Static Task Graph Programming (STGP) in Taskflow

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

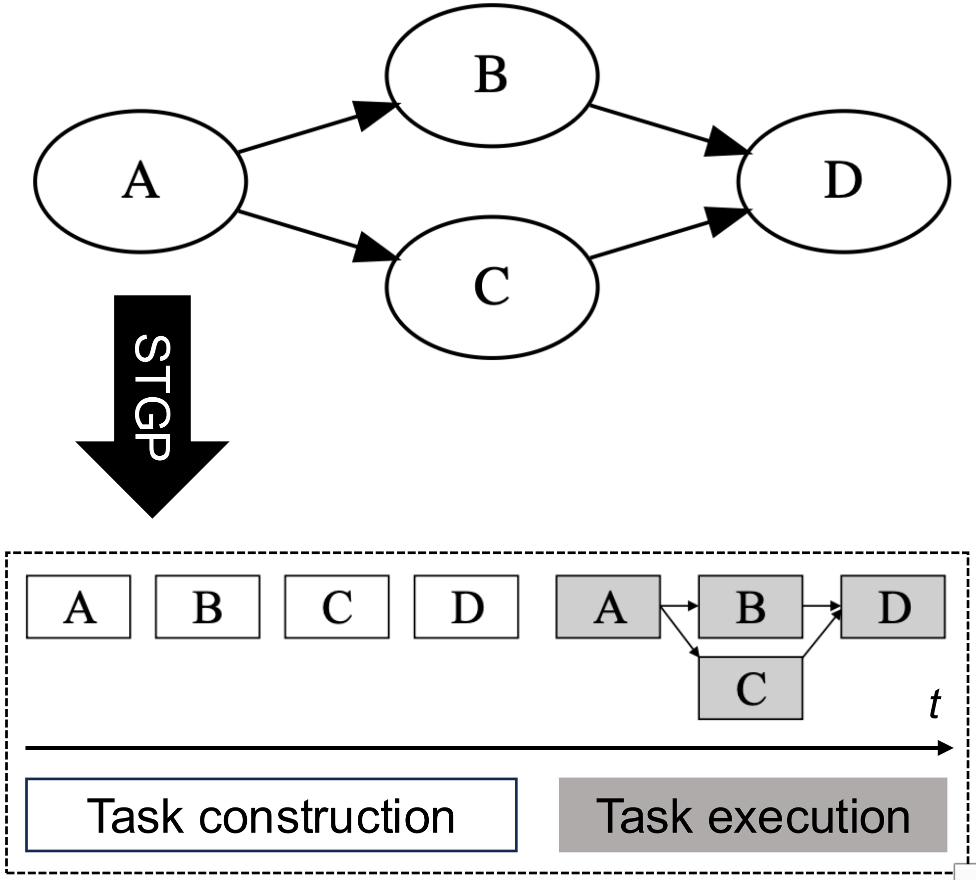
Takeaways

- Learn how to program static task graph parallelism in Taskflow
- Understand the design philosophy behind Taskflow
- Showcase a real-world application of static task graph programming
- Conclude the talk

Static Task Graph Programming (STGP) in Taskflow

```
#include <taskflow/taskflow.hpp>  // Live: https://godbolt.org/z/j8hx3xnnx

int main(){
  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();
  return 0;
}
```



A Task in Taskflow is a Callable Object

- **Anything for which the operation `std::invoke`[1] is applicable**
  - Lambda expression (recommended), functor, function pointer, bind expression, etc.
- **Two major methods in Taskflow for creating a task**
  - `tf::Taskflow::placeholder` – creates a placeholder task whose work can be assigned later
  - `tf::Taskflow::emplace` – creates a task with work assigned immediately upon creation

```
tf::Taskflow taskflow;

tf::Task A = taskflow.placeholder();
tf::Task B = taskflow.emplace([](){ std::cout << "task B\n"; });

A.precede(B);
```

Task `A` is a placeholder task and does not have any callable assigned to run.

Task `B` will run the assigned callable.

Notice that a placeholder task is a valid task and occupies a node in a task graph. Soon after its creation, you can use it right away to build dependencies with other tasks.

[1]: C++ `std::invoke`: https://en.cppreference.com/w/cpp/utility/functional/invoke.html

The `tf::Task` Handle

- **A copy-cheap wrapper over the node pointer to a task in taskflow**
- **Provides a set of methods to access and modify the task attributes**
  - Building dependencies, assigning a name, changing the work, querying the statistics, etc.

```
// for each task, taskflow creates a node in the graph and returns a task handle
tf::Task A = taskflow.placeholder();
tf::Task B = taskflow.emplace([](){ std::cout << "task B\n"; });

A.name("Task1")                                  // assign a name to task A
 .precede(B);                                    // assign a dependency to task B

std::cout << A.name() << '\n';                   // query the name of task A
std::cout << A.num_successors() << '\n';         // query the # of successors of A
std::cout << A.num_predecessors() << '\n';       // query the # of predecessors of A

// assign a work to replace the placeholder with a callable
A.work([](){ std::cout << "task A\n"; });
```

Traverse a Taskflow Graph

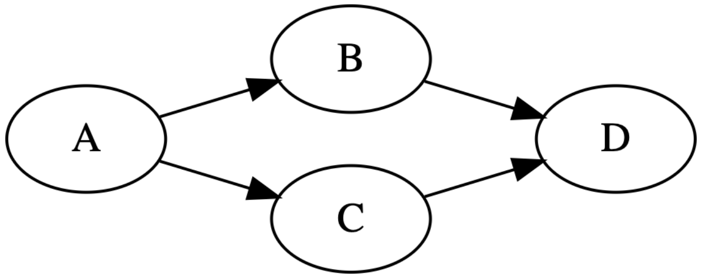
```
// traverse every task in taskflow using the given unary function
taskflow.for_each_task([](tf::Task task){

  // print the name of the task
  std::cout << “Task ” << task.name() << ‘\n’;

  // traverse all successors of the task
  task.for_each_successor([](tf::Task s) mutable {
    std::cout << task.name() << “->” << s.name() << ‘ ’;
  });

  std::cout << “\n”;

  // traverse all predecessors of the task
  task.for_each_predecessor([] (tf::Task p) mutable {
    std::cout << p.name() << “->” << task.name() << ‘ ’;
 });
});
```




```
Task A
  A->B A->C
Task B
  B->D
  A->B
Task C
  C->D
  A->C
Task D
  B->D C->D
```

Submit a Taskflow to an Executor

- **A `tf::Executor` manages a set of worker threads to run submitted tasks**
  - Implements a work-stealing algorithm to achieve dynamic load balancing[1]
  - Implements a notification algorithm to adapt worker availability to dynamic task parallelism
- **Execution methods are either blocking or non-blocking**

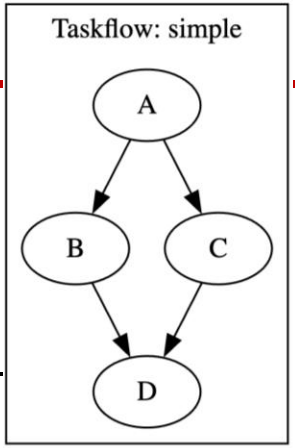
```
tf::Taskflow taskflow1, taskflow2, taskflow3;
tf::Executor executor;

// use run methods to submit a taskflow for execution
tf::Future<void> future1 = executor.run(taskflow1);       // run once
tf::Future<void> future2 = executor.run_n(taskflow2, 10); // run multiple times
tf::Future<void> future3 = executor.run_until(            // run repeatedly until
  taskflow3, [i=0]() mutable { return i++>5; }            // the condition is met
);
// synchronize the execution
future1.wait();           // block until taskflow1 finishes
executor.wait_for_all(); // block until all submitted tasks finish
```

`tf::Future` is derived from `std::future` with a few Taskflow-specific operations added (e.g., cancellation).

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

Visualize a Taskflow Graph



- **Dump a taskflow to a DOT format through a C++ output stream**
  - Copy and paste the DOT output to an online visualizer to visualize the task graph
    - Ex: https://dreampuf.github.io/GraphvizOnline/

```
tf::Taskflow taskflow("simple");
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);

taskflow.dump(std::cout);
```

Dump the taskflow through the standard output (`std::cout`).

```
digraph Taskflow {
  subgraph cluster_p0x7 {
    label="Taskflow: simple";
    p0x3744000[label="A" ];
    p0x3744000 -> p0x600037440f0;
    p0x3744000 -> p0x600037441e0;
    p0x37440f0[label="B" ];
    p0x37440f0 -> p0x600037442d0;
    p0x37441e0[label="C" ];
    p0x37441e0 -> p0x600037442d0;
    p0x37442d0[label="D" ];
  }
}
```

Takeaways

- Learn how to program static task graph parallelism in Taskflow
- Understand the design philosophy behind Taskflow
- Showcase a real-world application of static task graph programming
- Conclude the talk

Taskflow Doesn't Touch Data Abstraction!

- **Taskflow is designed to focus on task parallelism, NOT data parallelism**
  - Taskflow is **NOT** a data-parallel programming model like dataflow, which often targets fine-grained parallelism, but rather a task-based model which targets *coarse-grained* parallelism
  - Taskflow is **NOT** designing yet another data abstraction for parallel programming
    - Why? Because the way your data should be optimized for parallelism is completely depending on your applications, which we don't know too much about!
- **How do I communicate data among different tasks?**

Approach #1: Stateful capture

```
auto task1 = taskflow.emplace(
  [&data1] () { some_func(data1); }
);
auto task2 = taskflow.emplace(
  [&data2] () { std::cout << data2; }
);
task1.precede(task2);
```

Capture data in the lambda when creating tasks.

Approach #2: C-styled pointer

```
auto data = 5;
auto task = taskflow.placeholder();
task.data(&data)
    .work([task](){
      std::cout << *(int*)task.data();
    });
```

Attach user data to a task using C-styled pointer, `void*`, that can be set and accessed through `tf::Task::data`.

Understand the Lifetime of a Task

- **Every task belongs to a graph at a time and remains alive with that graph**
  - As long as a taskflow remains alive, all of its associated tasks remain alive
- **The lifetime of a task affects its callable, particularly the captured values**
  - When a taskflow is destroyed, all of its tasks and their captured variables are also destroyed

```
tf::Executor executor;
{
  tf::Taskflow taskflow;
  taskflow.emplace(
    [](){ std::cout << "Task A\n"; },
    [](){ std::cout << "Task B\n"; },
    [](){ std::cout << "Task C\n"; },
    [](){ std::cout << "Task D\n"; }
  );
  executor.run(taskflow);
}  // taskflow is destroyed after the block
executor.wait_for_all();
```

```
tf::Executor executor;
tf::Taskflow taskflow;
taskflow.emplace(
  [](){ std::cout << "Task A\n"; },
  [](){ std::cout << "Task B\n"; },
  [](){ std::cout << "Task C\n"; },
  [](){ std::cout << "Task D\n"; }
);
executor.run(taskflow);
executor.wait_for_all();
```

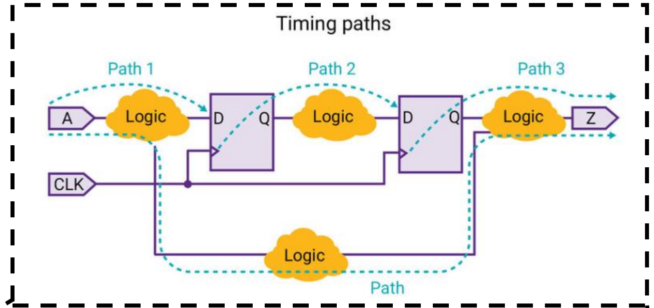
It is your responsibility to keep relevant taskflows alive during their execution.

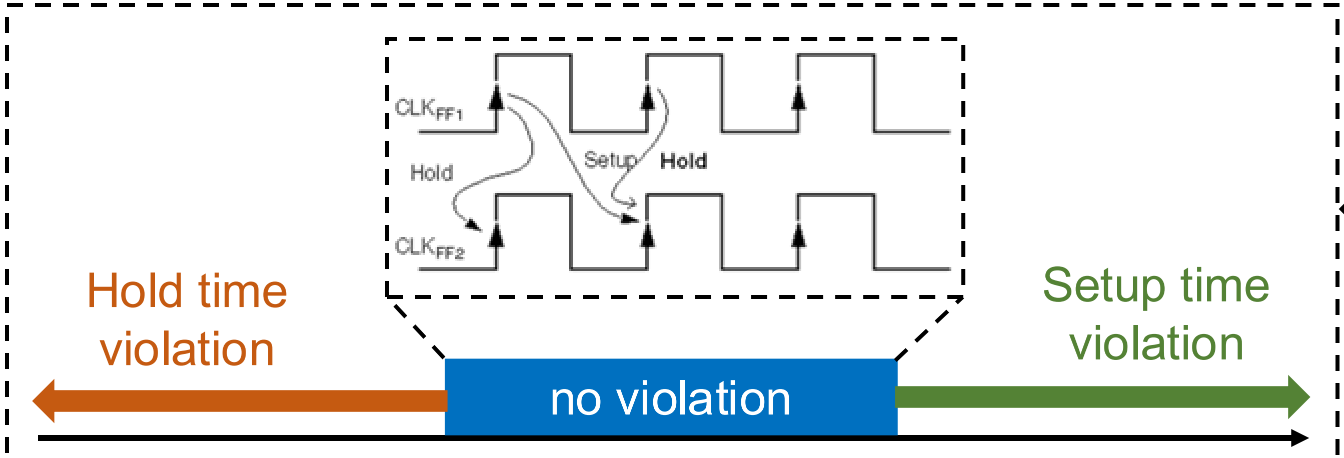
Takeaways

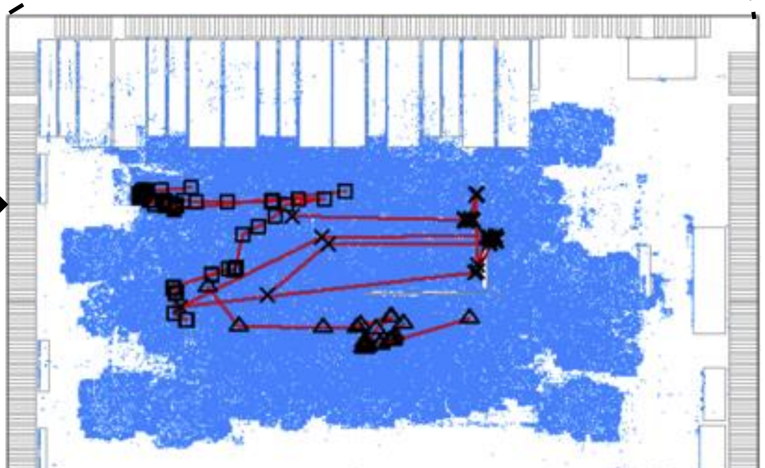
- Learn how to program static task graph parallelism in Taskflow
- Understand the design philosophy behind Taskflow
- Showcase a real-world application of static task graph programming
- Conclude the talk

VLSI Static Timing Analysis (STA)

- **A key step in EDA to validate the expected timing behaviors of a circuit**
  - Derives a timing graph from a given circuit design
  - Induces timing constraints from sequential elements (e.g., flops)
  - Propagates timing quantities from inputs to outputs
    - Ex: slew, delay, arrival time, required arrival time
  - Validates data paths and identify any violations







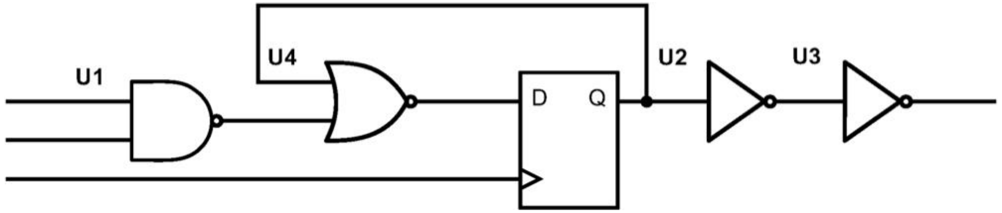
<u>STA Goal</u>: *What signals cannot arrive on time (i.e., violation)?*

[1]: J. Bhasker and Rakesh Chadha, "Static Timing Analysis for Nanometer Designs: A Practical Approach", *Springer*, 2009

Parallelizing STA using Taskflow

- **Task-parallel timing propagation**[1]
  - Task: per-pin propagation function
    - Ex: cell delay, net delay calculator
  - Edge: pin-to-pin dependency
    - Ex: intra-/inter-gate dependencies

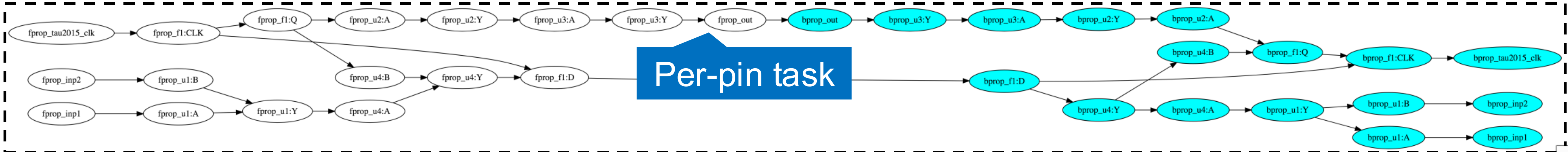
A circuit of four comb gates and one seq gate




```
ot> report_timing # report the most critical path
Startpoint    : inp1
Endpoint      : f1:D
Analysis type : min
------------------------------------------------
Type Delay Time Dir Description
------------------------------------------------
port 0.000 0.000 fall inp1
pin  0.000 0.000 fall u1:A (NAND2X1)
…
pin  0.000 2.967 fall f1:D (DFFNEGX1)
…
------------------------------------------------
slack -23.551 VIOLATED
```

Derive a timing propagation task graph

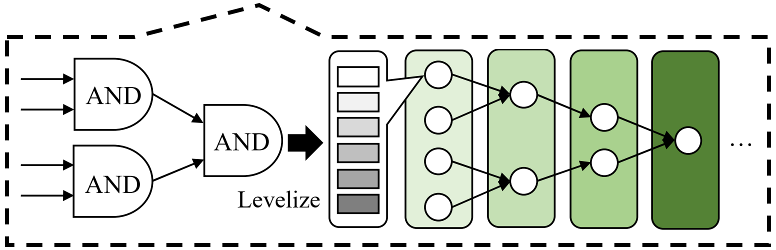
Evaluate and report violated data paths

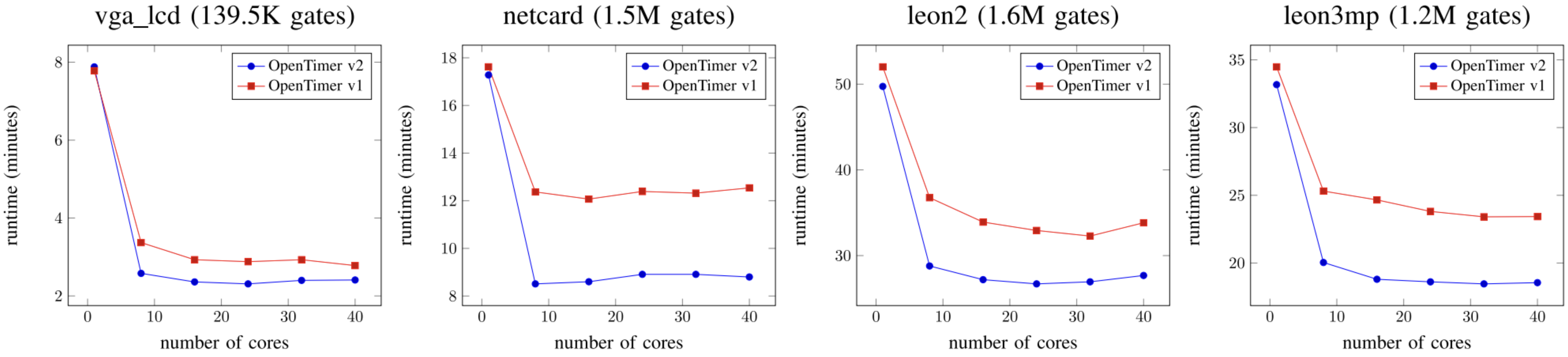


[1]: Tsung-Wei Huang, et al, “OpenTimer v2: A New Parallel Incremental Timing Analysis Engine,” *IEEE TCAD*, 2022

How Good is Task-parallel STA?

- **OpenTimer v1: levelization-based (or loop-parallel) timing propagation[1]**
  - Implemented using OpenMP "`parallel_for`" primitive
- **OpenTimer v2: task-parallel timing propagation[2]**
  - Implemented using Taskflow STGP





💡Task-parallelism allows us to more asynchronously parallelize the timing propagation

[1]: Tsung-Wei Huang and Martin Wong, "OpenTimer: A High-Performance Timing Analysis Tool," *IEEE/ACM ICCAD*, 2015
[2]: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2022

Example Implementation in OpenTimer[1]

- **Implemented forward and backward timing propagation using Taskflow**

```
void Timer::_build_prop_tasks() {
  // build propagation candidates
  _build_prop_cands();

  // emplace fprop tasks
  // (1) propagate the rc timing
  // (2) propagate the slew
  // (3) propagate the delay
  // (4) propagate the constraint
  for(auto pin : _fprop_cands) {
    pin->_ftask = _taskflow.emplace([pin]{
      _fprop_rc_timing(*pin);
      _fprop_slew(*pin);
      _fprop_delay(*pin);
      _fprop_at(*pin);
      _fprop_test(*pin);
    });
  }
```

```
  // Build the dependency
  for(auto to : _fprop_cands) {
    for(auto arc : to->_fanin) {
      if(arc->_has_state(Arc::LOOP_BREAKER))
      {
        continue;
      }
      if(auto& from = arc->_from;
        from._has_state(Pin::FPROP_CAND)) {
        from._ftask->precede(to->_ftask);
      }
    }
  }

  ... // continue for backprop tasks
}
```

Traverse the timing graph to build a taskflow.

[1]:OpenTimer: https://raw.githubusercontent.com/OpenTimer/OpenTimer/refs/heads/master/ot/timer/timer.cpp

Takeaways

- Learn how to program static task graph parallelism in Taskflow
- Understand the design philosophy behind Taskflow
- Showcase a real-world application of static task graph programming
- 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();
```

Most Applications can be Realized by STGP

- **Many parallel workloads exhibit static behavior and dependency structure**
  - No recursion – parallelism is flat and happens only in the first hierarchy of the graph
  - No control flow – execution flow of the graph is acyclic and predictable
- **Advantages of STGP:**
  - ☑ Programming models are very simple
  - ☑ Reasoning about the task graph is very easy
  - ☑ Code complexity grows linearly with the graph size
  - ☑ Scheduling overhead is the least
  - ☑ Better opportunity for compile-time optimization
- **Disadvantages of STGP:**
  - ⛔ Graph structure must be known before execution
    - Either at programming time or runtime time
  - ⛔ Graph structure cannot depend on runtime variables
  - ⛔ Graph structure cannot depend on control-flow results

```
// LOC is linear to the graph size
auto [S, a0, b0, ..., a3, b3, T] =
taskflow.emplace(
   [](){ std::cout << “S”; },
   [](){ std::cout << “a0”; };
   ...
   [](){ std::cout << “T”; }
);
// create dependencies
 S.precede(a0, a1, b0);
a0.precede(a1, b2);
a1.precede(a2, b3);
...
a3.precede(T);
b3.precede(T);
```