Taskflow: A General-purpose Task-parallel Programming System

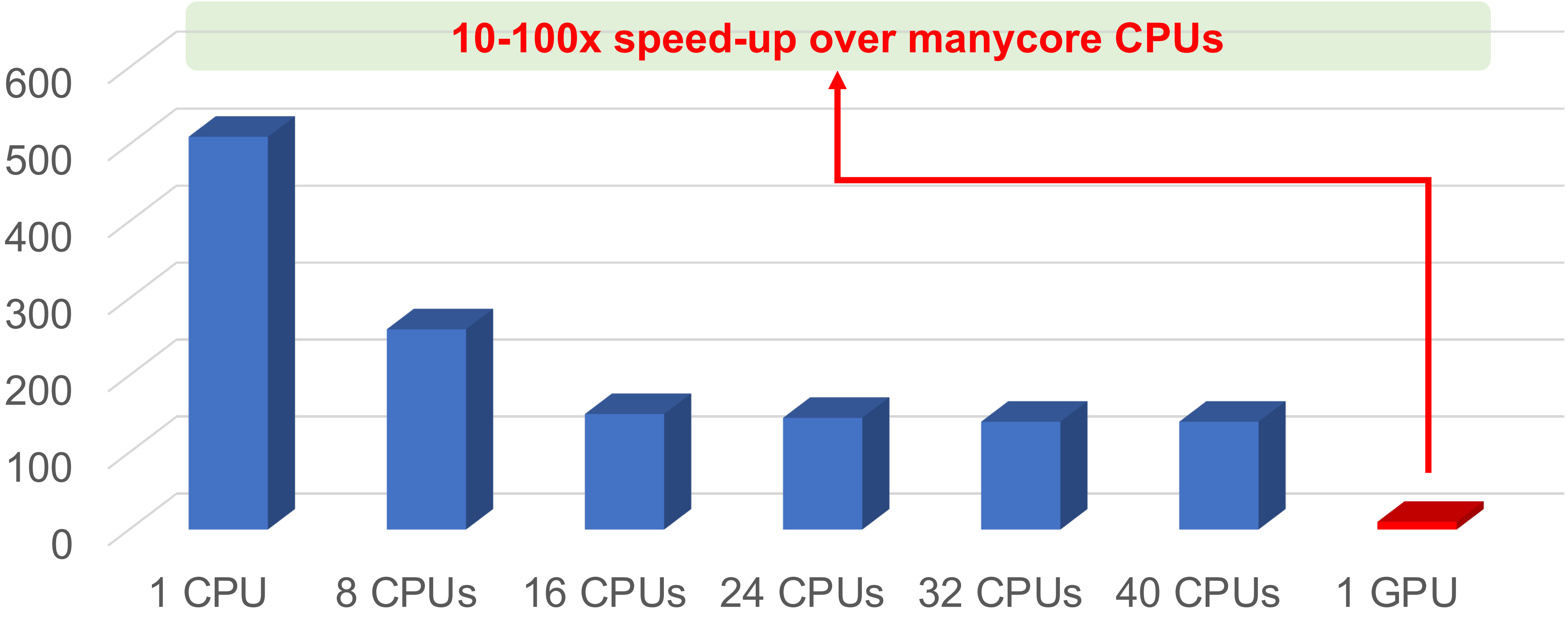
Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://taskflow.github.io/

Takeaways

- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

Why Parallel Computing?

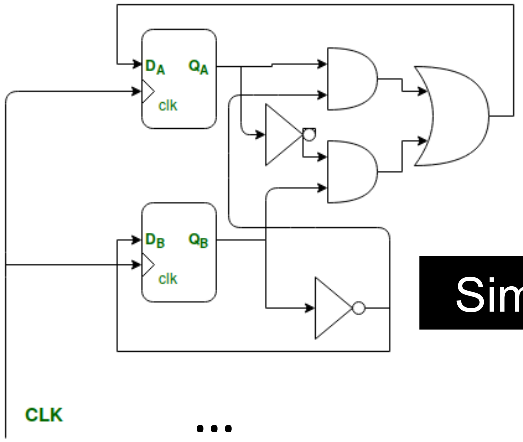
- **Advances performance to a new level previously out of reach**



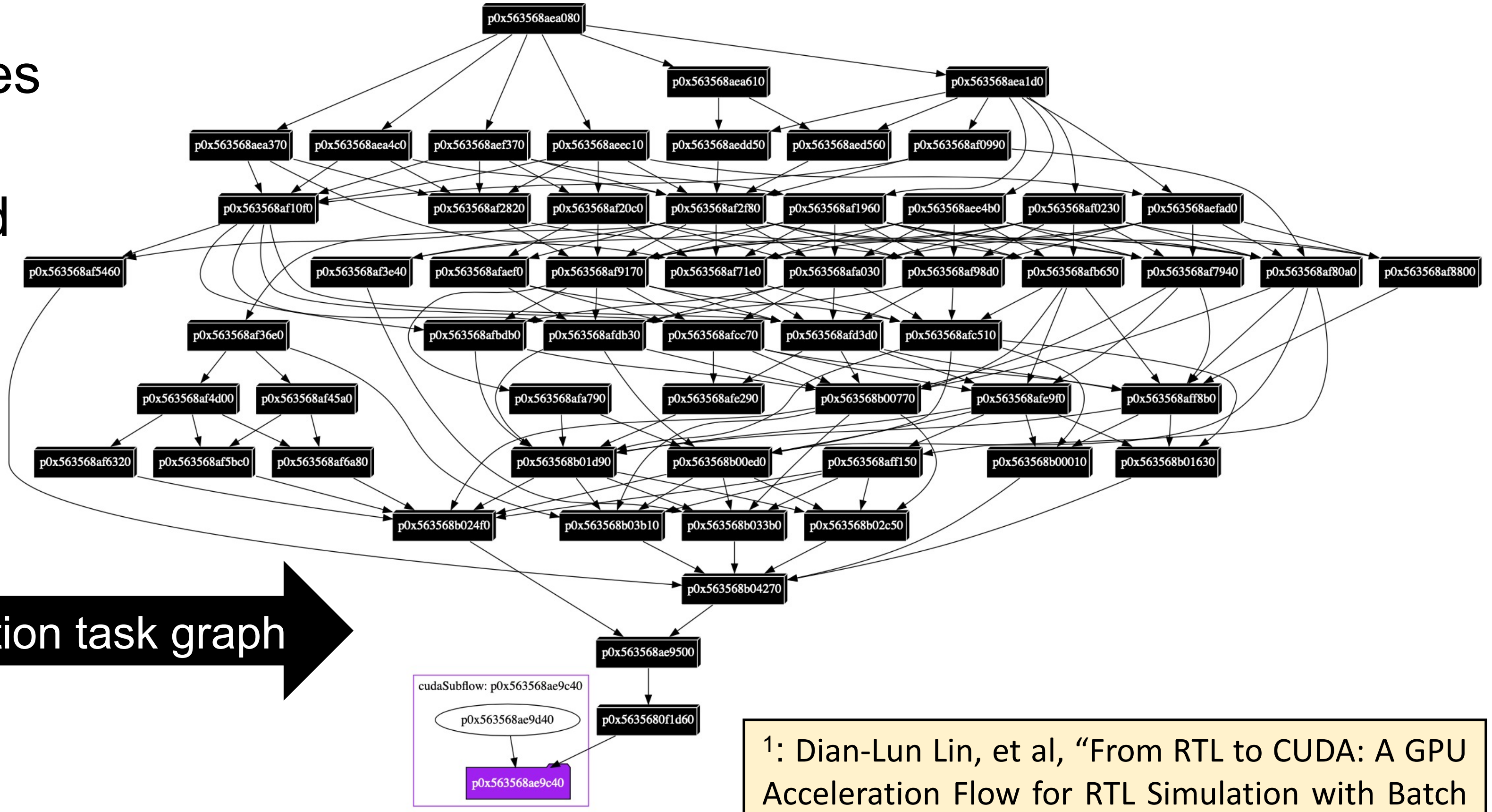
Time (minutes) to speed up a circuit timing analysis algorithm

Today’s Parallel Workload is Very Complex

- **GPU-accelerated circuit analysis on a design of <u>500M</u> gates[1]**
  - >100 kernels
  - >100 dependencies
  - >500s to finish
  - >10hrs turnaround



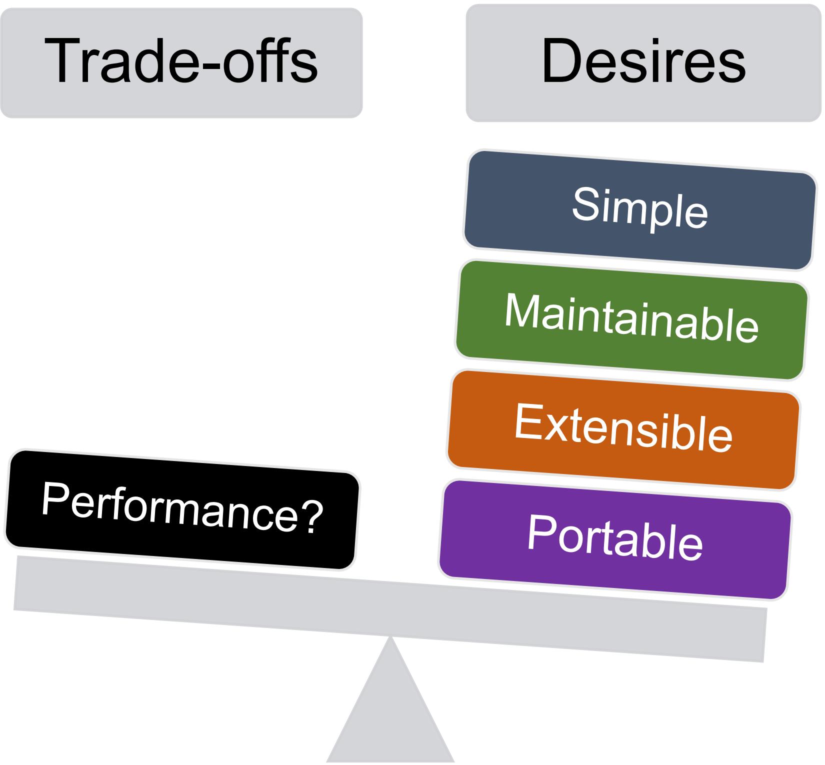
Simulation task graph



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

Parallel Programming is Not Easy

- **You need to deal with A LOT OF technical details**
  - Parallelism abstraction (software + hardware)
  - Concurrency control
  - Task and data race avoidance
  - Dependency constraints
  - Scheduling efficiencies (load balancing)
  - Performance portability
  - …
- **And, don’t forget about trade-offs**
  - Performance vs Desires



Trade-offs

Desires

Simple

Maintainable

Extensible

Portable

Performance?

Need a Good Programming Abstraction

- **From user's perspective, the biggest challenge is *transparency***
  - Programming abstraction, runtime optimization, load balancing, etc.
- **Observing from the evolution of parallel programming:**
  - ***Task graph parallelism*** (TGP) is the best model for future parallel arch
  - Capture programmers' intention in decomposing a heterogeneous algorithm into a top-down task graph
  - Runtime can schedule dependent tasks across a large number of processing units (e.g., CPUs, GPUs)

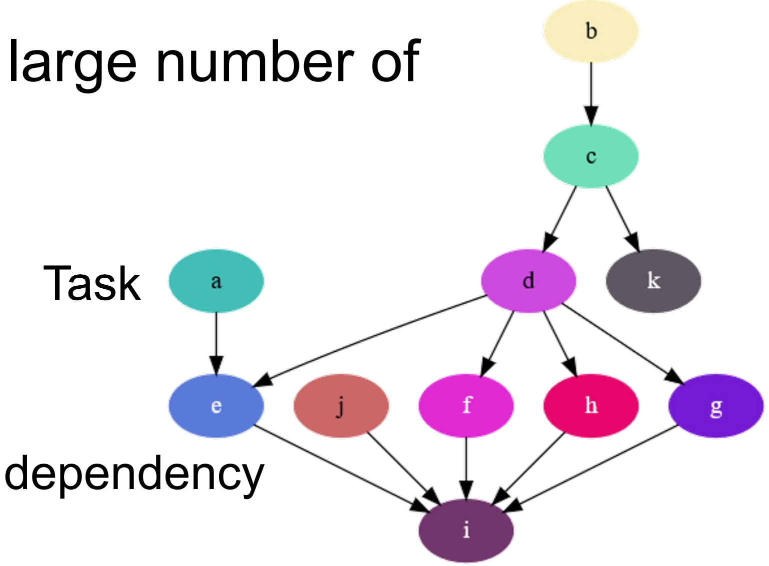
OpenMP

StarPU

kokkos

TBB

PaRSEC



Two Problems of Existing Tools for EDA …

- **EDA has *<u>complex task dependencies</u>***
  - **Example**: analysis algorithms compute the circuit network of multi-millions of nodes and dependencies
  - **Problem**: existing tools are often good at loop parallelism but weak in expressing heterogeneous task graphs at this large scale
- **EDA has *<u>complex control flow</u>***
  - **Example**: synthesis algorithms make essential use of *dynamic control flow* to implement various patterns
    - Combinatorial optimization (e.g., graph algorithms, discrete math)
    - analytical methods (e.g., physical synthesis)
  - **Problem**: existing tools are often good at static task graph parallelism but weak in expressing a dynamic task graph that depends on runtime values
    - Ex: run different task graph structures depending on a control-flow result

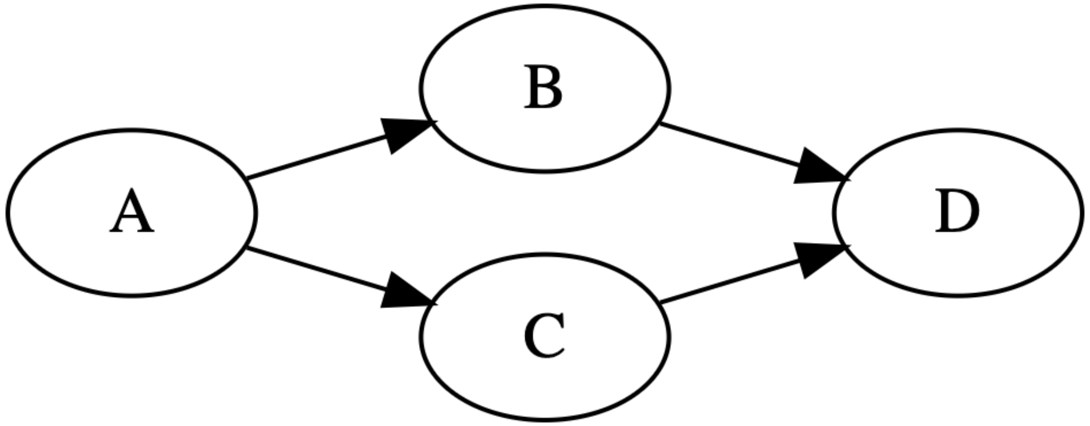
Takeaways

- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

“Hello World” in Taskflow[1]

```
#include <taskflow/taskflow.hpp>
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;
}
```

// live: https://godbolt.org/z/j8hx3xnnx



[1]: T.-W. Huang, et. al, “Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System,” *IEEE TPDS*, vol. 33, no. 6, pp. 1303-1320, June 2022

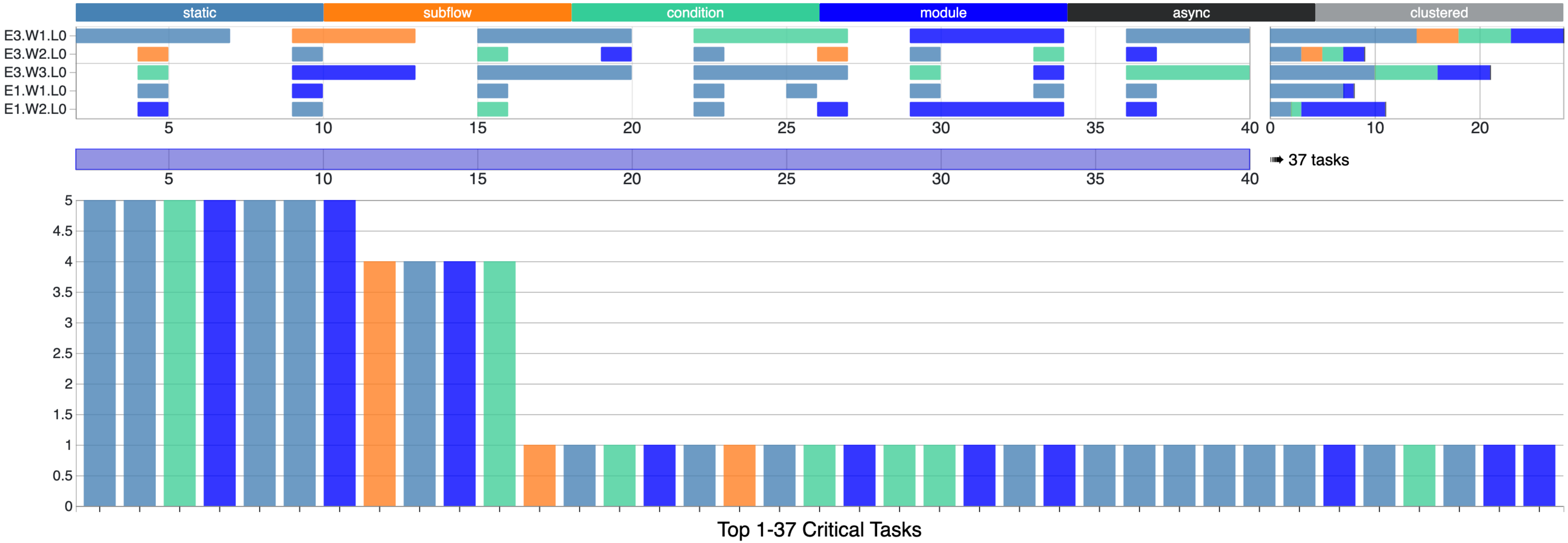
Drop-in Integration

- **Taskflow is header-only – *no wrangle with installation***

```
# clone the Taskflow project
~$ git clone https://github.com/taskflow/taskflow.git
~$ cd taskflow
# compile your program and tell it where to find Taskflow header files
~$ g++ -std=c++20 examples/simple.cpp –I ./ -O2 -pthread -o simple
~$ ./simple
TaskA
TaskC
TaskB
TaskD
```

Built-in Task Execution Visualizer

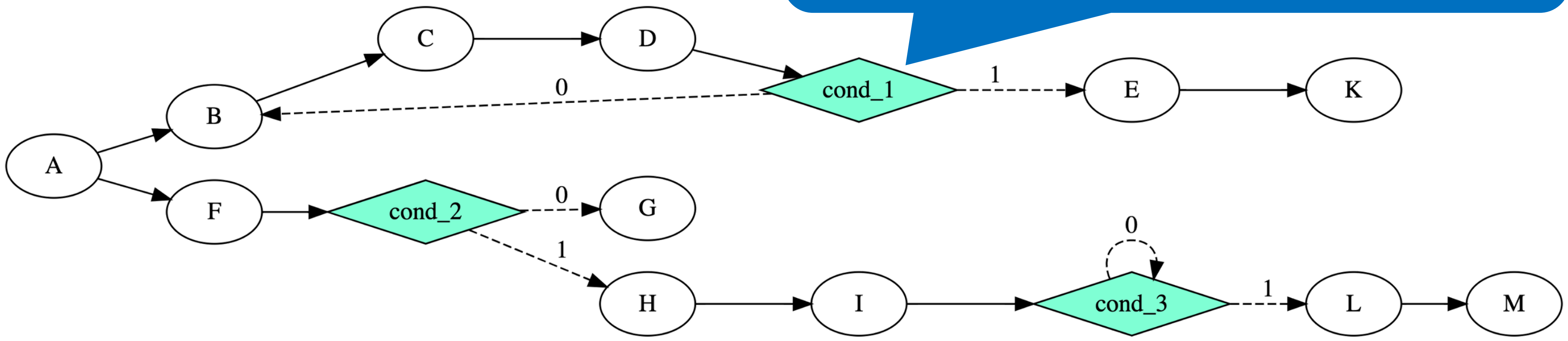
```
# run you program with the env variable TF_ENABLE_PROFILER enabled
# and paste the JSON content to https://taskflow.github.io/tfprof/
~$ TF_ENABLE_PROFILER=simple.json ./simple
```



Control Taskflow Graph Programming (CTFG)

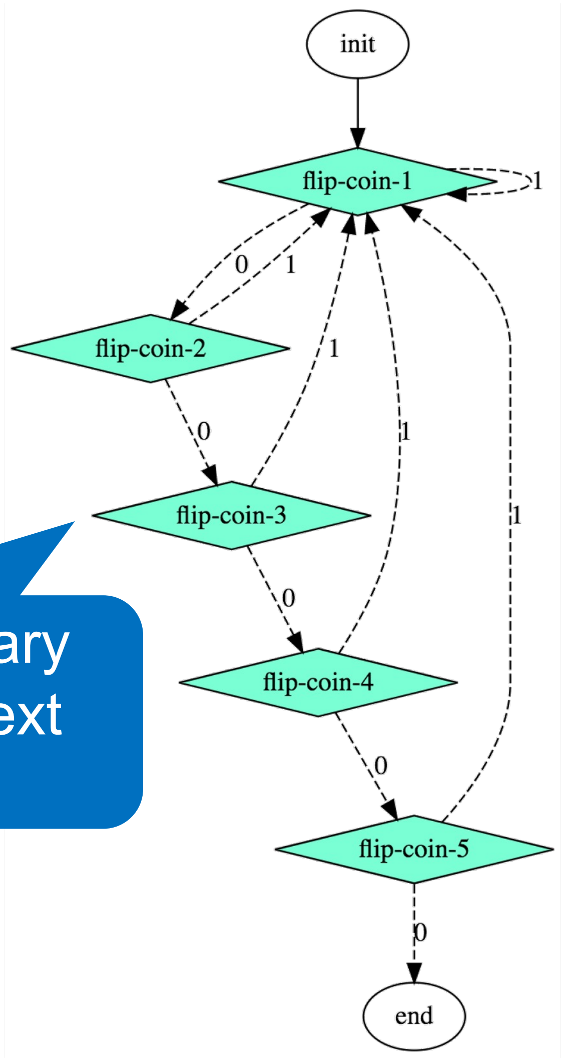
```
// CTFG goes beyond the limitation of traditional DAG-based models
auto cond_1 = taskflow.emplace([](){ return run_B() ? 0 : 1; }); // 0: is the index of B
auto cond_2 = taskflow.emplace([](){ return run_G() ? 0 : 1; }); // 0: is the index of G
auto cond_3 = taskflow.emplace([](){ return loop() ? 0 : 1; }); // 0: is the index of cond_3
cond_1.precede(B, E);          // cycle
cond_2.precede(G, H);          // if-else
cond_3.precede(cond_3, L);     // loop
```

Very difficult for existing DAG-based systems to express an efficient overlap between tasks and control flow …



Non-deterministic Control Flow with CTFG

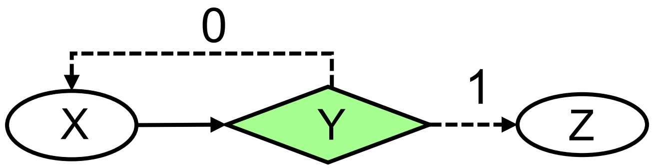
```
auto A = taskflow.emplace([&](){ } );
auto B = taskflow.emplace([&](){ return rand()%2; } );
auto C = taskflow.emplace([&](){ return rand()%2; } );
auto D = taskflow.emplace([&](){ return rand()%2; } );
auto E = taskflow.emplace([&](){ return rand()%2; } );
auto F = taskflow.emplace([&](){ return 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 task flips a binary coin to decide the next task to run

Existing Frameworks on Control Flow?

- **Most existing libraries are DAG-based**
  - Do not anticipate conditional execution …
- **Unroll a task graph over fixed iterations**
  - Task graph size becomes very large …
- **What about dynamic control flow?**
  - Have no choice but resort to a client-side partition of the task graph
  - Synchronize the execution of partitioned task graphs around decision-making points
  - Lack end-to-end parallelism

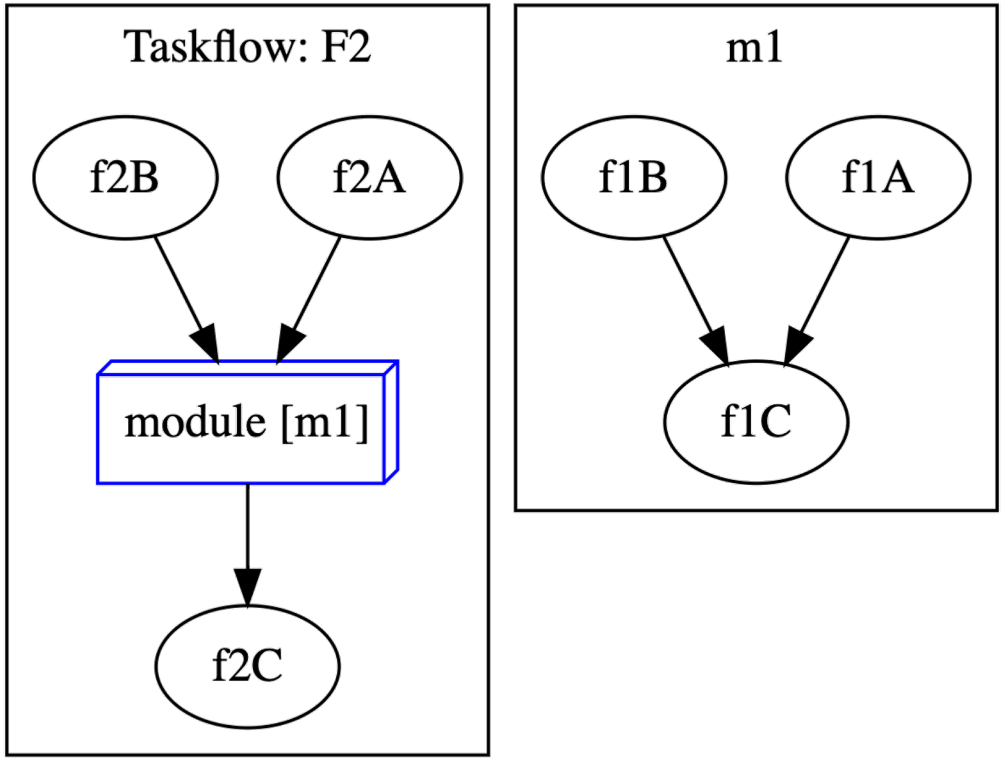


```
tf::Taskflow G;
auto X = G.emplace([](){});
auto Y = G.emplace([](){
  return converged() ? 1 : 0;
});
cond.precede(Z, X);
executor.run(G).wait();
```

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

Composable Tasking

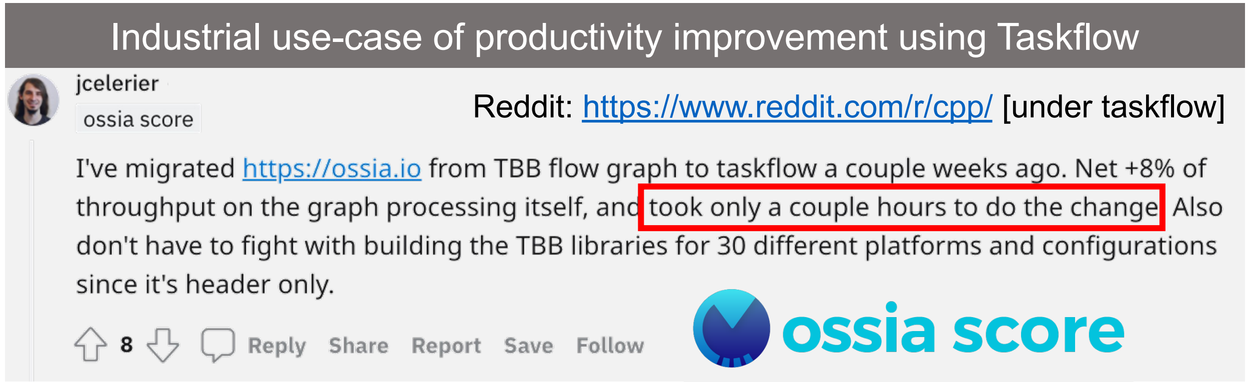
```
tf::Taskflow f1, f2;
auto [f1A, f1B] = f1.emplace(
  [](){ std::cout << "Task f1A\n"; },
  [](){ std::cout << "Task f1B\n"; }
);
auto [f2A, f2B, f2C] = f2.emplace(
  [](){ std::cout << "Task f2A\n"; },
  [](){ std::cout << "Task f2B\n"; },
  [](){ std::cout << "Task f2C\n"; }
);
auto f1_module_task = f2.composed_of(f1);
f1_module_task.succeed(f2A, f2B)
                    .precede(f2C);
```



Everything is Composable in Taskflow

- **End-to-end parallelism in one graph**
  - Task, dependency, control flow all together
  - Scheduling with whole-graph optimization
  - Efficient overlap among heterogeneous tasks
- **Largely improved productivity!**

Composition
(HPDC'22, ICPP'22, HPEC'19)

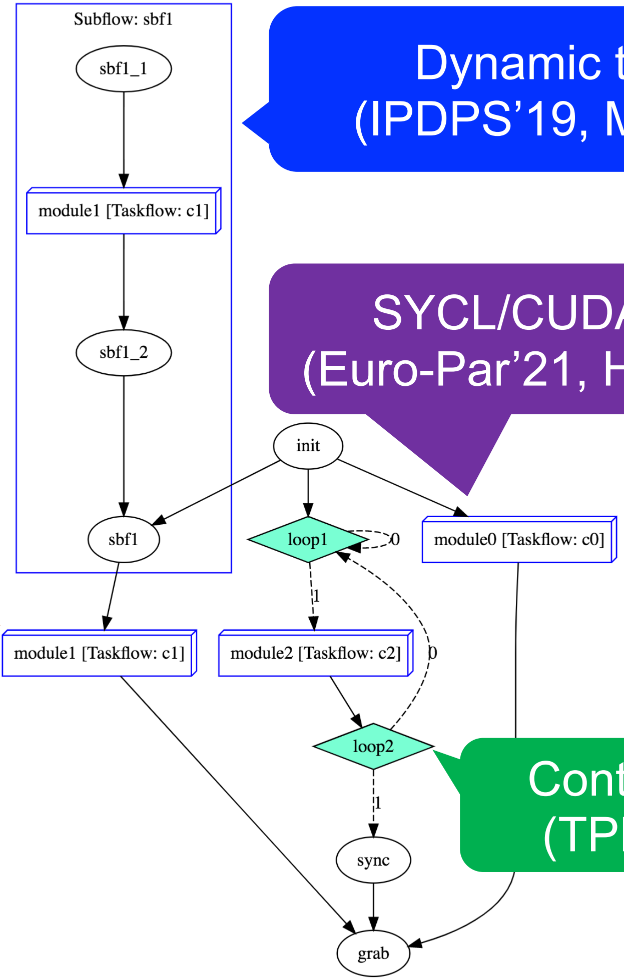


Industrial use-case of productivity improvement using Taskflow

jcelerier
ossia score

Reddit: https://www.reddit.com/r/cpp/ [under taskflow]

I've migrated https://ossia.io from TBB flow graph to taskflow a couple weeks ago. Net +8% of throughput on the graph processing itself, and took only a couple hours to do the change. Also don't have to fight with building the TBB libraries for 30 different platforms and configurations since it's header only.



Dynamic task
(IPDPS'19, MM'19)

SYCL/CUDA task
(Euro-Par'21, HPEC'20)

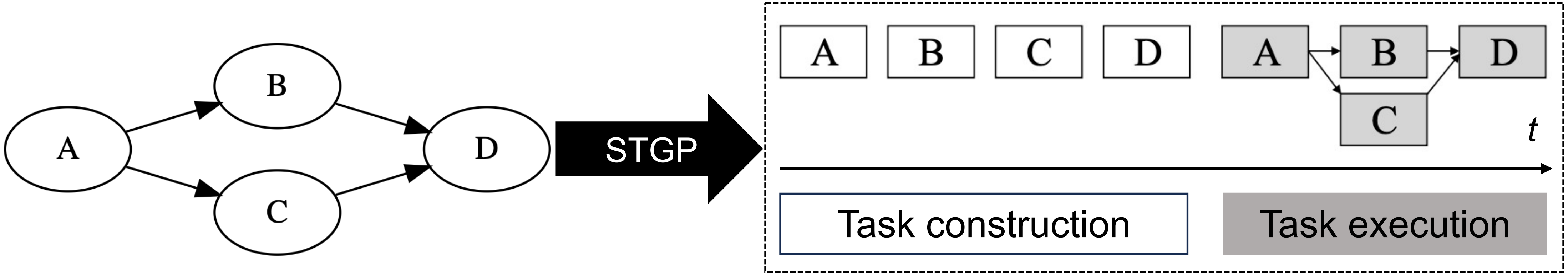
Control flow
(TPDS'22)

Takeaways

- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

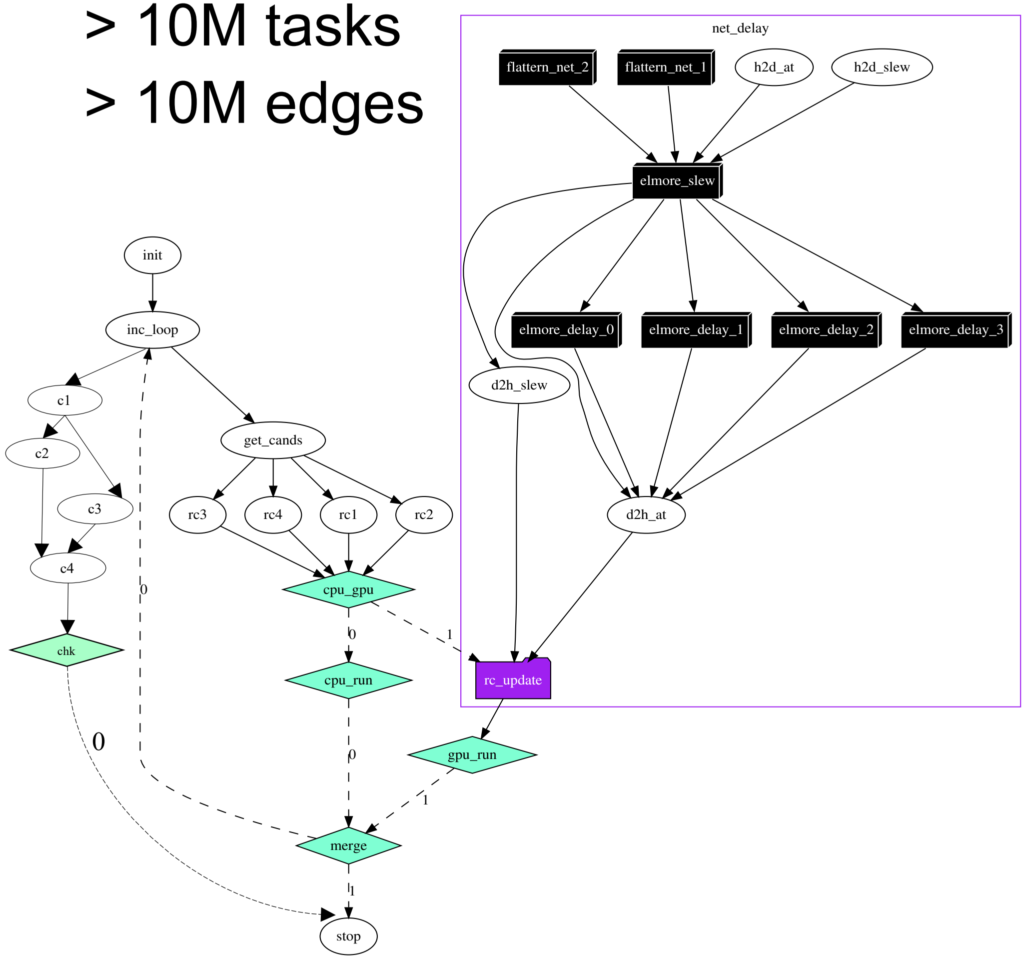
Static Task Graph Parallelism (STGP)

- **In STGP, the graph structure must be known up front**
  - Execution of STGP is based on the *construct-and-run* model
- **Lack of overlap between task construction and task execution**
  - For large task graphs (e.g., multi-million tasks and dependencies), such an overlap can bring a significant performance advantage
- **Lack of flexible and dynamic expression of TGP**
  - Task graph structure cannot depend on runtime values or control-flow results

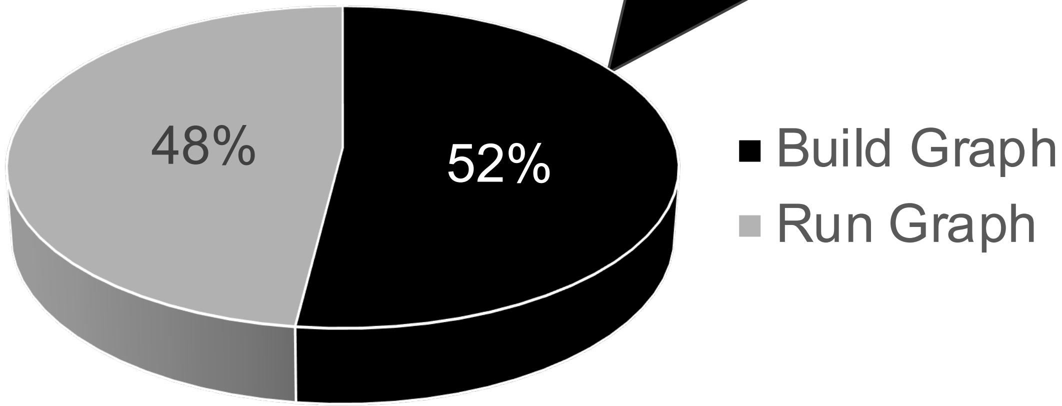


Problem of STGP: Example #1

- **Runtime breakdown of a task-parallel circuit analysis algorithm**[1]
  - > 10M tasks
  - > 10M edges



Task graph construction time takes over 50% of the entire runtime (typically done in one thread)



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

Problem of STGP: Example #2

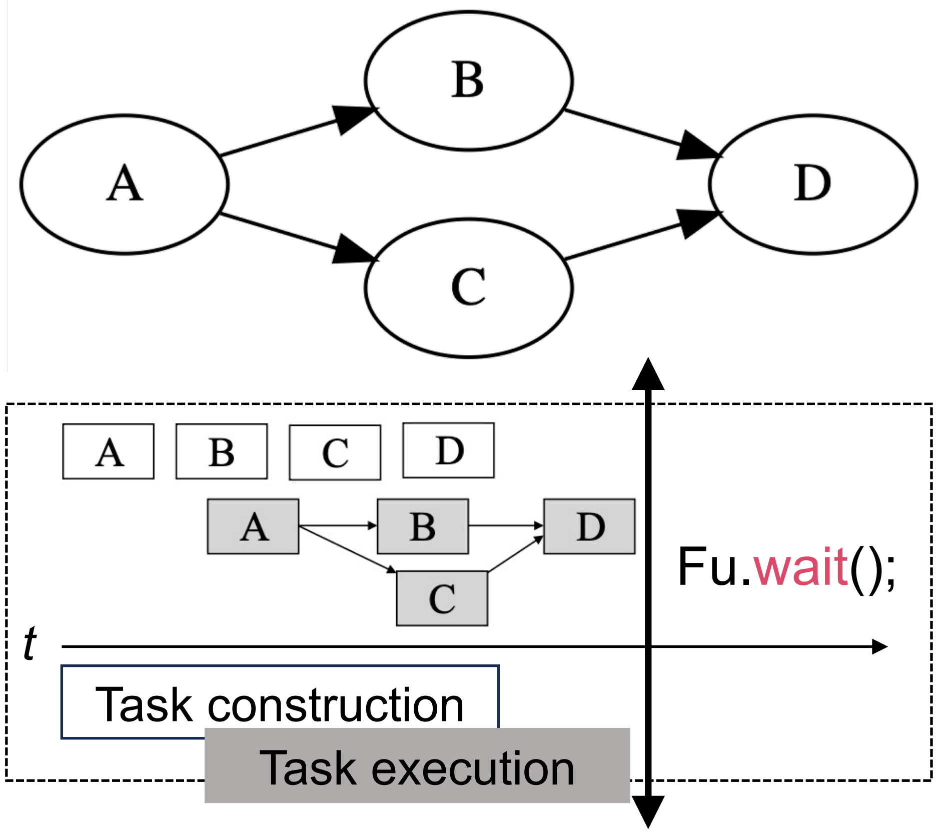
- **Express TGP that depends on runtime variables…?**

```
if (a == true) {
  G1 = build_task_graph1();
  if (b == true) {
    G2 = build_task_graph2();
    G1.precede(G2);
    if (c == true) {
      … // need another different TGP
    }
  }
  else {
    G3 = build_task_graph3();
    G3.precede(G1);
}
```

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

Dynamic TGP (DTGP) in Taskflow

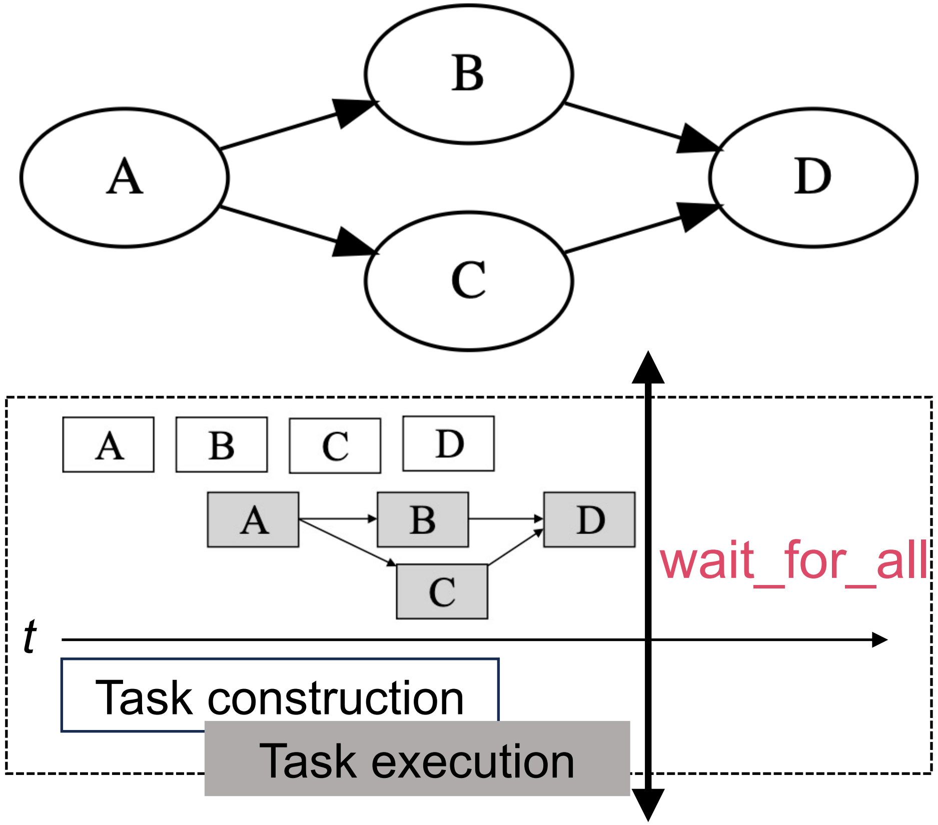
```
// Live: https://godbolt.org/z/j76ThGbWK
tf::Executor executor;
auto A = executor.silent_dependent_async([](){
    std::cout << "TaskA\n";
});
auto B = executor.silent_dependent_async([](){
    std::cout << "TaskB\n";
}, A);
auto C = executor.silent_dependent_async([](){
    std::cout << "TaskC\n";
}, A);
auto [D, Fu] = executor.dependent_async([](){
    std::cout << "TaskD\n";
}, B, C);
Fu.wait();
```



Specify arbitrary task dependencies using C++ variadic parameter pack

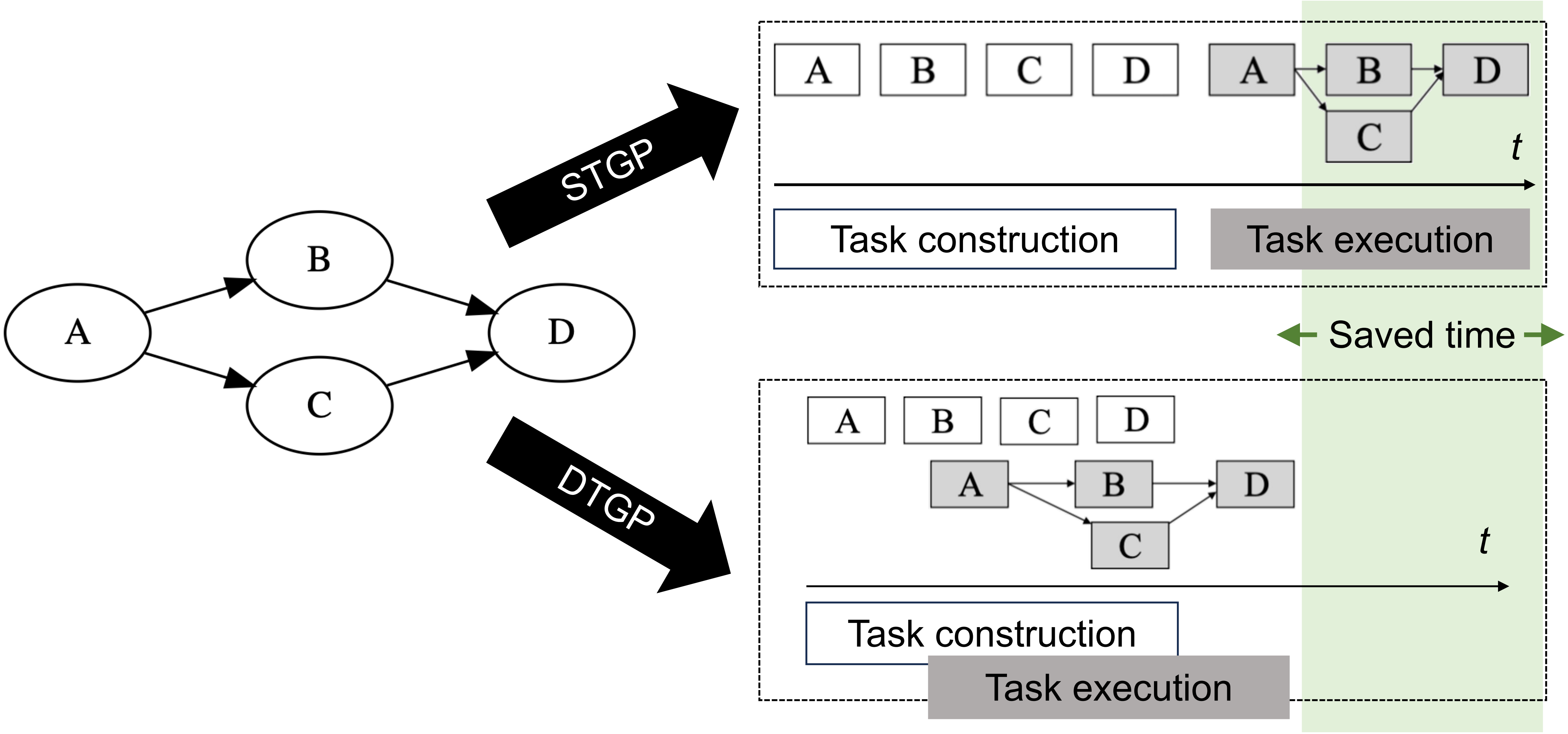
Wait for All Tasks to Finish

```
// 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();
```



Block the caller until all tasks (A, B, C, and D) finish

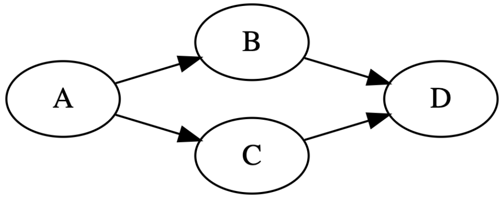
Comparison between STGP and DTGP

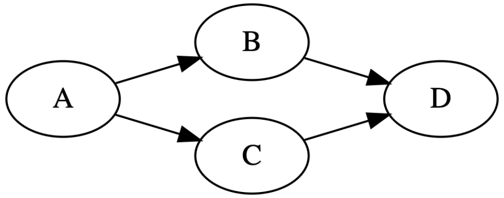


DTGP Needs a Correct Topological Order

```
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);
```

Topological order #1: A→B→C→D



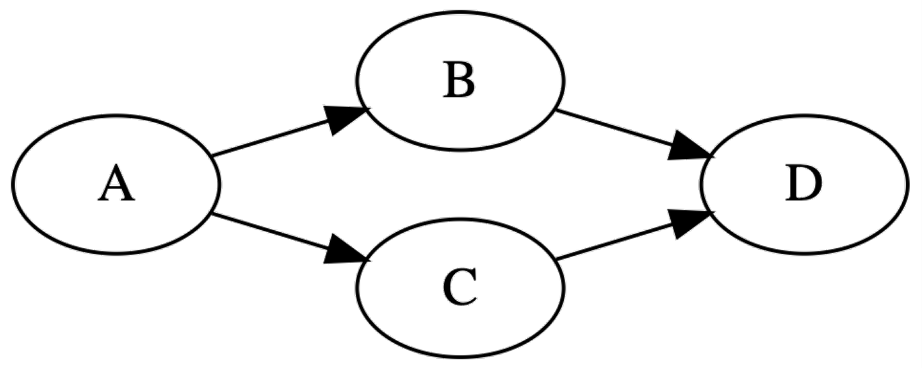


Topological order #2: A→C→B→D

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

Incorrect 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

Variable Range of Task Dependencies

- **Both methods accept a range of dependent tasks**
  - useful when the task dependencies come as a runtime variable

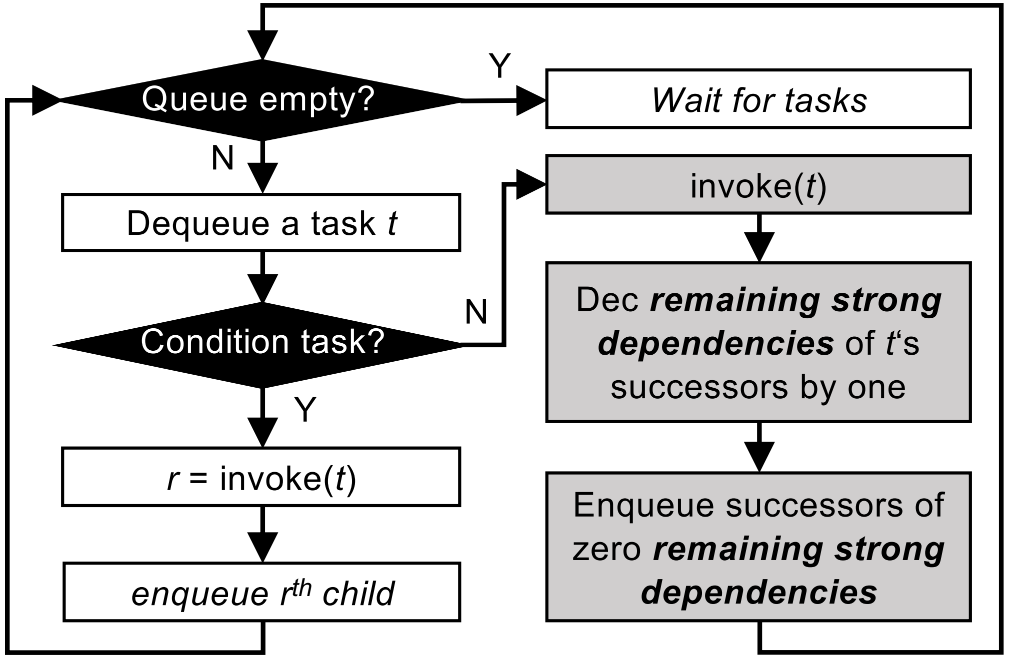
```
// Live: https://godbolt.org/z/6Pvco4KeE
std::vector<tf::AsyncTask> tasks = {
  executor.silent_dependent_async([](){ std::cout << "TaskA\n"; }),
  executor.silent_dependent_async([](){ std::cout << "TaskB\n"; }),
  executor.silent_dependent_async([](){ std::cout << "TaskC\n"; }),
  executor.silent_dependent_async([](){ std::cout << "TaskD\n"; })
};
// create a dependent-async tasks that depends on tasks, A, B, C, and D
executor.dependent_async([](){}, tasks.begin(), tasks.end());

// create a silent-dependent-async tasks that depends on tasks, A, B, C, and D
executor.silent_dependent_async([](){}, tasks.begin(), tasks.end());
```

Takeaways

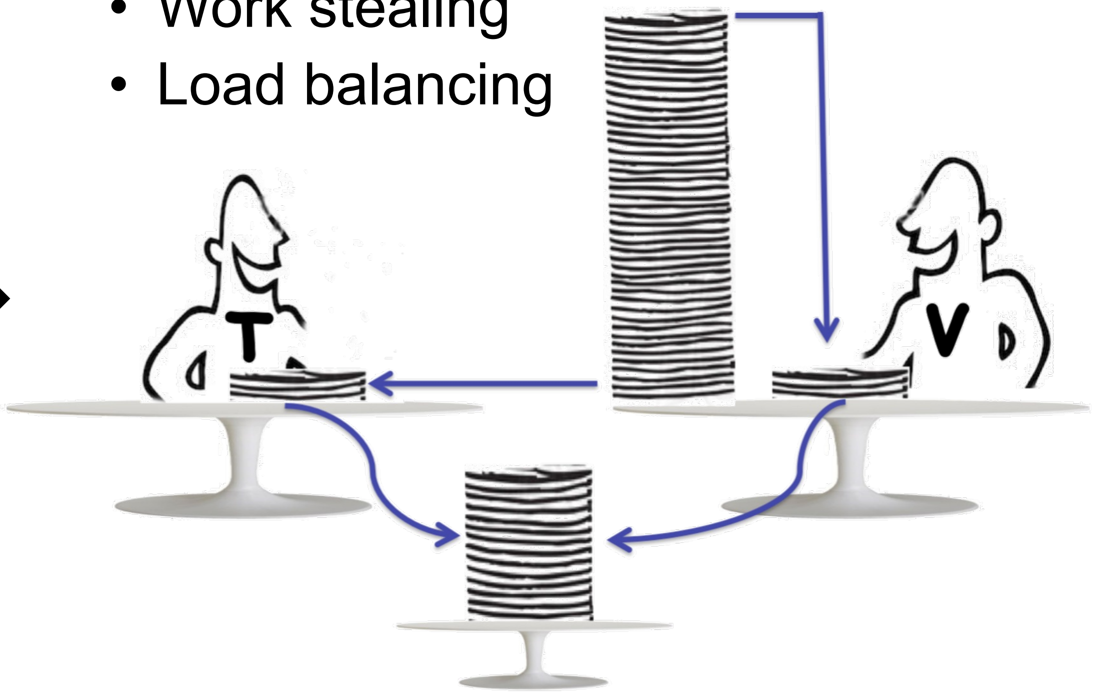
- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

STGP Scheduling Algorithm

- **Task-level scheduling**


Queue empty?
Y
Wait for tasks
N
Dequeue a task *t*
invoke(*t*)
Condition task?
N
Dec ***remaining strong dependencies*** of *t*'s successors by one
Y
r = invoke(*t*)
enqueue r^{th} child
Enqueue successors of zero ***remaining strong dependencies***

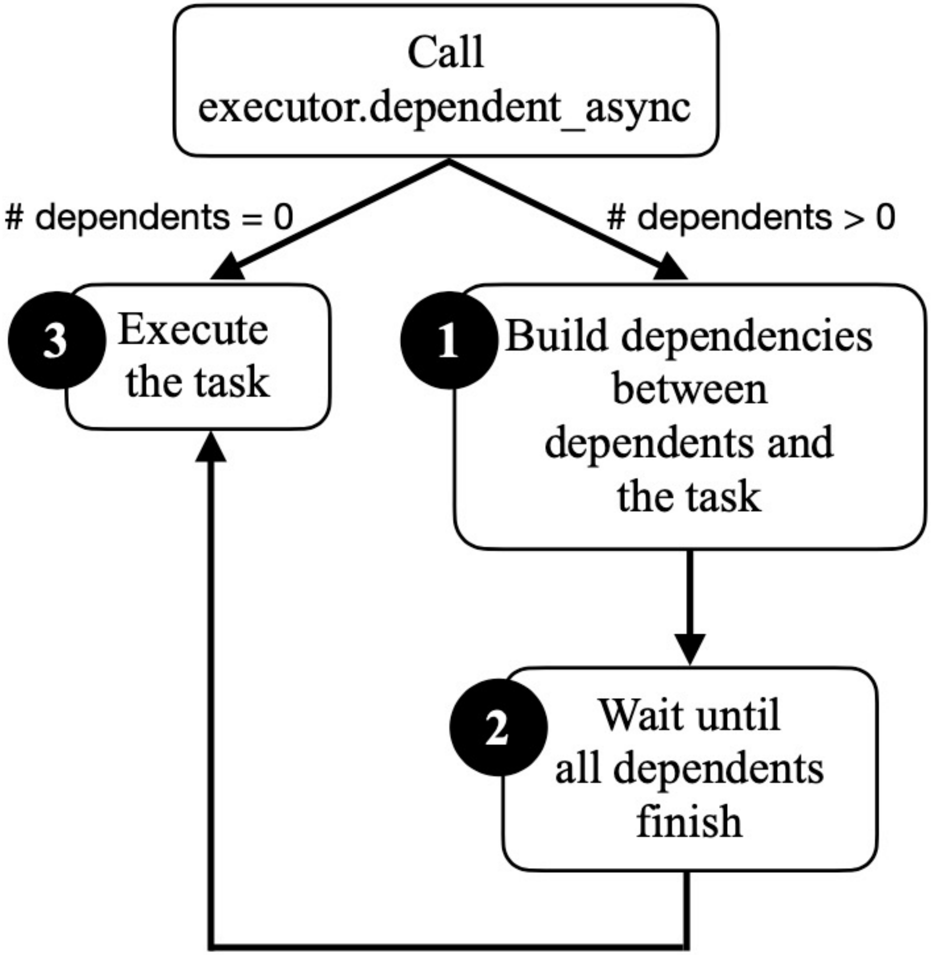
- **Worker-level scheduling**
  - Work stealing
  - Load balancing



Key results: schedule tasks with in-graph control flow with a strong balance between the number of active workers and dynamically generated tasks – *low latency, energy efficient, and high throughput*

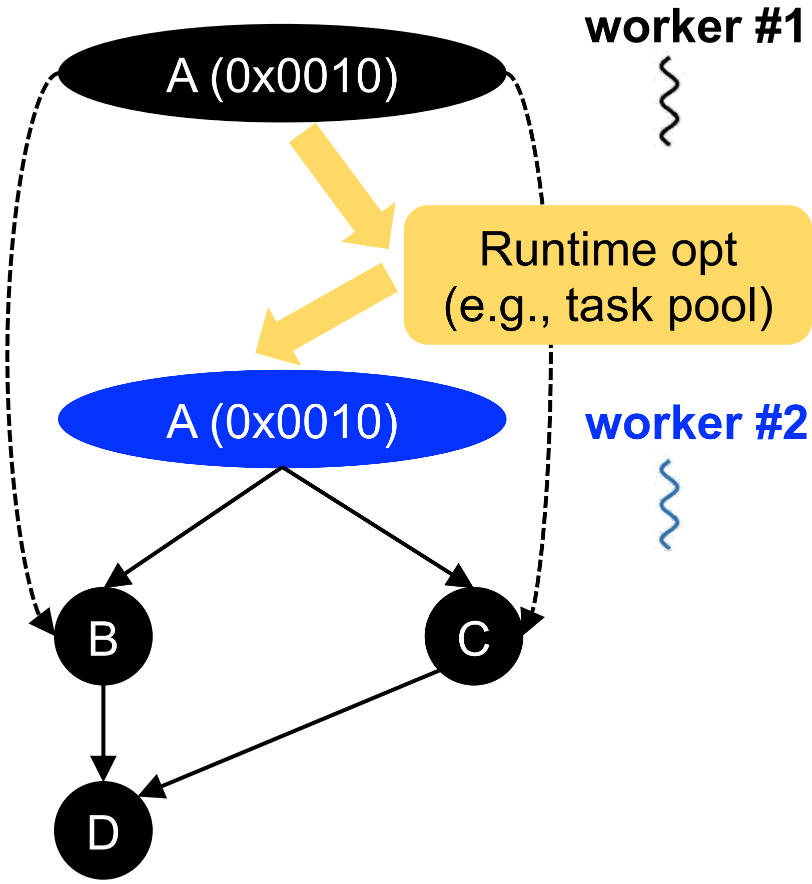
DTGP Scheduling Algorithm

- **The algorithm has three parts:**
  - Build dependencies
  - Wait for dependents to finish
  - Execute the task
- **Three key scheduling challenges:**
  1. **ABA** – *a specified dependent task must exist correctly*
  2. **Data race** – *multiple threads may simultaneously add and remove successors to and from a task*
  3. **Synchronization** – *application can issue a global synchronization at anytime to wait for all tasks to finish*



Solving Challenge #1: ABA Problem[1]

```
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();
```



[1] : ABA Problem: https://en.wikipedia.org/wiki/ABA_problem

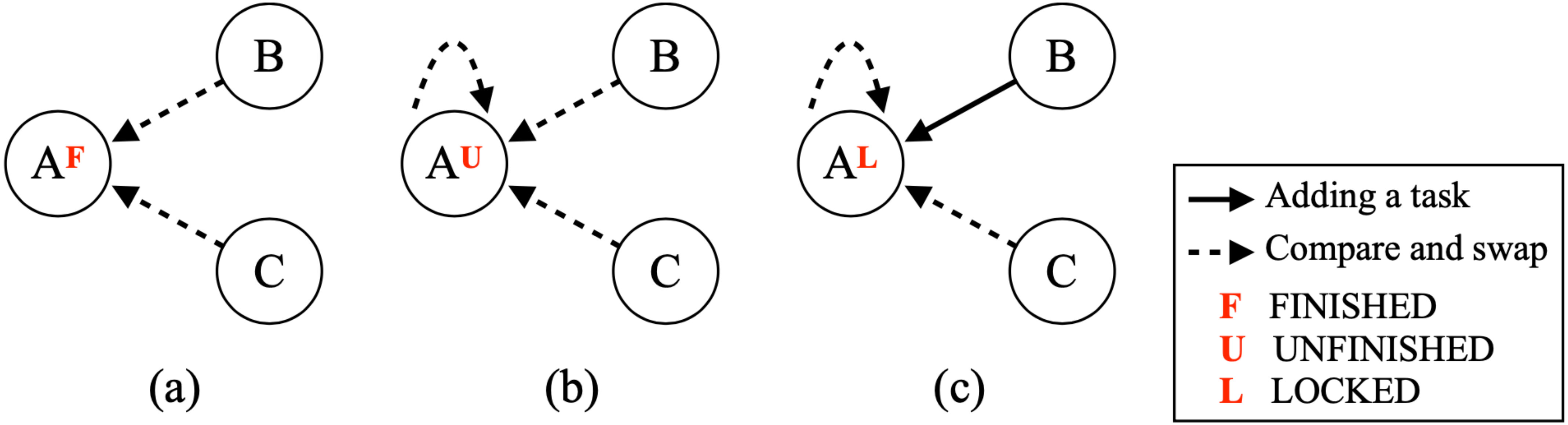
Retain Shared Ownership of Every Task

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

tf::AsyncTask acks like a std::shared_ptr to ensure tasks stay alive when they are used

Solving Challenge #2: Data Race

- **Both B and C want to add themselves to the successors of A**
  - In the meantime, A may want to remove its successor
- **Apply compare-and-swap (CAS) to enable exclusive access**
  - As a result, constructing a dynamic task graph can be completely thread-safe



Solving Challenge #3: Synchronization

- **Application can issue a global synchronization at any time**
  - `executor.wait_for_all();`

```
tf::Executor executor;
auto A  = executor.silent_dependent_async([](){});
auto B  = executor.silent_dependent_async([](){}, A);
executor.wait_for_all();  // wait for A and B to finish

auto C  = executor.silent_dependent_async([](){}, A);
auto D  = executor.silent_dependent_async([](){}, B, C);
executor.wait_for_all();  // wait for C and D to finish
```

```
// lock-based solution
std::unique_lock lock(mutex);
cv.wait(lock, [&](){
   return num_tasks == 0;
});
```

```
// atomic wait-based solution
auto n = num_tasks.load();
while(n != 0) {
   num_tasks.wait(n);
   n = num_tasks.load();
});
```

Lock-free Scheduling Algorithm[1]

Algorithm 1 dependent_async(callable, deps)

```
Create a $future$
$num\_deps \leftarrow$ sizeof($deps$)
$task \leftarrow$ initialize_task($callable, num\_deps, future$)
for all $dep \in deps$ do
    process_dependent($task, dep, num\_deps$)
end for
if $num\_deps == 0$ then
    schedule_async_task($task$)
end if
return $(task, future)$
```

Algorithm 2 process_dependent(task, dep, num_deps)

```
1:  $dep\_state \leftarrow dep.state$
2:  $target\_state \leftarrow UNFINISHED$
3:  if $dep\_state$.CAS($target\_state, LOCKED$) then
4:      $dep.successors$.push($task$)
5:      $dep\_state \leftarrow UNFINISHED$
6:  else if $target\_state == FINISHED$ then
7:      $num\_deps \leftarrow$ AtomDec($task.join\_counter$)
8:  else
9:      goto line 2
10: end if
```

Algorithm 3 schedule_async_task(task)

```
$target\_state \leftarrow UNFINISHED$
while not $task.state$.CAS($target\_state, FINISHED$) do
    $target\_state \leftarrow UNFINISHED$
end while
Invoke($task.callable$)
for all $successor \in task.successors$ do
    if AtomDec($successor.join\_counter$) $== 0$ then
        schedule_async_task($successor$)
    end if
end for
if AtomDec($task.ref\_count$) $== 0$ then
    Delete $task$
end if
```

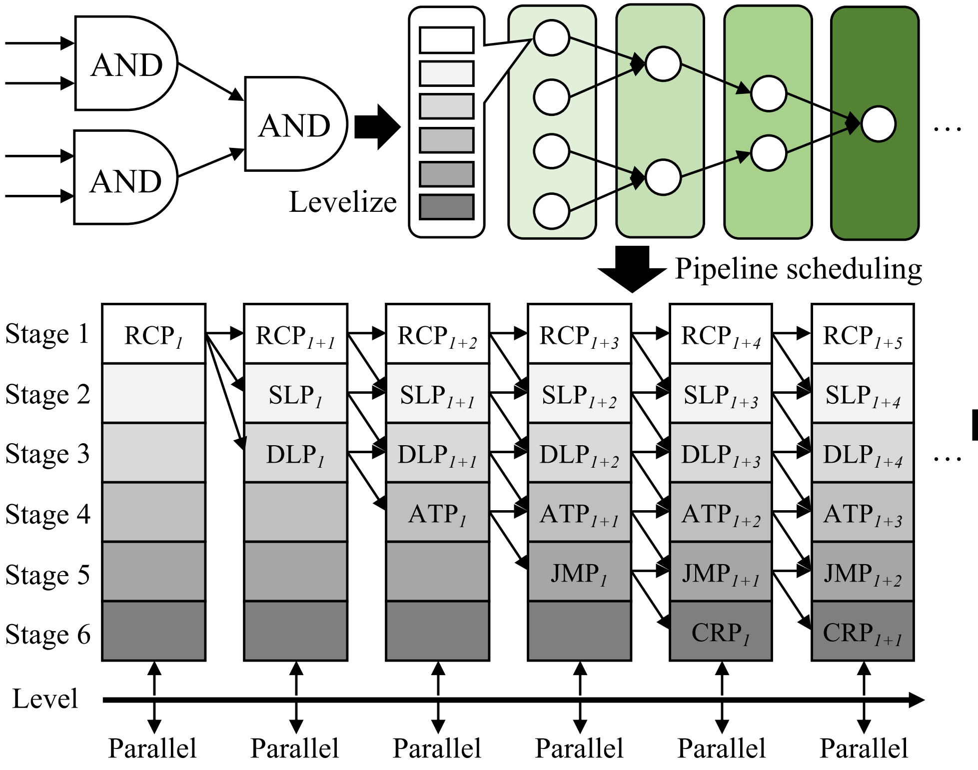
[1]: Cheng-Hsiang Chiu, et. al, "Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms," *IEEE/ACM ICCAD*, CA, 2023

Takeaways

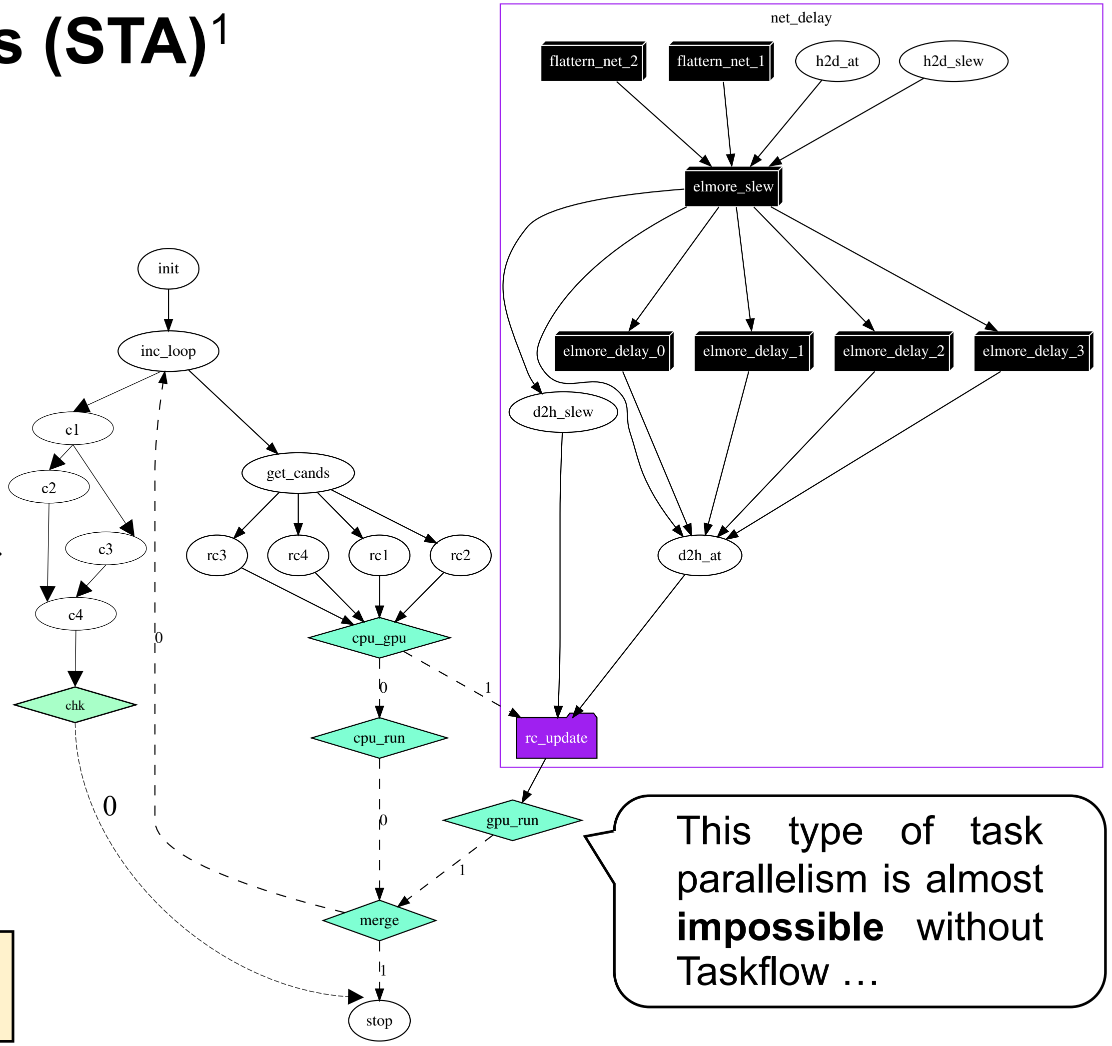
- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

Case Study 1: Task-parallel STA w/ STGP

- **Accelerating static timing analysis (STA)**[1]



Stage 1	RCP$_l$	RCP$_{l+1}$	RCP$_{l+2}$	RCP$_{l+3}$	RCP$_{l+4}$	RCP$_{l+5}$
Stage 2		SLP$_l$	SLP$_{l+1}$	SLP$_{l+2}$	SLP$_{l+3}$	SLP$_{l+4}$
Stage 3		DLP$_l$	DLP$_{l+1}$	DLP$_{l+2}$	DLP$_{l+3}$	DLP$_{l+4}$
Stage 4			ATP$_l$	ATP$_{l+1}$	ATP$_{l+2}$	ATP$_{l+3}$
Stage 5				JMP$_l$	JMP$_{l+1}$	JMP$_{l+2}$
Stage 6					CRP$_l$	CRP$_{l+1}$
	Parallel	Parallel	Parallel	Parallel	Parallel	Parallel

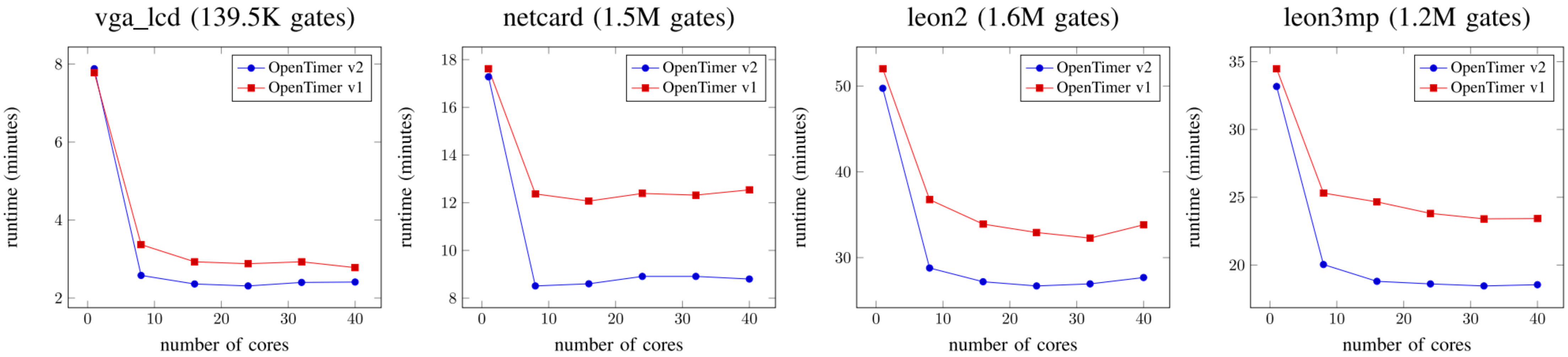


This type of task parallelism is almost **impossible** without Taskflow …

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

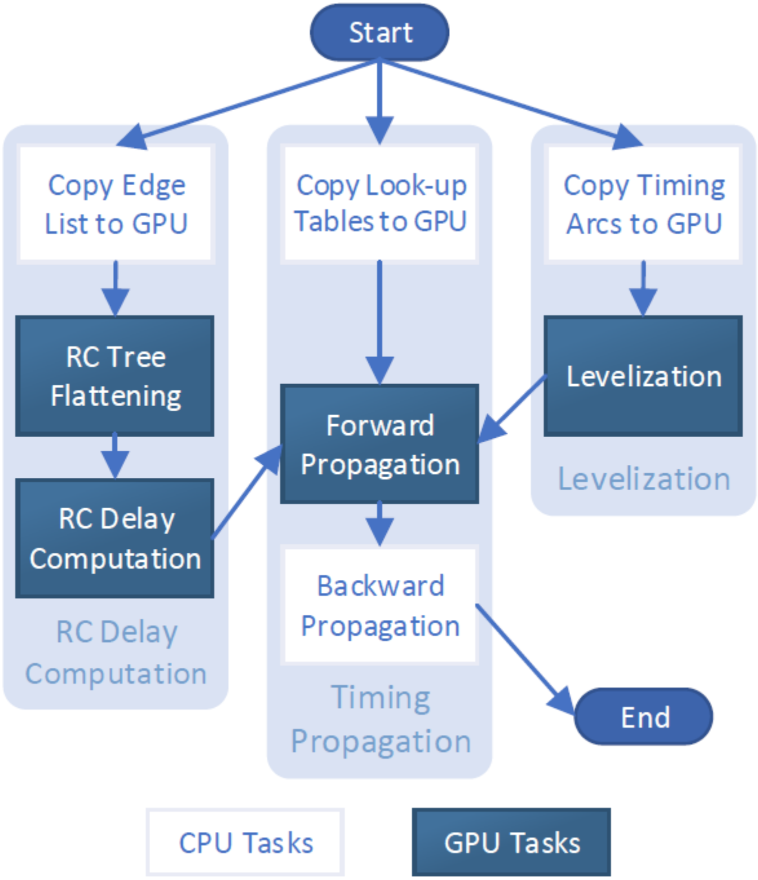
Levelization-based vs Task-parallel STA

- **OpenTimer v1: levelization-based parallel timing propagation**[1]
  - Implemented using OpenMP “parallel_for” primitive
- **OpenTimer v2: task-parallel timing propagation**[2]
  - Implemented using Taskflow (https://taskflow.github.io/)



[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

Our Research on Task-parallel STA



Start

Copy Edge List to GPU

RC Tree Flattening

RC Delay Computation

RC Delay Computation

Copy Look-up Tables to GPU

Forward Propagation

Backward Propagation

Timing Propagation

Copy Timing Arcs to GPU

Levelization

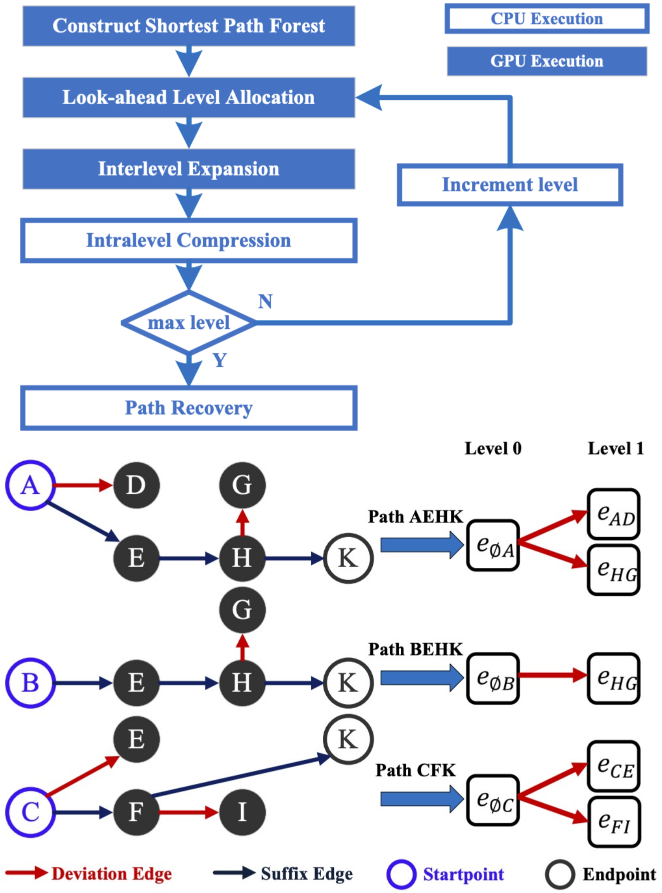
Levelization

End

CPU Tasks

GPU Tasks

GPU-based graph analysis (ICCAD'20)



Construct Shortest Path Forest

Look-ahead Level Allocation

Interlevel Expansion

Intralevel Compression

max level

N

Y

Path Recovery

Increment level

CPU Execution

GPU Execution

Level 0

Level 1

A D G E H K

Path AEHK

$e_{\emptyset A}$ e_{AD} e_{HG}

G B E H K

Path BEHK

$e_{\emptyset B}$ e_{HG}

E K C F I

Path CFK

$e_{\emptyset C}$ e_{CE} e_{FI}

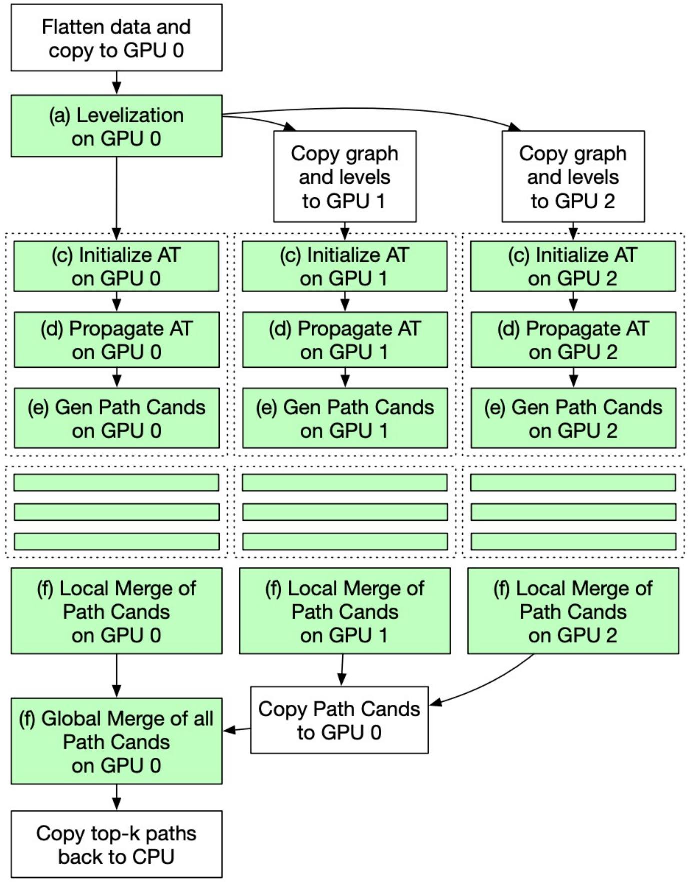
Deviation Edge

Suffix Edge

Startpoint

Endpoint

GPU-based path analysis (DAC'21)



Flatten data and copy to GPU 0

(a) Levelization on GPU 0

Copy graph and levels to GPU 1

Copy graph and levels to GPU 2

(c) Initialize AT on GPU 0

(c) Initialize AT on GPU 1

(c) Initialize AT on GPU 2

(d) Propagate AT on GPU 0

(d) Propagate AT on GPU 1

(d) Propagate AT on GPU 2

(e) Gen Path Cands on GPU 0

(e) Gen Path Cands on GPU 1

(e) Gen Path Cands on GPU 2

(f) Local Merge of Path Cands on GPU 0

(f) Local Merge of Path Cands on GPU 1

(f) Local Merge of Path Cands on GPU 2

(f) Global Merge of all Path Cands on GPU 0

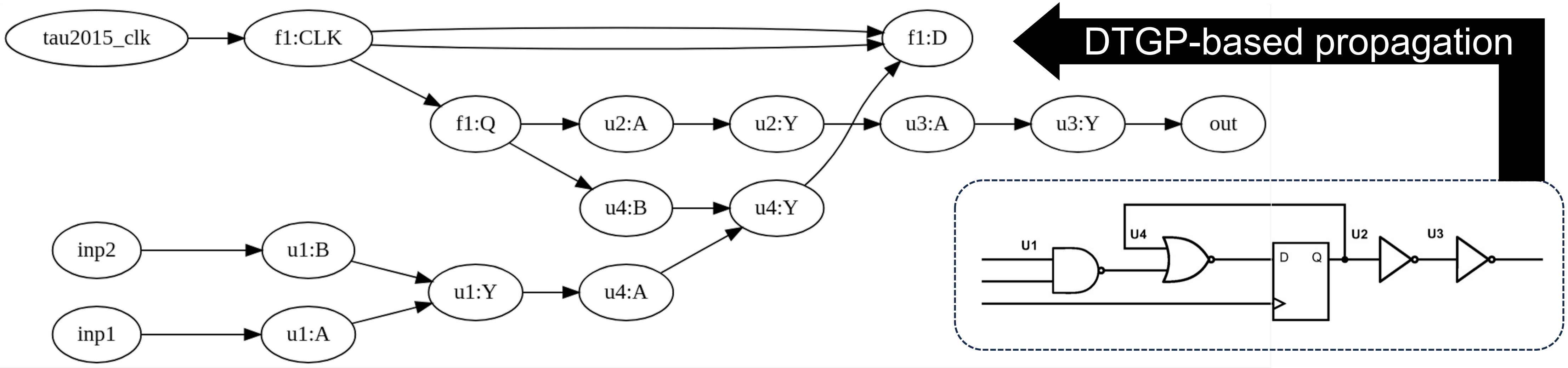
Copy Path Cands to GPU 0

Copy top-k paths back to CPU

GPU-based CPPR (ICCAD'21)

Case Study 2: Task-parallel STA w/ DTGP

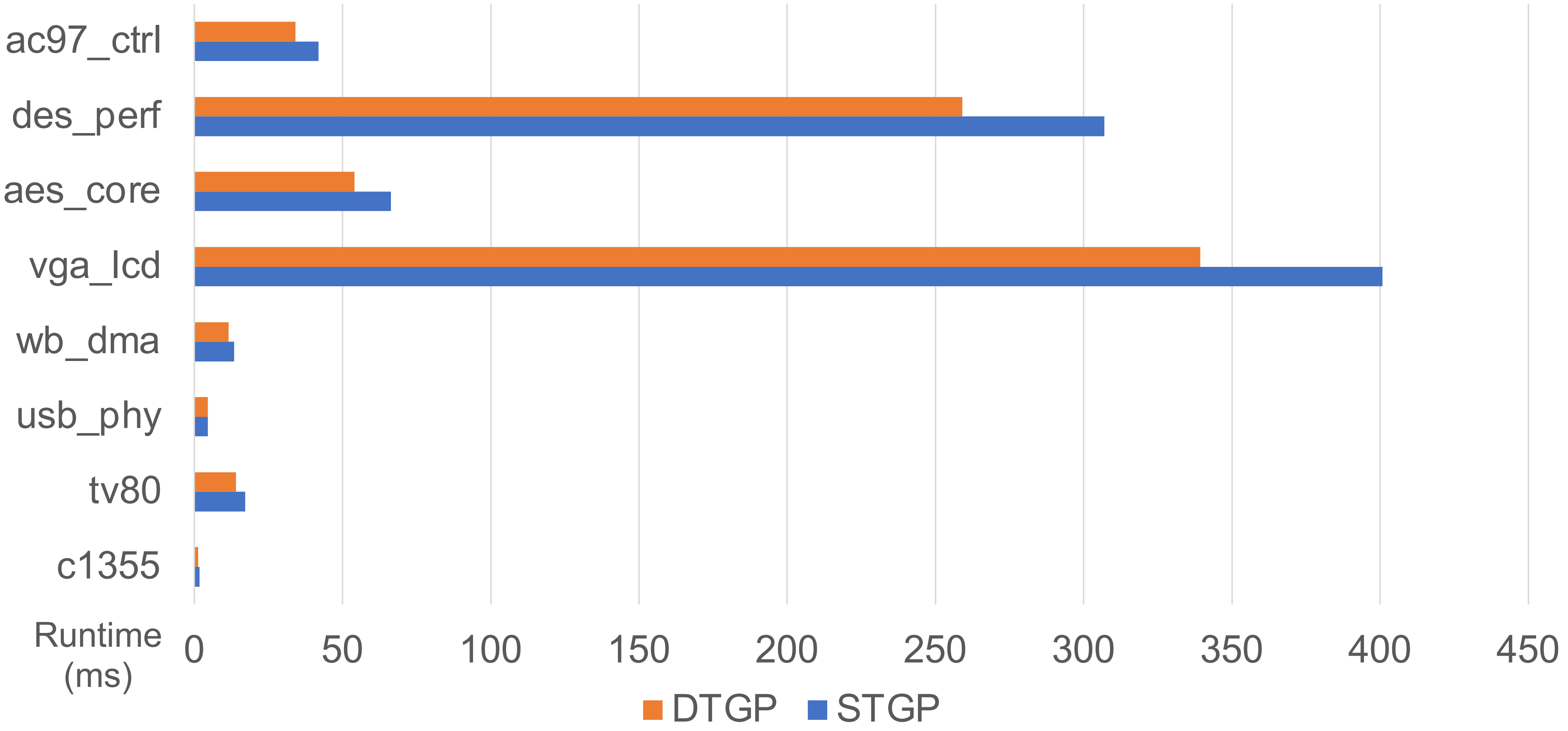
- **STGP works pretty well for task-parallel static timing analysis**
  - However, STGP may result in suboptimal performance for large circuits
  - Why? constructing a large task graph can be “very” time-consuming …
- **Reformulated the timing propagation into a dynamic task graph**
  - Ex (below): a task graph for a full-timing propagation on a five-gate circuit



17 tasks & 18 dependencies

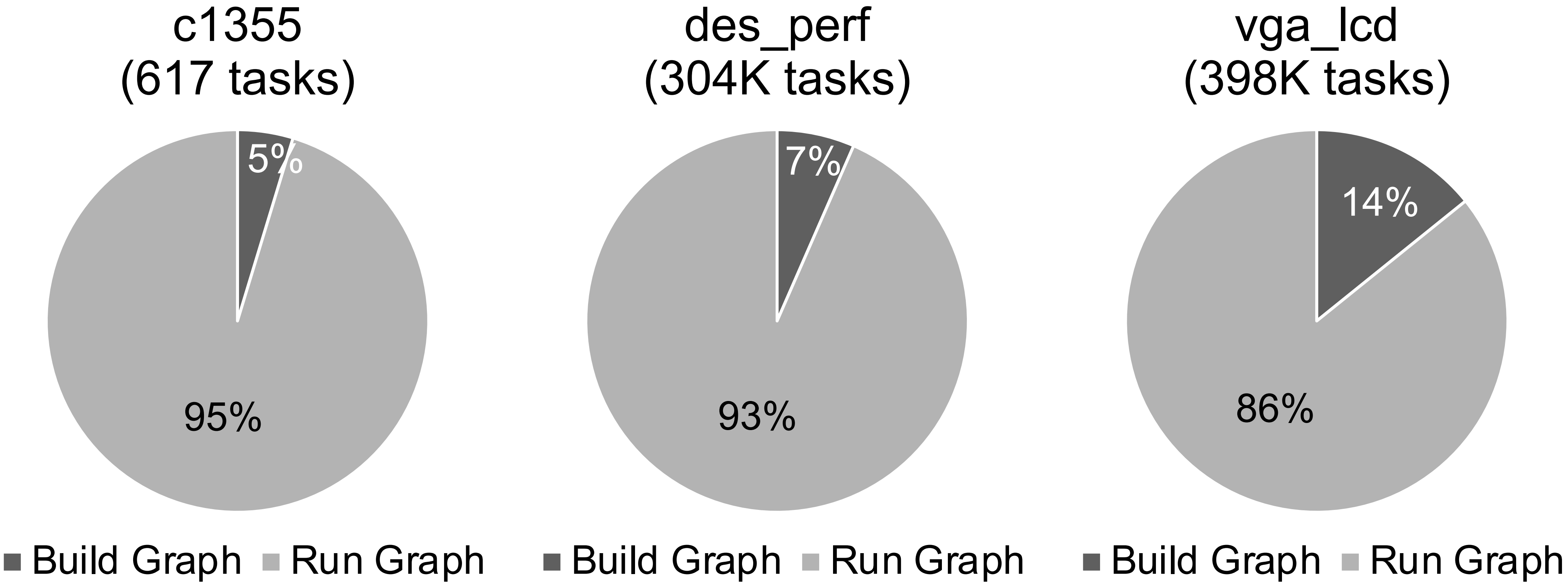
[1]: T.-W. Huang, et. al, ”OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776-789, April 2021

Runtime Comparison: STGP vs DTGP



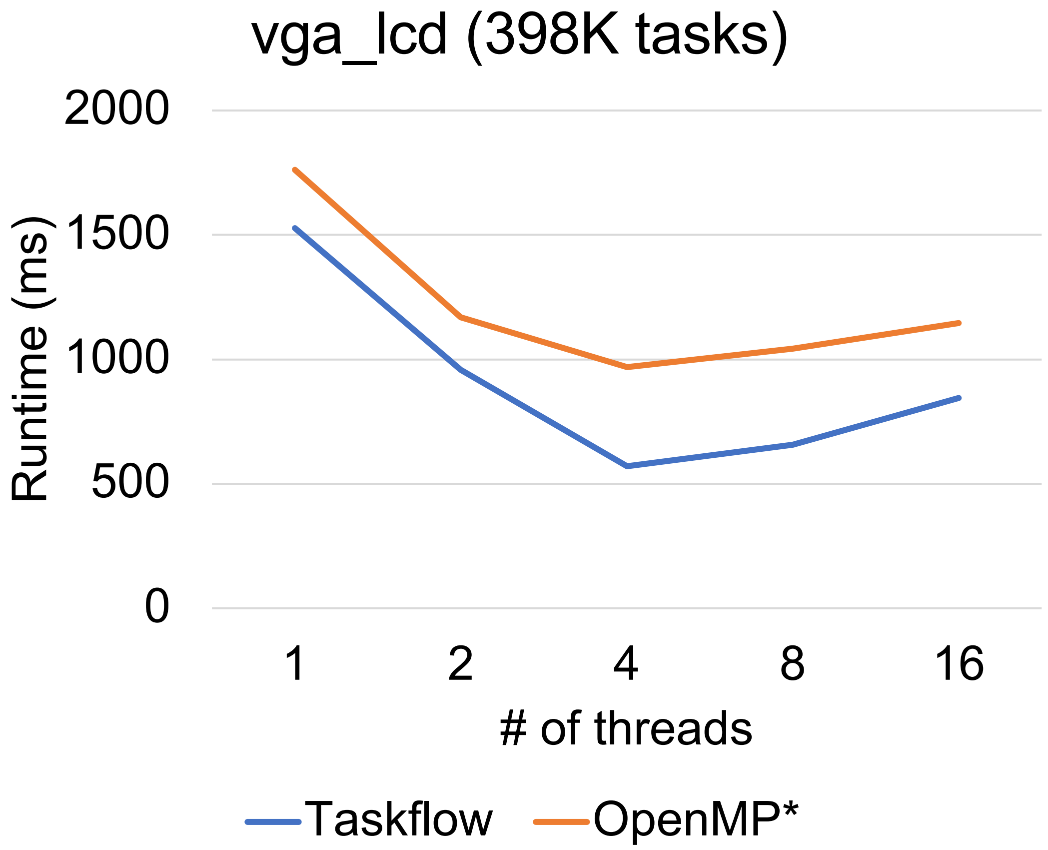
Runtime Breakdown of STGP

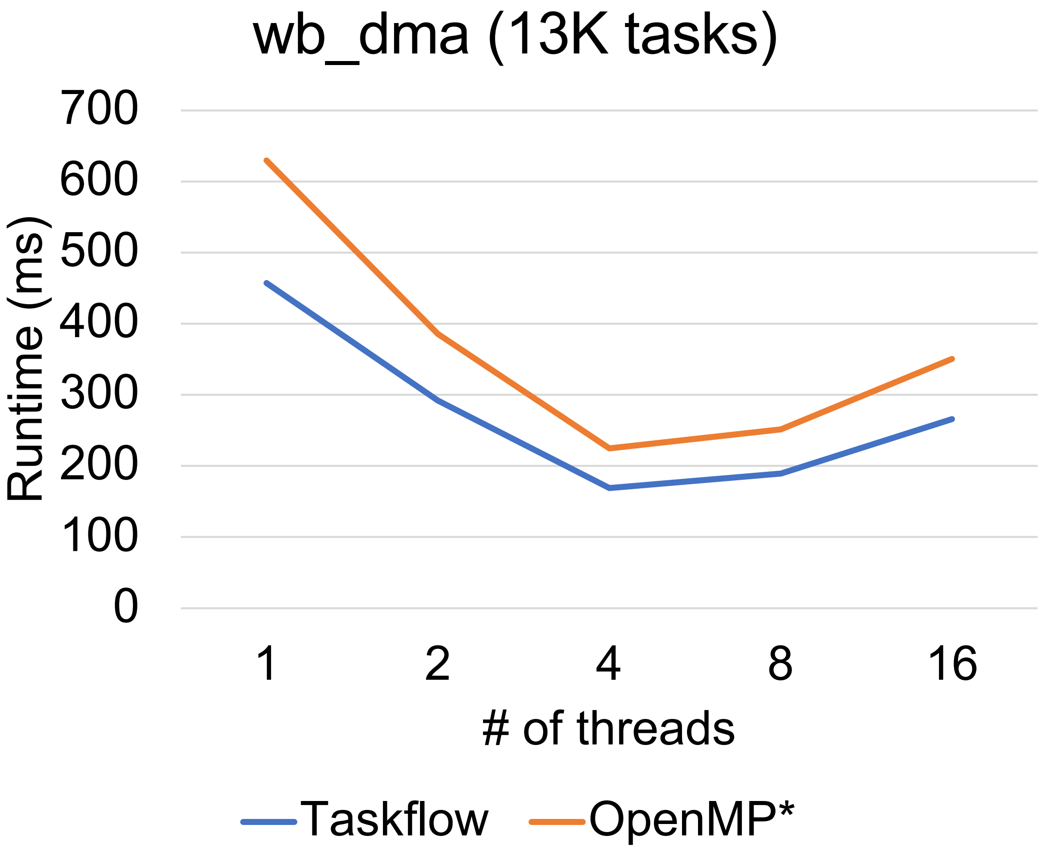
- **Graph construction time increases as the circuit size increases**



Runtime Comparison with OpenMP*

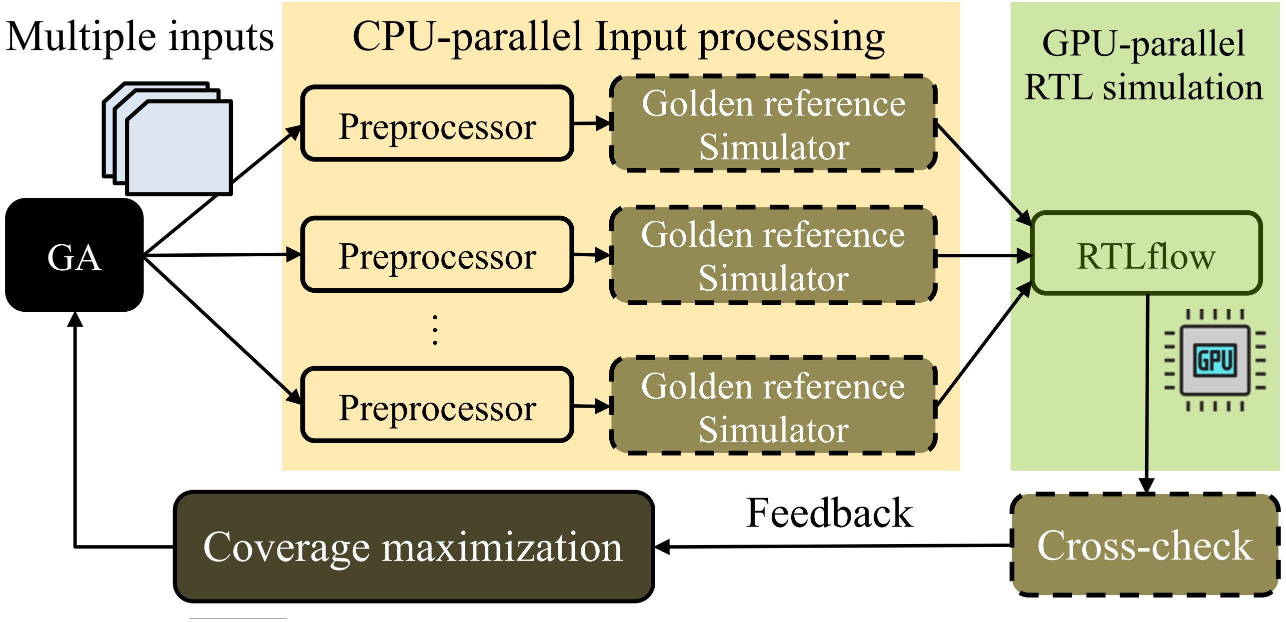
- **OpenMP*: Revised Libomp scheduling algorithm with Taskflow**

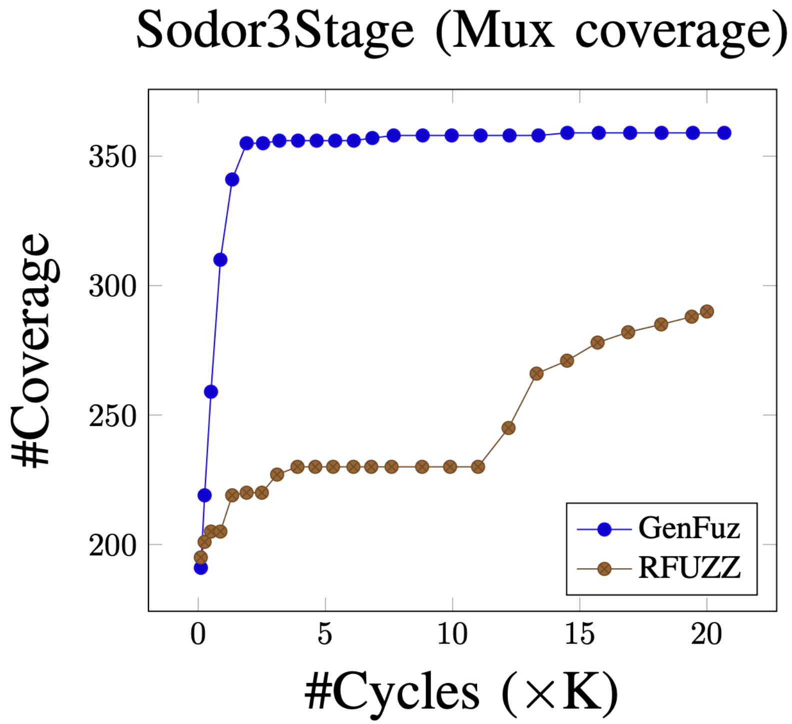




Case Study 3: Hardware Fuzzing[1]

- **Applied Taskflow to accelerate hardware fuzzing**
  - A new genetic algorithm to largely improve coverage quality using GPU
  - **10–80x** faster over existing fuzzers and found undiscovered hardware bugs





[1]: Dian-Lun Lin, et al, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," *ACM/IEEE DAC*, CA, 2023

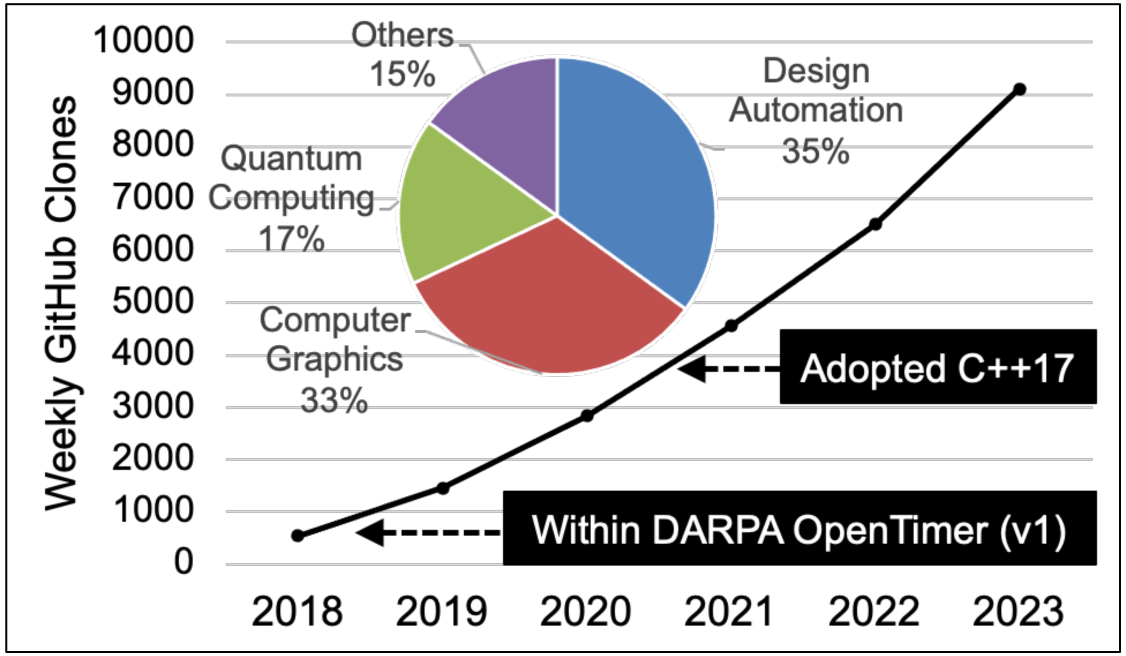
Other Industrial Applications of Taskflow

- **Quantum computing**
  - Xanadu deploys Taskflow in their quantum simulator
- **Computer graphics/rendering**
  - Vulkan officially recommends using Taskflow
- **FPGA synthesis**
  - Vivado uses Taskflow for synthesis
- **Embedded/edge computing**
  - Tesseract (robotics planning)
  - Cruise (autonomous car)
  - Reveal.Tech (drone vision)
  - Tesseract Robotic (planning tool)
  - …

KHRONOS GROUP

Vulkan

https://vkguide.dev/docs/extra-chapter/multithreading/



Conclusion

- **Expressed your parallelism in the right way**
- **Programmed static task graph parallelism using Taskflow**
- **Programmed dynamic task graph parallelism using Taskflow**
- **Overcame the scheduling challenges**
- **Demonstrated the efficiency of Taskflow**
- **Concluding the talk**

Thank You for using Taskflow!

OT

ossia

deal.II

MYDATAMODELS

XANADU

LiveHD

ROS industrial

REVEAL

SNIG Inference Engine

24h

OPERON

R

NVIDIA.

Explosion

METHANE powered graphics

Chainblocks

Rapid Fuzz

JET BRAINS RESEARCH

atomicDEX

ReAgent

RAPIDS

XILINX

TESSERACT

COCOS

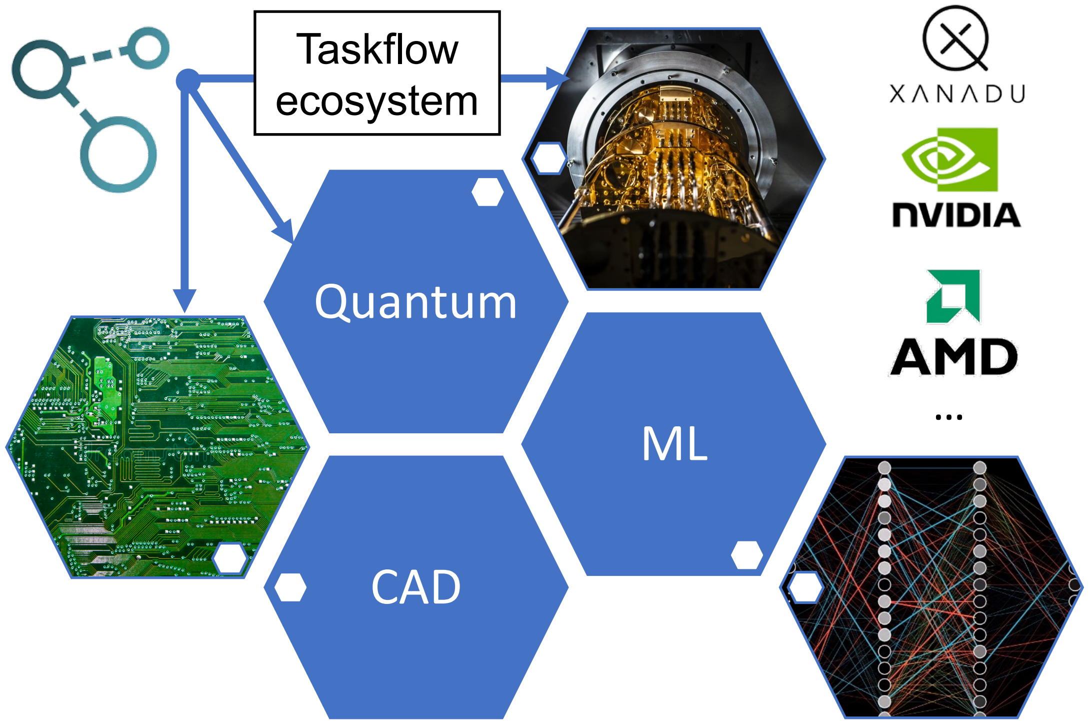
MUNICH QUANTUM TOOLKIT

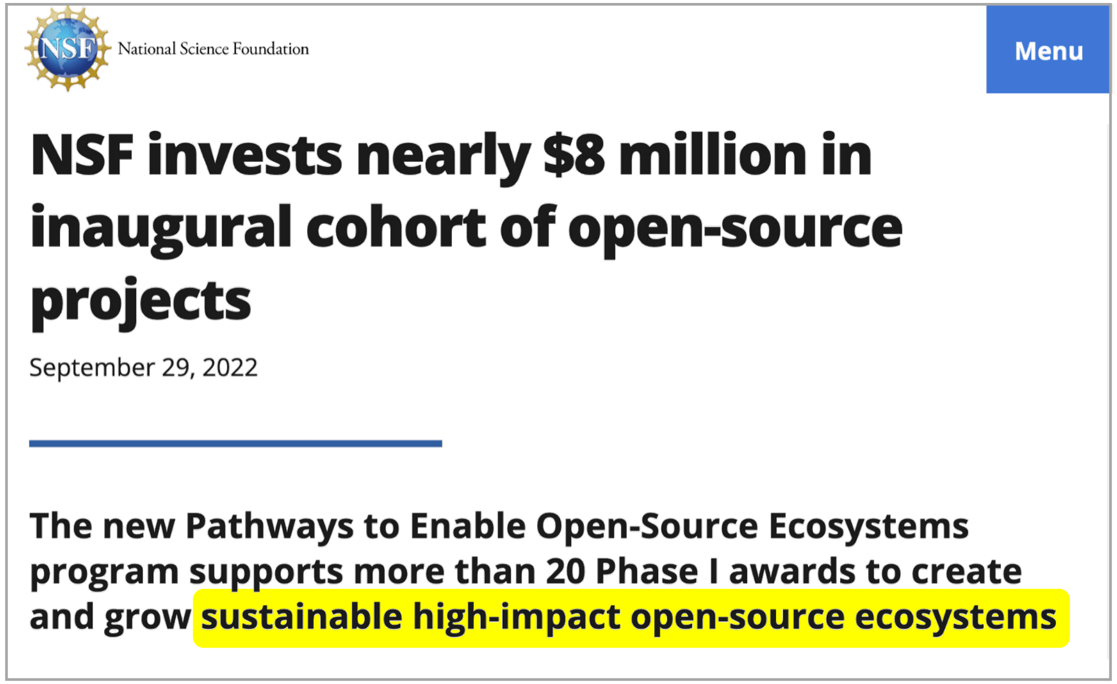
ModuleWorks

…

Our NSF POSE Project[1]: Sustainability

- **Create a sustainable Taskflow application ecosystem**





NSF invests nearly $8 million in inaugural cohort of open-source projects

September 29, 2022

The new Pathways to Enable Open-Source Ecosystems program supports more than 20 Phase I awards to create and grow sustainable high-impact open-source ecosystems

https://beta.nsf.gov/tip/updates/nsf-invests-nearly-8-million-inaugural-cohort-open

[1]: “POSE: Phase I: Toward a Task-Parallel Programming Ecosystem for Modern Scientific Computing,” $298K, 09/15/2022—08/31/2023, NSF POSE, TI-2229304

Thank you for Sponsoring Taskflow!

NSF

DARPA

NUMFOCUS
OPEN CODE = BETTER SCIENCE

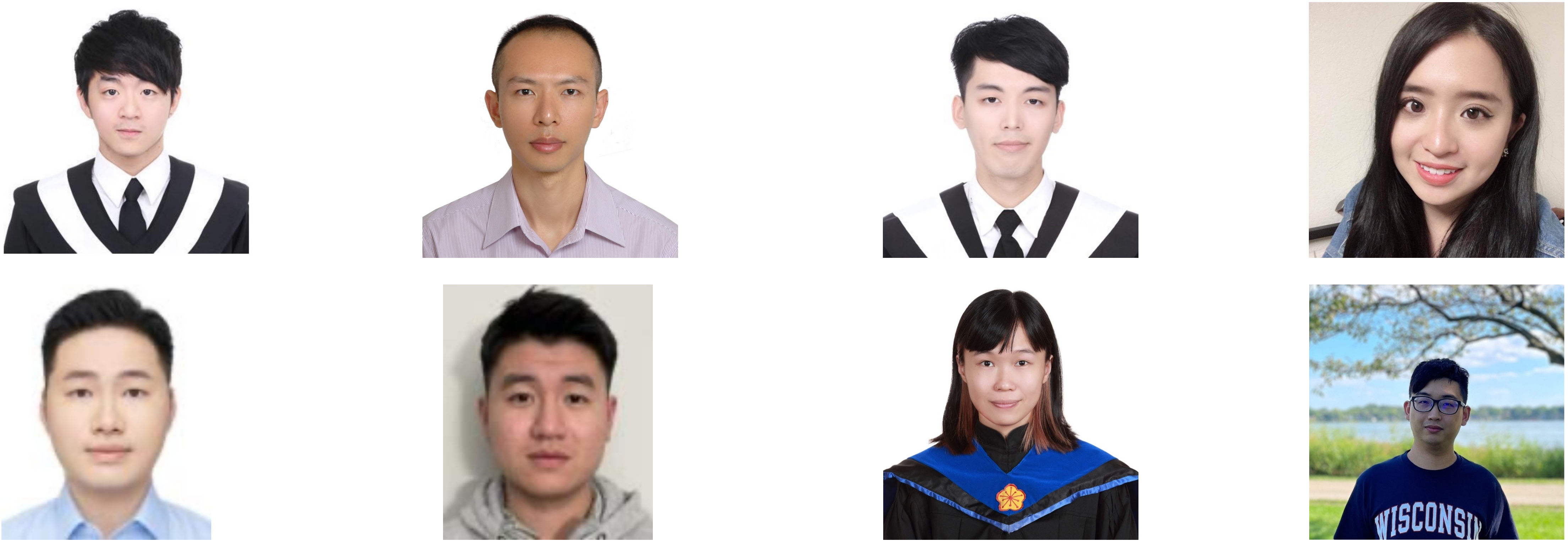
intel

Google Summer of Code

NVIDIA

Acknowledgment: Excellent PhD Students

- **Please contact me if you have any intern/full-time openings!**
  - We specialize in CAD, HPC, and modern C++ heterogeneous programming
  - https://tsung-wei-huang.github.io/team/ (or tsung-wei.huang@wisc.edu)



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();
return 0;
```

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();
```