An Efficient and Composable Parallel Task Programming Library

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

Tsung-Wei Huang
ECE Dept, University of Utah
Salt Lake City, UT, US
twh760812@gmail.com

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

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

***Abstract*—Composability is a key component to improve programmers' productivity in writing fast market-expanding applications such as parallel machine learning algorithms and big data analytics. These applications exhibit both regular and irregular compute patterns, and are often combined with other functions or libraries to compose a larger program. However, composable parallel processing has taken a back seat in many existing parallel programming libraries, making it difficult to achieve modularity in large-scale parallel programs. In this paper, we introduce a new parallel task programming library using composable tasking graphs. Our library efficiently supports task parallelism together with an intuitive task graph construction and flexible execution API set to enable reusable and composable task dependency graphs. Developers can quickly compose a large parallel program from small and modular parallel building blocks, and easily deploy the program on a multicore machine. We have evaluated our library on real-world applications. Experimental results showed our library can achieve comparable performance to Intel Threading Building Blocks with less coding effort.**

***Index Terms*—parallel programming, multithreading**

I. Introduction

The key to make developers productive in writing software is *composability*. We use libraries written by other developers to compose a large program, or we decompose a job into smaller pieces to tame the complexity in software development. Composability is especially important in developing fast market-expanding applications such as high-performance machine learning, data analytics, and parallel simulation engines [1]. These applications exhibit both *regular* and *irregular* compute patterns, and are often combined with other functions to compose large software that will be deployed on a multicore machine or a distributed cloud [2], [3]. However, *composable parallel processing* is rarely addressed as the first-class concept by existing parallel programming libraries [4]. Many libraries were designed to solve a single hard problem as fast as possible, leaving users to decide composition with their own practice. This can create a lot of pain and data engineering tasks for developers of different teams to collaborate on a large parallel application. Some common problems include confusing API mix-uses, unwanted coupling layers, error-prone dependency wrappers, inconsistent threading models, and suboptimal scheduling results.

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

The traditional interface for program decomposition is *function call*. Developers break down a large *sequential* program into a specific set of tasks each wrapped in a function call with clear definition of data exchange. These function calls are often *modular* and *reusable* to make the codebase maintainable and readable. However, *composable parallel programming* is way more challenging. Modern parallel workloads typically combine a broad mix of algorithms, functions, and libraries. Each library manages its own threads and task execution, making it difficult to perform optimization across different libraries. When coupling these software pieces together, we need to tackle the *dependencies* both inside and outside the libraries. Some libraries are already parallel and they are being used by other parallel programs and so forth. There are many practical issues to consider such as thread management, resource over-subscription, and concurrency controls. As a result, the lack of a clear and unified interface has a serious impact on performance, even when individual libraries are heavily optimized.

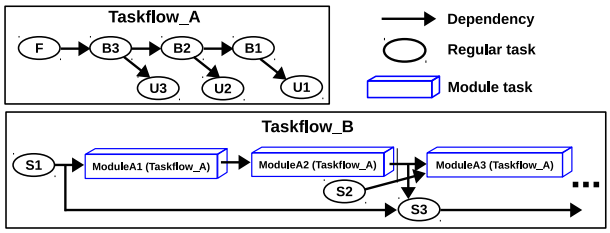

Fig. 1. Using our composable task dependency graph to describe a parallel neural network training workload. Taskflow object A represents one training iteration and is used to compose taskflow object B for the entire training procedure.

In this paper, we introduce a powerful parallel programming model that enables efficient composition. Our model is built on top of the modern C++ tasking library, Cpp-Taskflow [5], but largely enhanced its capability with two key design changes: (1) separating the task dependency graph and execution kernel and (2) making the task dependency graph reusable and composable. In our model, a *taskflow object* consists of a *composable task dependency graph* and user-friendly APIs to facilitate the creation of *modular* and *reusable* parallel compute patterns and libraries. These libraries can recursively compose large and complex parallel computations on a sin-

gle machine, taking advantage of multicore processing while sharing thread resources to minimize overhead. Figure 1 gives an example of using taskflow objects to describe a parallel training algorithm of a deep neural network (DNN). Taskflow object A represents a training pipeline. Taskflow object B is composed of multiple As and other tasks to complete the training procedure. Also, users can easily couple B with other parallel computations. There is no redundancy from the programmability standpoint. We summarize our contributions as follows:

- **A new composable parallel programming model**. We developed a composable task interface to enable efficient composition of parallel workloads. Our library lets users quickly describe a large parallel program through composition of modular and reusable task graphs that embed both regular and irregular compute patterns. The program runs on a multicore machine with automatic scheduling optimization across different layers of composed tasks.
- **A unified task composition interface**. We developed a unified task graph construction interface that can capture a diverse set of tasks from single sequential functions to large parallel dependent tasks or even out-of-context executions such as third-party calls and process forks. The unified interface empowers developers with both explicit and implicit task graph composition to explore cross-layer optimizations of their parallel workloads.
- **A simple and efficient composition API**. We developed a user-friendly API to describe task dependency graph composition using modern C++17 syntax. Users can fully take advantage of the rich features of our engine together with robust standard C++ libraries to productively compose many parallel applications. Our library effectively separates users from low-level difficult concurrency details and offers *transparent* scaling to many cores and future hardware generation.

Our work highlights the *interface for composing parallel computing workloads* as an important area to work in order to embrace high performance and high developer productivity at the same time. Today's parallel workloads are large and complex, and are often combined with many sub-components of parallel computations developed by different people. Most existing parallel programming libraries feature domain-specific functions to help users optimize their workloads but require them to fully understand the entire program structure. This inevitably degrades the productivity of developers to collaborate on large projects through efficient composition. Our library gives developers enough freedom to implement their own computations, while supporting the performance optimizations across libraries.

II. Composable Parallel Processing

In this section, we discuss in detail our composable parallel task programming library *Cpp-Taskflow v2*[1].

[1]Source code is available in [6]

A. Cpp-Taskflow: A Modern C++ Parallel Task Programming Library

We developed our composable task interface on top of Cpp-Taskflow, an open-source parallel programming library based on task dependency graphs [5]. Cpp-Taskflow leverages the power of modern C++ and task-based approaches to enable efficient implementations of parallel decomposition strategies. Users focus on high-level description of dependent tasks for their parallel workloads, leaving difficult details such as concurrency controls, work stealing, and scheduling to the library. Cpp-Taskflow supports both static and dynamic tasking in a uniform fashion. Static tasking lets users create task dependency graphs at programming time while dynamic tasking occurs at runtime. However, the ordinary task dependency graph in Cpp-Taskflow is not composable nor reusable. Like other libraries, the lack of composability prevents users from modular design to reuse available components of heavily refined task graphs. Our goal is thus to enable a composable task interface to enhance the capability of Cpp-Taskflow.

B. A New Task Dependency Graph

Cpp-Taskflow v2 introduces a new composable task interface, the *tf::Taskflow class*. tf::Taskflow is the main gateway to create a composable task dependency graph. It inherits all the task construction methods from Cpp-Taskflow. Listing 1 demonstrates how to create a task dependency graph with two tasks A and B where B runs after A and B spawns a new task B1 during runtime. Cpp-Taskflow v2 separates the task dependency graph from executor. Users now have full control over their task dependency graphs but are also responsible for their lifetime.

```
1  tf::Taskflow taskflow;
2
3  // Add a static task
4  auto taskA = taskflow.emplace([](){
5    std::cout << "Task A\n";
6  });
7
8  // Dynamic tasking
9  auto taskB = taskflow.emplace([](auto &subflow){
10   std::cout << "Task B\n";
11   subflow.emplace([](){
12     std::cout << "Task B1\n";
13   });
14 });
15
16 taskA.precede(taskB);
```

Listing 1. Create a task dependency graph of two dependent static tasks and one dynamic task.

C. Execute a Task Dependency Graph

A significant change by Cpp-Taskflow v2 is the decoupling of executor and task graph. Cpp-Taskflow v2 defines tf::Executor class that has a rich set of methods to run a task dependency graph. A task dependency graph can be run by an executor multiple times in arbitrary order. Users can also give a predicate to specify the stopping criteria. Listing 2 demonstrates a set of common methods to run a task dependency graph. Line 1:3 creates a taskflow object and adds

some tasks. Line 6 creates an executor. An executor is nothing but a pluggable scheduler to dispatch tasks to threads in a shared pool. The simplest way is to execute a task dependency graph only once via the run method (line 9). Alternatively, users can call run_n to run a task dependency graph multiple times (line 13). The bottommost call is run_until (line 19), which keeps running until the predicate becomes true. All methods accept a callable object as a callback after the task execution completes. To enable more asynchronous control, each of these methods returns a std::future for users to inspect the execution status or incorporate non-blocking program flow. It should be noticed that running a task dependency graph multiple times exhibits the most basic composability, by which the same graph is encapsulated in a linear chain of tasks.

```
1  tf::Taskflow taskflow;
2
3  // Add some tasks ...
4
5  // Create an executor with 4 threads
6  tf::Executor executor {4};
7
8  // Run the taskflow object once
9  auto fut = executor.run(taskflow);
10 fut.get();
11
12 // Run the taskflow object 4 times with a callback
13 taskflow.run_n(taskflow, 4, [](){
14   std::cout << "Finish!\n";
15 }).get();
16
17 // Run the taskflow object with a predicate
18 int counter {4};
19 executor.run_until(taskflow,
20   [&](){ return --counter == 0; }
21 ).get();
```

Listing 2. Different ways to execute a task dependency graph.

D. Task Dependency Graph Composition

Task dependency graph composition is the most important feature in Cpp-Taskflow v2. It allows users to create heavily optimized task dependency graphs and reuse them to compose larger graphs and so on so forth. The tf::Taskflow class defines a method composed_of to enable composition. Specifically, the caller taskflow object adds a module task of the callee taskflow object. Listing 3 shows an example of taskflow object composition. Line 1:10 creates a taskflow object with three dependent tasks A1, A2, and A3. Line 12:19 creates another taskflow object with three tasks B1, B2, and B3. Line 22 adds a module task from the first taskflow object and line 25:27 specifies the dependency between tasks. Unlike the emplace method that creates a *regular task*, the composed_of method creates a *module task* in the graph. A module task is a special task that is aware of which taskflow object to probe during its execution context. We would like to highlight three points of our composition interface. First, there is no copy during the composition, leading to efficient graph sharing and resource utilization. We have strived to resolve many scheduling conflicts due to shared tasks, while providing a high-level execution API to completely separate this low-level controls from users. Second, recursive and nested composition are feasible. A taskflow object can be used to compose multiple taskflow objects and the resulting taskflow object can compose another taskflow object with no restriction. During the composition, user can add free-standing tasks to the graph to perform computation across different task layers. Adding dependency is extremely easy and flexible through the precede method. Finally, the module task works seamlessly with both static and dynamic tasking. This gives users a powerful and unified tasking interface to accomplish large and complex parallel workloads.

```
1  tf::Taskflow fA;
2
3  // Add three tasks
4  auto [A1, A2, A3] = fA.emplace(
5    [](){ std::cout << "Task A1\n"; },
6    [](){ std::cout << "Task A2\n"; },
7    [](){ std::cout << "Task A3\n"; }
8  );
9
10 A3.gather(A1, A2);
11
12 tf::Taskflow fB;
13
14 // Add three tasks
15 auto [B1, B2, B3] = fB.emplace(
16   [](){ std::cout << "Task B1\n"; },
17   [](){ std::cout << "Task B2\n"; },
18   [](){ std::cout << "Task B3\n"; }
19 );
20
21 // Compose taskflow object
22 auto moduleA = fB.composed_of(fA);
23
24 // Build dependency between module and regular tasks
25 B1.precede(moduleA);
26 B2.precede(moduleA);
27 moduleA.precede(B3);
```

Listing 3. Cpp-Taskflow v2 taskflow object composition code (19 LOC and 167 tokens).

At this point, we are interested in the difference between our composition code and existing libraries. Listing 4 is the implementations of Listing 3 using TBB flow graph [7]. As shown in the two listings, Cpp-Taskflow v2 has the least lines of code and is more readable compared with TBB code. To our best knowledge, TBB has no API to directly compose task graphs, so we have to capture the task graph into another task and execute the task graph. This results in longer lines of code and tends to produce bugs if one forgets to execute the task graph. This example clearly shows the conciseness and ease-of-use of the task composition interface in Cpp-Taskflow v2.

```
1  using namespace tbb;
2  using namespace tbb::flow;
3
4  graph fA;
5
6  continue_node<continue_msg> A1(fA, []
7    (const continue_msg&) {
8      std::cout << "Task A1\n";
9    }
10 );
11 continue_node<continue_msg> A2(fA, []
12   (const continue_msg&) {
13     std::cout << "Task A2\n";
```

```
  }
);
continue_node<continue_msg> A3(fA, []
  (const continue_msg&) {
    std::cout << "Task A3\n";
  }
);

make_edge(A1, A3);
make_edge(A2, A3);

graph fB;

continue_node<continue_msg> B1(fB, []
  (const continue_msg&) {
    std::cout << "Task B1\n";
  }
);
continue_node<continue_msg> B2(fB, []
  (const continue_msg&) {
    std::cout << "Task B2\n";
  }
);
continue_node<continue_msg> B3(fB, []
  (const continue_msg&) {
    std::cout << "Task B3\n";
  }
);
continue_node<continue_msg> moduleA(fB, [&]
  (const continue_msg&) {
    A1.try_put(continue_msg());
    A2.try_put(continue_msg());
    fA.wait_for_all();
  }
);

make_edge(B1, moduleA);
make_edge(B2, moduleA);
make_edge(moduleA, B3);
```

Listing 4. TBB hard-coded composition code (48 LOC and 256 tokens).

In addition to the composability, another useful feature is the modularity. Through *inheritance* from tf::Taskflow class, users can define their own task dependency graph class as a *single module*. The task dependency graph composition and execution APIs can be directly applied to the customized class as well, obviating the need of an additional wrapper.

With the composability and modularity, a complex design can be decomposed into small components with different parallel patterns. Users can implement and test those patterns individually and combine them in various ways such as nested or concatenation to deliver complex functionality. This can substantially increase programmers' productivity as it enables a structural and efficient way for software engineering.

E. Unified Task Execution

Cpp-Taskflow v2 modifies the execution kernel to enable seamless integration of the reusable and composable task dependency graph with existing task types. To make a task dependency graph reusable, it's necessary to ensure the graph remains unchanged after each execution. During runtime, a task might expand the graph by spawning new nodes to precede the parent node such as dynamic tasking. As a result, in Cpp-Taskflow v2 a task that spawns new tasks will restore its own precedence before scheduling its successor tasks. Letting each task perform the restoration on itself also minimizes the overhead.

Apart from the regular tasks, a task dependency graph can have module tasks through composing other graphs. The execution flow of module task is similar to dynamic tasking except that a module task directly dispatches the composed graph rather than a subflow. A module task will be executed twice:

- First time:
  - The executor first collects the source and sink tasks in the composed graph and let the sink tasks precede the module task.
  - The executor dispatches the source tasks to execution.
- Second time:
  - The executor clears the successor of sink tasks in composed graph.
  - The executor dispatches the module task's successors to execution.

Figure 2 is an example that illustrates scheduling a module task.

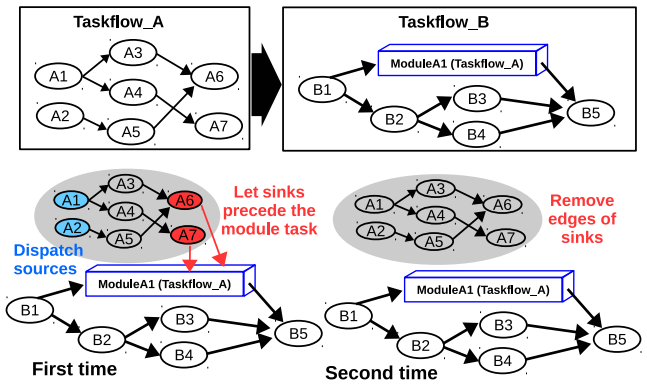

Fig. 2. An example to illustrate the execution of module task.

F. Visualize a task dependency graph with both regular and module tasks.

Cpp-Taskflow v2 provides the same APIs as Cpp-Taskflow to support visualization of task dependency graph to facilitate debugging. A taskflow object can be assigned a name by the name method and it has a dump method to export its task dependency graph in DOT language [8]. A module task is represented by a *cuboid* to differentiate from the regular tasks. Figure 3 shows an example of visualizing composed task dependency graphs.

III. Evaluation

We conduct experiments to emulate two real-world applications, a parallel machine learning hyperparameter search and a Very-large-scale integration (VLSI) circuit timing analysis. We focus on comparison with TBB as both Cpp-Taskflow v2 and TBB adopt task dependency graph for programming model. We demonstrate Cpp-Taskflow v2 achieves comparable

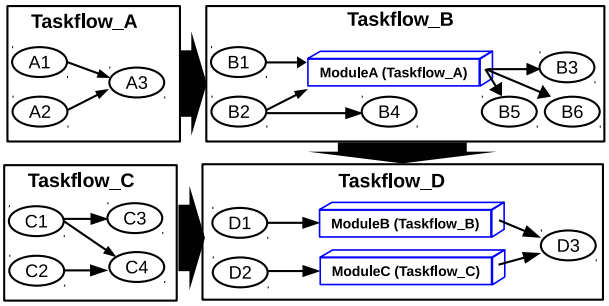

Fig. 3. Visualize the task dependency graph D with its regular and module tasks. Note the arrows between taskflow objects are added deliberately here for clarity.

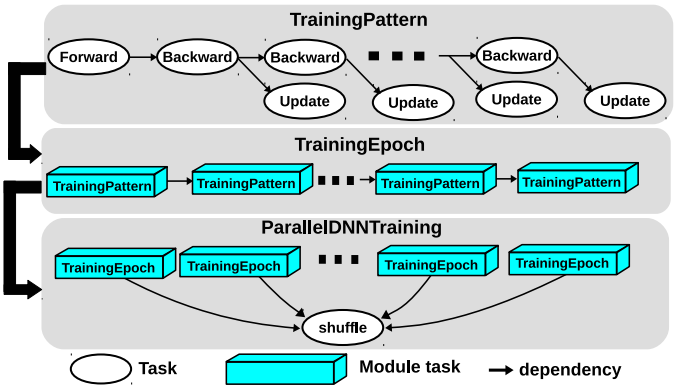

Fig. 4. Parallel DNN training through hierarchical composition.

performance with less software development effort using the task graph composition.

The experiment platform has 128 GB memory and a 2.5 GHz Intel Xeon W-2175 processor with 14 physical cores and 14 threads. The operating system is Ubuntu 18.04.2. We use the library provided APIs to control the number of threads and the system utility `taskset` to pin each thread to a specific core to minimize the migration overhead. All programs are compiled using g++-7.4 with optimization flag `O2` and C++ standard flag `-std=c++17`. We use Intel TBB 2019 Update 2 (flow graph) as the baseline for evaluation.

A. Machine Learning Application

Machine learning involves lots of computations and parallel computing plays a key role in building machine learning applications. Deep neural network (DNN) is a fundamental machine learning model and a DNN typically has many parameters to tune such as learning rate, layer number, and weight initialization [9]. Finding a good parameter set is an important topic in machine learning study and many approaches have been proposed [9] [10] [11] [12]. An intuitive way to explore the parameter space is to concurrently train multiple DNNs with different parameter sets. We emulate this process by creating a parallel DNN training framework on the MNIST dataset [13]. In this framework, we train multiple DNNs concurrently and synchronize them every epoch and shuffle the data for next epoch.

To implement the proposed framework with Cpp-Taskflow v2, we first create a `TrainingPattern` class that performs a training pass (forward/backward propagation) over a batch of data. We use the task pipelining strategy proposed by [5] in the `TrainingPattern` to enable parallelism. Then, we build a `TrainingEpoch` class to iterate through all batches by composing each `TrainingPattern` into a linearized task graph. Lastly, we gather those `TrainingPatterns` with a data shuffle task into a `ParallelDNNTraining` task graph. Figure 4 depicts the framework. For TBB, we first use the flow graph interface to build the `TrainingPattern` for each DNN. Then we create another flow graph for `ParallelDNNTraining`. The `ParallelDNNTraining` flow graph explicitly captures a `TrainingPattern` in a node and launches the training during execution.

We train ten DNNs concurrently with each DNN has five layers (784x64x32x16x8x10). For simplicity, we adopt gradient descent optimization and set the learning rate of each DNN to 0.0001. Table I is the code complexity analysis reported by Lizard [14]. The lines of code (NLOC) of Cpp-Taskflow v2's implementation is 22% shorter than the TBB's. There are two reasons for this: first TBB uses template syntax for task construction and second we need to explicitly dispatch the composed graph to execution in TBB's implementation.

TABLE I
CODE COMPLEXITY ANALYSIS OF THE PARALLEL DNN TRAINING FRAMEWORK.

Library	NLOC (total)	CCN (avg)	Token (avg)
Cpp-Taskflow v2	60	2.0	90.6
TBB	77	2.8	125.0

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

We use 10 cores in this experiment and train those DNNs from 10 to 100 epochs. Figure 5 plots the two libraries' runtime of the parallel DNN training. Both libraries exhibit a similar trend in runtime growth when increasing the number of epochs and Cpp-Taskflow v2 outperforms TBB in all cases.

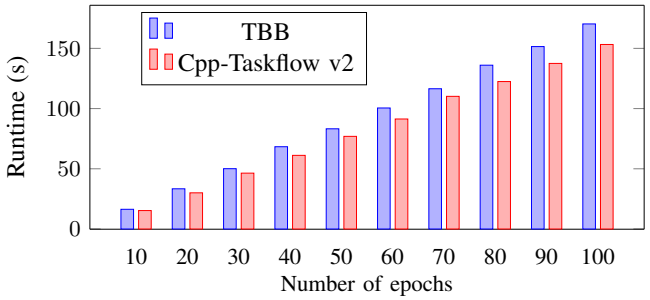

Fig. 5. Runtime of training 10 DNNs using 10 cores in parallel.

B. VLSI Timing Analysis

The second application is VLSI timing analysis. Timing analysis is an important step in circuit design as a circuit

must meet the timing requirements before tape-out [15] [16]. Timing analysis contains different workloads such as timing simulation and correlation, and each workload can be managed either by a tool or a library. In order to run a complete timing analysis, users have to integrate libraries into their programs and write hard-coded scripts to compose those tools. Obviously, hard-coded script is not flexible and is very difficult to scale up to complex control flow. In this experiment, we demonstrate using Cpp-Taskflow v2 to quickly compose those tools together to emulate the timing analysis flow. The idea is to encapsulate each tool in a task and those tasks use process fork to execute the tools. Our method can shorten the analysis runtime by letting tools run concurrently and can also effectively handle the complex control flow.

Figure 6 depicts the timing analysis flow. In the process framework, we launch multiple timers with each running the timing analysis for a scenario[2] and dumping the top 100 critical paths to disk. After timers finish, those paths are read into memory by a reader task. A simulation framework composes multiple process frameworks to parallelize the path reading. To find the correlation between each pair of scenarios using those critical paths, we create a scenario task to calculate the correlation coefficients between a scenario and others. Those scenario tasks are gathered into the scenario framework. Lastly, we build a timing analysis framework by composing the simulation framework and the scenario framework.

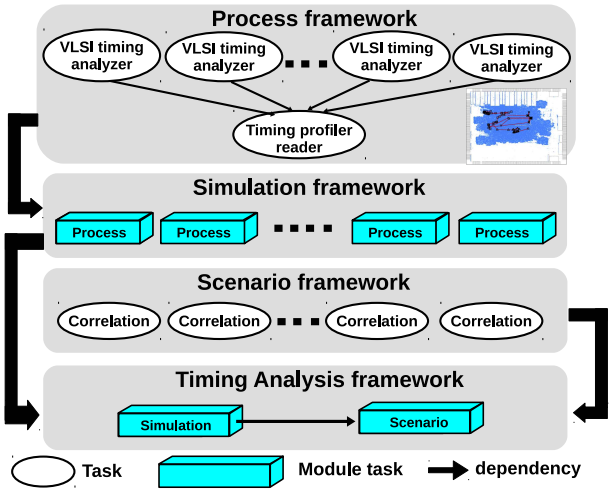

Fig. 6. A parallel VLSI timing analysis framework.

We create individual C++ struct for the first three frameworks by inheriting tf::Taskflow and build the timing analysis framework in a single function (top) using these frameworks. We use OpenTimer [16], an open-source VLSI timing analyzer, to perform static timing analysis (STA). Table II shows the code complexity of each implementation measured by Lizard. The results show that Cpp-Taskflow v2 takes fewer lines of code than TBB.

Next for performance profiling, we create 1024 design constraints with each representing a scenario. For each scenario we launch a timer to analyze the industrial circuit tv80 (5.3k gates and 5.3k nets) [17] and group 8 timers in a process framework. Figure 7 shows the results of scaling from 1 core to 14 cores. Both TBB's and Cpp-Taskflow v2's runtimes exhibit the same scaling and are very close in all cases.

[2]In this experiment, a scenario denotes a different design constraint.

TABLE II
CODE COMPLEXITY ANALYSIS OF THE TIMING ANALYSIS FLOW. THE FIRST TABLE IS THE WHOLE FILE AND THE SECOND TABLE IS FOR INDIVIDUAL FRAMEWORK.

Library	NLOC (total)	CCN (avg)	Token (avg)
Cpp-Taskflow v2	61	3.0	132
TBB	104	2.6	106.6

Framework	NLOC (total)		CCN (avg)		Token (avg)	
	TF	TBB	TF	TBB	TF	TBB
Process	10	14	2	2	118	155
Simulation	5	9	2	2	47	63
Scenario	15	16	4	4	137	155
Top	23	29	4	4	226	283

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

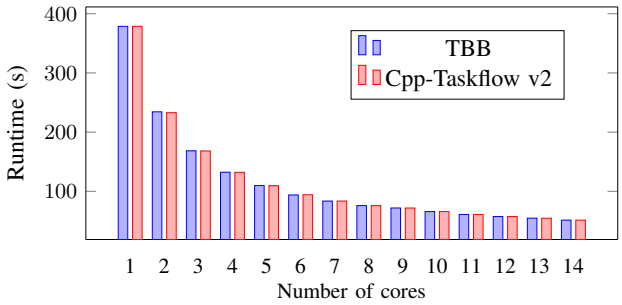

Fig. 7. Runtime comparisons of the proposed VLSI timing analysis flow on circuit tv80 using TBB and Cpp-Taskflow v2

IV. Conclusions

This paper presents Cpp-Taskflow v2, a parallel programming library on composable task graph. Cpp-Taskflow v2 introduces a task graph composition interface to enable composable parallel programming. Users can use the composition interface to quickly build a large parallel program through composing modular and reusable task graphs. The experimental results show that Cpp-Taskflow v2 can achieve comparable performance to Intel Threading Building Blocks with fewer lines of code. Future work will focus on extension to heterogeneous computing, especially under the mixed workload of CPUs and GPUs.

V. Acknowledgement

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

References

[1] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *NIPS*, pages 2503–2511, 2015.

[2] Eduard Ayguad and Daniel Jimnez-Gonzlez. An approach to task-based parallel programming for undergraduate students. *J. Parallel Distrib. Comput.*, 118(P1):140–156, August 2018.

[3] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *J. Supercomput.*, 74(4):1422–1434, April 2018.

[4] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. Evaluating end-to-end optimization for data analytics applications in weld. *VLDB*, 11(9):1002–1015, 2018.

[5] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS*, pages 974–983, 2019.

[6] Cpp-Taskflow. https://github.com/cpp-taskflow/cpp-taskflow.

[7] Intel Threading Building Blocks. [online]. available: https://www.threadingbuildingblocks.org/intel-tbb-tutorial.

[8] The DOT Language. https://www.graphviz.org/.

[9] Y. Bengio. Practical recommendations for gradient-based training of deep architectures. *In: Montavon G., Orr G.B., Muller KR. (eds) Neural Networks: Tricks of the Trade, Lecture Notes in Computer Science*, 7700, 2012.

[10] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[11] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 2951–2959, USA, 2012.

[12] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, Department of Computer Science, National Taiwan University, 2010.

[13] MNIST. [online]. available: http://yann.lecun.com/exdb/mnist/.

[14] Lizard. [online]. available: http://www.lizard.ws/.

[15] Tsung-Wei Huang and Martin D. F. Wong. UI-Timer 1.0: An ultrafast path-based timing analysis algorithm for cppr. *IEEE TCAD*, 35(11):1862–1875, Nov 2016.

[16] Tsung-Wei Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *IEEE/ACM ICCAD*, pages 895–902, 2015.

[17] J. Hu, G. Schaeffer, and V. Garg. Tau 2015 contest on incremental timing analysis. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 882–889, Nov 2015.