Taskflow: A General-purpose Task-parallel Programming System using Modern C++

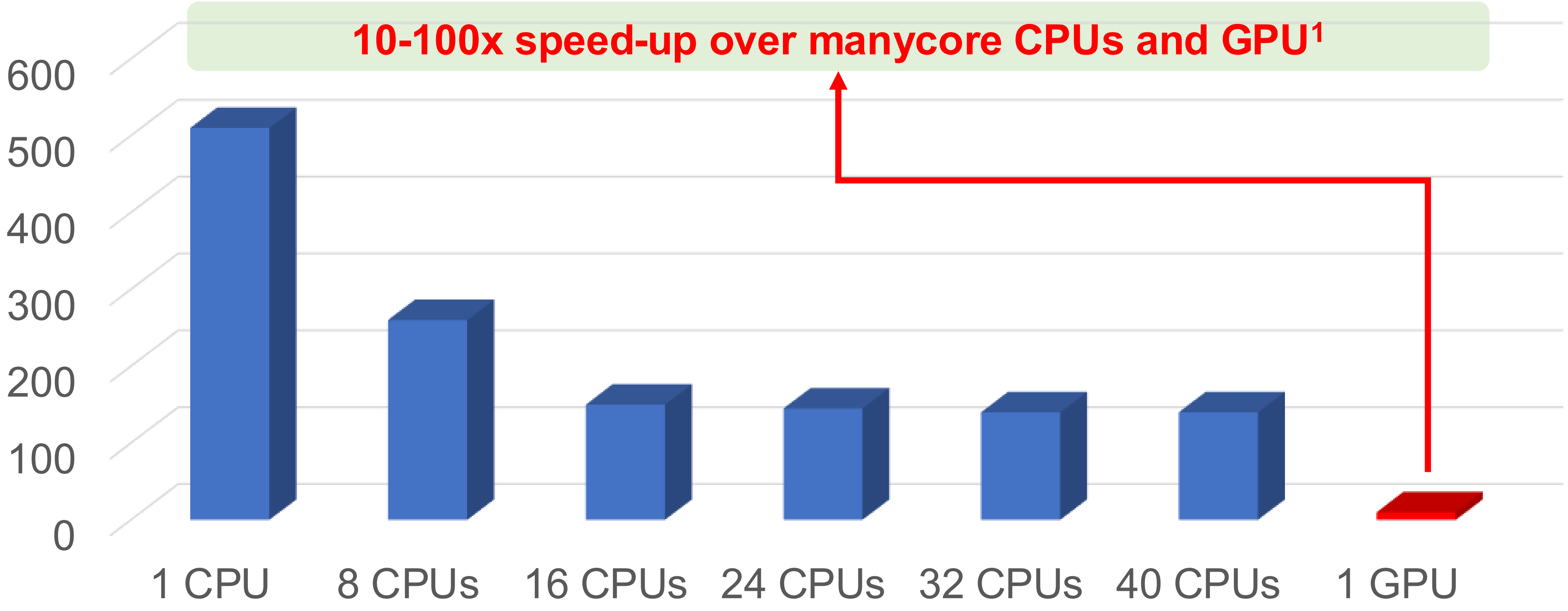
Dr. Tsung-Wei (TW) Huang, Associate Professor
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://taskflow.github.io/

Takeaways

- Understand the importance of task-parallel programming
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Showcase real-world applications of Taskflow
- Conclude the topic

Why Parallel Computing?

- **Advances performance to a new level previously out of reach**

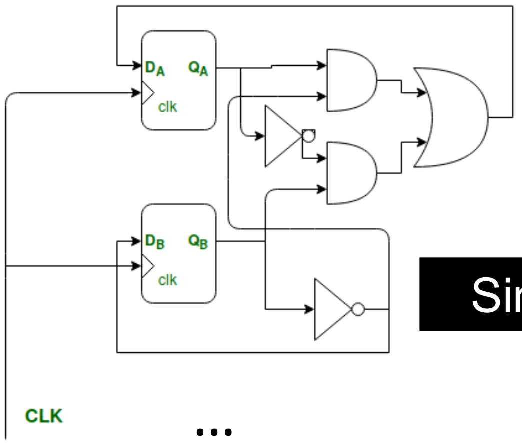


Time (minutes) to speed up a circuit timing analysis algorithm

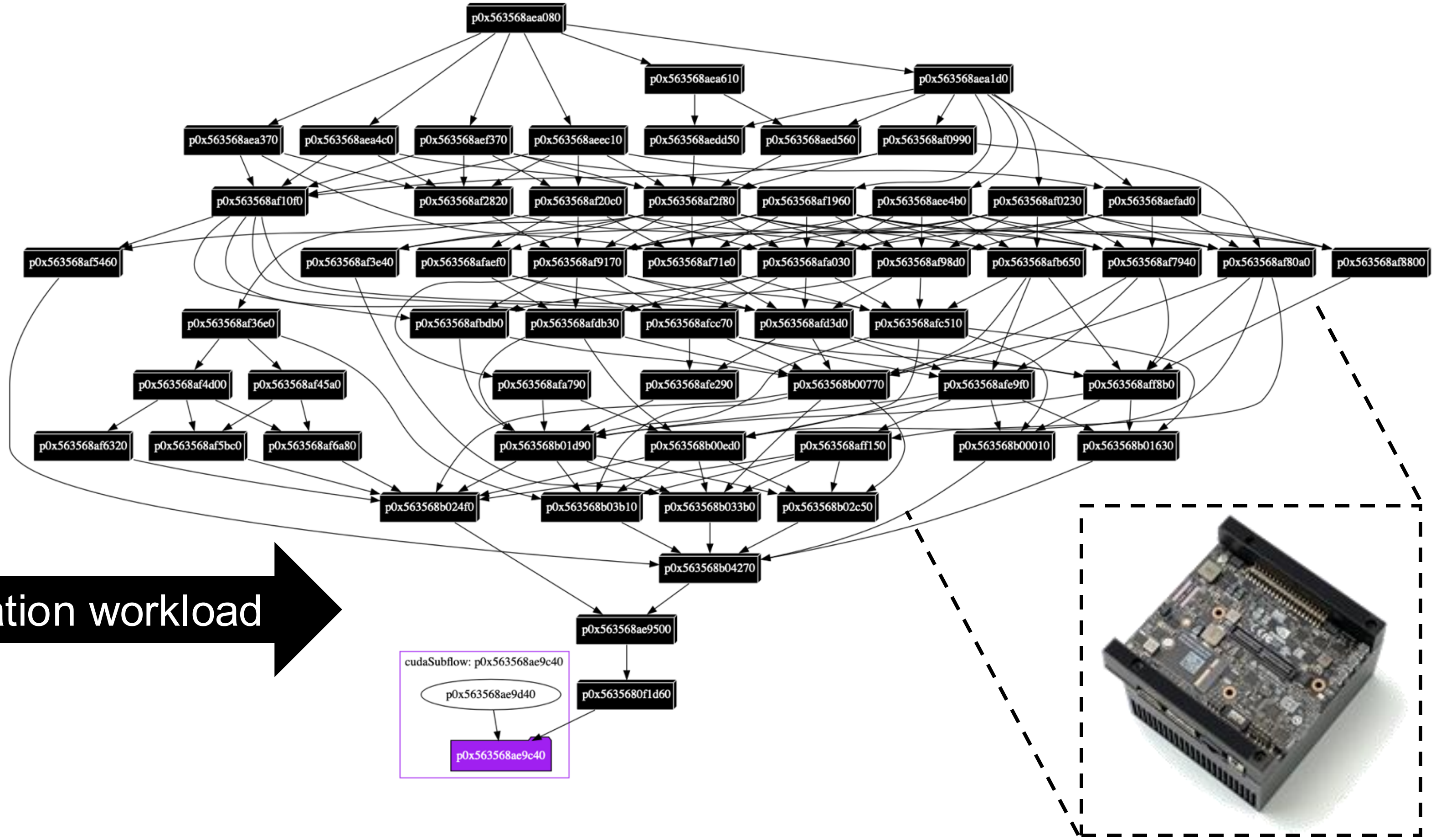
[1]: Zizheng Guo, Tsung-Wei Huang, and Yibo Lin, "GPU-accelerated Static Timing Analysis," *IEEE/ACM ICCAD*, CA, 2020

Today’s Parallel Workload is Very Irregular …

- **GPU-parallel circuit simulation task graph of Nvidia’s NVDLA design[1]**
  - > 500 M gates and nets
  - > 1000 kernels
  - > 1000 dependencies
  - > hours to finish



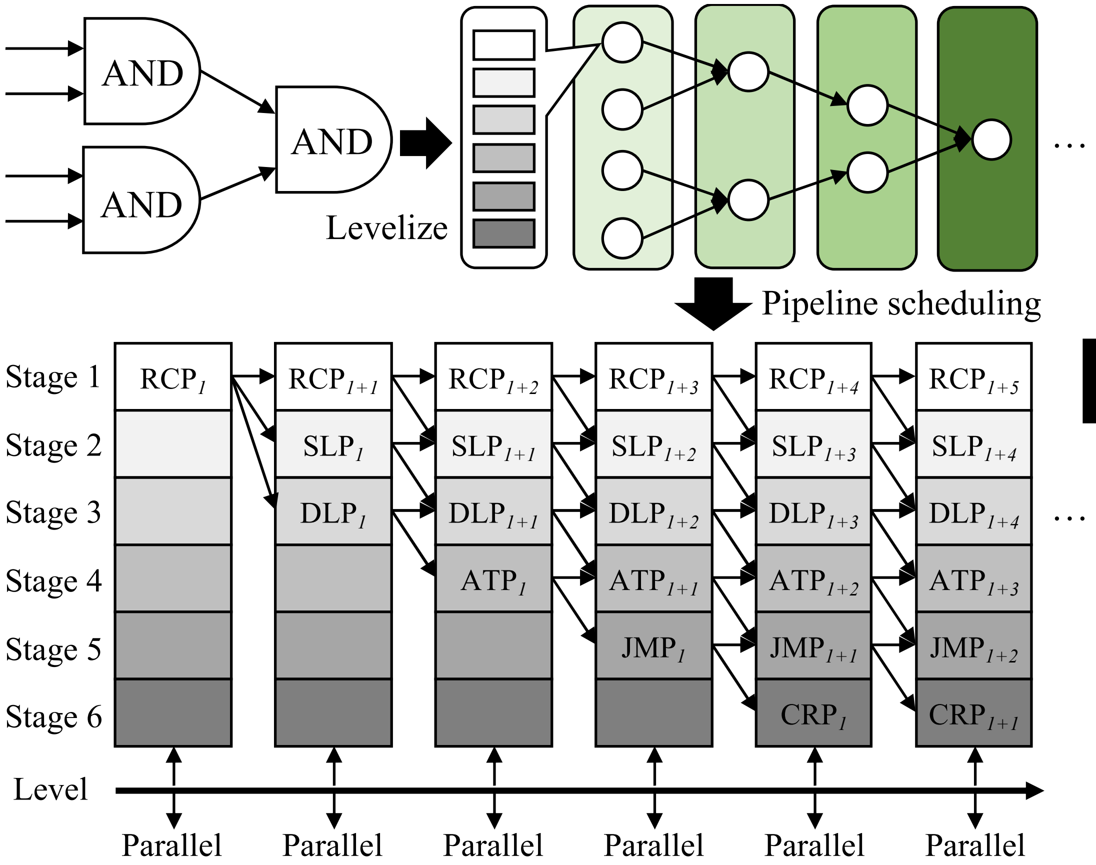
Simulation workload



[1]: Dian-Lun Lin, et al, “From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus,” *ACM ICPP*, 2022

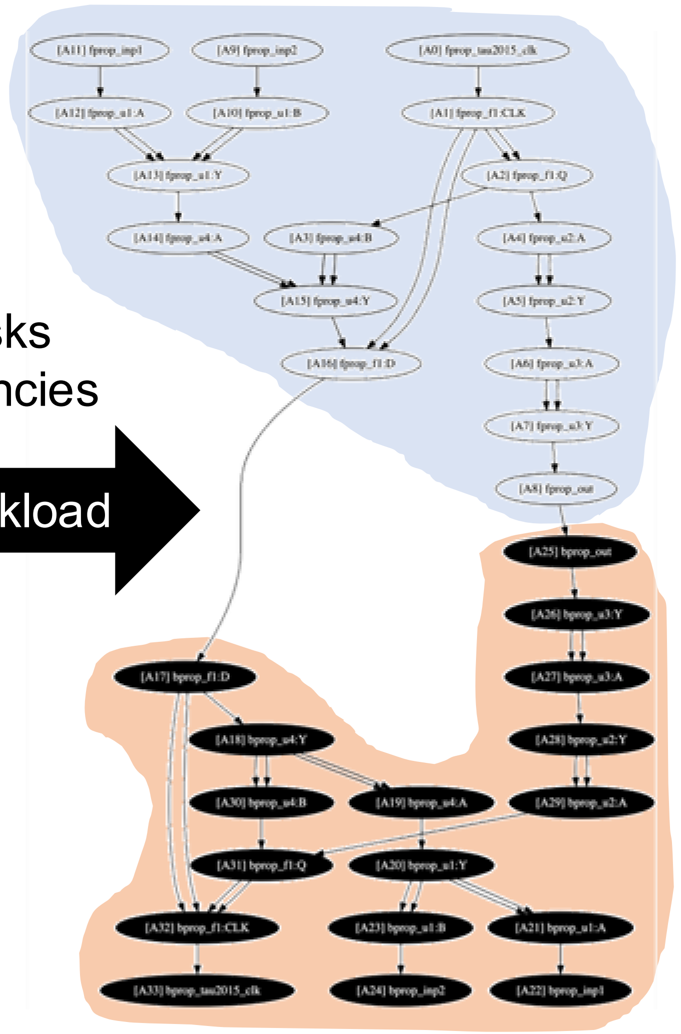
Another Example of Irregular Workload

- **CPU-parallel VLSI static timing analysis**



> 100M async tasks
> 150M dependencies

Propagation workload

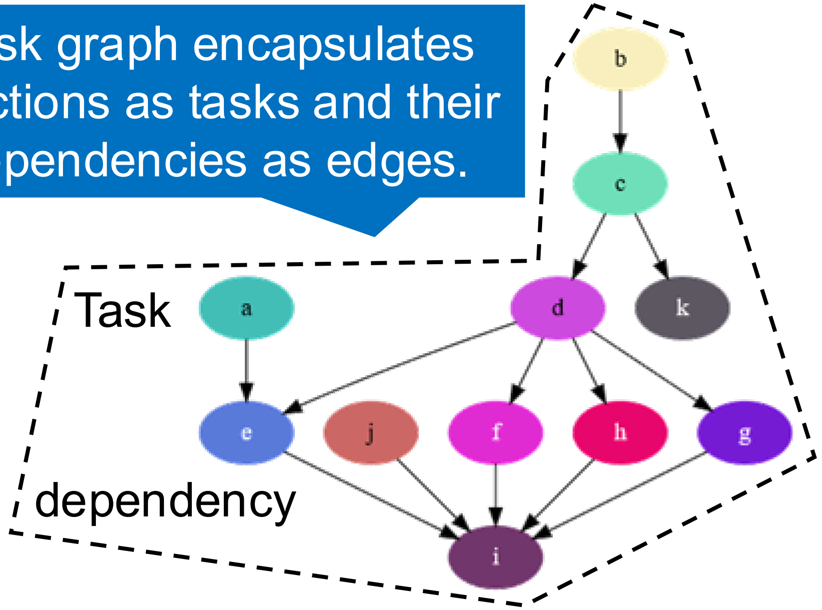


[1]: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2022

Why Task-parallel Programming (TPP)?

- **TPP is particularly effective for parallelizing irregular workloads**
  - Captures developers' intention in decomposing an algorithm into a *top-down* task graph
  - Delegates difficult scheduling details (e.g., load balancing) to an optimized runtime
- **Modern parallel programming libraries are moving towards task parallelism**
  - OpenMP 4.0 task dependency clauses (`omp depend`)
  - C++26 execution control (`std::exec`)
  - TBB dataflow programming (`tbb::flow::graph`)
  - Taskflow control taskflow graph (CTFG) model
  - … (many others)

Task graph encapsulates functions as tasks and their dependencies as edges.



OpenMP StarPU kokkos

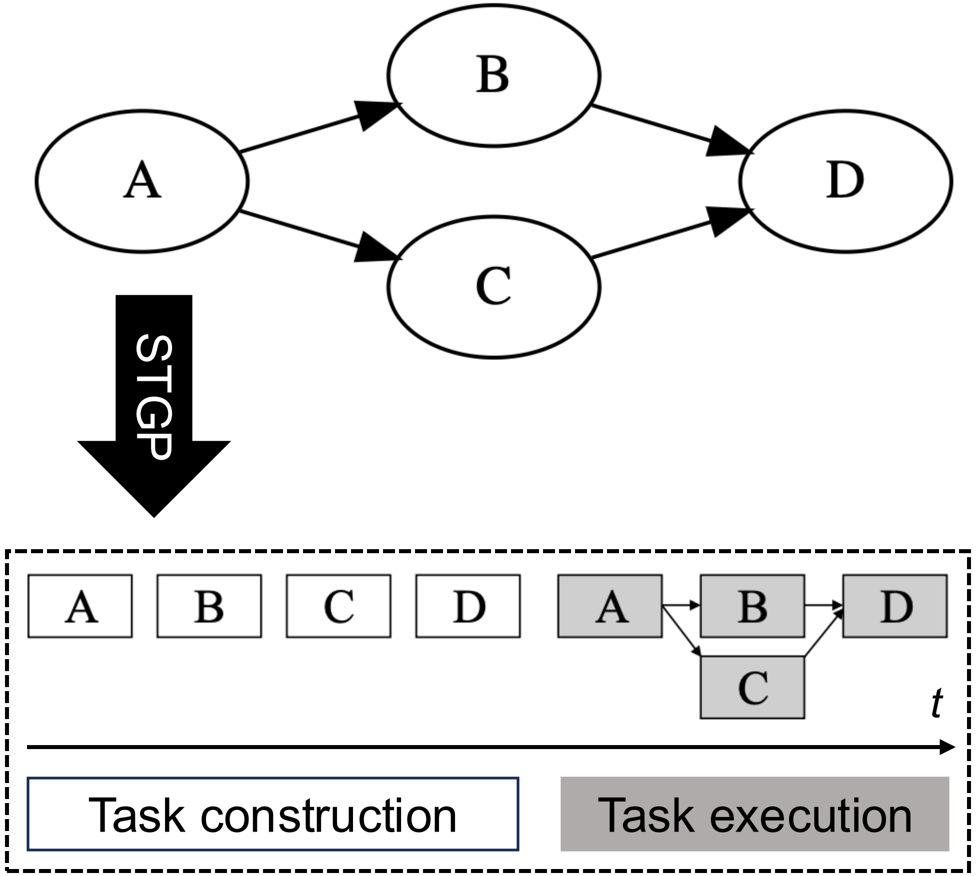
TBB PaRSEC

Takeaways

- Understand the importance of task-parallel programming
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Showcase real-world applications of Taskflow
- Conclude the topic

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;
}
```

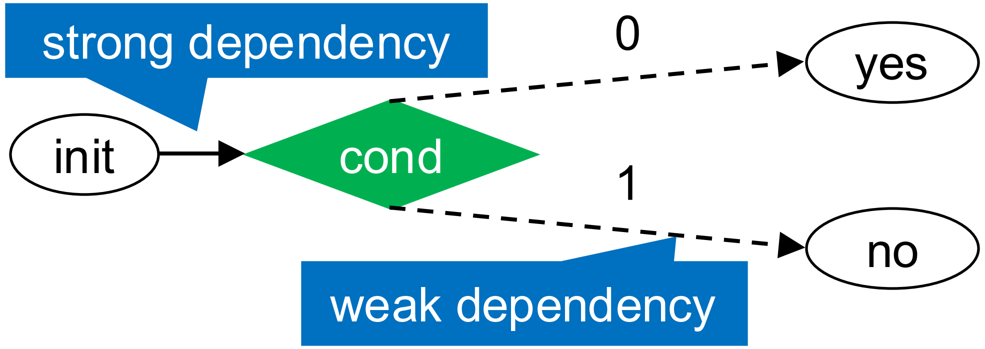


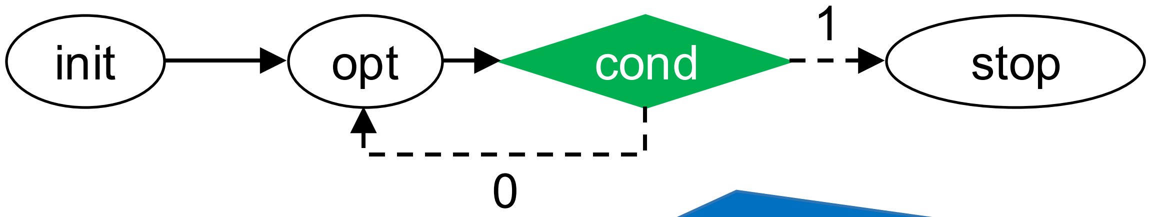
Control Taskflow Graph (CTFG) Programming Model

- **A key innovation that distinguishes Taskflow from existing TPP libraries**

```
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);
```





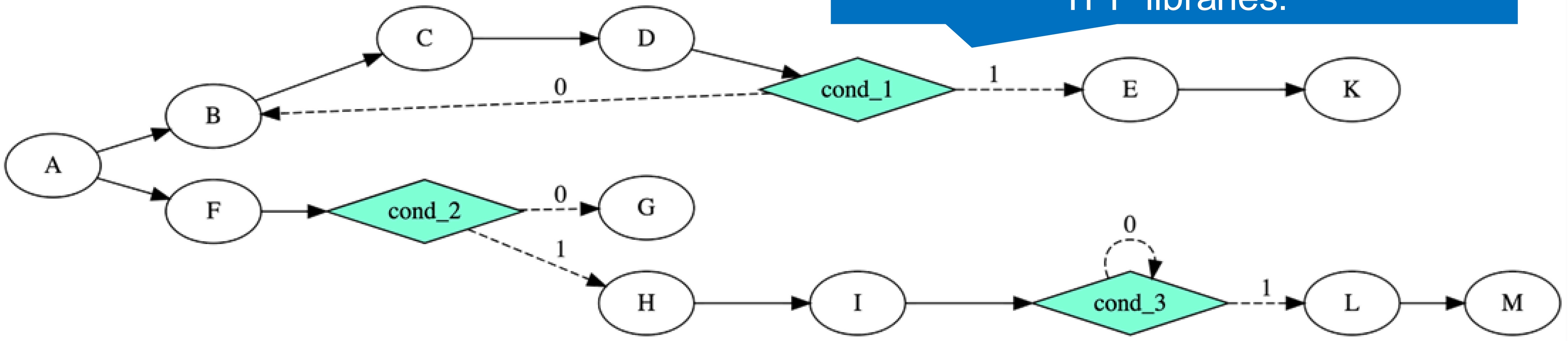
CTFG goes beyond the limitation of traditional DAG-based frameworks (no in-graph control flow).

Power of CTFG Programming Model

- **Enables very efficient overlap among tasks alongside 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 parallelism is almost impossible to achieve using existing TPP libraries.

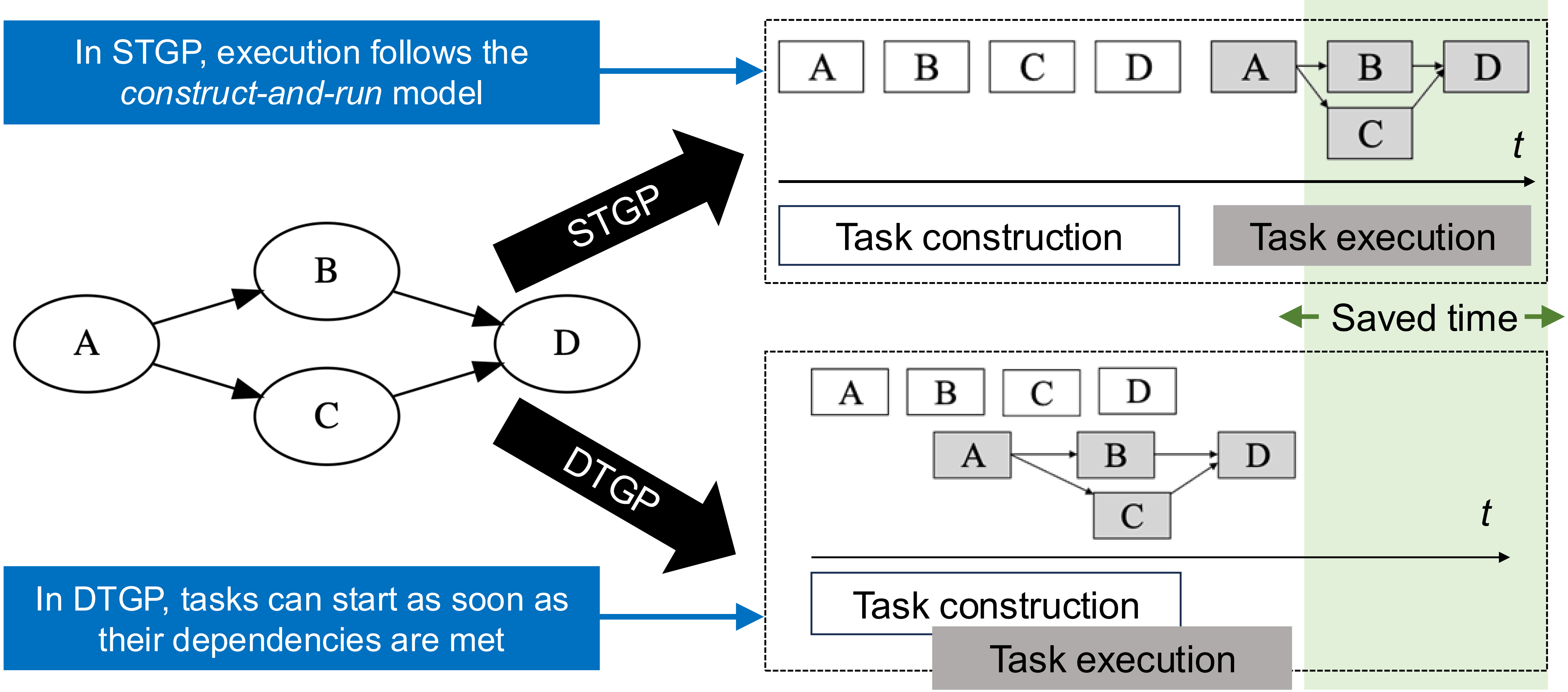


Takeaways

- Understand the importance of task-parallel programming
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Showcase a real-world application of Taskflow
- Conclude the topic

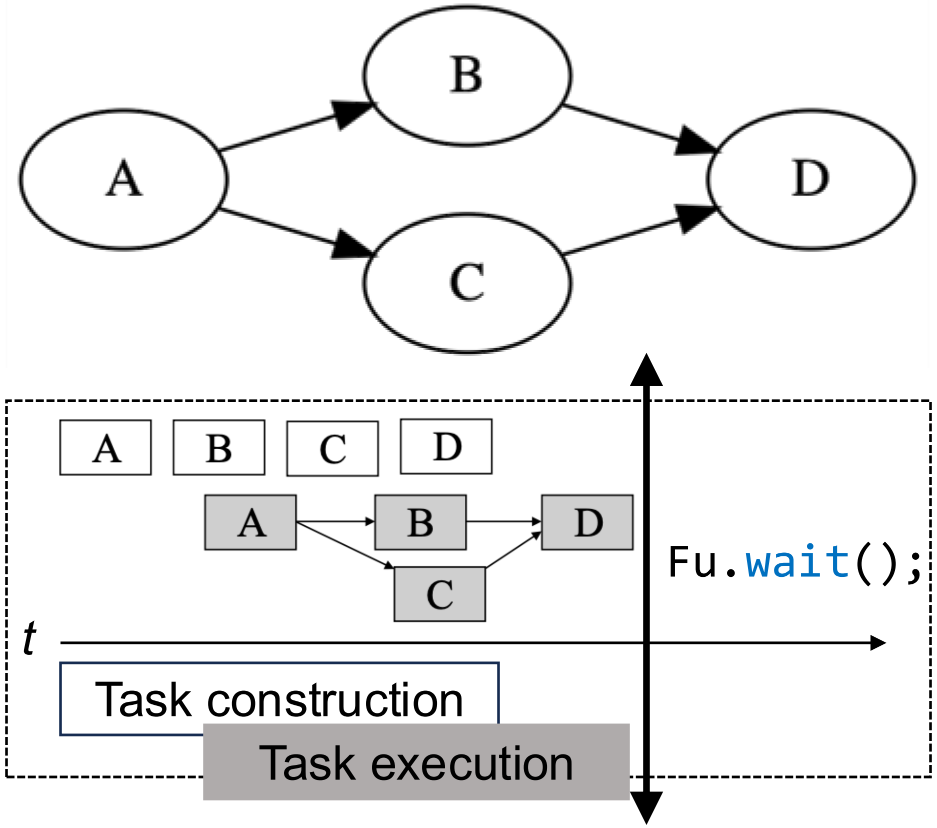
Static vs Dynamic Task Graph Parallelism (DTGP)

- **Examples we've discussed so far are static task graph parallelism (STGP)**



AsyncTask: Dynamic Task Graph Programming in Taskflow

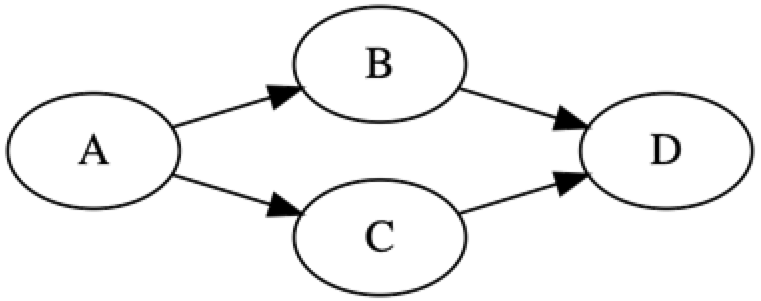
```cpp
// Live: https://godbolt.org/z/j76ThGbWK

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, Fu] = executor.dependent_async([](){
    std::cout << "TaskD\n";
}, B, C);

Fu.wait();
```



Specify variable task dependencies using C++ variadic parameter pack

DTGP Needs a Correct Topological Order

```
tf::Executor executor;

auto A = executor.silent_dependent_async([](){
    std::cout << "TaskA\n";
});
auto D = executor.silent_dependent_async([](){
    std::cout << "TaskD\n";
}, B-is-unavailable-yet, C-is-unavailable-yet);

auto B = executor.silent_dependent_async([](){
    std::cout << "TaskB\n";
}, A);
auto C = executor.silent_dependent_async([](){
    std::cout << "TaskC\n";
}, A);

executor.wait_for_all();
```



An incorrect topological order (A→D→B→C) disallows us from expressing correct DTGP.

DTGP is Flexible for Runtime-driven Execution

- **Assemble task graphs driven by runtime variables and control flow:**

```
if (a == true) {
  G1 = build_task_graph1();
  if (b == true) {
    G2 = build_task_graph2();
    G1.precede(G2);
    if (c == true) {
      … // defined other TGPs
    }
  }
  else {
    G3 = build_task_graph3();
    G1.precede(G3);
  }
}
```

```
G1 = build_task_graph1();
G2 = build_task_graph2();
if (G1.num_tasks() == 100) {
  G1.precede(G2);
}
else {
  G3 = build_task_graph3();
  G1.precede(G2, G3);
  if(G2.num_dependencies()>=10){
    … // define another TGP
  } else {
    … // define another TGP
  }
}
```

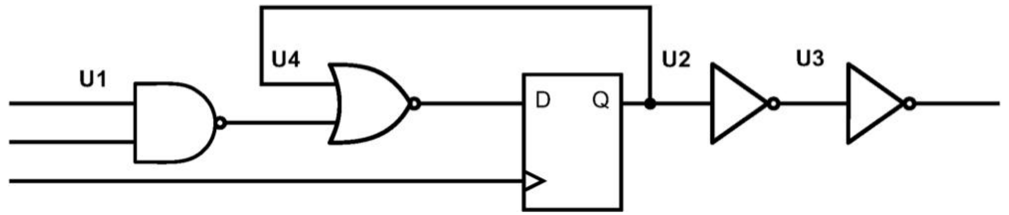
Takeaways

- Understand the importance of task-parallel programming
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Showcase a real-world application of Taskflow
- Conclude the topic

VLSI Static Timing Analysis (STA)

- **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

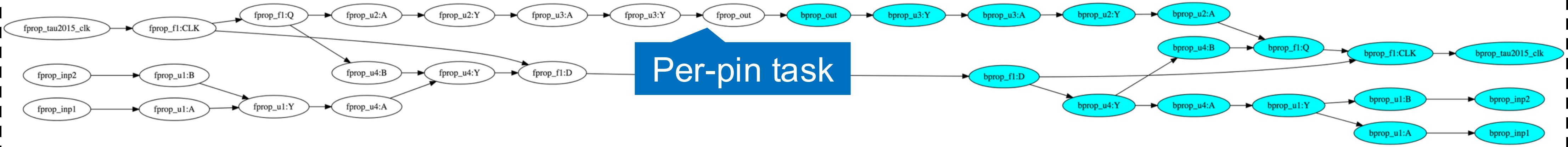
A circuit of four comb gates and one seq gate




```
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

Implementation using Taskflow

- **Forward propagation and backward propagation using Taskflow[1]**

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

  // Emplace the fprop task
  // (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);
    });
  }
```

Traverse each node to build tasks

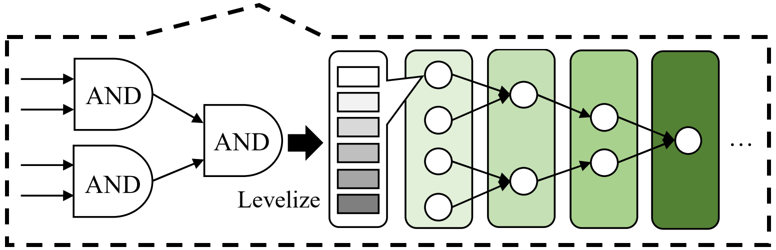
```cpp
  // Build the dependency
  for(auto to : _fprop_cands) {
    for(auto arc : to->_fanin) {
      if(arc->_has_state(Arc::LOOP_BREAKER))
      {
        continue;
      }
      if(auto& from = arc->_from;
        from._has_state(Pin::FPROP_CAND)) {
        from._ftask->precede(to->_ftask);
   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

  ... // continue for bprop tasks
}
```

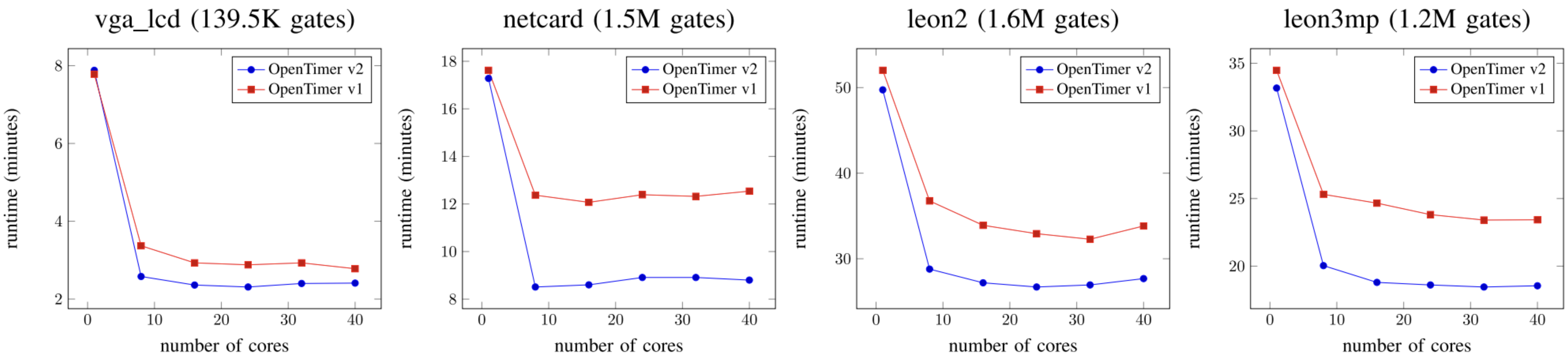
Traverse each edge to build dependencies

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

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

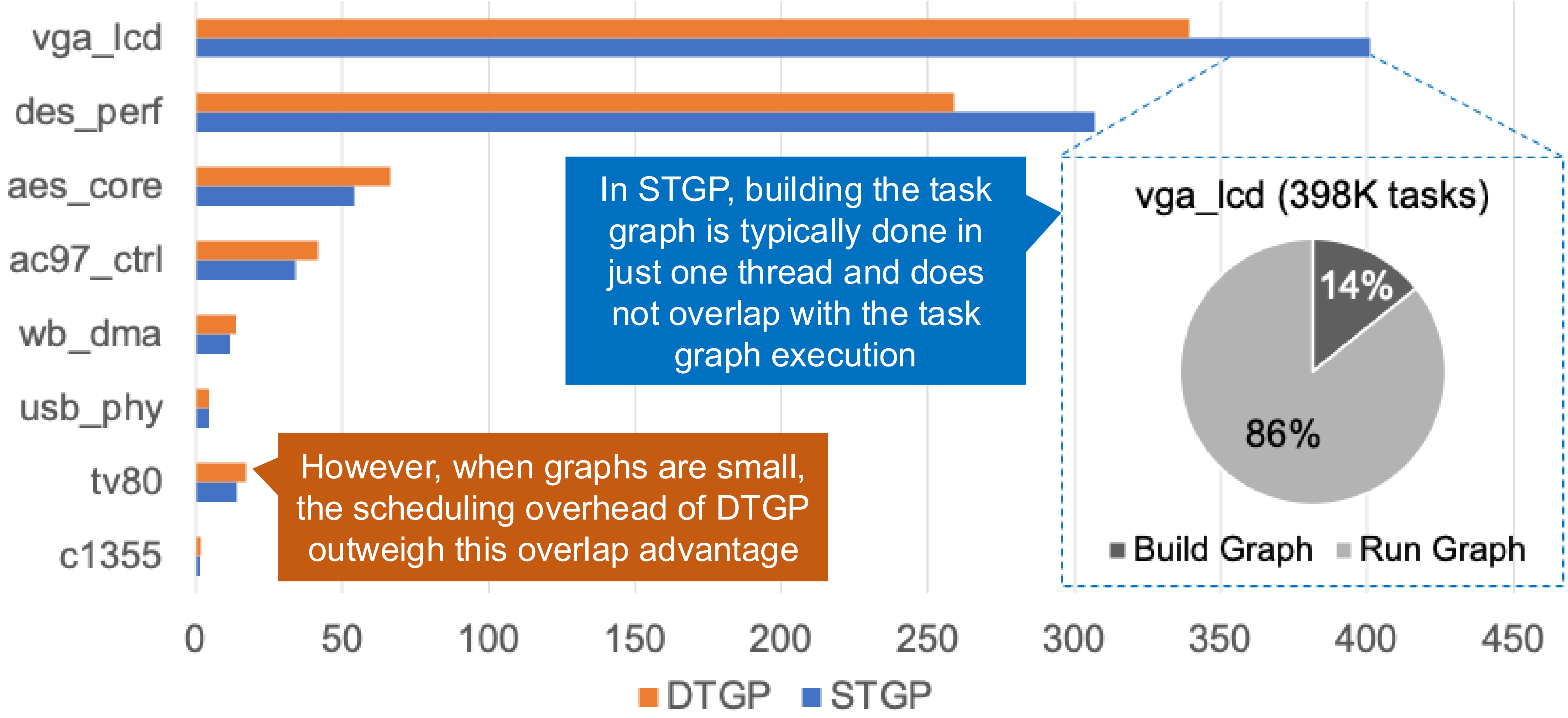




💡Task-parallelism allows us to more asynchronously parallelize the timing propagation

[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

STGP vs DTGP



In STGP, building the task graph is typically done in just one thread and does not overlap with the task graph execution

However, when graphs are small, the scheduling overhead of DTGP outweigh this overlap advantage

Takeaways

- Understand the importance of task-parallel programming
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Showcase a real-world application of Taskflow
- Conclude the topic

Thank you for using Taskflow!

. . .

Questions?

Taskflow: https://taskflow.github.io

Static task graph parallelism

```
// 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 parallelism

```
// 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();
```