



# 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/>





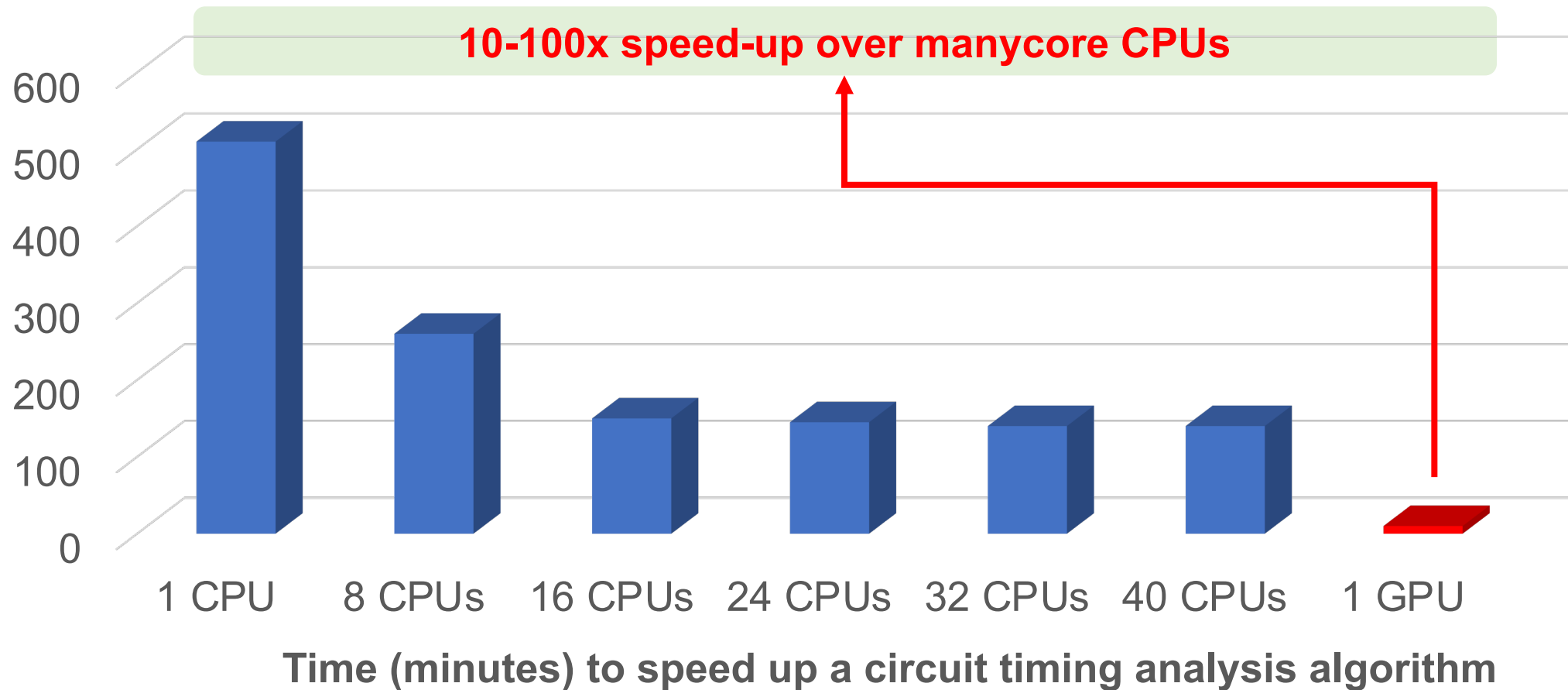
# Takeaways

---

- **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?

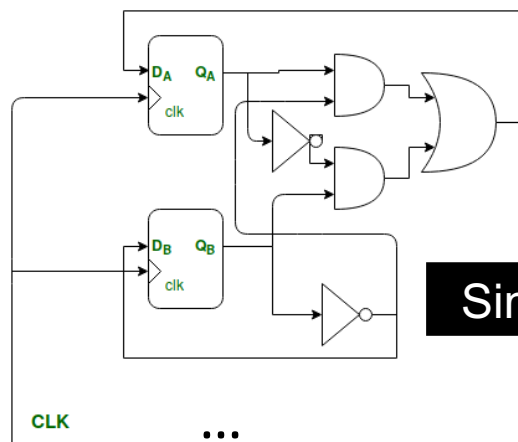
- Advances performance to a new level previously out of reach



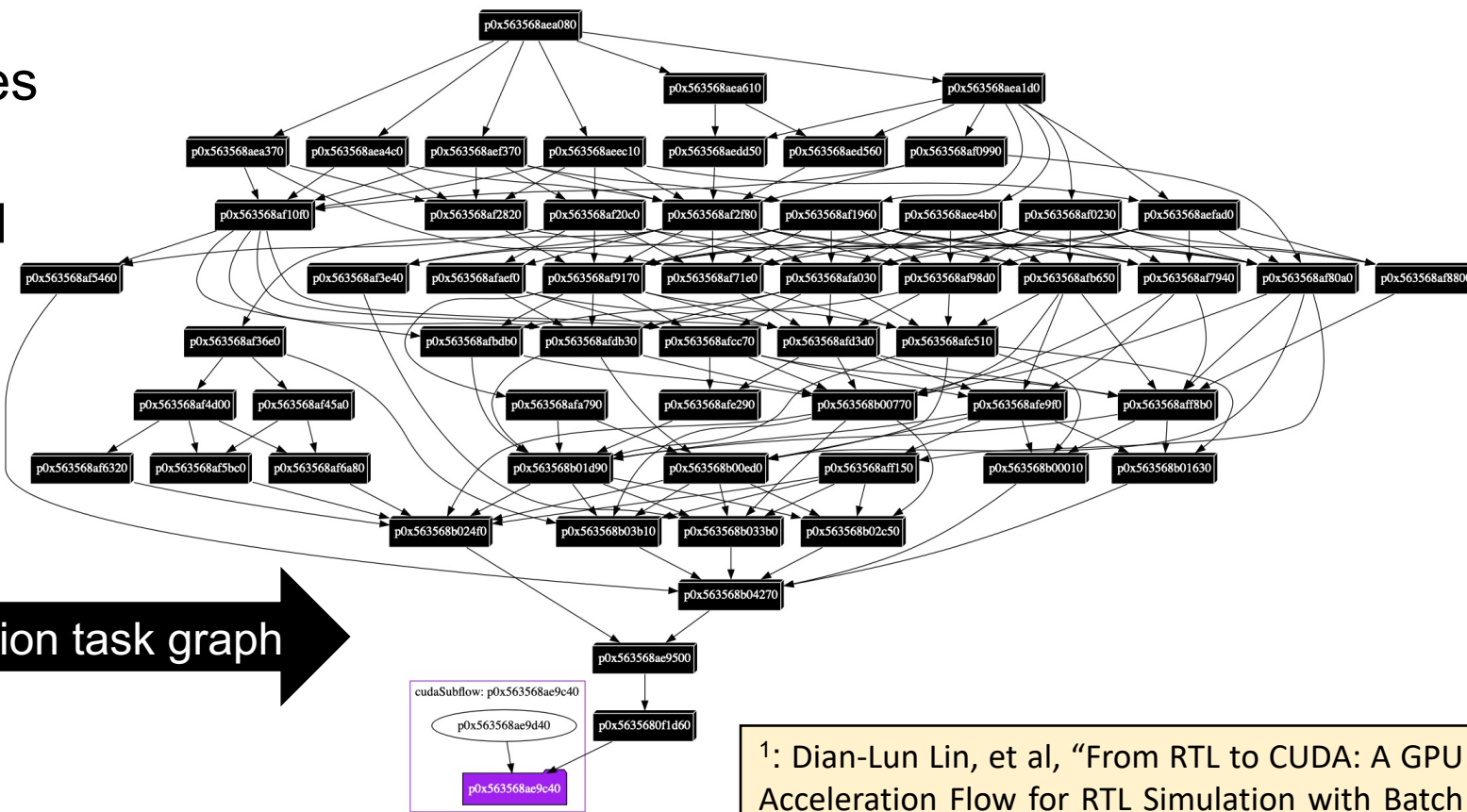
# Today's Parallel Workload is Very Complex

- GPU-accelerated circuit analysis on a design of 500M gates<sup>1</sup>

- >100 kernels
- >100 dependencies
- >500s to finish
- >10hrs turnaround



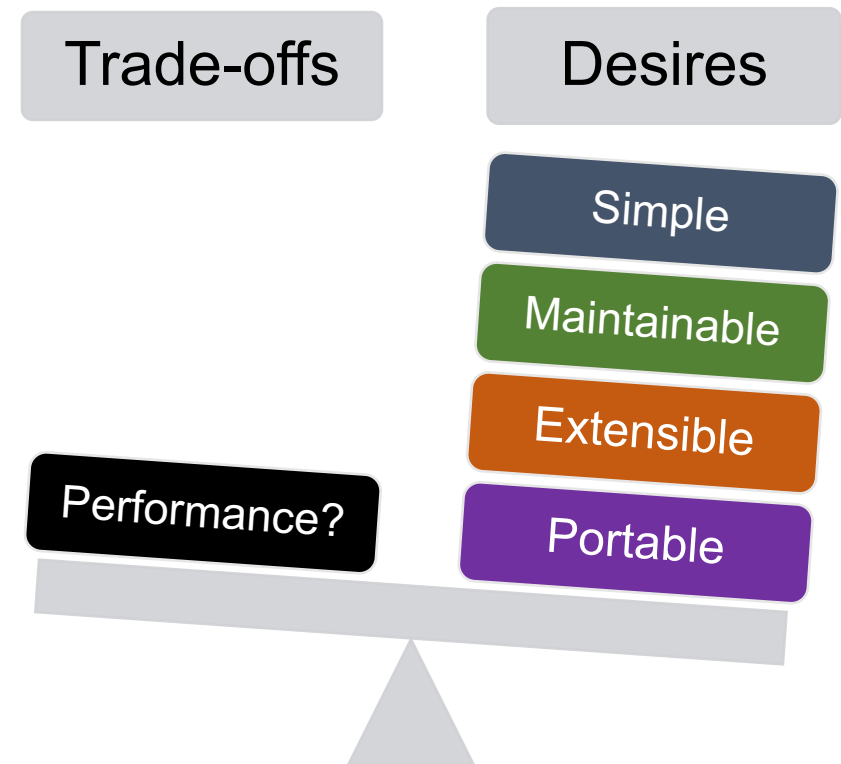
Simulation task graph



<sup>1</sup>: 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



# Need a Good Programming Abstraction

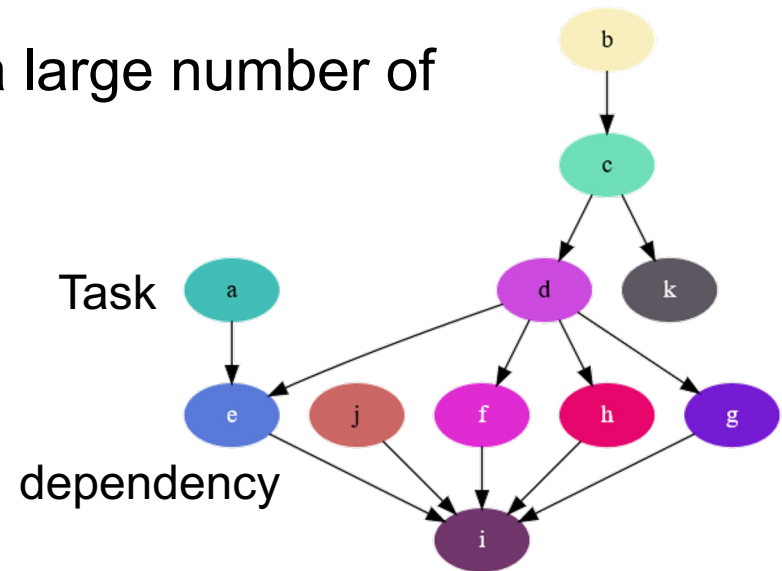
- From user's perspective, the biggest challenge is *transparency*
  - Programming abstraction, runtime optimization, load balancing, etc.
- Observing from the evolution of parallel programming:
  - **Task graph parallelism** (TGP) is the best model for future parallel arch
  - Capture programmers' intention 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)



StarPU



PaRSEC





# Takeaways

---

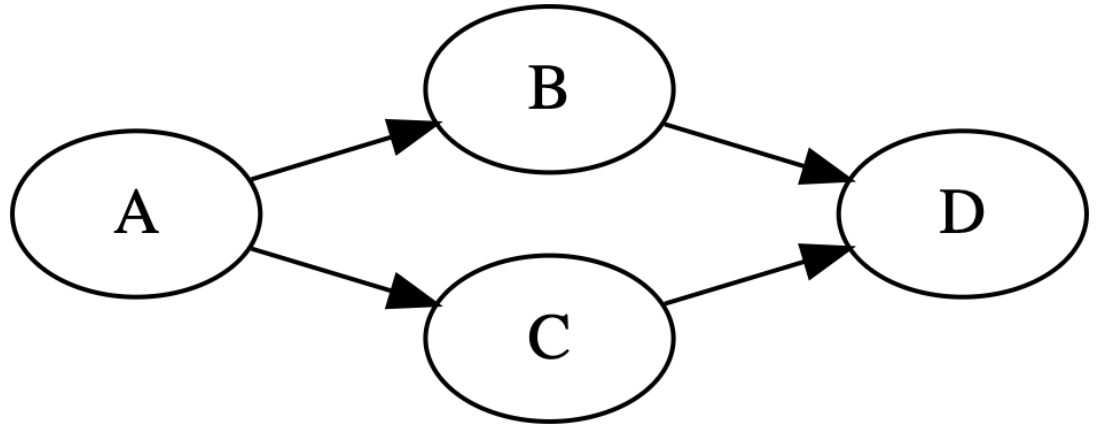
- Express your parallelism in the right way
- **Program static task graph parallelism using Taskflow**
- Program dynamic task graph parallelism using Taskflow
- Overcome the scheduling challenges
- Demonstrate the efficiency of Taskflow
- Conclude the talk



# “Hello World” in Taskflow<sup>1</sup>

```
#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>



<sup>1</sup>: 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

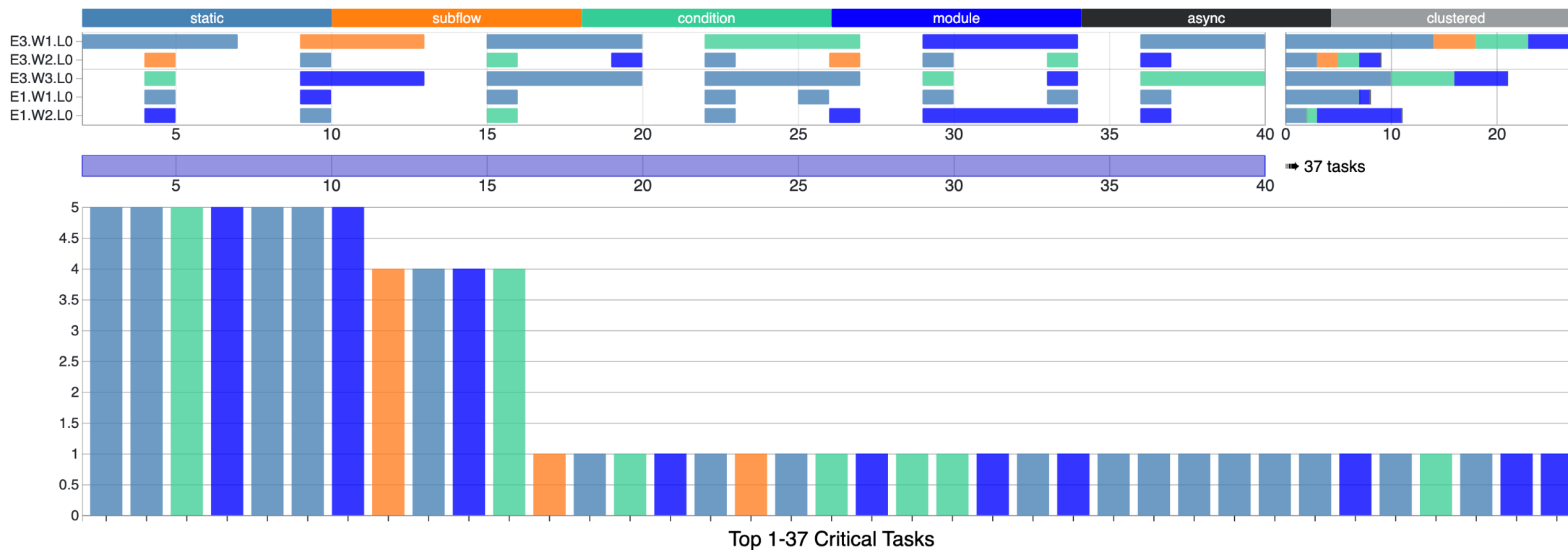
TaskB

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



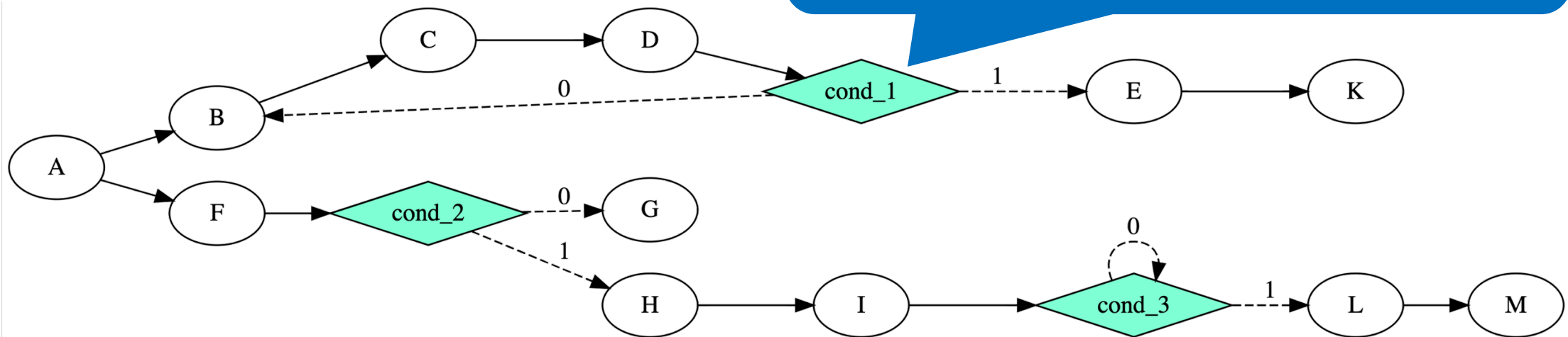
# Control Taskflow Graph Programming (CTFG)

// 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
  
```

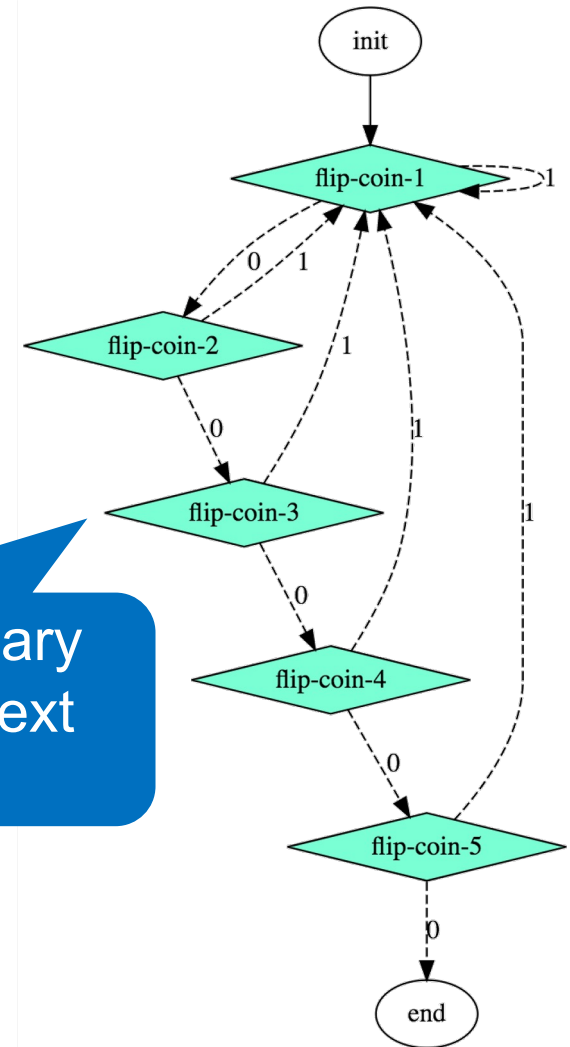
Very difficult for existing DAG-based systems to express an efficient overlap between tasks and control flow ...



# 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?

- **Unroll a task graph across fixed-length iterations**

- Task graph size becomes very large ...

- **Unknown or non-deterministic iterations?**

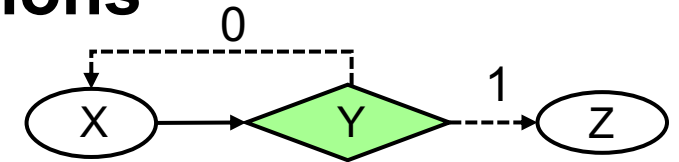
- Expensive dynamic tasking to run “if-else” on the fly

- **Dynamic control-flow tasks?**

- **Client-side partition at decision-making points**

- **Same problem in large-scale ML**

- TensorFlow with RNN (EuroSys'18)
- FlexFlow (MLSys'19, ICML'18)
- DGL (CoRR'19)
- ...

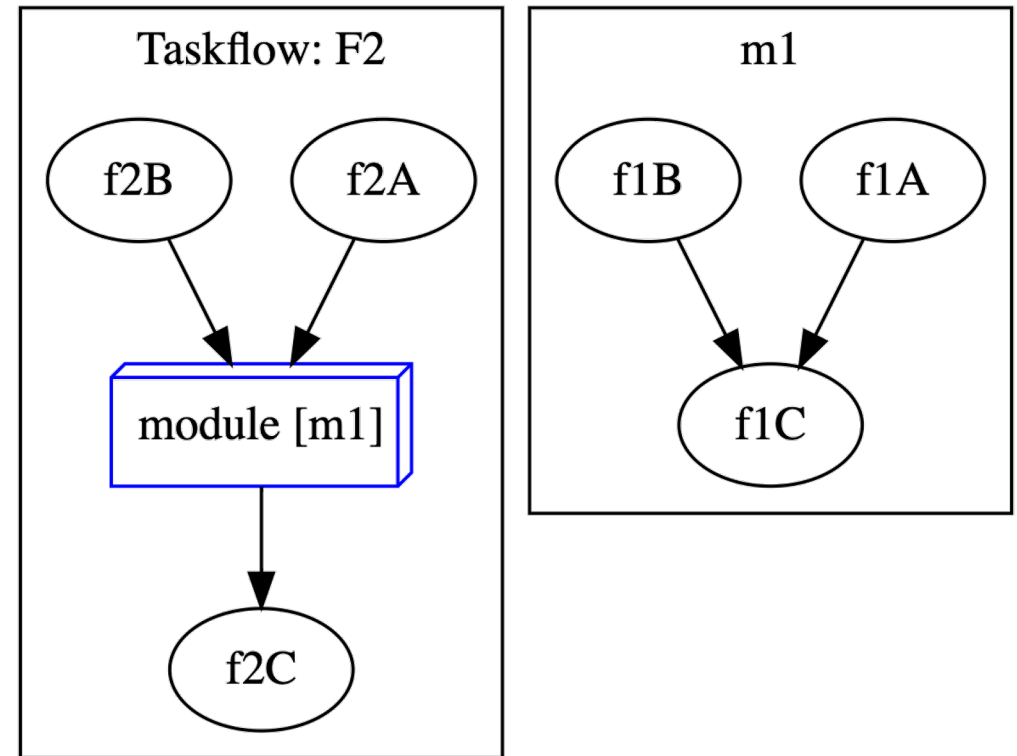


```
tf::Taskflow G;  
auto X = G.emplace([](){});  
auto Y = G.emplace([](){  
    return converged() ? 1 : 0;  
});  
cond.precede(Z, X);  
executor.run(G).wait();
```

```
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

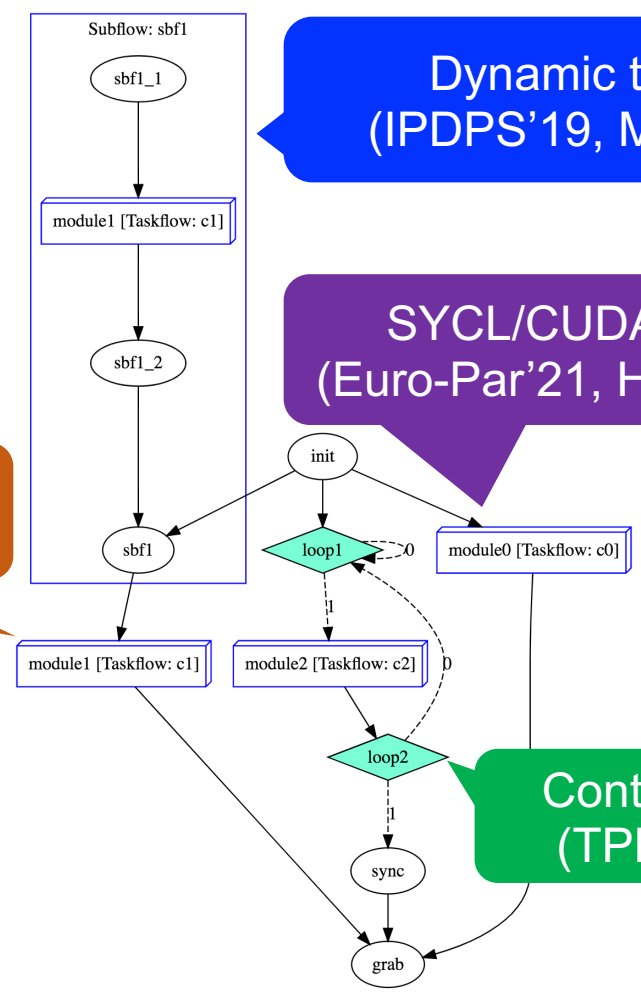
- **End-to-end parallelism in one graph**
  - Task, dependency, control flow all together
  - Scheduling with whole-graph optimization
  - Efficient overlap among heterogeneous tasks
- **Largely improved productivity!**

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

Dynamic task  
(IPDPS'19, MM'19)

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

Control flow  
(TPDS'22)



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.

8 ↓ Reply Share Report Save Follow





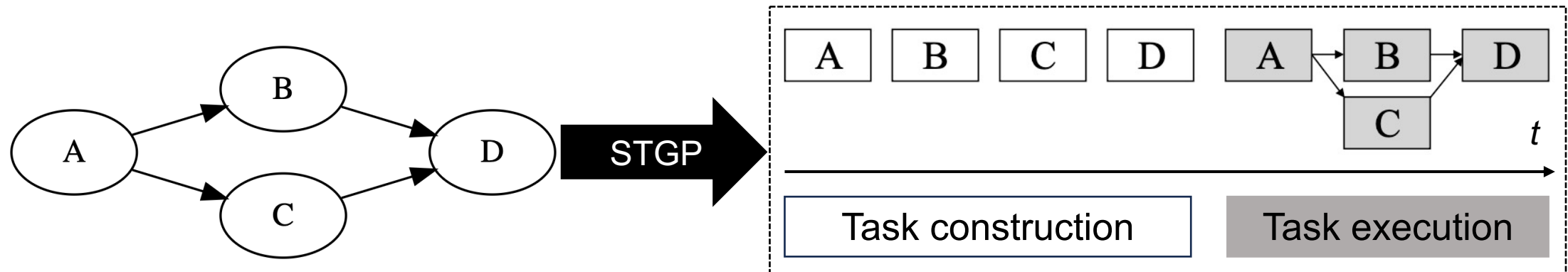
# Takeaways

---

- 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)

- **Lack of flexible and dynamic expressions of TGP**
  - Need to decide TGP up front – *task construction is separate from execution*
  - Need to decouple TGP from control flow – *task graph structure cannot depend on runtime values or control-flow results*
- **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



# Problem of STGP: Example #1

- Control flow-driven TGP programming ...

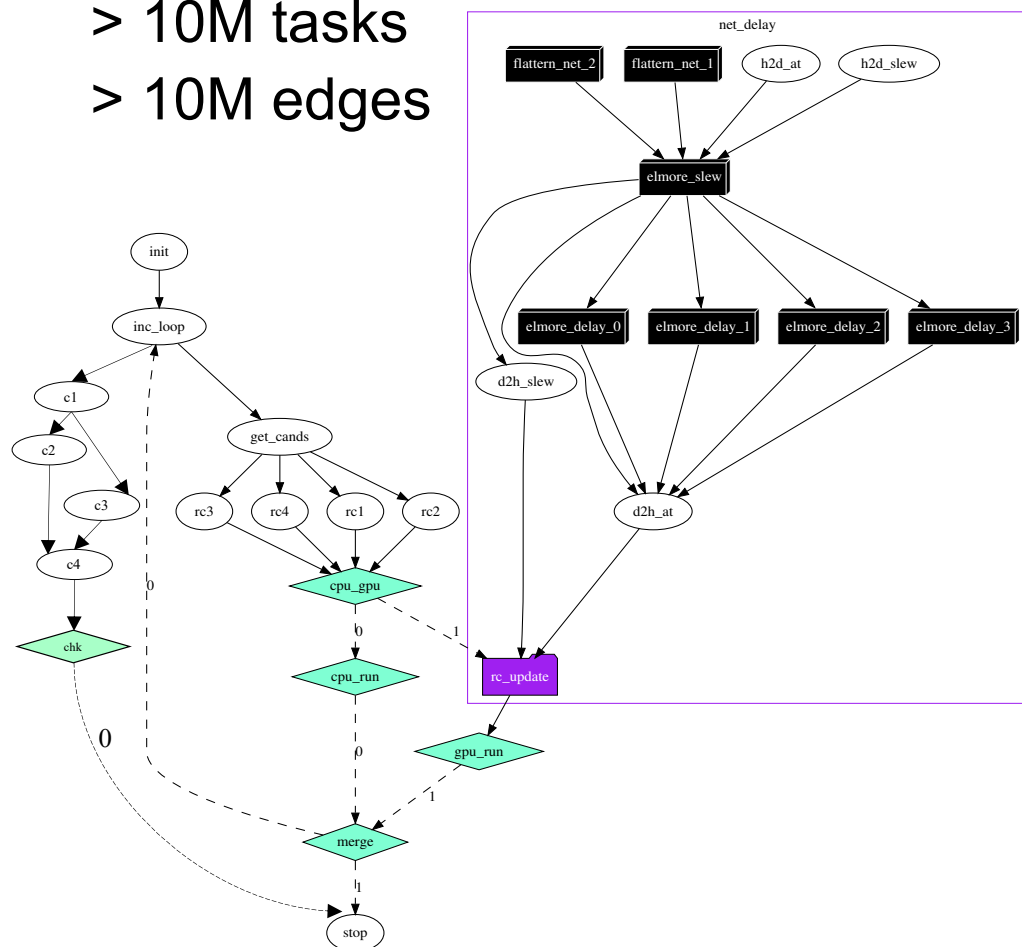
```
if (a == true) {
  G1 = build_task_graph1();
  if (b == true) {
    G2 = build_task_graph2();
    G1.precede(G2);
    if (c == true) {
      ... // another level of dynamic TGP
    }
  }
}
else {
  G3 = build_task_graph3();
  G3.precede(G1);
}
```

```
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) {
    {
      ... // another level of dynamic TGP
    }
  }
}
```

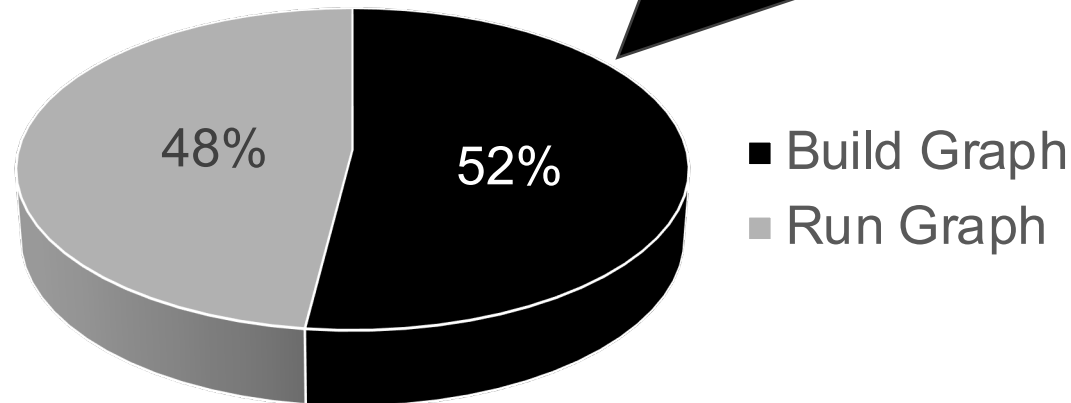
# Problem of STGP: Example #2

- Runtime breakdown of a task-parallel circuit analysis algorithm<sup>1</sup>

> 10M tasks  
> 10M edges



Task graph construction time takes over 50% of the entire runtime

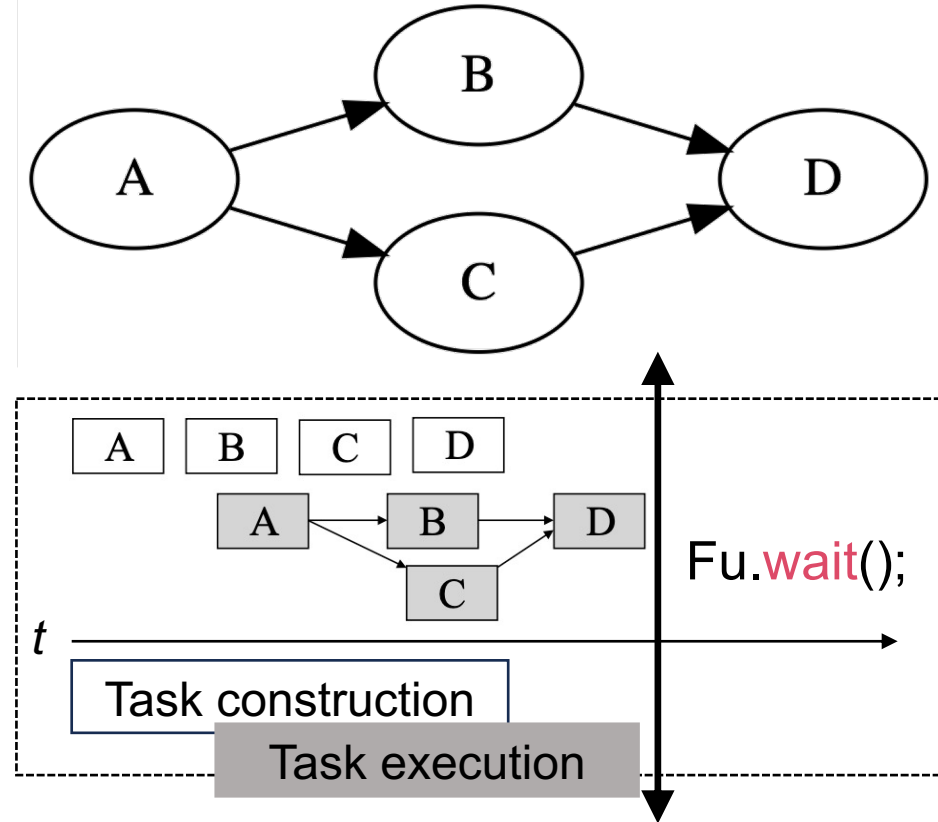


<sup>1</sup>: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2022

# 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();
```

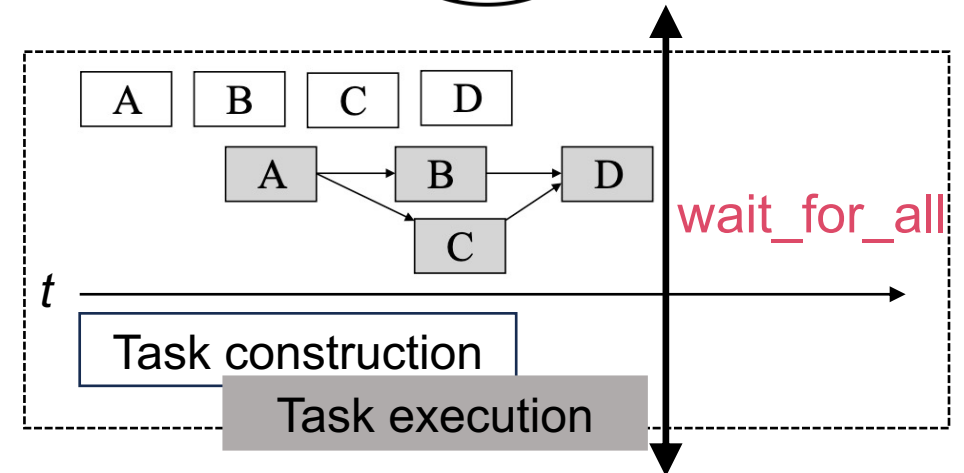
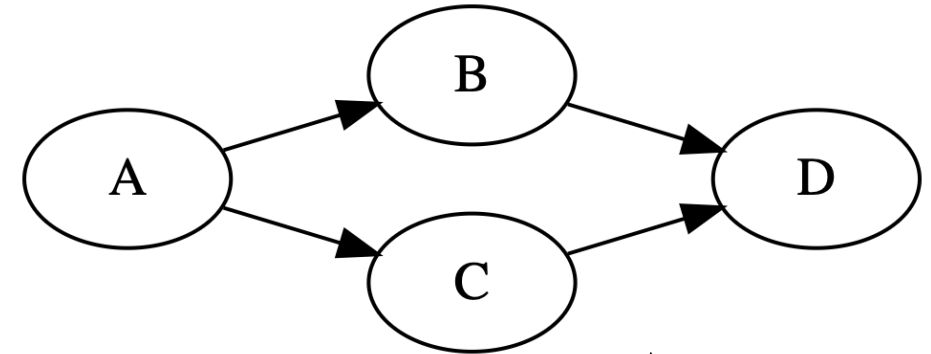


Specify arbitrary task dependencies using C++ variadic parameter pack

# Wait for All Tasks to Finish

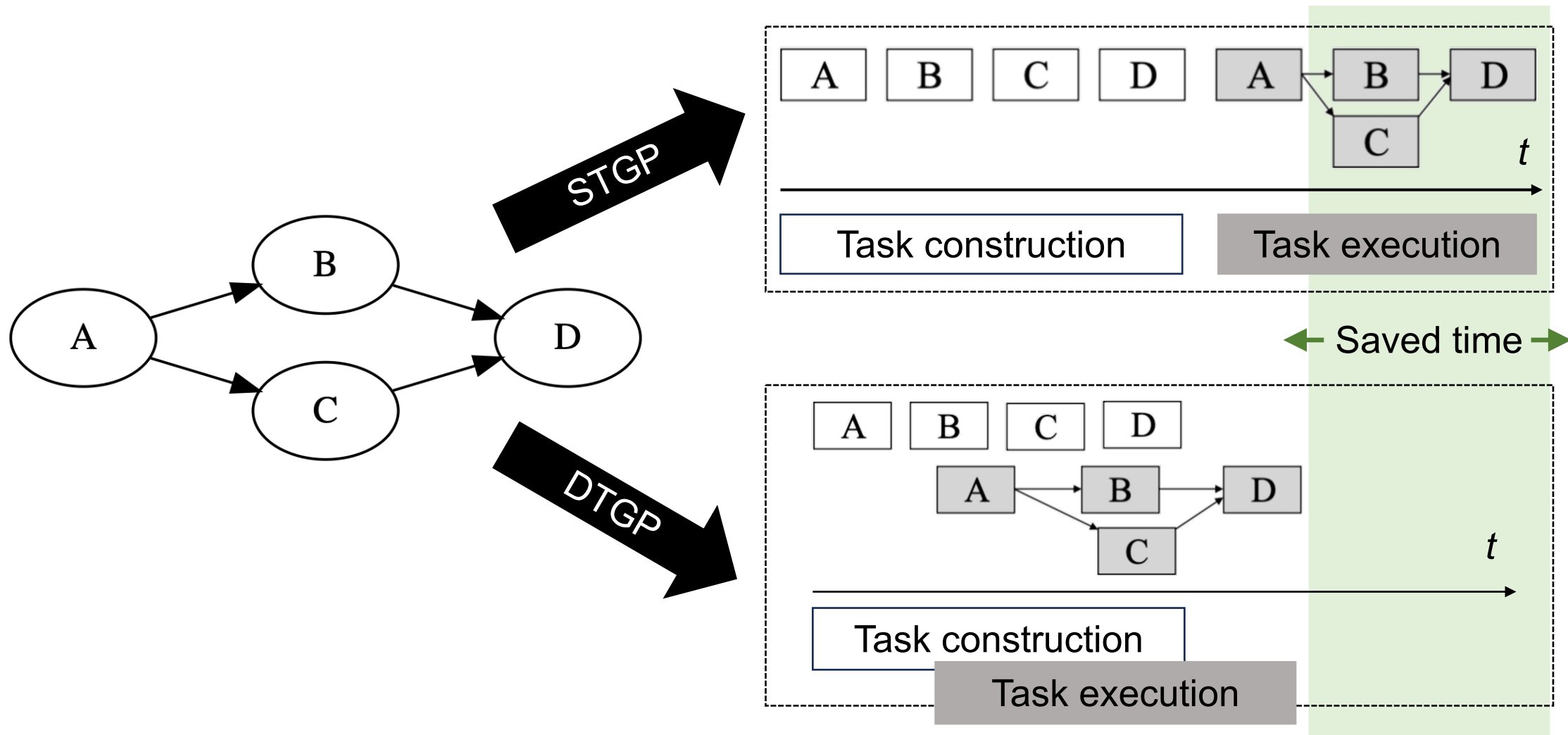
// Live: <https://godbolt.org/z/T87PrTarx>

```
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

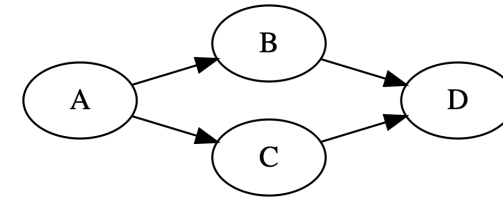
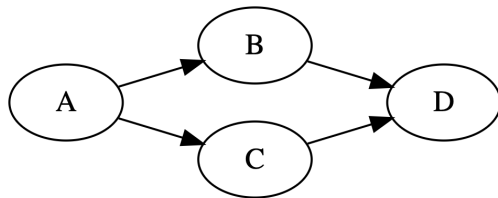




# DTGP Needs a Correct Topological Order

```
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

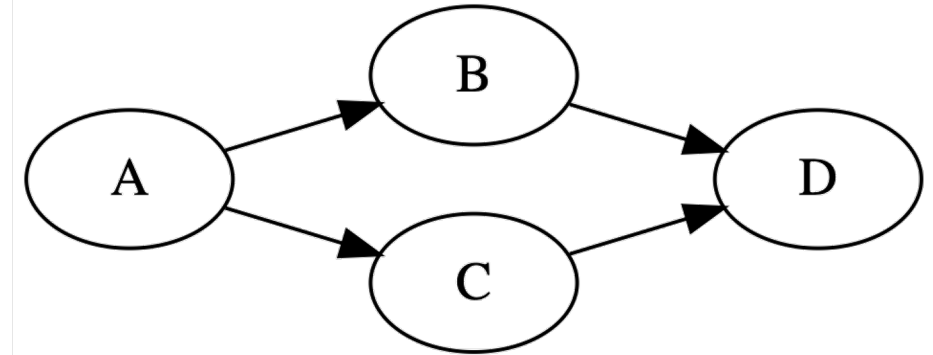


Topological order #2: A→C→B→D

```
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 are unknown at programming time

// 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());
```



# Existing DTGP Libraries: C++ Async

```
auto A = std::async([&]() { std::cout << "TaskA\n"; });
```

```
auto B = std::async([&]() {
```

```
  A.get(); ←
```

```
  std::cout << "TaskB\n";
```

```
});
```

```
auto C = std::async([&]() {
```

```
  A.get(); ←
```

```
  std::cout << "TaskC\n";
```

```
});
```

```
auto D = std::async([&]() {
```

```
  B.get();
```

```
  C.get(); ←
```

```
  std::cout << "TaskD\n";
```

```
}, B, C);
```

```
D.wait();
```

Block until A completes

Block until A completes

Block until B and C  
complete



# Existing DTGP Libraries: OpenMP/Kokkos

## OpenMP “dependency clauses”

```
#omp parallel num_threads(hardware_concurrency())
{
  int A_B, A_C, B_D, C_D;
  #pragma omp task depend(out: A_B, A_C)
  {
    std::cout << "TaskA\n" ;
  }
  #pragma omp task depend(in: A_B; out: B_D)
  {
    std::cout << "TaskB\n" ;
  }
  #pragma omp task depend(in: A_C; out: C_D)
  {
    std::cout << "TaskC\n" ;
  }
  #pragma omp task depend(in: B_D, C_D)
  {
    std::cout << "TaskD\n" ;
  }
}
```


## Kokkos “task template”

```
struct A {
  template <class TeamMember> KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member) { std::cout << "TaskA\n"; }
};
struct B {
  template <class TeamMember> KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member) { std::cout << "TaskB\n"; }
};
struct C {
  template <class TeamMember> KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member) { std::cout << "TaskC\n"; }
};
struct D {
  template <class TeamMember> KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member) { std::cout << "TaskD\n"; }
};
auto scheduler = scheduler_type(/* ... */);
auto futA = Kokkos::host_spawn(Kokkos::TaskSingle(scheduler), A());
auto futB = Kokkos::host_spawn(Kokkos::TaskSingle(scheduler, futA), B());
auto futC = Kokkos::host_spawn(Kokkos::TaskSingle(scheduler, futA), C());
auto futD = Kokkos::host_spawn(
  Kokkos::TaskSingle(scheduler, when_all(futB, futC)), D()
);
... (more code to follow)
```



# Existing DTGP Libraries: PARSEC

```
int A(parsec_task_t* this_task) {
    int *out;
    unpack_args(this_task, &out);
    printf("TaskA\n");
    return APARSEC_HOOK_RETURN_DONE;
}
int B(parsec_task_t* this_task) {
    int *out, *in;
    unpack_args(this_task, &in, &out);
    printf("TaskB\n");
    return APARSEC_HOOK_RETURN_DONE;
}
int C(parsec_task_t* this_task) {
    int *in, *out;
    unpack_args(this_task, &in, &out);
    printf("TaskC\n");
    return APARSEC_HOOK_RETURN_DONE;
}
int D(parsec_task_t* this_task) {
    int *in1, *in2, *out;
    unpack_args(this_task, &in1, &in2, &out);
    printf("TaskD\n");
    return APARSEC_HOOK_RETURN_DONE;
}
```



```
int main() {
    /* additional boilerplate code to initialize PARSEC runtime
    environment (e.g., taskpool, dtd_tp, dependency data, etc.) */
    int *out;
    parsec_dtd_insert_task(A,
        tile_of_key(dependency, 0), INPUT, ARSEC_DTD_ARG_END
    );
    parsec_dtd_insert_task(B,
        tile_of_key(dependency, 0), INPUT,
        tile_of_key(dependency, 1), OUTPUT, PARSEC_DTD_ARG_END
    );
    parsec_dtd_insert_task(C,
        tile_of_key(dependency, 0), INPUT,
        tile_of_key(dependency, 2), OUTPUT, PARSEC_DTD_ARG_END
    );
    parsec_dtd_insert_task(D,
        tile_of_key(dependency, 1), INPUT,
        tile_of_key(dependency, 2), INPUT,
        tile_of_key(dependency, 3), OUTPUT, PARSEC_DTD_ARG_END
    );
    parsec_taskpool_wait();
}
```



# Limitations of Existing DTGP Libraries

---

- 1. Suffer from large verbosity from the ease-of-use standpoint**
  - The code does not explain itself – *I know this is subjective, but most applications just care how fast they can get things done ...*
- 2. Count on dataflow to express task dependencies**
  - Users need to explicitly define per-edge data to specify a task dependency
    - OpenMP's **in-out** dependency clauses
    - PARSEC's **INPUT-OUTPUT** keywords and **tile\_of\_key** constructs
- 3. Rely on complex lock-based data structure to schedule tasks**
  - Libomp implements a lock-based hash table to schedule tasks
    - Key: address of the input and output data of a task
    - Value: a list of tasks accessing that input and output data
  - Frequent access to this data structure results in large runtime overhead





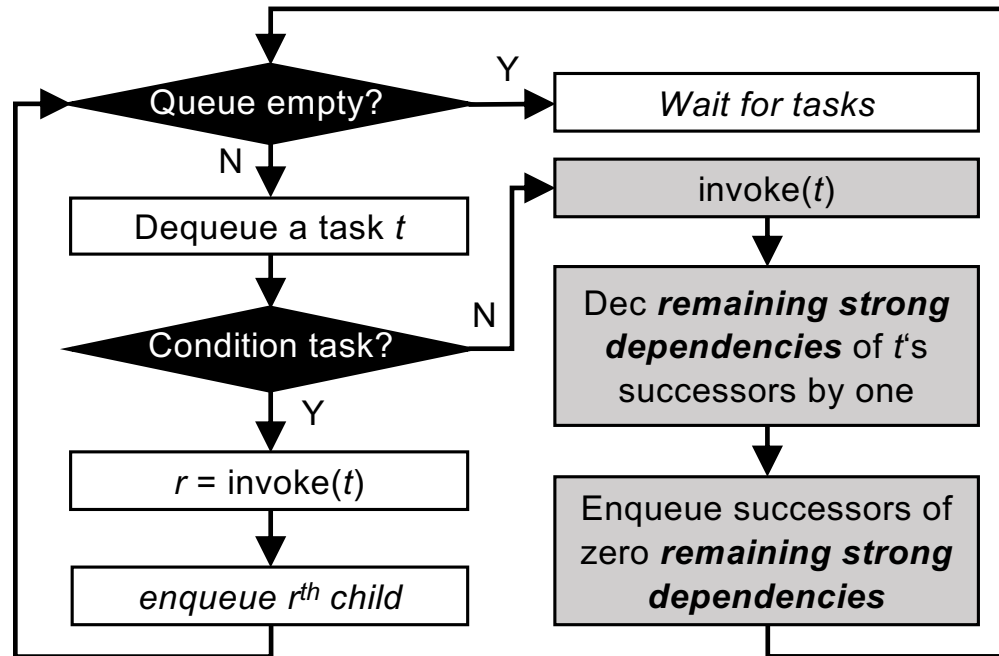
# Takeaways

---

- 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

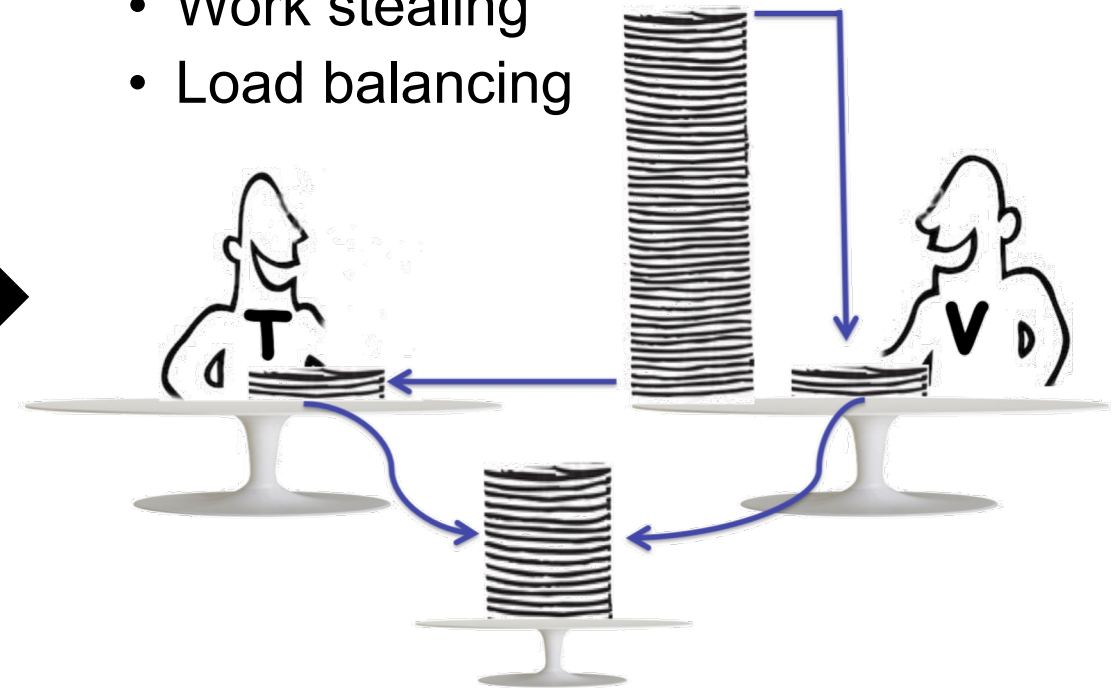
# STGP Scheduling Algorithm

- Task-level scheduling



- Worker-level scheduling

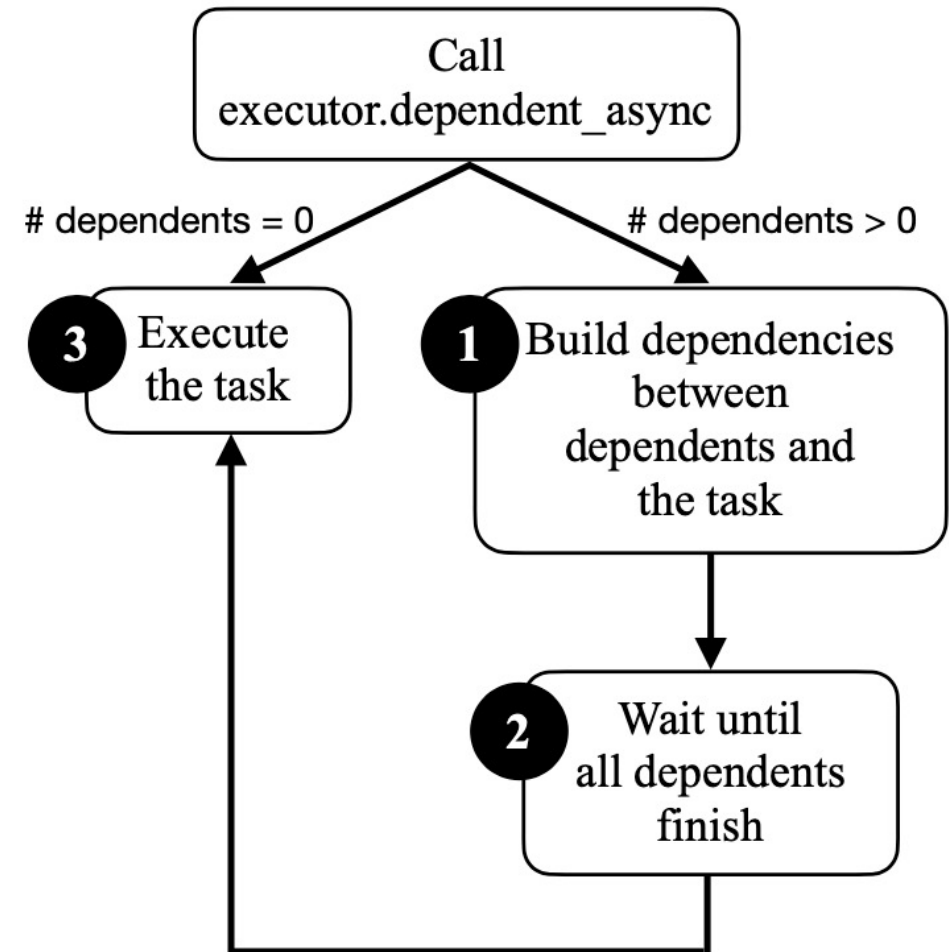
- Work stealing
- Load balancing



**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*

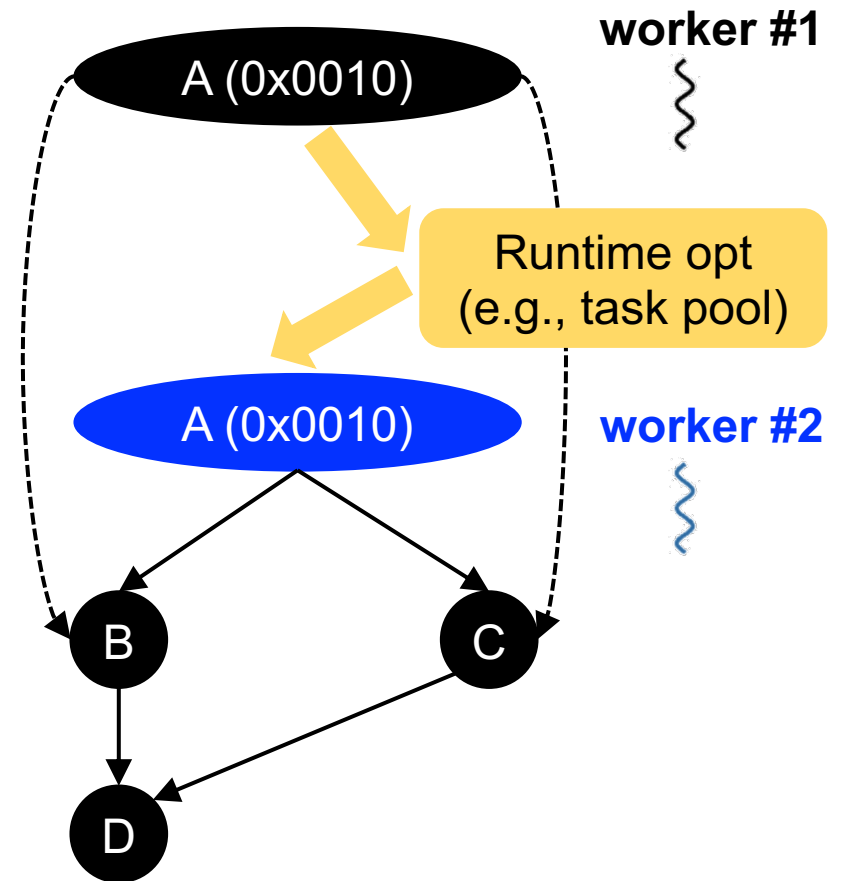
# DTGP Scheduling Algorithm

- **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<sup>1</sup>

```
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();
```



<sup>1</sup>: ABA Problem: [https://en.wikipedia.org/wiki/ABA\\_problem](https://en.wikipedia.org/wiki/ABA_problem)



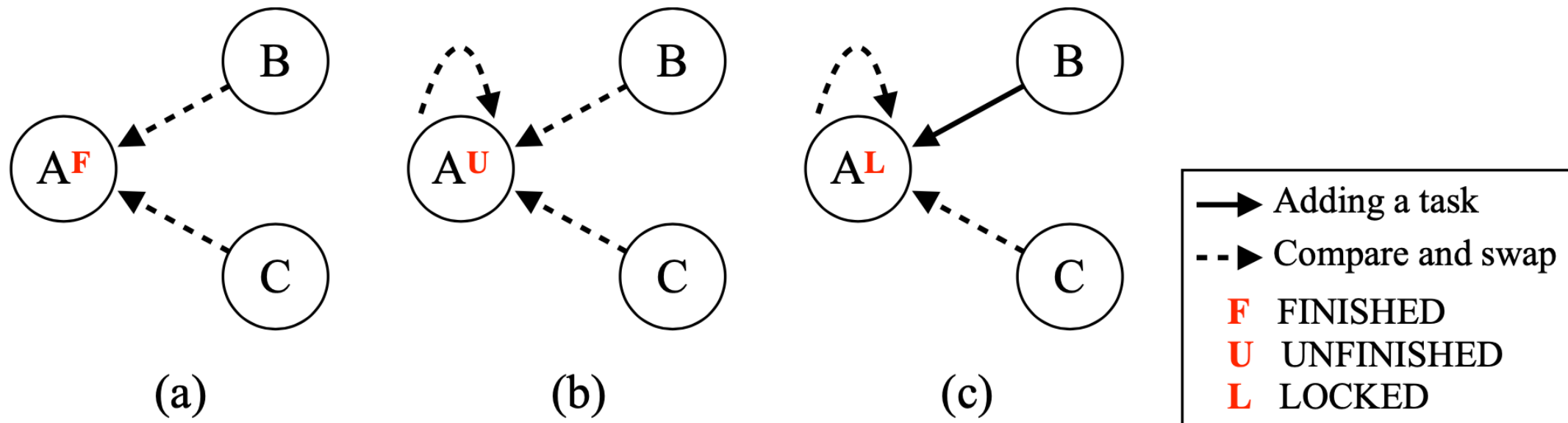
# Retain Shared Ownership of Every Task

```
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** acts 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<sup>1</sup>

---

## Algorithm 1 dependent\_async(callable, deps)

---

```
1: Create a future
2:  $num\_deps \leftarrow \text{sizeof}(deps)$ 
3:  $task \leftarrow \text{initialize\_task}(callable, num\_deps, future)$ 
4: for all  $dep \in deps$  do
5:    $\text{process\_dependent}(task, dep, num\_deps)$ 
6: end for
7: if  $num\_deps == 0$  then
8:    $\text{schedule\_async\_task}(task)$ 
9: end if
10: 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:    $num\_deps \leftarrow \text{AtomDec}(task.join\_counter)$ 
8: else
9:   goto line 2
10: end if
```

---

---

## Algorithm 3 schedule\_async\_task(task)

---

```
1:  $target\_state \leftarrow UNFINISHED$ 
2: while not  $task.state.CAS(target\_state, FINISHED)$ 
   do
3:    $target\_state \leftarrow UNFINISHED$ 
4: end while
5:  $\text{Invoke}(task.callable)$ 
6: for all  $successor \in task.successors$  do
7:   if  $\text{AtomDec}(successor.join\_counter) == 0$  then
8:      $\text{schedule\_async\_task}(successor)$ 
9:   end if
10: end for
11: if  $\text{AtomDec}(task.ref\_count) == 0$  then
12:    $\text{Delete } task$ 
13: end if
```

---

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





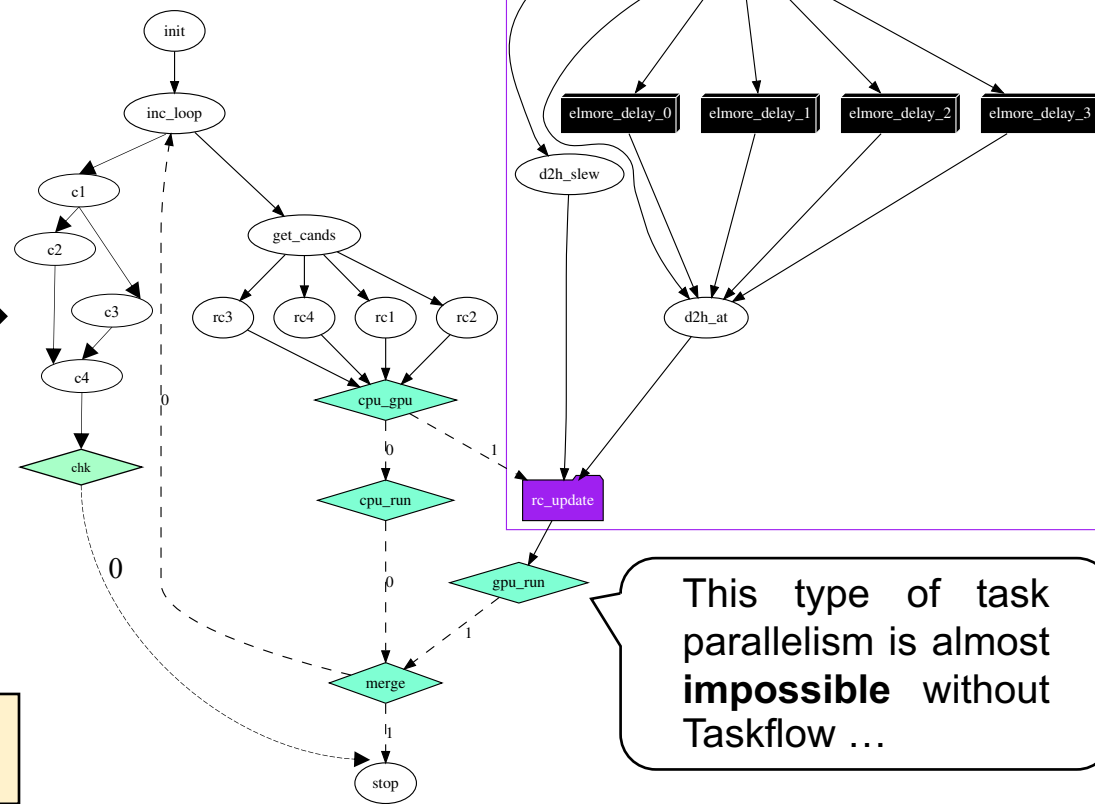
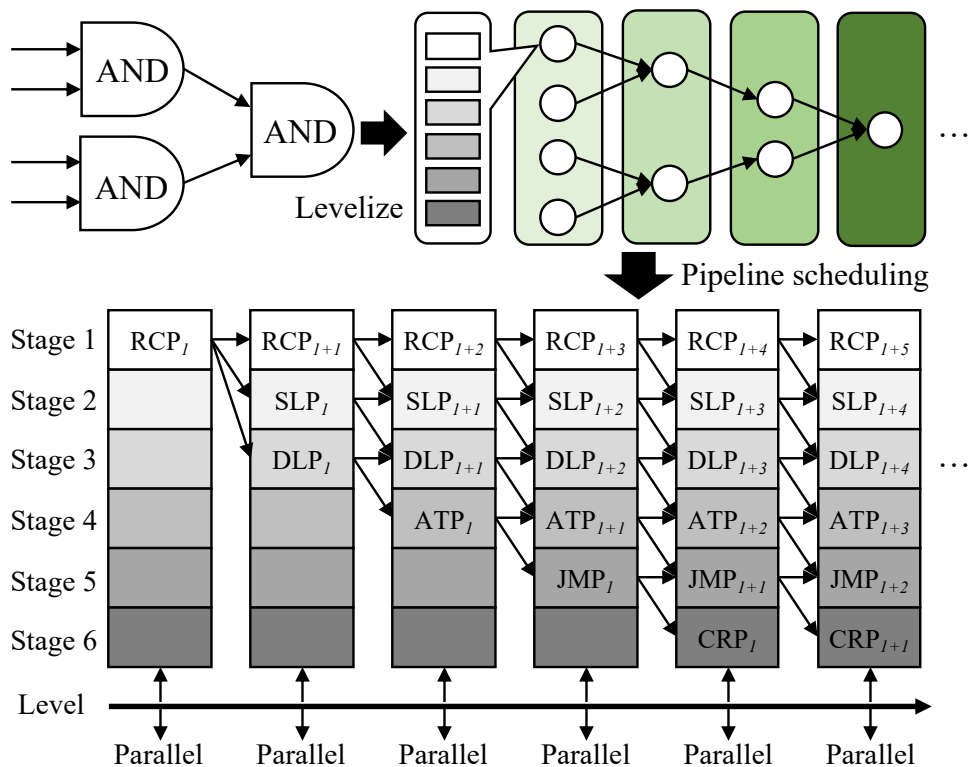
# Takeaways

---

- 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

# Experimental Results – STGP

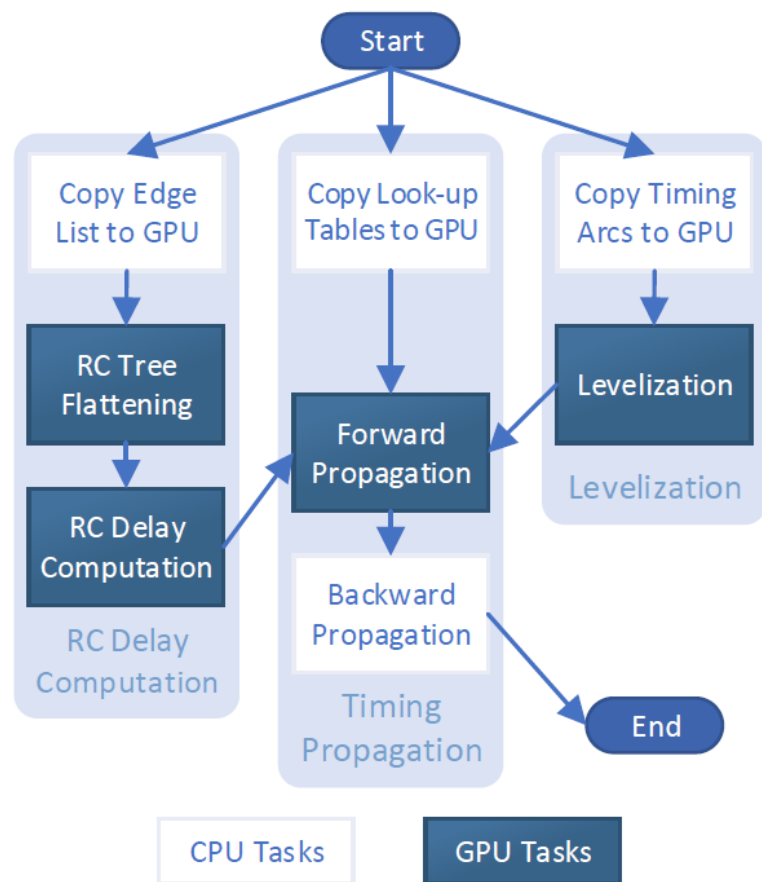
- Accelerating static timing analysis (STA)<sup>1</sup>



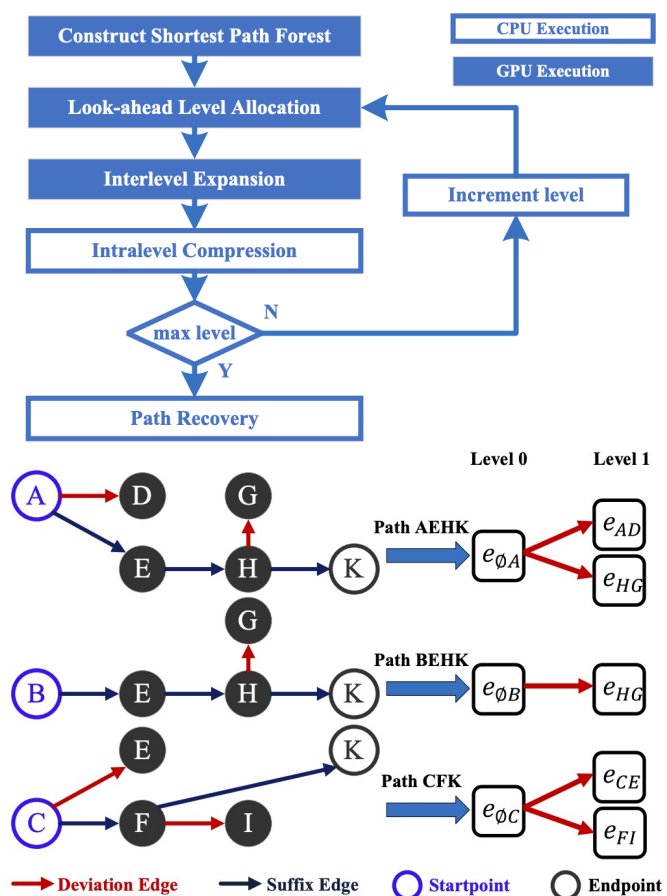
<sup>1</sup>: Tsung-Wei Huang, et al, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, 2022



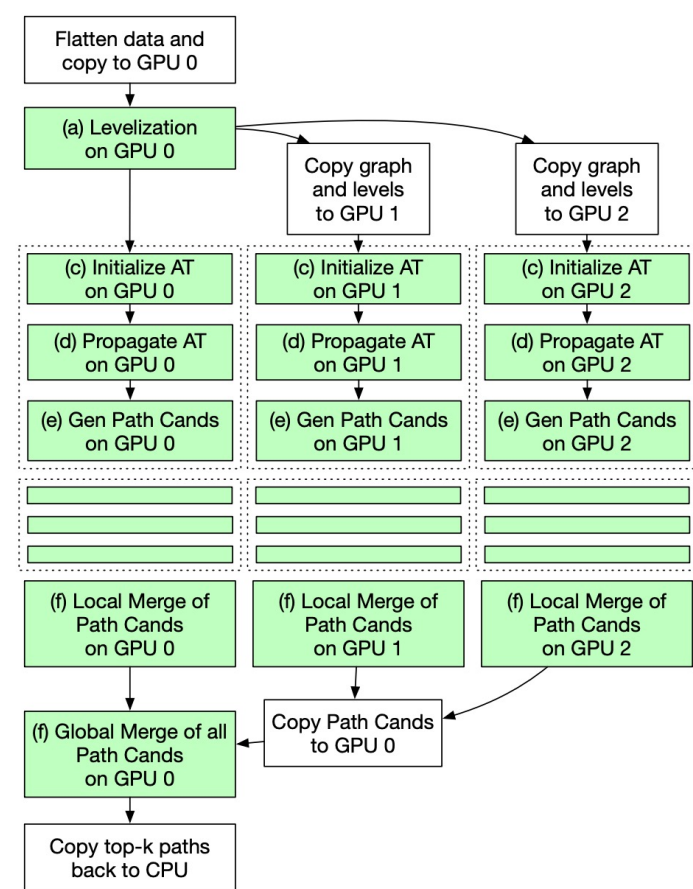
# Our Research on Task-parallel STA



GPU-based graph analysis (ICCAD'20)



GPU-based path analysis (DAC'21)



GPU-based CPPR (ICCAD'21)



# Example: Task-parallel Path Generation<sup>1</sup>

- **Applied Taskflow to accelerate path-based analysis on GPU**
  - Ex: leon3mp (1.6M gates): **611x speed-up** over 1 CPU (**44x** over 40 CPUs)
  - **Best paper award** in ACM TAU 2021

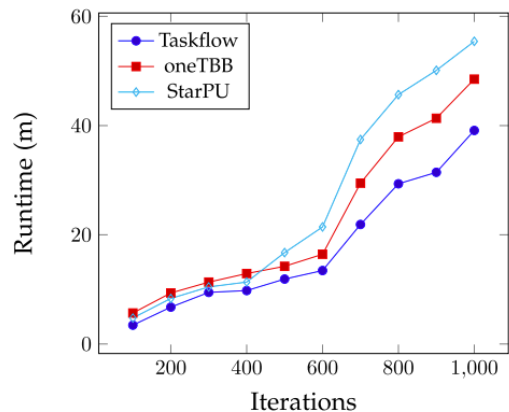
Benchmark	#Pins	#Gates	#Arcs	OpenTimer Runtime	Our Algorithm #MDL=10		Our Algorithm #MDL=15		Our Algorithm #MDL=20	
					Runtime	Speed-up	Runtime	Speed-up	Runtime	Speed-up
leon2	4328255	1616399	7984262	2875783	4708.36	611×	5295.49ms	543×	5413.84	531×
leon3mp	3376821	1247725	6277562	1217886	5520.85	221×	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×

<sup>1</sup>: Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong, "GPU-accelerated Path-based Timing Analysis," *IEEE/ACM Design Automation Conference (DAC)*, CA, 2021

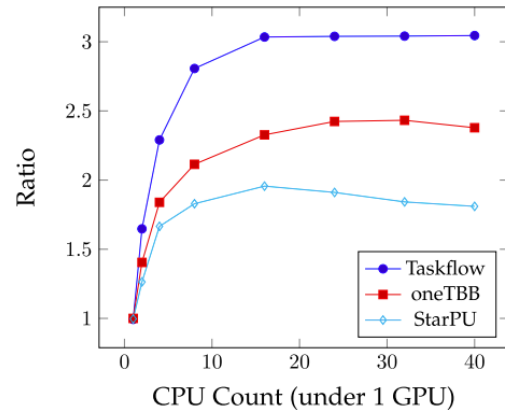


# Comparison with Existing TGP Systems

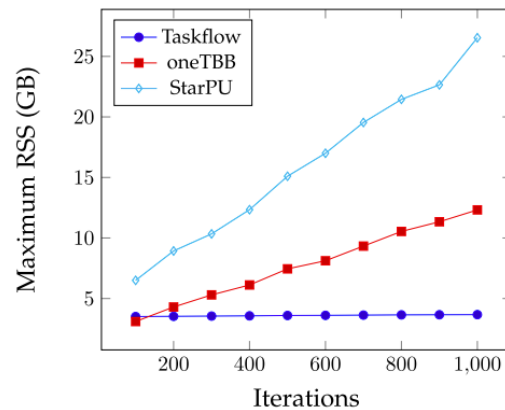
Runtime (40 CPUs 1 GPU)



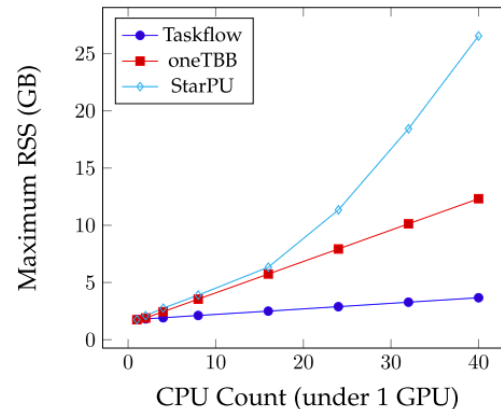
Speed-up (1000 iterations)



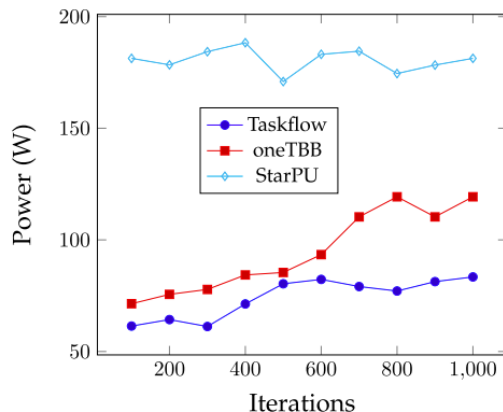
Memory (40 CPUs 1 GPU)



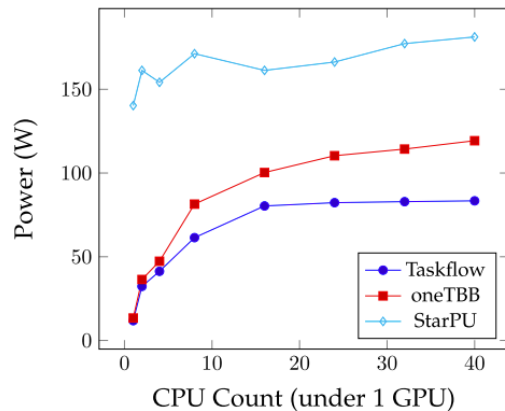
Memory (1000 iterations)



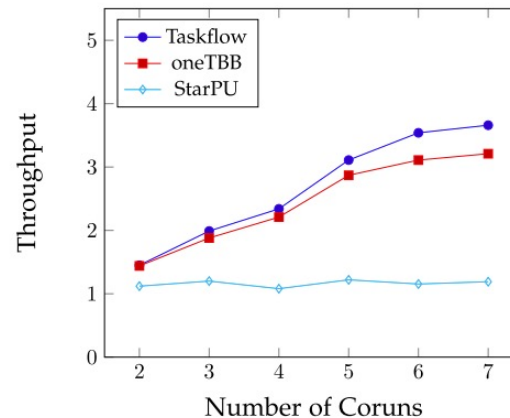
Power (40 CPUs 1 GPU)



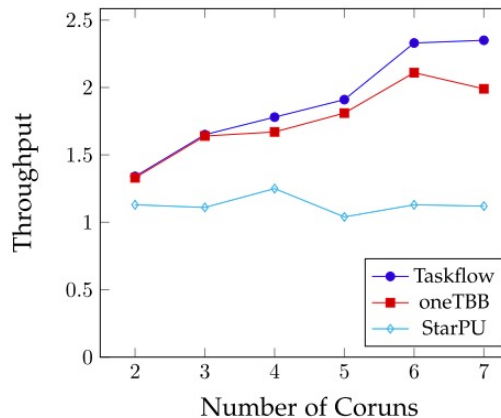
Power (1000 iterations)



Corun (500 iterations)

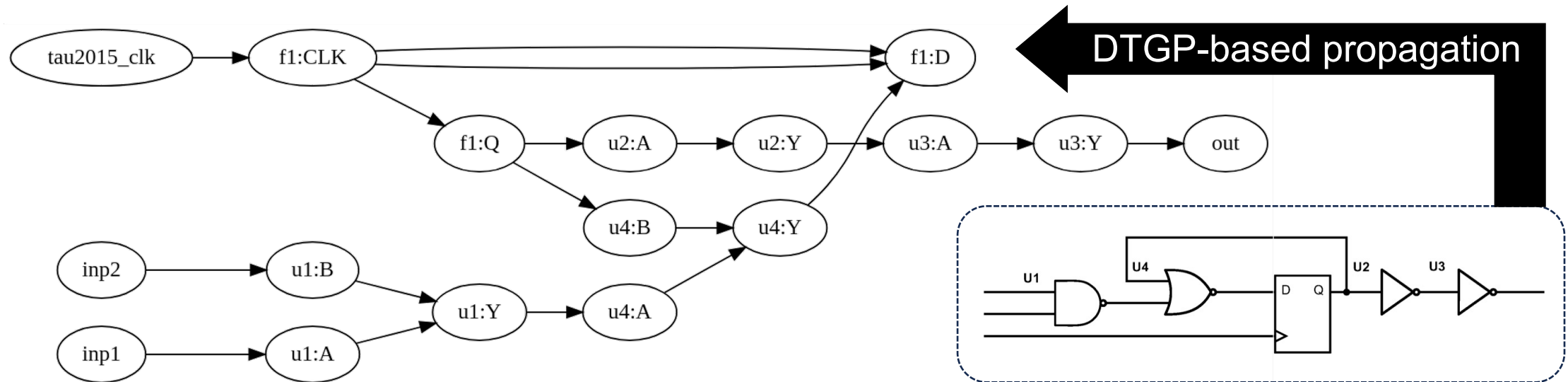


Corun (1000 Iterations)



# Experimental Results – DTGP

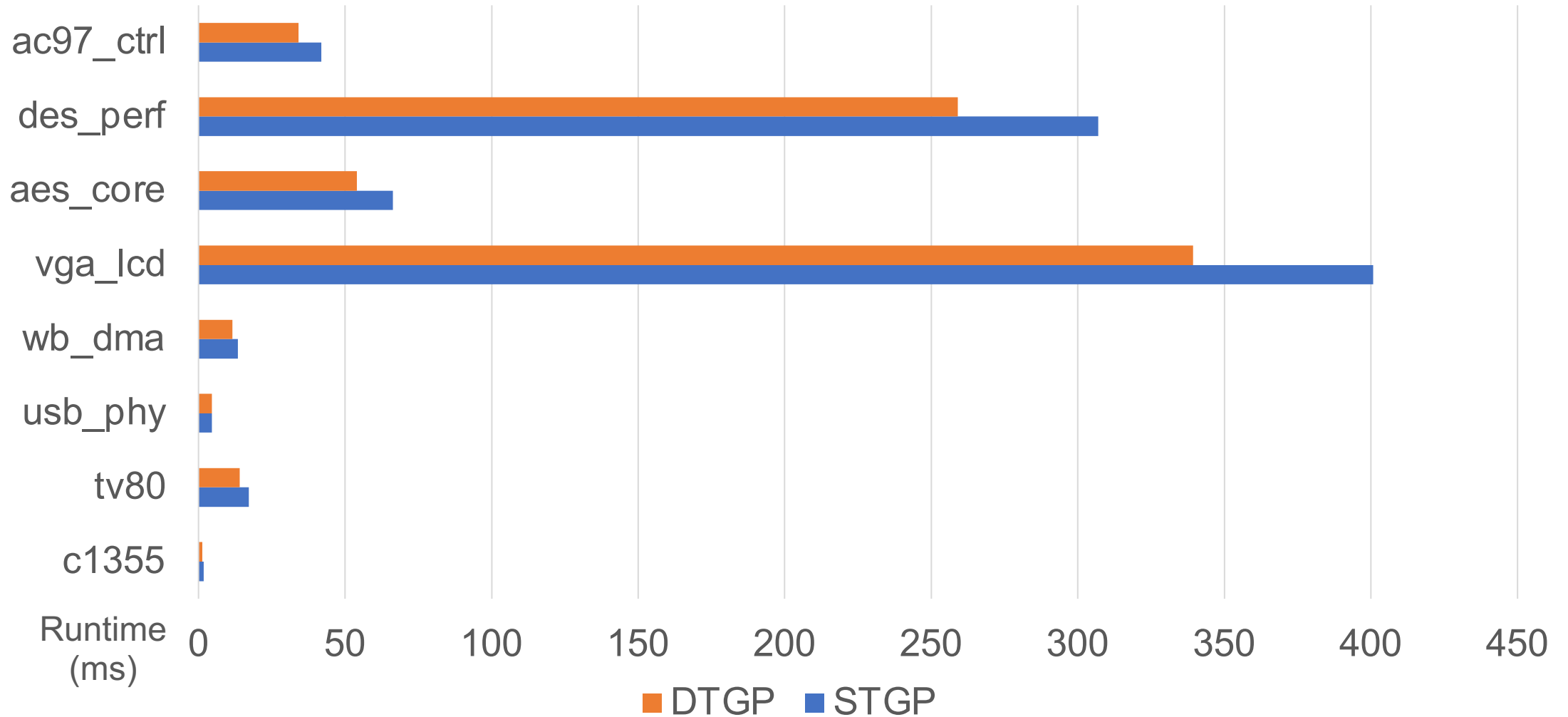
- **Evaluated on a real-world static timing analysis application<sup>1</sup>**
  - Formulated the timing propagation algorithm into a task graph
    - Large circuits can compose millions of tasks and dependencies
  - Ex (below): a task graph for a full-timing propagation on a five-gate circuit



<sup>1</sup>: 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



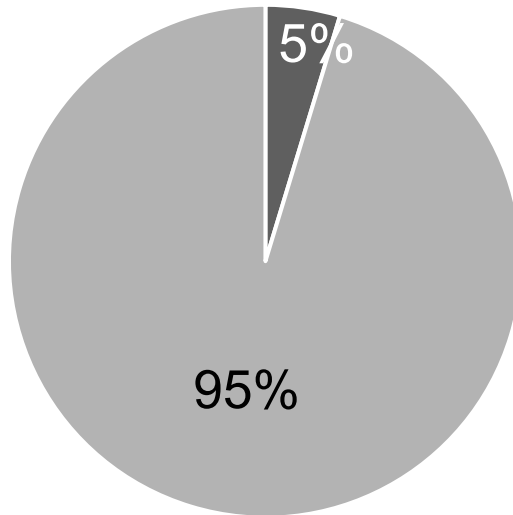
# Runtime Comparison: STGP vs DTGP



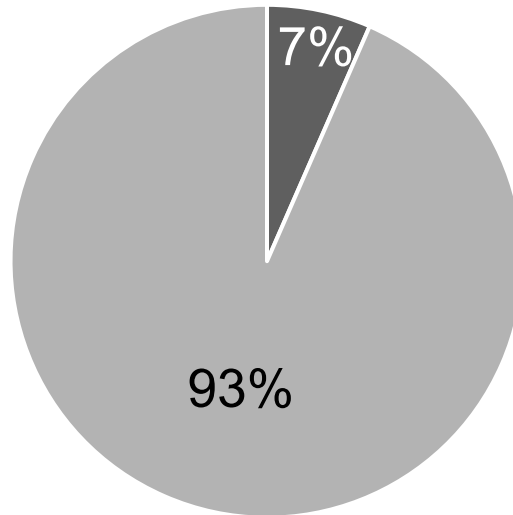
# Runtime Breakdown of STGP

- Graph construction time increases as the circuit size increases

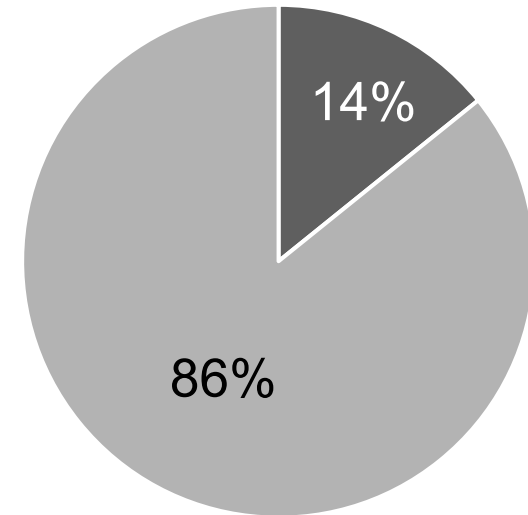
c1355  
(617 tasks)



des\_perf  
(304K tasks)



vga\_lcd  
(398K tasks)



■ Build Graph ■ Run Graph

■ Build Graph ■ Run Graph

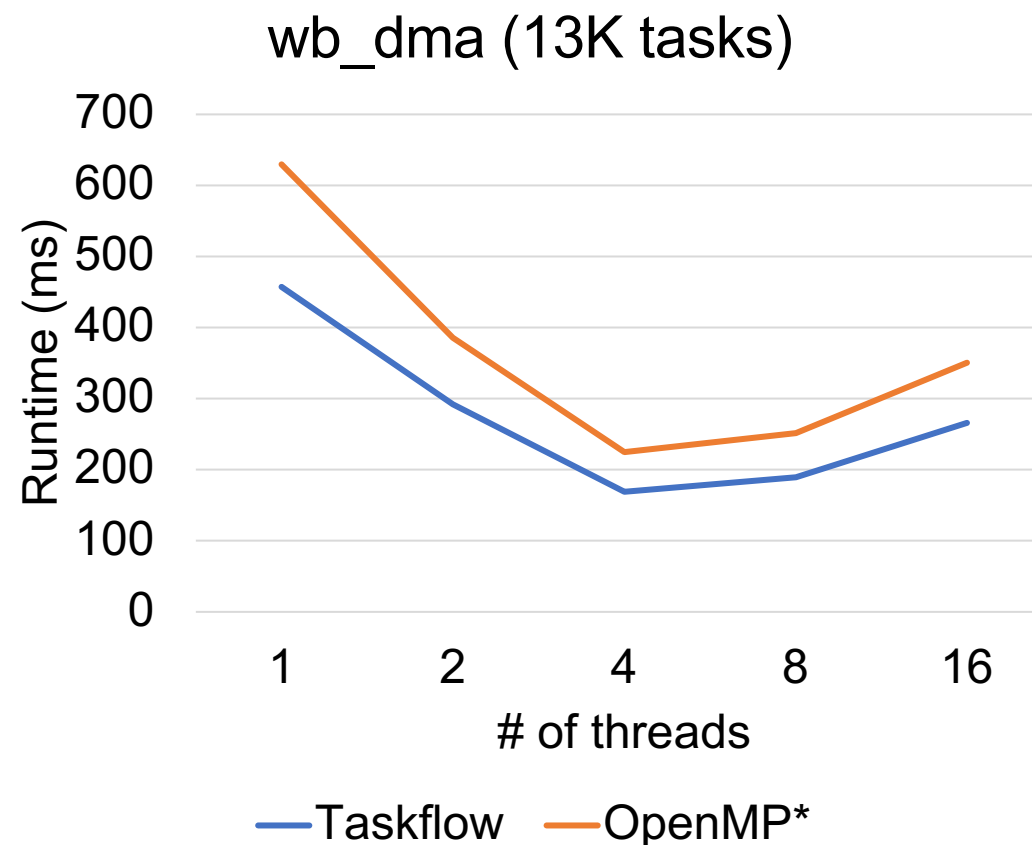
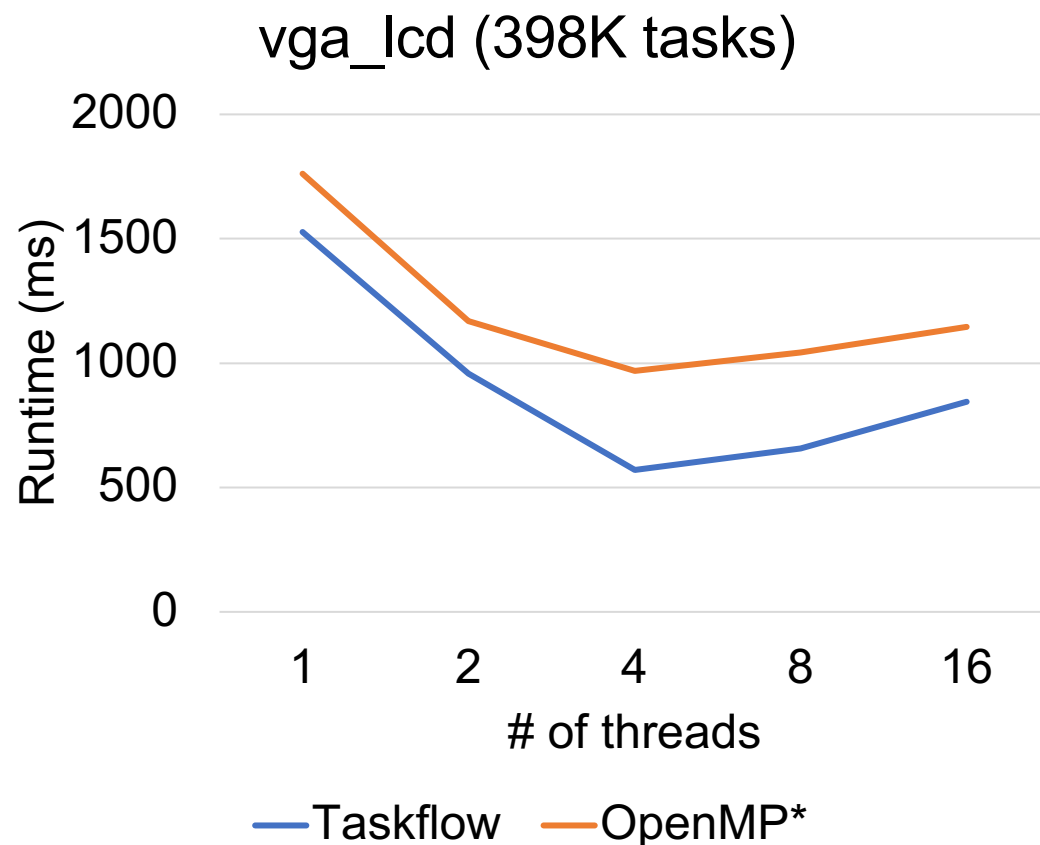
■ Build Graph ■ Run Graph





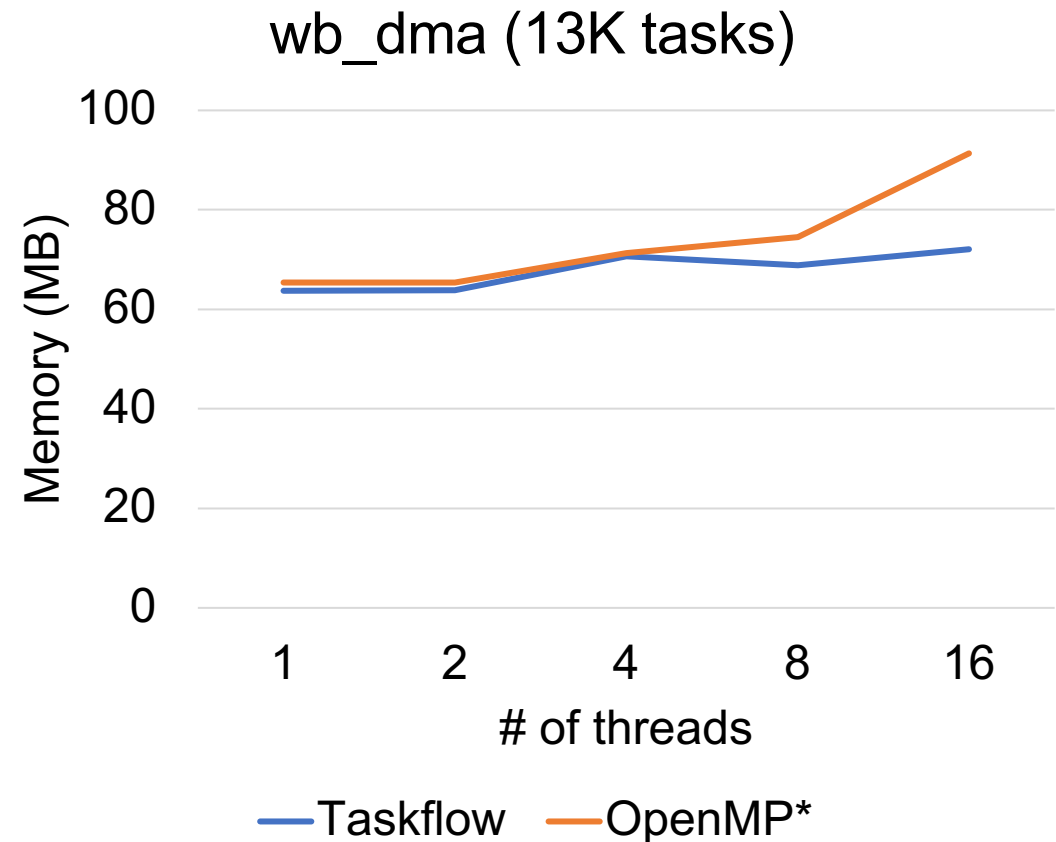
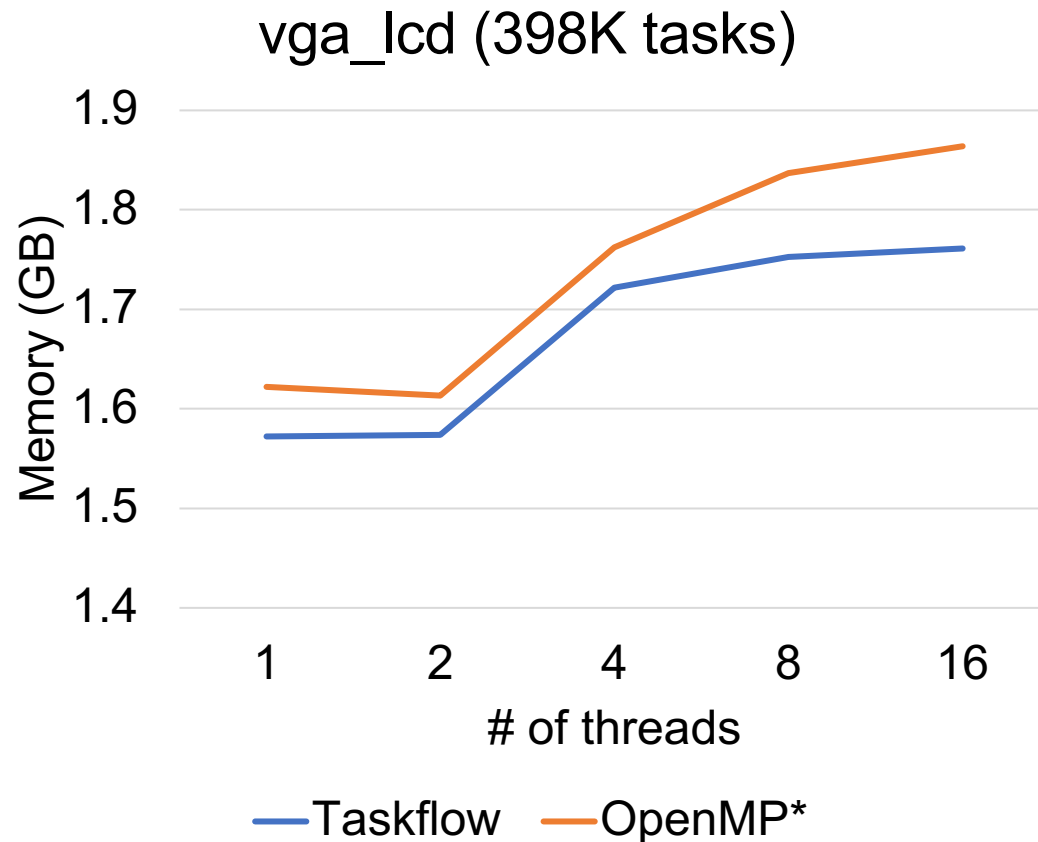
# Runtime Comparison with OpenMP\*

- **OpenMP\*:** Implemented OpenMP's scheduling alg in Taskflow



# Memory Comparison with OpenMP\*

- **OpenMP\*:** Implemented OpenMP's scheduling alg in 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!

- We are always open to collaborate and welcome new users!



Taskflow Contributors: <https://taskflow.github.io/taskflow/contributors.html>

# 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();
```