Cpp-Taskflow: Fast Parallel Programming with Task Dependency Graphs

Chun-Xun Lin*
Dept. of ECE, UIUC
IL, USA
clin99@illinois.edu

Tsung-Wei Huang*
Dept. of ECE, UIUC
IL, USA
twh760812@gmail.com

Guannan Guo
Dept. of ECE, UIUC
IL, USA
guannan4@gmail.com

Martin D. F. Wong
Dept. of ECE, UIUC
IL, USA
mdfwong@illinois.edu

ABSTRACT

As we move to multi-core era, it's important for software developers to apply multi-threading to increase the performance of their applications. However, parallel programming is much more difficult than writing a sequential program, especially when complex parallel patterns exist in the application. In this paper, we introduce Cpp-Taskflow [1], a zero-dependency library written in modern C++17 to help programmers quickly build parallel task dependency graphs. Cpp-Taskflow has a very neat and expressive API that allows users to master multi-threading in just a few minutes. Compared with existing libraries, Cpp-Taskflow is more cost-efficient in performance scaling and software integration.

1 INTRODUCTION

As a single chip accommodates more and more processing units, parallel computing is getting increasingly important and is no doubt a pivotal component in building high-performance software. Parallel computing is also very useful for EDA applications as the recent trend [2] [3] [4][5] shows the EDA tools gain significantly speedup by scaling to multiple cores. However, programming parallel applications is a very challenging task because of difficult concurrency controls. Immature parallelism might lead to unexpected result if one does not properly design the control flow. Therefore, compared to writing sequential code, developers need to devote more efforts to implementing a correct parallel program.

C++ is the most widely used programming language for developing EDA applications due to its high performance and robust standard library [6] in support for multi-threading. The standard library enables the programmers to build the parallel applications from scratch but it is not flexible because of low-level thread managements. In addition to the standard library, several libraries are invented to expedite the development of parallel program such as OpenMP [7] and Intel Thread Building Blocks (TBB) [8]. OpenMP provides users a set of predefined directives to annotate the desired parallelism and the compiler generates the parallel code based on the annotations. Intel TBB is a template library that supports many APIs for different parallel execution patterns. Although both libraries solve the low-level thread management problem of standard library, they require compiler support and library installation which causes the integration into existing projects quite inconvenient. Furthermore, both libraries are not expressive in API to describe parallel applications especially when tasks dependencies become complex.

In this paper, we present Cpp-Taskflow, a header-only and zero-dependency library written in modern C++17. Cpp-Taskflow lets users describe the task dependencies in a directed acyclic graph (DAG) fasion execute tasks in parallel. This model is very intuitive to use and is sufficient to express most parallel execution patterns. The thread execution is automatically undertaken by Cpp-Taskflow and users only need to focus on maximizing the parallelism without wrestling with the low-level thread managements. The library is header-only which allows drop-in software integration. In the following sections, we introduce the programming model of Cpp-Taskflow and the associated APIs for implementing a parallel program.

*Both authors contributed equally to the paper

2 CPP-TASKFLOW

Cpp-Taskflow uses a DAG to represent the task dependencies. A node in the DAG encapsulates a task and an edge indicates the dependency between two tasks. A task will be executed when all its predecessors are executed. Cpp-Taskflow internally employs a thread pool to execute all tasks.

2.1 Static Tasking

Listing 1 shows how to use the basic APIs of Cpp-Taskflow to create a DAG with 4 tasks.

```
1  // Cpp-Taskflow is header-only
2  #include <taskflow/taskflow.hpp>
3
4  int main(){
5
6    tf::Taskflow tf(4);   // taskflow with 4 workers
7
8    // Add 4 tasks
9    auto [A, B, C, D] = tf.silent_emplace(
10     [] () { std::cout << "TaskA\n"; },
11     [] () { std::cout << "TaskB\n"; },
12     [] () { std::cout << "TaskC\n"; },
13     [] () { std::cout << "TaskD\n"; }
14   );
15
16   A.precede(B);   // B runs after A
17   A.precede(C);   // C runs after A
18   B.precede(D);   // D runs after B
19   C.precede(D);   // D runs after C
20
21   tf.wait_for_all();   // block until finish
22
23   return 0;
24 }
```

Listing 1: A simple example.

First to use Cpp-Taskflow, only a single header taskflow.hpp needs to be included (line 2). In line 6, a Taskflow object tf is created with 4 threads in its thread pool. From line 9-14, four tasks A, B, C, D are added into tf using the API silent_emplace. The task dependency is annotated using the API precede from line 16-19. In line 21, the DAG is executed by invoking the API wait_for_all which blocks until all tasks finish. In this example, task A will be first executed. Then successors B and C will be executed *in parallel*. Task D is the last executed.

It is clear users can easily parallelize task execution with complex dependency by using those concise APIs to build DAGs. Debugging parallel programs are known to be difficult. Cpp-Taskflow eases the burden by providing a API dump to export the DAG in Graphviz [9] format. With the visualization capability, users can directly inspect the execution order to identify potential bugs. Figure 1 shows the dependency graph exported from Listing 2.

```cpp
#include <taskflow/taskflow.hpp>

int main(){

  tf::Taskflow tf(4);
  auto A = tf.silent_emplace([](){}).name("A");
  auto B = tf.silent_emplace([](){}).name("B");
  auto C = tf.silent_emplace([](){}).name("C");
  auto D = tf.silent_emplace([](){}).name("D");
  auto E = tf.silent_emplace([](){}).name("E");

  A.broadcast(B, C, E);
  C.precede(D);
  B.broadcast(D, E);

  std::cout << tf.dump();

  return 0;
}
```

Listing 2: A debugging example.

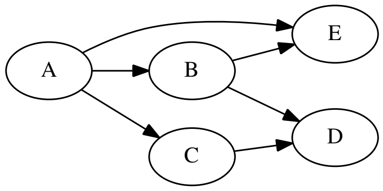

Figure 1: The dependency graph of Listing 2.

2.2 Dynamic Tasking

Another powerful feature of Cpp-Taskflow is *dynamic tasking*, i.e. a task can spawn new tasks with dependency during execution. The new tasks are grouped into a subflow graph and executed as a normal dependency graph. Listing 3 shows an example of dynamic tasking and figure 2 is the corresponding dependency graph.

```cpp
1  #include <taskflow/taskflow.hpp>
2
3  int main(){
4
5    tf::Taskflow tf(8);
6
7    // create three regular tasks
8    auto A = tf.silent_emplace([](){}).name("A");
9    auto C = tf.silent_emplace([](){}).name("C");
10   auto D = tf.silent_emplace([](){}).name("D");
11
12   // create a subflow graph (dynamic tasking)
13   auto B = tf.silent_emplace([] (auto& subflow) {
14     auto B1 = subflow.silent_emplace([](){}).name("B1");
15     auto B2 = subflow.silent_emplace([](){}).name("B2");
16     auto B3 = subflow.silent_emplace([](){}).name("B3");
17     B1.precede(B3);
18     B2.precede(B3);
19   }).name("B");
20
21   A.precede(B);  // B runs after A
22   A.precede(C);  // C runs after A
23   B.precede(D);  // D runs after B
24   C.precede(D);  // D runs after C
25
26   // execute the graph without cleanning up topologies
27   tf.wait_for_all();
28
29   return 0;
30 }
```

Listing 3: A dynamic tasking example.

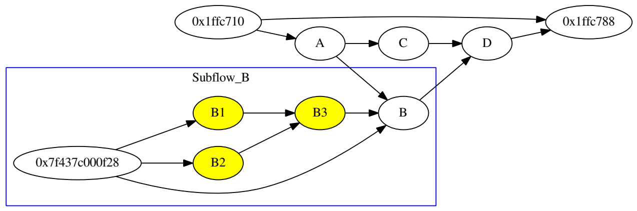

Figure 2: The dependency graph of Listing 3.

In Listing 3, we create a dependency graph with 4 tasks A, B, C and D from line 8-13. From line 14-18, task B further spawns 3 tasks B1, B2 and B3 by grouping them in a subflow. By default, a subflow joins to the parent node (task B here), which guarantees the subflow will be executed before the successors of its parent. The subflow can change to join with the end of the dependency graph by calling subflow.detach(). Figure 3 shows the resulting detached subflow.

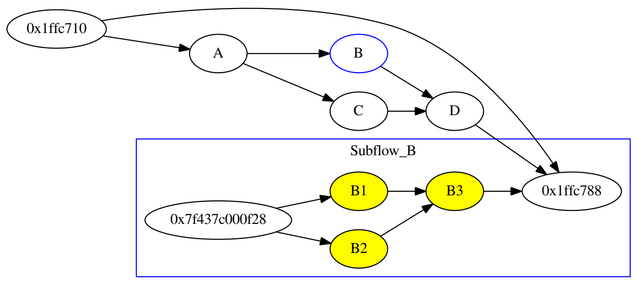

Figure 3: A detached subflow example.

A subflow can also spawn new tasks, i. e. users can recursively create nested subflows. Listing 4 show a nested subflow and the dependency graph is figure 4.

```cpp
#include <taskflow/taskflow.hpp>

int main(){

  auto A = tf.silent_emplace([] (auto& sbf){
    std::cout << "A spawns A1 & subflow A2\n";
    auto A1 = sbf.silent_emplace([] () {
      std::cout << "subtask A1\n";
    }).name("A1");

    auto A2 = sbf.silent_emplace([] (auto& sbf2){
      std::cout << "A2 spawns A2_1 & A2_2\n";
      auto A2_1 = sbf2.silent_emplace([] () {
        std::cout << "subtask A2_1\n";
      }).name("A2_1");
      auto A2_2 = sbf2.silent_emplace([] () {
        std::cout << "subtask A2_2\n";
      }).name("A2_2");
      A2_1.precede(A2_2);
    }).name("A2");
    A1.precede(A2);
  }).name("A");

  return 0;
}
```

Listing 4: A nested subflow example.

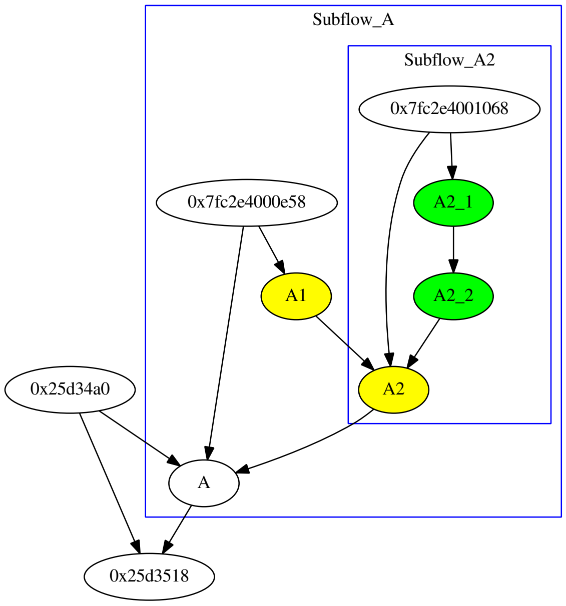

Figure 4: The dependency graph of Listing 4.

2.3 Thread Pool

The Threadpool class is the task execution arena of Cpp-Taskflow. It manages how tasks are distributed to allocated threads in a shared storage. Before each thread executes its task, it needs to obtain a Worker struct which contains a local conditional variable and a function element. When a new task needs to be executed by a thread in the pool, it will actively search for idle Workers that rest in the thread pool. Once a spare Worker is found, one sleeping thread will be notified through the local conditional variable in the Worker. The Worker won't be released until the task completes. On the other hand, if all threads are busy, the new task will be pushed to a global pending queue waiting for a thread to finish its current task. The advantage of this scheduling strategy is to reduce the overhead of pushing the task to the pending queue, popping it out of the queue and related competition over the shared mutex if a thread is found idle when a new task in inserted. Therefore, it can achieve a promising throughput for millions of lightweight tasks. Figure 5 outlines the thread pool in Cpp-Taskflow.

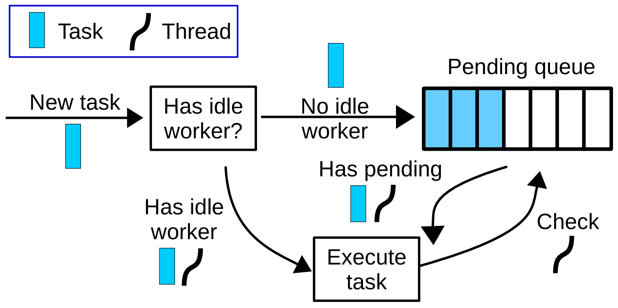

Figure 5: Thread pool in Cpp-Taskflow.

REFERENCES

[1] Cpp-Taskflow. https://github.com/cpp-taskflow/cpp-taskflow.
[2] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. Multi-threaded collision-aware global routing with bounded-length maze routing. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 200–205, New York, NY, USA, 2010. ACM.
[3] Tsung-Wei Huang and Martin D. F. Wong. Opentimer: A high-performance timing analysis tool. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 895–902, Piscataway, NJ, USA, 2015. IEEE Press.
[4] L. Stok. Developing parallel eda tools [the last byte]. *IEEE Design Test*, 30(1):65–66, Feb 2013.
[5] Yi-Shan Lu and Keshav Pingali. *Can Parallel Programming Revolutionize EDA Tools?*, pages 21–41. Springer International Publishing, Cham, 2018.
[6] Thread support library. https://en.cppreference.com/w/cpp/thread.
[7] OpenMP. https://www.openmp.org/.
[8] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.
[9] Graphviz. https://www.graphviz.org/.