A Modern C++ Parallel Task Programming Library

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

Tsung-Wei Huang*
ECE Dept, University of Utah, UT
twh760812@gmail.com

Guannan Guo
ECE Dept, UIUC, IL
gguo4@illinois.edu

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

ABSTRACT

In this paper we present Cpp-Taskflow, a C++ parallel programming library that enables users to quickly develop parallel applications using the task dependency graph model. Developers formulate their application as a task dependency graph and Cpp-Taskflow will manage the task execution and concurrency control. The task graph model is expressive and composable. It can express both regular and irregular parallel patterns, and developers can quickly compose large programs from small parallel modules. Cpp-Taskflow has an intuitive and unified API set. Users only need to learn the APIs to build and dispatch a task graph and no complex parallel programming concept is required. We have conducted experiments using both micro-benchmarks and real-world applications and Cpp-Taskflow outperforms state-of-the-art parallel programming libraries in both runtime and coding effort. Cpp-Taskflow is open-source and has been used in both industry and academic projects. From our users' feedback, we believe Cpp-Taskflow can benefit the industry and research community greatly through its ease-of-programming and inspire new research directions in multimedia system/software design.

KEYWORDS

Parallel programming; task parallelism; task dependency graph

ACM Reference Format:
Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. 2019. A Modern C++ Parallel Task Programming Library. In *Proceedings of the 27th ACM International Conference on Multimedia (MM '19), October 21–25, 2019, Nice, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3343031.3350537

1 INTRODUCTION

Multicore processors are prevalent from desktops, laptops, tablets to mobile devices. *How to effectively utilize those computing resources to maximize software performance?* This is a critical question that software developers must consider, especially when building complex parallel applications such as artificial intelligence, numerical simulation, machine learning and multimedia big data analytics [1] [2] [3] [4] [5]. Writing parallel code is considered much more difficult than the sequential counterpart. Programmers have to pay extra attentions to the concurrency control to avoid unexpected behavior during runtime, for example, using locks to protect shared data or atomic variables to avoid data race. The situation is getting more challenging when applications exhibit complex data or operation dependency, which is typical in real-world problems. As a result, it's necessary to have an efficient approach to write parallel code.

In this paper, we present Cpp-Taskflow, a modern C++ task-based parallel programming library. Cpp-Taskflow was motivated from a real-world project of VLSI timing analysis [6]. Cpp-Taskflow lets users express their parallelism using the intuitive task graph model. The task graph model is simple yet very powerful as it can represent both regular and irregular parallel patterns. The task graph model abstracts away complex concurrency management and allows users to focus on exploiting parallelism within their applications. Cpp-Taskflow provides well-designed APIs to keep the code concise and readable. We have a unified task graph construction interface for both static and dynamic parallelism, so users can learn those APIs quickly and utilize them to implement various parallel patterns. Cpp-Taskflow supports visualization for program debugging and profiling. Users can dump the task graph to inspect the program execution flow and they can view the thread activities in a Chrome browser. We have conducted experiments on a set of micro-benchmarks and a real-world application [7] against Intel Threading Building Blocks [8] and OpenMP [9]. Cpp-Taskflow achieves comparable performance with fewer lines of code, faster runtime, and better scalability.

We understand each library has its own uniqueness and value, and it's up to users to decide which best suits their needs. Cpp-Taskflow has been used in many industrial and academic projects [10]. We are committed to free sharing of our technical innovation to facilitate the research in parallel computing, machine learning, and multimedia. We are working actively with our users to improve Cpp-Taskflow. The project is open-source and more details can be found in [10].

2 CPP-TASKFLOW

In Cpp-Taskflow, the programming is centered around two classes: tf::Taskflow and tf::Executor. We will explain how to use them in this section.

2.1 Task Dependency Graph

In Cpp-Taskflow, a task is a C++ object of *Callable* type [7]. To create tasks, the first step is to create an object of the tf::Taskflow class. A taskflow object allows you to build a task dependency graph where nodes are tasks and directed edges indicate dependency. Listing 1 shows an example of adding three tasks via the

*Both authors contributed equally to this research.

MM '19, October 21–25, 2019, Nice, France

ACM ISBN 978-1-4503-6889-6/19/10…$15.00
https://doi.org/10.1145/3343031.3350537

emplace method. The emplace method can create multiple tasks at one time. After tasks are created, users can assign names to tasks and specify the dependency between tasks via the name and precede method, respectively. A task A *precedes* a task B if task B can only run **after** task A completes its execution.

```
tf::Taskflow taskflow;

// Create a task
auto taskA = taskflow.emplace(
  [](){ std::cout << "Task A\n"; }
);

// Create two tasks at one time
auto [taskB, taskC] = taskflow.emplace(
 [](){ std::cout << "Task B\n"; },
 [](){ std::cout << "Task C\n"; },
);

// Name the tasks
taskA.name("taskA");

// Specify the dependency
taskA.precede(taskB, taskC);
```

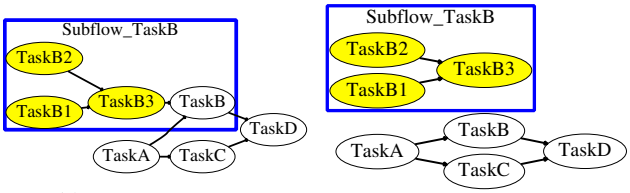
Listing 1: Create a task dependency graph.

2.2 Dynamic Tasking

Cpp-Taskflow has another powerful feature: *dynamic tasking* that enables a task to create and dispatch a task dependency graph at runtime to obtain dynamic parallelism. Listing 2 shows an example of dynamic tasking. In this example task B spawns a task dependency graph that has three tasks. A task that requires dynamic parallelism has to take an additional argument of type tf::Subflow and uses the emplace method to create a new task dependency graph. The new task dependency graph will by default *join* its parent task. However, users can make it run independently by calling the detach method. A detached task dependency graph will join the end of its parent's task dependency graph. Figure 1 shows the spawned task dependency graphs in joined and detached modes, respectively. Dynamic tasking empowers users to parallelize frequently used computing patterns such as recursive and nested flows.

```
tf::Taskflow flow;

// Create three tasks
auto [taskA, taskC, taskD] = flow.emplace(
  [](){ std::cout << "Task A\n"; },
  [](){ std::cout << "Task C\n"; },
  [](){ std::cout << "Task D\n"; }
);

// Create a task with subflow
auto taskB = flow.emplace(
 [](auto &subflow){
   std::cout << "Task B\n";
   // Spawn a new task dependency graph
   auto [B1, B2, B3] = subflow.emplace(
     []() { std::cout << "Task B1\n"; },
     []() { std::cout << "Task B2\n"; },
     []() { std::cout << "Task B3\n"; }
   );
   B3.gather(B1, B2);

   // Detach the new task dependency graph
   subflow.detach();
});

// Specify the dependency
taskA.precede(taskB, taskC);
taskD.gather(taskB, taskC);
```

Listing 2: An example of dynamic tasking.



(a) A joined subflow. **(b) A detached subflow.**

Figure 1: Comparison of joined and detached subflows.

2.3 Composition

An useful feature of task dependency graph is the composability. Users can use the composed_of method to compose several task dependency graphs to a large and complex task dependency graph. The composed_of method returns a *module task*. Users can use the precede method to add dependency between module tasks and other tasks. Listing 3 shows an example of task dependency graph composition.

```
tf::Taskflow fA;

// Create four tasks
auto [fA1, fA2, fA3, fA4] = fA.emplace(
  [](){ std::cout << "Task fA1\n"; },
  ...
);
fA1.precede(fA2, fA3);
fA4.gather(fA2, fA3);

tf::Taskflow fB;

// Create three tasks
auto [fB1, fB2, fB3] = fB.emplace(
  [](){ std::cout << "Task fB1\n"; },
  ...
);

auto moduleA = fB.composed_of(fA);

fB1.precede(moduleA, fB2);
moduleA.precede(fB3);
fB2.precede(fB3);
```

Listing 3: An example of task dependency graph composition.

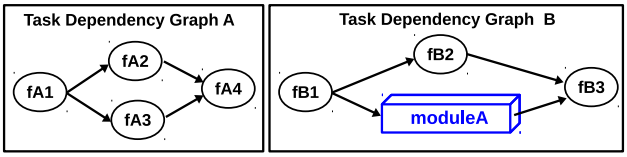


Figure 2: The task dependency graphs of the example in Listing 3.

2.4 Execution

After creating task dependency graphs, the next step is to dispatch graphs to an *executor* object of type tf::Executor . An executor object manages thread construction and destruction and provides several methods to execute task dependency graphs through an efficient work-stealing algorithm. Table 1 summarizes the execution methods and Listing 4 demonstrates the usage of those execution methods.

Method	Description
run	Execute a graph once
run_n	Execute a graph multiple times
run_until	Execute a graph until a condition is met
wait_for_all	Wait until all running graphs finish

Table 1: Summary of execution methods.

```
tf::Taskflow tf;

// Add tasks to tf
...

tf::Executor executor;

executor.run(tf);       // Run the flow once
executor.run_n(tf, 6); // Run the flow six times
// Run the flow until the number becomes 0
executor.run_until(tf, [number=4] () mutable {
  return number-- == 0:
});
```

Listing 4: Demonstration of different execution methods.

2.5 Debugging and Profiling

Debugging a parallel program is very challenging due to the non-deterministic nature. Cpp-Taskflow supports the visualization of task dependency graphs to let users inspect the task execution flow. Users can use the name method to assign a taskflow object a name and the dump method to export the object's task graph in DOT format [11]. Listing 5 shows an example of naming and dumping a task dependency graph. Figure 3 demonstrates the task dependency graphs of two taskflow objects.

```
tf::Taskflow fA;

// Naming the taskflow object
fA.name("Taskflow_A");

// Add seven tasks
auto [A1, A2, A3, A4, A5, A6, A7] = fA.emplace(
  [](){ std::cout << "A1\n"; },
  ...
);

// Specify dependency
A1.precede(A3, A4);
A2.precede(A5);
A6.gather(A3, A5);
A4.precede(A7);

// Dump the task dependency graph
std::cout << fA.dump() << std::endl;

tf::Taskflow fB;

// Add five tasks
auto [B1, B2, B3, B4, B5] = fB.emplace(
  [](){ std::cout << "B1\n"; },
  ...
);

// Compose taskflow A
auto moduleA1 = fB.composed_of(fA);

// Specify dependency
B1.precede(B2, moduleA1);
B2.precede(B3, B4);
B5.gather(B3, B4, moduleA1);

std::cout << fB.dump() << std::endl;
```

Listing 5: Visualization of a task dependency graph.

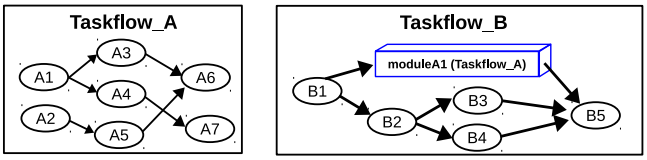


Figure 3: The task dependency graphs of two taskflow objects in Listing 5.

Profiling is very important when developers analyze their application's performance. Cpp-Taskflow allows users to record the thread activities and visualize them in a Chrome browser. To enable profiling, users create an *observer* of type tf::Executor Observer through executor's make_observer method. An observer will record each task's start time (via on_entry method) and end time (via on_exit method) during execution. The observer can dump the recorded timestamps into a JSON file and users can visualize the execution timeline by loading the JSON file in the *chrome://tracing* developer tool. Listing 6 shows how to create an observer to monitor the thread activities. Figure 4 displays the task execution timeline in a Chrome browser.

```
tf::Taskflow taskflow;
tf::Executor executor;
// Create an observer
auto observer = executor.make_observer<
    tf::ExecutorObserver>();
```

```
5
6  // Add tasks and dispatch the flow to execution
7  ...
8
9  // Dump the timestamps to a JSON file
10 std::ofstream ofs("timestamps.json");
11 observer->dump(ofs);
```

Listing 6: Use an observer to monitor the thread activities.

Figure 4: Thread activities displayed in chrome://tracing.

3 A MACHINE LEARNING APPLICATION

Machine learning has been successfully applied to several multimedia topics such as image classification, speech recognition and so on [4] [5]. We demonstrate applying Cpp-Taskflow to parallelize a machine learning application: MNIST [12] dataset, and compare its performance and coding effort with OpenMP [9]. The MNIST dataset contains images of handwritten digits and it is widely used to test the effectiveness of machine learning algorithms. In this demonstration, we build a 5-layer deep neural network (DNN) to classify those images. We adopt the task pipeline strategy proposed by [7] to parallelize the DNN training. Each batch starts with a task for forward propagation and then followed by a sequence of gradient calculation and weight update tasks for each layer. We pipeline the gradient calculation and weight update tasks between successive layers to enable parallelism within each batch. Next we create tasks for data shuffle per epoch. We allocate additional data storages to have a shuffle task start earlier preparing the data for later epochs. We compare the implementations of OpenMP [9] and Cpp-Taskflow. OpenMP is the most popular parallel programming library in high-performance computing and we use OpenMP's task depend clause to implement this parallelization strategy. For Cpp-Taskflow, we implement this application using the taskflow object and use the executor's run method for execution. We run all implementations on a machine with a Intel Xeon W-2175 processor and 128 GB memory and we launch 10 threads in this experiment. The operating system is Ubuntu 18.04 and the OpenMP version is 4.5 (201511). The learning rate is set to 0.0001 and for each number of epochs we run five times and take the average. During the experiment we use the system command `taskset` to bond the process to the first 10 cores. Figure 5 plots the runtime of both implementations and Table 2 shows the code complexity measured by Lizard [13][1]. The implementation of Cpp-Taskflow is slightly faster than OpenMP. Regarding the code complexity, OpenMP is 35% longer than Cpp-Taskflow in lines of code.

[1]Because Lizard takes compiler directives (starts with #) as a comment, we remove the # when measuring the OpenMP implementation.

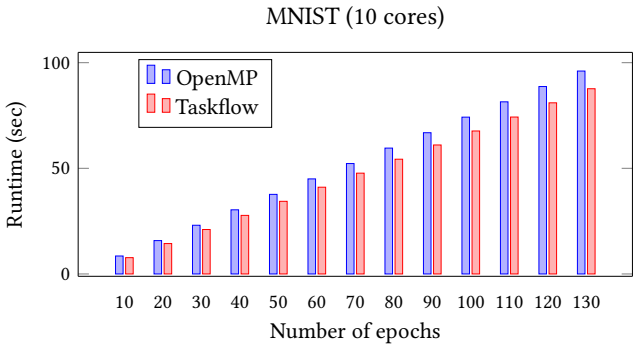


Figure 5: DNN training runtime of OpenMP and Cpp-Taskflow with using taskflow object.

Table 2: Code complexity [13] of the three implementations.

Library	Total NLOC	Avg Token	Avg CCN
OpenMP	93	1058	11
Taskflow	60	600	11

NLOC: lines of code. **CCN**: cyclomatic complexity number.

4 AVAILABILITY

Cpp-Taskflow is open-source on Github [10] under MIT license. The API documentation, tutorials and cookbook are also available on Github. We have presented Cpp-Taskflow at CppCon which is the premier C++ developer conference and the video is on YouTube [14].

5 ACKNOWLEDGEMENT

Cpp-Taskflow is supported by NSF Grant CCF-1718883 and DARPA Grant FA 8650-18-2-7843.

REFERENCES

[1] W. Zhu, P. Cui, Z. Wang, and G. Hua. Multimedia big data computing. *IEEE MultiMedia*, 22(3):96–c3, July 2015.
[2] Z. Wang, S. Mao, L. Yang, and P. Tang. A survey of multimedia big data. *China Communications*, 15(1):155–176, Jan 2018.
[3] Samira Pouyanfar, Yimin Yang, Shu-Ching Chen, Mei-Ling Shyu, and S. S. Iyengar. Multimedia big data analytics: A survey. *ACM Comput. Surv.*, 51(1):10:1–10:34, January 2018.
[4] Jitao Sang, Jun Yu, Ramesh Jain, Rainer Lienhart, Peng Cui, and Jiashi Feng. Deep learning for multimedia: Science or technology? In *Proceedings of the 26th ACM International Conference on Multimedia*, MM '18, pages 1354–1355, New York, NY, USA, 2018. ACM.
[5] Y. Yan, M. Chen, M. Shyu, and S. Chen. Deep learning for imbalanced multimedia data classification. In *2015 IEEE International Symposium on Multimedia (ISM)*, pages 483–488, Dec 2015.
[6] T.-W Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *IEEE/ACM ICCAD*, pages 895–902, 2015.
[7] T.-W Huang, C.-X. Lin, Guannan Guo, and Martin D. F. Wong. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. *IEEE IPDPS*, pages 974–983, 2019.
[8] Intel Threading Building Blocks. [Online]. Available: https://www.threadingbuildingblocks.org.
[9] OpenMP. [Online]. Available: https://www.openmp.org/.
[10] Cpp-Taskflow. [Online]. Available: https://github.com/cpp-taskflow/cpp-taskflow.
[11] The DOT Language. [Online]. Available: https://www.graphviz.org/.
[12] MNIST. [Online]. Available: http://yann.lecun.com/exdb/mnist/.
[13] Lizard. [Online]. Available: http://www.lizard.ws/.
[14] Cpp-Taskflow lightning talk. [Online]. Available: https://www.youtube.com/watch?v=ho9bqIJkvkc.