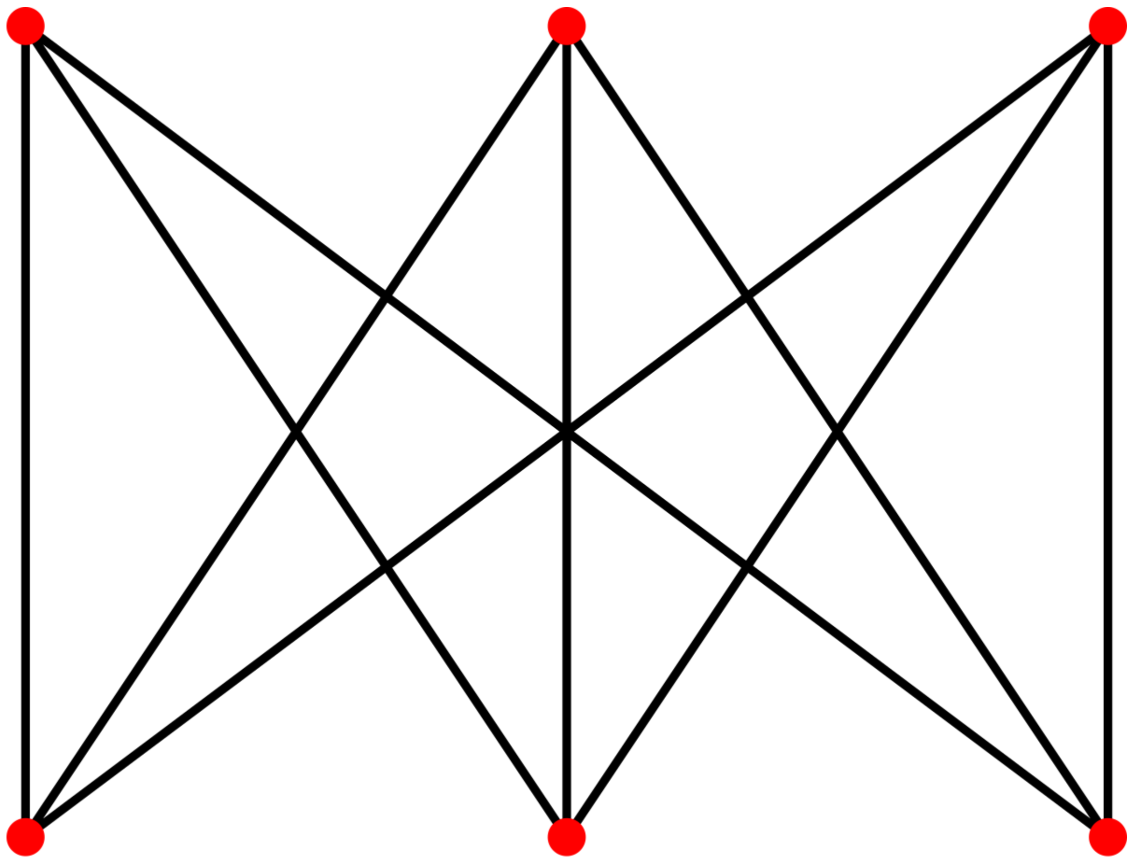
The Pains and Joys of C++ In-Process Graph Execution

Svyatoslav Feldsherov

Hi, I am Slava!

Graphs are everywhere!

Graph is dots connected by lines.



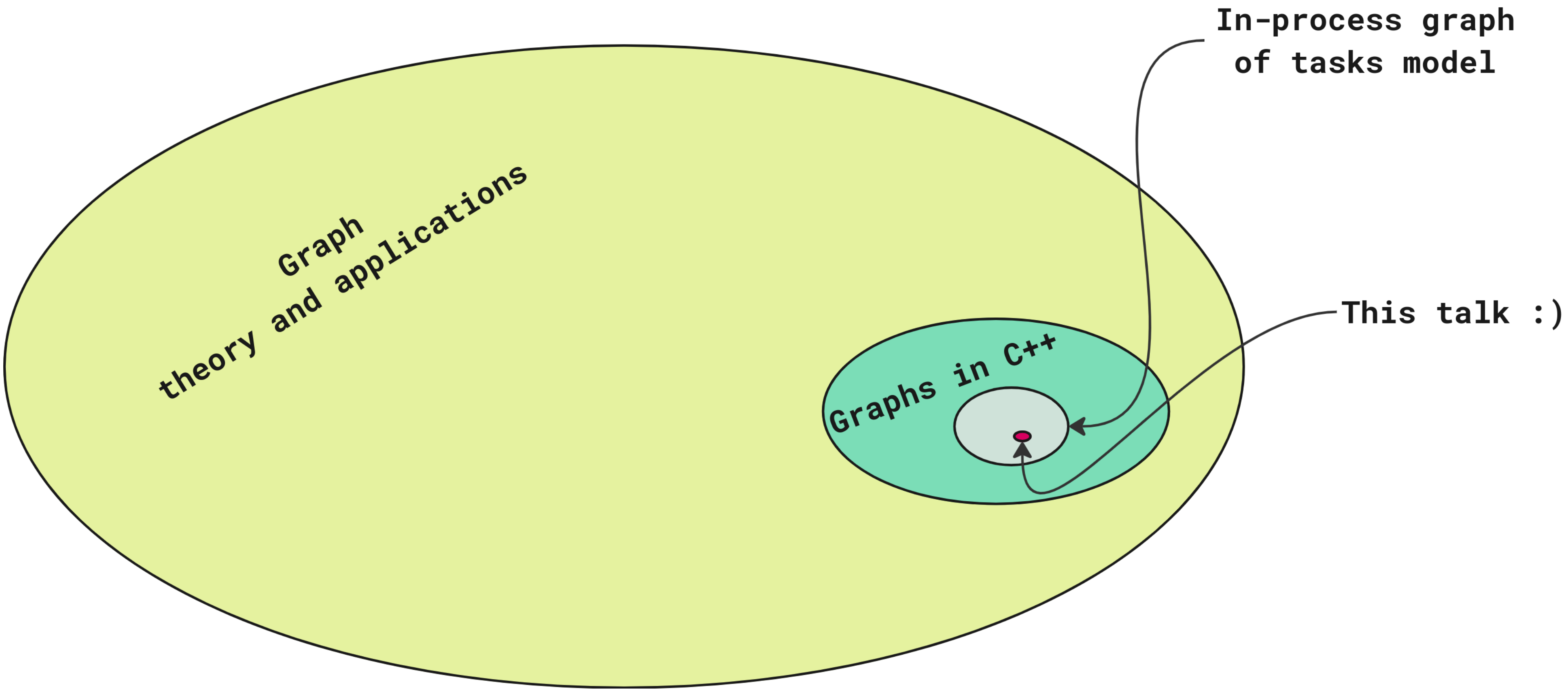
Graph is two sets…

$$\{\boldsymbol{V}, \boldsymbol{E} | \boldsymbol{E} \subset \boldsymbol{V}^2\}$$

Graph is tasks and dependencies between them!

```
feldsherov@feldsherov-ws01:~$ systemctl list-dependencies local-fs.target
local-fs.target
● ├─-.mount
● ├─boot-efi.mount
● ├─boot.mount
● ├─run-lock.mount
● ├─systemd-fsck-root.service
● ├─systemd-remount-fs.service
● └─tmp.mount
```

The talk



In-process graph
of tasks model
Graph
theory and applications
This talk :)
Graphs in C++

Disclaimers!

Disclaimers!

- The talk is based on my journey

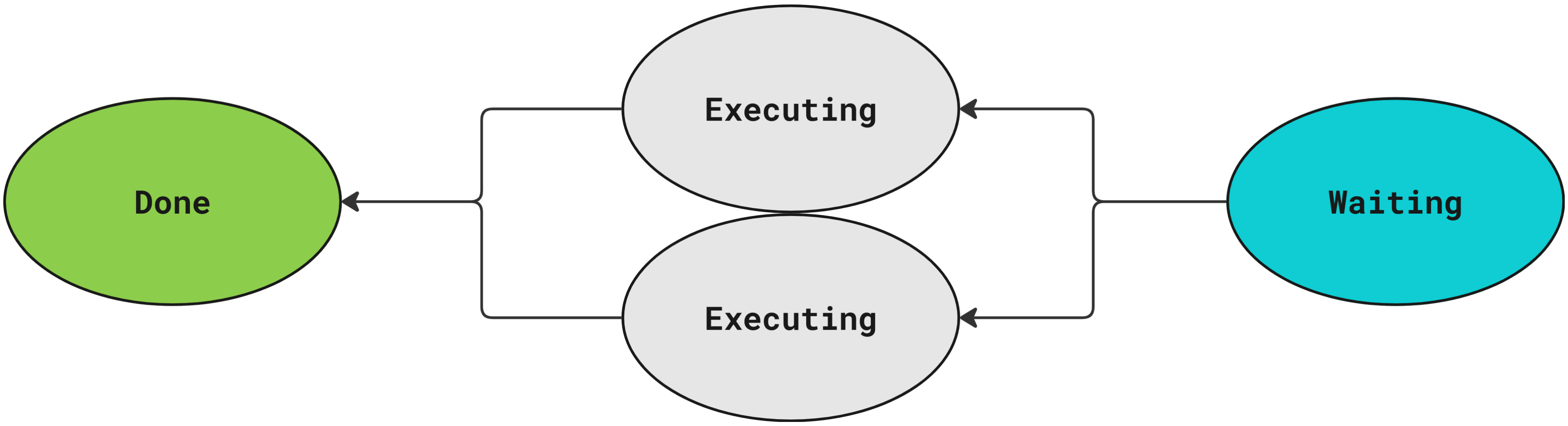
Disclaimers!

- The talk is based on my journey
- Code quality is often limited by slide size

Why graphs?

Nice abstraction to structure the code and .....

Parallelism!



Portrait of our client

Portrait of our client

- C++ binary or library ...

Portrait of our client

- C++ binary or library ...
- ... developed by a large team, ...

Portrait of our client

- C++ binary or library ...
- ... developed by a large team, ...
- with tasks and dependencies between them inside.

Generic requirements

Generic requirements

- Code is readable ...

Generic requirements

- Code is readable ...
- ... and reasonably performant*.

Generic requirements

- Code is readable ...
- ... and reasonably performant*.
- Tasks and dependencies are easy to add.

Domain specific requirements

- Granular control over threading model
- Conditions on edges
- Async operations in task
- Data transfer through edges
- Streaming of the partial results
- Cycles O_o

What options do we have?

* we are skipping trivial approaches and jumping directly to juicy parts

C++ 11 futures

Futures recap [1]

```cpp
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [1]

```cpp
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [1]

```cpp
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [1]

```
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [1]

```
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [1]

```
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [1]

```
int main() {
  std::promise<int> p;
  std::future<int> f = p.get_future();
  std::jthread t(
    [p=std::move(p)]() mutable {
        p.set_value(179);
    }
  );
  f.wait();
  std::cout << f.get() << std::endl;
  return 0;
}
```

[1] Godbolt link

Futures recap [2]

```cpp
int main() {
  std::promise<int> p;
  std::shared_future<int> f(p.get_future());

  std::jthread t_first([f](){
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
  });
  std::jthread t_second([f](){
    f.wait();
    std::cout << "t_second: " << f.get() << std::endl;
  });
  p.set_value(179);
  return 0;
}
```

[1] Godbolt link

Futures recap [2]

```cpp
int main() {
  std::promise<int> p;
  std::shared_future<int> f(p.get_future());

  std::jthread t_first([f](){
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
  });
  std::jthread t_second([f](){
    f.wait();
    std::cout << "t_second: " << f.get() << std::endl;
  });
  p.set_value(179);
  return 0;
}
```

[1] Godbolt link

Futures recap [2]

```cpp
int main() {
  std::promise<int> p;
  std::shared_future<int> f(p.get_future());

  std::jthread t_first([f](){
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
  });
  std::jthread t_second([f](){
    f.wait();
    std::cout << "t_second: " << f.get() << std::endl;
  });
  p.set_value(179);
  return 0;
}
```

[1] Godbolt link

Futures recap [2]

```
int main() {
  std::promise<int> p;
  std::shared_future<int> f(p.get_future());

  std::jthread t_first([f](){
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
  });
  std::jthread t_second([f](){
    f.wait();
    std::cout << "t_second: " << f.get() << std::endl;
  });
  p.set_value(179);
  return 0;
}
```

[1] Godbolt link

Futures recap [2]

```
int main() {
  std::promise<int> p;
  std::shared_future<int> f(p.get_future());

  std::jthread t_first([f](){
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
  });
  std::jthread t_second([f](){
    f.wait();
    std::cout << "t_second: " << f.get() << std::endl;
  });
  p.set_value(179);
  return 0;
}
```

[1] Godbolt link

Futures recap [2]

```
int main() {
  std::promise<int> p;
  std::shared_future<int> f(p.get_future());

  std::jthread t_first([f](){
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
  });
  std::jthread t_second([f](){
    f.wait();
    std::cout << "t_second: " << f.get() << std::endl;
  });
  p.set_value(179);
  return 0;
}
```

[1] Godbolt link

```
t_second:t_first:  179179\n\n
```

Futures recap [2] the fix

```
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

Futures recap [2] the fix

```
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

Futures recap [2] the fix

```
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

Futures recap [2] the fix

```
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

Futures recap [2] the fix

```
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

Futures recap [2] the fix

```cpp
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

Futures recap [2] the fix

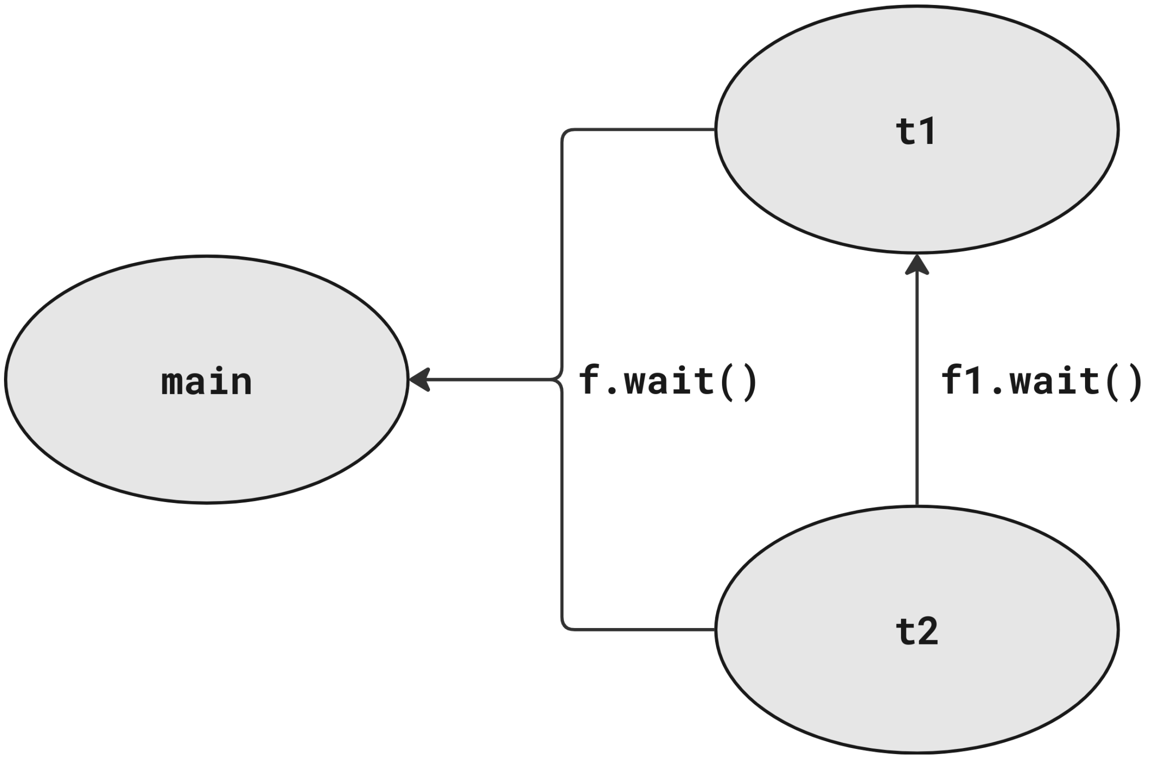
```
std::promise<int> p;
std::promise<void> t1_promise;
std::shared_future<int> f(p.get_future());

std::jthread t_second([f, f1 = t1_promise.get_future()](){
  f1.wait(); f.wait();
  std::cout << "t_second: " << f.get() << std::endl;
});
std::jthread t_first(
  [f, t1_p=std::move(t1_promise)]() mutable {
    f.wait();
    std::cout << "t_first: " << f.get() << std::endl;
    t1_p.set_value();
  });
p.set_value(179);
```

[1] Godbolt link

```
t_first: 179
t_second: 179
```

Ooops, we did a graph!



Let's make the task more real! [1]

```
class MyAwesomeTaskInterface {
  public:
    using FutureType = std::shared_future<MyAwesomeTaskResult>;

    virtual FutureType GetFuture() = 0;

  protected:
    virtual void Run() = 0;
};
```

[1] Godbolt link

Let's make the task more real! [1]

```cpp
class MyAwesomeTaskInterface {
  public:
    using FutureType = std::shared_future<MyAwesomeTaskResult>;

    virtual FutureType GetFuture() = 0;

  protected:
    virtual void Run() = 0;
};
```

[1] Godbolt link

Let’s make the task more real! [1]

```
class MyAwesomeTaskInterface {
  public:
    using FutureType = std::shared_future<MyAwesomeTaskResult>;

    virtual FutureType GetFuture() = 0;

  protected:
    virtual void Run() = 0;
};
```

[1] Godbolt link

Let's make the task more real! [1]

```cpp
class MyAwesomeTaskInterface {
  public:
    using FutureType = std::shared_future<MyAwesomeTaskResult>;

    virtual FutureType GetFuture() = 0;

  protected:
    virtual void Run() = 0;
};
```

[1] Godbolt link

Let's make the task more real! [2]

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
....

private:
  FirstDependencyInterface::FutureType first_dep_;
  SecondDependencyInterface::FutureType second_dep_;
  std::promise<MyAwesomeTaskResult> promise_;
  FutureType future_;
};
```

[1] Godbolt link

Let’s make the task more real! [2]

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
....

private:
  FirstDependencyInterface::FutureType first_dep_;
  SecondDependencyInterface::FutureType second_dep_;
  std::promise<MyAwesomeTaskResult> promise_;
  FutureType future_;
};
```

[1] Godbolt link

Let's make the task more real! [2]

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
....

private:
  FirstDependencyInterface::FutureType first_dep_;
  SecondDependencyInterface::FutureType second_dep_;
  std::promise<MyAwesomeTaskResult> promise_;
  FutureType future_;
};
```

[1] Godbolt link

Let's make the task more real! [2]

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
....

private:
  FirstDependencyInterface::FutureType first_dep_;
  SecondDependencyInterface::FutureType second_dep_;
  std::promise<MyAwesomeTaskResult> promise_;
  FutureType future_;
};
```

[1] Godbolt link

Let’s make the task more real! [2]

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
....

private:
  FirstDependencyInterface::FutureType first_dep_;
  SecondDependencyInterface::FutureType second_dep_;
  std::promise<MyAwesomeTaskResult> promise_;
  FutureType future_;
};
```

[1] Godbolt link

Let’s make the task more real! [2]

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
public:
  MyAwesomeTask(
      FirstDependencyInterface& first_dep,
      SecondDependencyInterface& second_dep
  );

  FutureType GetFuture() override {
      return future_;
  }
  ....
};
```

[1] Godbolt link

Let’s make the task more real! [2]

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
public:
  MyAwesomeTask(
      FirstDependencyInterface& first_dep,
      SecondDependencyInterface& second_dep
  );

  FutureType GetFuture() override {
      return future_;
  }
  ....
};
```

[1] Godbolt link

Let's make the task more real! [2]

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
public:
  MyAwesomeTask(
      FirstDependencyInterface& first_dep,
      SecondDependencyInterface& second_dep
  );

  FutureType GetFuture() override {
      return future_;
  }
  ....
};
```

[1] Godbolt link

Let's make the task more real! [2]

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
public:
  MyAwesomeTask(
      FirstDependencyInterface& first_dep,
      SecondDependencyInterface& second_dep
  );

  FutureType GetFuture() override {
      return future_;
  }
  ....
};
```

[1] Godbolt link

Let's make the task more real! [2]

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
....
protected:
  void Run() override {
      first_dep_.wait();
      second_dep_.wait();
      MyAwesomeTaskResult result;
      // Do something.
      promise_.set_value(std::move(result));
  }
....
};
```

[1] Godbolt link

Let’s make the task more real! [2]

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
....
protected:
  void Run() override {
      first_dep_.wait();
      second_dep_.wait();
      MyAwesomeTaskResult result;
      // Do something.
      promise_.set_value(std::move(result));
  }
....
};
```

[1] Godbolt link

But who will execute it?

Ugly fix...

```
class MyAwesomeTask : public MyAwesomeTaskInterface {
      MyAwesomeTask(
          FirstDependencyInterface& first_dep,
          SecondDependencyInterface& second_dep
      ) : ..., executor_([this](){Run();}) {}
  private:
      ...
      std::jthread executor_;
  };
```

[1] Godbolt link

What about non-ugly fix?

What about non-ugly fix?

- when_all and future.then from std::experimental

What about non-ugly fix?

- when_all and future.then from std::experimental
- ... or boost

Less ugly fix

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
  MyAwesomeTask(
    FirstDependencyInterface& first_dep,
    SecondDependencyInterface& second_dep
  ) {
    std::experimental::when_all(first_dep_, second_dep_)
      .then(
        [this](auto deps){
          Run(std::move(deps));
        }
      );
  }
  ....
};
```

Less ugly fix

```cpp
class MyAwesomeTask : public MyAwesomeTaskInterface {
  MyAwesomeTask(
    FirstDependencyInterface& first_dep,
    SecondDependencyInterface& second_dep
  ) {
    std::experimental::when_all(first_dep_, second_dep_)
      .then(
        [this](auto deps){
          Run(std::move(deps));
        }
      );
  }
  ....
};
```

Does it look awesome?

Futures conclusions

Futures conclusions

- Hard to prevent futures spaghetti.

Futures conclusions

- Hard to prevent futures spaghetti.
- People fail to find correct dependencies!

Futures conclusions

- Hard to prevent futures spaghetti.
- People fail to find correct dependencies!
- **Not easy to add new tasks to large graphs!**

Focus on tasks, not edges!

Target state

Target state

- Declare result data type

Target state

- Declare result data type
- List all types you depend on

Target state

- Declare result data type
- List all types you depend on
- Implement business logic in you cpp file

Target state

- Declare result data type
- List all types you depend on
- Implement business logic in you cpp file
- .... C++ magic ....

Target state

- Declare result data type
- List all types you depend on
- Implement business logic in you cpp file
- .... C++ magic ....
- Graph is ready to be executed

Attention slideware ahead!

Slideware for task

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for task

```
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
    const auto& dep1 = Get<DependencyFirst>();
    const auto& dep2 = Get<DependencySecond>();
    ResultType& result = GetOutput<ResultType>();
    // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (to also slideware)

Slideware for execution

```
std::expected<SomeDesiredOutputType, Error> result_or_error =
    GetGraphFor<SomeDesiredOutputType>()
      .Execute(first_input, second_input);
```

Why do we need this?
How to make it work?

Unique name

```cpp
#define _CONCAT(a, b) a##_##b
#define CONCAT(a, b) _CONCAT(a, b)
#define UNIQUE_NAME(base) CONCAT(base, __COUNTER__)

int UNIQUE_NAME(foo) = 1;
int UNIQUE_NAME(foo) = 2;
int UNIQUE_NAME(foo) = 3;

int main() {
    std::cout << foo_0 << " "
              << foo_1 << " "
              << foo_2 << std::endl;
    return 0;
}
```

[1] Godbolt link

Unique name

```cpp
#define _CONCAT(a, b) a##_##b
#define CONCAT(a, b) _CONCAT(a, b)
#define UNIQUE_NAME(base) CONCAT(base, __COUNTER__)

int UNIQUE_NAME(foo) = 1;
int UNIQUE_NAME(foo) = 2;
int UNIQUE_NAME(foo) = 3;

int main() {
    std::cout << foo_0 << " "
              << foo_1 << " "
              << foo_2 << std::endl;
    return 0;
}
```

[1] Godbolt link

Unique name

```cpp
#define _CONCAT(a, b) a##_##b
#define CONCAT(a, b) _CONCAT(a, b)
#define UNIQUE_NAME(base) CONCAT(base, __COUNTER__)

int UNIQUE_NAME(foo) = 1;
int UNIQUE_NAME(foo) = 2;
int UNIQUE_NAME(foo) = 3;

int main() {
    std::cout << foo_0 << " "
              << foo_1 << " "
              << foo_2 << std::endl;
    return 0;
}
```

[1] Godbolt link

Unique name

```cpp
#define _CONCAT(a, b) a##_##b
#define CONCAT(a, b) _CONCAT(a, b)
#define UNIQUE_NAME(base) CONCAT(base, __COUNTER__)

int UNIQUE_NAME(foo) = 1;
int UNIQUE_NAME(foo) = 2;
int UNIQUE_NAME(foo) = 3;

int main() {
    std::cout << foo_0 << " "
              << foo_1 << " "
              << foo_2 << std::endl;
    return 0;
}
```

[1] Godbolt link

Link time registration

```
#define REGISTER_TASK(task) int UNIQUE_NAME(task) =     \
  [](std::string name) -> int {                         \
    std::cout << "Registered: " << name << std::endl;   \
    return 0;                                           \
  }(#task);

....
REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link

Type to int mapping

```cpp
// h file
int GetNextId();

template<typename Type>
int GetTypeId() {
    static int type_id = GetNextId();
    return type_id;
}

// cpp file
int GetNextId() {
    static std::atomic<int> next_id{0};
    return next_id.fetch_add(1);
}
```

[1] Godbolt link

Type to int mapping

```cpp
// h file
int GetNextId();

template<typename Type>
int GetTypeId() {
    static int type_id = GetNextId();
    return type_id;
}

// cpp file
int GetNextId() {
    static std::atomic<int> next_id{0};
    return next_id.fetch_add(1);
}
```

[1] Godbolt link

Type to int mapping

```cpp
// h file
int GetNextId();

template<typename Type>
int GetTypeId() {
    static int type_id = GetNextId();
    return type_id;
}

// cpp file
int GetNextId() {
    static std::atomic<int> next_id{0};
    return next_id.fetch_add(1);
}
```

[1] Godbolt link

Type to int mapping

```cpp
// h file
int GetNextId();

template<typename Type>
int GetTypeId() {
    static int type_id = GetNextId();
    return type_id;
}

// cpp file
int GetNextId() {
    static std::atomic<int> next_id{0};
    return next_id.fetch_add(1);
}
```

[1] Godbolt link

Type to int mapping usage

```
// Usage in other cc file
int main() {
  std::cout << GetTypeId<int>() << std::endl;
  std::cout << GetTypeId<MyResultType>() << std::endl;
  return 0;
}
```

[1] Godbolt link

Bringing registration together

[1] Godbolt link

Bringing registration together

- Get a global factory in the registration lambda.

[1] Godbolt link

Bringing registration together

- Get a global factory in the registration lambda.
- Exchange TaskType and ResultTypes to ids.

[1] Godbolt link

Bringing registration together

- Get a global factory in the registration lambda.
- Exchange TaskType and ResultTypes to ids.
- Build graph with this ids.

[1] Godbolt link

Bringing registration together

- Get a global factory in the registration lambda.
- Exchange TaskType and ResultTypes to ids.
- Build graph with this ids.
- Execute it in any way you want :)

[1] Godbolt link

What is still missing?

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {

  void Run() {
      const auto& dep1 = Get<DependencyFirst>();
      const auto& dep2 = Get<DependencySecond>();
      ResultType& result = GetOutput<ResultType>();
      // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (also to slideware)

What is still missing?

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
      const auto& dep1 = Get<DependencyFirst>();
      const auto& dep2 = Get<DependencySecond>();
      ResultType& result = GetOutput<ResultType>();
      // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (also to slideware)

What is still missing?

```
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {
  void Run() {
      const auto& dep1 = Get<DependencyFirst>();
      const auto& dep2 = Get<DependencySecond>();
      ResultType& result = GetOutput<ResultType>();
      // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (also to slideware)

What is still missing?

```cpp
class MyAwesomeTask : public BaseTask<ResultType,
                                      DependencyFirst,
                                      DependencySecond> {

  void Run() {
      const auto& dep1 = Get<DependencyFirst>();
      const auto& dep2 = Get<DependencySecond>();
      ResultType& result = GetOutput<ResultType>();
      // do something and set result
  }
};

REGISTER_TASK(MyAwesomeTask);
```

[1] Godbolt link (also to slideware)

Get implementation idea

```
template<typename ResultType, typename... Deps>
class BaseTask {
public:
  template<typename T>
  const T& Get() {
    static_assert(contains_type_v<T, Deps...> == true);
    return *static_cast<const T*>(*storage_.get(GetTypeId<T>()))
  }
  ....
};
```

[1] Godbolt link

Get implementation idea

```cpp
template<typename ResultType, typename... Deps>
class BaseTask {
public:
  template<typename T>
  const T& Get() {
    static_assert(contains_type_v<T, Deps...> == true);
    return *static_cast<const T*>(*storage_.get(GetTypeId<T>))
  }
  ....
};
```

[1] Godbolt link

Get implementation idea

```
template<typename ResultType, typename... Deps>
class BaseTask {
public:
  template<typename T>
  const T& Get() {
    static_assert(contains_type_v<T, Deps...> == true);
    return *static_cast<const T*>(*storage_.get(GetTypeId<T>()))
  }
  ....
};
```

[1] Godbolt link

Get implementation idea

```
template<typename ResultType, typename... Deps>
class BaseTask {
public:
  template<typename T>
  const T& Get() {
    static_assert(contains_type_v<T, Deps...> == true);
    return *static_cast<const T*>(*storage_.get(GetTypeId<T>()))
  }
  ....
};
```

[1] Godbolt link

Conclusions

Conclusions

- This trick allows to separate task development from graph description.

Conclusions

- This trick allows to separate task development from graph description.
- Works really nice for a certain scope of tasks!

Conclusions

- This trick allows to separate task development from graph description.
- Works really nice for a certain scope of tasks!
- Requires quite a lot of hustle to make it work.

Conclusions

- This trick allows to separate task development from graph description.
- Works really nice for a certain scope of tasks!
- Requires quite a lot of hustle to make it work.
- Not all "domain specific" requirements are feasible

Conclusions

- This trick allows to separate task development from graph description.
- Works really nice for a certain scope of tasks!
- Requires quite a lot of hustle to make it work.
- Not all "domain specific" requirements are feasible
- ...but with coroutines a few more items may be supported.

Conclusions

- This trick allows to separate task development from graph description.
- Works really nice for a certain scope of tasks!
- Requires quite a lot of hustle to make it work.
- Not all "domain specific" requirements are feasible
- ...but with coroutines a few more items may be supported.
- Very nice overall but "hardly composable".

Stuff I haven't seen in prod,
but I would want to try!

Taskflow

```cpp
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(  // create four tasks
  [] () { std::cout << "TaskA\n"; },
  [] () { std::cout << "TaskB\n"; },
  [] () { std::cout << "TaskC\n"; },
  [] () { std::cout << "TaskD\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C

executor.run(taskflow).wait();
```

[1] CppCon 2020 talk about taskflow

Taskflow

```cpp
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(  // create four tasks
  [] () { std::cout << "TaskA\n"; },
  [] () { std::cout << "TaskB\n"; },
  [] () { std::cout << "TaskC\n"; },
  [] () { std::cout << "TaskD\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C

executor.run(taskflow).wait();
```

[1] CppCon 2020 talk about taskflow

Taskflow

```
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(  // create four tasks
  [] () { std::cout << "TaskA\n"; },
  [] () { std::cout << "TaskB\n"; },
  [] () { std::cout << "TaskC\n"; },
  [] () { std::cout << "TaskD\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C

executor.run(taskflow).wait();
```

[1] CppCon 2020 talk about taskflow

Taskflow

```cpp
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(  // create four tasks
  [] () { std::cout << "TaskA\n"; },
  [] () { std::cout << "TaskB\n"; },
  [] () { std::cout << "TaskC\n"; },
  [] () { std::cout << "TaskD\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C

executor.run(taskflow).wait();
```

[1] CppCon 2020 talk about taskflow

Taskflow

```
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(  // create four tasks
  [] () { std::cout << "TaskA\n"; },
  [] () { std::cout << "TaskB\n"; },
  [] () { std::cout << "TaskC\n"; },
  [] () { std::cout << "TaskD\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C

executor.run(taskflow).wait();
```

[1] CppCon 2020 talk about taskflow

Taskflow

```cpp
tf::Executor executor;
tf::Taskflow taskflow;

auto [A, B, C, D] = taskflow.emplace(  // create four tasks
  [] () { std::cout << "TaskA\n"; },
  [] () { std::cout << "TaskB\n"; },
  [] () { std::cout << "TaskC\n"; },
  [] () { std::cout << "TaskD\n"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C

executor.run(taskflow).wait();
```

[1] CppCon 2020 talk about taskflow

stdexec

```
exec::static_thread_pool pool(3);
auto sched = pool.get_scheduler();

auto zero = stdexec::just(0) | stdexec::then(fun);
auto one = stdexec::just(1) | stdexec::then(fun);
auto two = stdexec::just(2) | stdexec::then(fun);
auto work = stdexec::when_all(
    stdexec::starts_on(sched, zero),
    stdexec::starts_on(sched, one),
    stdexec::starts_on(sched, two)
);

// Launch the work and wait for the result
auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
```

[1] Godbolt link

stdexec

```
exec::static_thread_pool pool(3);
auto sched = pool.get_scheduler();

auto zero = stdexec::just(0) | stdexec::then(fun);
auto one = stdexec::just(1) | stdexec::then(fun);
auto two = stdexec::just(2) | stdexec::then(fun);
auto work = stdexec::when_all(
    stdexec::starts_on(sched, zero),
    stdexec::starts_on(sched, one),
    stdexec::starts_on(sched, two)
);

// Launch the work and wait for the result
auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
```

[1] Godbolt link

stdexec

```
exec::static_thread_pool pool(3);
auto sched = pool.get_scheduler();

auto zero = stdexec::just(0) | stdexec::then(fun);
auto one = stdexec::just(1) | stdexec::then(fun);
auto two = stdexec::just(2) | stdexec::then(fun);
auto work = stdexec::when_all(
    stdexec::starts_on(sched, zero),
    stdexec::starts_on(sched, one),
    stdexec::starts_on(sched, two)
);

// Launch the work and wait for the result
auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
```

[1] Godbolt link

stdexec

```
exec::static_thread_pool pool(3);
auto sched = pool.get_scheduler();

auto zero = stdexec::just(0) | stdexec::then(fun);
auto one = stdexec::just(1) | stdexec::then(fun);
auto two = stdexec::just(2) | stdexec::then(fun);
auto work = stdexec::when_all(
    stdexec::starts_on(sched, zero),
    stdexec::starts_on(sched, one),
    stdexec::starts_on(sched, two)
);

// Launch the work and wait for the result
auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
```

[1] Godbolt link

stdexec

```cpp
exec::static_thread_pool pool(3);
auto sched = pool.get_scheduler();

auto zero = stdexec::just(0) | stdexec::then(fun);
auto one = stdexec::just(1) | stdexec::then(fun);
auto two = stdexec::just(2) | stdexec::then(fun);
auto work = stdexec::when_all(
    stdexec::starts_on(sched, zero),
    stdexec::starts_on(sched, one),
    stdexec::starts_on(sched, two)
);

// Launch the work and wait for the result
auto [i, j, k] = stdexec::sync_wait(std::move(work)).value();
```

[1] Godbolt link

P2300 - std::execution

Conclusions

Conclusions

- There are many more async execution libraries.

Conclusions

- There are many more async execution libraries.
- Huge graphs are not trivial to declare.

Conclusions

- There are many more async execution libraries.
- Huge graphs are not trivial to declare.
- P2300 should make decomposition of graphs easier.

All talk in one slide

All talk in one slide

- C++11 futures are not enough in large teams.

All talk in one slide

- C++11 futures are not enough in large teams.
- Recipe of a scalable task graph framework.

All talk in one slide

- C++11 futures are not enough in large teams.
- Recipe of a scalable task graph framework.
- Link time registration is a nice tool for the above.

All talk in one slide

- C++11 futures are not enough in large teams.
- Recipe of a scalable task graph framework.
- Link time registration is a nice tool for the above.
- Coroutines can bring async tasks to you graphs.

All talk in one slide

- C++11 futures are not enough in large teams.
- Recipe of a scalable task graph framework.
- Link time registration is a nice tool for the above.
- Coroutines can bring async tasks to you graphs.
- I hope P2300 will make everything composable.

QUESTIONS?

- [0] Tg: @feldsherov
- [1] Link: svyat
- [2] Mail: svyat@feldsherov.name