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

Tsung-Wei Huang*, Chun-Xun Lin*, Guannan Guo*, and Martin Wong*
*Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA

***Abstract*—In this paper we introduce Cpp-Taskflow, a new C++ tasking library to help developers quickly write parallel programs using task dependency graphs. 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, incremental flows, and dynamic data structures. 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 real-world applications with million-scale tasking. In a machine learning example, Cpp-Taskflow achieved 1.5–2.7× less coding complexity and 14–38% speed-up over two industrial-strength libraries OpenMP Tasking and Intel Threading Building Blocks (TBB).**

I. Introduction

This paper addresses a long-standing problem, "*how can we make it easier for C++ developers to write efficient parallel programs under complex task dependencies?*" 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 [1]. The traditional *loop-based* parallelism is not sufficient for exploiting the scalability of complex software and parallel algorithms that require irregular compute patterns such as graph traversal and dynamic flows [2]. 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 *task dependency graph*. The library programmability can affect a C++ developer from subtle implementation details to algorithm-level decisions of parallel decomposition strategies [3]. However, related research remains nascent, particularly on the front of using modern C++ to enhance the functionality and performance that were previously not possible.

Consequently, we introduce Cpp-Taskflow, a new C++ tasking library to help developers quickly write parallel programs using task dependency graphs [4]. Cpp-Taskflow is written in C++17, allowing users to use powerful modern C++ features and standard libraries together with our parallelization framework to write fast and scalable parallel programs. We have designed a simple and expressive graph description language that empowers developers with both static and dynamic graph constructions and refinements to fully exploit task parallelism. Listing 1 demonstrates an example Cpp-Taskflow program. The code *explains itself*. The program creates a task dependency graph of four tasks, A, B, C, and D. 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 nor complex lock controls in the code.

```
tf::Taskflow tf;

auto [A, B, C, D] = tf.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
B.precede(D);      // B runs before D
C.precede(D);      // C runs before D

tf.wait_for_all();  // block until finish
```

Listing 1: A simple task dependency graph in Cpp-Taskflow.

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 number of cores. 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:

- **Programming model**. We developed a simple parallel task programming model that enables efficient implementations of parallel algorithms. 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 minutes.
- **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.
- **Unified interface**. We developed a unified programming interface for both static and dynamic tasking. The same

API used for static tasking all applies to dynamic tasking. Programmers need not to learn a different API set.

We have evaluated Cpp-Taskflow on both micro-benchmarks and real-world applications. The performance 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 [4].

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. Timing analysis is a very important component in the overall design flow [5]. 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. 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. Figure 1 shows an example analysis benchmark from IBM chip designs. Timing this circuit can take several hours or days when sign-off is taken into count. Computing an analysis loop requires fairly expensive computations and must take advantage of multicore to speed up the runtime.

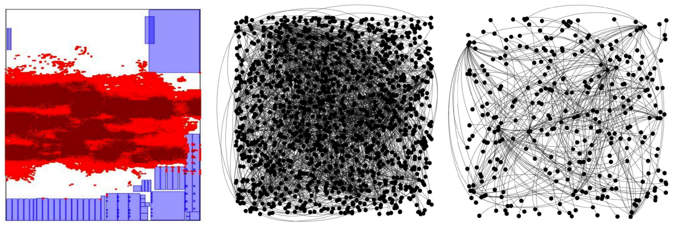
(a) Circuit (1.01mm^2) (b) Graph (3M gates) (c) A signal path

Fig. 1: An industrial IBM circuit design benchmark [6].

A. Challenge 1: Large and 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, 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 [6].

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.

C. Challenge 3: Dynamic Flow

Optimization or physical synthesis programs 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 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 were written in C++ and focus on loop-level parallelization [6], [7]. The most common approach, including industrial implementations, is to *levelize* the circuit graph into a topological order, and apply language-specific "`parallel_for`" level by level. Two mainstream library choices are OpenMP task dependency clause and Intel Threading Building Blocks (TBB) FlowGraph [8], [9]. However, there are many limitations 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. These issues combined to make it difficult to go beyond the loop-based approach, disallowing computations to flow naturally with the timing graph. After many years of research, we and our industry partners conclude the biggest hurdle to a scalable parallel timing analyzer is a suitable *task programming library*. Inspired by our problem domains, we are interested in the workload of million- to billion-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 a general tasking library for writing parallel applications [4].

Cpp-Taskflow aims to help C++ developers quickly write parallel programs and implement efficient parallel decomposition strategies using the task-based approach.

— Cpp-Taskflow's Project Mantra

A. Create a Task

Cpp-Taskflow is *object-oriented*. A task in Cpp-Taskflow is defined as a *callable* object for which the operation `std::invoke` is applicable. Listing 2 demonstrates the creation of a task in Cpp-Taskflow. The first entry to a Cpp-Taskflow program is declaring a taskflow object from the class `tf::Taskflow`. A taskflow object is where to create task dependency graphs and dispatch them to threads for execution. The method `emplace` creates a task from a given callable object. Users can also create multiple tasks at one time.

```
tf::Taskflow tf;

// create a task with a closure
auto A = tf.emplace(
  [] () { std::cout << "Task A\n"; }
);

// create multiple tasks at one time
auto [X, Y, Z] = tf.emplace(
  [] () { std::cout << "Task X\n"; },
  [] () { std::cout << "Task Y\n"; },
  [] () { std::cout << "Task Z\n"; }
);
```

Listing 2: Create a task in a taskflow object.

Each time users create a task, the taskflow object adds a node to the present graph and returns a *task handle*. A task 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 pre-allocate 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 a very intuitive method called `precede` for users to create a task dependency between a pair of tasks. 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.

```
tf::Taskflow tf;
auto [a0, a1, a2, a3, b0, b1, b2] = tf.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);
a2.precede(a3);
b0.precede(b1);
b1.precede(a2, b2);
b2.precede(a3);

tf.wait_for_all();
```

Listing 3: Cpp-Taskflow code of Figure 2 (17 LOC and 178 tokens).

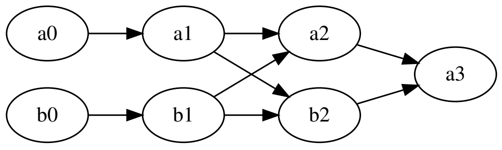
$$\begin{tikzcd}
a0 \arrow[r] & a1 \arrow[r] \arrow[dr] & a2 \arrow[dr] & \\
 & & & a3 \\
b0 \arrow[r] & b1 \arrow[r] \arrow[ur] & b2 \arrow[ur] &
\end{tikzcd}$$

Fig. 2: A static task dependency graph of seven tasks and eight dependency constraints.

Figure 2 shows an example of static task dependency graph and Listing 3 demonstrates its implementation with Cpp-Taskflow. These task dependencies are described intuitively using the method `precede` 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 demonstrated 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 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. 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 switching to a different compiler vendor or integrating with other projects where OpenMP needs disabled. 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 getting started 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 effort.

```
#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 Figure 2 (22 LOC and 181 tokens).

```
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 Figure 2 (37 LOC and 295 tokens).

C. Dispatch a Task Dependency Graph

Each taskflow object has exactly one graph at a time that represents a task dependency graph constructed so far. Once a task dependency graph is decided, the next step is to dispatch it to threads for execution. The graph exists and remains in control until users dispatch it for execution. Figure 3 illustrates the program flow of dispatching a task dependency graph. In Cpp-Taskflow, we call a dispatched graph a *topology*. A topology is a data structure that wraps up a dispatched graph and stores a few metadata obtained at runtime. Each taskflow object has a list of topologies to keep track of the execution status of dispatched graphs. The communication is based on a pair of C++ `shared_future` and `promise`. Users can retrieve this information later on for graph inspection and debugging. All tasks are executed in a shared thread storage coupled with an *executor* to decide which thread runs which task.

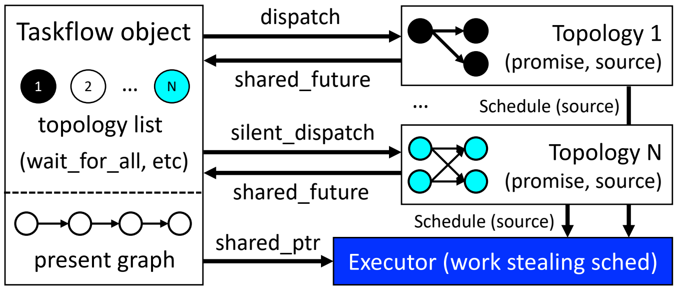

Fig. 3: Program flow of dispatching task dependency graphs.

Cpp-Taskflow provides two ways to dispatch a task dependency graph, *blocking* and *non-blocking* executions. Listing 6 demonstrates the usage of these two methods. The first method `wait_for_all` is *blocking*. It dispatches the present graph to threads and blocks until all tasks finish. On the contrary, the second method `dispatch` is *non-blocking*. It dispatches the present graph to threads and returns immediately to the program without waiting for all tasks to finish. This allows programmers to perform other computations to overlap the graph execution. Users can acquire a `std::shared_future` object returned from `dispatch` to access the execution status of the graph or call `get` to block on completion. Cpp-Taskflow provides also a method `silent_dispatch` for users to ignore the execution status.

```
tf::Taskflow tf;

auto [A, B] = tf.emplace(
  [] () { std::cout << "Task A\n"; }),
  [] () { std::cout << "Task B\n"; })
});
A.precede(B);  // task A runs before task B
```

```
tf.wait_for_all();   // block until finish

std::tie(A, B) = tf.emplace(
  [] () { std::cout << "New Task A\n"; }),
  [] () { std::cout << "New Task B\n"; })
);
B.precede(A);   // task B runs before task A

auto shared_future = tf.dispatch();
// do something to overlap the graph execution
// ...
shared_future.get();  // block until finish
```

Listing 6: Dispatch task dependency graphs for execution.

D. Dynamic Tasking

Another powerful feature of Cpp-Taskflow is *dynamic 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 dispatched graph. These tasks are spawned from a parent task and are grouped together to form a task dependency graph called *subflow*. We believe the biggest difference and advantage that stand Cpp-Taskflow out of existing tasking frameworks is our unified interface for static tasking and dynamic tasking. 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 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.

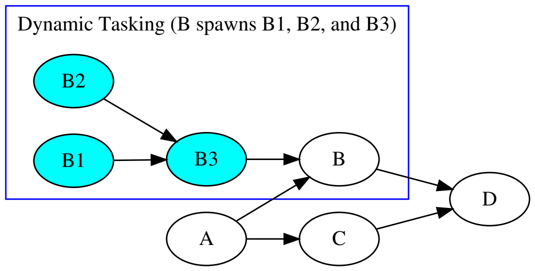

Fig. 4: A dynamic task dependency graph of four static tasks (A, B, C, and D) and three dynamic tasks (B1, B2, and B3).

Listing 7 demonstrates Cpp-Taskflow's implementation on a dynamic task dependency graph in Figure 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::SubflowBuilder` 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 7 and 8 compare two implementations of Figure 4 using Cpp-Taskflow and TBB. In a rough view, Cpp-Taskflow has the least amount of code (20 vs 38). Our user feedbacks lead us to believe that our dynamic tasking ends up being cleaner and more expressive [4]. 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.

```
tf::Taskflow tf;

auto [A, C, D] = tf.emplace(
  [] () { std::cout << "A\n"; },
  [] () { std::cout << "C\n"; },
  [] () { std::cout << "D\n"; }
);
auto B = tf.emplace([] (auto& subflow) {
  std::cout << "B\n";
  auto [B1, B2, B3] = subflow.emplace(
    [] () { std::cout << "B1\n"; },
    [] () { std::cout << "B2\n"; },
    [] () { std::cout << "B3\n"; }
  );
  B1.precede(B3);
  B2.precede(B3);
});
A.precede(B, C);
B.precede(D);
C.precede(D);

tf.wait_for_all();
```

Listing 7: Cpp-Taskflow code of Figure 4 (20 LOC and 190 tokens).

```
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 another 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());  // explicit source A
  G.wait_for_all();
```

Listing 8: TBB code of Figure 4 (38 LOC and 299 tokens).

E. Executor

Each taskflow object has an *executor* to schedule in which list of tasks to execute per thread. While detailing the scheduler is out of the scope of this paper, we briefly highlight our algorithm. The default task scheduler performs a mixed strategy of *work stealing* and *work sharing*, as presented in Algorithm 1. In addition to a typical work stealing loop, we introduced two heuristics. First, we keeps each worker thread an *exclusive* task cache to reduce the access times to its task queue. Per-thread local cache enables *speculative* execution and ensures no context switch for tasks with linear tasks dependency (line 16:25). Second, we maintain a list of *idlers* for those worker threads preempted (line 8). This allows us to precisely wake up a spare worker to run tasks or balance the load through stealing (line 26:28).

Cpp-Taskflow's executor interface is *pluggable* and *shareable*. Users can customize their own scheduler for specific problems or proprietary platforms. Sharing an executor among multiple taskflow objects facilitates modular developments in large Cpp-Taskflow applications, while avoiding the problem of thread over-subscription. We use `std::shared_ptr` to manage the ownership of an executor. A real case from our users employs this functionality to design an efficient animation program, where a main taskflow object handles renders and others tackle the dependency of resource loading [4].

F. Algorithm and Application Encapsulations

One of the key benefits of task-based programming is the encapsulation of an algorithm or an application into a *task pattern*. Cpp-Taskflow facilitates the realization of this concept to promote rapid developments of large parallel programs through smaller and structurally correct patterns.

Algorithm 1: WorkStealingScheduler

```
1  while stop ≠ true do
2      if auto t ← worker.queue.pop(); t == nullopt then
3          t ← steal(worker.last_victim);
4      end
5      if t == nullopt then
6          unique_lock.lock();
7          if all queues are empty then
8              insert_idler(worker);
9              while worker.idler == true do
10                 worker.cv.wait(unique_lock);
11             end
12         end
13         unique_lock.unlock();
14         std::swap(t, worker.cache);
15     end
16     while t ≠ nullopt do
17         std::invoke(t.value());
18         if worker.cache then
19             t ← std::move(worker.cache);
20             worker.cache ← nullopt;
21         end
22         else
23             t ← nullopt;
24         end
25     end
26     if p ← random(); p == a given probability then
27         awaken_one_idler_for_load_balancing();
28     end
29 end
```

Cpp-Taskflow has a built-in algorithm collections that implemented common parallel workloads such as `parallel_for`, `reduce`, and `transform`. Our implementations follow the conventions of the C++ standard libraries. Users can easily write generic code through powerful template instantiation and splice it to their task dependency graphs to compose larger application modules. An on-going project is building a set of machine learning patterns on top of Cpp-Taskflow.

G. Debugging a Task Dependency Graph

Debugging a parallel program can be extremely difficult due to subtly buggy implementations of a task dependency graph. One of the biggest advantages of Cpp-Taskflow is the built-in support for dumping a task dependency graph 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. This largely facilitates the ease of debugging and speeds up the learning curve of task-based programming. Figure 5 shows a visualization of a nested subflow graph.

IV. Experimental Results

We discuss the experimental results on two fronts, micro-benchmarks and real-world applications. Micro-benchmarks measure the *pure* tasking performance of each library on processing two graph structures that represent regular and irregular compute patterns. Next we move to two real-world

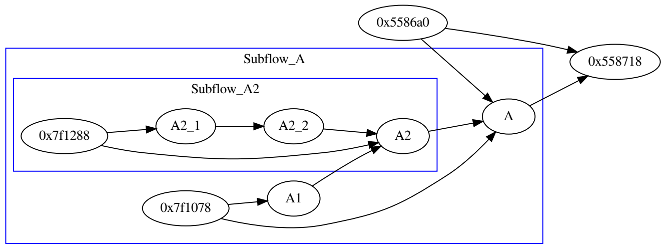

Fig. 5: Visualization of a nested sublfow using the `dump` method.

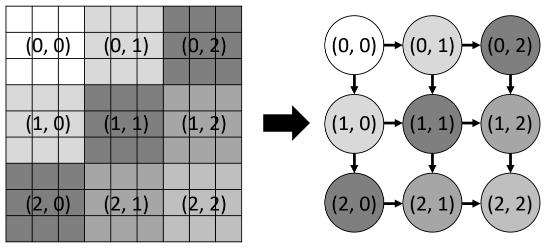

Fig. 6: A 2D wavefront example and its task dependency graph.

TABLE I: Software Costs Comparison on Micro-benchmarks

Software Costs	Cpp-Taskflow		OpenMP		TBB		Sequential	
	LOC	CC	LOC	CC	LOC	CC	LOC	CC
Wavefront	30	7	64	12	38	8	14	3
Graph Traversal	40	6	213	28	59	8	14	3

CC: cyclomatic complexity of the implementation

applications, a large-scale VLSI timing analyzer and a 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 in existing approaches. All experiments ran on a CentOS Linux 7.6.1810 machine with 256 GB RAM and 64 AMD Opteron Processors at 2.1 GHz. We lock each thread to one CPU 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 CPUs. We compiled all programs using g++-8.2.0 with optimization flag `O2` and 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 U3 (FlowGraph) as our baseline to execute task dependency graphs [8], [9]. The compiler provides the OpenMP support through the gcc tool chain.

A. Micro-benchmark

We consider two classic workloads, wavefront computing and graph traversal. We modified the wavefront computing pattern from the official TBB blog [10]. As shown in Figure 6, a 2D 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 Figure 6, 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 [7]. The resulting task dependency graph represents an irregular compute pattern.

We begin by examining the software costs using the popular tools SLOCCount and Lizard [11], [12]. 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. Figure 7 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 with increasing problem size under 8 CPUs. 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 7.9× 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 CPUs at the largest problem size (262144 tasks in wavefront and 711002 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 two plots, Cpp-Taskflow is consistently faster than TBB regardless of CPU numbers. There are two important observations. First, Cpp-Taskflow is about 32-84% faster than TBB at one CPU. This reveals the overhead of TBB's internal data structure to carry out its flow graph model. Second, both libraries start to saturate at about 8 CPUs in graph traversal. On the wavefront graph, Cpp-Taskflow scales up to 9 CPUs whereas TBB stops at 4 CPUs. While this number is application-dependent, we can see Cpp-Taskflow outperforms TBB in task scheduling.

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 [13]. The first release v1 in 2015 implemented the levelization algorithm (see Section II-D) using the OpenMP 4.5 task dependency clause [7]. 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 lines of code, 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 under the organic

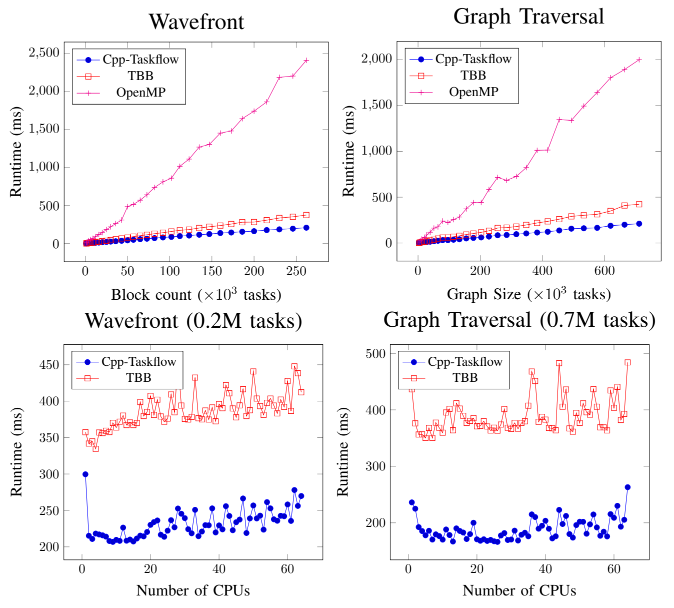

Fig. 7: Performance comparisons between Cpp-Taskflow, TBB, and OpenMP on two micro-benchmarks.

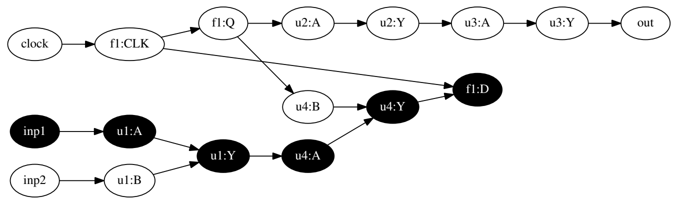

Fig. 8: An example task dependency graph of a single timing update.

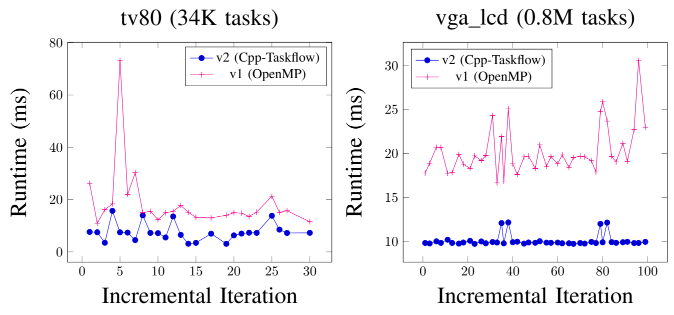

Fig. 9: Runtime comparisons of the incremental timing between OpenTimer v1 (OpenMP) and v2 (Cpp-Taskflow) under 16 CPUs.

TABLE II: Software Costs 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).

mode [11]. 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 vs 4482). The maximum cyclomatic complexity in a single function is reduced from 58 to 20. We attribute this to 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. Figure 8 shows an example task dependency graph (critical path on black) that represent a single timing update on a sample circuit.

Figure 9 compares the performance between OpenTimer v1 and v2. We evaluated the runtime versus incremental iterations under 16 CPUs on two industrial circuit designs tv80 (5.3K gates and 5.3K nets) and vga_lcd (139.5K gates and 139.6K nets) with 45nm NanGate cell library [6]. 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 Figure 9, v2 is consistently faster than v1. The maximum speed-up is 9.8$\times$ on tv80 and 3.1$\times$ 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$\times$ on tv80 and 2.0$\times$ 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 Figure 10. We used two million-scale designs, netcard (1.4M gates) and leon3mp (1.2M gates) from the OpenCores [6], to evaluate the runtime of v1 and v2 across different number of CPUs. There are two important observations. First, v2 is slightly slower than v1 at one CPU (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 CPU 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.

C. Deep Neural Network

We applied Cpp-Taskflow to speed up the training of a deep neural network (DNN) classifier on the famous MNIST dataset [14]. 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 [15]. Users can further employ advanced data structures

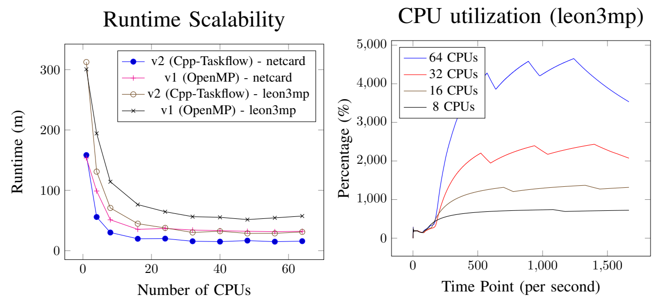

Fig. 10: Scalability and CPU profile of Cpp-Taskflow on large circuit designs netcard (8M tasks) and leon3mp (6.7M tasks).

(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.

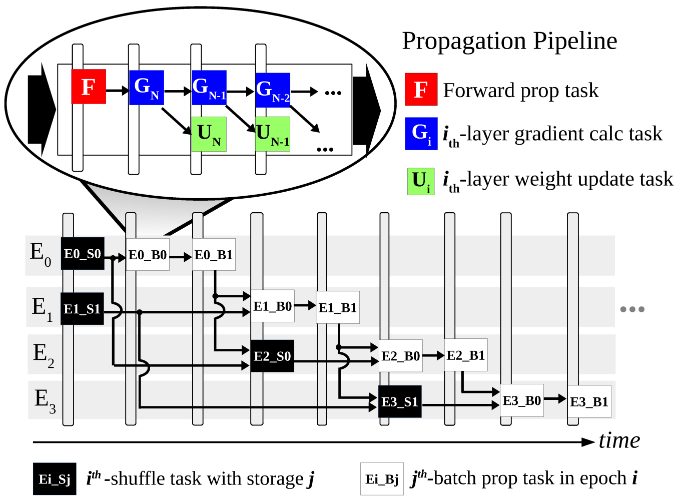

Fig. 11: Task decomposition strategy for parallel DNN training.

We considered two DNN architectures, three layers (784×32×32×10) and five layers (784×64×32×16×8×10). We used a gradient descent optimizer with a mini-batch size 100 and 0.001 learning rate on a training set of 60K images. These parameters are inspired from the official TensorFlow MNIST example [15]. We adopted a coarse-grained task decomposition strategy that is applicable to any parallel training frameworks (see Figure 11). 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 encapsulated to standalone function calls written with Eigen-3.3.7 [16].

TABLE III: Software Costs 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

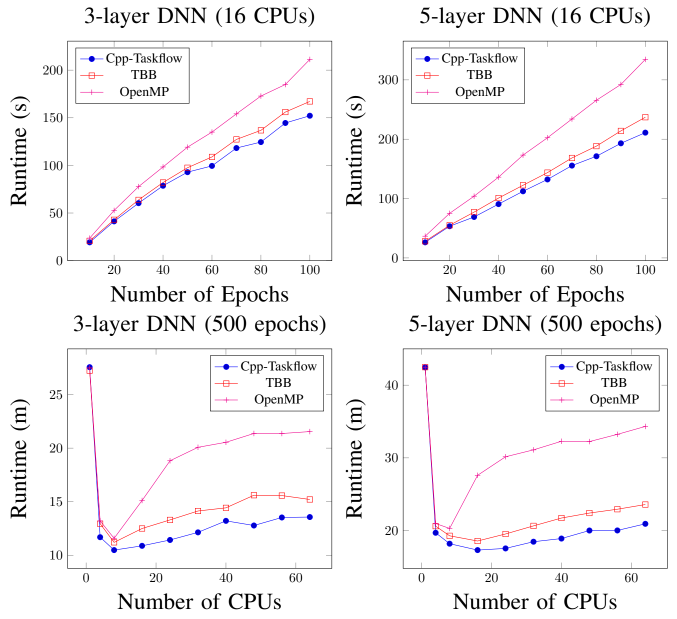

Fig. 12: Performance comparisons between Cpp-Taskflow, TBB, and OpenMP on training two different DNN classifiers.

Table III presents the software costs (reported by SLOCCount and Lizard [11], [12]) 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 hours). 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, but it does highlight the impact of a library's task model on engineering productivity.

Figure 12 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 CPUs. Under 16 CPUs, 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 CPU numbers. 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 CPUs. Similar trends can also be observed at other CPU configurations.

V. Acknowledgment

This work is supported by NSF Grant CCF-1718883 and DARPA Grant FA-8650-18-2-7843. The authors would like to thank all users of Cpp-Taskflow in providing insightful feedback and suggestions [4].

VI. Related Works

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 [8]. 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++ [17], [18], [19], [20]. On the other hand, Intel released in 2017 the Threading Building Blocks (TBB) library that supports loop-level parallelism and task-based programming (FlowGraph) [9]. 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 (DSL) and runtime to support tasking in a global address space (GAS) environment [21], [22], [23], [24], [25]. QURAK, StarPU, PaRSEC, and ParalleX are capable of tracking data between different memory and scheduling tasks on heterogeneous resources [26], [27], [28], [29]. While these systems are orthogonal to Cpp-Taskflow, we are leveraging their experience to handle new types of workload. An on-going project is using Cpp-Taskflow to improve TensorFlow's tasking kernel on heterogeneous architectures.

VII. Conclusion

In this paper, we have presented Cpp-Taskflow, a new C++ library to help developers quickly write parallel programs using the task-based approach. 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 flows. We have evaluated Cpp-Taskflow on both micro-benchmarks and real-world applications. On a machine learning example, Cpp-Taskflow achieved 1.5–2.7× less coding complexity and 14–38% speed-up over two industrial-strength libraries OpenMP Task Dependency Clause and Intel Threading Building Blocks (TBB) FlowGraph.

References

[1] E. Ayguad *et al.*, "An approach to task-based parallel programming for undergraduate students," *JPDC*, vol. 118, pp. 140–156, 2018.

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

[3] B. B. Fraguela, "A comparison of task parallel frameworks based on implicit dependencies in multi-core environments," in *HICSS*, 2017.

[4] "Cpp-Taskflow." [Online]. Available: https://github.com/cpp-taskflow

[5] J. Bhasker *et al.*, *Static Timing Analysis for Nanometer Designs: A Practical Approach.* Springer, 2009.

[6] J. Hu *et al.*, "TAU 2015 contest on incremental timing analysis," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.

[7] T.-W. Huang *et al.*, "OpenTimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.

[8] "OpenMP 4.5." [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

[9] "Intel TBB." [Online]. Available: https://github.com/01org/tbb

[10] "Intel Developer Zone." [Online]. Available: https://software.intel.com/en-us/node/506116

[11] "SLOCCount." [Online]. Available: https://dwheeler.com/sloccount/

[12] "Lizard." [Online]. Available: https://github.com/terryyin/lizard

[13] "OpenTimer." [Online]. Available: https://github.com/OpenTimer

[14] "MNIST." [Online]. Available: http://yann.lecun.com/exdb/mnist/

[15] "TensorFlow." [Online]. Available: https://www.tensorflow.org/

[16] "Eigen." [Online]. Available: https://eigen.tuxfamily.org/dox/

[17] R. D. Blumofe *et al.*, "Cilk: An efficient multithreaded runtime system," *JPDC*, vol. 37, no. 1, pp. 55–69, 1996.

[18] A. Duran *et al.*, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[19] J. M. Prez *et al.*, "A dependency-aware task-based programming environment for multi-core architectures." IEEE Computer Society, 2008, pp. 142–151.

[20] "Nanos++." [Online]. Available: https://pm.bsc.es/nanox

[21] H. Kaiser *et al.*, "HPX: A task based programming model in a global address space," in *PGAS*, 2014, pp. 6:1–6:11.

[22] B. Chamberlain *et al.*, "Parallel programmability and the chapel language," *IJHPCA*, vol. 21, pp. 291–312, 2007.

[23] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005, pp. 519–538.

[24] L. V. Kale *et al.*, "Charm++: A portable concurrent object oriented system based on C++," in *OOPSLA*, 1993, pp. 91–108.

[25] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," 2014.

[26] A. Yarkhan, "Dynamic task execution on shared and distributed memory architectures," *PhD thesis*, 2012.

[27] E. Agullo *et al.*, "Harnessing clusters of hybrid nodes with a sequential task-based programming model," in *PMAA*, 2014.

[28] G. Bosilca *et al.*, "PaRSEC : A programming paradigm exploiting heterogeneity for enhancing scalability," 2013.

[29] G. R. Gao *et al.*, "ParalleX: A study of a new parallel computation model," in *IPDPS*, 2007, pp. 1–6.