Taskflow: A General-purpose Task-parallel Programming System

Dr. Tsung-Wei (TW) Huang, Assistant Professor

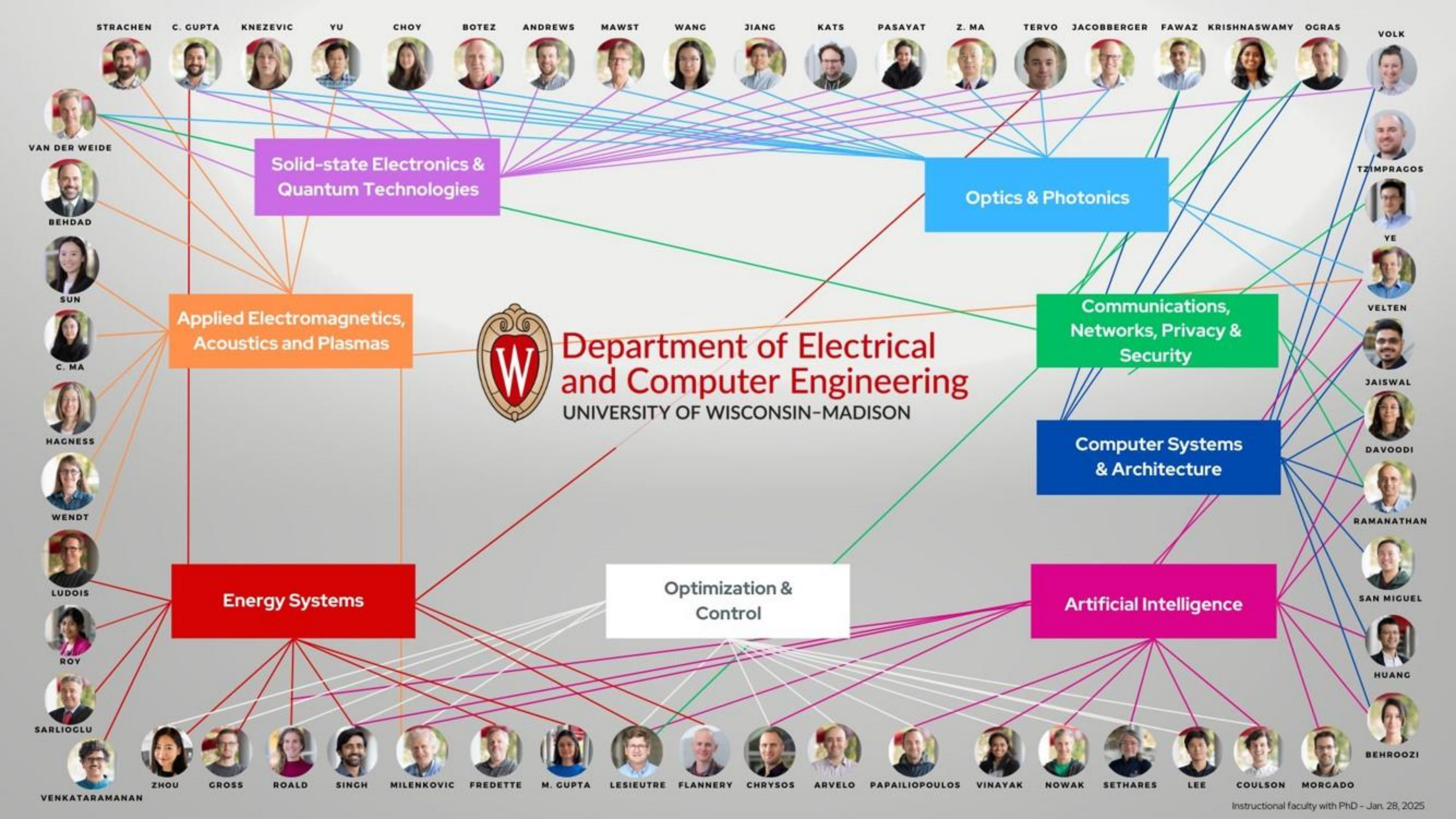
Department of Electrical and Computer Engineering

University of Wisconsin at Madison, Madison, WI

https://tsung-wei-huang.github.io/

About UW-Madison ECE Department

- **Located at the heart of the campus in the beautiful Madison city**
  - Surrounded by lakes, consistently ranked top places to live in the US
- **Highly ranked undergraduate and graduate ECE programs**
  - #9 (EE) and #9 (CE) graduate ranking among public universities – USNEWS’25


Department of Electrical
and Computer Engineering
UNIVERSITY OF WISCONSIN–MADISON
Solid-state Electronics & Quantum Technologies
Optics & Photonics
Applied Electromagnetics, Acoustics and Plasmas
Communications, Networks, Privacy & Security
Computer Systems & Architecture
Energy Systems
Optimization & Control
Artificial Intelligence
STRACHEN
C. GUPTA
KNEZEVIC
YU
CHOY
BOTEZ
ANDREWS
MAWST
WANG
JIANG
KATS
PASAYAT
Z. MA
TERVO
JACOBBERGER
FAWAZ
KRISHNASWAMY
OGRAS
VOLK
VAN DER WEIDE
BEHDAD
SUN
C. MA
HAGNESS
WENDT
LUDOIS
ROY
SARLIOGLU
VENKATARAMANAN
ZHOU
GROSS
ROALD
SINGH
MILENKOVIC
FREDETTE
M. GUPTA
LESIEUTRE
FLANNERY
CHRYSOS
ARVELO
PAPAILIOPOULOS
VINAYAK
NOWAK
SETHARES
LEE
COULSON
MORGADO
TZIMPRAGOS
YE
VELTEN
JAISWAL
DAVOODI
RAMANATHAN
SAN MIGUEL
HUANG
BEHROOZI
Instructional faculty with PhD - Jan. 28, 2025

Connect with UW-Madison ECE

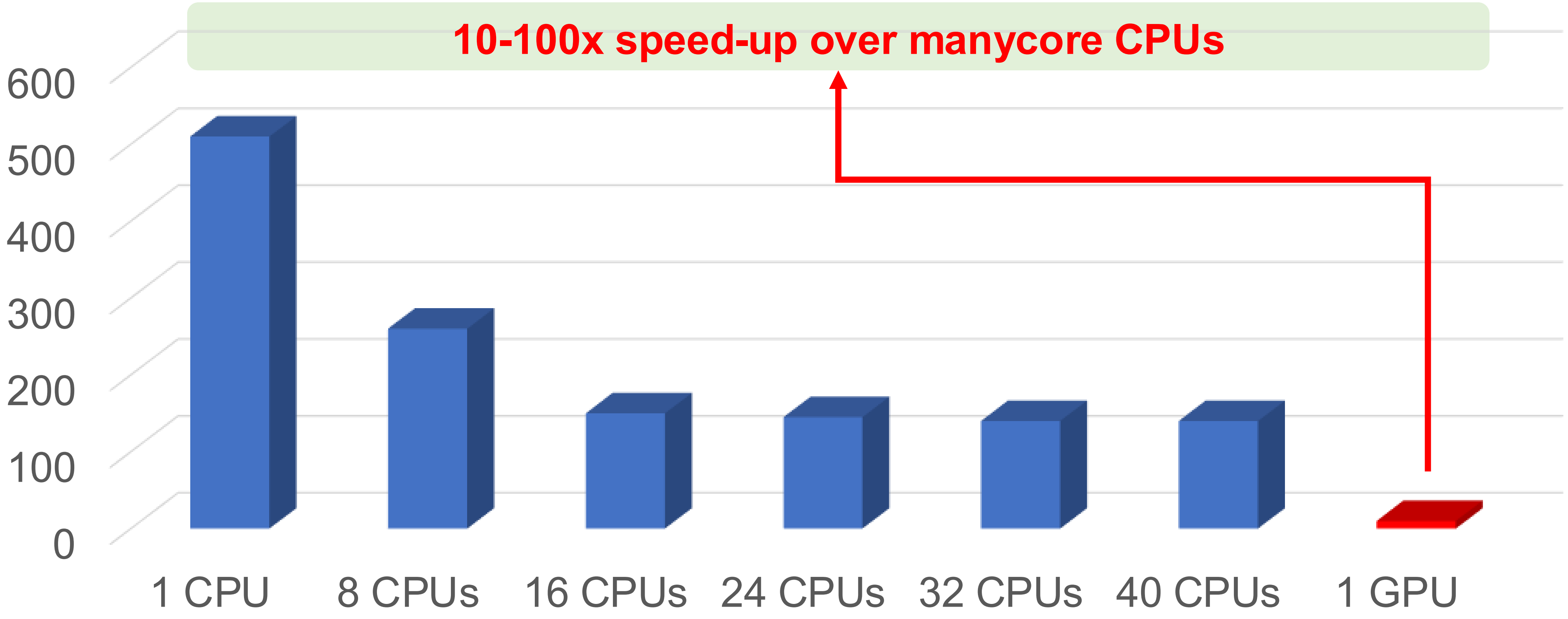
- **Host intern, co-op, and other job opportunities**
  - Engineering Career Services team helps match top talent with internship, co-op, and full-time opportunities
- **Sponsor research for badger engineers**
  - Philanthropic support
    - Student, faculty or research support
    - Brand reputation & talent pipeline
    - No indirect costs & Tax benefits
  - Sponsored research
    - Drives innovation in mutually beneficial areas
    - Access to broad expertise
    - IP rights & early tech access
- **Give guest lecturers to our classes**
  - Share new ideas of your companies with students

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?

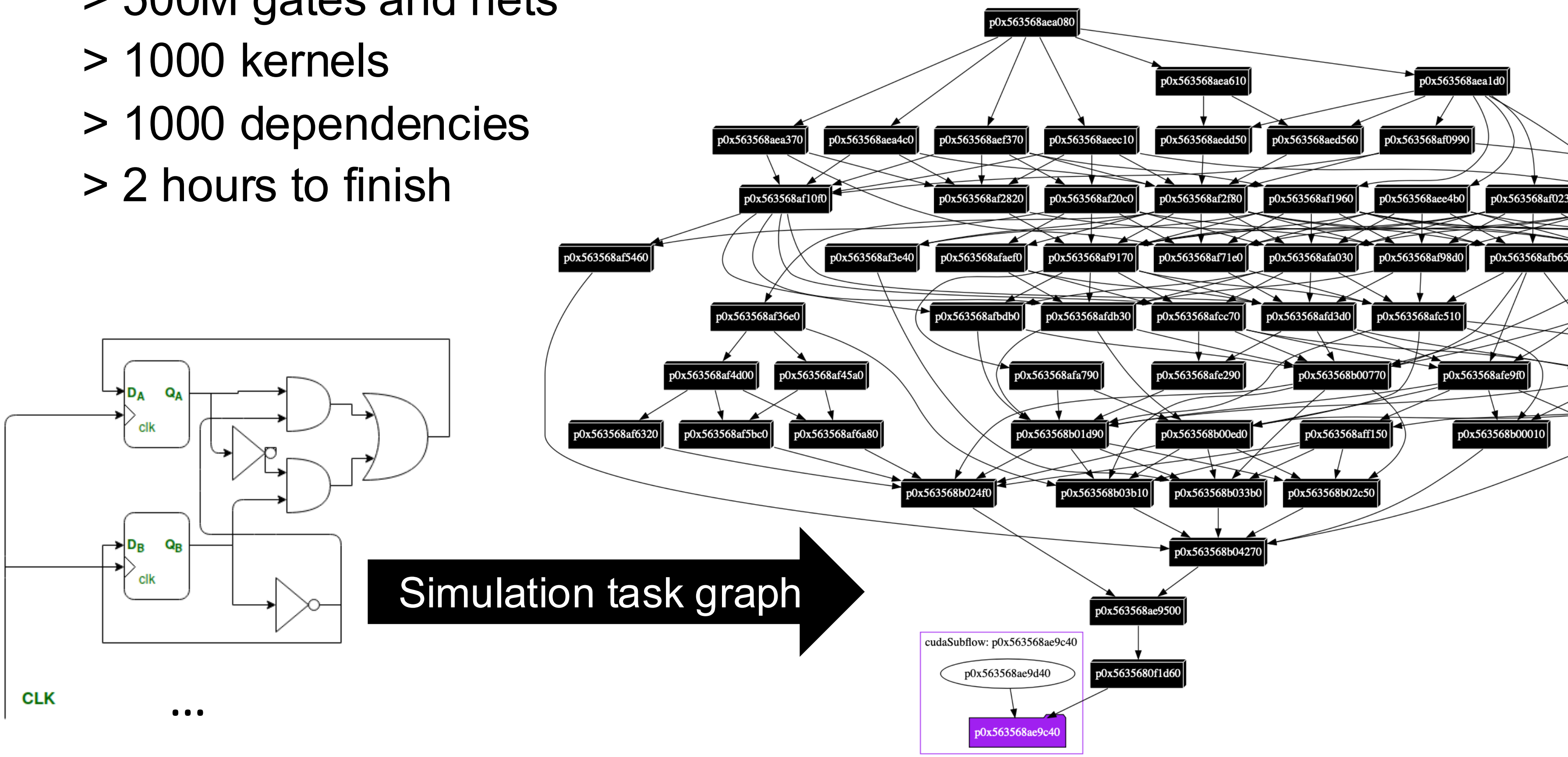
- **Advances performance to a new level previously out of reach**



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

Today's Parallel Workload is Very Complex

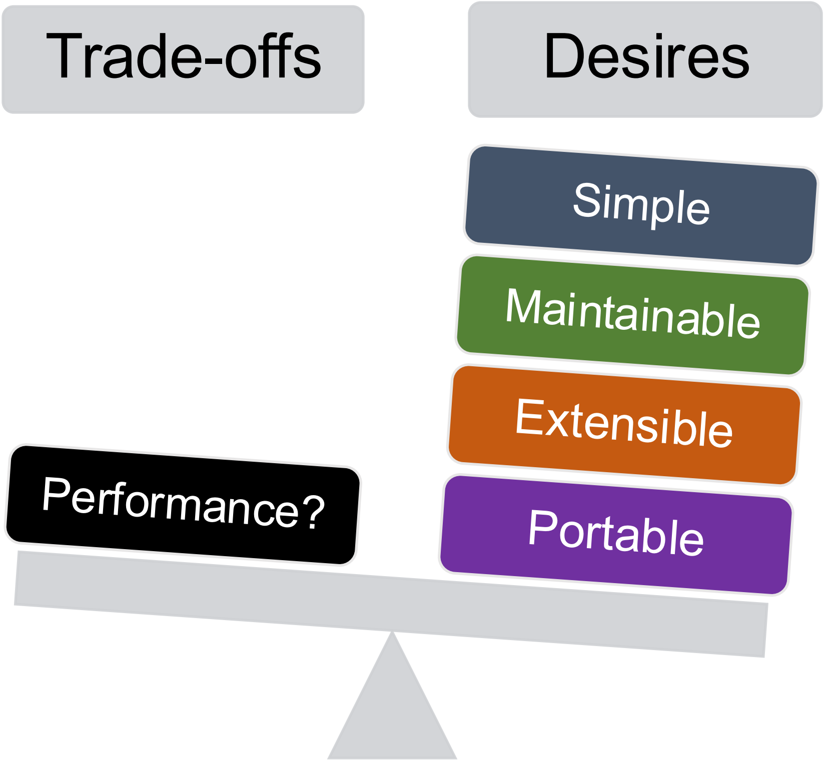
- **GPU-parallel circuit simulation task graph of Nvidia's NVDLA design[1]**
  - > 500M gates and nets
  - > 1000 kernels
  - > 1000 dependencies
  - > 2 hours to finish



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

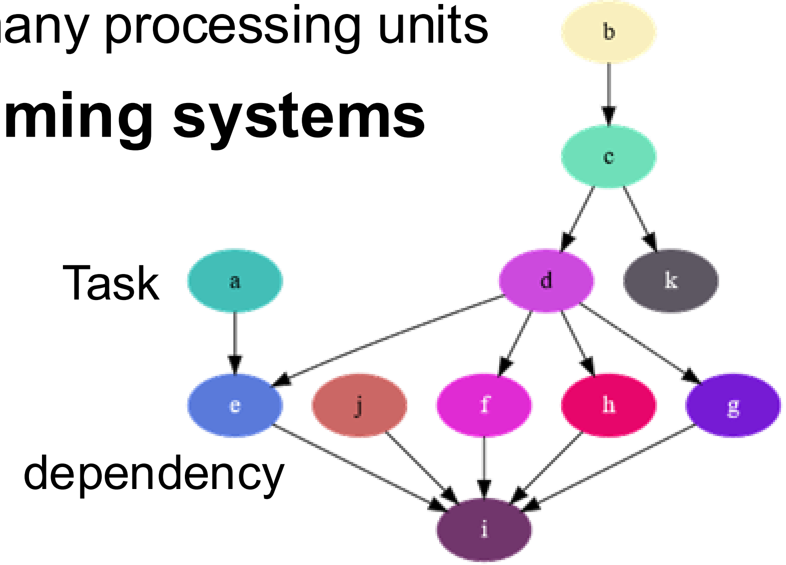
Parallel Programming is Not Easy

- **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



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 standards:**
  - ***Task graph parallelism*** (TGP) is the best model for future parallel architectures
    - Capture programmers' intention in decomposing a heterogeneous algorithm into a top-down task graph
    - Runtime can schedule dependent tasks across many processing units
- **Increasing numbers of task-parallel programming systems**



Two Problems of Existing Tools for EDA …

- **EDA has very complex task dependencies**
  - **Example**: analysis algorithms compute the circuit network of multi-millions of nodes and dependencies
  - **Problem**: existing tools are often good at loop parallelism (e.g., embarrassingly-parallel loops) but weak in expressing task graphs at this large scale
- **EDA has very complex control flow**
  - **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 limited to *direct acyclic graph* (DAC) models, requiring users to manually partition their workloads around control-flow or decision-making points

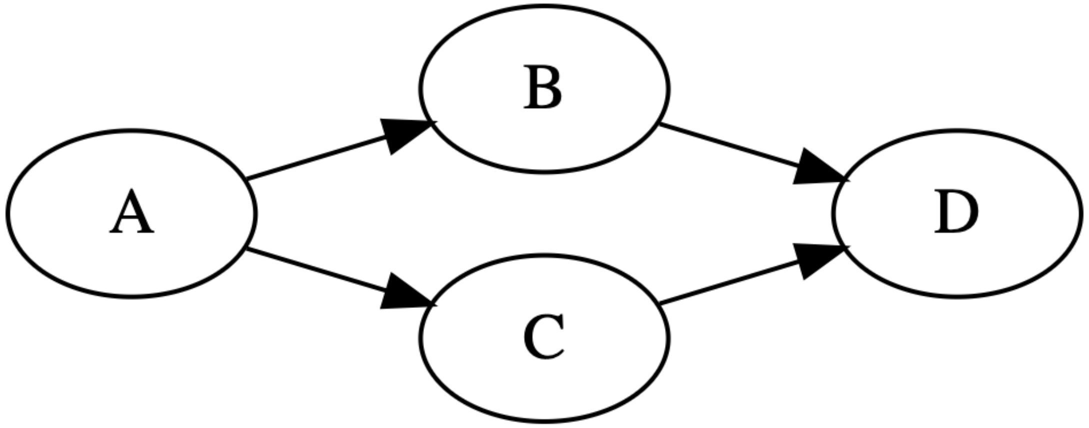
Takeaways

- 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*, 2022

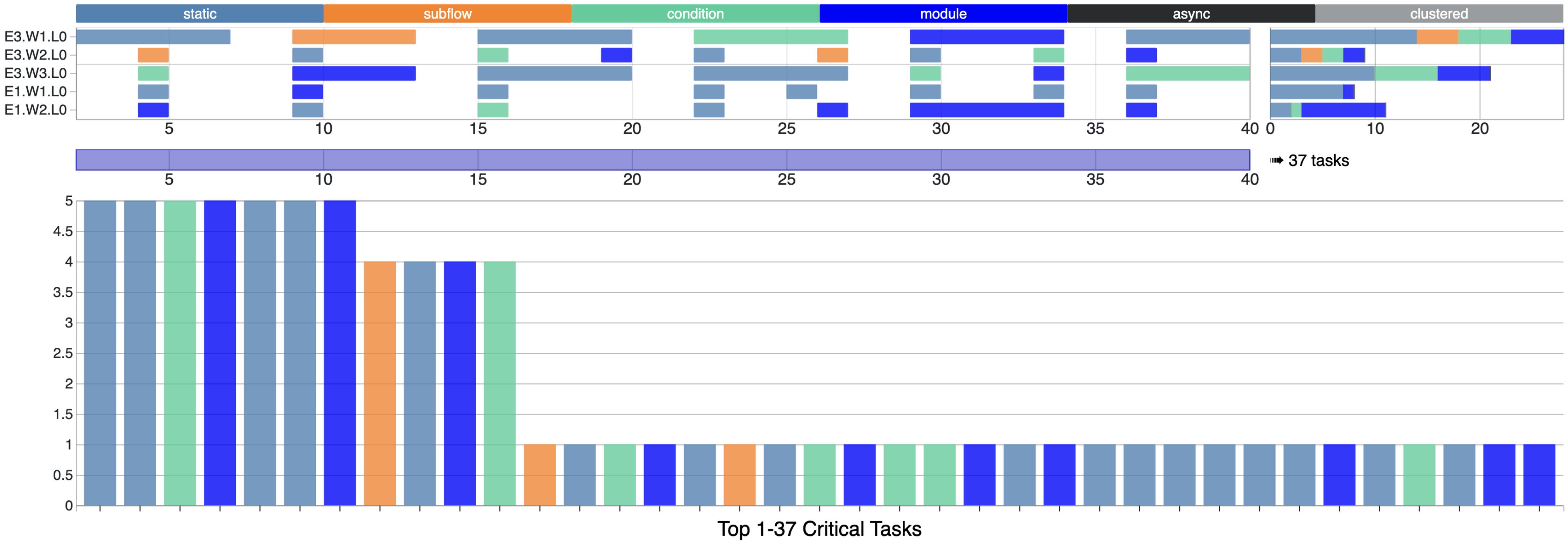
Taskflow Supports Drop-in Integration

- **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

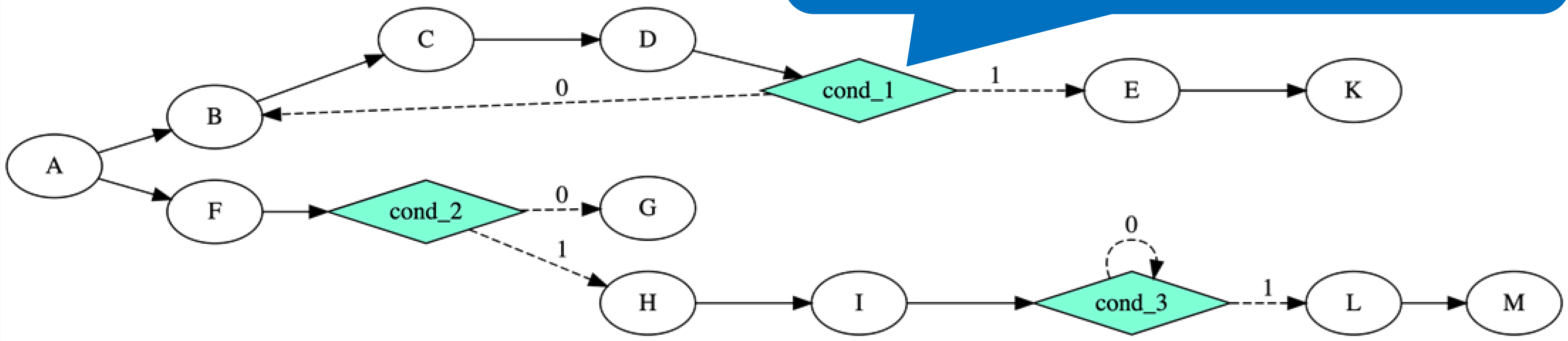
```
# run you program with the env variable TF_ENABLE_PROFILER enabled
# and paste the JSON content to https://taskflow.github.io/tfprof/
~$ TF_ENABLE_PROFILER=simple.json ./simple
```



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
```

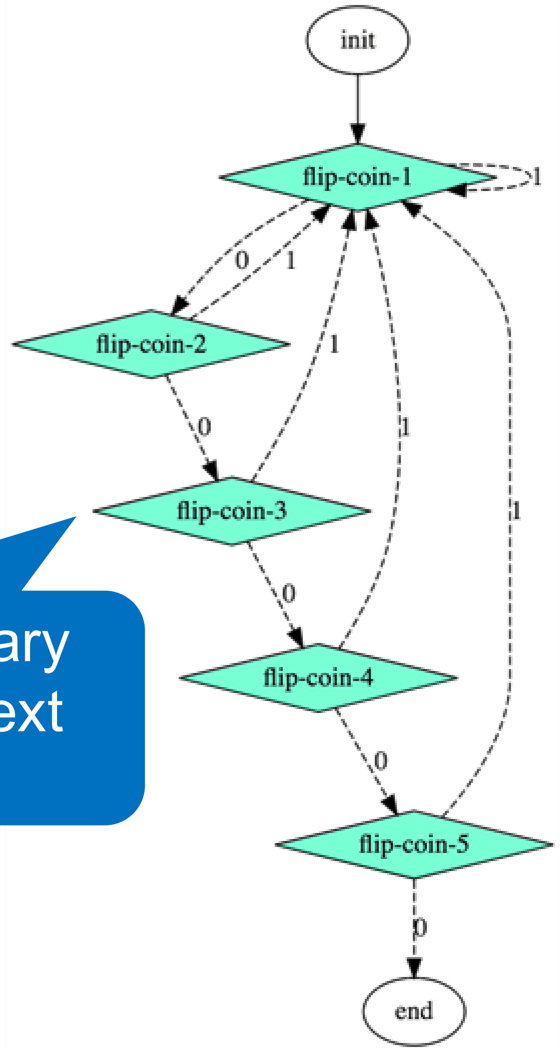
Very difficult for existing DAG-based systems to express an efficient overlap between tasks and control flow …



Non-deterministic Control Flow with CTFG

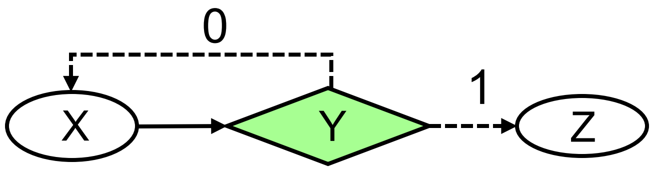
```
auto A = taskflow.emplace([&](){ } );
auto B = taskflow.emplace([&](){ return rand()%2; } );
auto C = taskflow.emplace([&](){ return rand()%2; } );
auto D = taskflow.emplace([&](){ return rand()%2; } );
auto E = taskflow.emplace([&](){ return rand()%2; } );
auto F = taskflow.emplace([&](){ return 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?**
  - Resort to client-side partitions of the task graph around each decision-making points
  - Synchronize the execution of partitioned task graphs around decision-making points
  - Lack end-to-end parallelism



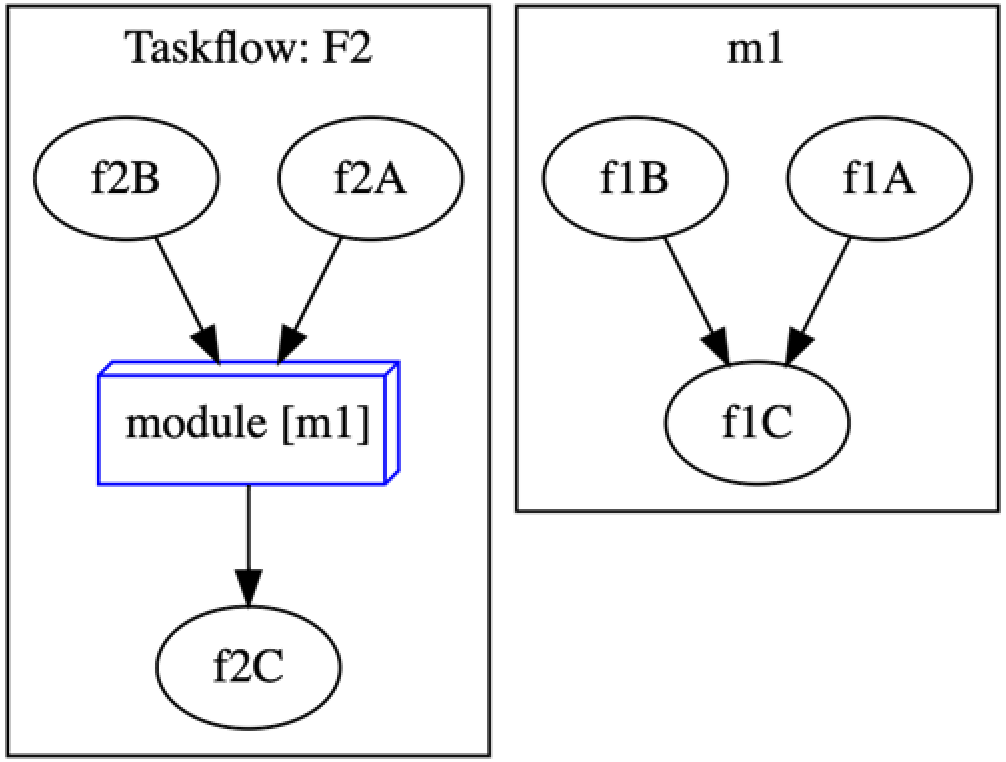

```
tf::Taskflow G;
auto X = G.emplace([](){});
auto Y = G.emplace([](){
  return converged() ? 1 : 0;
});
cond.precede(Z, X);
executor.run(G).wait();
```

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

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

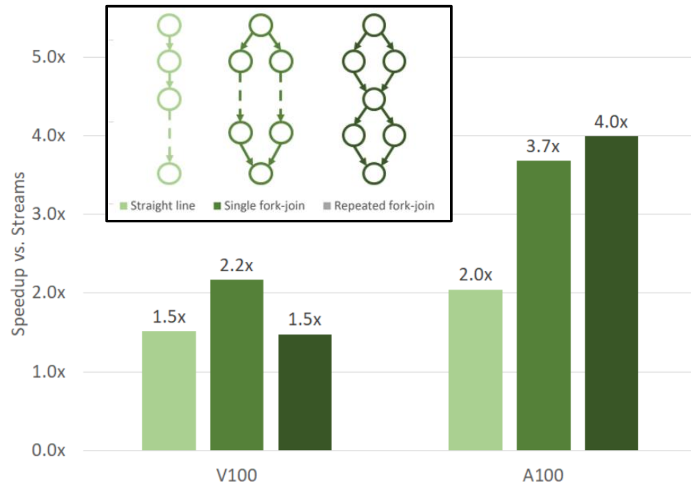
Composable Tasking

```
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);
```

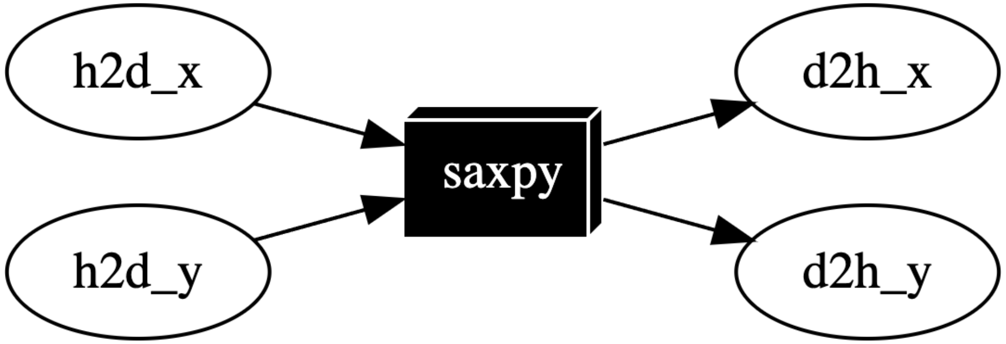


GPU Tasking with CUDA Graph[1]

```
tf::cudaGraph cg;
auto h2d_x = cg.copy(dx, hx.data(), N);
auto h2d_y = cg.copy(dy, hy.data(), N);
auto d2h_x = cg.copy(hx.data(), dx, N);
auto d2h_y = cg.copy(hy.data(), dy, N);

// saxpy kernel with 4 blocks each of 512 threads
auto kernel = cg.kernel(4, 512, 0, saxpy, N, 2f, dx, dy);
kernel.succeed(h2d_x, h2d_y).precede(d2h_x, d2h_y);

// create an executable and run it 10 times
tf::cudaGraphExec exec(cg);
tf::cudaStream stream;
stream.run_n(exec, 10).synchronize();
```



Advantage of CUDA Graph



[1]: Dian-Lun Lin and Tsung-Wei Huang, "Efficient GPU Computation using Task Graph Parallelism," *Euro-Par*, 2021

Everything is Composable in Taskflow

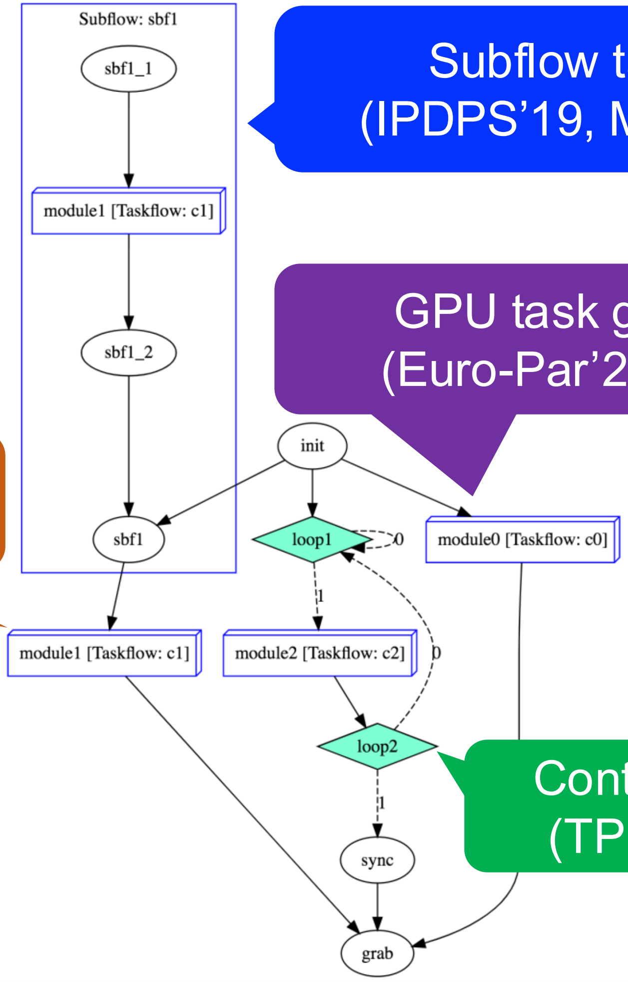
- **End-to-end parallelism in one graph**
  - Task, dependency, control flow all together
  - Scheduling with whole-graph optimization
  - Efficient overlap among heterogeneous tasks
- **Largely improved productivity!**

Subflow task (IPDPS’19, MM’19)

GPU task graph (Euro-Par’21,’24)

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

Control flow (TPDS’22)



Industrial use-case of productivity improvement using Taskflow[1]

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.

8 Reply Share Report Save Follow

ossia score

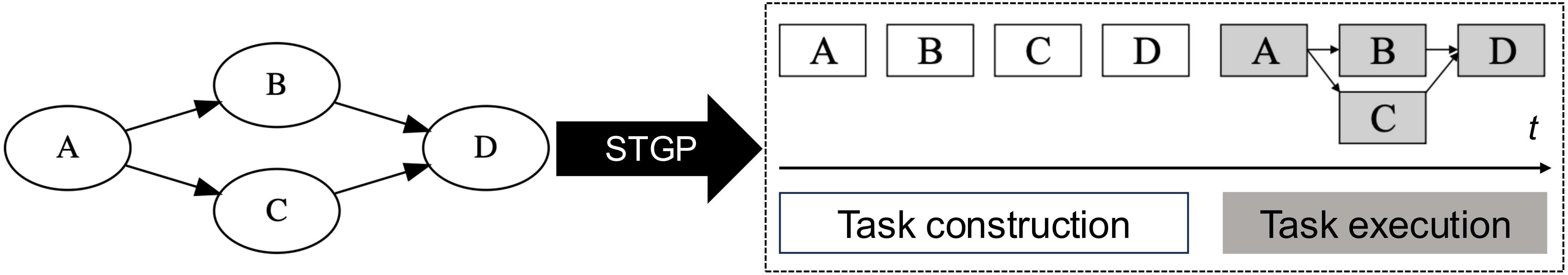
[1]: OSSIA’s experience with Taskflow: https://www.reddit.com/r/cpp/comments/g8ebnj/cpptaskflow_a_generalpurpose_parallel_and/

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

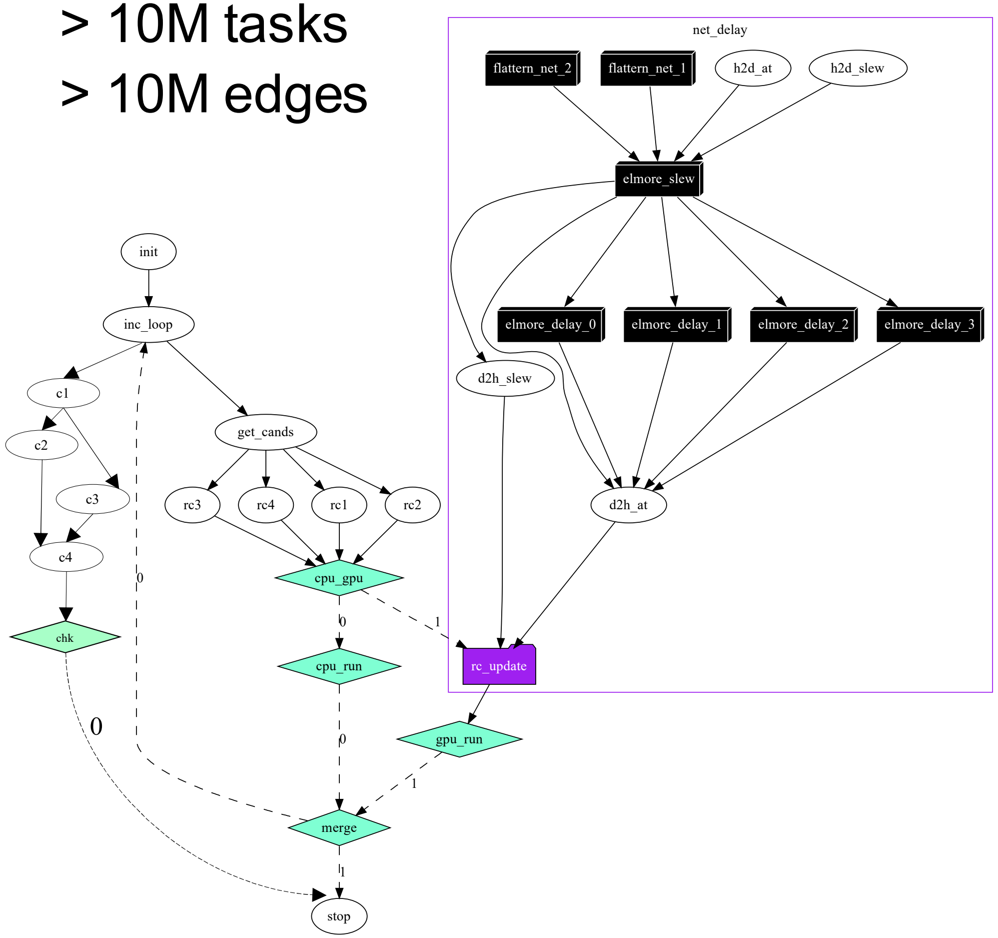
Static Task Graph Parallelism (STGP)

- **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

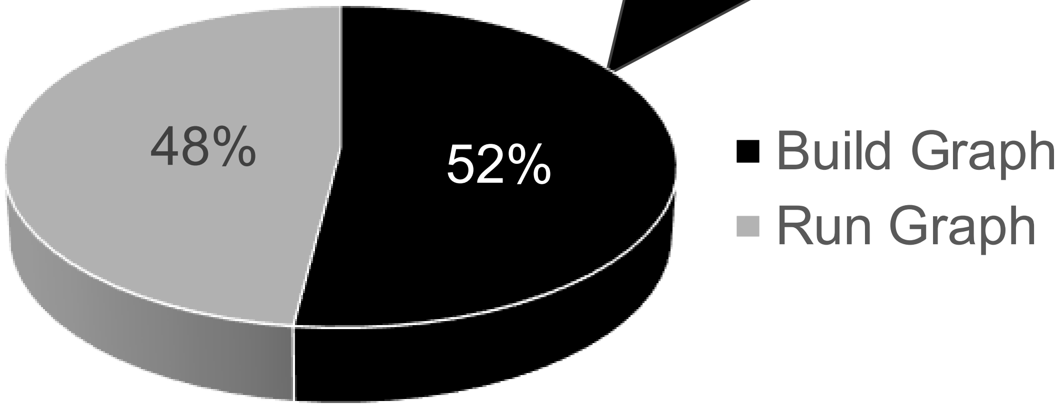


Problem of STGP: Example #1

- **Runtime breakdown of a task-parallel circuit timing analyzer**[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

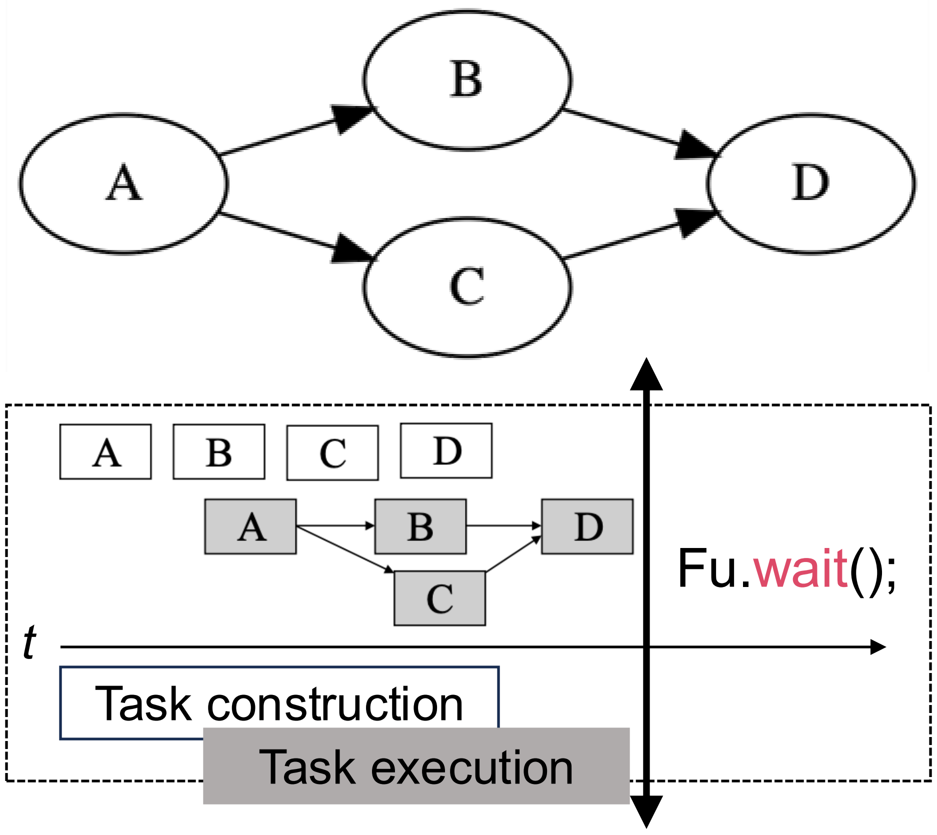
- **Express TGP that depends on runtime variables…?**

```
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
    }
}
```

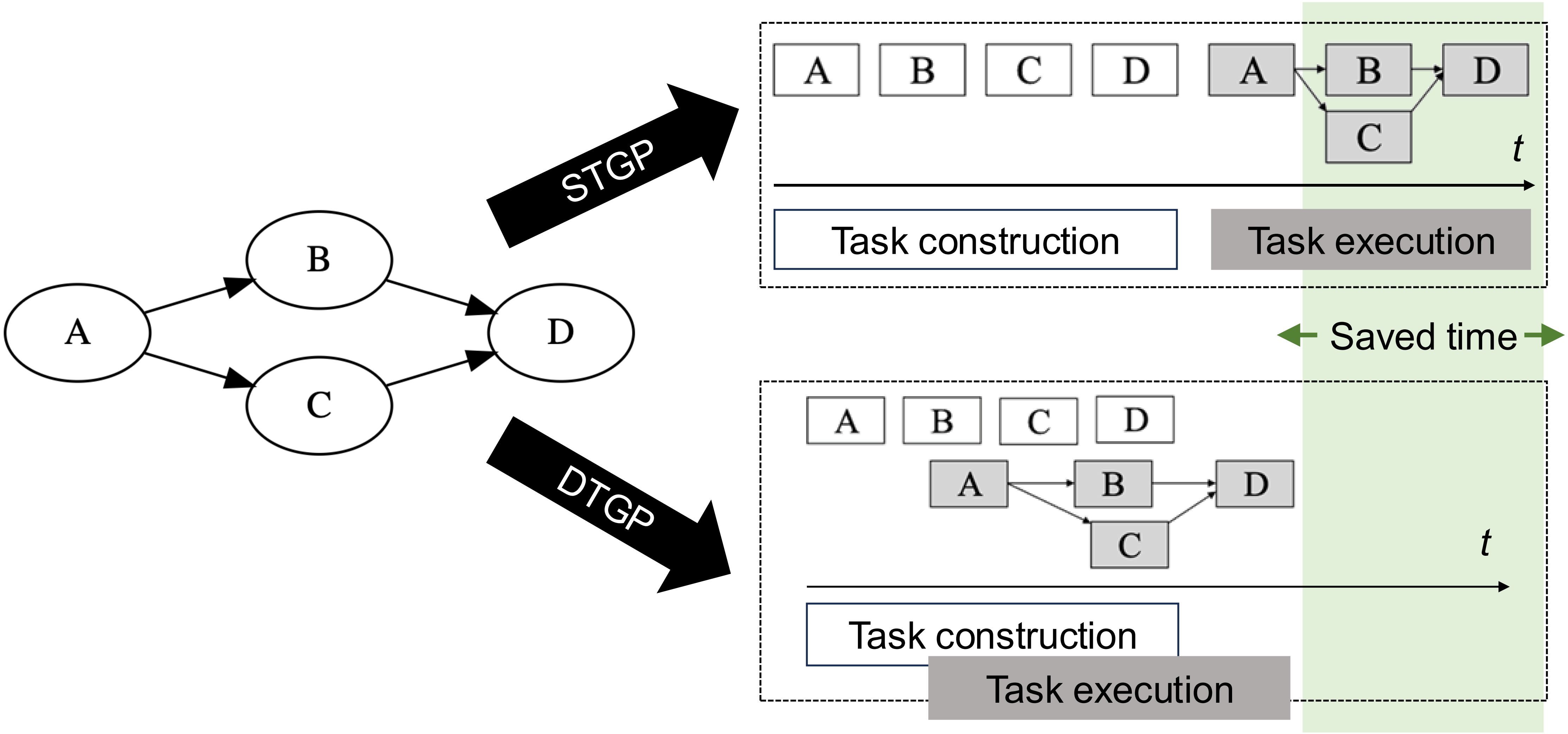
Dynamic TGP (DTGP) in Taskflow

```
// 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

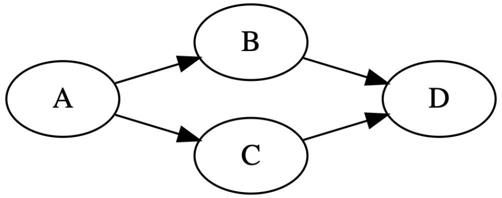
Comparison between STGP and DTGP

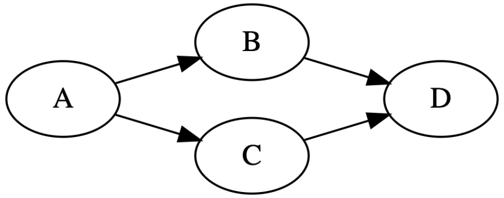


DTGP Needs a Correct Topological Order

```
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



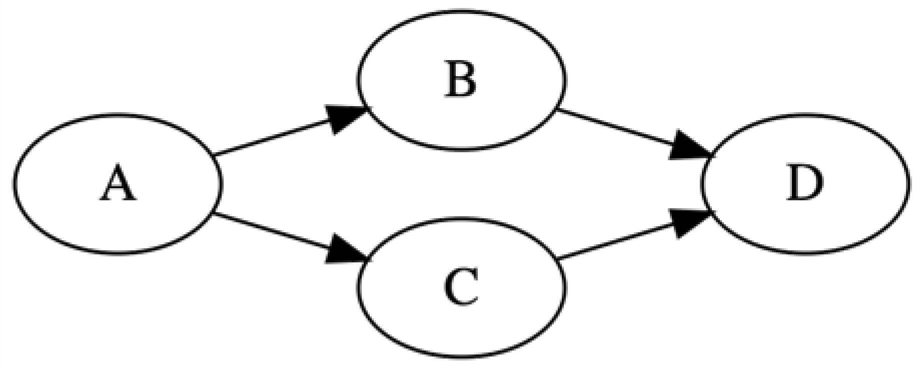


Topological order #2: A→C→B→D

```
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 can take a range of dependent-async tasks**
  - useful when the task dependencies come as a runtime variable

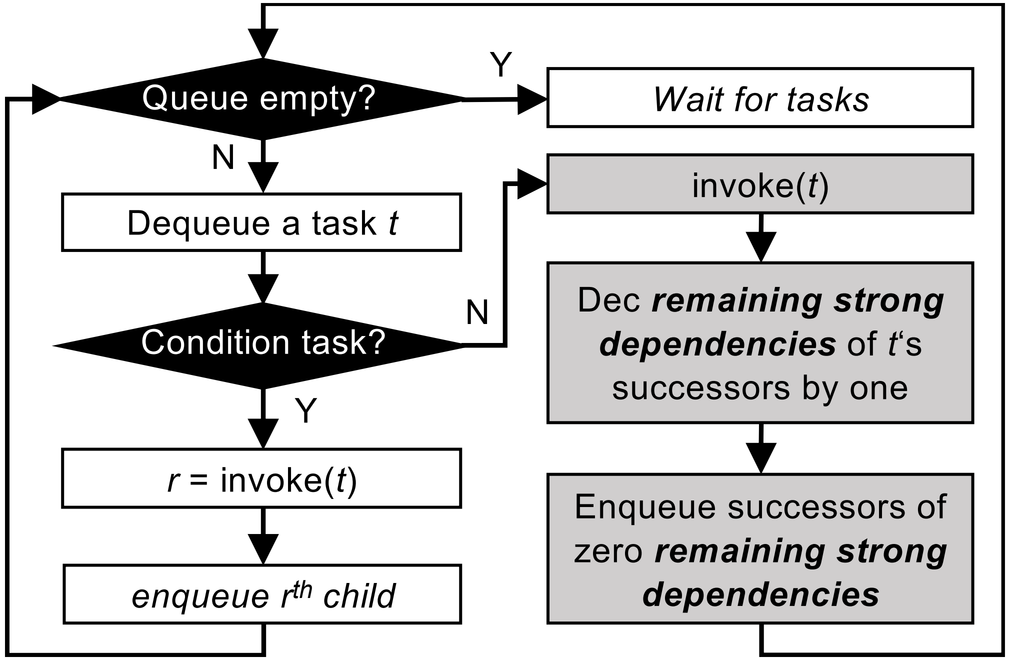
```
// Live: https://godbolt.org/z/6Pvco4KeE
std::vector<tf::AsyncTask> tasks = {
  executor.silent_dependent_async([](){ std::cout << "TaskA\n"; }),
  executor.silent_dependent_async([](){ std::cout << "TaskB\n"; }),
  executor.silent_dependent_async([](){ std::cout << "TaskC\n"; }),
  executor.silent_dependent_async([](){ 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

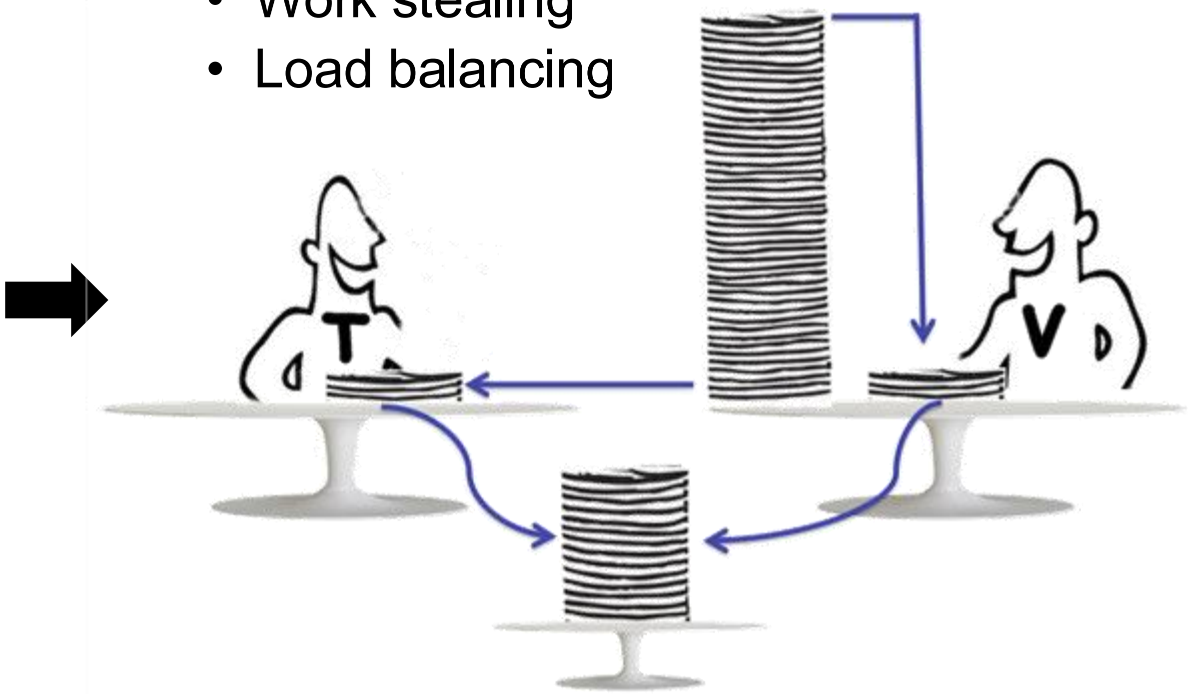
- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

STGP Scheduling Algorithm

- **Task-level scheduling**



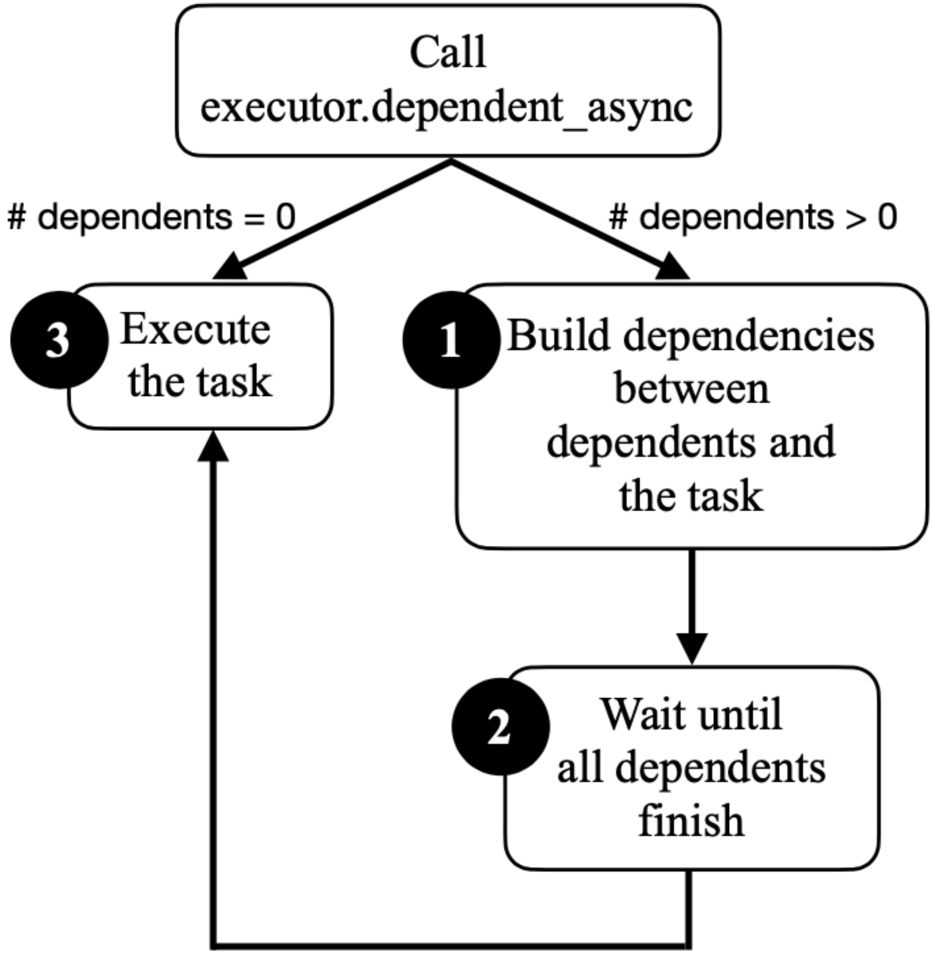
- **Worker-level scheduling**
  - Work stealing
  - Load balancing



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*

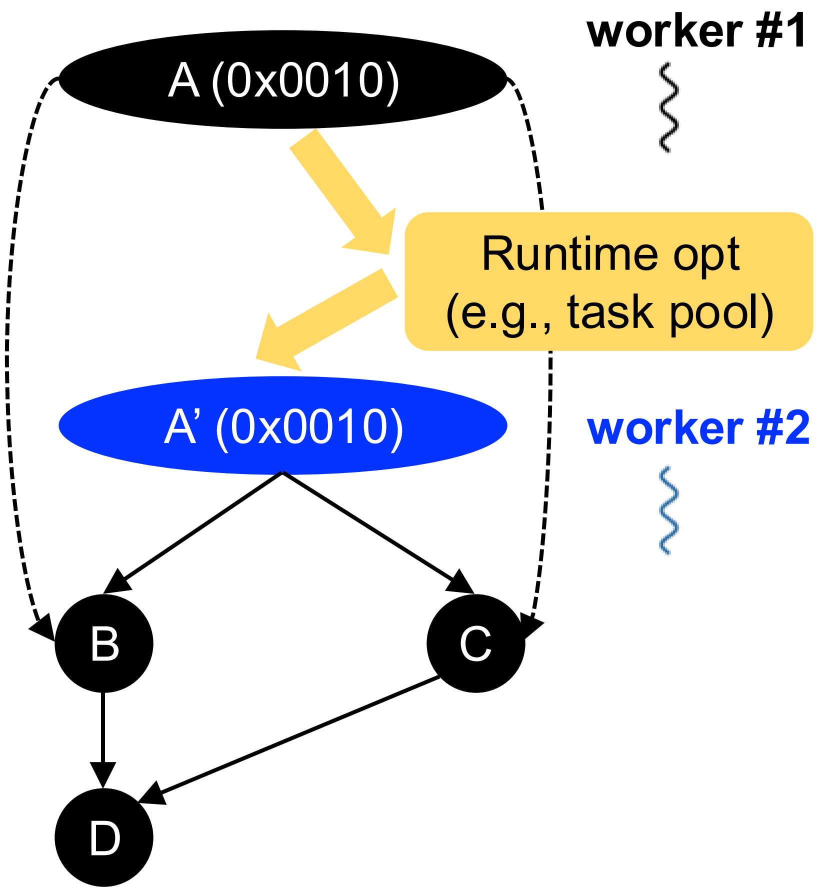
DTGP Scheduling Algorithm

- **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

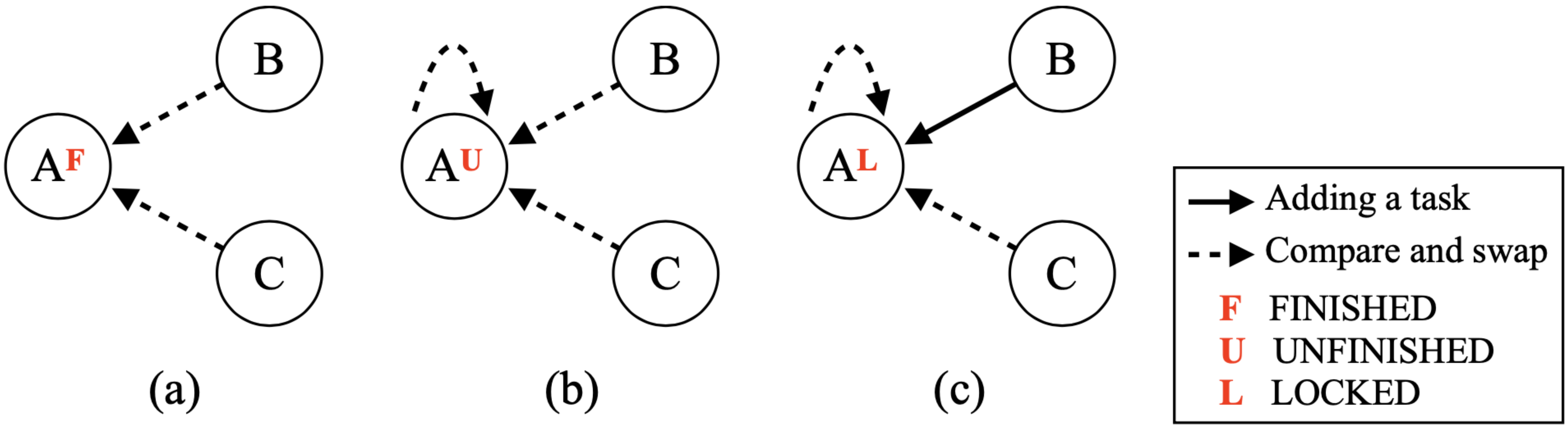
Retain Shared Ownership of Every Task

```
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**
  - Ex: executor.wait_for_all(), future.get(), etc.

```
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;
});
```

C++17

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

C++20

Taskflow uses C++20 atomic variables to perform waiting/notifying operations; many synchronizations can happen at user space instead of kernel

[1]: C++20 atomic wait/notify: https://en.cppreference.com/w/cpp/atomic/atomic/wait

Lock-free Scheduling Algorithm[1]

Algorithm 1 dependent_async(callable, deps)

```
Create a future
num_deps ← sizeof(deps)
task ← initialize_task(callable, num_deps, future)
for all dep ∈ 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 ← dep.state
2: target_state ← UNFINISHED
3: if dep_state.CAS(target_state, LOCKED) then
4:     dep.successors.push(task)
5:     dep_state ← UNFINISHED
6: else if target_state == FINISHED then
7:     num_deps ← AtomDec(task.join_counter)
8: else
9:     goto line 2
10: end if
```

Algorithm 3 schedule_async_task(task)

```
target_state ← UNFINISHED
while not task.state.CAS(target_state, FINISHED) do
    target_state ← UNFINISHED
end while
Invoke(task.callable)
for all successor ∈ 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
```

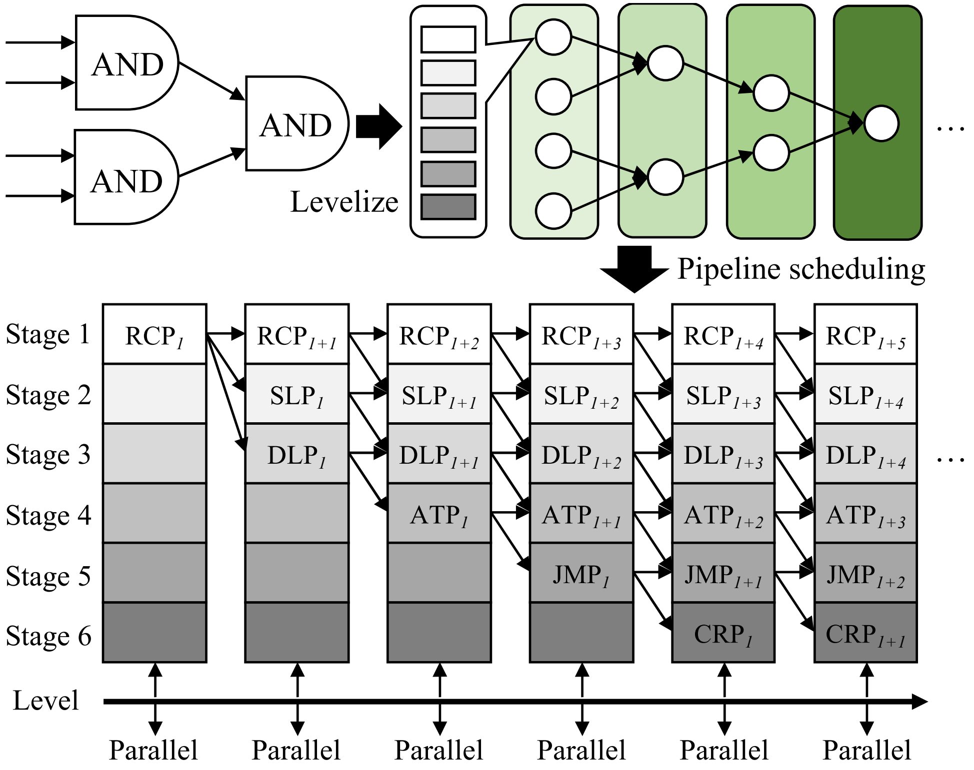

[1]: Cheng-Hsiang Chiu, et. al, "Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms," *IEEE/ACM ICCAD*, 2023

Takeaways

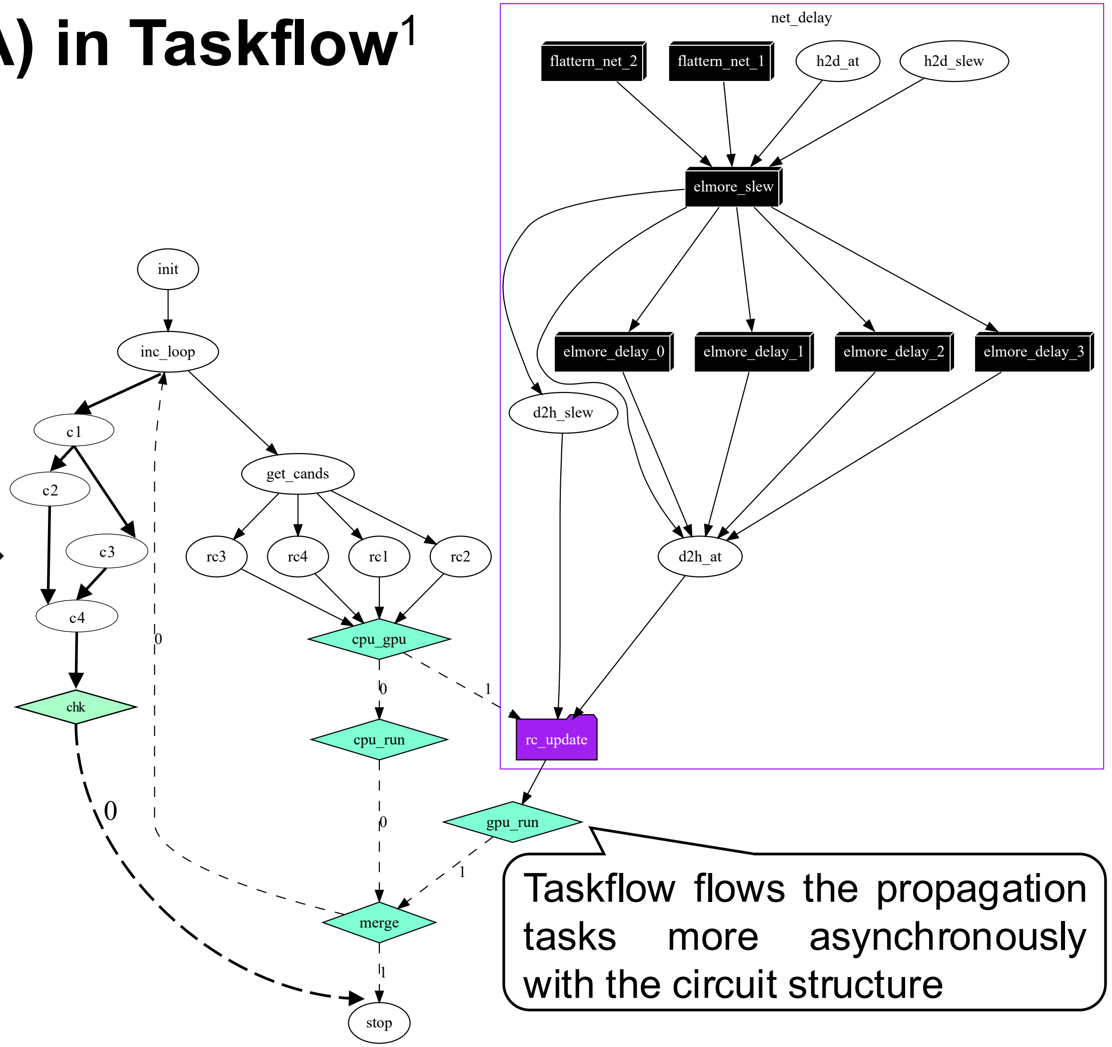
- Express your parallelism in the right way
- Program static task graph parallelism using Taskflow
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk

Case Study 1: Task-parallel STA

- **Model graph-based analysis (GBA) in Taskflow[1]**



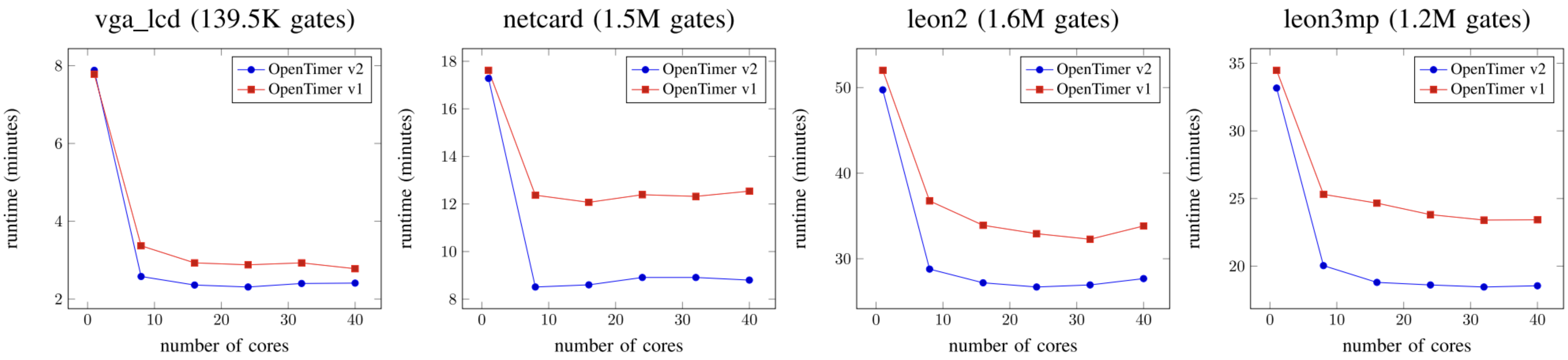
Loop-parallel level-by-level timing propagation in GBA



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

Levelization-based vs Task-parallel GBA

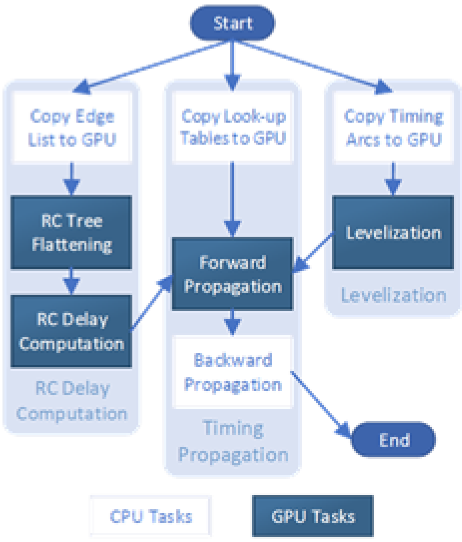
- **OpenTimer v1: levelization-based parallel timing propagation[1]**
  - Implemented using OpenMP “parallel_for” primitive
- **OpenTimer v2: task-parallel timing propagation[2]**
  - Implemented using Taskflow (https://taskflow.github.io/)



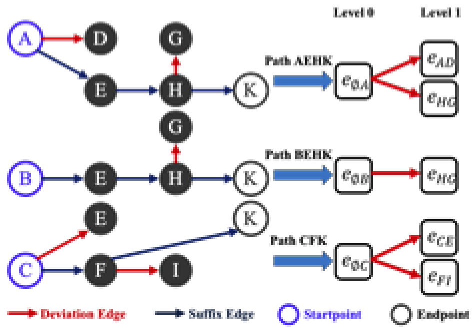
💡Taskflow allows us to more asynchronously parallelize the timing propagation

[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

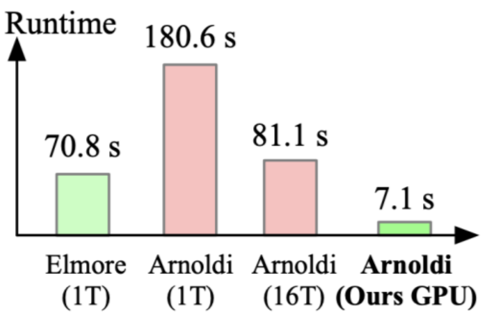
Our Research atop Task-parallel STA



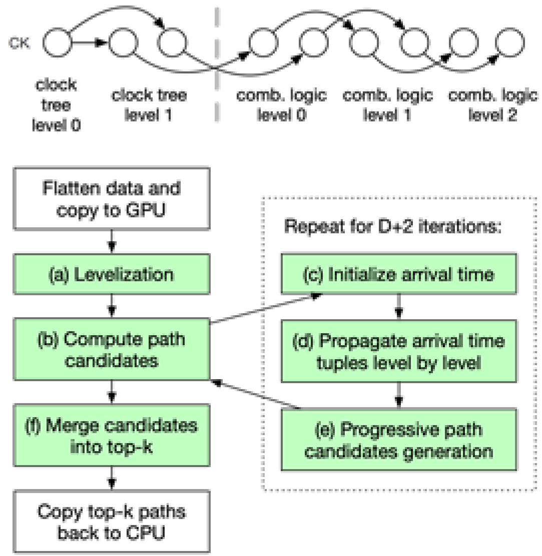
GPU-accelerated GBA (ICCAD'20, DAC'24)



GPU-accelerated PBA (DAC'21, TCAD'23)



GPU-accelerated CCS (DATE'24, DAC'24)



GPU-accelerated CPPR (ICCAD'21, TCAD'22)

Taskflow-powered STA engine[1]

4x faster

25–45x faster

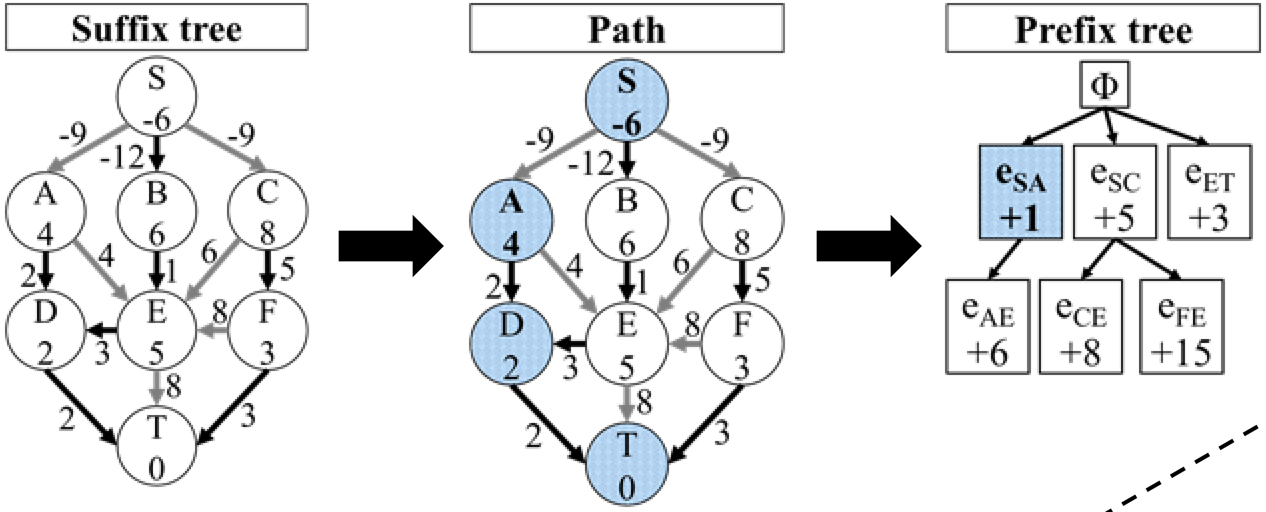
7–14x faster

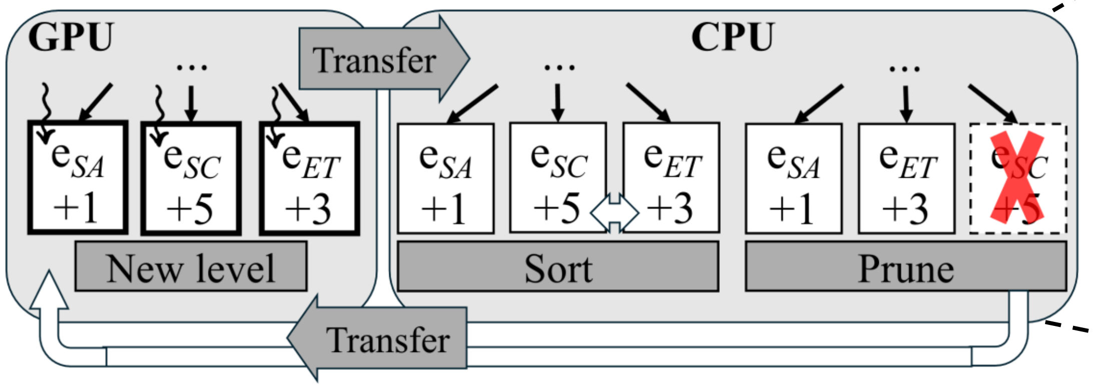
16x faster

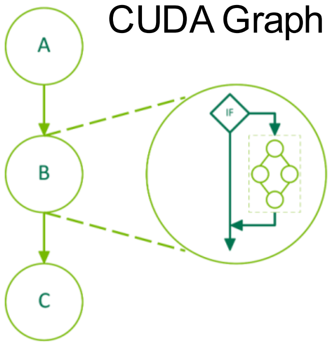
[1]: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2022

GPU-accelerated Path-based Analysis (PBA)

- **A GPU-parallel path generation algorithm[1]**







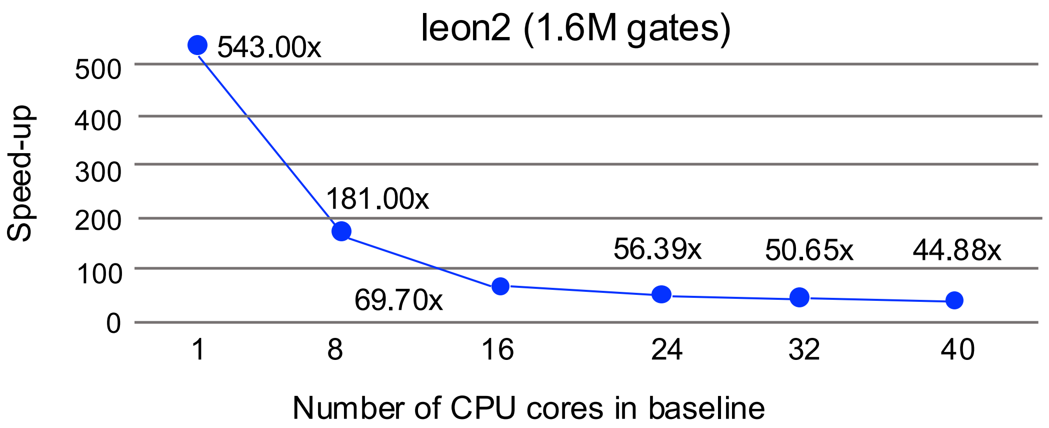
Input : G^+ in CSR format, N as #vertices, M as #edges, vertices$[N]$, edges$[M]$, weights$[M]$
Input : Shortest path forest, forest$[N]$ as edge array, distances$[N]$ as distance array
Input : currLevel as current level
Input : levelSize as the number of entries in current level
Result: Explore critical path candidates in next level

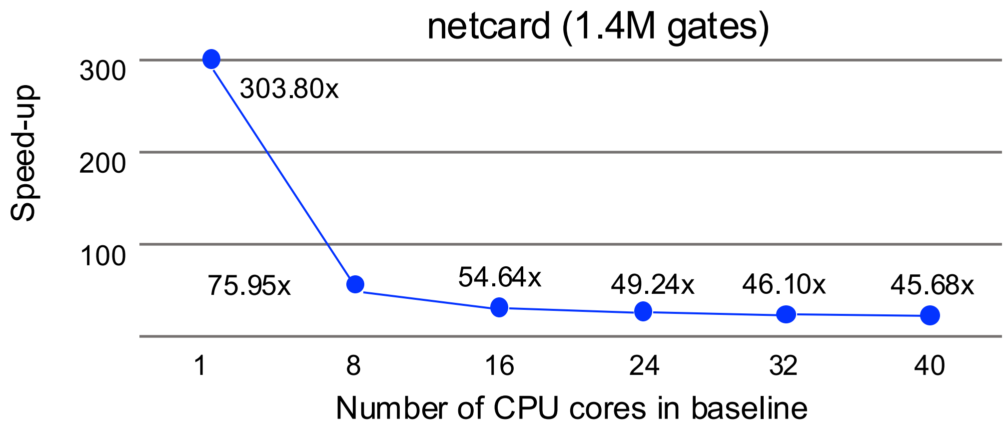
```
tid ← blockIdx.x ∗ blockDim.x + threadIdx.x;
if tid ≥ levelSize then
    return;
end
offset ← (tid == 0) ? 0 : currLevel[tid-1].childOffset;
level ← currLevel[tid].level;
slack ← currLevel[tid].slack;
v ← currLevel[tid].to;
while v is not endpoint do
    edgeStart ← vertices[v];
    edgeEnd ← (v == N-1) ? M : vertices[v+1];
    for eid ← edgeStart to edgeEnd do
        neighbor ← edges[eid];
        weight ← weights[eid];
        if eid is deviation edge then
            /* Fill out child path data */
            newPath ← nextLevel[offset];
            newPath.level ← level+1;
            newPath.from ← v;
            newPath.to ← neighbor;
            newPath.parent ← tid;
            newPath.childOffset ← 0;
            newPath.slack ← slack + distances[neighbor] +
              weight - distances[v];
            offset ← offset + 1;
        end
    end
    /* Traverse along the shortest path forest */
    v = forest[v];
end
return;
```


[1]: G. Guo, et al, "A GPU-Accelerated Framework for Path-Based Timing Analysis," *IEEE TCAD*, 2023

Performance of GPU-based Path Generation

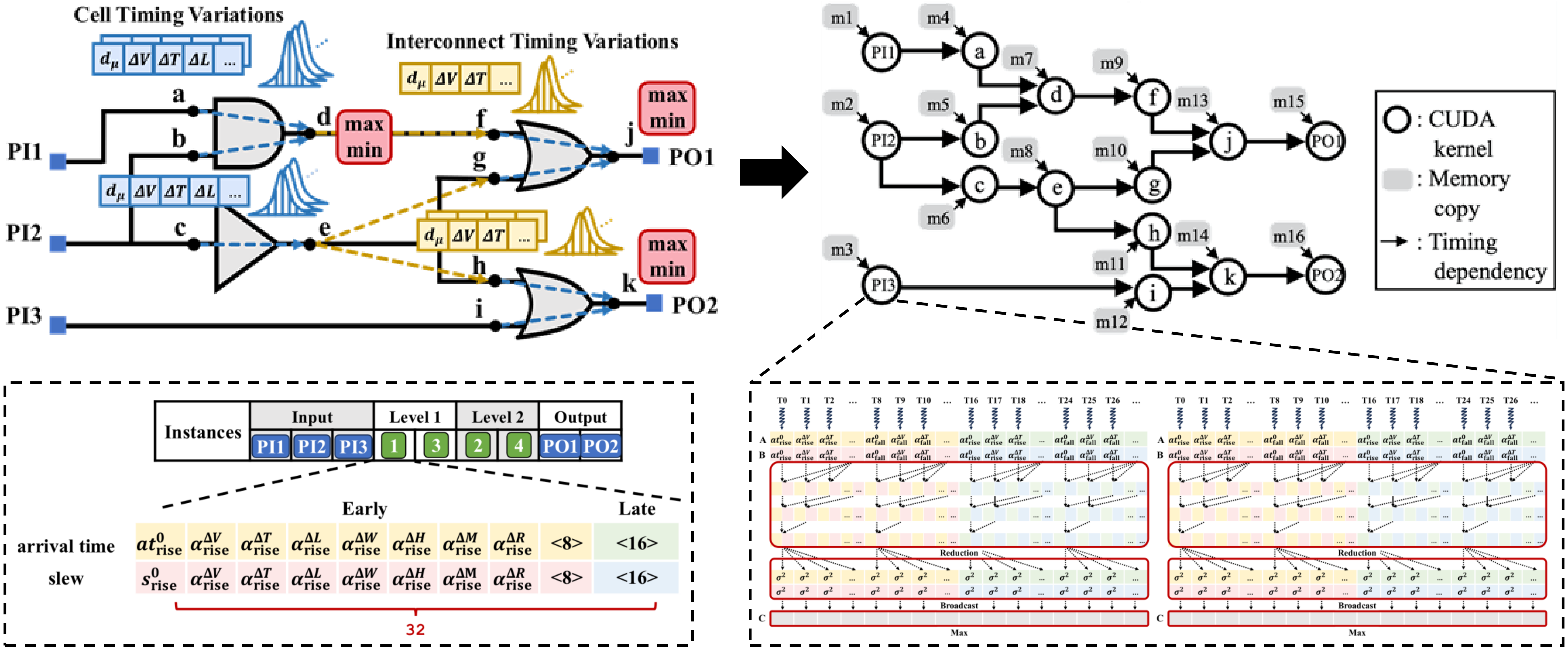
Benchmark	#Pins	#Gates	#Arcs	OpenTimer Runtime	Our Algorithm #MDL=10		Our Algorithm #MDL=15		Our Algorithm #MDL=20	
					Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	7984262	2875783	4708.36	611×	5295.49ms	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×	7091.79ms	172×	8182.84	149×
netcard	3999174	1496719	7404006	752188	2050.60	367×	2475.90ms	304×	2484.08	303×
vga_lcd	397809	139529	756631	53204	682.94	77.9×	683.04ms	77.9×	706.16	75.3×
vga_lcd_iccad	679258	259067	1243041	66582	720.40	92.4×	754.35ms	88.3×	766.29	86.9×
b19_iccad	782914	255278	1576198	402645	2144.67	188×	2948.94ms	137×	3483.05	116×
des_perf_ispd	371587	138878	697145	24120	763.79	31.6×	766.31ms	31.5×	780.56	30.9×
edit_dist_ispd	416609	147650	799167	614043	1818.49	338×	2475.12ms	248×	2900.14	212×
mgc_edit_dist	450354	161692	852615	694014	1463.61	474×	1485.65ms	467×	1493.90	465×
mgc_matric_mult	492568	171282	948154	214980	994.67	216×	1075.90ms	200×	1113.26	193×





Case Study 2: GPU-accelerated SSTA

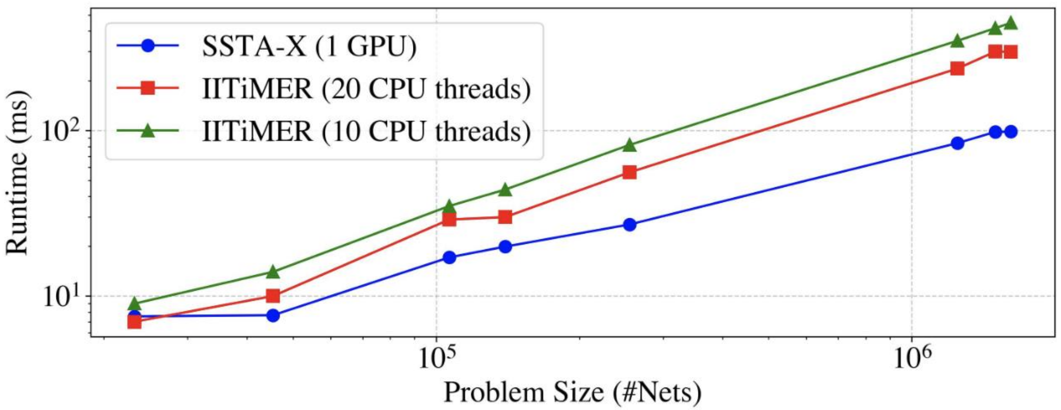
- **Model the SSTA propagation workload into a GPU task graph**

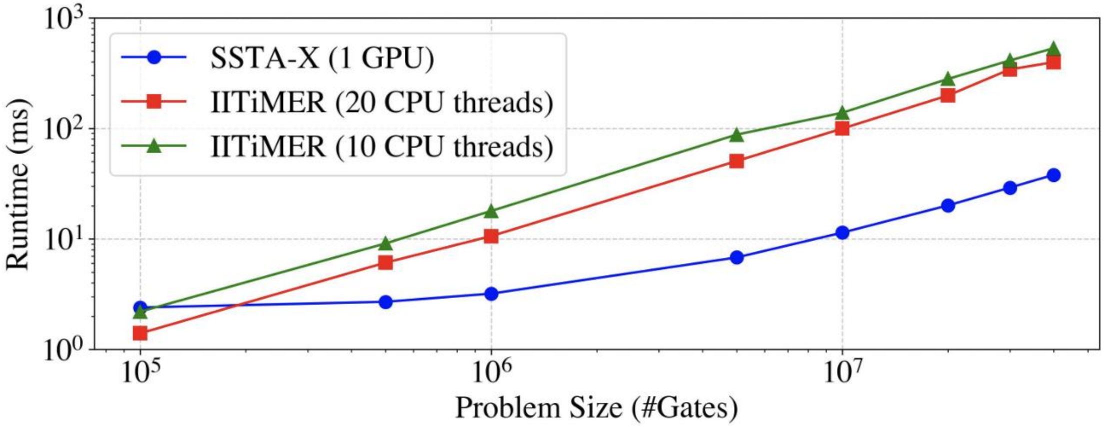


[1]: Chih-Chun Chang, et. al, "SSTA-X: GPU-Accelerated First-Order Block-Based Statistical Static Timing Analysis," *under submission*

GPU-accelerated STA vs CPU-parallel SSTA[1]

Circuit	#Gates	#Nets	#Pins	Runtime (ms)				Speedup of SSTA-X over			Error Rate vs. MC-SSTA
				MC-SSTA 20 CPUs	IITiMER 1 CPUs	IITiMER 20 CPUs	**SSTA-X 1 GPU**	MC-SSTA 20 CPUs	IITiMER 1 CPUs	IITiMER 20 CPUs	μ and σ^2 (%)
aes_core	22938	23199	66751	3483	129	26	**20**	174.15×	6.45×	1.30×	0.29
b19	255278	255300	782914	54989	1411	335	**134**	410.36×	10.50×	2.50×	0.38
cordic	45359	45393	127993	7286	226	70	**30**	242.86×	7.53×	2.33×	0.07
des_perf	138878	139112	371587	11403	580	115	**52**	219.28×	11.15×	2.21×	0.36
leon2	1616369	1616984	4328255	178620	6912	1667	**554**	322.41×	12.47×	3.01×	0.32
leon3mp	1247725	1247979	3376832	167752	5455	1487	**476**	352.42×	11.46×	3.10×	0.40
netcard	1496719	1498555	3999174	209258	6668	1799	**590**	354.67×	11.30×	3.04×	0.34
vga_lcd	139529	139635	397809	20233	674	157	**72**	281.01×	9.36×	2.18×	2.11
vga_lcd_iccad	259067	259152	679258	27455	1068	295	**103**	266.55×	10.36×	2.86×	0.41
Average								327.96×	11.32×	2.81×	0.58

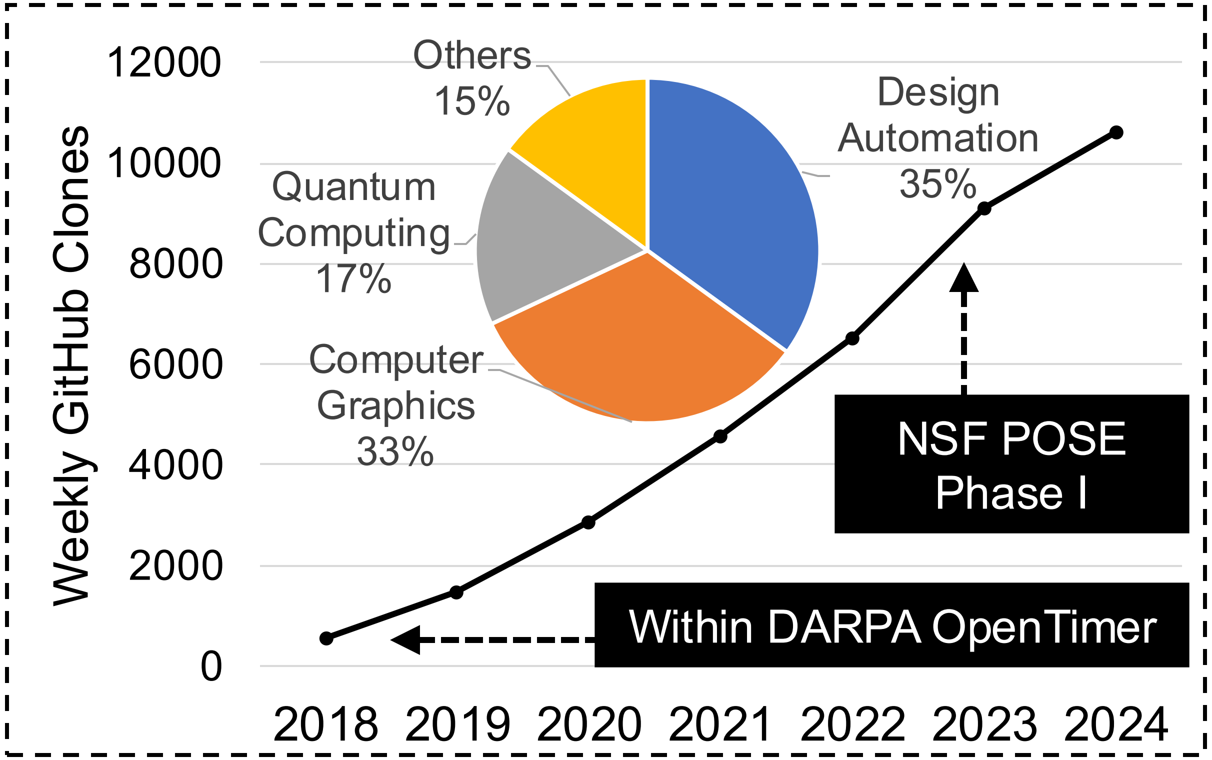




[1]: Chih-Chun Chang, et. al, "SSTA-X: GPU-Accelerated First-Order Block-Based Statistical Static Timing Analysis," *under submission*

Other Industrial Applications of Taskflow

- **Quantum computing**
  - Xanadu used Taskflow in their state vector-based simulator, JET
- **Computer graphics and game rendering**
  - Vulkan recommends Taskflow for parallelizing your rendering engines
- **FPGA physical design**
  - Vivado uses Taskflow for synthesis
- **Embedded/edge computing**
  - Tesseract (robotics planning)
  - Cruise (autonomous car)
  - Reveal.Tech (drone vision)
  - Tesseract Robotic (planning tool)
  - …
- **C++26 std::exec (coming soon)**
  - Benefit millions of C++ developers ☺



[1]: Nvidia's implementation of C++26 std::exec: https://github.com/NVIDIA/stdexec/tree/main/include/execpools/taskflow

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!

…

Thank you for Sponsoring Taskflow!

NSF

DARPA

NUMFOCUS
OPEN CODE = BETTER SCIENCE

intel

Google Summer of Code

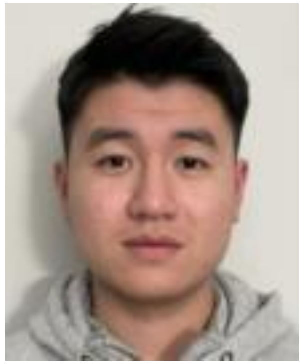
NVIDIA

Acknowledgment: Excellent PhD Students

- **Please contact me if you have any intern/full-time opportunities!**
  - We specialize in CAD, HPC, and GPU 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();
```