Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++

Tsung-Wei Huang, C.-X. Lin, G. Guo, and M. Wong
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, IL, USA

Cpp-Taskflow's Project Mantra

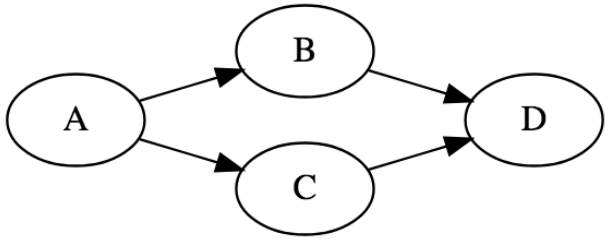
*A programming library helps developers **quickly** write **efficient** parallel programs on a **shared-memory** architecture using **task-based** approaches in **modern C++***

- **Task-based approach scales best with multicore arch**
  - We should write tasks instead of threads
  - Not trivial due to dependencies (race, lock, bugs, etc)
- **We want developers to write parallel code that is:**
  - Simple, expressive, and transparent
- **We don't want developers to manage:**
  - Explicit thread management
  - Difficult concurrency controls and daunting class objects

Hello-World in Cpp-Taskflow

```
#include <taskflow/taskflow.hpp>   // Cpp-Taskflow is header-only
int main(){
   tf::Taskflow tf;
   auto [A, B, C, D] = tf.emplace(
      [] () { std::cout << "TaskA\n"; }
      [] () { std::cout << "TaskB\n"; },
      [] () { std::cout << "TaskC\n"; },
      [] () { std::cout << "TaskD\n"; }
   );
   A.precede(B);            // A runs before B
   A.precede(C);            // A runs before C
   B.precede(D);            // B runs before D
   C.precede(D);            // C runs before D
   tf::Executor().run(tf);  // create an executor to run the taskflow
   return 0;
}
```

*Only **15 lines** of code to get a parallel task execution!*



Hello-World in OpenMP

```
#include <omp.h>  // OpenMP is a lang ext to describe parallelism in compiler directives
int main(){
   #omp parallel num_threads(std::thread::hardware_concurrency())
   {
      int A_B, A_C, B_D, C_D;
      #pragma omp task depend(out: A_B, A_C)
      {
         s t d : : c o u t << ”TaskA\n” ;
      }
      #pragma omp task depend(in: A_B; out: B_D)
      {
         s t d : : c o u t << ” TaskB\n” ;
      }
      #pragma omp task depend(in: A_C; out: C_D)
      {
         s t d : : c o u t << ” TaskC\n” ;
      }
      #pragma omp task depend(in: B_D, C_D)
      {
         s t d : : c o u t << ”TaskD\n” ;
      }
   }
   return 0;
}
```

Task dependency clauses

Task dependency clauses

Task dependency clauses

Task dependency clauses

OpenMP task clauses are static and explicit; Programmers are responsible a proper order of writing tasks consistent with sequential execution

Hello-World in Intel's TBB Library

```
#include <tbb.h>  // Intel's TBB is a general-purpose parallel programming library in C++
int main(){
    using namespace tbb;
    using namespace tbb:flow;
    int n = task_scheduler init::default_num_threads () ;
    task scheduler_init init(n);
    graph g;
    continue_node<continue_msg> A(g, [] (const continue msg &) {
        s t d : : c o u t << "TaskA" ;
    }) ;
    continue_node<continue_msg> B(g, [] (const continue msg &) {
        s t d : : c o u t << "TaskB" ;
    }) ;
    continue_node<continue_msg> C(g, [] (const continue msg &) {
        s t d : : c o u t << "TaskC" ;
    }) ;
    continue_node<continue_msg> C(g, [] (const continue msg &) {
        s t d : : c o u t << "TaskD" ;
    }) ;
    make_edge(A, B);
    make_edge(A, C);
    make_edge(B, D);
    make_edge(C, D);
    A.try_put(continue_msg());
    g.wait_for_all();
}
```

Use TBB's FlowGraph for task parallelism

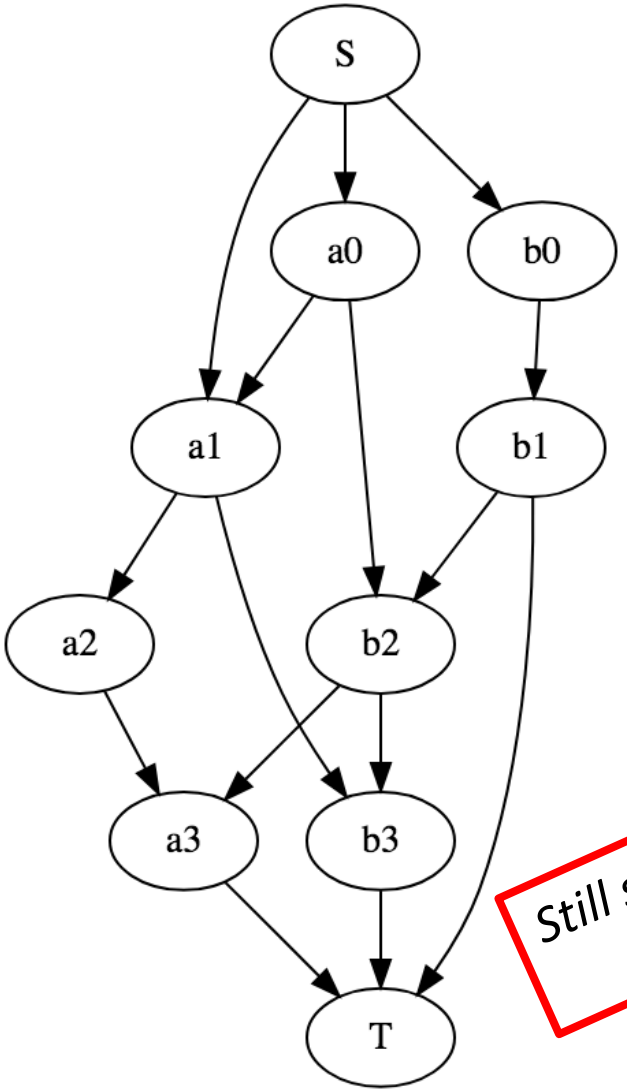
Declare a task as a continue_node

TBB has excellent performance in generic parallel computing. Its drawback is mostly in the ease-of-use standpoint (simplicity, expressivity, and programmability).

Somehow, this looks more like "hello universe" …

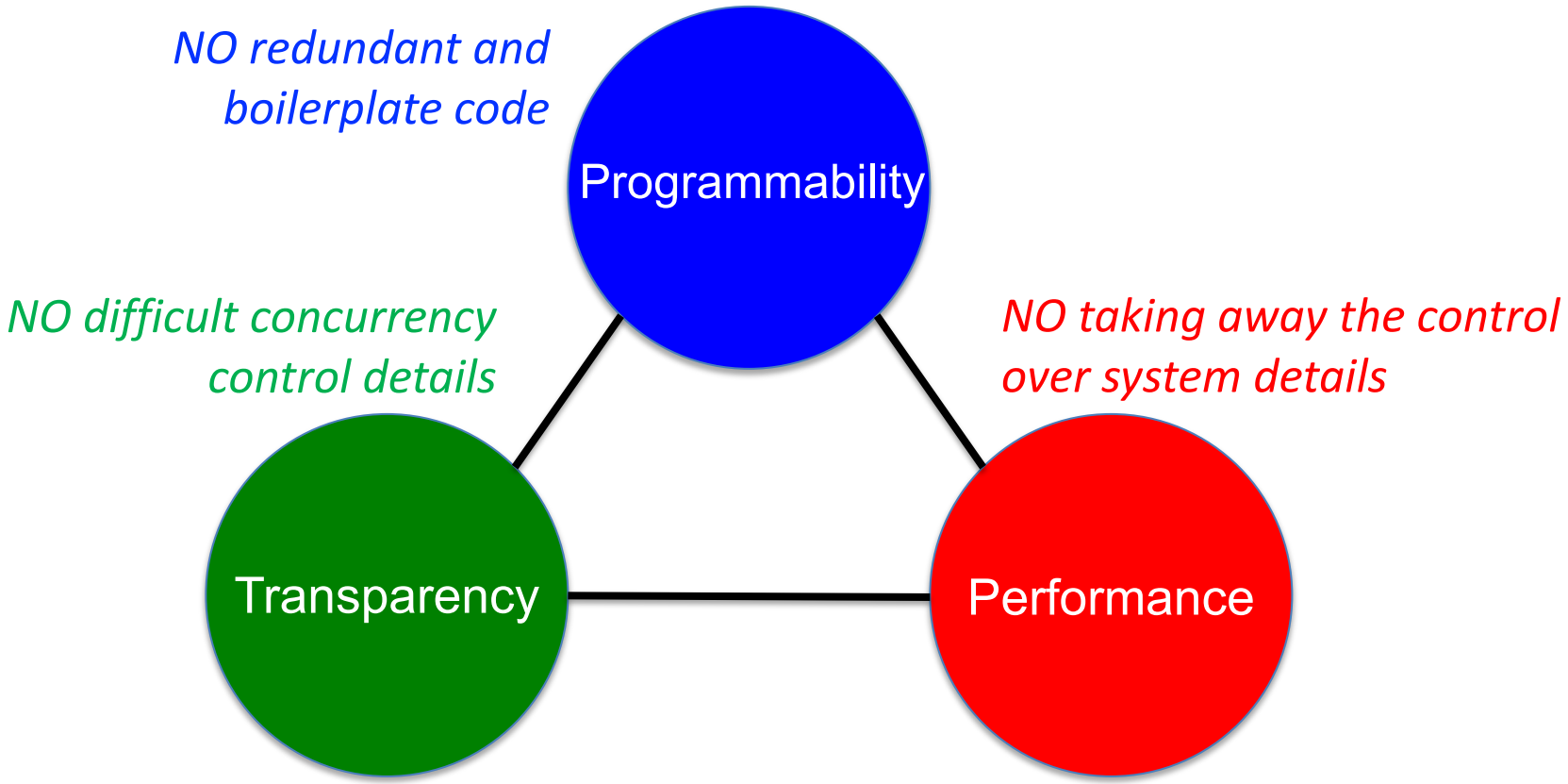
A Slightly More Complicated Example

```
// source dependencies
S.precede(a0);      // S runs before a0
S.precede(b0);      // S runs before b0
S.precede(a1);      // S runs before a1
// a_ -> others
a0.precede(a1);     // a0 runs before a1
a0.precede(b2);     // a0 runs before b2
a1.precede(a2);     // a1 runs before a2
a1.precede(b3);     // a1 runs before b3
a2.precede(a3);     // a2 runs before a3
// b_ -> others
b0.precede(b1);     // b0 runs before b1
b1.precede(b2);     // b1 runs before b2
b2.precede(b3);     // b2 runs before b3
b2.precede(a3);     // b2 runs before a3
// target dependencies
a3.precede(T);      // a3 runs before T
b1.precede(T);      // b1 runs before T
b3.precede(T);      // b3 runs before T
```



Still simple in Cpp-Taskflow

Our Goal of Parallel Task Programming

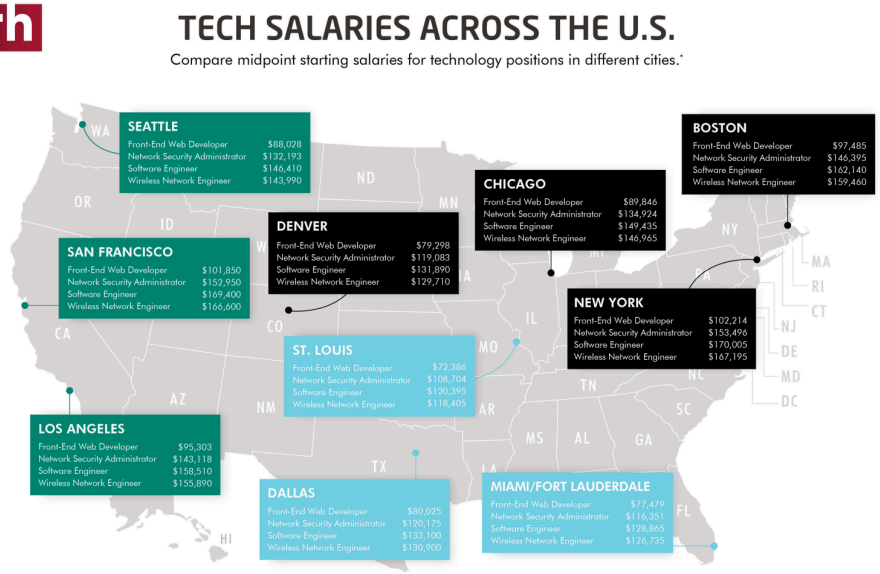


"We want to let users easily express their parallel computing workload without taking away the control over system details to achieve high performance, using our expressive API in modern C++"

Keep Programmability in Mind

- **In the cloud era …**
  - Hardware is just a commodity
  - Building a cluster is cheap
  - Coding takes people and time

Google Cloud

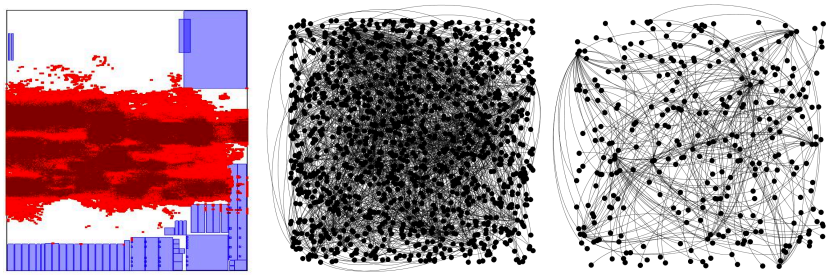


Programmability can affect the performance and productivity in many aspects (details, styles, high-level decisions, etc.)!

2018 Avg Software Engineer salary (NY) > $170K

Why Task Parallelism?

- **Project Motivation: Large-scale VLSI timing analysis**
  - Extremely large and complex task dependencies
  - Irregular compute patterns
  - Incremental and dynamic control flows
- **Existing solutions (including OpenTimer*)**
  - Based on OpenMP mostly
  - Loop-based parallelism
  - Specialized data structures
- **Need task-based approach**
  - Flow computations naturally with the graph structure
  - Tasks and dependencies are just the timing graph



(a) Circuit (1.01mm^2) (b) Graph (3M gates) (c) A signal path

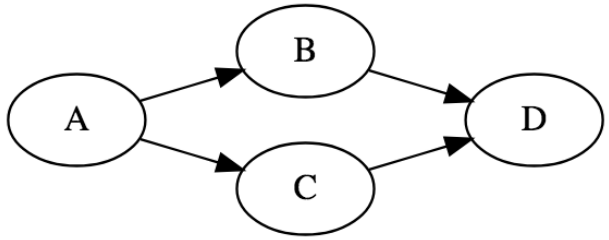
*A High-performance VLSI timing analyzer: https://github.com/OpenTimer/OpenTimer

Getting Started with Cpp-Taskflow

- **Step 1: Create a taskflow object and task(s)**
  - Use `tf::Taskflow` to create a task dependency graph
  - A task is a C++ callable objects (`std::invoke`)
- **Step 2: Add dependencies between tasks**
  - Force one task to run before (or after) another
- **Step 3: Create an executor to run the taskflow**
  - An executor manages a set of worker threads
  - Schedules the task execution through work-stealing

Revisit Hello-World in Cpp-Taskflow

```cpp
#include <taskflow/taskflow.hpp>
int main(){
   tf::Taskflow tf;
   auto [A, B, C, D] = tf.emplace(
      [] () { std::cout << "TaskA\n"; }
      [] () { std::cout << "TaskB\n"; },
      [] () { std::cout << "TaskC\n"; },
      [] () { std::cout << "TaskD\n"; }
   );
   A.precede(B);   // A runs before B
   A.precede(C);   // A runs before C
   B.precede(D);   // B runs before D
   C.precede(D);   // C runs before D
   tf::Executor().run(tf);
   return 0;
}
```



Step 1:
- *Create a taskflow object*
- *Create tasks*

Step 2:
- *Add task dependencies*

Step 3:
- *Create an executor to run*

Multiple Ways to Create a Task

```
// Create tasks one by one
tf:Task A = tf.emplace([] () { std::cout << "TaskA\n"; });
tf:Task B = tf.emplace([] () { std::cout << "TaskB\n"; });

// Create multiple tasks at one time
auto [A, B] = tf.emplace(
   [] () { std::cout << "TaskA\n"; }
   [] () { std::cout << "TaskB\n"; }
);

// Create an empty task (placefolder)
tf:Task empty = tf.placeholder();

// Modify task attributes
empty.name("empty task");
empty.work([] () { std::cout << "TaskA\n"; });
```

tf::Task is a lightweight handle to let you access/modify a task's attributes

Add a Task Dependency

```
// Create two tasks A and B
tf:Task A = tf.emplace([] () { std::cout << "TaskA\n"; });
tf:Task B = tf.emplace([] () { std::cout << "TaskB\n"; });
…

// Create a preceding link from A to B
A.precede(B);

// You can also create multiple preceding links at one time
A.precede(C, D, E);

// Create a gathering link from F to A (A run after F)
A.gather(F);

// Similarly, you can create multiple gathering links at one time
A.gather(G, H, I);
```

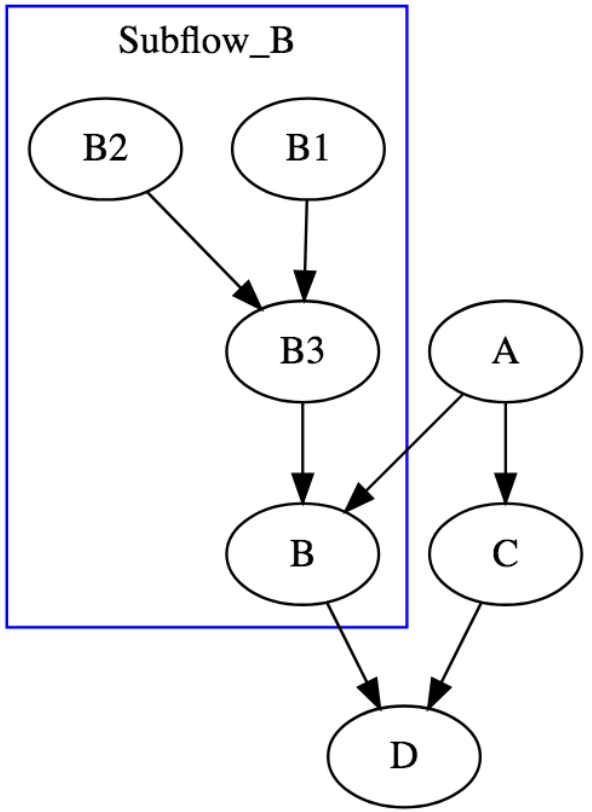
You can build any dependency graphs using `precede`

Static Tasking vs Dynamic Tasking

- **Static tasking**
  - Defines the static structure of a parallel program
  - Tasks are within the first-level dependency graph
- **Dynamic tasking**
  - Defines the runtime structure of a parallel program
  - Dynamic tasks are spawned by a parent task
  - These tasks are grouped together to form a “subflow”
    - A subflow is a taskflow created by a task
    - A subflow can join or be detached from its parent task
  - Subflow can be nested
- **Cpp-Taskflow has a *uniform* interface for both**

Unified Interface for Static & Dynamic Tasking

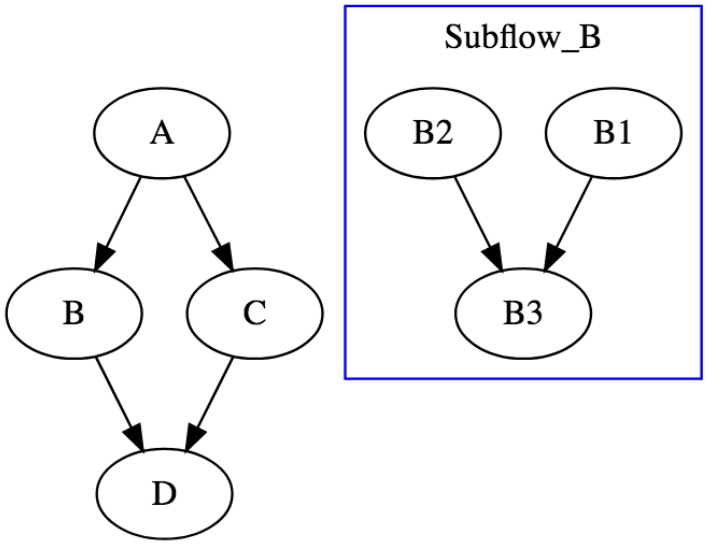
```
// create three regular tasks
tf::Task A = tf.emplace([](){}).name("A");
tf::Task C = tf.emplace([](){}).name("C");
tf::Task D = tf.emplace([](){}).name("D");

// create a subflow graph (dynamic tasking)
tf::Task B = tf.emplace([] (tf::Subflow& subflow) {
   tf::Task B1 = subflow.emplace([](){}).name("B1");
   tf::Task B2 = subflow.emplace([](){}).name("B2");
   tf::Task B3 = subflow.emplace([](){}).name("B3");
   B1.precede(B3);
   B2.precede(B3);
}).name("B");

A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C
```



Cpp-Taskflow uses `std::variant` to enable a uniform interface for both static tasking and dynamic tasking

Detached Subflow

```
// create three regular tasks
tf::Task A = tf.emplace([](){}).name("A");
tf::Task C = tf.emplace([](){}).name("C");
tf::Task D = tf.emplace([](){}).name("D");

// create a subflow graph (dynamic tasking)
tf::Task B = tf.emplace([] (tf::Subflow& subflow) {
   tf::Task B1 = subflow.emplace([](){}).name("B1");
   tf::Task B2 = subflow.emplace([](){}).name("B2");
   tf::Task B3 = subflow.emplace([](){}).name("B3");
   B1.precede(B3);
   B2.precede(B3);
   subflow.detach();
}).name("B");

A.precede(B); // B runs after A
A.precede(C); // C runs after A
B.precede(D); // D runs after B
C.precede(D); // D runs after C
```

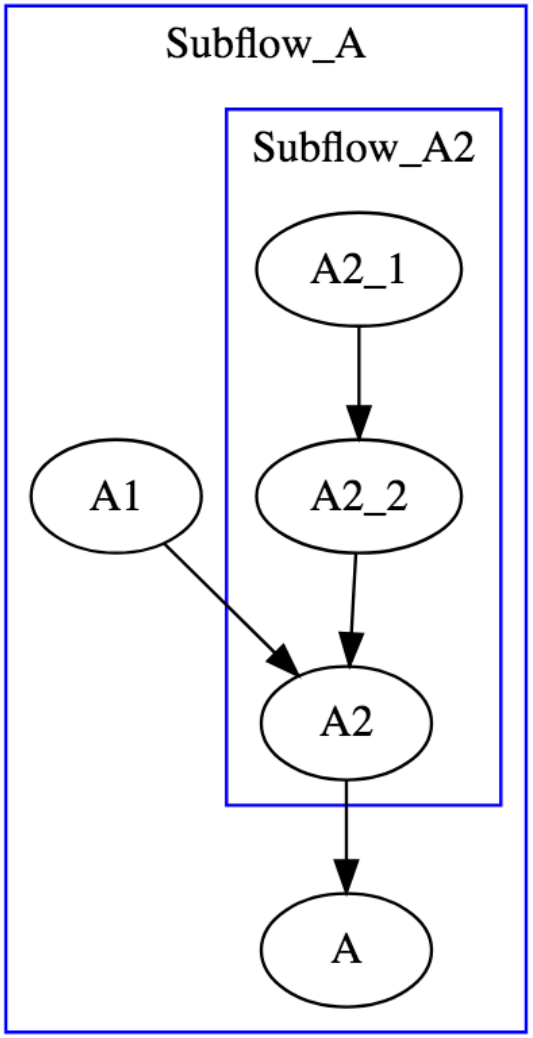


Detaching a subflow separates its execution from its parent flow, allowing execution to continue independently

Nested Subflow

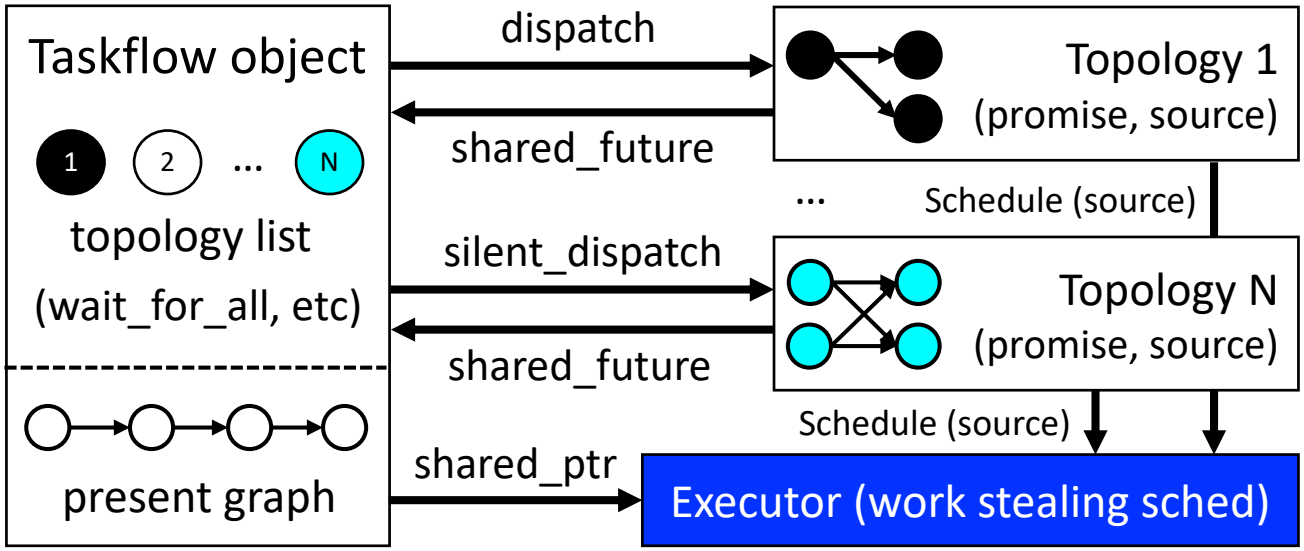
```
tf::Task A = tf.emplace([] (tf::Subflow& sbf) {
  std::cout << "A spawns A1 & subflow A2\n";
  tf::Task A1 = sbf.emplace([] () {
    std::cout << "subtask A1\n"; }
  ).name("A1");
  tf::Task A2 = sbf.emplace([] (tf::Subflow& sbf2) {
    std::cout << "A2 spawns A2_1 & A2_2\n";
    tf::Task A2_1 = sbf2.emplace([] () {
       std::cout << "subtask A2_1\n";
    }).name("A2_1");
    tf::Task A2_2 = sbf2.emplace([] () {
       std::cout << "subtask A2_2\n"; }
    ).name("A2_2");
    A2_1.precede(A2_2);
  }).name("A2");
  A1.precede(A2);
}).name("A");
```

Powerful in defining recursive dynamic workloads



Executor

- **Executor is an execution object that manages:**
  - A set of worker threads in a shared thread pool
  - Task scheduling using a work-stealing algorithm
- **Each dispatched taskflow is wrapped by a *topology***
  - A lightweight data structure used for synchronization



Execute a Task Dependency Graph

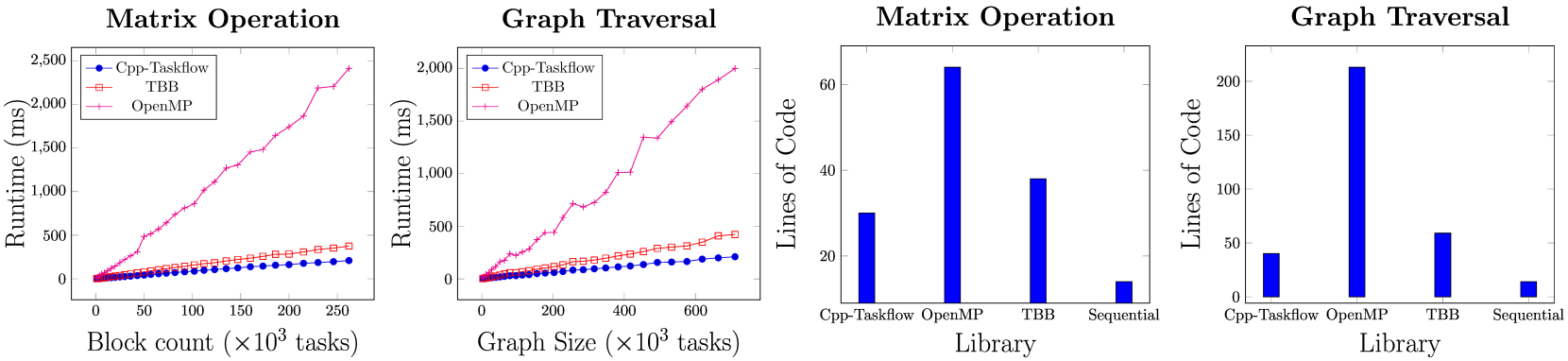
```
// Create an executor with default worker numbers (hardware_curr)
tf:Executor executor;

auto future = executor.run(taskflow);  // run the taskflow once
auto future2 = executor.run(taskflow, [](){ std::cout << "done 1 run\n"; } );

executor.run_n(taskflow, 4);  // run four times
executor.run_n(taskflow, 4, [](){ std::cout << "done 4 runs\n"; });

// run n times until the predicate becomes true
executor.run_until(taskflow, [counter=4](){ return --counter == 0; } );
executor.run_until(taskflow, [counter=4](){ return --counter == 0; },
  [](){ std::cout << "Execution finishes\n"; }
);
```

run methods are non-blocking

Multiple runs on a same taskflow will automatically synchronize to a sequential chain of execution

Micro-benchmark Performance

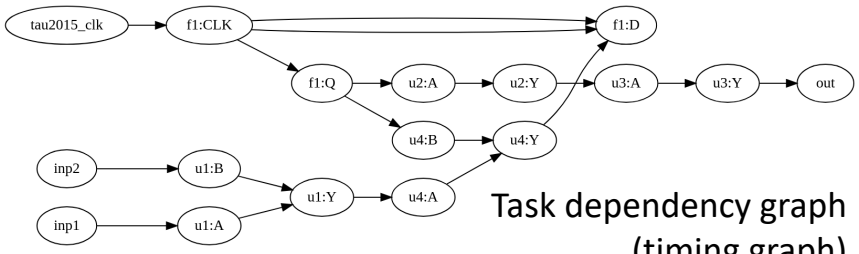
- **Measured the "pure" tasking performance**
  - Wavefront computing (regular compute pattern)
  - Graph traversal (irregular compute pattern)
  - Compared with OpenMP 4.5 and Intel TBB FlowGraph
    - G++ v8 with –fomp –O2 –std=c++17
    - Evaluated on a 4-core AMD CPU machine



Cpp-Taskflow scales the best when task counts (problem size) increases, using the least amount of code

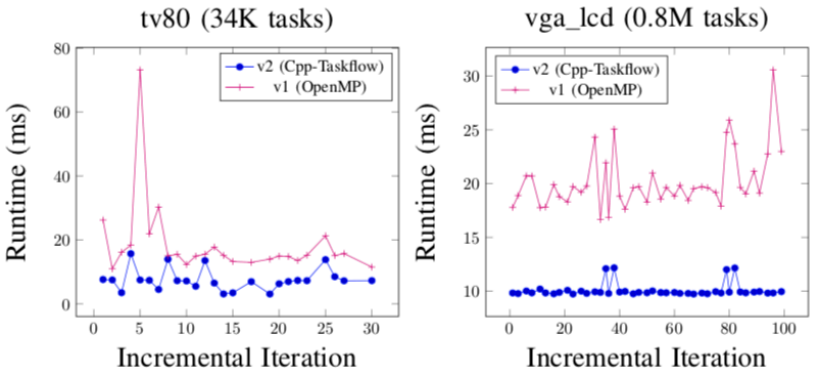
Large-Scale VLSI Timing Analysis

- **OpenTimer v1: A VLSI Static Timing Analysis Tool**
  - v1 first released in 2015 (open-source under GPL)
  - Loop-based parallelism using OpenMP 4.0
- **OpenTimer v2: A New Parallel Incremental Timer**
  - v2 first released in 2018 (open-source under MIT)
  - Task-based parallel decomposition using Cpp-Taskflow



Task dependency graph (timing graph)

Cost to develop is $275K with OpenMP vs $130K with Cpp-Taskflow! (https://dwheeler.com/sloccount/)



v2 (Cpp-Taskflow) is 1.4-2x faster than v1 (OpenMP)

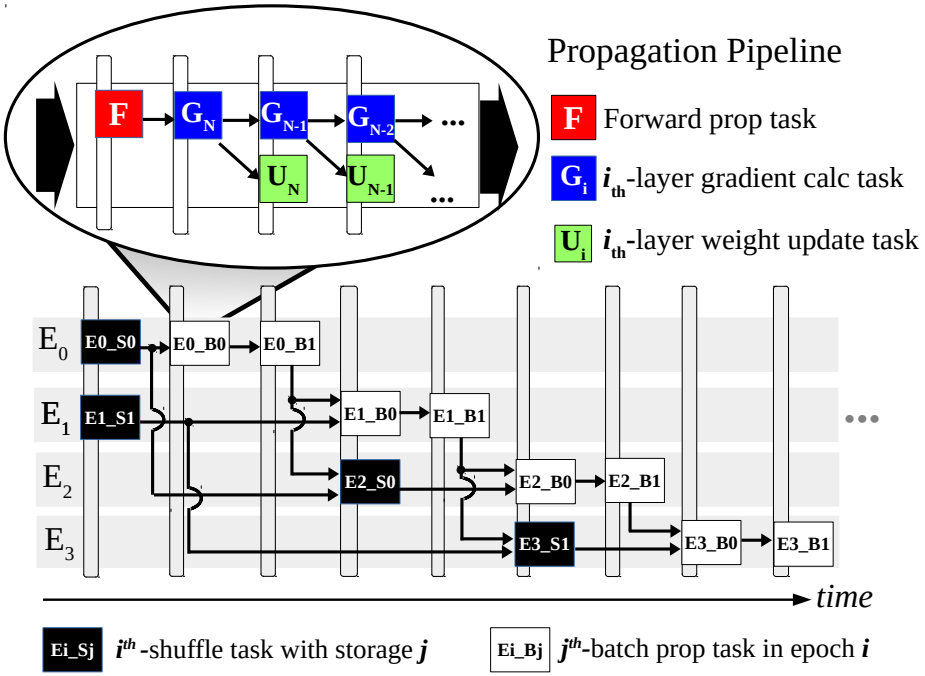
Deep Learning Model Training

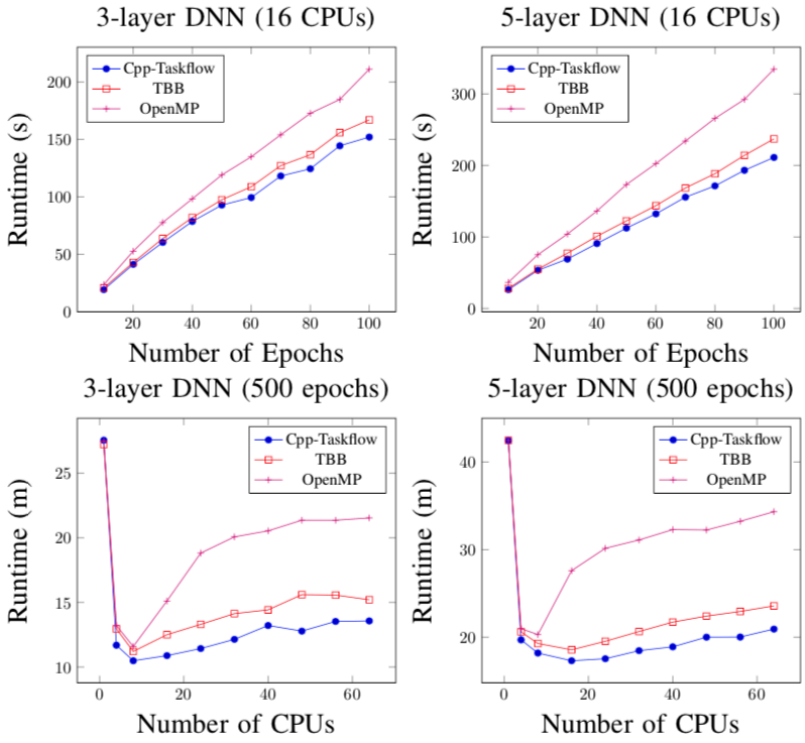
❑ 3-layer DNN and 5-layer DNN image classifier

Cpp-Taskflow			OpenMP			TBB			Sequential		
LOC	CC	T	LOC	CC	T	LOC	CC	T	LOC	CC	T
59	11	3	162	23	9	90	12	3	33	9	2

CC: cyclomatic complexity of the implementation
T: development time (in hours) by an experienced programmer

Dev time (hrs): 3 (Cpp-Taskflow) vs 9 (OpenMP)





Cpp-Taskflow is about 10%-17% faster than OpenMP and Intel TBB in avg, using the least amount of source code

Community

- **GitHub: https://github.com/cpp-taskflow (MIT)**
  - README to start with Cpp-Taskflow in just a few mins
  - Doxygen-based C++ API and step-by-step tutorials
    - https://github.com/cpp-taskflow/cpp-taskflow
  - Showcase presentation: https://cpp-taskflow.github.io/
  - Cpp-learning: https://cpp-learning.com/cpp-taskflow/

"Cpp-Taskflow has the cleanest C++ Task API I have ever seen," *Damien Hocking*

"Cpp-Taskflow has a very simple and elegant tasking interface; the performance also scales very well," *Totalgee*

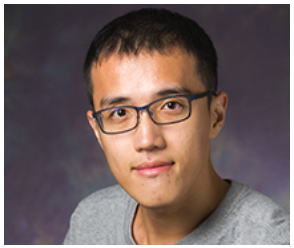
"Best poster award for open-source parallel programming library," *Cpp-Conference (voted by 500+ developers)*

Conclusion & Takeaways

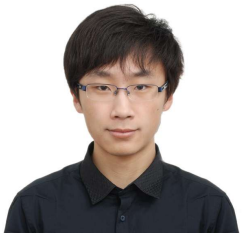
- **Cpp-Taskflow: Modern C++ Parallel Task Programming**
  - Helps C++ developers quickly write parallel task programs
  - Open source at https://github.com/cpp-taskflow
- **Solution at programming level matters a lot**
  - Of course, performance is always a top goal
    - Productivity is key to handle complex parallel workloads
  - Performance bottleneck might be surprising
    - Parallel code itself vs the supporting data structures
- **Parallel programming should be apparent to everybody**
  - Like machine learning but keep in mind the difference in:
    - Need to understand what the application is
    - In ML, you can just predicate a cat without knowing what a cat is
    - In PDC, you need to understand what a cat is to work things out

Thank You (and all users)!

T.-W. Huang



C.-X. Lin



G. Guo

M. Wong

GitHub: https://github.com/cpp-taskflow

*Please **star** our project is your like it!*

Back-up Slides

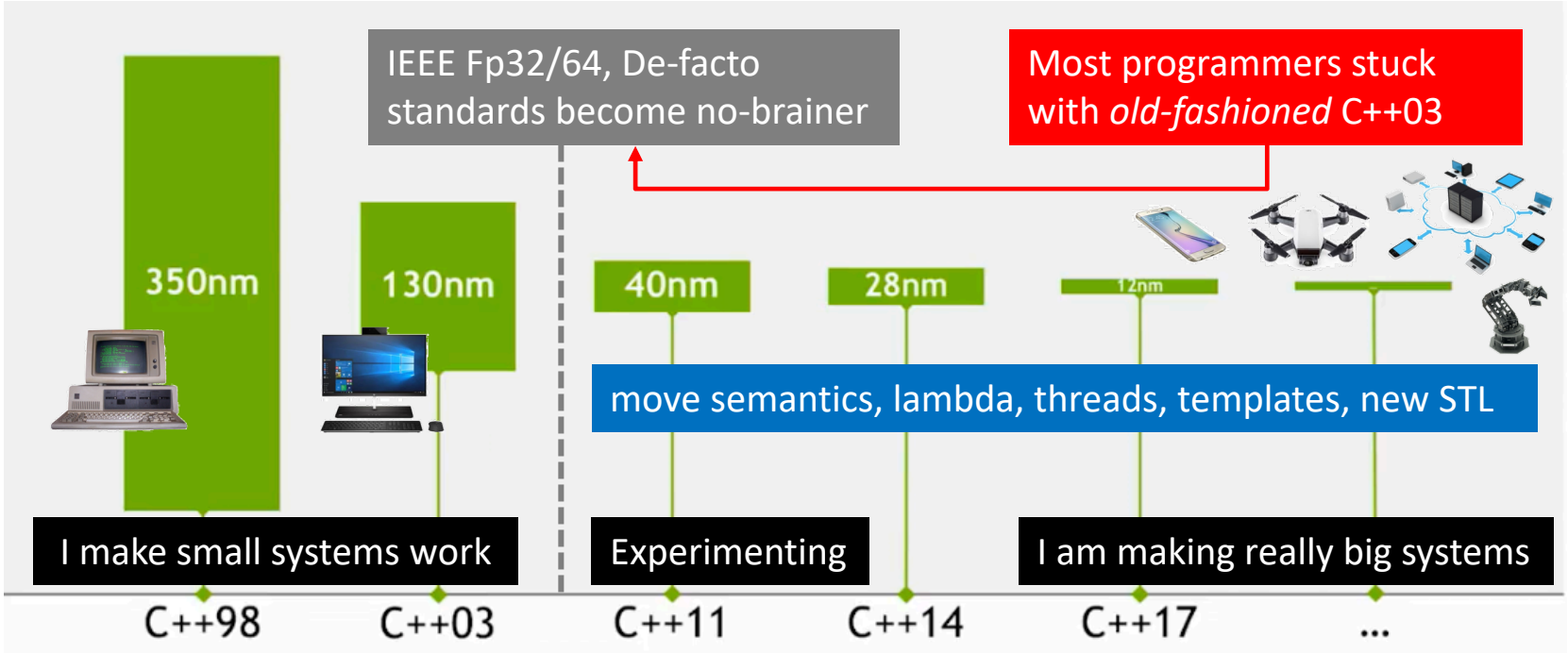
- Be gentle to existing tools
- Modern C++ enables new technology

Be Gentle to Existing Tools

- **Nobody can claim their parallel programming lib general**
  - If yes, I understand it's for business purpose ☺
- **High-performance computing (HPC) language**
  - ✓ Enabled the vast majority of HPC results for 20 years
  - x Too many distinct notations for parallel programming
- **Big-data community**
  - ✓ Good for data-driven and MapReduce workload
  - x Often not good for CPU/memory-intensive applications
- **Cpp-Taskflow**
  - ✓ A higher-level alternative to parallel task programming
  - ✓ Transparent concurrency through a new C++ programming model
  - x Currently best suitable for those with irregular compute patterns

Modern C++ Enables New Technology

- If you were able to tape out C++ …



- Achieved the performance previously not possible
- It's much more than just being modern
  - Must "rethink" the way we used to write a program