Control Taskflow Graph (CTFG) Programming Model

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

Takeaways

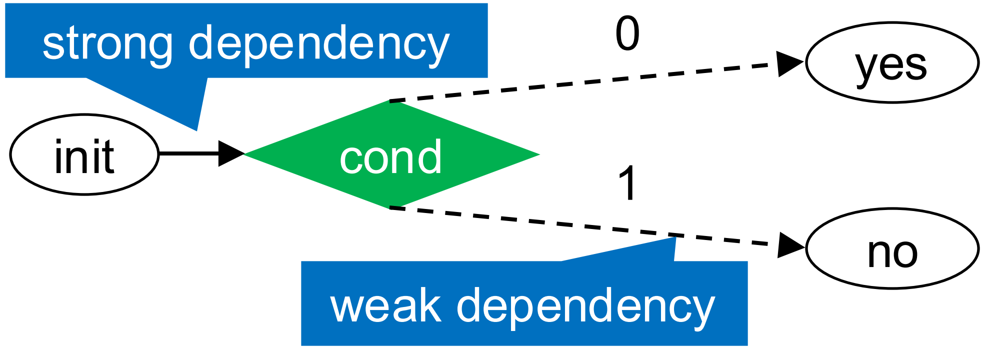
- Learn the control Taskflow graph (CTFG) programming model
- Compare CTFG with existing models and recognize their limitations
- Understand the scheduling flow of a CTFG
- Showcase some real-world applications of CTFG
- Conclude the talk

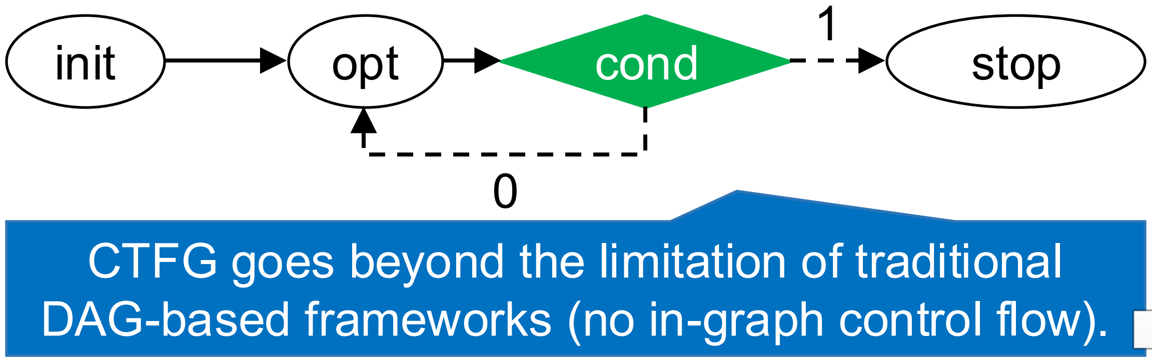
Control Taskflow Graph (CTFG) Programming Model

- **A key innovation that distinguishes Taskflow from existing models**

```
auto [init, cond, yes, no] =
taskflow.emplace(
  [] () { },
  [] () { return 0; },
  [] () { std::cout << “yes”; },
  [] () { std::cout << “no”; }
);
cond.succeed(init)
    .precede(yes, no);
```

```
auto [init, opt, cond, stop] =
taskflow.emplace(
  [&](){ initialize_data_structure(); },
  [&](){ some_optimizer(); },
  [&](){ return converged() ? 1 : 0; },
  [&](){ std::cout << “done!\n”; }
);
opt.succeed(init).precede(cond);
converged.precede(opt, stop);
```



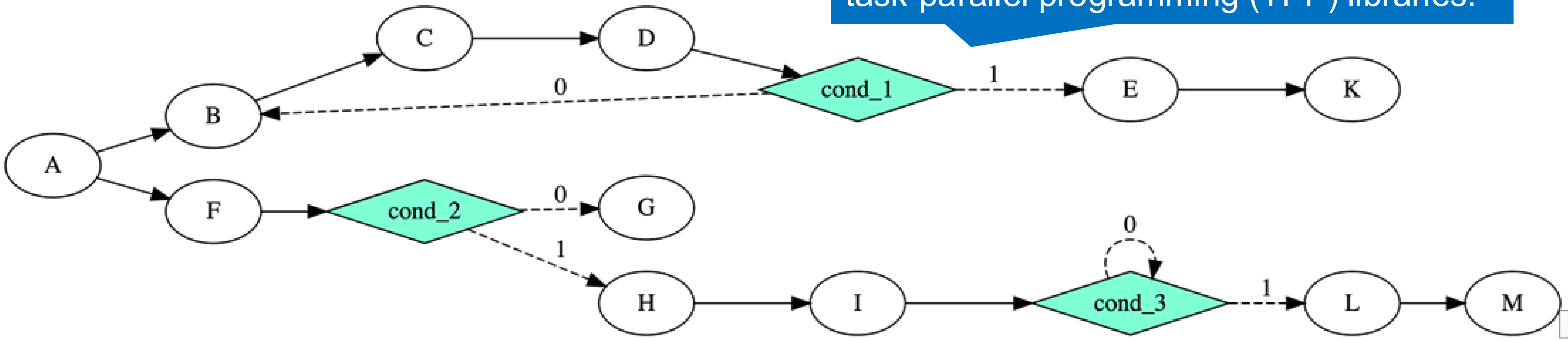


Power of CTFG Programming Model

- **Enables very efficient overlap among tasks and control flow**

```
auto cond_1 = taskflow.emplace([](){ return run_B() ? 0 : 1; });
auto cond_2 = taskflow.emplace([](){ return run_G() ? 0 : 1; });
auto cond_3 = taskflow.emplace([](){ return loop() ? 0 : 1; });
cond_1.precede(B, E);
cond_2.precede(G, H);
cond_3.precede(cond_3, L);
```

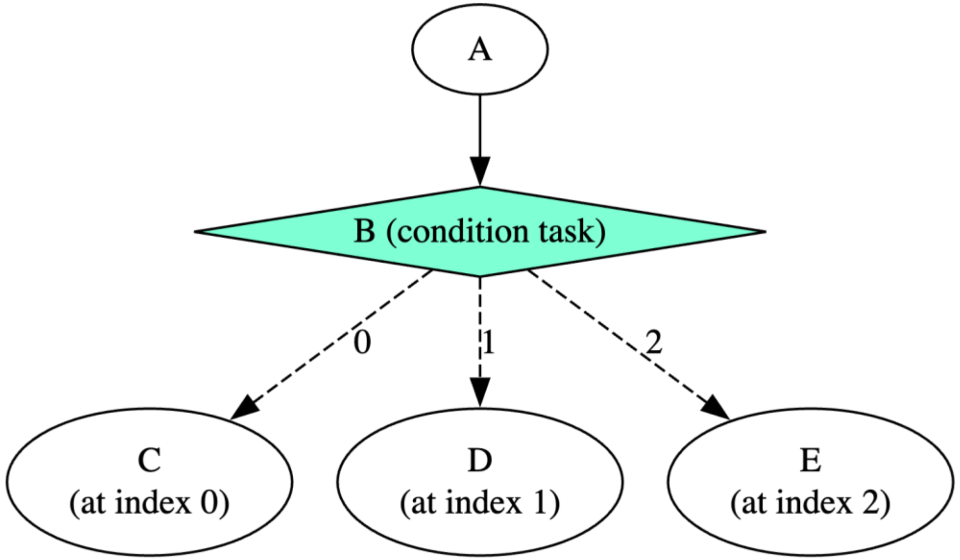
This type of end-to-end parallelism is almost impossible to achieve using existing task-parallel programming (TPP) libraries.



Condition Task

- **A callable that returns an integer indicating which successor task to run**
  - The returned integer represents the index of the successor task in the dependency list

```
tf::Taskflow taskflow;
auto [A, B, C, D, E] = taskflow.emplace(
  [&](){ std::cout << "A\n"; },
  [&]() -> int {
    std::cout << "B is a condition task\n";
    return 2;
  },  // B is a condition task
  [&](){ std::cout << "C\n"; },
  [&](){ std::cout << "D\n"; },
  [&](){ std::cout << "E\n"; }
);
A.precede(B);
B.precede(C, D, E);
```



If the returned integer index is beyond the valid successor range (e.g., `return 100`), the scheduler will not insert any task to run.

Iterative Control Flow

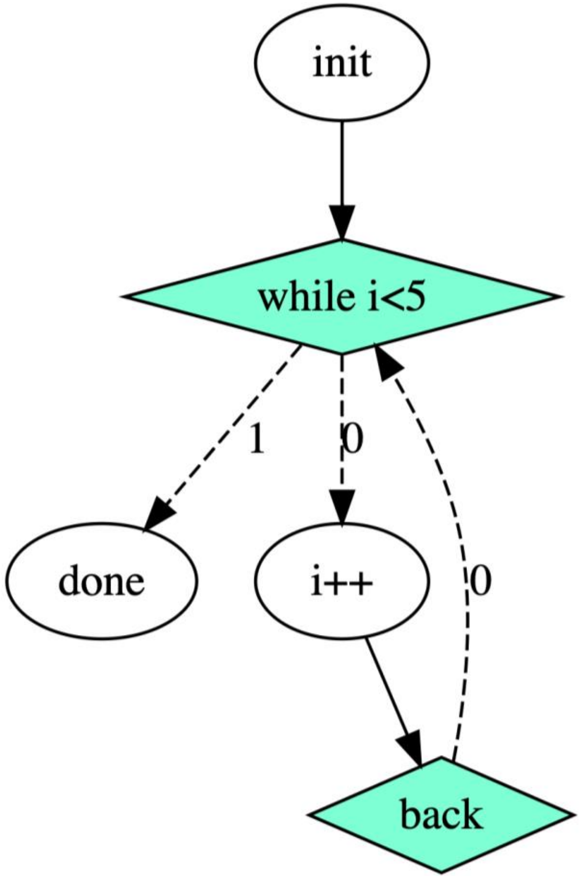
- **Use condition tasks to build an iterative control flow within a task graph**

```
tf::Taskflow taskflow;

int i;
auto [init, cond, body, back, done] = taskflow.emplace(
  [&]{ std::cout << “i=0”; i=0; },
  [&]{ std::cout << "while i<5\n"; return i < 5 ? 0 : 1; },
  [&]{ std::cout << "i++=" << i++ << '\n'; },
  [&]{ std::cout << "back\n"; return 0; },
  [&]{ std::cout << "done\n"; }
);

init.precede(cond);        // init
cond.precede(body, done); // while i<5
body.precede(back);        // i++
back.precede(cond);        // back
```

The cond task acts as the *while condition*, checking if *i* is less than 5.

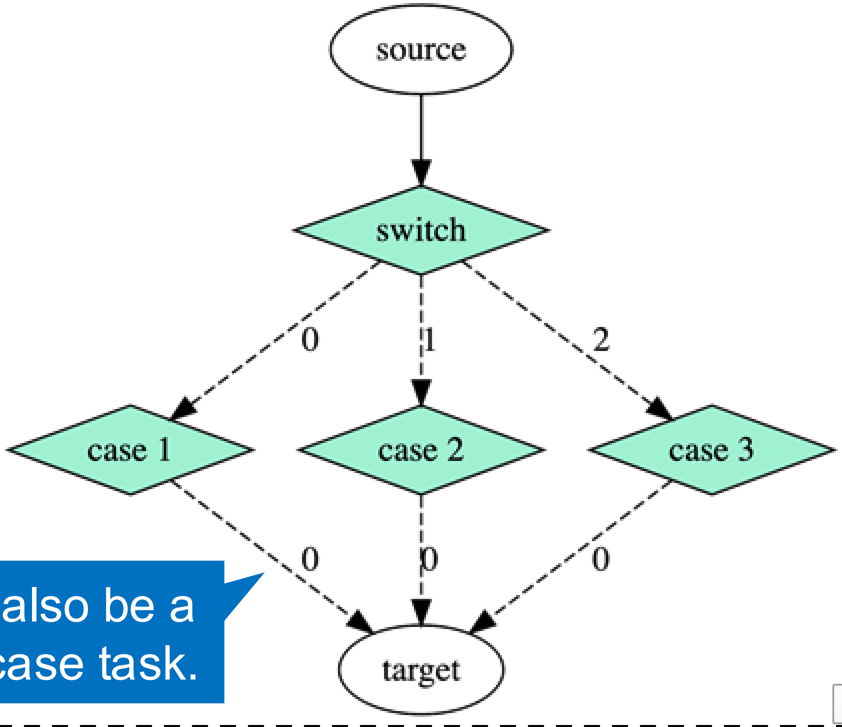


Switch-Case Control Flow

- **Use condition tasks to build a switch-case control flow within a task graph**

```
auto [source, swcond, case1, case2, case3, target] = taskflow.emplace(
  [](){ std::cout << "source\n"; },
  [](){ std::cout << "switch\n"; return rand()%3; },
  [](){ std::cout << "case 1\n"; return 0; },
  [](){ std::cout << "case 2\n"; return 0; },
  [](){ std::cout << "case 3\n"; return 0; },
  [](){ std::cout << "target\n"; }
);

source.precede(swcond);
swcond.precede(case1, case2, case3);
target.succeed(case1, case2, case3);
```



The three case tasks (`case 1`, `case 2`, `case 3`) must also be a condition task because `target` depends only on one case task.

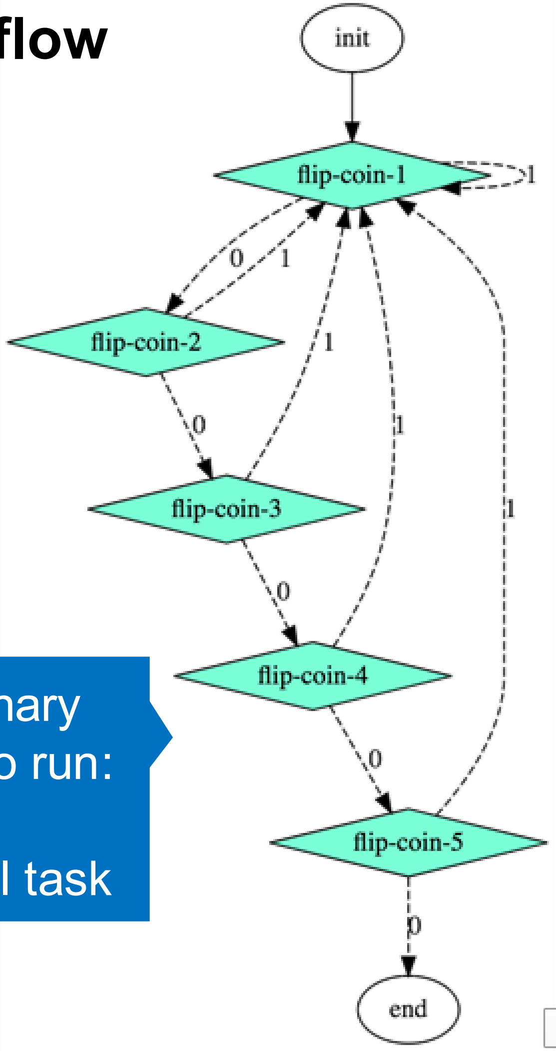
Condition Tasks can Handle Complicated Scenarios

- **Use condition tasks to build a non-deterministic control flow**

```
auto A = taskflow.emplace([&](){ });
auto B = taskflow.emplace([&](){ return rand()%2; } );
auto C = taskflow.emplace([&](){ return rand()%2; } );
auto D = taskflow.emplace([&](){ return rand()%2; } );
auto E = taskflow.emplace([&](){ return rand()%2; } );
auto F = taskflow.emplace([&](){ return rand()%2; } );
auto G = taskflow.emplace([&](){});

A.precede(B).name("init");
B.precede(C, B).name("flip-coin-1");
C.precede(D, B).name("flip-coin-2");
D.precede(E, B).name("flip-coin-3");
E.precede(F, B).name("flip-coin-4");
F.precede(G, B).name("flip-coin-5");
G.name(“end”);
```

Each condition task flips a binary coin to decide the next task to run:
0: goes to the next task
1: goes back to the original task



Multi-condition Task

- **A generalized version of condition task to jump to multiple successor tasks**

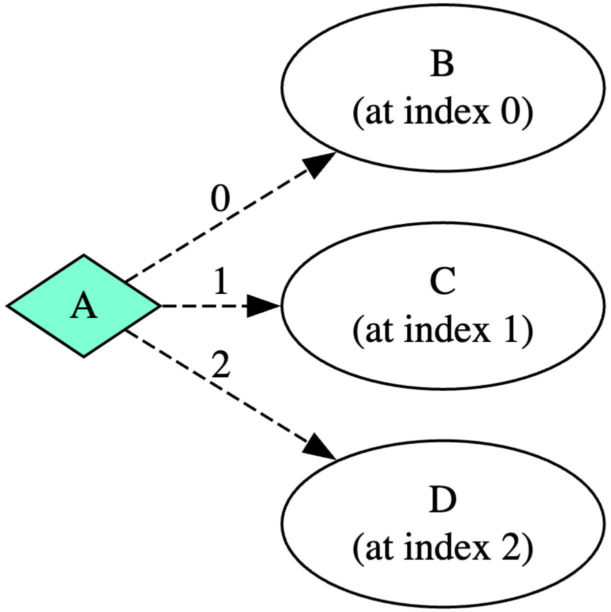
```
tf::Executor executor;
tf::Taskflow taskflow;

auto A = taskflow.emplace([&]() -> tf::SmallVector<int> {
  std::cout << "A\n";
  return {0, 2};
});
auto B = taskflow.emplace([&](){ std::cout << "B\n"; });
auto C = taskflow.emplace([&](){ std::cout << "C\n"; });
auto D = taskflow.emplace([&](){ std::cout << "D\n"; });

A.precede(B, C, D);

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

`tf::SmallVector<int>` is similar to `std::vector` but comes with small buffer optimization.



If the returned integer index is beyond valid successor range (e.g., `return -1`), the scheduler will not insert any task to run.

Takeaways

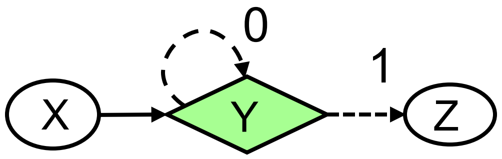
- Learn the control Taskflow graph (CTFG) programming model
- Compare CTFG with existing models and recognize their limitations
- Understand the scheduling flow of a CTFG
- Showcase some real-world applications of CTFG
- Conclude the talk

How Do Existing TPP Libraries Handle Control Flow?

- **Most existing TPP libraries rely on Directed Acyclic Graph (DAG) models**
  - Do not natively support conditional execution or iterative execution (e.g., cycle)
  - As a result, all the control-flow decision making must happen outside the task graph
- **A common approach or workaround is as follows:**
  - You manually partition the task graph around control-flow points – *out-of-graph control flow*
  - Then, each of those partitioned graphs synchronizes at the control-flow boundaries
  - This approach works just fine but not optimal, as it limits **end-to-end parallelism** and increases user-side code complexity

```
tf::Taskflow G;
auto X = G.emplace([](){});
auto Y = G.emplace([](){
  return converged() ? 1 : 0;
});
Y.precede(Y, Z).succeed(X);
executor.run(G).wait();
```

Manually partition the task graph using a do-while loop (out-of-graph control flow)



Iterative synchronization cost at the end of each iteration.

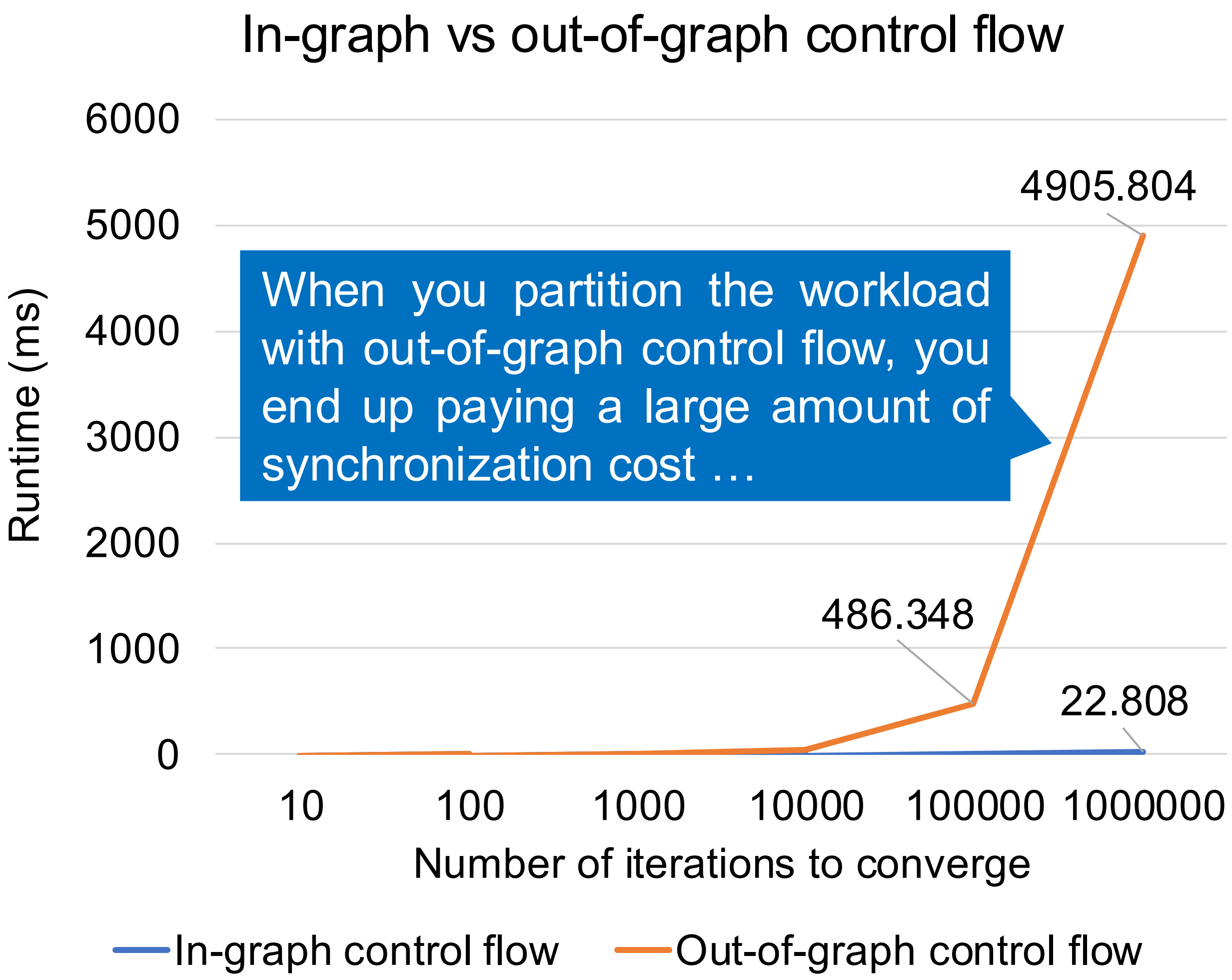
```
tbb::flow::graph[1] X, Y;
tbb::flow::graph Z;
X.run();        // sync
do {
  Y.run();      // sync
} while (!converged());
Z.run();
```

[1]: oneAPI Threading Building Blocks (oneTBB): https://github.com/uxlfoundation/oneTBB

Performance: In-graph vs Out-of-graph Control Flow

```
// in-graph control flow (CTFG)
tf::Taskflow G;
auto X = G.emplace([](){});
auto Y = G.emplace([](){
  return converged() ? 1 : 0;
});
Y.precede(Y, Z).succeed(X);
executor.run(G).wait();
```

```
// out-of-graph control flow
tf::Taskflow X, Y, Z;
executor.run(X).wait();  // sync
do {
  executor.run(Y).wait();// sync
} while (!converged());
executor.run(Z).wait();
```



When you partition the workload with out-of-graph control flow, you end up paying a large amount of synchronization cost …

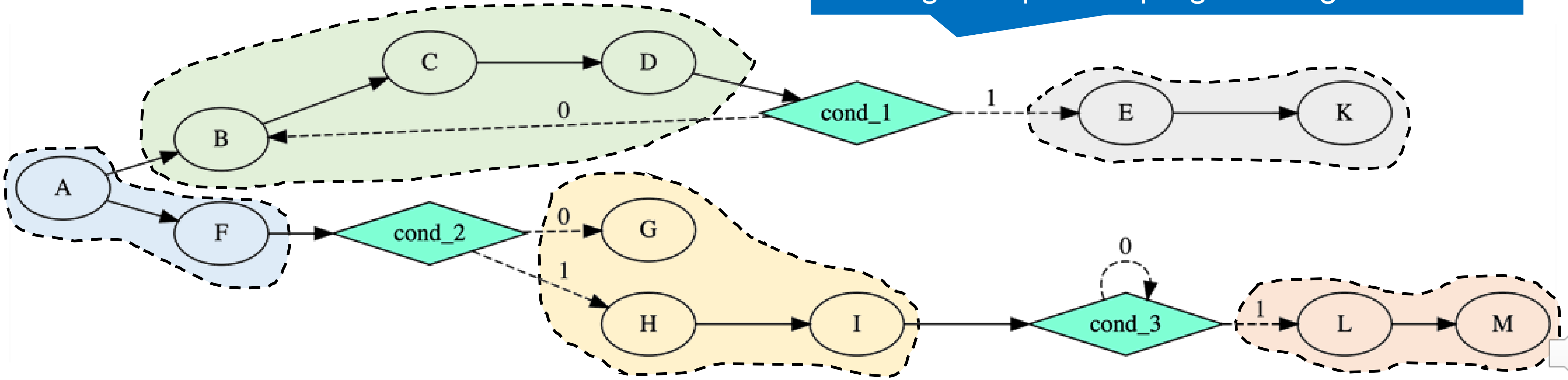
[1]: In-graph control flow vs Out-of-graph control flow implementations: https://github.com/taskflow/taskflow/blob/master/examples/for_loop.cpp

How to Partition this Task Graph?

- **Three condition tasks (cond_1, cond_2, cond_3) within a task graph**

```
auto cond_1 = taskflow.emplace([](){ return run_B() ? 0 : 1; });
auto cond_2 = taskflow.emplace([](){ return run_G() ? 0 : 1; });
auto cond_3 = taskflow.emplace([](){ return loop() ? 0 : 1; });
cond_1.precede(B, E);
cond_2.precede(G, H);
cond_3.precede(cond_3, L);
```

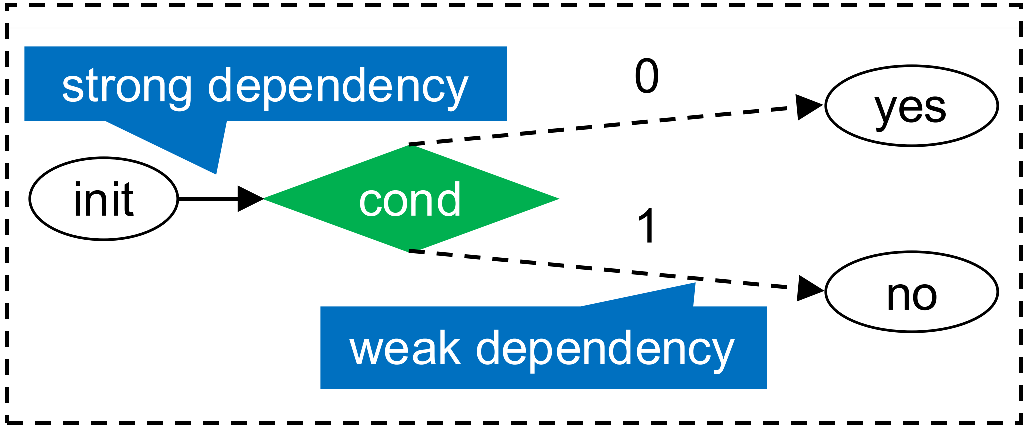
Again, without CTFG, this type of end-to-end parallelism is very difficult to achieve using existing task-parallel programming libraries.

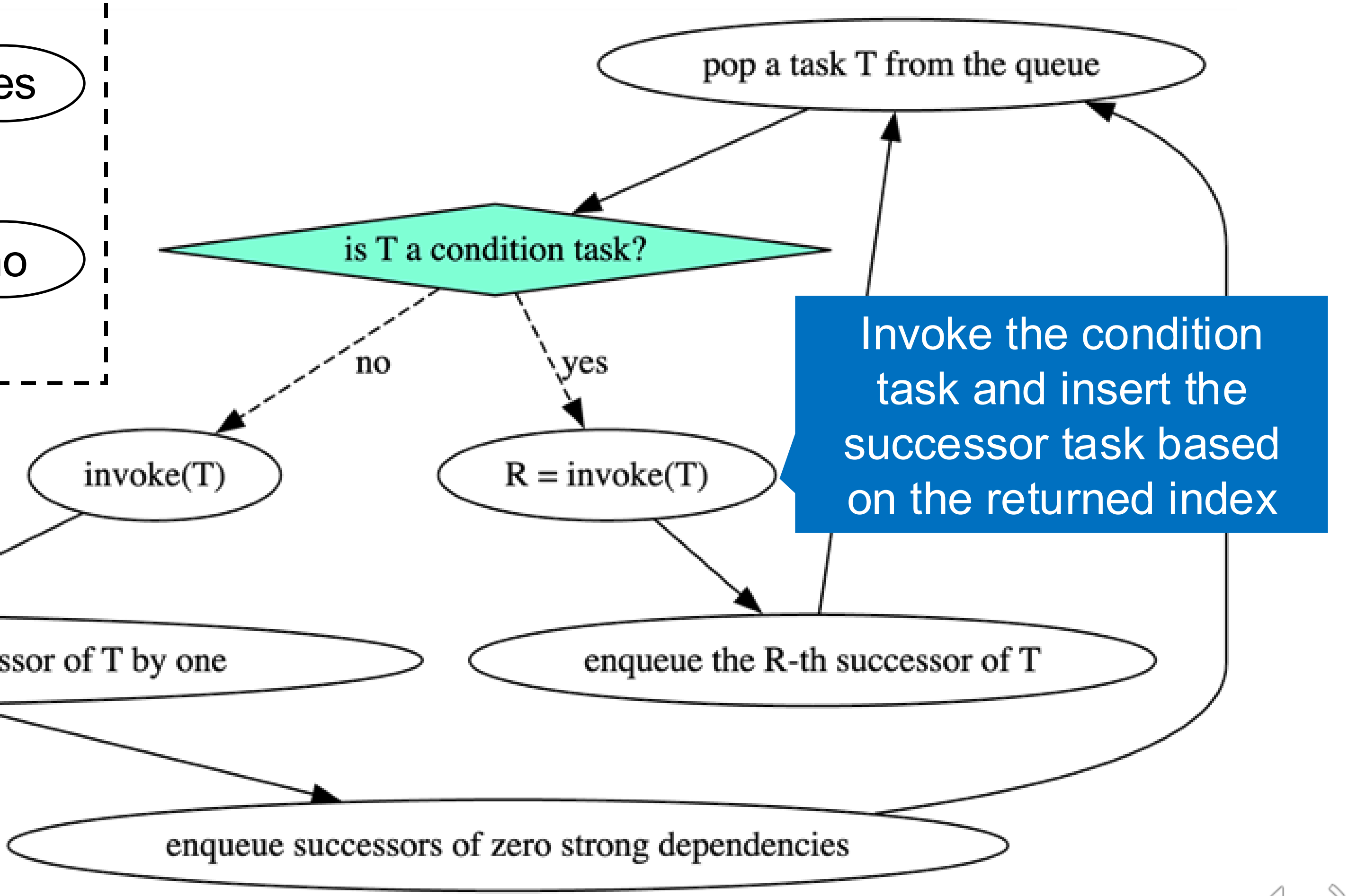


Takeaways

- Learn the control Taskflow graph (CTFG) programming model
- Compare CTFG with existing models and recognize their limitations
- Understand the scheduling flow of a CTFG
- Showcase some real-world applications of CTFG
- Conclude the talk

Scheduling Flow of a Control Taskflow Graph

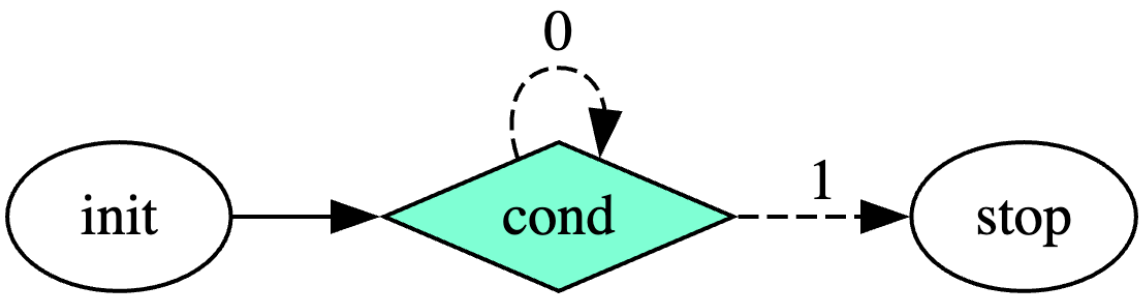




Taskflow always starts with tasks that have zero strong and weak dependencies.

Example: In-graph Iterative Control flow

1. **Starts with `init` because it has zero strong and weak dependencies**
   - If your CTFG doesn't have any zero-dependency tasks, it won't be schedulable
2. **Moves on to the condition task cond**
   - If `cond` returns 0, the scheduler enqueues `cond` and runs it again
   - If `cond` returns 1, the scheduler enqueues `stop` and then moves on



Taskflow always starts with tasks that have zero strong and weak dependencies (i.e., `init` in this example).

task	# strong dep	# weak dep
`init`	0	0
`cond`	1	1
`stop`	0	1

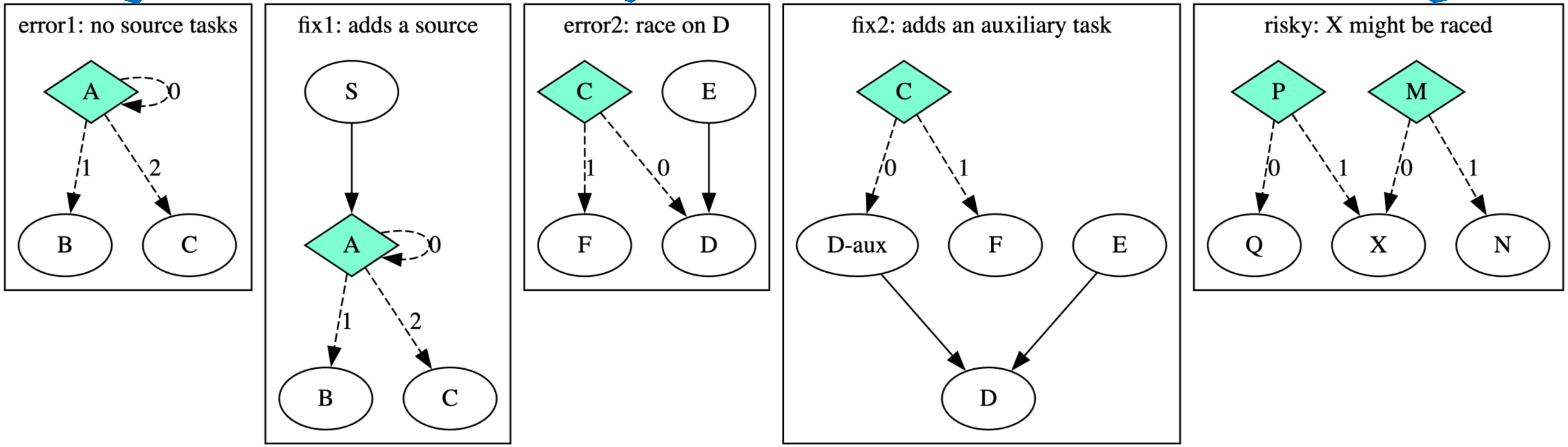
Conditional Tasking can be Easy to Make Mistakes!

- **It is your responsibility to ensure the given CTFG is valid**
  - A valid CTFG is a task graph that is schedulable and race-free (no task race)
- **Some common pitfalls in CTFG programming below:**

Not schedulable (#1 mistake in CTFG)

Task race occurs on `D` when `C` returns 0

`P` and `M` must not return 1 and 0

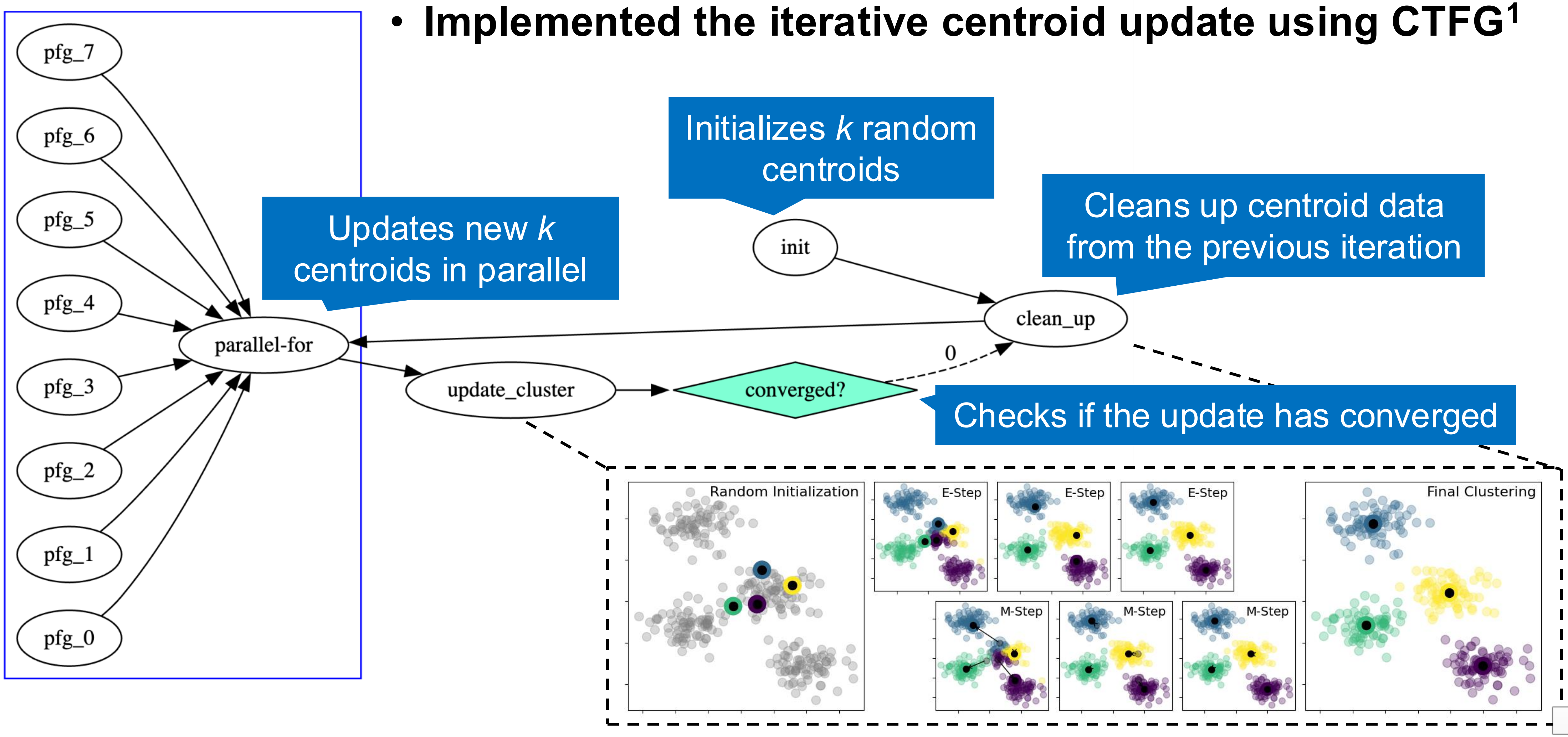


Takeaways

- Learn the control Taskflow graph (CTFG) programming model
- Compare CTFG with existing models and recognize their limitations
- Understand the scheduling flow of a CTFG
- Showcase some real-world applications of CTFG
- Conclude the talk

Use Case #1: K-means Clustering

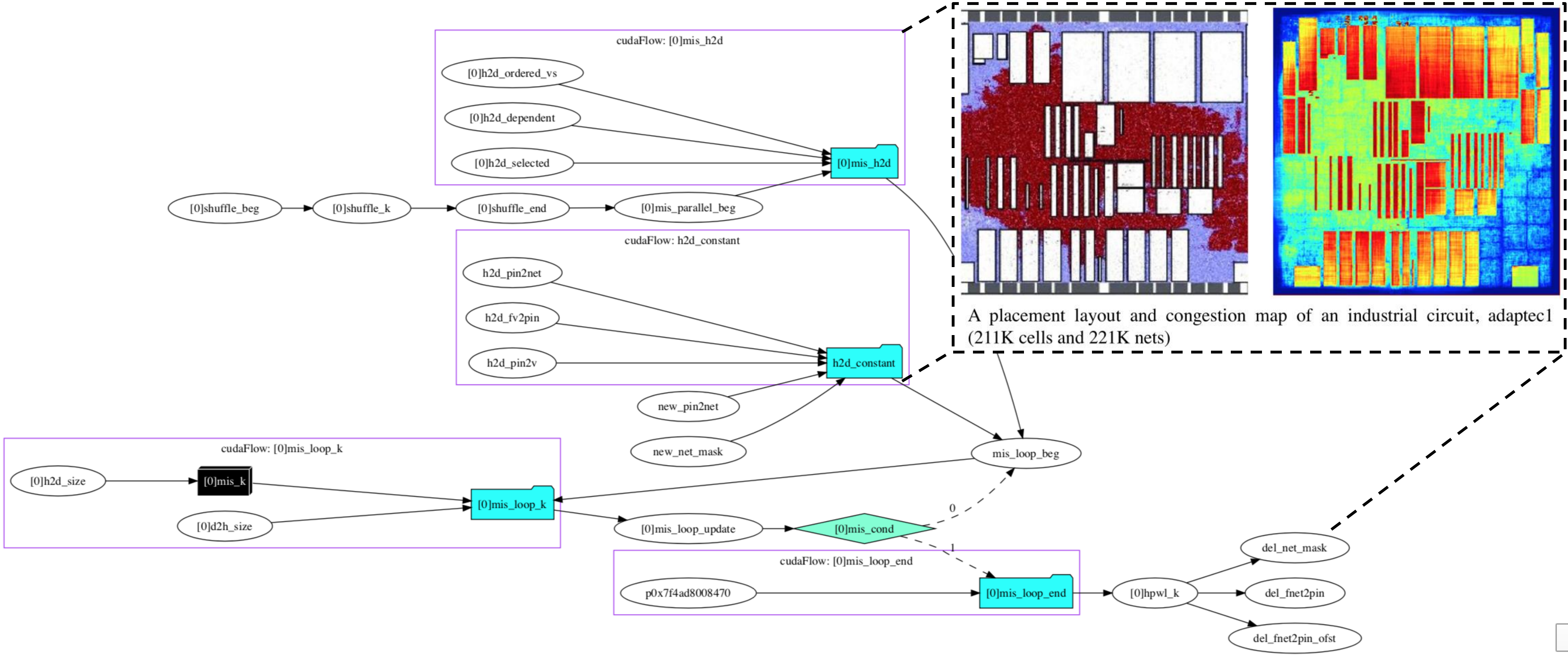
- **Implemented the iterative centroid update using CTFG[1]**



[1]: Taskflow Handbook "k-means clustering example": https://taskflow.github.io/taskflow/kmeans.html

Use Case #2: VLSI Detailed Placement

- **Implemented an iterative gate location update algorithm using CTFG[1]**



A placement layout and congestion map of an industrial circuit, adaptec1 (211K cells and 221K nets)

[1]: Tsung-Wei Huang, et. al, “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System,” *IEEE TPDS*, 2022

Use Case #3: Robotics Processing Planning

TESSERACT

Graph Taskflow Generator Revision #17

Merged **Levi-Armstrong** merged 7 commits into `tesseract-robotics:master` from `marip8:update/graph_taskf`

Conversation 28 | Commits 7 | Checks 0 | Files changed 5

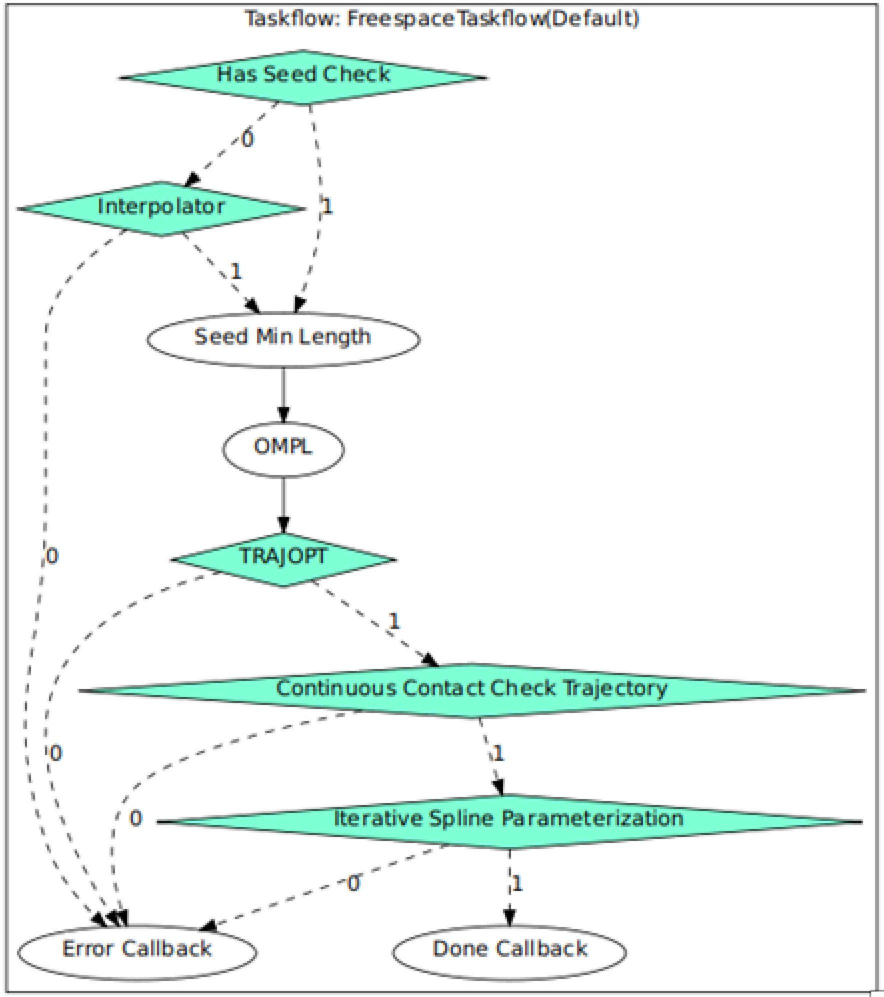
marip8 commented on Jan 19, 2021 • edited — Contributor

This PR revises the graph Taskflow generator class to be easier to use.

Assumptions

- End conditions
  - The required error task is only applicable to conditional nodes. I assign the error task as the connection at index 0 to all conditional tasks
  - The done callback is applicable to both conditional and non-conditional nodes.
    - If a conditional node has any edges, I do not connect that node to the done callback
    - If a conditional node is a leaf node (i.e. has no defined edges), the connection at index 0 is to the error task (as mentioned above) and the connection at index 1 is to the done task
    - If a non-conditional node is a leaf node, it's only connection is to the done callback
- No graph checking is implemented for loops and "island" states. I think we should allow users the freedom to define their graphs in whatever way possible. It's easy enough to dump the taskflow to a file and understand if the graph looks correct

All of these assumptions should be documented in the header file



[1]: Tesseract Planning Software: https://github.com/tesseract-robotics/tesseract_planning/pull/17

Takeaways

- Learn the control Taskflow graph (CTFG) programming model
- Compare CTFG with existing models and recognize their limitations
- Understand the scheduling flow of a CTFG
- Showcase some real-world applications of CTFG
- Conclude the talk

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();
```

Taskflow: https://taskflow.github.io

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();
```