Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale

Tsung-Wei Huang, Yibo Lin, *Member, IEEE*, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong, *Fellow, IEEE*

***Abstract*—This article introduces Cpp-Taskflow, a high-performance parallel task programming system, to streamline the building of large and complex parallel applications. Cpp-Taskflow leverages the power of modern C++ and task-based approaches to enable efficient implementations of parallel decomposition strategies. Our programming model can quickly handle not only traditional loop-level parallelism but also irregular patterns, such as graph algorithms and dynamic control flows. Compared with existing libraries, Cpp-Taskflow is more cost efficient in performance scaling and software integration. We have evaluated Cpp-Taskflow on both micro-benchmarks and large-scale design automation problems of million-scale tasking. In a particular timing analysis workload, Cpp-Taskflow outperformed OpenMP by 2× faster using 2× fewer lines of code. We have also shown Cpp-Taskflow achieved up to 47.81% speed-up with 28.5% less code over the industrial-strength library, Intel Threading Building Blocks, on a detailed placement problem.**

***Index Terms*—Computer-aided design (CAD), parallel programming systems, task parallelism.**

I. Introduction

DEVELOPING high-performance parallel computer-aided design (CAD) software is an extremely challenging job. CAD algorithms consist of a broad mix of domain knowledge, irregular compute patterns, layered heuristics, expert-level parameter tuning, and many computing components that are not part of regular software development. Existing ad-hoc attempts at parallelizing CAD applications focus on incrementally recoding existing CAD software using portable operating system interface (POSIX) threads, OpenMP, or C/C++11 thread libraries [2]. While results of some of those efforts have shown scalability, most are not scaling beyond a few threads, and some CAD tools have not been replaced with more scalable equivalents at all. Neither parallel programming models nor runtimes, despite some improvements, are mature enough to allow CAD engineers to migrate their applications to parallel targets in a timely manner [1], [3], [4]. The growing computational complexity will soon far exceed what existing ad-hoc approaches will be able to achieve. In order to continue scaling application performance, we must find a new solution that allows developers to efficiently rearchitect CAD software to discover and express high degrees of parallelism.

Considering the long-term goal of enabling CAD amenable to parallelization, the Cpp-Taskflow project addresses the question of "how can we make it easier for C++ developers to quickly write parallel programs with high performance and simultaneous high productivity?" Through the evolution of parallel programming standards, *task-based* model has been proven to pave the path to scale up with future processor generations and architectures [5]. The traditional *loop-based* parallelism is not sufficient for exploiting the scalability of modern CAD tools and complex parallel algorithms that require irregular compute patterns, such as graph traversal and dynamic control flows [6]. For many C++ developers, writing a correct and efficient task parallel program is challenging, not only because of the capability of a tasking library but also its productivity to express a *parallel computation task graph*. Programmability of a library has large impact on the way C++ developers organize their parallel code, from subtle implementation details to algorithm-level decisions of parallel decomposition strategies [7]. However, related research remains nascent, particularly on the front of using modern C++ to enhance the functionality and performance productivity that were previously not possible.

Consequently, we introduce Cpp-Taskflow, a general-purpose parallel task programming system to help C++ developers quickly write parallel applications using task dependency graphs [8]. Listing 1 demonstrates a simple Cpp-Taskflow program. The code *explains itself*. The program creates a task dependency graph of four tasks, A, B, C, and D. Each task is a callable object which can be a C++ lambda, a function object, or a binding expression. The dependency constraints state that task A runs before task B and task C, and task D runs after task B and task C. There is no explicit thread managements or complex lock controls in the code.

The design principle of Cpp-Taskflow is to let users write *simple* and *efficient* parallel code. What we advocate here is *expressive*, *readable*, and *transparent* code that scales to large

Manuscript received January 22, 2020; revised June 12, 2020 and August 16, 2020; accepted September 6, 2020. Date of publication September 18, 2020; date of current version July 19, 2021. This work was supported in part by NSF under Grant CCF-1718883, and in part by DARPA under Grant FA-650-18-2-7843. Preliminary version of this paper has been presented at the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS'19), Rio de Janeiro, Brazil, May 2019 [1]. This article was recommended by Associate Editor S. Held. *(Corresponding author: Tsung-Wei Huang.)*

Tsung-Wei Huang is with the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT 84112 USA (e-mail: twh760812@gmail.com).

Yibo Lin is with the Department of Computer Science, Peking University, Beijing 100871, China.

Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong are with the ECE Department, University of Illinois at Urbana–Champaign, Urbana, IL 61801 USA, on leave from Chinese University of Hong Kong, Hong Kong.

Digital Object Identifier 10.1109/TCAD.2020.3025075

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

auto [A, B, C, D] = taskflow.emplace(
  [] () { std::cout << "Task A\n"; },
  [] () { std::cout << "Task B\n"; },
  [] () { std::cout << "Task C\n"; },
  [] () { std::cout << "Task D\n"; }
);

A.precede(B, C);   // A runs before B and C
D.succeed(B, C);   // D runs after  B and C

executor.run(tf).wait();
```

Listing 1. Simple task dependency graph in Cpp-Taskflow.

number of cores. Users write a parallel program in terms of *tasks* rather than *threads* and they do not need to deal with any hardware or system details. Cpp-Taskflow explores a minimum set of core routines that are sufficient enough for users to implement a broad set of parallel decomposition strategies such as parallel loops, graph algorithms, and dynamic flows. We leverage the power of modern C++ to strike a balance between performance and usability of our application programming interface (API). Our API is not only flexible on the user front but is also extensible with the evolution of future C++. We summarize our contributions as follows.

1) *Programming Model:* Cpp-Taskflow developed a simple and efficient parallel task programming model using modern C++17. Developers can leverage powerful modern C++ features and standard libraries together with our parallelization framework to implement fast and scalable parallel programs. Our user experiences lead us to believe that while it requires some effort to learn, a C++ programmer can master our APIs and apply Cpp-Taskflow to his/her jobs in just a few hours.
2) *Transparency:* Cpp-Taskflow is transparent. Users need no understanding of standard concurrency controls, such as thread managements and lock mechanisms, which are difficult to program correctly. Instead, we offer a lightweight abstraction for users to focus on high-level developments and leave system details to Cpp-Taskflow.
3) *Composition:* Cpp-Taskflow is composable. Users can create large parallel graphs through composition of modular and reusable blocks that are easier to optimize at an individual scope. The program runs on a multicore machine with automatic scheduling optimization across different layers of composed task graphs.
4) *Unified Interface:* Cpp-Taskflow has a unified graph description model that empowers developers with both static and dynamic graph constructions and refinement to fully exploit task parallelism. The same API used for static tasking all applies to dynamic tasking. Programmers need not to learn a different API set. The unified interface makes it easier to develop and debug complex parallel decomposition strategies.

We have evaluated Cpp-Taskflow on both micro-benchmarks and real-world applications. The performance

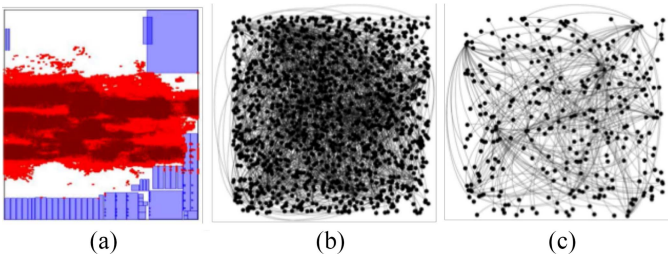


Fig. 1. Timing graph of an industrial design [11]. (a) Circuit (1.01 mm^2. (b) Graph (3M gates). (c) Signal path.

scales from a single processor to multiple cores with millions of tasks. We believe Cpp-Taskflow stands out as a unique tasking library considering the ensemble of software tradeoffs and architecture decisions we have made. Cpp-Taskflow is open-source and is being used by many industrial and academic research projects [8]. That being said, different systems have their pros and cons, and deserve a particular reason to exist. We would like to position Cpp-Taskflow as a higher-level alternative to help streamline the building of large and complex parallel applications.

II. Project Motivation

Cpp-Taskflow is motivated by our research project on developing a high-performance timing analysis tool for very large scale integration (VLSI) systems [9]. Timing analysis is a very important component in the overall design flow [10]. It verifies the expected timing behaviors of a digital circuit to ensure correct functionalities after tape-out. During the chip design flow, the timing analyzer is used as an inner loop of an optimization algorithm to *iteratively* and *incrementally* improve the timing of a circuit layout. The optimization engine typically applies millions of design transforms to modify the design both locally and globally, and the timer has to quickly update the timing information to guarantee slack integrity. However, today's circuit is very large and is made up of billions of transistors. Fig. 1 shows an analysis graph on an industrial design of 2M gates [11]. Timing a million-gate circuit can take several hours or days for sign-off. Analysis loops require fairly expensive computations and must take advantage of multicore to speed up the runtime.

A. Challenge 1: Complex Task Dependencies

The biggest challenge to write a parallel timing analyzer is the large and complex task dependencies. In order to construct a timing graph, we need to collect a number of information, such as load capacitance, slew, delay, and arrival time. However, these quantities are dependent of each other and are expensive to compute. The resulting task dependency in terms of encapsulated function calls is very complex. For example, we need to sweep the graph to *forward* propagate the arrival time at each pin and then *backward* propagate the required arrival time at each pin. Then we construct the timing graph based on the two timing metrics and perform path searching on top of the graph. In order to obtain the arrival time at a pin, we need to collect a number of timing information, such as load capacitance, slew, and delay.

Also, we cannot start the backward propagation until all computations in the forward phase finish. The resulting task dependency in terms of encapsulated function calls is very complex. For example, in a million-gate circuit design, the graph can encounter billions of tasks and dependencies. In fact, many workloads in the VLSI domain are more connected and complex than that of social media and scientific computing [11].

B. Challenge 2: Irregular Compute Pattern

Updating a timing graph involves extremely irregular memory patterns and significant diverse behavior across different computations. The task programming model must be flexible for both regular and irregular blocks, whether the data is structured in local blocks or is flat in the global scope. We must be able to capture different data representations inside a task, for carrying out different timing propagation algorithms and pruning heuristics. Exploiting parallelism from this type of problem formulation is very different from data-oriented parallel computing, such as linear algebra, scientific computing, and single instruction multiple data (SIMD)-style parallelization. Instead, the flexibility to encapsulate a variety of timing workloads into a task unit plays a key role in developing efficient parallel decomposition strategies.

C. Challenge 3: Dynamic Control Flows

Optimization and physical synthesis engines often call an *incremental* timer millions of times in their inner loop. For large designs, the process can take several hours or days to finish. To mitigate the long runtime, the timing analyzer needs to *incrementally* answer timing queries after one or more changes to the circuit were made. The process is highly iterative and *unpredictable*, and consists of many *dynamic*, *cyclic*, and *conditional* workloads that cannot be foreseen in static graph constructions.

D. State-of-the-Art Solutions and Their Bottleneck

Almost all existing timing analyzers, including both academic and industrial tools, were written in C++ using *loop-based* parallelization [9]–[12]. A common approach *levelizes* the circuit graph into a topological order, and applies language-specific "parallel_for" level by level (see Fig. 2). For each node, we need to update a number of dependent tasks, such as slew, parasitics, delay, and required arrival time [9]. This level-based decomposition strategy is advantageous in its simple pipeline concept. We can easily apply parallel_for to each level and update the timing of nodes at the same level in parallel. However, this suffers from many performance pitfalls. For example, the number of nodes can vary at different levels, and the resulting thread utilization can be significantly unbalanced. Also, there is a synchronization barrier between successive levels to model the task dependencies. The overhead is large for graph with long data paths.

Two mainstream library choices to implement this loop-based decomposition strategy are OpenMP task dependency clause and Intel Threading Building Blocks (TBBs) FlowGraph [13], [14]. However, there are many limitations

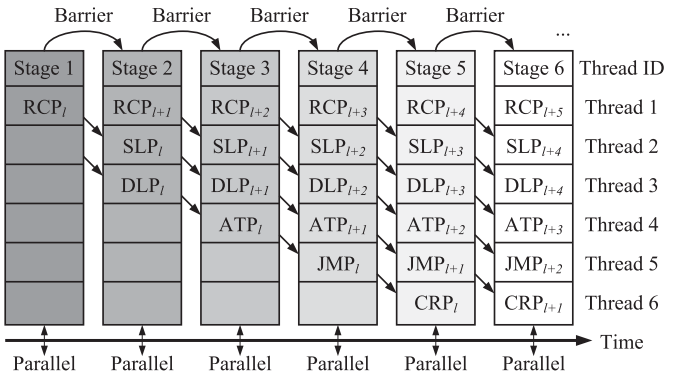


Fig. 2. Loop-based parallel timing propagations. Each level applies a "parallel_for" to update timing from the fanin of each node [9].

in using these libraries. For example, OpenMP relies on *static* task annotations with a valid order in line with a sequential execution, making it very difficult to handle dynamic flows where the graph structure is unknown at programming time. TBB is disadvantageous mostly from an ease-of-programming standpoint. Its task graph description language is very complex and often results in large source lines of code (LOC) that are hard to read and debug. There are libraries designed for graph processing [15], [16]. However, most of them are restrictive to graph problems and cannot capture or pipeline generic tasks beyond the graph formulation. After many years of research, we and our industry partners conclude the biggest hurdle to a scalable parallel timing analyzer is a suitable *parallel task programming model*. Inspired by our problem domains, we are interested in million-scale tasks with runtime in the order of seconds to minutes. We focus on C++ on a shared memory architecture.

III. CPP-TASKFLOW

While Cpp-Taskflow was initiated to support our VLSI projects, we decided to disclose its knowledge and make it an open-source tasking library for generic parallel programming [8]

> Cpp-Taskflow aims to help C++ developers quickly write high-performance parallel programs with simultaneous high productivity using task models.
>
> — Cpp-Taskflow's Project Mantra

A. Create Task

Cpp-Taskflow is *object-oriented*. A task in Cpp-Taskflow is defined as A task can be either a functor, a lambda expression, a bind expression, or other class objects that define the operator `()`. Listing 2 demonstrates the creation of a task in Cpp-Taskflow. The first entry to a Cpp-Taskflow program is declaring a taskflow object of type `tf::Taskflow`. A taskflow object is where to create a *task dependency graph*. The method `emplace` creates a task from a given callable object. Users can create one or multiple tasks at one time using C++ structured binding.

Each time users create a task, the taskflow object adds a node to the present graph and returns a *task handle*. A task

```
tf::Taskflow taskflow;

auto [X, Y, Z] = taskflow.emplace(
  [] () { std::cout << "Task X\n"; },
  [] () { std::cout << "Task Y\n"; },
  [] () { std::cout << "Task Z\n"; }
);
```

Listing 2. Create three tasks from a taskflow object.

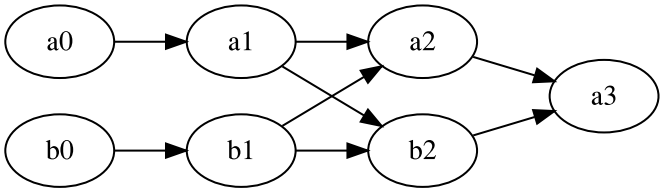


Fig. 3. Static task dependency graph of seven tasks and eight dependency constraints.

handle is a lightweight class objects that wraps up a particular node in a graph. Adding this layer of abstraction provides an extensible mechanism to modify the task attributes and prevents users from direct access to the internal graph storage. Each node has a general-purpose polymorphic function wrapper to store and invoke any callable target (task) given by users. Hereafter, we use "task A" to represent the task stored in node A. A task handle can be empty, often used as a placeholder when it is not associated with a node. This is particularly useful when the callable target cannot be decided until some points at the program, while we need to preallocate a storage for the task in advance.

B. Static Tasking

After tasks are created, the next step is to add dependencies. A task dependency is a *directed* edge from one task A to another task B such that task A runs before task B. To be more specific, node B will not invoke its task until node A finishes its task. Cpp-Taskflow defines two very intuitive methods, `precede` and `succed` for users to create a dependency from one task to another. The most basic graph concept in Cpp-Taskflow is *static tasking*. Static tasking captures the static parallel structure of a decomposition strategy and is defined only by the program itself. It has a flat task hierarchy and cannot spawn new tasks from a running dependency graph.

Fig. 3 shows an example of static task dependency graph and Listing 3 gives its implementation in Cpp-Taskflow. The task dependency graph consists of seven tasks and eight dependencies. Task a0 runs before task a1, task a1 runs before task a2 and task b2, and task a2 runs before task a3. On the other side, task b0 runs before task b1, task b1 runs before task a2 and task b2, and task b2 runs before task a3. These task dependencies are described intuitively using the methods `precede` and `succeed` from individual task handles. We implemented `precede` using C++ *function parameter pack*, allowing users to write multiple dependencies at one time. Listings 4 and 5 demonstrate the counterparts written in OpenMP task dependency clause and TBB FlowGraph. While analyzing programmability is a very complex procedure, we believe in this example Cpp-Taskflow is more concise and

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

auto [a0, a1, a2, a3, b0, b1, b2]
  = taskflow.emplace(
  [] () { std::cout << "a0\n"; },
  [] () { std::cout << "a1\n"; },
  [] () { std::cout << "a2\n"; },
  [] () { std::cout << "a3\n"; },
  [] () { std::cout << "b0\n"; },
  [] () { std::cout << "b1\n"; },
  [] () { std::cout << "b2\n"; },
);

a0.precede(a1);
a1.precede(a2, b2);
b0.precede(b1);
b1.precede(a2, b2);
a3.succeed(a2, b2);

executor.run(taskflow).wait();
```

Listing 3. Cpp-Taskflow code of Fig. 3 (17 LOC and 183 tokens).

effective than the others. In terms of LOC, the task dependency graph takes only 17 lines of Cpp-Taskflow code but 22 and 37 lines for OpenMP and TBB, respectively. Compared with OpenMP, programmers need to explicitly specify the dependency clause on both sides of a constraint. For instance, we declare eight variables (`a0_a1`, `a1_a2`, `a1_b2`, `a2_a3`, `b0_b1`, `b1_b2`, `b1_a2`, and `b2_a3`) and place each in the the depend clause to capture the eight dependent links in the task graph. This hard-coded list makes it very inflexible for programmers to modify the graph as it requires another hand-written clause list. Also, it is users' responsibility to identify a correct topological order to describe each task such that it is consistent with the sequential program flow. For example, the `#pragma task` block for a1 cannot go above a0. Otherwise, the program can produce unexpected results when OpenMP is disabled in a different compiler setting. On the other hand, the TBB-based implementation is quite verbose. Programmers need to understand the complex template class `continue_node` and the role of the message class before starting with a simple task dependency graph. To run a flow graph, users need to explicitly tell TBB the *source* tasks and call the method `try_put` to either enable a nominal message or an actual data input. All these add up to extra programming efforts.

C. Dynamic Tasking

In contrast to static tasking, *dynamic tasking* refers to the creation of a task dependency graph at runtime or, more specifically, in the execution context of a task. Dynamic tasks are created from a running graph. These tasks are spawned from a parent task and are grouped together to form a task dependency graph called *subflow*. Dynamic tasking is useful when the graph is not able to draw before executing a task or a program. We applied `std::variant` to our polymorphic function wrapper and exposed the same task building blocks to users for both static and dynamic graph constructions. The

```
#pragma omp parallel
{
  #pragma omp single
  {
    int a0_a1, a1_a2, a1_b2, a2_a3;
    int b0_b1, b1_b2, b1_a2, b2_a3;

    #pragma omp task depend(out: a0_a1)
    std::cout << "a0\n";

    #pragma omp task depend(out: b0_b1)
    std::cout << "b0\n";

    #pragma omp task depend(in: a0_a1) depend(
        out: a1_a2, a1_b2)
    std::cout << "a1\n";

    #pragma omp task depend(in: b0_b1) depend(
        out: b1_b2, b1_a2)
    std::cout << "b1\n";

    #pragma omp task depend(in: a1_a2, b1_a2)
        depend(out: a2_a3)
    std::cout << "a2\n";

    #pragma omp task depend(in: a1_b2, b1_b2)
        depend(out: b2_a3)
    std::cout << "b2\n";

    #pragma omp task depend(in: a2_a3, b2_a3)
    std::cout << "a3\n";
  }
}
```

Listing 4. OpenMP code of Fig. 3 (22 LOC and 181 tokens).

same methods defined for static tasking are all applicable for dynamic tasking. Programmers do not need to learn a different API set to create dynamic workloads.

Listing 6 demonstrates Cpp-Taskflow's implementation on a dynamic task dependency graph in Fig. 4. The task dependency graph has four static tasks, A, C, D, and B. The precedence constraints force task A to run before tasks B and C, and task D to run after tasks B and C. During the execution of task B, it spawns another task dependency graph of three tasks B1, B2, and B3 (marked as cyan), where task B1 and task B2 run before task B3. In Cpp-Taskflow, tasks B1, B2, and B3 are grouped to a subflow parented at task B. We allow users to describe this dynamic dependencies using the same method `emplace`, with one additional argument of type `tf::Subflow` that will be created by the taskflow object at runtime passing to task B. A subflow builder is a lightweight object that inherits all graph building blocks from static tasking. By default, a spawned subflow joins its parent task. This forces a subflow to follow the subsequent dependency constraints of its parent task. Depending on applications, users can detach a subflow from its parent task using the method `detach`, allowing its execution to flow independently. A detached subflow will eventually join the end of the topology of its parent task.

Listings 6 and 7 compare two implementations of Fig. 4 using Cpp-Taskflow and TBB. In a rough view, Cpp-Taskflow has the least amount of code (19 versus 38). Our user

```
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> a0(g, [] (const
    continue_msg &) {
  std::cout << "a0\n";
});
continue_node<continue_msg> a1(g, [] (const
    continue_msg &) {
  std::cout << "a1\n";
});
continue_node<continue_msg> a2(g, [] (const
    continue_msg &) {
  std::cout << "a2\n";
});
continue_node<continue_msg> a3(g, [] (const
    continue_msg &) {
  std::cout << "a3\n";
});
continue_node<continue_msg> b0(g, [] (const
    continue_msg &) {
  std::cout << "b0\n";
});
continue_node<continue_msg> b1(g, [] (const
    continue_msg &) {
  std::cout << "b1\n";
});
continue_node<continue_msg> b2(g, [] (const
    continue_msg &) {
  std::cout << "b2\n";
});

make_edge(a0, a1);
make_edge(a1, a2);
make_edge(a1, b2);
make_edge(a2, a3);
make_edge(b0, b1);
make_edge(b1, b2);
make_edge(b1, a2);
make_edge(b2, a3);

a0.try_put(continue_msg());
b0.try_put(continue_msg());

g.wait_for_all();
```

Listing 5. TBB code of Fig. 3 (37 LOC and 295 tokens).

feedbacks lead us to believe that our dynamic tasking ends up being cleaner and more expressive [8]. The subflow spawned from a task belongs to the same graph of its parent task. Users do not need to create a separate graph object to spawn dynamic tasks as in TBB. While it is arguable which paradigm is better, we have found it simpler and safer to stick with the same graph, especially from the debugging aspect or when a subflow goes nested or recursive.

A powerful feature of Cpp-Taskflow's dynamic tasking is *nested* subflow. Users can create a subflow inside a subflow and so on to perform recursive compute patterns. Fig. 5 shows a two-layer subflow example and Listing 8 demonstrates its

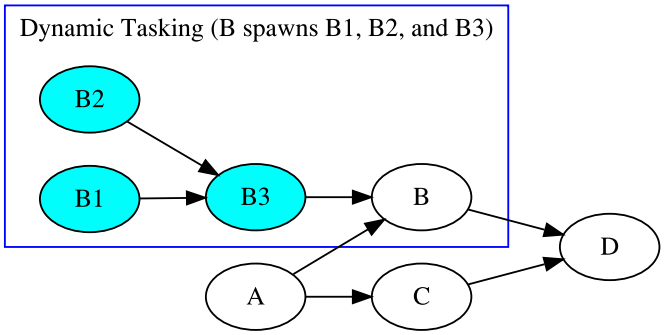


Fig. 4. Dynamic task dependency graph of four static tasks (A, B, C, and D) and three dynamic tasks (B1, B2, and B3).

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

auto [A, C, D] = taskflow.emplace(
  [] () { std::cout << "A\n"; },
  [] () { std::cout << "C\n"; },
  [] () { std::cout << "D\n"; }
);

auto B = taskflow.emplace([] (auto& sf) {
  std::cout << "B\n";
  auto [B1, B2, B3] = sf.emplace(
    [] () { std::cout << "B1\n"; },
    [] () { std::cout << "B2\n"; },
    [] () { std::cout << "B3\n"; }
  );
  B3.succeed(B1, B2);
});

A.precede(B, C);
D.succeed(B, C);

executor.run(taskflow).wait();
```

Listing 6. Cpp-Taskflow code of Fig. 4 (19 LOC and 185 tokens).

implementation in Cpp-Taskflow. The static task A spawns a subflow consisting of task A1 and task A2 at runtime. Then, the task A2 spawns another subflow consisting of two tasks A2_1 and A2_2 which join task A2. Our API to express this nested flow is very straightforward as a result of our unified interface for both static and dynamic tasking. Users can easily exploit fine-grained task parallelism using nested subflow to carry out dynamic parallel decomposition strategies, such as pruning heuristics and recursive flows.

D. Task Graph Composition

Composability is a key component to improve programmers' productivity in writing large-scale parallel applications. Cpp-Taskflow allows users to compose a large graph from small and modular parallel building blocks that can be distributed to different layers of optimization [17]. Fig. 6 demonstrates a two-level hierarchy of graph composition. Its implementation in Cpp-Taskflow is shown in Listing 9. This example consists of two graphs, G1 and G2, where G1 consists of four tasks and four dependencies, and G2 is composed of G1 and other three tasks. The method `composed_of` creates

```
using namespace tbb;
using namespace tbb::flow;

int n = task_scheduler_init::
    default_num_threads();
task_scheduler_init init(n);

graph G;  // create an outer graph

continue_node<continue_msg> A(G, [] (const
    continue_msg&) {
  std::cout << "A\n";
});
continue_node<continue_msg> C(G, [] (const
    continue_msg&) {
  std::cout << "C\n";
});
continue_node<continue_msg> D(G, [] (const
    continue_msg&) {
  std::cout << "D\n";
});
continue_node<continue_msg> B(G, [](const
    continue_msg&) {
  std::cout << "B\n";

  graph subgraph;  // create an inner graph
  continue_node<continue_msg> B1(subgraph, []
      (const continue_msg&) {
    std::cout << "B1\n";
  });
  continue_node<continue_msg> B2(subgraph, []
      (const continue_msg&) {
    std::cout << "B2\n";
  });
  continue_node<continue_msg> B3(subgraph, []
      (const continue_msg&) {
    std::cout << "B3\n";

  });

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

  B1.try_put(continue_msg());
  B2.try_put(continue_msg());
  subgraph.wait_for_all();
});

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();
```

Listing 7. TBB code of Fig. 4 (38 LOC and 299 tokens).

a *module* task from an existing taskflow. A module task does not owns or copies the taskflow but maintains a soft mapping to a taskflow. Users create dependencies on a module task in the same way as other tasks. Our runtime will expand the associated taskflow graph with a module task and executes all dependent tasks in all hierarcies. Similar to dynamic tasking, our composition can be recursive and nested. This can largely save space of tasks required for building graphs with repetitive compute patterns.

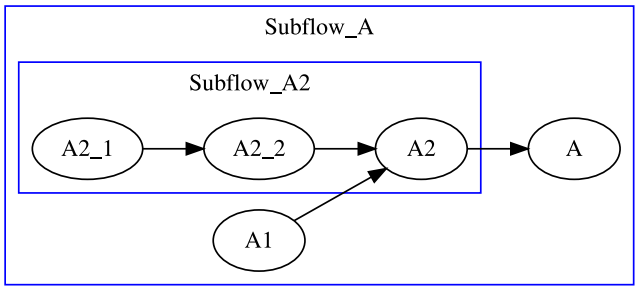


Fig. 5. Two-layer nested subflow (A and A2).

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

  auto A2 = sf.emplace([](auto& sf2){
    std::cout << "A2 spawns A2_1 & A2_2\n";
    auto A2_1 = sf2.emplace([] () {
      std::cout << "subtask A2_1\n";
    }).name("A2_1");

    auto A2_2 = sf2.emplace([] () {
      std::cout << "subtask A2_2\n";
    }).name("A2_2");

    A2_1.precede(A2_2);
  }).name("A2");

  A1.precede(A2);
}).name("A");
```

Listing 8. Cpp-Taskflow code of Fig. 5.

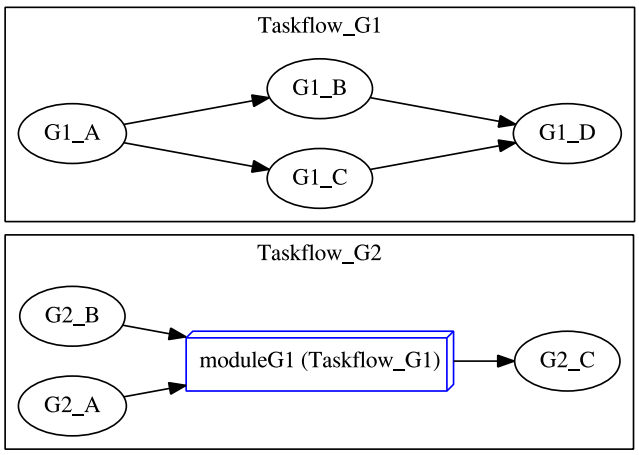


Fig. 6. Composition of task graphs.

E. Executor

Cpp-Taskflow separates the task graph construction and execution in different classes. To execute a task graph, users declare an executor object of type `tf::Executor` and submit the task graph to the executor. An executor object manages a set of *worker threads* to schedule and execute dependent tasks. By default, Cpp-Taskflow uses `std::thread::hardware_concurrency` to decide the number of worker threads. In most cases, the number of workers is equal to the number of logical cores on the machine. Users can also specify the number of workers to construct an executor. As shown in Listing 10, a task graph can be submitted to an executor in many ways. The simplest method is to execute a graph once via the `run` method. Users can also specify the number of executions via `run_n` or a stopping binary predicate via `run_until`. All methods accept an optional callback in the last argument to invoke after the execution completes, and return a `std::future` for users to access the execution status or create further continuation. Issuing multiple runs on the same taskflow automatically synchronize to a sequential chain of executions in the order of run calls. The method `wait_for_all` blocks the executor until all associated tasks finish. Executor is *thread-safe*. Touching an executor from multiple threads is acceptable. Most applications need only one executor to which multiple threads submit different task graphs each representing a particular parallel decomposition of an algorithm.

By default, the executor employs *work-stealing* to schedule and distribute tasks across CPU cores. Upon construction, the executor spawns multiple worker threads. Each worker iteratively drains out the tasks from its local queue and transitions to a *thief* to steal a task from a randomly selected

```
tf::Taskflow G1;
auto [G1_A, G1_B, G1_C, G1_D] = G1.emplace(
  [] () { std::cout << "G1_A\n"; },
  [] () { std::cout << "G1_B\n"; },
  [] () { std::cout << "G1_C\n"; },
  [] () { std::cout << "G1_D\n"; }
);
G1_A.precede(G1_B, G1_C);
G1_D.succeed(G1_B, G1_C);

tf::Taskflow G2;
auto [G2_A, G2_B, G2_C] = G2.emplace(
  [] () { std::cout << "G2_A\n"; },
  [] () { std::cout << "G2_B\n"; },
  [] () { std::cout << "G2_C\n"; }
);
auto module_G1 = G2.composed_of(G1);
module_G1.succeed(G2_B, G2_A);
module_G1.precede(G2_C);
```

Listing 9. Cpp-Taskflow code of Fig. 6.

```
tf::Executor executor;

std::future<void> fu = executor.run(flow);
fu.wait();

executor.run_n(taskflow, 3);

executor.run_until(taskflow,
  [itr=0] () mutable { return itr++ == 5; }
);

executor.wait_for_all();
```

Listing 10. Task graph execution methods in Cpp-Taskflow.

peer called *victim*. When a task completes, it submits new tasks from its immediate successors whichever dependencies are met. The scheduler iterates this procedure until the program terminates or no tasks are available. Work-stealing is advantageous in decentralized controls over threads, providing improved performance by instantaneously balancing the load among parallel processing units [18]. In short, our work-stealing algorithm maintains an invariant of that at least one thief is always running when an worker is actively running a task. At the same time, we continuously *adapt* the number of thieves above this lower limit to the available tasks to avoid over- and under-subscription of thread resources.

F. Visualization

Cpp-Taskflow provides a built-in support for dumping a task dependency to a standard DOT format. Developers can use readily available tools such python GraphViz and Viz.js to visualize the graphs without extra programming effort. We also provide a customizable observer interface to let users observe when a thread starts or stops participating in task scheduling. From users' perspective, these facilities largely improve the ease of debugging and reduce the learning curve of Cpp-Taskflow [8].

IV. Experimental Results

We discuss the experimental results on two fronts, micro-benchmarks and real-world applications. Micro-benchmarks measure the *sole* tasking performance of each library on processing two graph structures that represent regular and irregular compute patterns. Next, we move to two real-world CAD applications, VLSI timing analysis and detailed placement, and one parallel machine learning workload. We will show Cpp-Taskflow largely simplifies the developments of realistic use-cases and boosts the performance that was not possible using existing approaches. The quality of results (QoRs) of our implementation remain the same as original ones and we shall focus on the discussion of parallelism. All experiments ran on a CentOS Linux 7.7.1908 machine with 256 GB RAM and 64 AMD Opteron Processors at 2.1 GHz. We lock each thread to one CPU core by using both 1) library-specific API to restrict the number of spawned threads and 2) OS-level utilities (`taskset`) to affine the running process to the same number of CPU cores. We compiled all programs using g++-8.2.0 with C++17 standards `-std=c++17` enabled. Due to the page limit, we considered two industrial-strength libraries OpenMP 4.5 (task dependency clause) and Intel TBB 2019 Update 2 (FlowGraph) as our baseline to execute task dependency graphs [13], [14]. The compiler provides the OpenMP support through the gcc tool chain. All data is an average of ten runs, and is reproducible at [8], [19], and [20].

Given the large amount of tasking libraries, it is impractical to compare Cpp-Taskflow with all of them. Instead, we stick with OpenMP and TBB because of their abundant user experiences and documentations. These activities prevent us from making immature mistakes due to undocumented pitfalls so we can make fair judgement. Also, OpenMP and TBB are widely used in CAD. The comparison is representative for other problems that share similar performance characteristics with CAD. In case readers are interested in comparison with other third-party libraries, data can be found, and reproduced in [8].

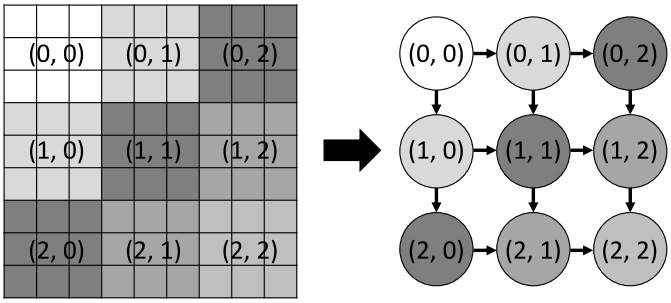


Fig. 7. Task dependency graph of a 2-D wavefront grid.

TABLE I
Software Cost Comparison on Micro-Benchmarks

Software Costs	Cpp-Taskflow		OpenMP		TBB		Sequential	
	LOC	CC	LOC	CC	LOC	CC	LOC	CC
Wavefront	23	7	42	21	31	8	7	3
Graph Traversal	31	6	181	28	45	8	11	3

CC: cyclomatic complexity of the implementation

A. Micro-Benchmark

We consider two classic workloads: 1) wavefront computing and 2) graph traversal. We modified the wavefront computing pattern from the official TBB blog [21]. As shown in Fig. 7, a 2-D matrix is partitioned into a set of identical square blocks. Each block is mapped to a task that performs a nominal operation with constant time complexity. The wavefront propagates task dependencies monotonically from the top-left block to the bottom-right block. Each task precedes one task to the right and another below. In Fig. 7, blocks (tasks) with the same color can run concurrently. The resulting task dependency graph exhibits a regular structure along with the matrix partition. On the other hand, the graph traversal benchmark reads in a randomly generated graph and casts it to a task dependency graph that performs a parallel traversal. Due to the static property of OpenMP task dependency clause, we need to write an exhaustive list to cover all combinations of input and output degrees. To avoid blowing up the OpenMP code, we limit each node to have at most four input and output edges. This experiment mimics the existing OpenMP-based circuit analysis methods and their limitations [9]. The resulting task dependency graph represents an irregular compute pattern.

We begin by examining the software costs using the popular tools SLOCCount and Lizard [22], [23]. Compared with OpenMP and TBB, Cpp-Taskflow achieves the least amount of development efforts in terms of LOC and cyclomatic complexity (see Table I). Our margin to a sequential baseline is also the smallest. The top half of Fig. 8 shows the overall performance of each library. Our measure includes library ramp-up time, construction and execution of the task dependency graph, and clean-up time. The top two plots show the runtime growth

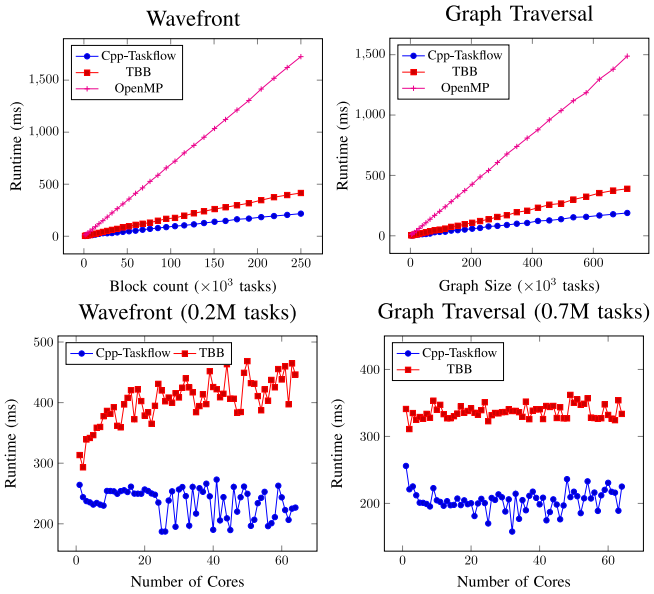


Fig. 8. Performance comparisons between Cpp-Taskflow, TBB, and OpenMP on two micro-benchmarks.

with increasing problem size under 8 cores. In general, Cpp-Taskflow scales up best. The performance margin to OpenMP and TBB becomes larger as the problem size increases. For instance, Cpp-Taskflow is 7.4× and 1.9× faster than OpenMP and TBB in graph traversal at size 348K. Next, we compare the performance between Cpp-Taskflow and TBB on different number of cores at the largest problem size (262144 tasks in wavefront and 711200 tasks in graph traversal). We skip the comparison with OpenMP as it is slower than both TBB and Cpp-Taskflow. As shown in the bottom half of Fig. 8, Cpp-Taskflow is consistently faster than TBB regardless of core numbers. Both libraries start to saturate at about eight cores.

B. VLSI Timing Analysis

We demonstrate the performance of Cpp-Taskflow in a real-world VLSI timing analyzer. We consider our research project *OpenTimer*, an open-source static timing analyzer that has been used in many industrial and academic projects [19]. The first release v1 in 2015 implemented the levelization algorithm (see Section II-D) using the OpenMP 4.5 task dependency clause [9], [24]. To overcome the performance bottleneck, we rewrote the core incremental timing engine using Cpp-Taskflow in the recent release v2. Since OpenTimer is a large project of more than 50K LOC, it is difficult to rewrite the core with TBB. We focus on comparing with OpenMP which had been available in v1. Table II measures the software costs of two OpenTimer versions using the Linux tool SLOCCount [22]. The cost and schedule estimates are based on the constructive cost model (COCOMO) under the organic mode—*small teams with good experience working on a research-driven environment* [25]. In OpenTimer v2, a large amount of exhaustive OpenMP dependency clauses that were used to carry out task dependencies are now replaced with only a few lines of flexible Cpp-Taskflow code (9123 versus 4482). The maximum cyclomatic complexity in a single function is reduced from 58 to 20. We attribute this to

TABLE II
SOFTWARE COST OF OPENTIMER V1 AND V2

Tool	Task Model	LOC	MCC	Effort	Dev	Cost
v1	OpenMP 4.5	9,123	58	2.04	2.90	$275,287
v2	Cpp-Taskflow	4,482	20	0.97	1.83	$130,523

MCC: maximum cyclomatic complexity in a single function
Effort: development effort estimate, person-years (COCOMO model)
Dev: estimated average number of developers (efforts / schedule)
Cost: total estimated cost to develop (average salary = $56,286/year).

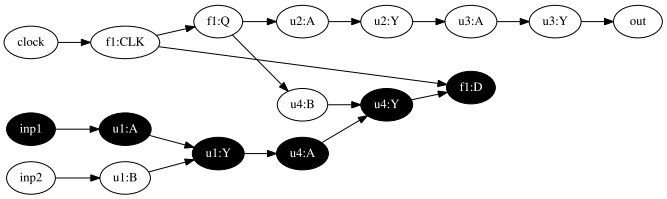


Fig. 9. Example task dependency graph of a single timing update. The black path marks the most critical timing path.

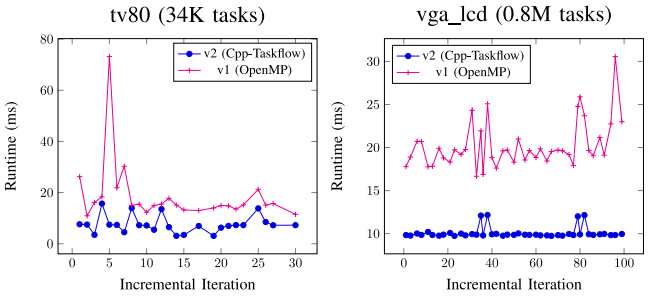


Fig. 10. Runtime comparisons of the incremental timing between OpenTimer v1 (OpenMP) and v2 (Cpp-Taskflow) for two circuits tv80 and vga_lcd under 16 cores.

Cpp-Taskflow's programmability, which can affect the way developers design efficient algorithms and parallel decomposition strategies. For example, OpenTimer v1 relied on a bucket-list data structure to model the task dependency in a pipeline fashion using OpenMP. We found it very difficult to go beyond this paradigm because of the insufficient support for dynamic dependencies in OpenMP. With Cpp-Taskflow in place, we can break this bottleneck and easily model both static and dynamic task dependencies at programming time and runtime. The task dependency graph flows computations naturally and asynchronously with the timing graph, producing faster runtime performance. Fig. 9 shows an example task dependency graph (critical path on black) that represent a single timing update on a sample circuit.

Fig. 10 compares the performance between OpenTimer v1 and v2. We evaluated the runtime versus incremental iterations under 16 cores on two industrial circuit designs tv80 (5.3K gates and 5.3K nets) and vga_lcd (139.5K gates and 139.6K nets) with 45 nm NanGate cell library [11]. Each incremental iteration refers a design modification followed by a timing query to trigger a timing update. In v1, this includes the time to reconstruct the data structure required by OpenMP to alter the task dependencies. In v2, this includes the time to create and launch a new task dependency graph to perform a parallel timing update. As shown in Fig. 10, v2 is consistently faster than v1. The maximum speed-up is 9.8× on tv80

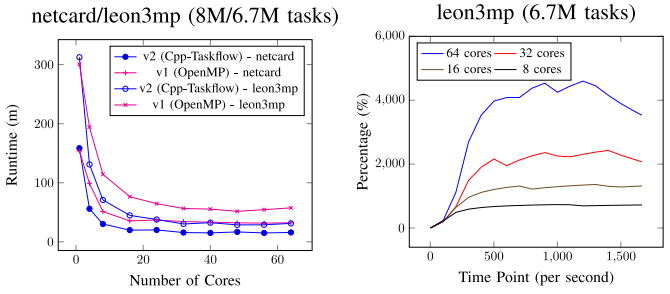


Fig. 11. Scalability and CPU profile of Cpp-Taskflow on two large circuits netcard and leon3mp.

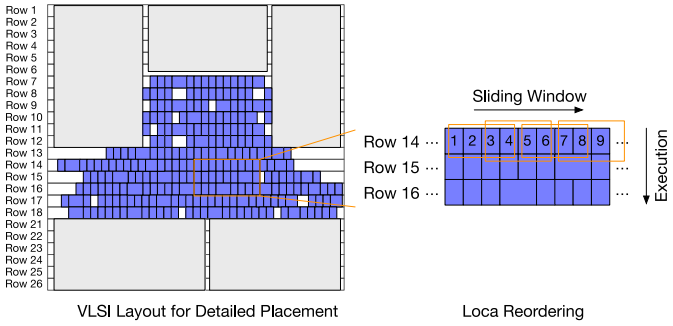


Fig. 12. Example of VLSI layout for detailed placement and the execution of the local reordering algorithm. The blue rectangles denote the movable logic gates for detailed placement. The gray ones denote fixed components that cannot be moved. The brown ones denote the primary IO pins of the circuit. Logic gates are physically aligned to placement rows. A VLSI circuit can be viewed as a hypergraph with logic gates incident to hyperedges. The interconnections between gates are not drawn for brevity.

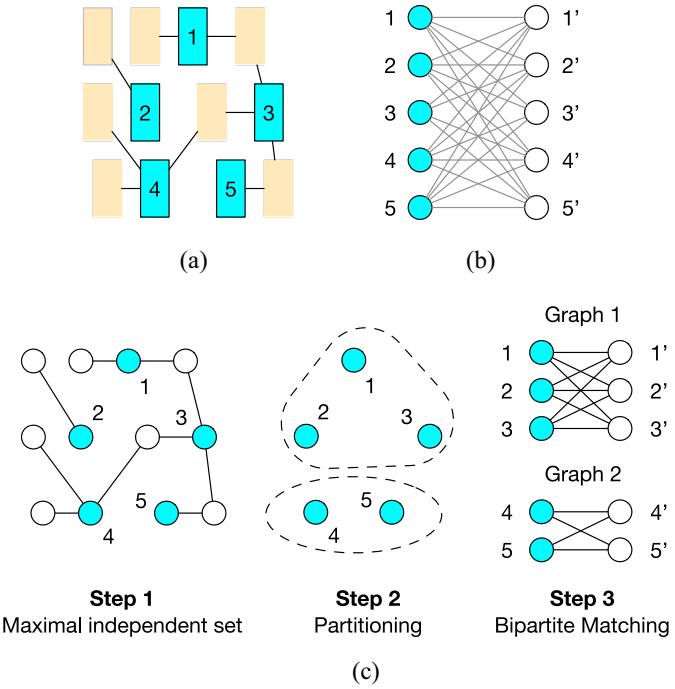


Fig. 13. Matching-based detailed placement algorithm. (a) Example of placed cells and their interconnects. Independent cells are marked in cyan. (b) Weighted bipartite matching to find the best permutation of cell locations. (c) Three-step iterative implementation of the algorithm. During the execution of the algorithm, step 1 of the next iteration can overlap with step 3 of the current iteration.

and 3.1× on vga_lcd. This also demonstrated the performance of Cpp-Taskflow on batch jobs each consisting of a different task pattern (average speed-up is 2.9× on tv80 and 2.0× on vga_lcd). The fluctuation of the curve is caused by design modifiers; some are local changes and others affect the entire timing landscape giving rise to large task dependency graphs. The scalability of Cpp-Taskflow is shown in Fig. 11. We used two million-scale designs, netcard (1.4M gates) and leon3mp (1.2M gates) from the OpenCores [11], to evaluate the runtime of v1 and v2 across different number of cores. There are two important observations. First, v2 is slightly slower than v1 at one core (3%–4%), where all OpenMP's constructs are literally disabled. This shows the graph overhead of Cpp-Taskflow; yet it is negligible. Second, v2 is consistently faster than v1 regardless of core counts except one. This justifies Cpp-Taskflow's programming model largely improved the design of a parallel VLSI timing analyzer that would not be possible with OpenMP.

Both Cpp-Taskflow and OpenMP stagnate at about ten cores. There are many factors that affects the scalability. First, the structure of a timing graph dominates the maximum parallelism. For example, Fig. 9 has 17 tasks, but it cannot scale to 17 cores due to dependencies between pins. Such constraints are decided by circuit graph structures. Second, timing is not data-intensive. During incremental timing, data can arrive intensively or sparsely. Computational patterns are highly irregular and graph-oriented. It is less likely to have a linear strong scalability over increasing problem size that often relies on SIMD-styled parallelism (e.g., matrix operations).

C. VLSI Detailed Placement

We demonstrate the performance of Cpp-Taskflow in a real-world VLSI detailed placement engine, DREAMPlace [20]. Detailed placement is an important step in VLSI backend design to locally refine the physical locations of logic gates or *cells* for minimal wirelength. It often needs to be invoked many times in different stages of the optimization flow, and the algorithm efficiency is very critical for fast design closure. Fig. 12 shows an example of detailed placement on a physical layout. Mainstream detailed placement algorithms are based on iterative *local reordering* [20], [26], [27]. Fig. 13 presents a common matching-based algorithm. The key idea is to extract a maximal independent set (marked in cyan) from a cell set and model the wirelength minimization problem on these nonoverlapped cells into a weighted bipartite matching graph. The entire process is very time-consuming especially for large designs with millions of cells. A typical implementation iterates the following three steps: 1) a parallel maximal independent set finding step using Blelloch's Algorithm [28]; 2) a sequential partitioning step to cluster adjacent cells; and 3) a parallel bipartite matching step to find the best permutation of cell locations. Fig. 13(c) illustrates the process. Many CAD algorithms resemble such techniques to exploit parallelism under net-to-net dependencies [29].

We implemented the detailed placement algorithm in two versions, TBB and Cpp-Taskflow, on top of the DREAMPlace facility [20]. DREAMPlace is a modern placement engine

Fig. 14. Partial task graph for the detailed placement algorithm in Fig. 13. Steps of two consecutive iterations overlap in the task graph.

TABLE III
SOFTWARE COST COMPARISON BETWEEN TBB AND CPP-TASKFLOW ON DETAILED PLACEMENT

Task Model	LOC	MCC	Effort	Dev	Cost
TBB	449	51	0.11	0.48	$15,083
Cpp-Taskflow	321	33	0.09	0.41	$11,491

MCC: maximum cyclomatic complexity in a single function
Effort: development effort estimate, person-years (COCOMO model)
Dev: estimated average number of developers (efforts / schedule)
Cost: total estimated cost to develop (average salary = $56,286/year).

that incorporate advanced parallel algorithms to gain significant runtime benefits. We leveraged Cpp-Taskflow's dynamic tasking to describe the iterative process of the algorithm. An example of the computation graph is shown in Fig. 14. Since TBB does not support condition tasks, we expand the task graph along iterations to form a flat hierarchy. In fact, most existing programming frameworks promote this workaround as a de-facto solution for dynamic control flows [4]. Table III compares the software cost between TBB and Cpp-Taskflow using the Linux tool SLOCCount under the organic mode [22]. In terms of LOC, Cpp-Taskflow is 28.5% fewer than TBB (and 37.1% fewer tokens). We attribute this saving to condition tasks, by which we are able to express the iterative process in a cyclic task graph, rather than flatterning the control flows across iterations. In the later case, extra programming efforts cause the TBB code to produce a much larger cyclomatic complexity than Cpp-Taskflow (51 versus 33).

We evaluate the performance on ISPD 2005 placement contest benchmarks [30]. For brevity, we use c1–c8 to represent the eight circuits, adaptec1–adaptec4, and bigblue1–bigblue4,

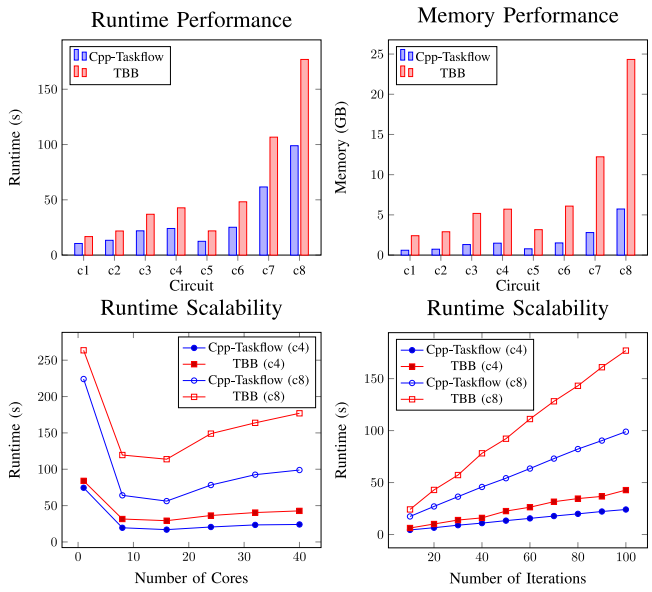


Fig. 15. Performance comparison between between Cpp-Taskflow and TBB on ISPD 2005 benchmarks [30].

respectively. The top half of Fig. 15 draws the runtime and memory comparison between Cpp-Taskflow and TBB on the eight circuits under 40 cores and 100 iterations. The memory value is measured at the maximum resident set size reported by the Linux utility `time`. In both cases, Cpp-Taskflow outperformed TBB across all circuits. The largest margin we observed in runtime is 47.81% on c6 (48.19 versus 25.15 s). The memory usage of Cpp-Taskflow is about 3.8–4.4× fewer than TBB. Take c8 for example, Cpp-Taskflow reached the goal using only 5.73 GB memory whereas TBB requires 24.33 GB (4.24× more). We can clearly see the advantage of our dynamic tasking in handling dynamic and iterative control flows. The bottom half of Fig. 15 scales the runtime with increasing cores (fixed 100 iterations) and placement iterations (fixed 40 cores). We can observe Cpp-Taskflow is consistently faster and more scalable than TBB in all scenarios.

D. Machine Learning

We applied Cpp-Taskflow to speed up the training of a deep neural network (DNN) classifier on the famous MNIST dataset [31]. Training a DNN is an extremely compute-intensive process and exposes many types of parallelism at different levels. For example, the well-know TensorFlow library permit users to alter inter- and intra-operation parallelism [32]. Users can further employ advanced data structures (e.g., RunQueue) to control threads to enable more fine-grained parallelism. However, these separate and low-level concurrency controls impose large burden to users even for experienced developers [**?**]. The goal of this experiment is thus to investigate a task-based approach to simplify the development of parallel machine learning.

We considered two DNN architectures, three layers ($784\times32\times32\times10$) and five layers ($784\times64\times32\times16\times8\times10$). We used a gradient descent optimizer with a mini-batch size 100 and 0.001 learning rate on a training set of 60K images.

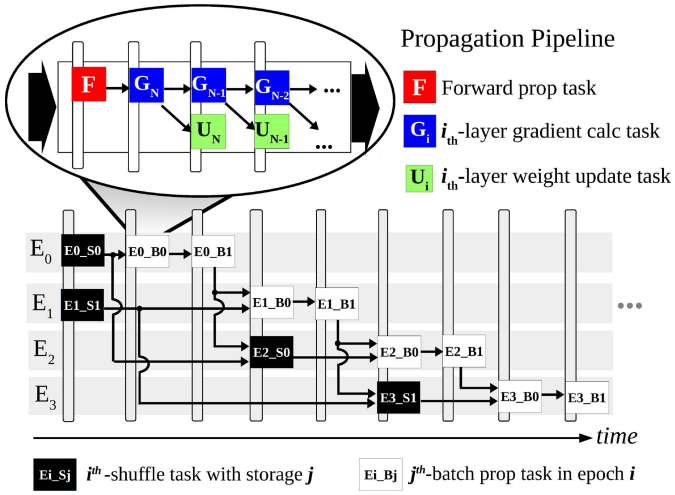


Fig. 16. Parallel task decomposition strategy for DNN training.

TABLE IV
SOFTWARE COST COMPARISON ON MACHINE LEARNING

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

These parameters are inspired from the official TensorFlow MNIST example [32]. We adopted a coarse-grained task decomposition strategy that is applicable to any parallel training frameworks (see Fig. 16). First, we group the backward propagation into two tasks, *gradient calculation* (G_i) and *weight update* (U_i), and pipeline these tasks layer by layer. Second, we create a task for per-epoch data shuffle to enable *epoch-level* parallelism ($E_i\_S_j$). To avoid too much memory overhead in storing shuffled data, we limit the degree of storages to twice the number of threads. Spare threads can start shuffling the data for subsequent epochs. Indeed, shuffling the data can be very time-consuming especially when applications adopt complex algorithms to randomize data blocks to improve the stochastic gradient descent. All matrix operations are written in Eigen-3.3.7 [33].

Table IV presents the software costs (reported by SLOCCount and Lizard [22], [23]) of Cpp-Taskflow, OpenMP, and TBB in implementing our core parallel decomposition strategy. In general, Cpp-Taskflow has the fewest LOC and the lowest cyclomatic complexity. The development effort is measured by the time it took for an experienced programmer (7-year C++ and 2-year machine learning practice) to finish each implementation. TBB's programming model is very similar to Cpp-Taskflow and thus both took roughly the same time to develop (3 h). However, it is tricky to implement the task dependency graph with OpenMP. In order to ensure proper dependencies between tasks, we need to hard-code an order of task dependency clauses that is only specific to a DNN architecture. The development time was twice longer than that of Cpp-Taskflow. In fact, most time was spent on debugging the order of dependent tasks. This measurement can be subjective,

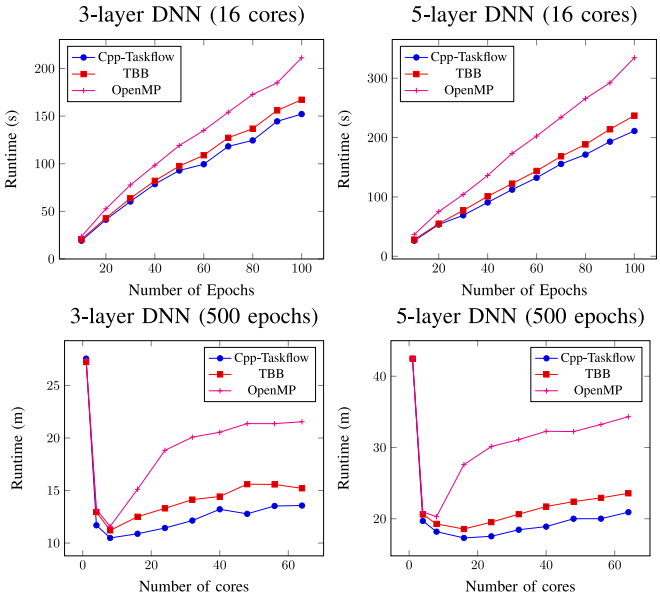


Fig. 17. Performance comparison between Cpp-Taskflow, TBB, and OpenMP on training two different DNN classifiers.

but it does highlight the impact of a library's task model on engineering productivity.

Fig. 17 shows the overall performance of each library on training the two DNN architectures. Each epoch consists of 4201 tasks and 6601 tasks for the three-layer DNN and the five-layer DNN, respectively. All libraries reached performance saturation at about 8–16 cores. Under 16 cores, Cpp-Taskflow is consistently faster than OpenMP and TBB on both DNN architectures, regardless of the number of training epochs. The margin becomes even larger when we increase the epoch count. While the scalability is mostly dominated by the maximum concurrency of the training graph, Cpp-Taskflow is faster than others under different core numbers (see Fig. 17). For example, Cpp-Taskflow finished the training of the three-layer DNN by 1.38× and 1.14× faster than OpenMP and TBB under 16 cores. Similar trends can also be observed at other CPU core configurations. The performance of all implementations saturate at about 16 cores. In summary, Cpp-Taskflow achieved the best software costs, performance, and scalability.

V. RELATED WORK

Cpp-Taskflow is mostly related to OpenMP task dependency clause and TBB FlowGraph. In OpenMP 4.0, the task group and depend clause (`depend(type : list)`) were included into its directives [13]. The clause allows users to define lists of data items that are only inputs, only outputs, or both to form a task dependency graph. The biggest problem of this paradigm is the programmability. Users need a descent understanding about the graph structure in order to annotate tasks in a specific order consistent with the sequential execution. Also, OpenMP has very limited support for increasingly adopted C++14 and C++17 standards. This is unfortunate as these new standards largely help the development of every kind of applications. Similar issues exist in other directive-driven libraries such as Cilk, Ompss, Cells, SMPSs, and

Nanos++ [34]–[37]. On the other hand, Intel released in 2017 the TBBs library that supports loop-level parallelism and task-based programming (FlowGraph) [14]. The TBB task model is object-oriented. It supports a variety of methods to create a highly optimized flow graph and provides users runtime interaction with the scheduler. Nevertheless, TBB does have drawbacks, mostly from an ease-of-programming standpoint. Because of various supports, the TBB task graph description language is very complex and can often result in handwritten code which are hard to debug and read.

The high-performance computing (HPC) community has long been managing task-based programming frameworks. Many of such systems are inspired by scientific computing and clusters. Chapel, X10, Charm++, HPX, and Legion introduced new domain specific languages (DSLs) and runtime to support tasking in a global address space (GAS) environment [38]–[42]. QURAK, StarPU, PaRSEC, and ParalleX are capable of tracking data between different memory and scheduling tasks on heterogeneous resources [43]–[46]. While these systems are orthogonal to Cpp-Taskflow, we are leveraging their experience to handle new types of workload.

VI. Conclusion

In this article, we have presented Cpp-Taskflow, a high-performance task programming system to streamline parallel processing. Cpp-Taskflow leverages modern C++ to enable efficient implementations of parallel decomposition strategies for both regular loop-based parallelism and irregular patterns such as graph algorithms and dynamic control flows. We have evaluated Cpp-Taskflow on both micro-benchmarks and real-world applications, including VLSI design automation problems and machine learning. Results have shown promising performance and scalability of Cpp-Taskflow over two industrial-strength libraries, OpenMP Tasking, and Intel TBBs FlowGraph. On a VLSI detailed placement example, Cpp-Taskflow achieved 10%–30% speed-up over OpenMP with similar coding complexity.

Our future work focuses on applying Cpp-Taskflow to express heterogeneous timing analysis algorithms [47], [48] and speed up large-scale machine learning problems using task graph parallelism [49]. In addition, we are extending Cpp-Taskflow to a distributed environment based on our distributed execution engine, DtCraft [50].

Acknowledgment

The authors would like to thank all users of Cpp-Taskflow for insightful feedbacks and suggestions [8]. The source of Cpp-Taskflow is published on GitHub under MIT license [8]. Project details, API reference, and profiler are available on the repo.

References

[1] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern C++," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Rio de Janeiro, Brazil, 2019, pp. 974–983.

[2] L. Stok, "Developing parallel EDA tools [the last byte]," *IEEE Design Test*, vol. 30, no. 1, pp. 65–66, Feb. 2013.

[3] T.-W. Huang, "A general-purpose parallel and heterogeneous task programming system for VLSI CAD," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 1–2.

[4] Y.-S. Lu and K. Pingali, "Can parallel programming revolutionize EDA tools?" in *Advanced Logic Synthesis, Springer*. Cham, Switzerland: Springer, 2018, pp. 21–41.

[5] E. Ayguadé and D. Jiménez-González "An approach to task-based parallel programming for undergraduate students," *J. Parallel Distrib. Comput.*, vol. 118, pp. 140–156, Aug. 2018.

[6] P. Thoman *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *J. Supercomput.*, vol. 74, no. 4, pp. 1422–1434, Apr. 2018.

[7] B. B. Fraguela, "A comparison of task parallel frameworks based on implicit dependencies in multi-core environments," in *Proc. Hawaii Int. Conf. Syst. Sci. (HICSS)*, 2017, p. 10.

[8] *CPP-Taskflow*. Accessed: 2020. [Online]. Available: https://github.com/taskflow

[9] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2015, pp. 895–902.

[10] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Boston, MA, USA: Springer, 2009.

[11] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, 2015, pp. 895–902.

[12] *OpenSTA, Parallax Inc*. Accessed: 2020. [Online]. Available: http://www.parallaxsw.com/

[13] (2015). *OpenMP 4.5*. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[14] *Intel TBB*. Accessed: 2020. [Online]. Available: https://github.com/01org/tbb

[15] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 135–146, 2013.

[16] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. 24th ACM Symp. Oper. Syst. Principles (SOSP)*, 2013, pp. 456–471.

[17] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "An efficient and composable parallel task programming library," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Waltham, MA, USA, 2019, pp. 1–7.

[18] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proc. 10th Annu. ACM Symp. Parallel Algorithms Archit. (SPAA)*, 1998, pp. 119–129.

[19] *OpenTimer*. Accessed: 2020. [Online]. Available: https://github.com/OpenTimer

[20] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPIace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Proc. 56th ACM/IEEE Design Autom. Conf. (DAC)*, Las Vegas, NV, USA, 2019, pp. 1–6.

[21] *Intel Developer Zone*. Accessed: 2020. [Online]. Available: https://software.intel.com/en-us/node/506116

[22] *SLOCCount*. Accessed: 2020. [Online]. Available: https://dwheeler.com/sloccount/

[23] *Lizard*. Accessed: 2020. [Online]. Available: https://github.com/terryyin/lizard

[24] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A new parallel incremental timing analysis engine," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, early access, Jul. 7, 2020, doi: 10.1109/TCAD.2020.3007319.

[25] C. F. Kemerer, "An empirical validation of software cost estimation models," *Commun. ACM*, vol. 30, no. 5, pp. 416–429, 1987.

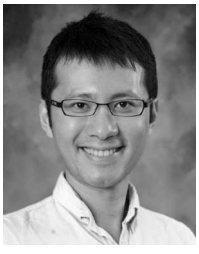
[26] M. Pan, N. Viswanathan, and C. Chu, "An efficient and effective detailed placement algorithm," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, San Jose, CA, USA, 2005, pp. 48–55.

[27] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE TCAD*, vol. 27, no. 7, pp. 1228–1240, Jul. 2008.

[28] G. E. Blelloch, J. T. Fineman, and J. Shun, "Greedy sequential maximal independent set and matching are parallel on average," in *Proc. 24th Symp. Parallelism Algorithms Archit. (SPAA)*, 2012, pp. 308–317.

[29] Y. O. M. Moctar and P. Brisk, "Parallel FPGA routing based on the operator formulation," in *Proc. 51st ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2014, pp. 1–6.

[30] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ISPD2005 placement contest and benchmark suite," in *Proc. Int. Symp. Phys. Design*, 2005, pp. 216–220.
[31] *MNIST*. Accessed: 2020. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[32] *TensorFlow*. Accessed: 2020. [Online]. Available: https://www.tensorflow.org/
[33] *Eigen*. Accessed: 2020. [Online]. Available: https://eigen.tuxfamily.org/dox/
[34] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, annd Y. Zhou, "Cilk: An efficient multithreaded runtime system," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 55–69, 1996.
[35] A. Duran *et al.*, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 173–193, 2011.
[36] J. M. Perez, R. M. Badia, and J. Labarta "A dependency-aware task-based programming environment for multi-core architectures," in *Proc. IEEE Int. Conf. Clust. Comput.*, Tsukuba, Japan, 2008, pp. 142–151.
[37] *Nanos++*. Accessed: 2020. [Online]. Available: https://pm.bsc.es/nanox
[38] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitioned Global Address Space Program. Models (PGAS)*, 2014, pp. 1–11.
[39] B. Chamberlain, D. Callahan, and H. P. Zima "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, Aug. 2007.
[40] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *Proc. 20th Annu. ACM SIGPLAN Conf. Object Orient. Program. Syst. Lang. Appl. (OOPSLA)*, 2005, pp. 519–538.
[41] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proc. 8th Annu. ACM SIGPLAN Conf. Object Orient. Program. Syst. Lang. Appl. (OOPSLA)*, 1993, pp. 91–108.
[42] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2014.
[43] A. Yarkhan, "Dynamic task execution on shared and distributed memory architectures," Ph.D. dissertation, Dept. Comput. Sci., Univ. Tennessee, Knoxville, TN, USA, 2012.
[44] E. Agullo *et al.*, "Harnessing clusters of hybrid nodes with a sequential task-based programming model," in *Proc. Int. Workshop Parallel Matrix Algorithms Appl. (PMAA)*, 2014, pp. 1–55.
[45] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *IEEE Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov./Dec. 2013.
[46] G. R. Gao *et al.*, "ParalleX: A study of a new parallel computation model," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2007, pp. 1–6.
[47] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated static timing analysis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2020, pp. 46–55.
[48] G. Guo, T. Huang, C. Lin, and M. Wong, "An efficient critical path generation algorithm considering extensive path constraints," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.
[49] D.-L. Lin and T.-W. Huang, "A novel inference algorithm for large sparse neural network using task graph parallelism," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, 2020, pp. 1–7.
[50] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "DtCraft: A high-performance distributed execution engine at scale," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 6, pp. 1070–1083, Jun. 2019.



Tsung-Wei Huang received the B.S. and M.S. degrees from the Department of Computer Science, National Cheng Kung University, Tainan, Taiwan, in 2010 and 2011, respectively, and the Ph.D. degree from the Electrical and Computer Engineering (ECE) Department, University of Illinois at Urbana–Champaign, Urbana, IL, USA, in 2017.

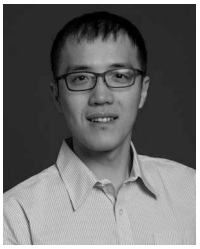
He is currently an Assistant Professor with the ECE Department, University of Utah, Salt Lake City, UT, USA. He has been building software systems from the ground up with a specific focus on parallel processing and timing analysis.

Dr. Huang is a recipient of the ACM/SIGDA Outstand Ph.D. Dissertation Award in 2019.

Yibo Lin (Member, IEEE) received the B.S. degree in microelectronics from Shanghai Jiao Tong University, Shanghai, China, in 2013, and the Ph.D. degree from the Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX, USA, in 2018.

He is currently an Assistant Professor with the Computer Science Department associated with the Center for Energy-Efficient Computing and Applications, Peking University, Beijing, China. His research interests include physical design, machine learning applications, GPU acceleration, and hardware security.

Dr. Lin has received four Best Paper Awards at premier venues (ISPD 2020, DAC 2019, VLSI Integration 2018, and SPIE 2016). He has also served in the Technical Program Committees of many major conferences, including ICCAD, ICCD, ISPD, and DAC.



Chun-Xun Lin received the B.S. degree in electrical engineering from the National Cheng Kung University, Tainan, Taiwan, in 2009, and the M.S. degree in electronics engineering from the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan, in 2011. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, USA.

His current research interests include parallel processing and physical design.

Guannan Guo received the B.S. degree from the Department of Electrical and Computer Engineering, University of Illinois at Urbana–Champaign, Urbana, IL, USA, where he is currently pursuing the Ph.D. degree.

His research focus on circuit timing analysis, distributed systems, and scheduling algorithms.

Martin D. F. Wong (Fellow, IEEE) received the B.S. degree in mathematics from the University of Toronto, Toronto, ON, Canada, in 1979, the M.S. degree in mathematics from the University of Illinois at Urbana–Champaign (UIUC), Urbana, IL, USA, in 1981, and the Ph.D. degree in computer science from UIUC in 1987.

From 1987 to 2002, he was a Faculty Member of Computer Science with the University of Texas at Austin, Austin, TX, USA. He returned to UIUC in 2002, where he was the Executive Associate Dean of the College of Engineering from 2012 to 2018 and the Edward C. Jordan Professor of Electrical and Computer Engineering. He is currently the Dean of the Faculty of Engineering with the Chinese University of Hong Kong, Hong Kong. He has published over 500 technical papers and graduated more than 50 Ph.D. students in the area of electronic design automation.

Dr. Wong is a Fellow of ACM.