Dynamic Task Graph Programming (DTGP) in Taskflow

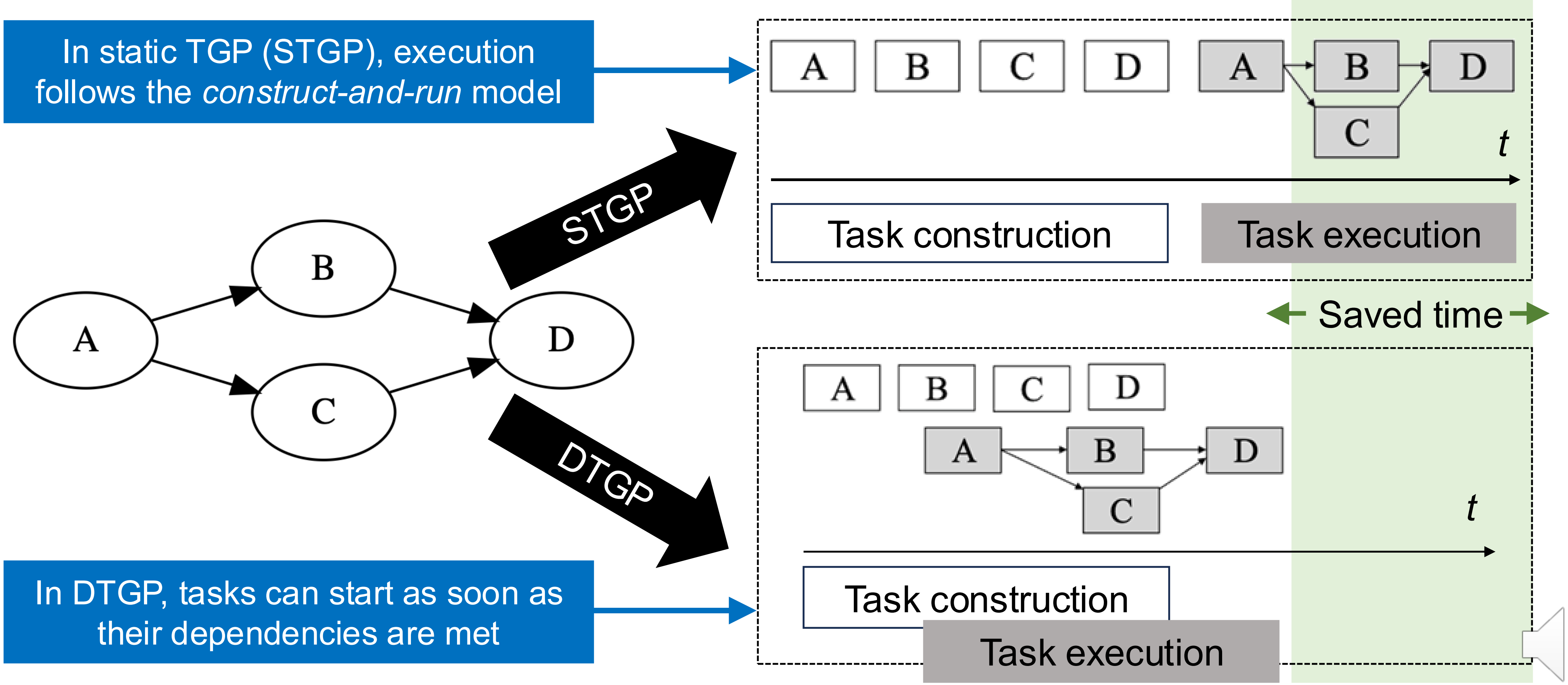
Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://taskflow.github.io/

Takeaways

- Introduce the dynamic task graph programming (DTGP) model in Taskflow
- Recognize the limitations of existing DTGP models
- Overcome the scheduling challenges to support our model
- Conclude the talk

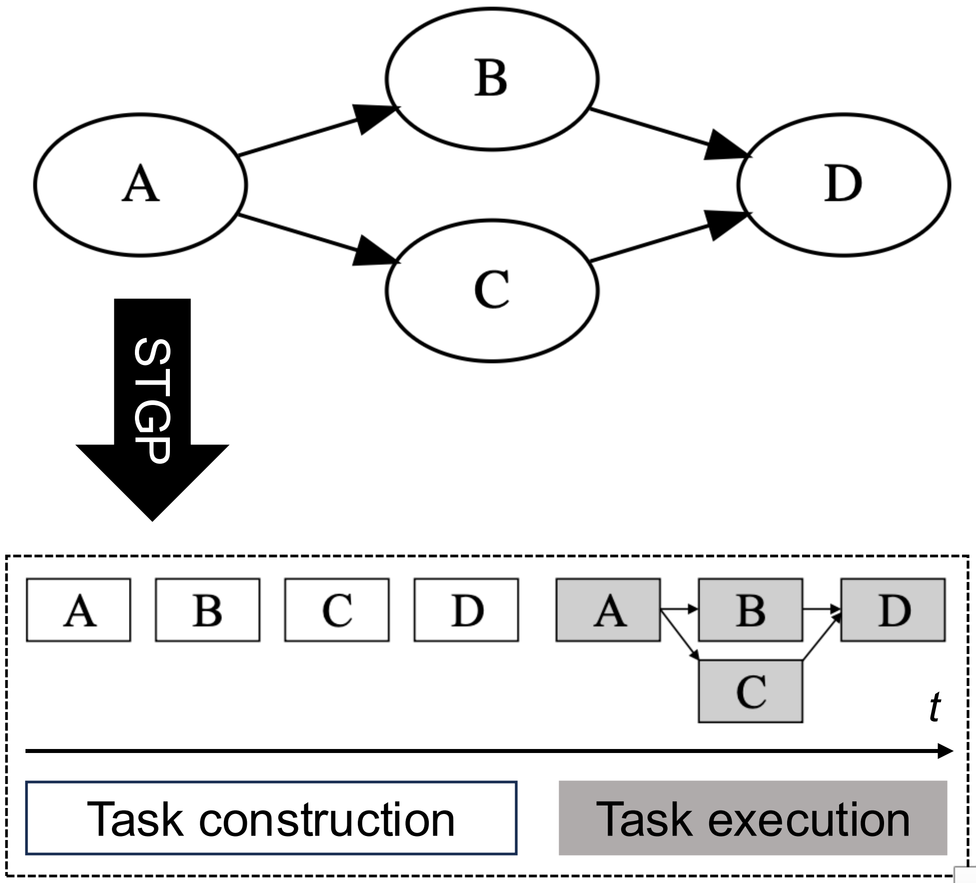
Static vs Dynamic Task Graph Programming (DTGP)

- **Taskflow enables both STGP and DTGP in a unified execution interface**



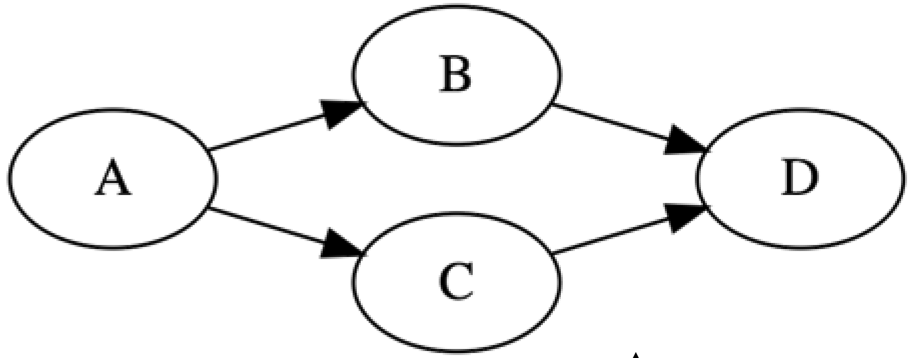
Static Task Graph Programming (STGP) in Taskflow

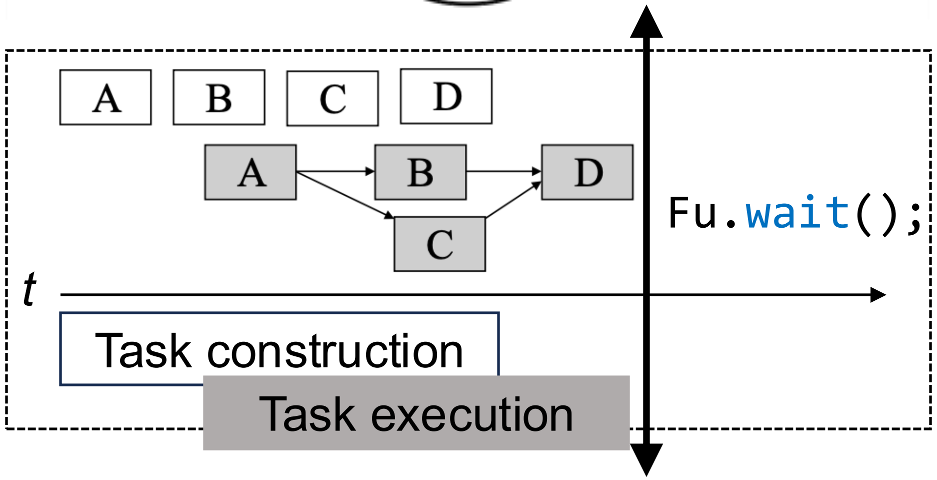
```
#include <taskflow/taskflow.hpp>  // Live: https://godbolt.org/z/j8hx3xnnx

int main(){
  tf::Taskflow taskflow;
  tf::Executor executor;
  auto [A, B, C, D] = taskflow.emplace(
    [] () { std::cout << "TaskA\n"; }
    [] () { std::cout << "TaskB\n"; },
    [] () { std::cout << "TaskC\n"; },
    [] () { std::cout << "TaskD\n"; }
  );
  A.precede(B, C);
  D.succeed(B, C);
  executor.run(taskflow).wait();
  return 0;
}
```



Dynamic Task Graph Programming (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();
```

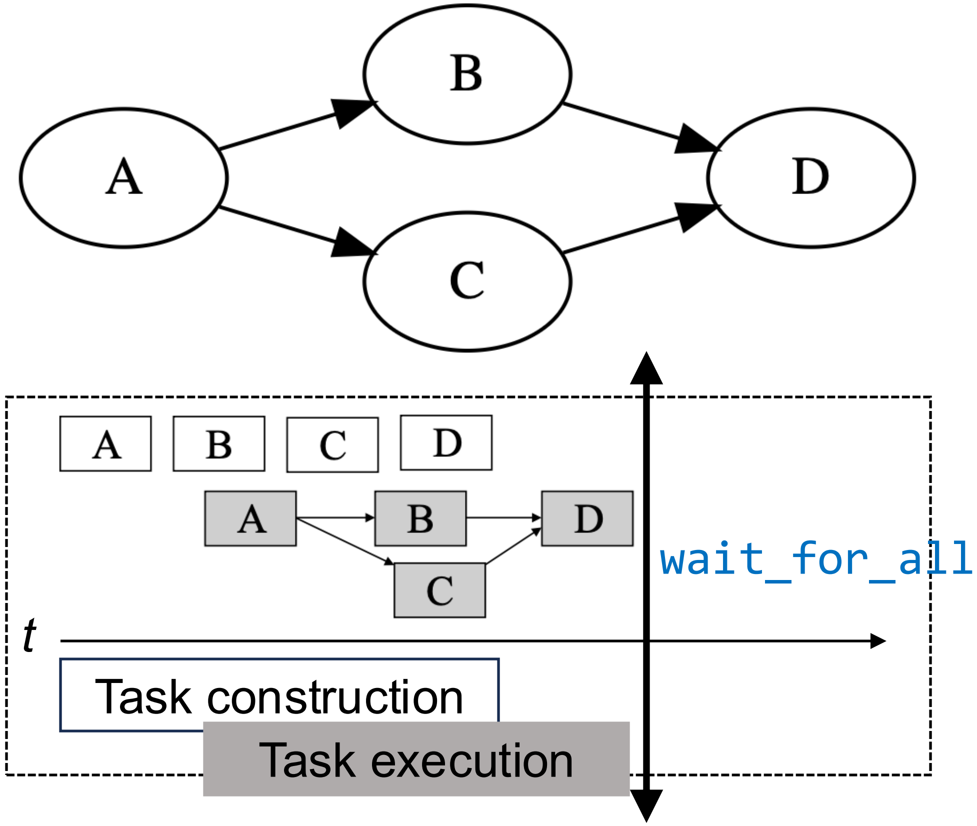




Specify variable task dependencies using C++ variadic parameter pack

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();
```

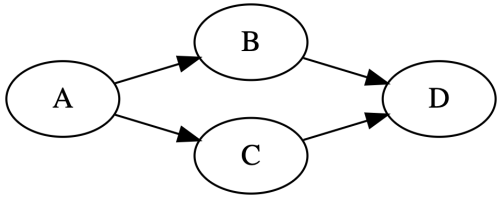


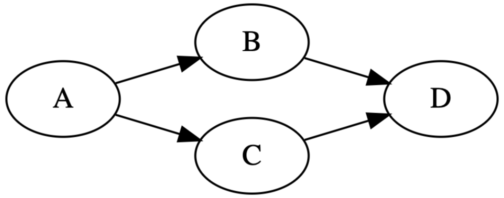
Wait for the entire graph to finish.

Need 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



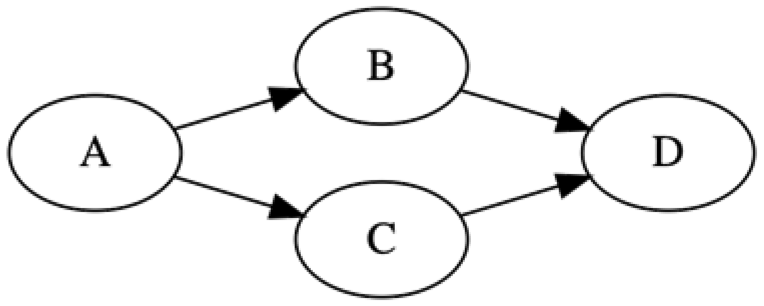


Topological order #2: A→C→B→D

```
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) prevents you from expressing a correct dynamic task graph.

Variable Range of Task Dependencies

- **Both methods can take a variable range of dependent-async tasks**
  - Useful when the task dependencies come as a runtime variable (e.g., loaded from a file)

```
// 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 task that depends on tasks, A, B, C, and D
executor.silent_dependent_async([](){}, tasks.begin(), tasks.end());
```

DTGP is Flexible for Runtime-driven Execution

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

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

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

This type of dynamic task graph is very difficult to achieve using static task graph programming …

We Don't Touch Data Abstraction

- **Focus on coarse-grained task parallelism not fine-grained data parallelism**
  - Our goal is to have users describe tasks and their dependencies in an expressive language

```
template <typename F, typename... Tasks>
auto dependent_async(F&& func, Tasks&&... tasks) {
    ...
}
```

This is how `std::async` is implemented (e.g., `args` are captured with perfect forwarding)

  - Users describe `func` as a lambda and capture necessary data or `func` arguments themselves
- **The advantage of this decision is twofold:**
  - Users retain full control over data layout and ownership, allowing them to optimize data structures and memory layout in their specific application domains
  - Letting users decide how and where to store data keeps our model lightweight and non-intrusive – no need to modify existing data structures to fit our framework
    - Ex: Models that count on data abstraction (e.g., Fastflow, TBB pipeline) require users to rewrite their code to library-specific data abstraction in order to gain parallelism

Takeaways

- Introduce the dynamic task graph programming (DTGP) model in Taskflow
- Recognize the limitations of existing DTGP models
- Overcome the scheduling challenges to support our model
- Conclude the talk

Create an Asynchronous Task using std::async[1]

- **A high-level standard library facility to launch a task asynchronously**

```
#include <future>
#include <iostream>

int compute(int v) {
  return v;
}

int main() {
  std::future<int> fu = std::async(std::launch::async, compute, 42);
  std::cout << fu.get() << std::endl;  // prints 42
}
```

Use `std::async` to asynchronously run the function `compute(42)` on a new thread.

Return a `std::future` to wait for this asynchronous task to finish and access its result (i.e., 42)

[1]: C++ std::async interface: https://en.cppreference.com/w/cpp/thread/async.html

An Example Implementation of `std::async`

```
template <typename F, typename... Args>
auto async(F&& func, Args&&... args) {
  using ReturnType = std::invoke_result_t<F, Args...>;
  // promise-future pair for inter-thread sync
  std::promise<ReturnType> prom;
  std::future<ReturnType> fu = prom.get_future();
  std::thread t([prom=std::move(prom),
    f=std::forward<F>(func), ...args=std::forward<Args>(args)] () mutable {
    if constexpr(std::is_void_v<ReturnType>) {
      f(std::move(args)...);
      prom.set_value();
    } else {
      prom.set_value(f(std::move(args)...));
    }
  });
  t.detach();  // mimic fire-and-forget behavior of std::async
  return fu;
}
```

I promise you that I will run your function, and you can access the result from the future object …

We create a thread from a lambda function object that captures the function and its argument (with perfect forwarding[1]) and invoke the function in the body.

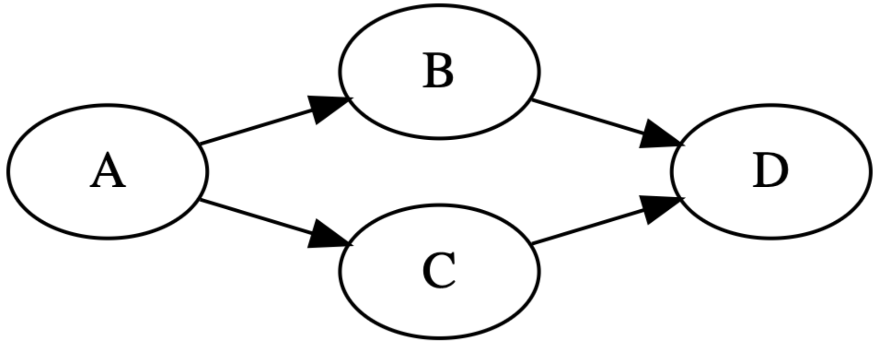
[1]: C++ `std::forward`: https://en.cppreference.com/w/cpp/utility/forward.html

Build a Task Graph w/ `std::async` and `std::future`

- **`std::future` allows us to perform task-specific synchronization**

```
auto A = std::async(std::launch::async,
  [](){ std::cout << "A\n"; }
);
A.wait();
auto B = std::async(std::launch::async,
  [](){ std::cout << "B\n"; }
);
auto C = std::async(std::launch::async,
  [](){ std::cout << "C\n"; }
);
B.wait();
C.wait();
auto D = std::async(std::launch::async,
  [](){ std::cout << "D\n"; }
);
D.wait();
```

We need to wait for A to finish before launching B and C asynchronously.



We need to wait for B and C to finish before launching D asynchronously

By properly synchronizing tasks using `future.wait`, we can dynamically create a task graph (i.e., dynamic task graph)

Sender-Receiver Version (with `std::exec`[1])

- **A standardized abstraction for composing tasks and dependencies**

Schedule tasks on a pool of worker threads

```
exec::static_thread_pool pool;
auto scheduler = pool.get_scheduler();

// create a sender task for A
auto sa = exec::then(exec::schedule(scheduler), []{ std::cout<<"A\n"; });
exec::sync_wait(sa);  // wait for A

// create two parallel sender tasks for B and C
auto sb = exec::then(exec::schedule(scheduler), []{ std::cout<<"B\n"; });
auto sc = exec::then(exec::schedule(scheduler), []{ std::cout<<"C\n"; });
exec::sync_wait(exec::when_all(sb, sc));  // wait for B and C

// create a sender task for D
auto sd = exec::then(exec::schedule(scheduler), []{ std::cout<<"D\n"; });
exec::sync_wait(sd);  // wait for D
```

[1]: C++ execution control library (experimental): https://en.cppreference.com/w/cpp/experimental/execution.html

Intel's TBB Library with `tbb::task_group`[1]

- **A class to create asynchronous tasks and wait for their completion**

```cpp
tbb::task_group tg;

// A
tg.run([] { std::cout << "A\n"; });
tg.wait();

// B and C in parallel
tg.run([] { std::cout << "B\n"; });
tg.run([] { std::cout << "C\n"; });
tg.wait();

// D
tg.run([] { std::cout << "D\n"; });
tg.wait();
```

A class in TBB to create asynchronous tasks and wait for their completion

Need to `task_group::wait` on A before running B and C

Need to `task_group::wait` on B and C before running D

[1]: TBB task group: https://oneapi-spec.uxlfoundation.org/specifications/oneapi/v1.3-rev-1/elements/onetbb/source/task_scheduler/task_group/task_group_cls

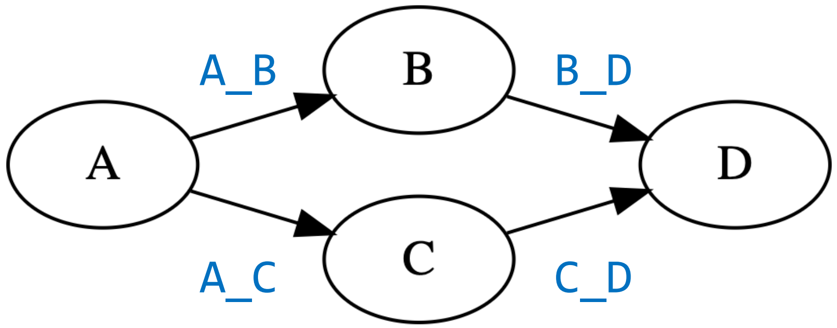
OpenMP Tasking Model with depend Clauses[1]

- **Leverages compiler directives to define tasks and dependencies**

```
#omp parallel
{
  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 << “TaskB\n”;

  #pragma omp task depend(in: B_D, C_D)
  std::cout << “TaskB\n”;
}
```

Define dependency handles

Specify task dependencies using in and out clauses when creating an OpenMP task



With these OpenMP directives, the compiler will insert parallel code that launches asynchronous tasks and enforces their dependencies.

[1]: OpenMP task dependency clauses (version 5, 2008): https://www.openmp.org/spec-html/5.0/openmpsu99.html

OpenCilk Version

- **A fork-join programming model relying on compiler-generated parallel code**
  - With language extensions like `cilk_spawn` and `cilk_sync`

```
void A() { std::cout << "A\n"; }
void B() { std::cout << "B\n"; }
void C() { std::cout << "C\n"; }
void D() { std::cout << "D\n"; }
int main() {
  A();

  cilk_spawn B();
  C();
  cilk_sync;

  D();
}
```

You need a compiler that supports OpenCilk syntax to run this code.

Spawn a child task on B using `cilk_spawn` and continue with C in the main thread

Synchronize both B and C using `cilk_sync` before running task D

[1]: OpenCilk: https://www.opencilk.org/

Limitations of Existing DTGP Models – (1/3)

Tasks and their dependencies are decoupled during task graph creation

- If dependencies are not expressed alongside the task creation logic, it's difficult to reason about the overall task graph structure
- Without a clear dependency structure, the runtime loses opportunities to optimize task placement and load balancing when constructing a task

C++ sender-receiver model

```
// create a sender task for A
auto sa = exec::then(exec::schedule(scheduler), []{ std::cout<<"A\n"; });
exec::sync_wait(sa);  // wait for A

// create two parallel sender tasks for B and C
auto sb = exec::then(exec::schedule(scheduler), []{ std::cout<<"B\n"; });
auto sc = exec::then(exec::schedule(scheduler), []{ std::cout<<"C\n"; });
exec::sync_wait(exec::when_all(sb, sc));

// create a sender task for D
auto sd = exec::then(exec::schedule(scheduler), []{ std::cout<<"D\n"; });
exec::sync_wait(sd);  // wait for D
```

Tasks and their dependencies are decoupled during task graph creation

Limitations of Existing DTGP Models – (2/3)

Correct placement of `wait` calls is left to programmers

- Programmers must determine a correct synchronization order at a fine-grained level
  - In the worst case, the number of `wait` functions equals the number of dependencies
- In practice, many applications only care about the completion of the entire task graph instead of intermediate tasks, making such fine-grained waiting unnecessary, costly, and buggy

TBB model

```
tbb::task_group tg;

tg.run([] { std::cout << "A\n"; });
tg.wait();

tg.run([] { std::cout << "B\n"; });
tg.run([] { std::cout << "C\n"; });
tg.wait();

tg.run([] { std::cout << "D\n"; });
tg.wait();
```

Correct placement of `wait` call is left to programmers

Correct placement of `wait` call is left to programmers

Limitations of Existing DTGP Models – (3/3)

Limited support for building highly dynamic task graphs

- Highly dynamic task graphs → those whose structures, dependencies, and task content are highly dependent on runtime variables or dynamic control-flow results
  - Ex: OpenMP is not a good fit for this scenario as it relies on static compiler directives

May require a non-standard C++ compiler to generate parallel code

OpenMP model

```
#omp parallel
{
  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 << "TaskB\n";

  #pragma omp task depend(in: B_D, C_D)
  std::cout << "TaskB\n";
}
```

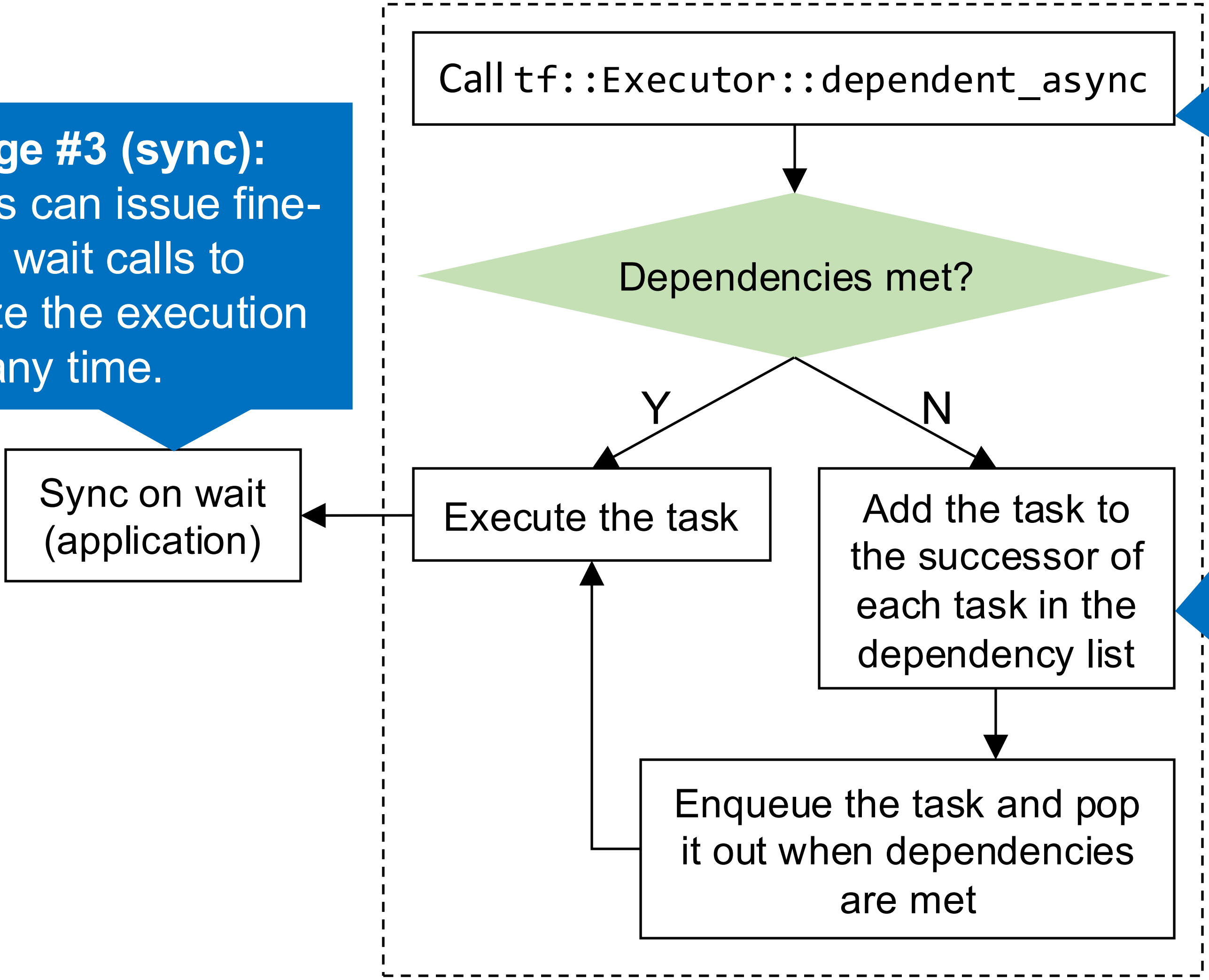
Directive-based models can't handle highly dynamic task graphs that depend on control-flow results ...

Takeaways

- Introduce the dynamic task graph programming (DTGP) model in Taskflow
- Recognize the limitations of existing DTGP models
- Overcome the scheduling challenges to support our model
- Conclude the talk

Scheduling a Dynamic Task Graph

Challenge #3 (sync):
Applications can issue fine-grained wait calls to synchronize the execution at any time.

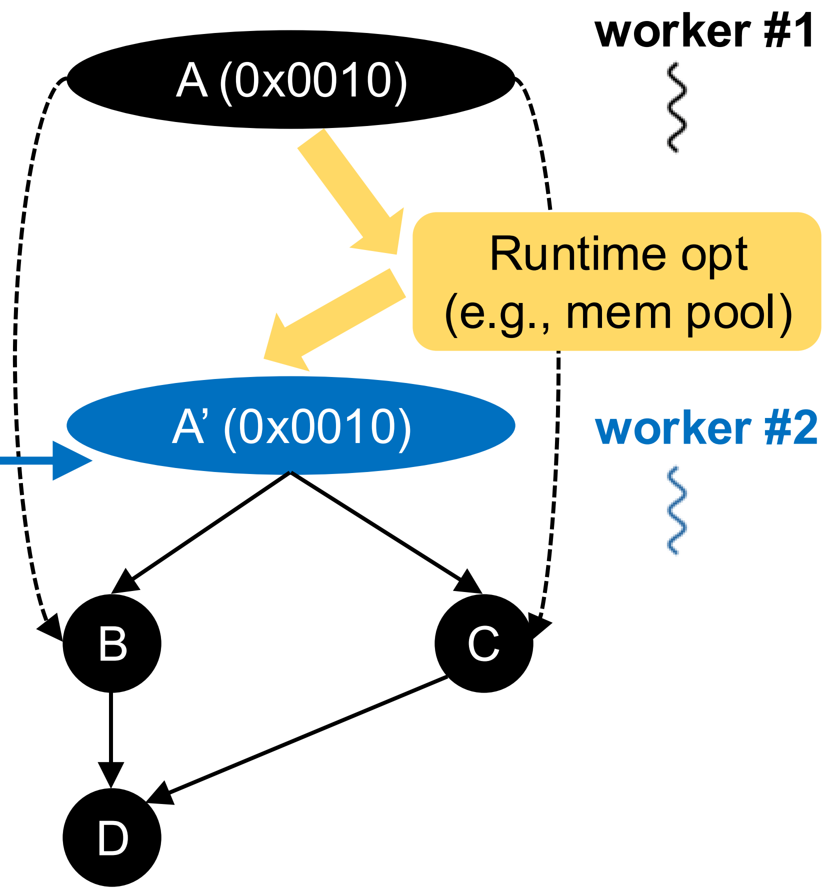


Challenge #1 (ABA):
Dependent async tasks must exist correctly – we cannot specify a task dependency that points to an invalid memory location.

Challenge #2 (race):
Since task graph construction and execution in DTGP can happen simultaneously, multiple workers may concurrently access the successor of a dependent-async task.

Solving Challenge #1: ABA Problem

```
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

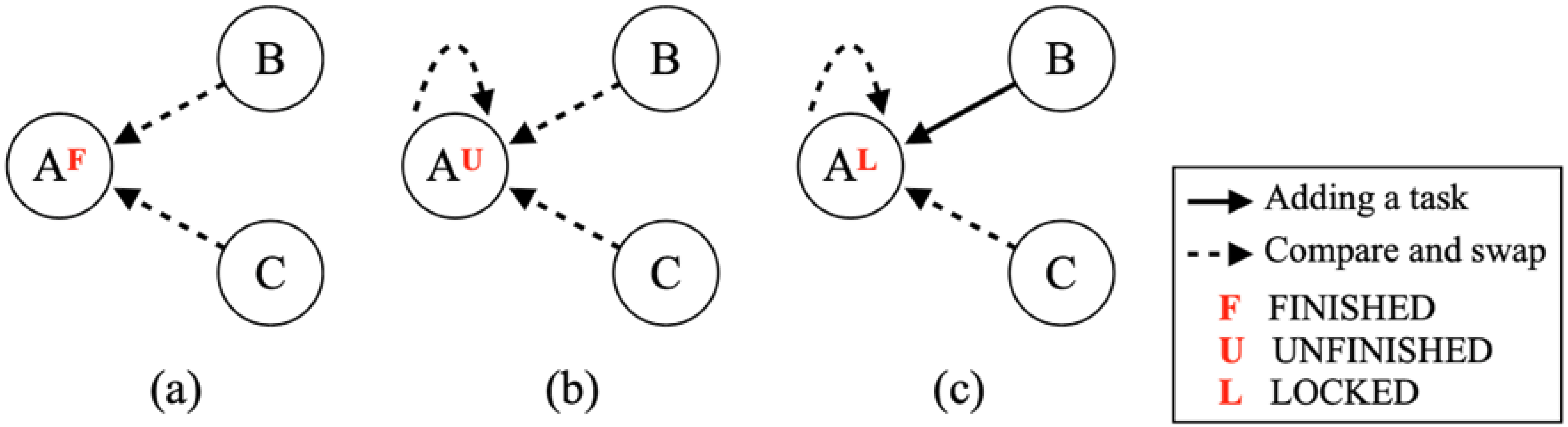
Retain a Shared Ownership of Each Task Needed

```
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**
  - Meanwhile, A may want to remove some of its successor when the task finishes



- **Use compare-and-swap (CAS) with spinning to enable exclusive access**
  - Spinning does not incur much overhead because most task graphs are sparse
  - If you task graph is very dense, probably DTGP is not the right solution to your application

Solving Challenge #3: Synchronization

- **Users can issue both coarse- and fine-grained synchronizations at any time**
  - Coarse-grained sync: `executor.wait_for_all()`
  - Fine-grained sync: `future.wait()`

```
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

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

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

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

We leverage C++20 atomic variables to perform waiting/notifying operations[1], which allow much of the synchronization to occur in user space rather than in the kernel space (~11% performance improvement).

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

Our Scheduling Algorithm is Lock-free[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*, 2023

Takeaways

- Introduce the dynamic task graph programming (DTGP) model in Taskflow
- Recognize the limitations of existing DTGP models
- Overcome the scheduling challenges to support our model
- Conclude the talk

Question?

Static Task Graph Programming (STGP)

```
// Live: https://godbolt.org/z/j8hx3xnnx

tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = taskflow.emplace(
  [](){ std::cout << "TaskA\n"; }
  [](){ std::cout << "TaskB\n"; },
  [](){ std::cout << "TaskC\n"; },
  [](){ std::cout << "TaskD\n"; }
);

A.precede(B, C);
D.succeed(B, C);
executor.run(taskflow).wait();
```

Taskflow: https://taskflow.github.io

Dynamic Task Graph Programming (DTGP)

```
// Live: https://godbolt.org/z/T87PrTarx

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