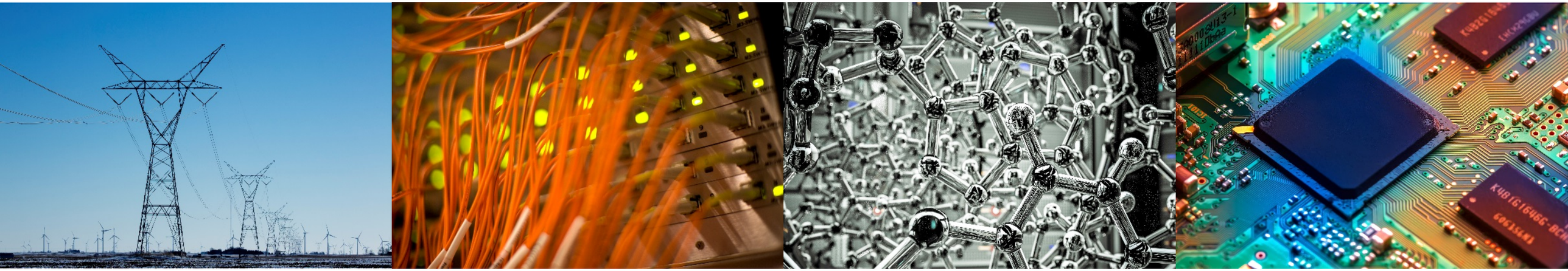
An Efficient and Composable Parallel Task Programming Library

Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, Martin D. F. Wong



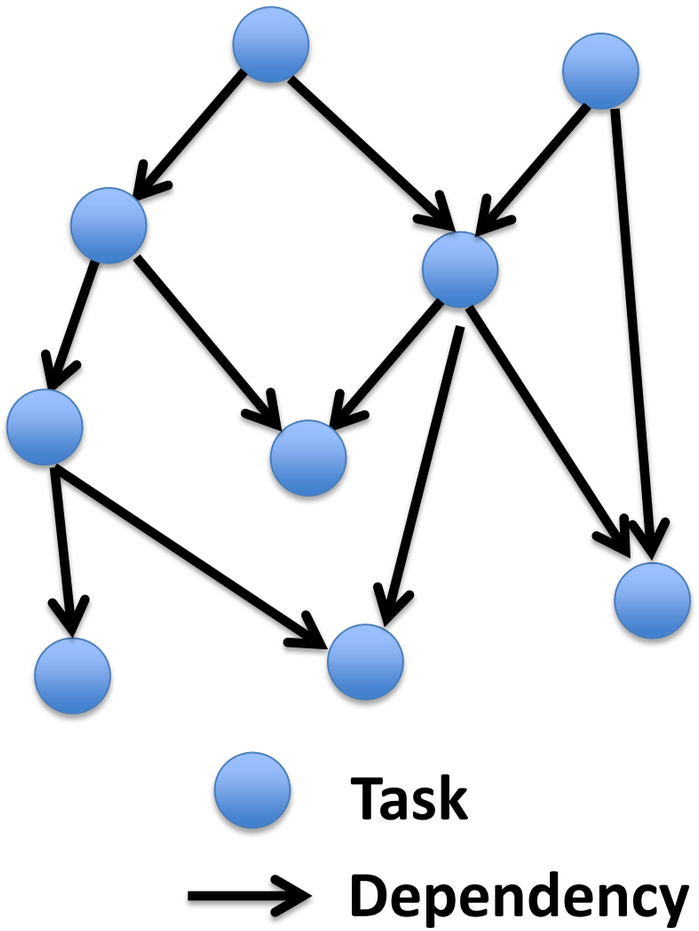
ILLINOIS
Electrical & Computer Engineering
GRAINGER COLLEGE OF ENGINEERING

Outline

- Cpp-Taskflow
- Cpp-Taskflow v2
  - Task dependency graph
    - Separation of graph and executor
    - Composition
    - Modularity
  - Execution methods
- Experimental results

Cpp-Taskflow*

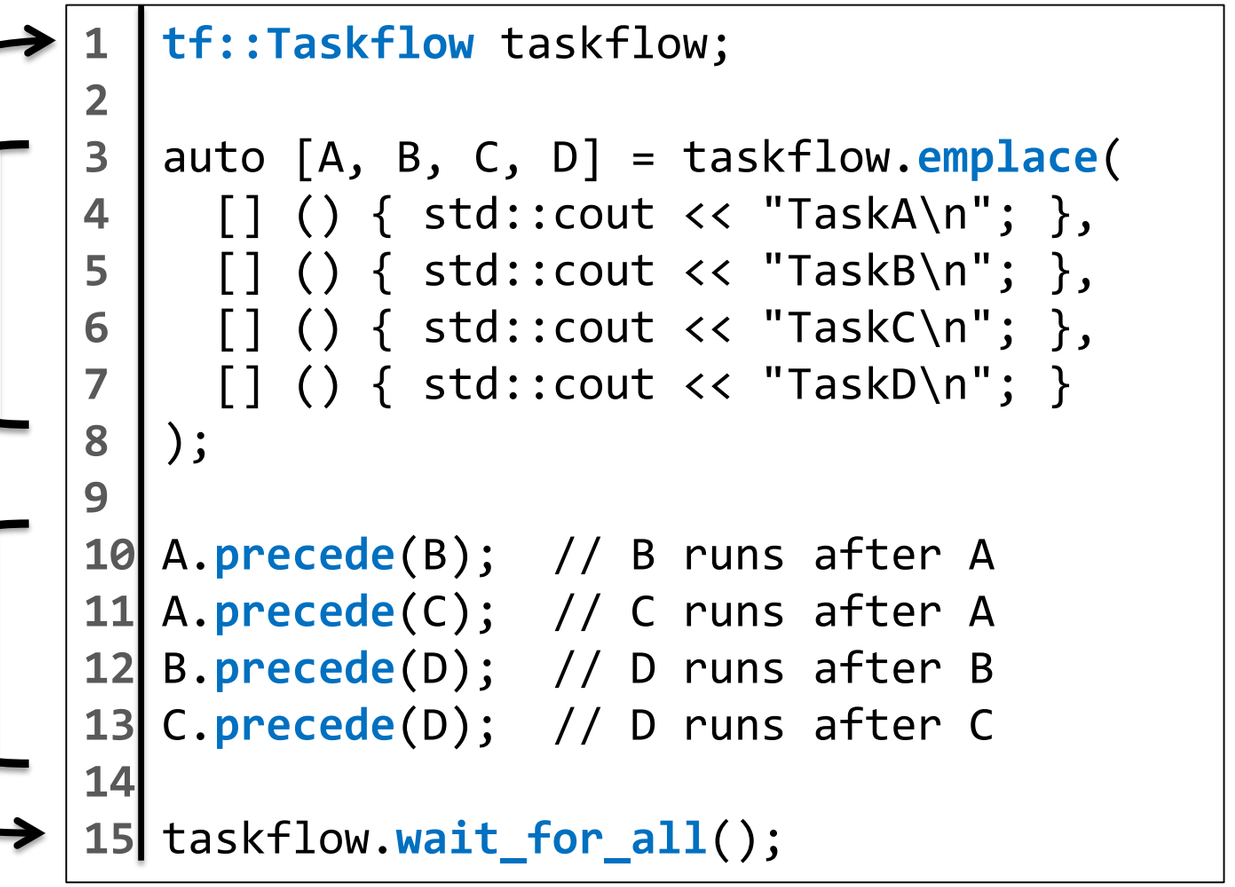
- A C++ task-based programming library
- Task-dependency graph (DAG)
- Static and dynamic tasking
- Simple and unified APIs
- Visualization



** Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++ in IPDPS 2019*

Cpp-Taskflow

- Create a taskflow object
- Add tasks
- Specify dependency
- Run!



```
1  tf::Taskflow taskflow;
2
3  auto [A, B, C, D] = taskflow.emplace(
4    [] () { std::cout << "TaskA\n"; },
5    [] () { std::cout << "TaskB\n"; },
6    [] () { std::cout << "TaskC\n"; },
7    [] () { std::cout << "TaskD\n"; }
8  );
9
10 A.precede(B);  // B runs after A
11 A.precede(C);  // C runs after A
12 B.precede(D);  // D runs after B
13 C.precede(D);  // D runs after C
14
15 taskflow.wait_for_all();
```

Cpp-Taskflow

- Dynamic tasking
- Same API as static tasking
- Detach or join mode

```
1  tf::Taskflow taskflow;
2
3  auto [A, B, C] = taskflow.emplace(
4     [] () { std::cout << "TaskA\n"; },
5     [] () { std::cout << "TaskB\n"; },
6     [] (auto &subflow) {
7       auto C1 = subflow.emplace([](){});
8       auto C2 = subflow.emplace([](){});
9       C1.precede(C2);   // C2 runs after C1
10    }
11 );
12
13 A.precede(B);  // B runs after A
14 A.precede(C);  // C runs after A
15
16 taskflow.wait_for_all();
```

Cpp-Taskflow

- Easy to use
- No complex concurrency control
- Static and dynamic tasking

- Issues:
  - A task graph can only be executed once
  - No **separate** task graphs. All task graphs are stored in a single taskflow object.

Cpp-Taskflow v2

- Separate executor and task graph

Make sure task graphs persist during execution

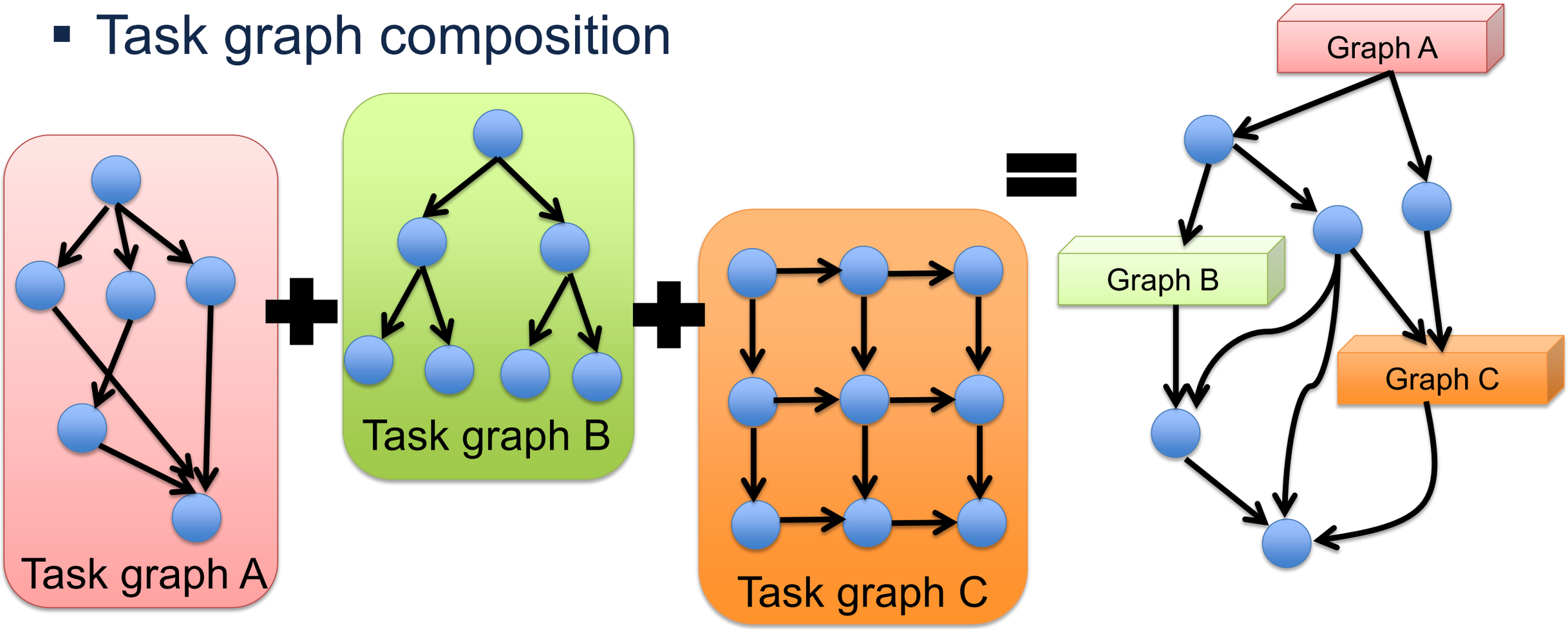
```
1  tf::Taskflow flow;
2  auto t1 = flow.emplace([](){});
3  auto t2 = flow.emplace([](){});
4  t1.precede(t2);
5
6  tf::Taskflow flow2;
7  auto t3 = flow2.emplace([](){});
8  auto t4 = flow2.emplace([](){});
9  t3.precede(t4);
10
11 tf::Executor executor;
12 executor.run(flow);
13 executor.run(flow2);
14 executor.wait_for_all();
```

Create task graphs independent to each other

Create an executor to run the graph in any order!

Cpp-Taskflow v2

- Task graph composition



Cpp-Taskflow v2

- Task graph composition
- API: **composed_of**
- Seamless integration with existing task interface
- A task graph can be composed many times.

```
1  tf::Taskflow flow;
2  auto t1 = flow.emplace([](){});
3  auto t2 = flow.emplace([](){});
4  t1.precede(t2);
5
6  tf::Taskflow flow2;
7  auto t3 = flow2.emplace([](){});
8  auto t4 = flow2.emplace([](){});
9
10 // Composition
11 auto t5 = flow2.composed_of(flow);
12 t5.precede(t3, t4);
13
14 tf::Executor executor;
15 executor.run(flow2).wait();
```

Cpp-Taskflow v2

- Modularity
- Object-oriented design
- Inheritance from taskflow class
- Same tasking APIs

```
1  struct MyGraph: tf::Taskflow {
2    // Define your own graph
3  }
4
5  MyGraph G;
6
7  auto t1 = G.emplace([](){});
8  auto t2 = G.emplace([](){});
9  t1.precede(t2);
10
11 tf::Executor executor;
12 executor.run(G).wait();
13
```

Cpp-Taskflow v2

- Task graph is reusable
- A rich set of execution methods
  - Run once
  - Run multiple times
  - Run until a predicate is met
  - Block until all graphs finish
  - All methods return a future object

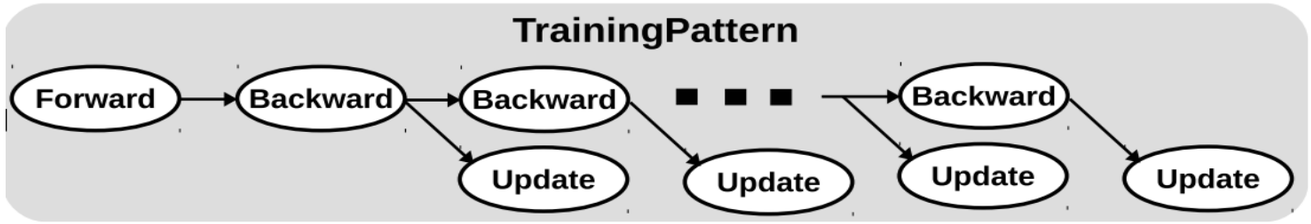
```
1  tf::Taskflow flow;
2  auto t1 = flow.emplace([](){});
3  auto t2 = flow.emplace([](){});
4  t1.precede(t2);
5
6  tf::Executor executor;
7  // Run once
8  executor.run(flow).wait();
9
10 // Run five times
11 executor.run_n(flow, 5).wait();
12
13 // Run until a predicate is met
14 executor.run_until(flow,
15  [counter=4](){ return --counter == 0;}
16 ).wait();
```

Experiments

- A machine learning example
- A VLSI timing example
- Platform
  - 2.5 GHz Intel Xeon W-2175 with 14 cores
  - OS: Ubuntu 18.04
  - Compiler: GCC 7.4
- Comparison: Intel Threading Building Blocks Flow Graph

Experiments

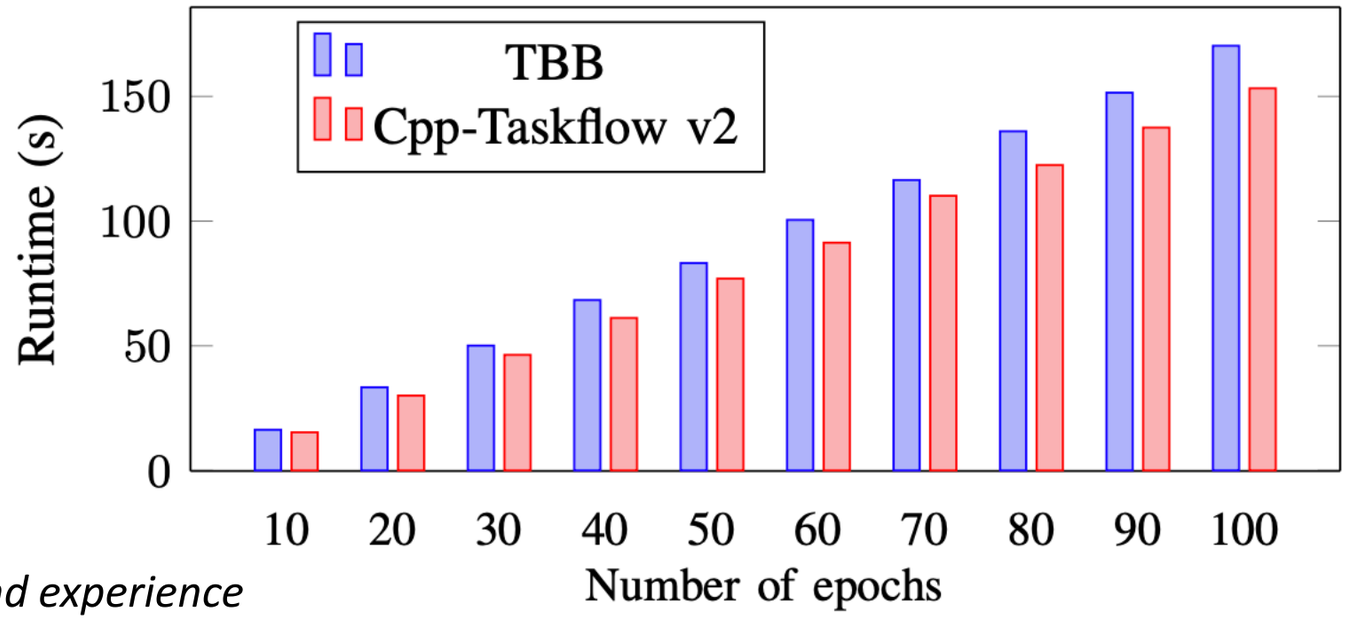
- Training 10 DNNs for MNIST dataset
- Task pipelining
- Use 10 cores



Library	NLOC (total)	CCN (avg)	Token (avg)
Cpp-Taskflow v2	60	2.0	90.6
TBB	77	2.8	125.0

NLOC: lines of code. **CCN**: cyclomatic complexity number.

Code complexity measured by Lizard*



**C++ software quality in the ATLAS experiment: tools and experience*

Experiments

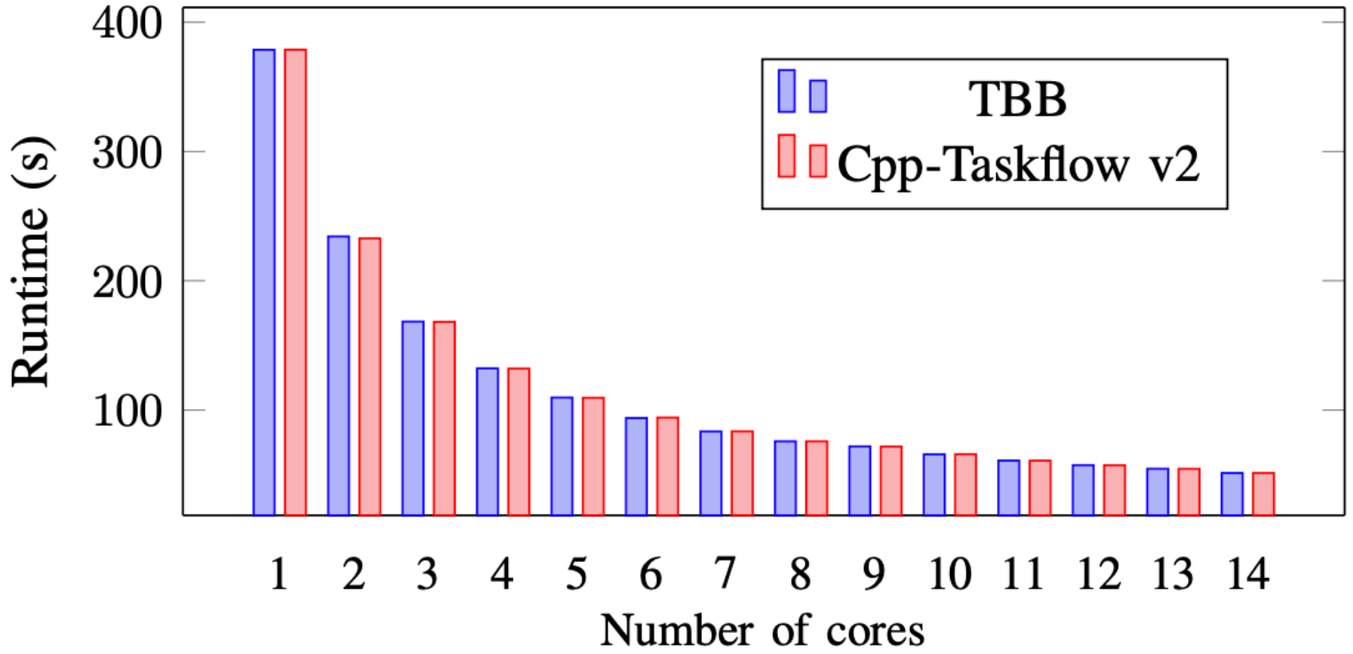
- VLSI timing analysis
- Launch multiple timers (*fork*) to do timing analysis and then find the correlations between the results

Library	NLOC (total)	CCN (avg)	Token (avg)
Cpp-Taskflow v2	61	3.0	132
TBB	104	2.6	106.6

Framework	NLOC (total)		CCN (avg)		Token (avg)	
	TF	TBB	TF	TBB	TF	TBB
Process	10	14	2	2	118	155
Simulation	5	9	2	2	47	63
Scenario	15	16	4	4	137	155
Top	23	29	4	4	226	283

NLOC: lines of code. **CCN**: cyclomatic complexity number.

Code complexity measured by Lizard



Thank You!

GitHub: https://github.com/cpp-taskflow/cpp-taskflow

Tutorial: https://cpp-taskflow.github.io/

Documentation: https://cpp-taskflow.github.io/cpp-taskflow/