Composable Tasking in Taskflow

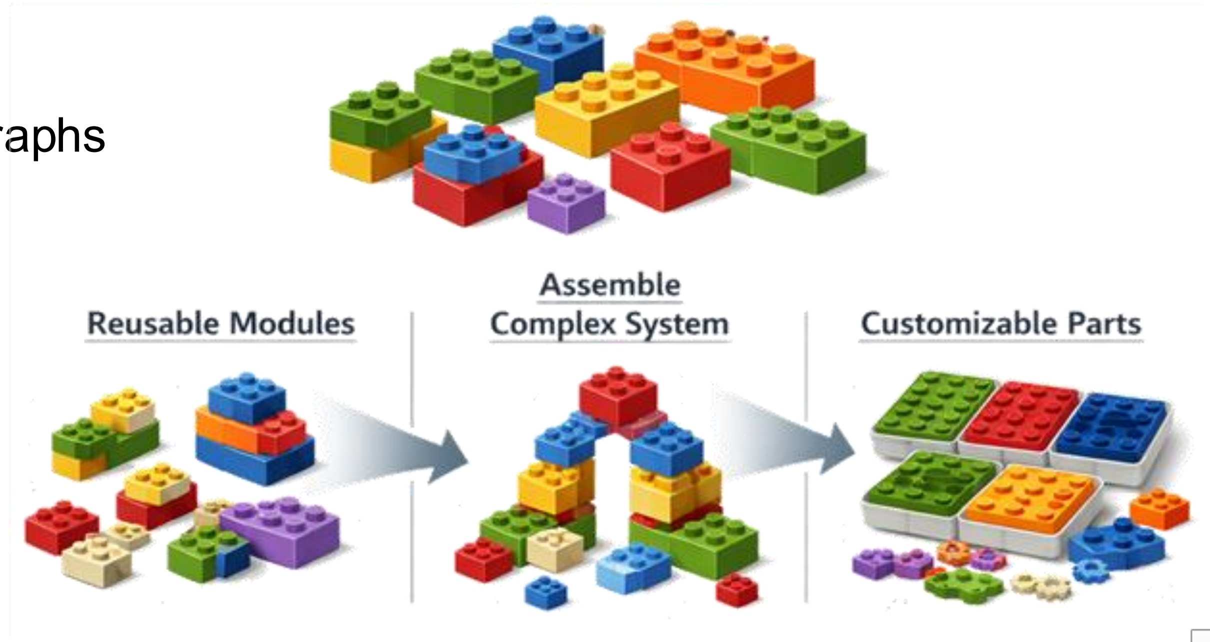
Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://taskflow.github.io/

Takeaways

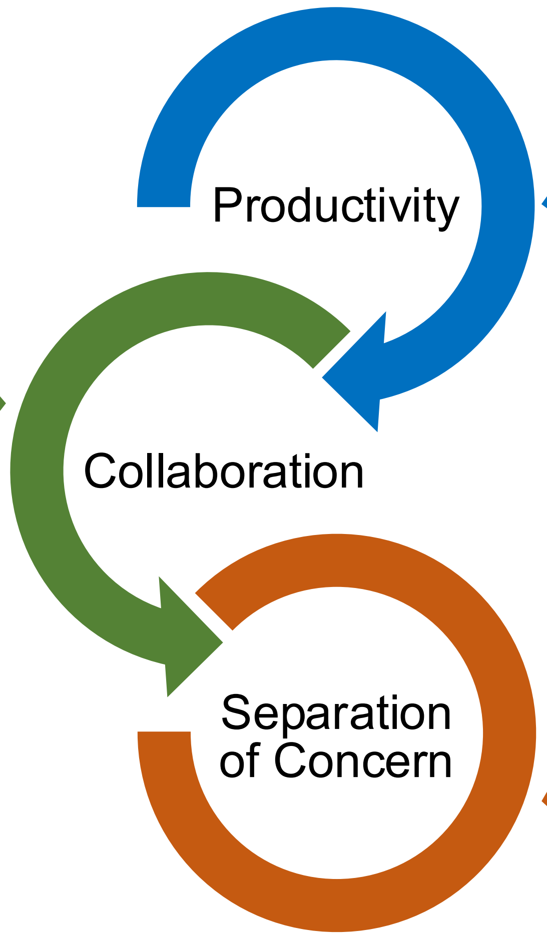
- Understand the importance of task graph composition
- Learn how to compose task graphs in Taskflow
- Showcase two real-world examples of task graph composition
- Conclude

Task Graph Composition

- **Build complex computations by composing small task graphs**
  - Instead of designing one giant monolithic task graph, we design modular subgraphs that can be connected, reused, and assembled
  - Task graph composition treats parallel programs like Lego blocks, not hand-crafted sculptures
- **Key advantages of task graph composition:**
  - Exposes parallelism at multiple levels
    - Parallelism within subgraphs
    - Parallelism between different subgraphs
  - Modular interface
    - Easier to test, verify, and reason
  - Reusable blocks
    - Algorithmic patterns and primitives
  - Scalable design



Why is Task Graph Composition Important?



Developers reason about small graphs instead of massive ones, which facilitates faster design, testing, and iteration.

Different people can work on different algorithmic components without stepping on each other, which enables clean interface and ownership between boundaries.

Composition lets you separate “what computation does” from “how it is integrated”, enabling independent design, reuse, and faster evolution of each component.

Composable Tasking is Not Well-supported by Existing Tools

- **Most task-parallel programming libraires assume a flat task graph structure**
  - They can only handle a single, flat task graph at once, with no notion of reuse
  - As a result, existing libraries often fail to scale to very large and complex parallel workloads

```
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp task
    taskA();

    #pragma omp task
    taskB();

    #pragma omp task
    taskC();
  }
}
```

```
tbb::flow::graph g;
tbb::flow::continue_node<tbb::flow::continue_msg> task1(g,
  [](const tbb::flow::continue_msg&){ std::cout << "taskA\n";}
);
tbb::flow::continue_node<tbb::flow::continue_msg> task2(g,
  [](const tbb::flow::continue_msg&){ std::cout << "taskB\n";}
);
make_edge(taskA, taskB);
```

In TBB, you can create nodes and edges but cannot encapsulate a task graph as a reusable object.

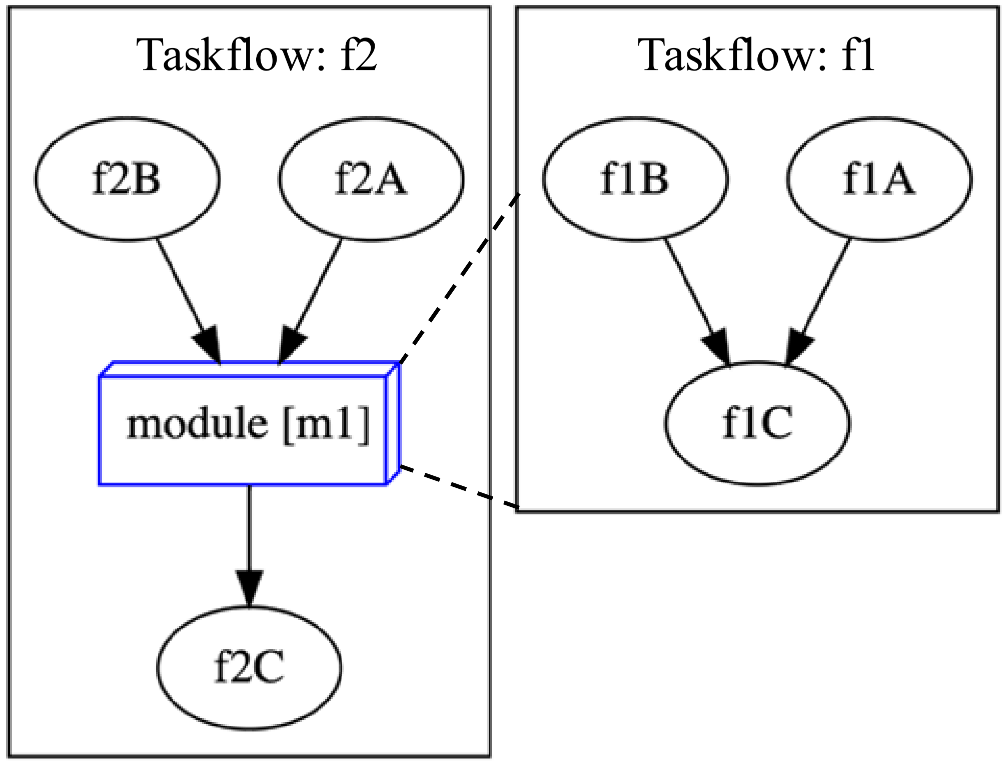
In OpenMP tasking, you cannot treat each task as a reusable task that can compose other tasks or graphs.

Takeaways

- Understand the importance of task graph composition
- Learn how to compose task graphs in Taskflow
- Showcase two real-world examples of task graph composition
- Conclude

Task Graph Composition in Taskflow: Module Task

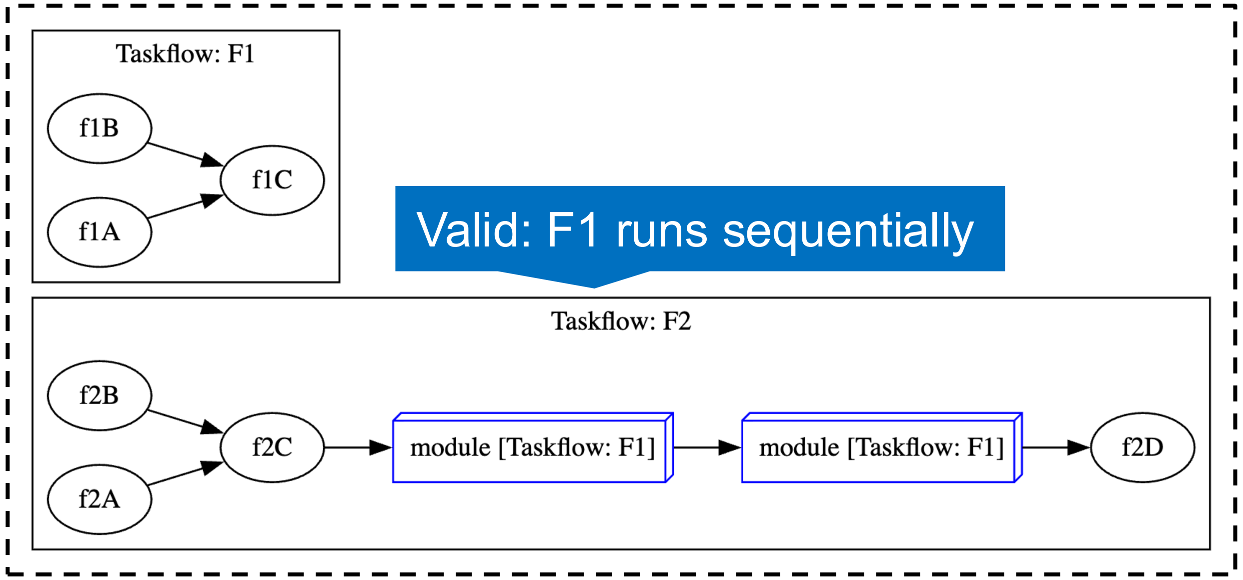
```
tf::Taskflow f1, f2;

auto [f1A, f1B, f1C] = f1.emplace(
  [](){ std::cout << "Task f1A\n"; },
  [](){ std::cout << "Task f1B\n"; },
  [](){ std::cout << "Task f1C\n"; }
);
f1C.succeed(f1A, f1B);

auto [f2A, f2B, f2C] = f2.emplace(
  [](){ std::cout << "Task f2A\n"; },
  [](){ std::cout << "Task f2B\n"; },
  [](){ std::cout << "Task f2C\n"; }
);

auto module_task = f2.composed_of(f1);
f1_module_task.succeed(f2A, f2B)
                  .precede(f2C);
```

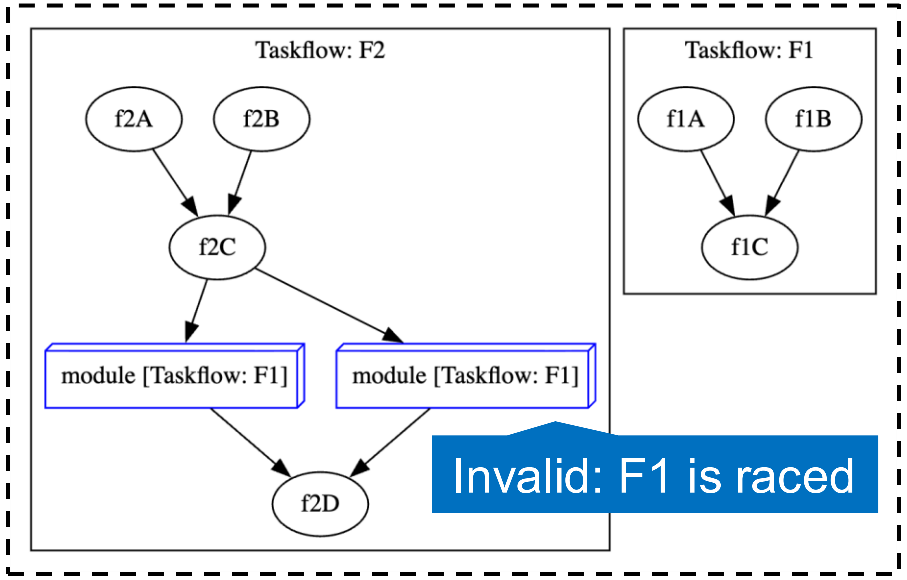


Once created, a module task can be used just like other Taskflow tasks to build dependencies.

A Module Task Does Not Own its Task Graph

- **A module task only keeps a reference to the task graph it encapsulates**
  - Just like the executor that does not retain any ownership over a running taskflow
- **It is your responsibility to ensure that a composed graph remains:**
  - Alive throughout its execution (to avoid invalid references or program crashes)
  - Exclusive during its execution (to prevent concurrent modification or task races)

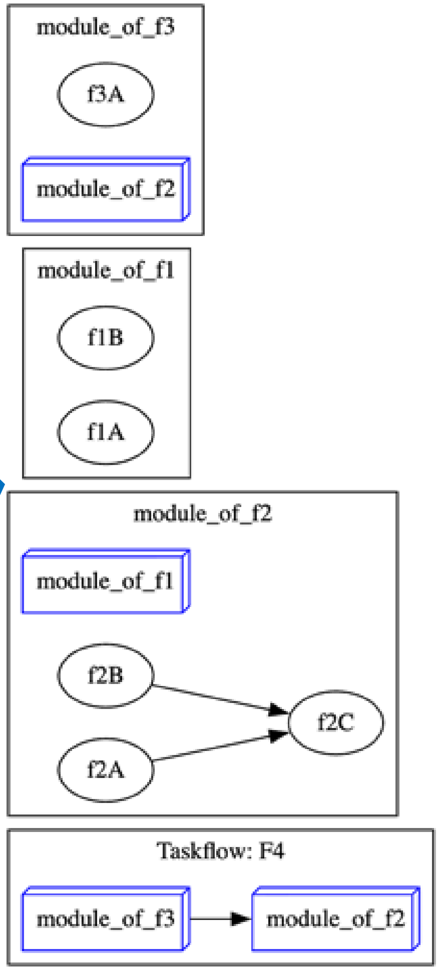




A Module Task can Recursively Encapsulate Graphs

```
tf::Taskflow f1("F1");
auto f1A = f1.emplace([&](){ std::cout << "F1 TaskA\n"; }).name("F1A");
auto f1B = f1.emplace([&](){ std::cout << "F1 TaskB\n"; }).name("F1B");

tf::Taskflow f2("F2");
auto f2A = f2.emplace([&](){ std::cout << " F2 TaskA\n"; }).name("f2A");
auto f2B = f2.emplace([&](){ std::cout << " F2 TaskB\n"; }).name("f2B");
auto f2C = f2.emplace([&](){ std::cout << " F2 TaskC\n"; }).name("f2C");
f2C.succeed(f2A, f2B);
f2.composed_of(f1).name("module_of_f1");

tf::Taskflow f3("F3");
f3.composed_of(f2).name("module_of_f2");
f3.emplace([](){ std::cout << "  F3 TaskA\n"; }).name("f3A");

tf::Taskflow f4("F4");
auto f3_module_task = f4.composed_of(f3).name("module_of_f3");
auto f2_module_task = f4.composed_of(f2).name("module_of_f2");
f3_module_task.precede(f2_module_task);
```

Module tasks can express hierarchical task parallelism in a very natural way.



Another Way to Create a Module Task

- `tf::make_module_task` **returns a task that encapsulates a task graph**
  - Need `#include<taskflow/algorithm/module.hpp>`

```
tf::Taskflow f1, f2;

auto [f1A, f1B] = f1.emplace(
  [](){ std::cout << "Task f1A\n"; },
  [](){ std::cout << "Task f1B\n"; }
);
auto [f2A, f2B, f2C] = f2.emplace(
  [](){ std::cout << "Task f2A\n"; },
  [](){ std::cout << "Task f2B\n"; },
  [](){ std::cout << "Task f2C\n"; }
);
auto module_task = f2.composed_of(f1);

f1_module_task.succeed(f2A, f2B)
                .precede(f2C);
```

=

Semantically equivalent

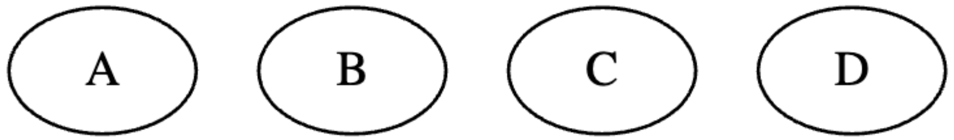
```
tf::Taskflow f1, f2;

auto [f1A, f1B] = f1.emplace(
  [](){ std::cout << "Task f1A\n"; },
  [](){ std::cout << "Task f1B\n"; }
);
auto [f2A, f2B, f2C] = f2.emplace(
  [](){ std::cout << "Task f2A\n"; },
  [](){ std::cout << "Task f2B\n"; },
  [](){ std::cout << "Task f2C\n"; }
);
auto module_task = 
f2.emplace(tf::make_module_task(f1));

f1_module_task.succeed(f2A, f2B)
                .precede(f2C);
```

Launch a Module Task Asynchronously

- **`tf::make_module_task` can work with asynchronous tasking**

```
tf::Executor executor;

tf::Taskflow A, B, C, D;

A.emplace([](){ printf("Taskflow A\n"); });
B.emplace([](){ printf("Taskflow B\n"); });
C.emplace([](){ printf("Taskflow C\n"); });
D.emplace([](){ printf("Taskflow D\n"); });

executor.async(tf::make_module_task(A));
executor.async(tf::make_module_task(B));
executor.async(tf::make_module_task(C));
executor.async(tf::make_module_task(D));
executor.wait_for_all();
```

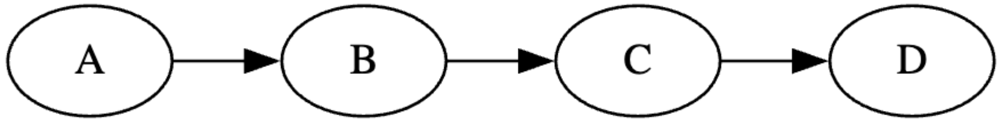
Creates four async tasks each encapsulating a task graph



Launch a Module Task Asynchronously (cont'd)

- **`tf::make_module_task` can work with dynamic task graph parallelism**

```
tf::Executor executor;

tf::Taskflow A, B, C, D;

A.emplace([](){ printf("Taskflow A\n"); });
B.emplace([](){ printf("Taskflow B\n"); });
C.emplace([](){ printf("Taskflow C\n"); });
D.emplace([](){ printf("Taskflow D\n"); });

auto TA = executor.silent_dependent_async(tf::make_module_task(A));
auto TB = executor.silent_dependent_async(tf::make_module_task(B), TA);
auto TC = executor.silent_dependent_async(tf::make_module_task(C), TB);
auto [TD, FD] = executor.dependent_async(tf::make_module_task(D), TC);
FD.get();
```



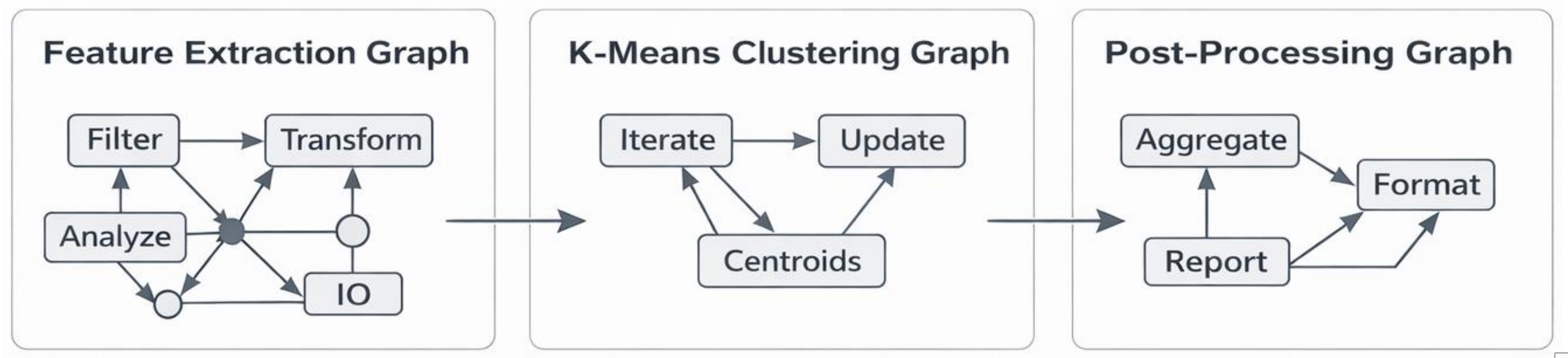
Creates a dynamic task graph where each task encapsulates a task graph

Takeaways

- Understand the importance of task graph composition
- Learn how to compose task graphs in Taskflow
- Showcase two real-world examples of task graph composition
- Conclude

Example: Data Processing Workflow

- **Processes a dataset in three stages:**
  1) Feature extraction: transform raw data into meaningful numerical or categorical features
  2) Data analytics algorithm: apply algorithms such as clustering, classification, or regression
  3) Post-process the result: refine, visualize, or summarize results
- **These are not just different tasks but are different parallel algorithms**
  - Instead of mixing all logic into one large, flat task graph, we can compose three task graphs, each specialized for one stage, so the overall program is easier to test, debug, and optimize

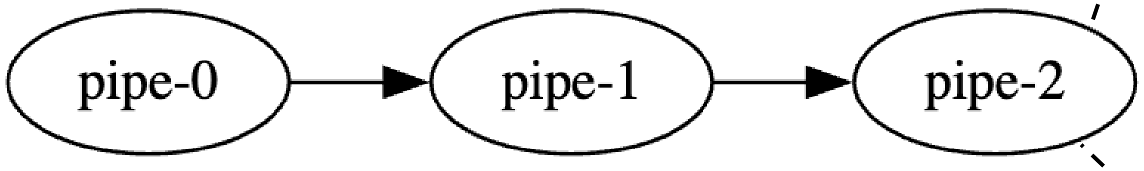


Example: Data Processing Workflow (cont’d)

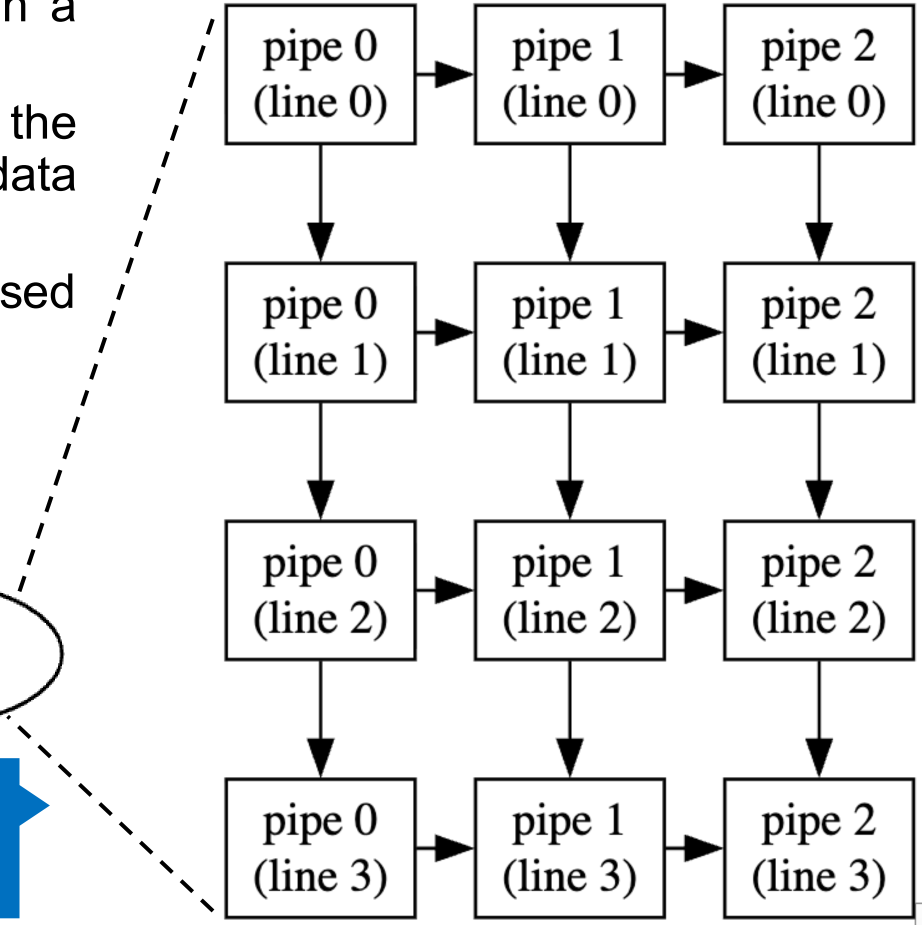
```
tf::Executor executor;
tf::Taskflow feature_extraction, clustering, post_processing, workflow;

// Each stage is implemented as its own task graph, so the implementation and
// integration are modular, self-contained, and focused on a single responsibility
create_feature_extraction_task_graph(feature_extraction);
create_clustering_task_graph(clustering);
create_post_processing_task_graph(post_processing);
auto A = workflow.composed_of(feature_extraction).name(“feature extraction”);
auto B = workflow.composed_of(clustering).name(“clustering”);
auto C = workflow.composed_of(post_processing).name(“post_processing”);
A.preceed(B);
B.preceed(C);

// Executor runs the composed workflow correctly, orchestrating the parallelism
// both inside and outside the three module tasks efficiently
executor.run(workflow).wait();
```

Example: Algorithm Primitive (e.g., Pipeline)

- **What is the pipeline parallelism?**
  - A parallel execution of multiple data tokens through a linear chain of stages or pipes
  - Each stage processes one data token sent from the previous stage, applies the given callable to that data token, and then sends the result to the next stage
  - Parallelism occurs when multiple tokens are processed simultaneously across different stages
- **Example: A three-stage pipeline**
  - Processes four tokens in four parallel lines



Flattened task graph showing four tokens executing over four parallel lines (degree of parallelism).

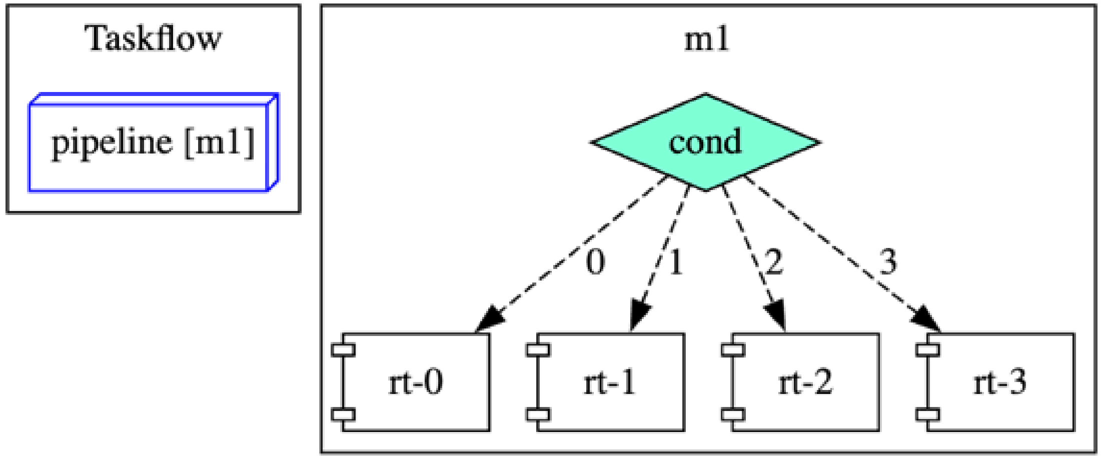


Task-parallel Pipeline Algorithm in Taskflow

- A `tf::Pipeline` is a composable graph for building pipeline parallelism[1]
  - Needs #include<taskflow/algorithm/pipeline.hpp>

```
tf::Taskflow taskflow("pipeline");
tf::Executor executor;

// algorithm interface to create a pipeline task graph
tf::Pipeline pl(num_parallel_lines,
  tf::Pipe{tf::PipeType::SERIAL, [&](tf::Pipeflow& pf){
    // processes only 4 tokens
    if(pf.token() == 4) { pf.stop(); }
    else {
      printf("token=%zu\n", pf.token());
    }
  }},
  … // code to build other stages
);
taskflow.composed_of(pl).name("pipeline");
executor.run(taskflow).wait();
```

All scheduling, synchronization, and execution details are handled automatically by the module task underneath – *separation of concern*



[1]: Task-parallel pipeline algorithm in Taskflow: https://taskflow.github.io/taskflow/TaskParallelPipeline.html

Takeaways

- Understand the importance of task graph composition
- Learn how to compose task graphs in Taskflow
- Showcase two real-world examples of task graph composition
- Conclude

Question?

Static Task Graph Programming (STGP)

```
// Live: https://godbolt.org/z/j8hx3xnnx

tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = taskflow.emplace(
  [](){ std::cout << "TaskA\n"; }
  [](){ std::cout << "TaskB\n"; },
  [](){ std::cout << "TaskC\n"; },
  [](){ std::cout << "TaskD\n"; }
);

A.precede(B, C);
D.succeed(B, C);
executor.run(taskflow).wait();
```

Dynamic Task Graph Programming (DTGP)

```
// Live: https://godbolt.org/z/T87PrTarx

tf::Executor executor;
auto A = executor.silent_dependent_async([]{
  std::cout << "TaskA\n";
});
auto B = executor.silent_dependent_async([]{
  std::cout << "TaskB\n";
}, A);
auto C = executor.silent_dependent_async([]{
  std::cout << "TaskC\n";
}, A);
auto D = executor.silent_dependent_async([]{
  std::cout << "TaskD\n";
}, B, C);
executor.wait_for_all();
```

Taskflow: https://taskflow.github.io