Executor

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

Takeaways

- **Learn the basic usage of the Taskflow executor**
- **Understand cooperative execution within the executor**
- **Observe and customize worker behavior in the executor**
- **Conclude the talk**

Create an Executor

- **A `tf::Executor` manages a set of worker threads to run submitted tasks**
  - Implements a work-stealing algorithm to achieve dynamic load balancing[1]
  - Implements a notification algorithm to adapt worker availability to dynamic task parallelism
- **Constructor of `tf::Executor` takes an unsigned integer to spawn `N` workers**

```
tf::Executor executor1(4);   // create an executor of four worker threads

tf::Executor executor2;      // create an executor with the number of workers equal
                             // to std::thread::hardware_concurrency
```

Creating an executor is not free — it involves overhead like spawning threads, setting up work-stealing data structures, etc. Unless you really need multiple executors, the best practice is to create just one executor in your application and reuse it as much as possible.

```
inline tf::Executor global_executor;
```

[1]: Tsung-Wei Huang, et. al, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," *IEEE TPDS*, 2022

Submit a Taskflow to an Executor

- **Execution methods in the executor are either blocking or non-blocking**
  - Blocking methods wait for all tasks to finish before returning
  - Non-blocking methods return immediately

`tf::Future` is derived from `std::future` with a few Taskflow-specific operations added (e.g., cancellation).

```
tf::Taskflow taskflow1, taskflow2, taskflow3;
tf::Executor executor;

// use run methods to submit a taskflow for execution
tf::Future<void> future1 = executor.run(taskflow1);       // run once
tf::Future<void> future2 = executor.run_n(taskflow2, 10); // run multiple times
tf::Future<void> future3 = executor.run_until(            // run repeatedly until
  taskflow3, [i=0]() mutable { return i++>5; }            // the condition is met
);

// synchronize the execution
future1.wait();           // block until taskflow1 finishes
executor.wait_for_all(); // block until all submitted tasks finish
```

Request Cancellation of a Submitted Execution

- **You can call `tf::Future::cancel` to stop a running taskflow**

```
tf::Executor executor;
tf::Taskflow taskflow;

for(int i=0; i<1000; i++) {
  taskflow.emplace([](){ std::this_thread::sleep_for(std::chrono::seconds(1)); });
}

// submit the taskflow
tf::Future<void> fu = executor.run(taskflow);

// request to cancel the above submitted execution
fu.cancel();

// wait until the cancellation completes
fu.wait();
```

Requesting a cancellation does not guarantee that a running taskflow will stop immediately. Depending on the scheduler, the cancellation may or may not take effect.

Execution Order

- **Each call to the run function is an independent submission called *topology***
  - A topology is an internal data structure that the executor creates to manage a submission
  - A submission can be a single `run`, multiple runs (`run_n`), or a `run_until` execution

```
tf::Future<void> future1 = executor.run(taskflow1);        // one submission
tf::Future<void> future2 = executor.run_n(taskflow1, 10); // one submission
tf::Future<void> future3 = executor.run_until(            // one submission
  taskflow1, [i=0]() mutable { return i++>5; }
);

tf::Future<void> future4 = executor.run(taskflow2);        // one submission
```

- **Multiple submissions of the same taskflow are executed sequentially**
  - In the example above, `future1` becomes ready before `future2`, then `future3`, and so on
- **Multiple submissions of different taskflows may be executed out of order**
  - In the example above, `futurue4` may become ready before `future1`, `future2`, or `future3`

Executor Doesn’t Assume any Ownership of a Graph

- **Every task belongs to a graph at a time and remains alive with that graph**
  - As long as a taskflow remains alive, all of its associated tasks remain alive
- **The lifetime of a task affects its callable, particularly the captured variables**
  - When a taskflow is destroyed, all of its tasks and their captured variables are also destroyed

❌

```
tf::Executor executor;
{
  tf::Taskflow taskflow;
  taskflow.emplace(
    [](){ std::cout << "Task A\n"; },
    [](){ std::cout << "Task B\n"; },
    [](){ std::cout << "Task C\n"; },
    [](){ std::cout << "Task D\n"; }
  );
  executor.run(taskflow);
}  // taskflow is destroyed after the block
executor.wait_for_all();
```

✓

```
tf::Executor executor;
tf::Taskflow taskflow;
taskflow.emplace(
  [](){ std::cout << "Task A\n"; },
  [](){ std::cout << "Task B\n"; },
  [](){ std::cout << "Task C\n"; },
  [](){ std::cout << "Task D\n"; }
);
executor.run(taskflow);
executor.wait_for_all();
```

It is your responsibility to keep relevant taskflows alive during their execution.

Never Modify a Graph During its Execution

- **Modifying a graph during its execution can result in undefined behavior**
  - Ex: data race, program crash, unpredictable results, etc.

✗

```
tf::Executor executor;
tf::Taskflow taskflow;
taskflow.emplace(
  [](){ std::cout << “Task A\n”; },
  [](){ std::cout << “Task B\n”; },
  [](){ std::cout << “Task C\n”; },
  [](){ std::cout << “Task D\n”; }
);
auto fu = executor.run(taskflow);
taskflow.emplace([](){});
```

The graph may still be running while it is being modified.

✓

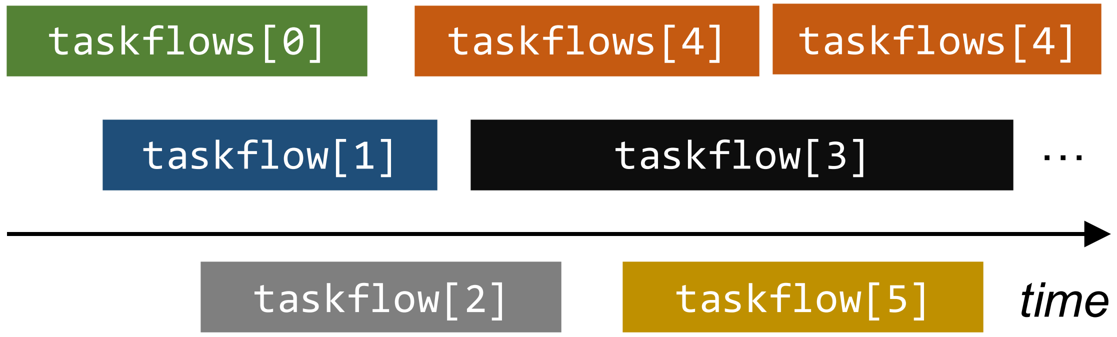
```
tf::Executor executor;
tf::Taskflow taskflow;
taskflow.emplace(
  [](){ std::cout << “Task A\n”; },
  [](){ std::cout << “Task B\n”; },
  [](){ std::cout << “Task C\n”; },
  [](){ std::cout << “Task D\n”; }
);
executor.run(taskflow).wait();
taskflow.emplace([](){});
```

After “`wait`”, the graph finishes its execution and can be safely modified.

Execution Methods are Thread-safe

- **It is completely safe to issue multiple submissions from different threads**
  - These threads can be either internal to the executor or external to the executor (e.g., a freelancing thread)
  - Note that the execution order of *different* taskflows submitted by threads is non-deterministic, while multiple submissions of the same taskflows are still executed sequentially

```
tf::Executor executor;
std::array<tf::Taskflow, 10> taskflows;

// thread i submits a random taskflow
for(int i=0; i<100; ++i) {
  std::jthread([&](){
    auto t = rand()%10;
    executor.run(taskflows[t]);
  });
}
executor.wait_for_all();
```



There is no particular order in which a taskflow will start or finish after submission, except for repeated submissions of the same taskflow which will execute sequentially (e.g., `taskflow[4]`).

Query the Worker ID from an Executor

- **Each of the `N` workers in an executor is assigned an integer ID in `[0, N)`**
  - You can query the ID of the calling thread by `tf::Executor::this_worker_id`
  - If the calling thread is not a worker of the executor, `-1` will be returned
- **Worker ID is handy for establishing worker-specific data structures**
  - Ex: worker-local states or caches that avoid going through a globally shared global storage which typically incurs centralized synchronization overhead

```
std::vector<WorkerData> worker_data[8];   // worker-specific data vector
tf::Taskflow taskflow;
tf::Executor executor(8);                 // an executor of eight workers
assert(executor.this_worker_id() == -1);  // calling thread is not a worker of the
                                          // executor, so -1 is returned
taskflow.emplace([&](){
  int w = executor.this_worker_id();      // w is in the range of [0, 8)
  auto& vec = worker_data[w];             // worker w takes work-specific data at
  ...                                     // worker_data[w]
});
```

Takeaways

- Learn the basic usage of the Taskflow executor
- Understand cooperative execution within the executor
- Observe and customize worker behavior in the executor
- Conclude the talk

Cooperative Execution

- **Blocks the calling worker until a condition happens while allowing progress**
  - "Allowing progress" means the worker keeps doing useful work through work stealing instead of sitting idle (e.g., sleeping, preempted by the OS), while waiting for the condition to happen
  - Cooperative execution is primarily used to avoid deadlock caused by idling workers
- **Executor supports two major cooperative execution methods:**
  - `tf::Executor::corun` and `tf::Executor::corun_until`

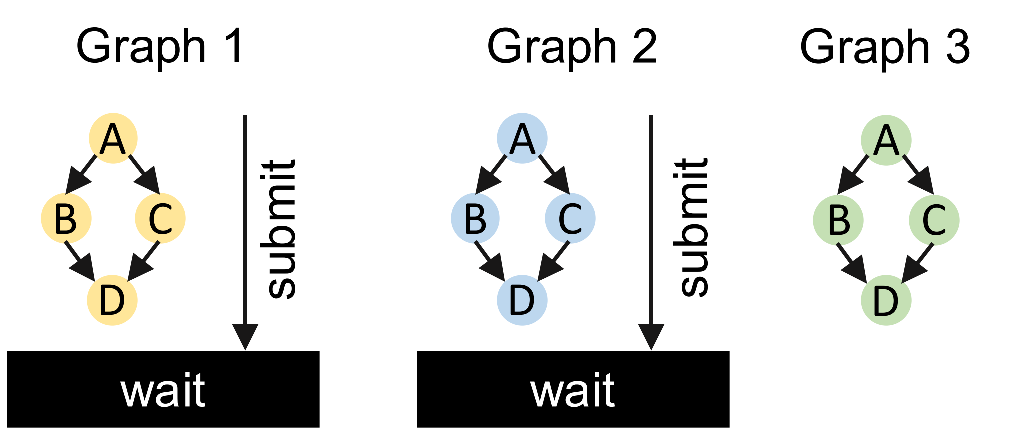
```
tf::Executor executor(2);
tf::Taskflow taskflow;
std::array<tf::Taskflow, 1000> others;
// non-cooperative execution: deadlock
for(size_t n=0; n<1000; n++) {
  taskflow.emplace([&, &tf=others[n]]{
    executor.run(tf).wait();
  });
}
executor.run(taskflow).wait();
```

May incur deadlock w/o cooperative execution!

```
tf::Executor executor(2);
tf::Taskflow taskflow;
std::array<tf::Taskflow, 1000> others;
// cooperative execution: no deadlock
for(size_t n=0; n<1000; n++) {
  taskflow.emplace([&, &tf=others[n]]{
    executor.corun(tf);
  });
}
executor.run(taskflow).wait();
```

No deadlock with cooperative execution!

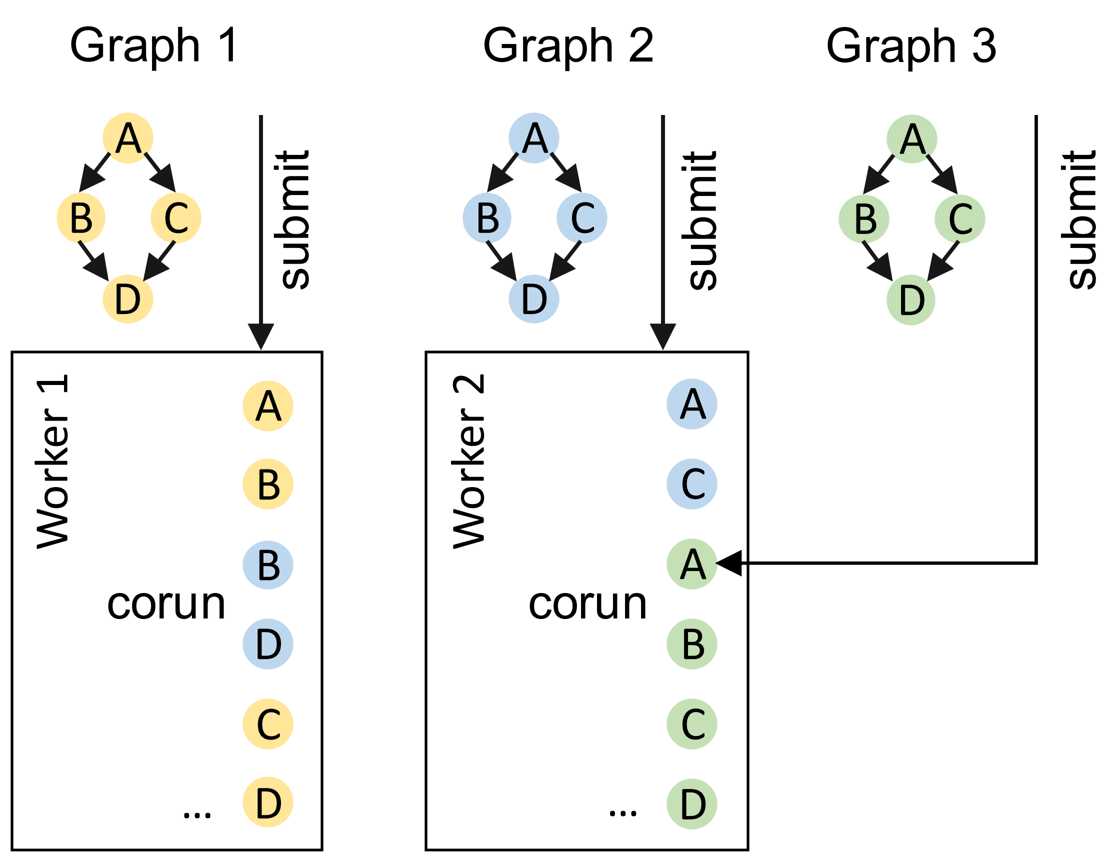
Illustration of Cooperative Execution



After submitting a taskflow, the worker is blocked at the "`wait`" call and makes no further progress. Eventually, no workers are available in the executor …

```
for(size_t n=0; n<1000; n++) {
  taskflow.emplace([&, &tf=others[n]]{
    executor.run(tf).wait();
  });
}
```

Calling worker is blocked



With `corun`, the worker waits for the taskflow to finish but still cooperatively helps execute tasks with other available workers.

Cooperative Execution with Custom Condition

- **Similar to `corun`, `corun_until` provides a more generalized interface**
  - Blocks the calling until the given condition is met, while allowing the worker to continue executing tasks through its work-stealing loop
  - Allows you to specify an arbitrary condition that determines when to stop cooperative execution — *useful for implementing custom synchronization or polling strategies*

```
taskflow.emplace([&](){
  auto fu = std::async([](){ std::sleep(100s); });
  executor.corun_until([](){
    return fu.wait_for(std::chrono::seconds(0)) == future_status::ready;
  });
});
```

Blocks the calling worker with cooperative execution (i.e., continue to participate in the work-stealing loop) until the future state is ready.

```
taskflow.emplace([&](){
  executor.corun_until([](){ other_custom_condition(); });
});
```

Takeaways

- Learn the basic usage of the Taskflow executor
- Understand cooperative execution within the executor
- Observe and customize worker behavior in the executor
- Conclude the talk

Observe Worker Activities

- **`ObserverInterface` provides a set of virtual methods to override**
  - Ex: Defines callbacks for setting up the observer, recording before and after a task finishes

```
class ObserverInterface {
  virtual ~ObserverInterface() = default;
  // called by the executor once when the observer is constructed
  virtual void set_up(size_t num_workers) = 0;
  // called by the executor before a worker begins to execute a task
  virtual void on_entry(tf::WorkerView worker_view, tf::TaskView task_view) = 0;
  // called by the executor after a worker finishes executing a task
  virtual void on_exit(tf::WorkerView worker_view, tf::TaskView task_view) = 0;
};
```

- **`WorkerView` and `TaskView` provide immutable access to workers and tasks**
  - Ex: query the worker's ID, query the task's name, etc.
  - Prevents users from accidentally modifying the underlying worker or task, which could lead to undefined behavior in scheduling

Example (1/3): Derive a Custom Observer

```
struct MyObserver : public tf::ObserverInterface {
  MyObserver(const std::string& name) {
    std::cout << "constructing observer " << name << '\n';
  }
  // called by the executor once when the observer is constructed
  void set_up(size_t num_workers) override final {
    std::cout << "setting up an observer with " << num_workers << " workers\n";
  }
  // called by the executor before a worker begins to execute a task
  void on_entry(tf::WorkerView w, tf::TaskView tv) override final {
    std::cout << std::format("worker {} ready to run {}\n", w.id(), tv.name());
  }
  // called by the executor after a worker finishes executing a task
  void on_exit(tf::WorkerView w, tf::TaskView tv) override final {
    std::cout << std::format("worker {} ready to run {}\n", w.id(), tv.name());
  }
};
```

Example (2/3): Create a Custom Observer

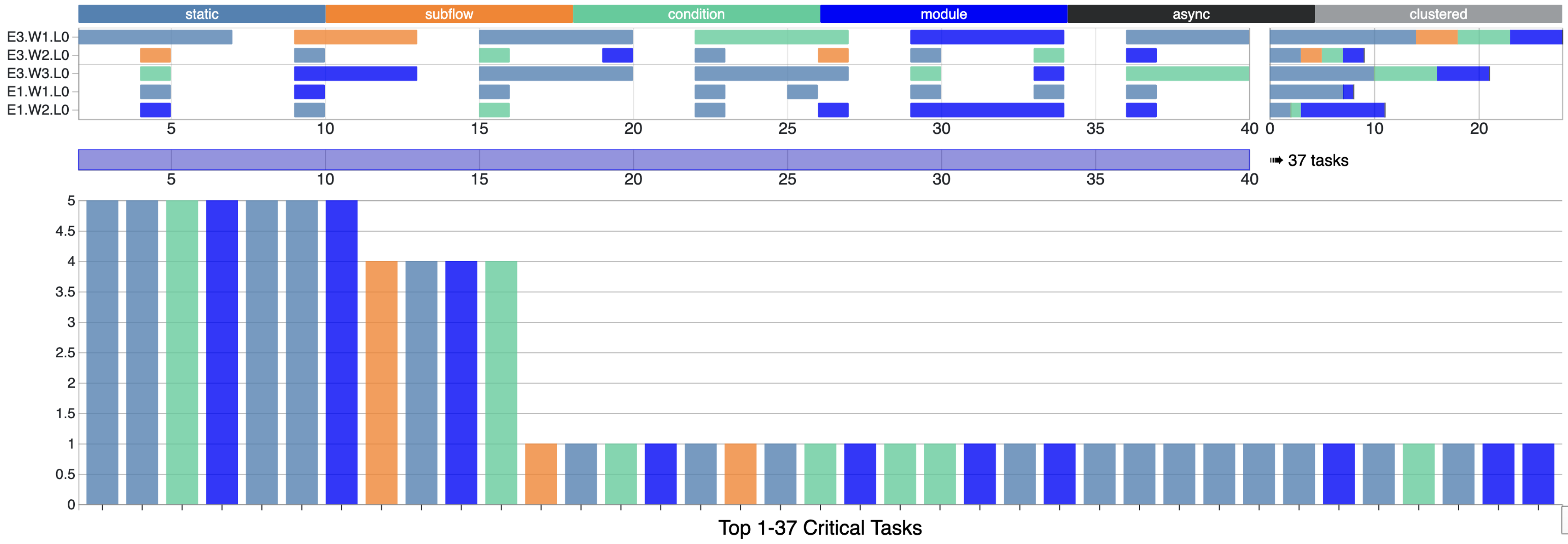
```
tf::Executor executor(4);  // create an executor of 4 workers (IDs={0, 1, 2, 3})
tf::Taskflow taskflow;
auto A = taskflow.emplace([](){ std::cout << “A\n”; }).name(“A”);
auto B = taskflow.emplace([](){ std::cout << “B\n”; }).name(“B”);
auto C = taskflow.emplace([](){ std::cout << “C\n”; }).name(“C”);
auto D = taskflow.emplace([](){ std::cout << “D\n”; }).name(“D”);
auto E = taskflow.emplace([](){ std::cout << “E\n”; }).name(“E”);
auto F = taskflow.emplace([](){ std::cout << “F\n”; }).name(“F”);
auto G = taskflow.emplace([](){ std::cout << “G\n”; }).name(“G”);
auto H = taskflow.emplace([](){ std::cout << “H\n”; }).name(“H”);

// you can also remove an observer if you no longer need it
std::shared_ptr<MyObserver> observer = executor.make_observer<MyObserver>(“obs”);
executor.run(taskflow).wait();

// you can also remove an observer if you no longer need it
executor.remove_observer(std::move(observer));
```

Example (3/3): Sample Output

```
constructing observer obs
setting up an observer with 4 workers
worker 2 ready to run A
A
worker 2 finished running A
worker 2 ready to run B
B
worker 1 ready to run C
worker 2 finished running B
C
…
D
E
worker 1 ready to run F
worker 2 finished running D
…
```

Default Observer: TFProf

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



Default Observer: TFProf (cont'd)

```
# enable the environment variable without any value
~$ TF_ENABLE_PROFILER=""    ./my_taskflow_program

# your program output
...
...
...
# Taskflow profile summary
==Observer 0: 1 workers completed 18 tasks in 28 us
    -Task-   Count  Time (us)   Avg (us)  Min (us)  Max (us)
    static       7          5   0.714286         0         4
 condition      11          0   0.000000         0         0

  -Worker-   Level      Task  Count  Time (us)  Avg (us)  Min (us)  Max (us)
        14       0    static      7          5  0.714286         0         4
                   condition     11          0  0.000000         0         0
                                 18          5  0.277778         0         4
```

The summary reports for each task type the number of executions (Count), the total execution time (Time), average execution time per task (Avg), and the minimum (Min) and the maximum (Max) execution time among all tasks.

Configure Worker Property

- **`WorkerInterface` provides a set of callbacks to modify worker property**

```
class WorkerInterface {                                    // base class to derive from
  virtual ~WorkerInterface() = default;
  void scheduler_prologue(Worker& worker) = 0;
  void scheduler_epilogue(Worker& worker, std::exception_ptr ptr) = 0;
};
```

  - Ex: Creates an executor of four workers and configures each worker's behavior using `tf::make_worker_interface`, which instantiates `CustomWorkerBehavior` class derived from `tf::WorkerInterface`

```
tf::Executor executor(4, tf::make_worker_interface<CustomWorkerBehavior>());
```

  - Most applications leverage `WorkerInterface` to affine a worker to a specific CPU core to improve cache locality and reduce context-switch overhead

- **Both prologue and epilogue functions can be invoked simultaneously**
  - It is user's responsibility to ensure no data race can occur during their executions

Execution Flow of `WorkerInterface`

```
for(size_t n=0; n<num_workers; n++) {
  create_thread([](Worker& worker) {
    worker_interface->scheduler_prologue(worker);
    try {
      // enter the work-stealing scheduling loop
      while(1) {
        auto stop = perform_work_stealing_algorithm();
        if(stop) {
          break;
        }
      }
    } catch(...) {
      exception_ptr = std::current_exception();
    }
    worker_interface->scheduler_epilogue(worker, exception_ptr);
  });
}
```

This is the hook that lets users customize what happens *right before* the worker enters the scheduling loop (i.e., work-stealing loop).

This is the cleanup hook. It lets users perform post-scheduling work *right after* a worker leaves the scheduling loop.

Takeaways

- Learn the basic usage of the Taskflow executor
- Understand cooperative execution within the executor
- Observe and customize worker behavior in the executor
- Conclude the talk

Question?

Static Task Graph Programming (STGP)

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

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

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

Dynamic Task Graph Programming (DTGP)

```cpp
// 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();
```

Taskflow: https://taskflow.github.io