An Efficient Task-Parallel Pipeline Programming Framework

Cheng-Hsiang Chiu
University of Wisconsin-Madison
Madison, Wisconsin, USA
chenghsiang.chiu@wisc.edu

Zhicheng Xiong
Tsinghua University
Beijing, China
xiongz462@gmail.com

Zizheng Guo
Peking University
Beijing, China
gzz@pku.edu.cn

Tsung-Wei Huang
University of Wisconsin-Madison
Madison, Wisconsin, USA
tsung-wei.huang@wisc.edu

Yibo Lin
Peking University
Beijing, China
yibolin@pku.edu.cn

ABSTRACT

The pipeline is a fundamental pattern to parallelize a series of stage tasks over a sequence of data in loops. Mainstream pipeline programming frameworks count on data abstractions to perform pipeline scheduling. Although this design is convenient for data-centric parallel applications, it is not efficient for algorithms that only exploit task parallelism in the pipeline. To address the limitation, we introduce a new task-parallel pipeline programming framework called *Pipeflow*. Pipeflow separates data abstractions and task scheduling, enabling a more efficient implementation of task-parallel pipeline algorithms than existing frameworks. We have evaluated Pipeflow on both micro-benchmarks and real-world applications. For example, in a timing analysis workload that explores pipeline parallelism to speed up the runtime performance, the Pipeflow's implementation outperforms the oneTBB's implementation up to 110.33% faster.

CCS CONCEPTS

• **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

Task-parallel, Pipeline Parallelism, Pipeline Scheduling

ACM Reference Format:
Cheng-Hsiang Chiu, Zhicheng Xiong, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2024. An Efficient Task-Parallel Pipeline Programming Framework. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2024), January 25–27, 2024, Nagoya, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3635035.3635037

1 INTRODUCTION

The pipeline is a fundamental parallel pattern to model parallel executions through a linear chain of stages. Each stage processes a *data token* after the previous stage, applies an abstract function to that token, and then resolves the dependency for the next stage. Multiple data tokens can be processed simultaneously across different stages whenever dependencies are met. For example, in circuit simulation [26, 35–37], some operations on a gate (e.g., NAND, OR, AND) do not depend on other gates and thus can be done at multiple logic levels simultaneously, while operations at the same levels require processing prior levels first. As pipeline parallelism widely exists in modern computing applications [5], there is always a need for new pipeline programming frameworks to streamline the implementation complexity of pipeline algorithms.

Recently, several pipeline programming frameworks have emerged to assist developers in implementing pipeline algorithms without worrying about scheduling details, such as oneTBB [1], FastFlow [4], GrPPI [13], Cilk-P [50], SPar [15], and HPX-pipeline [46]. While each of these frameworks has its pros and cons, a common design philosophy is to perform data synchronizations using buffers between stages (i.e., *data abstraction*) in their pipeline scheduling designs, as illustrated in Figure 1. This design is convenient for data-centric pipeline applications but also has two limitations. Firstly, users have to design their pipeline algorithms in the data-parallel manner. However, data management is often application-dependent. Many applications exhibit pipeline parallelism among *tasks* rather than data. For example, the VLSI timing analysis application [9, 10] formulates a sequence of linearly dependent propagation tasks in a graph node and runs independent nodes in parallel to efficiently update the timing data from a custom global shared graph data structure. The real need is a *pipeline scheduling framework* to schedule and run tasks while leaving data management completely to applications.

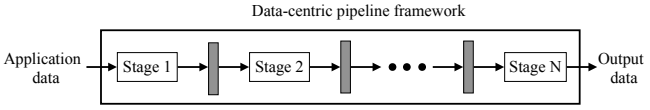

Figure 1: An illustration of data abstraction in a pipeline framework. Gray bars are buffers used for data synchronizations.

Secondly, scheduling algorithms involve complex synchronizations between data and buffer structures. These frameworks typically leverage object allocators and buffer structures to manage temporary data between stages (e.g., oneTBB [1]). However, the synchronizations can be redundant in some applications. For instance, ferret [5], a pipeline benchmark of PARSEC, defines six stages (loading, segmentation, extraction, indexing, ranking, and output) in its

HPCAsia 2024, January 25–27, 2024, Nagoya, Japan

ACM ISBN 979-8-4007-0889-3/24/01…$15.00
https://doi.org/10.1145/3635035.3635037

oneTBB implementation to perform image similarity search. Every stage is defined as a derived class of oneTBB's `tbb::filter` and has an overridden operator that takes an input `void*` pointer returned from the previous stage. The pointer points to a *global* data structure `all_data`, bypassing all the data abstractions in the oneTBB pipeline.

To overcome these limitations, we introduce in this paper *Pipeflow*, a new task-parallel pipeline programming framework. We summarize our contributions as follows:

- **Task-Parallel Pipeline.** We have introduced a new *task-parallel* pipeline programming concept that separates task scheduling and data abstraction. This separation allows us to concentrate on the pipeline tasking itself, enabling a more efficient implementation of task-parallel pipeline algorithms than existing frameworks.
- **Programming Model.** We have introduced a new C++ programming model to support our concept. Unlike existing models, we do not provide yet another data abstraction but a flexible framework for users to fully control their application data atop a task-parallel pipeline scheduling framework.
- **Scheduling Algorithm.** We have introduced a new scheduling algorithm to schedule stage tasks across parallel lines. Since we do not touch data abstraction, we can avoid complex data buffer designs and synchronization mechanisms to enable more lightweight and efficient scheduling.

We have evaluated Pipeflow on both micro-benchmarks and real-world applications. For example, in a real-world VLSI static timing analysis workload, the Pipeflow implementation outperforms the oneTBB implementation up to 110.3% faster. Right now, Pipeflow is merged into the open-source Taskflow project [33].

2 BACKGROUND

We first review the pipeline basics and then detail the motivation of Pipeflow. We then argue a new task-parallel pipeline programming model is needed for many important industrial and research areas, e.g., circuit design.

2.1 Pipeline Basics

Pipeline parallelism is commonly used to parallelize various applications, such as stream processing, video processing, and dataflow systems. These applications exhibit parallelism in the form of a *linear pipeline*, where a linear sequence of abstraction functions, namely *stages*, $F = \langle f_1, f_2, \cdots, f_j \rangle$, is applied to an input sequence of data tokens, $D = \langle d_1, d_2, \cdots, d_i \rangle$. A linear pipeline can be thought of as a loop over the data tokens of D. Each iteration i processes an input token d_i by applying the stage functions F to d_i in order. Depending on the number of *parallel lines*, $L = \langle l_1, l_2, \cdots, l_k \rangle$, to process data tokens, parallelism arises when iterations overlap in time. For instance, the execution of token d_i at stage f_j of parallel line l_k, denoted as $f_j^k(d_i)$, can overlap with $f_{j-1}^{k+1}(d_{i+1})$. A stage can be a *parallel* type or a *serial* type to specify whether $f_j^k(d_i)$ can overlap with $f_j^{k+1}(d_{i+1})$ or not. Figure 2 shows the dependency diagram of a 3-stage (serial-serial-parallel) pipeline.

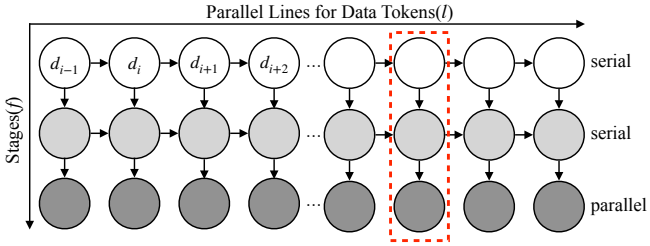

Figure 2: Dependency diagram of a 3-stage (serial-serial-parallel) pipeline. Each node represents a task that applies a stage function to a data token. Each edge represents a dependency between two tasks. The dashed rectangle denotes one parallel line.

2.2 Task-parallel Pipeline Parallelism

Pipeflow is motivated by our research projects on developing parallel timing analysis algorithms for very large scale integration (VLSI) computer-aided design (CAD) [17, 18, 20, 21, 25, 26, 75]. Timing analysis is a critical step in the overall CAD flow because it validates the timing performance of a digital circuit. As design complexity continues to grow exponentially, the need to efficiently analyze the timing has become the major bottleneck to the design closure flow. For instance, generating a comprehensive timing report (e.g., pessimism removal, hundreds of corners, etc.) for a multi-million-gate design can take several hours [45]. To reduce the analysis runtime, there is an increasing trend of adopting manycore parallelism by new timing analysis algorithms recently [8, 19, 22, 27, 38–41, 44, 54, 59].

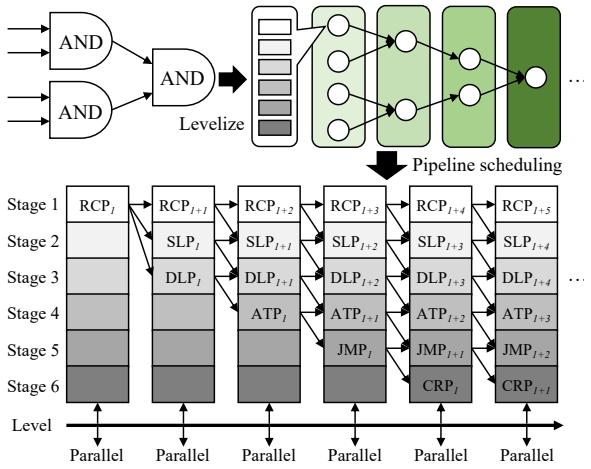

Figure 3: Parallel timing propagations [34]. Linearly dependent timing data (e.g., slew) is updated across graph nodes in a task-parallel pipeline fashion.

The most widely used strategy, including commercial timers, to parallelize timing analysis is *pipeline*. Figure 3 illustrates this strategy using forward timing propagation as an example [34]. The circuit graph is first levelized into a level list using topological sort. Nodes at the same level are independent of each other and can run in parallel. Each node runs a sequence of *linearly dependent*

propagation tasks, including parasitics (RCP), slew (SLP), delay (DLP), arrival time (ATP), jump points (JMP), and common path pessimism reduction (CRP) to update its timing data from a *custom global* and *application-dependent* circuit graph data structure. Different propagation tasks can overlap across different levels using pipeline parallelism.

This type of *task-parallel* pipeline strategy is ubiquitous in many parallel CAD algorithms, such as logic simulation [49, 58] and physical design [29, 30, 32, 48], because computations frequently flow through circuit networks. We have observed three important properties that make mainstream pipeline programming frameworks fall short of our needs: 1) Unlike the typical data-parallel pipeline, the pipeline parallelism in many CAD algorithms is driven by *tasks* rather than data. 2) Data is not directly involved in the pipeline but in the *graph data structure* defined by a custom algorithm. 3) From a user's standpoint, the real need is a *pipeline scheduling framework* to help schedule and run tasks on input tokens across parallel lines while leaving data management completely to applications; in our experience, users disfavor another library data abstraction to perform pipeline scheduling, as it often incurs development inconvenience and unnecessary data conversion overheads.

3 PIPEFLOW

Inspired by the need for parallel CAD algorithms, Pipeflow introduces a new task-parallel pipeline programming model for users to create a pipeline scheduling framework without data abstraction. In this section, we will dive into the technical details of Pipeflow.

3.1 Programming Model

Pipeflow leverages modern C++ and template techniques to strike a balance between expressiveness and generality. Listing 1 shows the Pipeflow code that implements the pipeline in Figure 2. Pipeflow has one API `Pipeline` that allows users to define the pipeline structure and explore the pipeline parallelism in their applications. There are two steps to create a Pipeflow application, 1) define the pipeline structure using template instantiation using the `Pipeline` API and 2) define the application data storage, if needed. In Pipeflow, the terms "pipe" and "stage" are interchangeable. For the first step, users define the number of parallel lines and the abstract function of each pipe in a `Pipeline` object. For each pipe, users define the pipe type and a pipe callable using `Pipe`. A pipe can be either a serial type (`PipeType::SERIAL`) or a parallel type (`PipeType::PARALLEL`). The pipe callable takes an argument of `Pipeflow` type, which is created by the scheduler at runtime. A `Pipeflow` object `pf` represents a *scheduling token* and contains several methods for users to query the runtime statistics of that token, including the parallel line, pipe, and token numbers.

```
const size_t num_lines  = 4;
std::variant<float, std::string> data_type;
std::array<data_type, num_lines> buffer;
Pipeline pl(num_lines,
  // First pipe
  Pipe{PipeType::SERIAL,
    [&](Pipeflow& pf) {
      if ( !data.ready() ) {
        pf.stop();
      } else {
        // Generate a float and save it in buffer
        buffer[pf.line()] = data.get();
      }
    }
  },
  // Second pipe
  Pipe{PipeType::SERIAL,
    [&](Pipeflow& pf) {
      // Generate a string and save it in buffer
      buffer[pf.line()] =
        make_string(std::get<0>(buffer[pf.line()]));
    }
  },
  // Third pipe
  Pipe{PipeType::PARALLEL,
    [&](Pipeflow& pf) {
      std::cout << std::get<1>(buffer[pf.line()]);
    }
  }
);
pl.run();
```

Listing 1: Pipeflow code of Figure 2, assuming the first pipe generates `float` and the second pipe generates `string` outputs.

Pipeflow does not have any data abstraction but gives applications full control over data management. In our example, since the first and the second pipes generate `float` and `std::string` outputs, respectively, we create a one-dimensional (1D) array, `buffer`, as the application data storage to store data in uniform storage using `std::variant<float, std::string>`. The dimension of the array is equal to the number of parallel lines, as Pipeflow schedules only one token per parallel line. Each entry `buffer[i]` stores the data that is being processed at parallel line i, which can be retrieved by `Pipeflow::line`. This organization is space-efficient because we use only a 1D array to represent data processing in a two-dimensional (2D) scheduling map. Additionally, by delegating data management to applications, we can avoid dynamic data conversion between the library and the application, which typically counts on virtual function calls to convert a generic type (e.g., *void**, *std::any*) to an arbitrary user type [1, 4].

Based on the pipeline structure and data layout defined above, we instantiate a `Pipeline` object, `pl`. This template-based design enables the compiler to optimize each pipe type, such as using a fixed-layout functor to store the callable and its captured data. Finally, we call `run` to submit the object `pl` to a runtime and execute it.

```
using P = Pipe<std::function<void(Pipeflow&)>>;
std::vector<P> p(6, create_pipe());  // 6 pipes
// Pipeline of 4 parallel lines and 6 pipes
ScalablePipeline pl(4, p.begin(), p.end());
pl.run(); // First run
p.resize(3); // Resize p to 3 pipes
// Pipeline of 4 parallel lines and 3 pipes
pl.reset(p.begin(), p.end());
pl.run(); //Second run
```

Listing 2: Scalable pipeline model in Pipeflow to accept variable assignments of pipes.

`Pipeline` requires instantiation of all pipes at the construction time. While this design gives compilers more freedom to optimize the layout of each pipe type, it prevents applications from varying the pipeline structure at runtime; for instance, the number of pipes might depend on the problem size, which can be runtime variables. To overcome this limitation, Pipeflow provides a scalable alternative, `ScalablePipeline`, to allow variable assignments of pipes using

range iterators. In Listing 2, we create a scalable pipeline, `pl`, from a vector of six pipes `p`. After the first run, we reset `pl` to another range of three pipes for the second run. A scalable pipeline is thus more flexible for applications to create a pipeline scheduling framework with dynamic structures.

Compared to the programming model of existing frameworks, such as oneTBB [1], Pipeflow's programming model has the following advantages: 1) Pipeflow is expressive and easy to write. Users only need pipe type, pipe callable, and the number of parallel lines to create a pipeline scheduling framework and explore the pipeline parallelism in their applications. Moreover, as Pipeflow does not provide data abstraction, users do not need to explicitly specify the input data and output data type at every pipe definition as oneTBB's users do. 2) Pipeflow is flexible. Users are able to modify the pipeline structure at runtime based on their specific needs.

3.2 Scheduling Algorithm

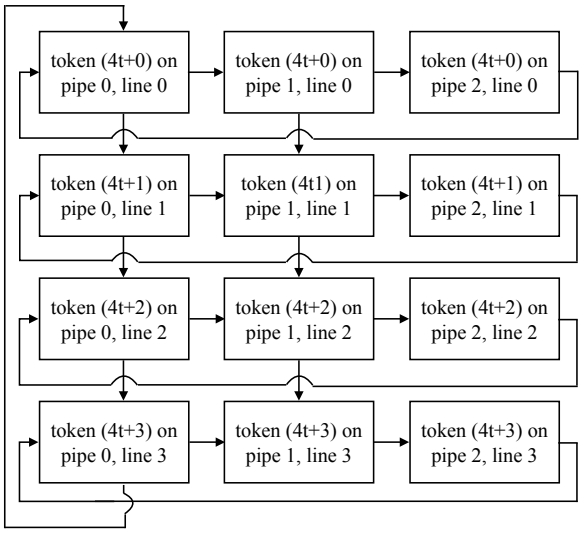

Figure 4: The scheduling diagram of the task-parallel pipeline in Listing 1. Each parallel line runs one scheduling token. Multiple parallel lines overlap tokens in a circular fashion. The text "token (4t+1) on pipe 1, line 1" means the token with ID `4t+1` runs on the pipe 1 **and the parallel line** 1**.**

As Pipeflow does not touch data abstraction, we can simplify the pipeline scheduling problem to decide which scheduling token to run at which pipe and parallel line. Our scheduling algorithm places only one scheduling token per parallel line. We then process all tokens in a circular fashion across the given number of parallel lines. Figure 4 illustrates our pipeline scheduling idea using the pipeline in Listing 1. Since the pipeline schedules tokens in a circular fashion, there are four edges (dependencies) from the last pipes (`pipe 2`) to the first pipes (`pipe 0`), and one edge from the first pipe of the last parallel line to the first pipe of the first parallel line. The last pipe (`pipe 2`) is a parallel type. There is no vertical edge between the last pipes of two consecutive parallel lines. Each parallel line runs only one scheduling token. Multiple parallel lines can overlap tokens whenever their dependencies are met. Even though the pipeline execution can involve many scheduling tokens, only four parallel lines are used in total.

3.3 Pseudocode

Algorithm 1: define_task(l)

```
   global: pipeflows: a vector of Pipeflow objects
   global: join_counters: a 2D array of join counters
   global: num_tokens: the number of tokens
   global: num_lines: the number of parallel lines
   global: num_pipes: the number of pipes
   Input: l: a parallel line id
 1 pf ← pipeflows[l];
 2 AtomicStore(join_counters[pf.line][pf.pipe], pf.join_counter);

 3 if pf.pipe == 0 then
 4 |   pf.token ← num_tokens;
 5 |   invoke_pipe_callable(pf);
 6 |   if pf.stop == True then
 7 |   |   return;
 8 |   end
 9 |   num_tokens = num_tokens + 1;
10 end
11 if pf.pipe != 0 then
12 |   invoke_pipe_callable(pf);
13 end
14 curr_pipe ← pf.pipe;
15 next_pipe ← (pf.pipe + 1)%num_pipes;
16 next_line ← (pf.line + 1)%num_lines;
17 pf.pipe ← next_pipe;
18 next_tasks = {};
19 if curr_pipe is SERIAL and
     AtomicDecrement(joun_counters[next_line][curr_pipe]) ==
     0 then
20 |   next_tasks.insert(1);
21 end
22 if
     AtomicDecrement(join_counters[pf.line][next_pipe]) ==
     0 then
23 |   next_tasks.insert(0);
24 end
25 if next_tasks.size == 2 then
26 |   call_scheduler(task_of_next_line);
27 |   goto Line 2;
28 end
29 if next_tasks.size == 1 then
30 |   if next_tasks[0] == 1 then
31 |   |   pf ← pipeflows[next_line];
32 |   end
33 |   goto Line 2;
34 end
```

Based on the idea discussed in Section 3.2, we formulate each parallel line as a task, which defines a function object to run by a

thread in the thread pool. Each task 1) deals with one scheduling token per parallel line and 2) decides which adjacent task to run on its next parallel line and pipe. Algorithm 1 implements such a task using efficient atomic operations. When a task is scheduled, we need to know which pipe at which parallel line for the scheduling token to work. We keep the parallel line and pipe information in a Pipeflow object. Each task owns a Pipeflow object `pf` of a specific parallel line (line 1). Once a scheduling token is done, there are two cases for its corresponding task to proceed: 1) for a parallel type, the task moves to the next pipe at the same parallel line; 2) for a serial type, the task additionally checks if it can move to the next parallel line. To carry out such a dependency constraint, each pipe keeps a join counter of an *atomic* integer to represent its dependency value. The values of a serial pipe and a parallel pipe can be up to 2 and 1, respectively. We create a 2D array `join_counters` to store the join counter of each pipe at each parallel line. Line 2 initializes these join counters to either 2 or 1 based on the corresponding pipe types that are enumerated on integer constants, 2 (serial) and 1 (parallel). At the first pipe (line 3), the Pipeflow object updates its token number (line 4) and checks if the pipe callable requests to stop the pipeline (lines 5:8). If continued, we increment the number of scheduled tokens by one (line 9). For other pipes, we simply invoke the pipe callables (lines 11:13).

After the pipe callable returns, we update the join counters based on the pipe type and determine the next possible tasks to run (lines 14:24). When the join counter of a pipe becomes 0, we bookmark this pipe as a task to run next (line 20 and line 23). If two tasks exist (line 25), the current task informs the scheduler to call a worker thread to run the task at the next parallel line (line 26) and reiterates itself on the next pipe (line 27). The idea here is to facilitate data locality as applications tend to deal with the next pipe at the same parallel line as soon as possible. If there is only one task available, the current task directly runs the next task with the updated `pf` object (lines 29:34).

Compared to existing algorithms, such as oneTBB [1], that count on non-trivial synchronization between tasks and internal data buffers, our algorithm focuses on the task parallelism of a pipeline itself. This design largely reduces the scheduling complexity of pipeline by using lightweight atomic operations without complex data buffer management.

3.4 Proof

We draw the following lemmas and sketch their proofs to highlight the correctness of our scheduling algorithm:

Lemma 1. *Only one task runs a pipe callable (line 5 and line 12 in Algorithm 1) on a scheduling token.*

Proof. Assume two tasks are running the same pipe callable, which means one task reiterates its execution from the previous pipe, and the other task comes from the previous parallel line. This is not possible in a parallel pipe as there is no dependency from the previous parallel line; only one runtime task decrements the join counter to 0 (line 22 in Algorithm 1). Take Figure 4 for example. There is no vertical edge pointing to token `4t+1` from token `4t+0` for pipe 2. Only the task that runs token `4t+1` on pipe 1 gets to decrement the join counter for task `4t+1` on pipe 2. In a serial pipe, this is also not possible because the dependency is resolved using atomic operations; only one task will acquire the zero value of the join counter (line 19 in Algorithm 1). For example, in Figure 4, either the task running token `4t+0` on pipe 1 or the task running token `4t+1` on pipe `0` decrements token `4t+1` on pipe 1 to zero and then runs the pipe. □

Lemma 2. *The scheduler does not miss any pipe.*

Proof. We consider the situation where one task moves to the next parallel line (line 31 in Algorithm 1) instead of the next pipe at the same parallel line. Under this circumstance, we need to make sure one task will run that next pipe. Take Figure 4 for example. Suppose a task finishes token `4t+1` at pipe `0` and precedes to token `4t+2` at pipe `0`, meaning that the join counter of token `4t+1` at pipe 1 is not 0 yet. Another task that works on token `4t+0` at pipe 1 will eventually decrement the join counter to run it (line 27 in Algorithm 1) or invoke another worker thread to run it (line 26 in Algorithm 1). □

4 EXPERIMENTAL RESULTS

We implemented Pipeflow using C++17 and evaluated the performance of Pipeflow on a micro-benchmark and a real-world industrial CAD application. We studied the performance across memory (RSS), runtime, and throughput. We did not use conventional pipeline benchmarks (e.g., PARSEC's ferret[5] has only six pipes, loading, segmentation, extraction, indexing, ranking, and output) as their problem sizes are relatively small compared to CAD, and the runtime difference between Pipeflow and the baseline is not obvious on small pipelines. We compiled all programs using g++12 with `-std=c++17` and `-O3` enabled. We ran all the experiments on a Ubuntu Linux 19.10 (Eoan Ermine) machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz and 256 GB RAM. All data is an average of ten runs.

4.1 Baseline

Given a large number of pipeline programming frameworks, it is infeasible to compare Pipeflow with all of them. We considered oneTBB [1] as our baseline for two reasons. First, oneTBB is the only library that provides a single pipeline API for users to explore pipeline parallelism. Others require combining several library-specific constructs to achieve this goal. For example, in Cilk-P users need `pipe_while`, `pipe_stage`, and `pipe_stage_wait` to add pipeline parallelism in applications. Second, Pipeflow is inspired by our CAD applications, and oneTBB is widely used in the CAD community due to its absolute speed and robustness. For a fair comparison, we implemented the same work-stealing strategy as oneTBB in our thread pool, in particular, `call_scheduler` in line 26 in Algorithm 1.

4.2 Micro-benchmark

The purpose of micro-benchmarks is to measure the pure scheduling performance without much computation bias from applications. We compared the memory, runtime, and throughput between Pipeflow and oneTBB for completing pipelines of different numbers of serial pipes, scheduling tokens, and threads. We did not use parallel pipes as their callable can be absorbed into the previous serial

pipe. Each pipe performs a nominal work of constant space and time complexity (e.g., small matrix multiplications) and forwards a scheduling token to the next pipe.

Figure 5 illustrates the maximum RSS between Pipeflow and oneTBB with different scheduling tokens and threads. The number of parallel lines and pipes of a pipeline is equal to the number of threads. We can see that oneTBB starts to consume more memory than Pipeflow as we increase the pipeline size. For example, with 2^{10} scheduling tokens Pipeflow needs 1.97% and 11.68% less memory than oneTBB when running with 16 and 64 threads, respectively. The same trend is also observed in the plot of processing 2^{15} scheduling tokens. In terms of memory usage, oneTBB is consistently higher than Pipeflow (e.g., 97.72% higher with 64 threads and 2^{15} scheduling tokens) because we do not manage any data buffers but focus on the task scheduling itself. That is, oneTBB needs to allocate space for its internal data buffer structures to perform pipeline scheduling. We can see the overhead of using data abstractions in pipeline scheduling next.

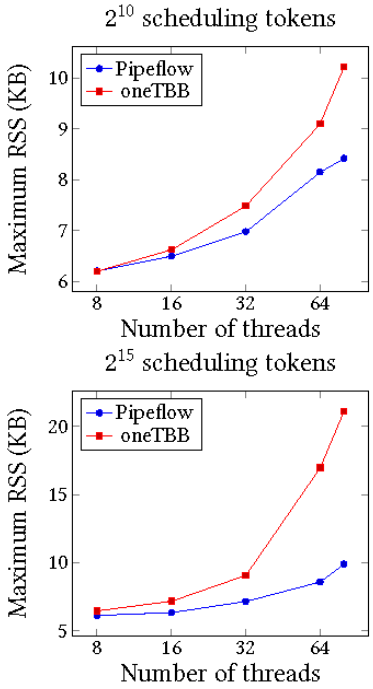

Figure 5: Maximum RSS comparison between Pipeflow and oneTBB with different threads and two scheduling tokens (2^{10} and 2^{15}) for the micro-benchmark. The number of threads is the same as the number of pipes in the pipeline.

Figure 6 draws the runtime comparisons between Pipeflow and oneTBB under different scheduling tokens and thread counts. The number of parallel lines and pipes of a pipeline is equal to the number of threads. We can see that the runtime gap between Pipeflow and oneTBB starts to increase as we increase the pipeline size. For example, at 2^{15} scheduling tokens Pipeflow runs 10.13%, 10.98%, 124.18%, and 201.38% faster than oneTBB with 8, 16, 64, and 80 threads, respectively. Furthermore, Pipeflow has better runtime performance than oneTBB in all situations. We attribute the performance improvements of Pipeflow over oneTBB to the reason that oneTBB relies on its internal data buffer to perform pipeline scheduling while Pipeflow only uses lightweight atomic operations. Moreover, as the number of threads is equal to the number of parallel lines and pipes in the experiment, the pipeline running with 80 threads has a bigger structure than the pipeline running with 8 threads. As a result, the former pipeline exhibits higher task scheduling overhead than the latter and thus spends more time to finish. Although the micro-benchmark only demonstrates the pure scheduling performance and forwards the scheduling token between pipes, the overhead of data abstraction design of oneTBB results in a significant runtime difference, especially in a large pipeline.

Figure 7 compares the throughput by corunning the same program up to 8 times. Corunning a program at different configurations is very common in some applications, such as [45]. The experiment emulates a server-like environment where different pipeline applications compete for the same resources. We use the weighted speedup to measure the system throughput, which is the sum of the individual speedup of each process over a baseline execution time[14]. A throughput of one implies that the corun throughput is the same as if those processes run consecutively. On the left plot, the pipeline has 16 pipes and 16 parallel lines and runs with 16 threads. On the right plot, the pipeline has 80 pipes and 80 parallel lines and runs with 80 threads. Both of them run 2^{15} scheduling tokens. We can see that Pipeflow outperforms oneTBB in all coruns. For example, at 8 coruns Pipeflow is 1.2x and 3.31x better than oneTBB with 16 and 80 pipes, respectively. Besides, Pipeflow remains around one up to 5 coruns while oneTBB decreases after 2 coruns. We attribute the finding to the reason that we use lightweight atomic operations rather than complex data buffer synchronizations to do the task scheduling. As Pipeflow is more lightweight than oneTBB in pipeline scheduling, corunning Pipeflow thus has better throughput than corunning oneTBB.

The above experiments were running with the configuration in which the number of pipes is the same as the number of threads. However, this configuration does not always guarantee the best runtime performance because of different applications and hardware environments. Next, we see how the number of threads could impact the performance. Figure 8 shows the impact of Pipeflow and oneTBB running with different numbers of threads in a small (16 pipes) and a big (80 pipes) pipeline in which 2^{10} and 2^{15} tokens were scheduled. For 16 pipes, we can see that the runtime trends of running 2^{10} and 2^{15} are similar. oneTBB has the best runtime performance with 32 threads; Pipeflow has the best performance at 16 threads. For 80 pipes, the runtime trends are the same. We find out that Pipeflow has the best performance with 80 threads while oneTBB with 32 threads. In this micro-benchmark, we know the selection of identical pipes and threads is the best option for Pipeflow but not for oneTBB. Selecting the best thread number is an important factor and the best thread number for one application may not be the best choice for another application.

4.3 VLSI Static Timing Analysis Algorithm

We applied Pipeflow to solve a large-scale VLSI static timing analysis (STA) problem. The goal is to analyze the timing landscape of a

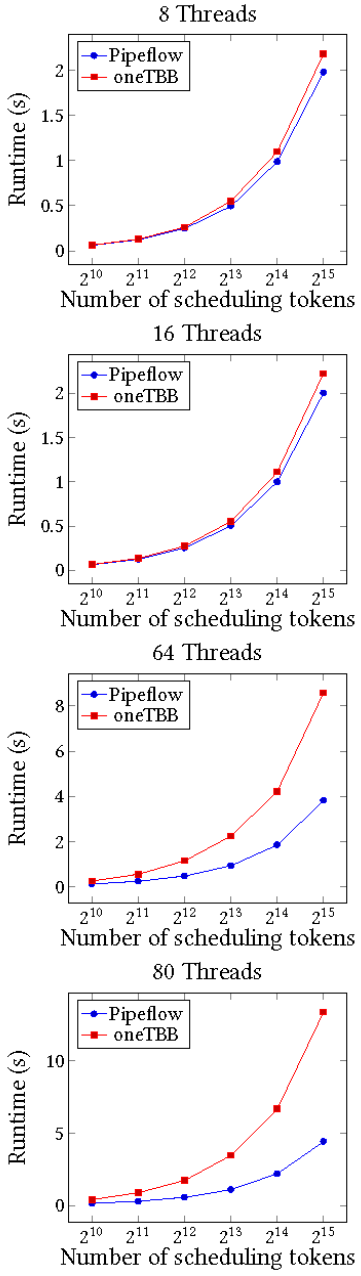

Figure 6: Runtime comparison between Pipeflow and oneTBB with different scheduling tokens and threads for the micro-benchmark. The number of threads is the same as the number of pipes in the pipeline.

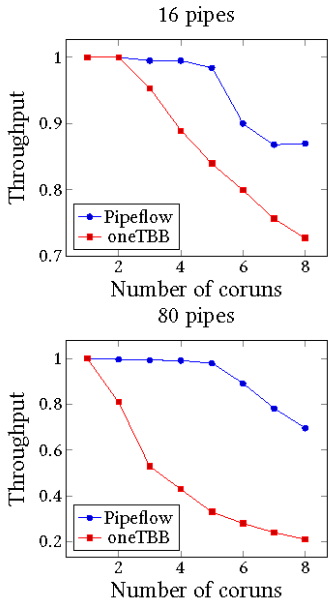

Figure 7: Throughput of corunning micro-benchmark programs with 16 and 80 pipes and 2^{15} scheduling tokens.

circuit design and report critical paths that do not meet the given constraints (e.g., setup and hold). As presented in Figure 3, modern STA engines leverage pipeline parallelism to speed up the timing propagations. However, nearly all of them count on OpenMP-based loop parallelism with layer-by-layer synchronization [34]. With Pipeflow, we can directly formulate the problem as a task-parallel pipeline to improve task asynchrony. As the analysis complexity continues to increase, more analysis tasks (e.g., RC, delay calculators, pessimism reduction) are incorporated into each node in the STA graph. These tasks can be encapsulated in a sequence of pipe functions to overlap in the graph across parallel lines. We modified a large circuit design of 1.5M nodes and 3.5M edges from [3, 34] and studied the performance under different pipe counts. Each node has a pipe task to calculate delay values at a specific configuration using linear interpolation. We levelized the STA graph and ran the nodes at the same level in parallel, such that different analysis tasks overlaps across different levels using pipeline parallelism, as illustrated in Figure 3.

Figure 9 evaluates the memory usage between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts. The number of pipes and lines in the pipeline is identical to the number of threads. As the pipeline size grows, the gap of memory usage starts to increase in both 1.5M and 5M graph sizes. For instance, with 1.5M graph size Pipeflow needs 0.07% and 5.6% less than oneTBB at 32 and 80 threads, respectively. We can observe a similar trend when we process 5M graph size. Since Pipeflow delegates data management directly to applications without touching data abstractions, Pipeflow does not allocate as much memory as oneTBB to perform pipeline scheduling.

Figure 10 compares the runtime performance between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts. The number of pipes and parallel lines of the pipeline is the same as the number of threads in this experiment. We can see that Pipeflow outperforms oneTBB when we increase the graph size. Taking 16 threads for example, Pipeflow runs 62.02%, 57.44%, and 46.08% faster than oneTBB with 1, 3, and 5M graph size, respectively. The performance improvements reduce because the overhead of setting

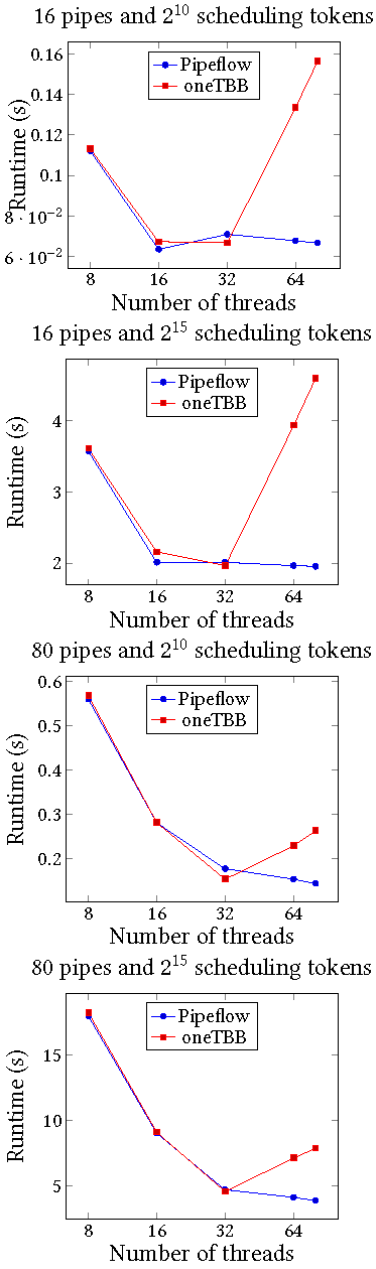

Figure 8: Impacts of selecting the number of threads on the runtime performance for the micro-benchmark. The number of pipes is not identical to the number of threads. The numbers of pipes are 16 and 80. The pipeline processes 2^{10} and 2^{15} scheduling tokens.

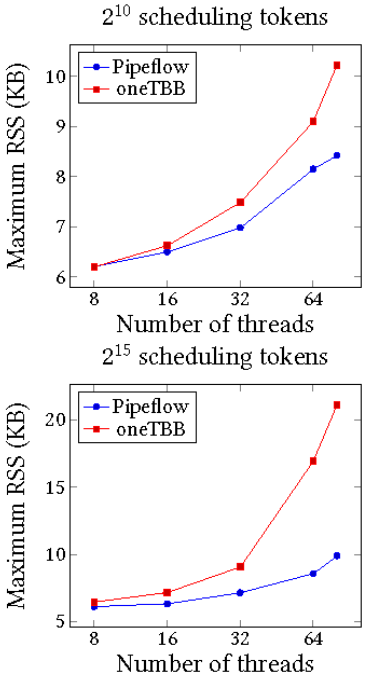

Figure 9: Maximum RSS comparison between Pipeflow and oneTBB at different graph sizes ($\|V\|+\|E\|$) and thread counts for the timing analysis workload. The number of threads is identical to the number of pipes in the pipeline.

up internal data buffers in oneTBB is gradually amortized when we increase the graph size in this workload. We also notice the runtime gap decreases when we use more threads. For example, at 5M graph size Pipeflow is 110.33%, 46.08%, and 20.08% faster with 8, 16, and 64 threads, respectively. The improvements also reduce because the cost of data buffers is amortized gradually as the pipeline size grows. Despite the runtime improvements gradually decrease when the pipeline size grows or the graph size increases, Pipeflow still outperforms oneTBB in all cases in the workload. Since our scheduling algorithm does not deal with data passing between pipes, we can process scheduling tokens more efficiently than oneTBB. In Pipeflow all pipe tasks perform computations directly on a global graph data structure captured in the pipe callable instead of passing data between successive pipes using buffers. The data passing interface between successive pipes in oneTBB thus becomes an unnecessary overhead.

Next, we compare the throughput by corunning the same program up to 8 times. Corunning the STA program is very common for reporting the timing data of a design at different input library files[45]. The effect of pipeline scheduling propagates to all simultaneous processes. Hence, throughput is a good measurement for the inter-operability of a pipeline-based STA algorithm. We corun the same analysis program up to 8 processes that compete for 40 cores. Again, we use the weighted speedup to measure the throughput. Figure 11 plots the throughput across 8 coruns at 16 and 80 pipes. The number of pipes is identical to the number of threads. We can see that Pipeflow outperforms oneTBB at all coruns. For instance, at 8 coruns Pipeflow is 1.04x and 1.14x better than oneTBB with 16 and 80 pipes, respectively. This is because Pipeflow leverages lightweight atomic operations and oneTBB relies on complex

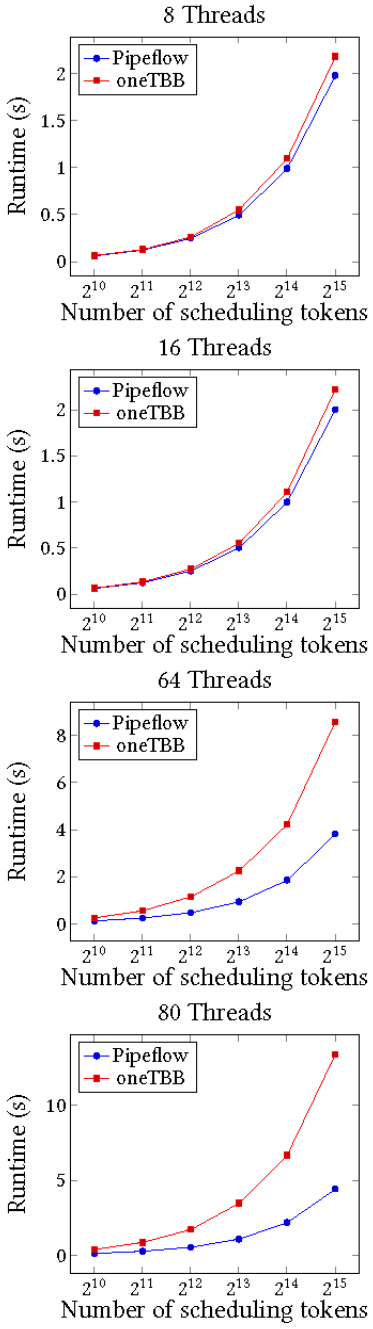

Figure 10: Runtime comparison between Pipeflow and oneTBB at different graph sizes ($\|V\| + \|E\|$) and thread counts for the timing analysis workload. The number of threads is identical to the number of pipes in the pipeline.

data buffer management in pipeline scheduling. Corunning a light-weight program has a higher throughput than corunning a heavy program. Besides, with more coruns both Pipeflow and oneTBB have a decreasing throughput.

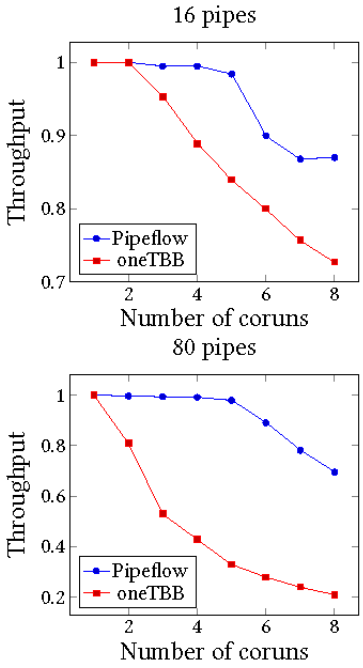

Figure 11: Throughput of corunning STA programs with 16 and 80 pipes and 1.5M graph size($\|V\| + \|E\|$).

So far, we ran the experiments of this workload using the number of threads same as the number of pipes. This configuration may not always give us the best runtime performance because of different hardware environment and workloads. Next, we demonstrate the importance of selecting the number of threads in this workload. Figure 12 shows the runtime performance of Pipeflow and oneTBB processing 1.5M and 5M graph size with different numbers of threads in 16-pipe and 80-pipe pipelines. For 16 pipes, we observe that both Pipeflow and oneTBB have a similar runtime trend, and both achieve the best performance with 64 threads for 1.5M graph size. With 5M graph size Pipeflow has the best performance with 80 threads while oneTBB with 64 threads. For 80 pipes, both Pipeflow and oneTBB have the best performance with 32 threads. From Figure 12 we learn that the alignment of threads and pipes does not achieve the best runtime performance in the workload. Hence, selecting the thread counts is an important factor while exploring pipeline parallelism in applications.

4.4 Importance of Task-Parallel Pipeline

As experienced parallel CAD researchers, Pipeflow has assisted us in overcoming many programming challenges. For example, in the previous experiments, the data is explicitly managed by the application algorithms and there is no need to go through any data abstraction. The real need is a task-parallel pipeline programming framework that 1) gives applications full control over data and 2) allows applications to probe each scheduled task. For instance, when implementing the STA algorithm, we captured the data from a global STA graph structure in each pipe callable and used the

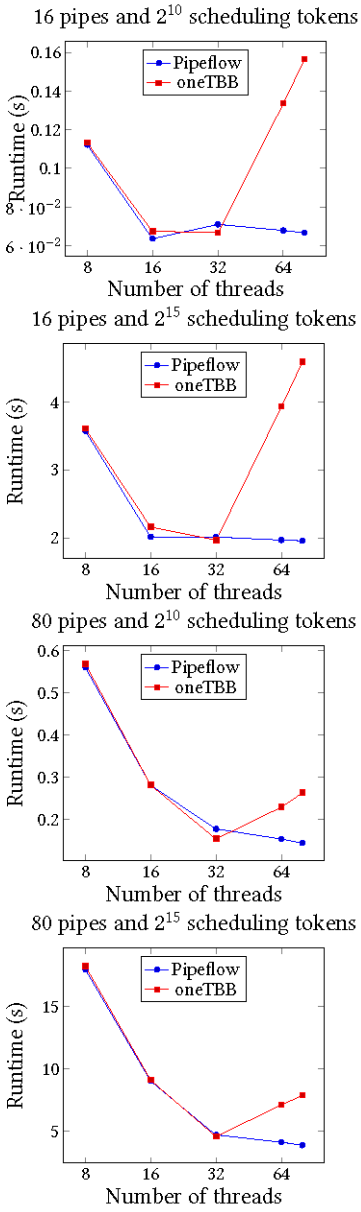

Figure 12: Impacts of selecting the number of threads on the runtime performance for the timing analysis workload. The number of pipes is not identical to the number of threads. The numbers of pipes are 16 and 80. The graph sizes ($\|V\| + \|E\|$) are 1.5M and 5M.

`pipeflow` variable to get the parallel line number of a scheduled task to index the corresponding entry in a result vector. However, oneTBB abstracts these components out, and we have to implement another mapping strategy to get these data from each filter, both of which incur significant yet unnecessary runtime overheads. Similar situations exist in other libraries too.

4.5 Selection of the Number of Parallel Lines

Selecting the number of parallel lines (or threads) for the best performance is application-dependent. For example, Figure 8 illustrates that Pipeflow obtains the best performance while aligning the pipe sizes and thread counts and oneTBB should run with 32 threads in the micro-benchmark. From Figure 12 when running the STA workload, both Pipeflow and oneTBB obtain the best runtime results with 32 threads in an 80-pipe pipeline. From the micro-benchmark and STA algorithm, we learn that the selection of the number of parallel lines (or threads) is a critical factor regarding the runtime performance. Moreover, as the performance of an application tends to saturate or peak at a certain limit, increasing the number of parallel lines exceeds the limit could negatively affect the runtime. As a result, Pipeflow makes the number of parallel lines a tunable parameter (similarly in oneTBB). Based on our experience, most applications can obtain decent performance when the number of parallel lines is equal to the number or twice the number of the cores.

5 RELATED WORK

Pipeline programming models. have received intensive research interest. Most of them are data-centric, using static template instantiation or dynamic runtime polymorphism to model data processing in a pipeline. To name a few popular examples: oneTBB [1] and TPL [51] require explicit definitions of input and output types for each stage; GrPPI [13] provides a composable abstraction for data- and stream-parallel patterns with a pluggable back-end to support task scheduling; FastFlow [4] models parallel dataflow using predefined sequential and parallel building blocks; TTG [7] focuses on dataflow programming using various template optimization techniques; SPar [15, 16, 24, 60] analyzes annotated attributes extracted from the data and stream parallelism domain, and automatically generates parallel patterns defined in FastFlow; Proteas [61] introduces a programming model for directive-based parallelization of linear pipeline; [73, 74] propose a self-adaptive mechanism to decide the degree of parallelism and generate the pattern compositions in FastFlow; OpenMP [2] uses task construct and depend clause to explore pipeline parallelism. Although these programming models are used in many applications, such as oneTBB in PARSEC [5], they constrain users to design pipeline algorithms using their data models, making it difficult to use, especially for applications that only need pipeline scheduling atop custom data structures. Pipeflow simply requires users to specify the pipeline structures (e.g., the number of parallel lines) and pipe callables, and provides a scalable pipeline API for users to flexibly define the pipeline scheduling frameworks with dynamic structures based on their specific needs.

Existing pipeline scheduling algorithms. typically co-design task scheduling and buffer structures to strive for the best performance. For instance, oneTBB [1] defines a per-stage buffer counter to synchronize data tokens among stages and parallel lines, coupled with a small object allocator to minimize the data allocation overhead; GRAMPS [70] designs a buffer manager with per-thread fix-sized memory pools to dynamically allocate new data and release used ones; FastFlow [4] designs a lock-free queue with a mechanism to transfer data ownership between senders and receivers, but

this method can incur imbalanced load and requires non-trivial back-pressure management; HPX [46] counts on a channel data structure and standard future objects to pass data around tasks, but the creation of share states becomes expensive when the pipeline is large; Cilk-P [50] employs per-stage queues coupled with two counter types to track static and dynamic dependencies of each node, but it targets on-the-fly pipeline parallelism, which is orthogonal to our focus; FDP [72] proposes a learning-based mechanism to adapt scheduling to an environment, but it requires expensive runtime profiling that may not work well for highly irregular applications like CAD. Pipeflow leverages C++ simple atomic operations and assigns every pipe an atomic variable denoting the dependency. Since there is no data synchronization involved, the scheduling algorithm of Pipeflow is lightweight and efficient. In terms of load balancing, most pipeline schedulers leverage work stealing, which has been reported with better performance than static policies [6, 23, 50, 52, 66, 70, 71]. However, for some special cases, such as fine-grained load-imbalanced pipelines, static policies perform comparably. For example, Pipelight [64, 65] implements a load-balancing technique based on two static scheduling algorithms, DSWP [67–69] and LBPP [47]; Pipelite [63, 65] and URTS [62, 65] introduce dynamic schedulers using ticket-based synchronization and directive-based model language for linear pipelines, respectively. Although, in some special cases, work stealing [53] may not give the best runtime performance, Pipeflow and the most frameworks still adopt this algorithm as it has the best performance in most applications. While co-designing task scheduling and buffer structures has certain advantages for data-centric pipeline (e.g., data locality), the cost of managing data can be significant yet unnecessary, especially for applications that only exploit task parallelism in pipeline.

6 CONCLUSION

In this paper, we have introduced Pipeflow, an efficient task-parallel pipeline programming framework to explore pipeline parallelism in applications. We have introduced a simple yet efficient scheduling algorithm based on our work-stealing runtime with dynamic load balancing. We have evaluated the performance of Pipeflow on a micro-benchmark and an industrial application. For example, in a VLSI static timing analysis workload that adopts pipeline parallelism to speed up the runtime performance, the Pipeflow's implementation is up to 110.33% faster than the oneTBB's implementation. Our future plans are to 1) apply Pipeflow to other applications than CAD applications to bring interdisciplinary ideas to the parallel computing community and 2) extend Pipeflow to task-parallel GPU computing platforms [11, 12, 43, 55–57] and distributed environment [28, 31, 42]

ACKNOWLEDGMENTS

We are grateful for the support of four National Science Foundation (NSF) grants, CCF-2349141, CCF-2349582 (CAREER), OAC-2349143, and TI-2349144.

REFERENCES

[1] Intel oneTBB. https://github.com/oneapi-src/oneTBB
[2] OpenMP. https://www.openmp.org/
[3] TAU 2018 Contest. https://sites.google.com/view/taucontest2018/home
[4] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *FastFlow: High-Level and Efficient Streaming on Multicore.* John Wiley and Sons, Ltd, Chapter 13, 261–280.
[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *International conference on Parallel architectures and compilation techniques (PACT).* 72–81.
[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* 207–216.
[7] George Bosilca, Robert J. Harrison, Thomas Herault, Mohammad Mahdi Javanmard, Poornima Nookala, and Edward F. Valeev. 2020. The Template Task Graph (TTG) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale. In *IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2).* 1–7.
[8] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE HPEC.* 1–7.
[9] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism Using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC).* 283–284.
[10] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results. In *ACM/IEEE Design Automation Conference (DAC).* 1388–1389.
[11] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2022. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *European Conference on Parallel Processing (Euro-Par).* 468–479.
[12] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).*
[13] David del Rio Astorga, Manuel F. Dolz, Javier Fernandez, and J. Daniel Garcia. 2017. A Generic Parallel Pattern Interface for Stream and Data Processing. In *Concurrency and Computation: Practice and Experience.*
[14] Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. 2012. BWS: balanced Work Stealing for Time-sharing Multicores. In *ACM European Conference on Computer Systems (EuroSys).* 365–378.
[15] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. In *Parallel Processing Letters.*
[16] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz G. Fernandes. 2019. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. In *The Journal of Parallel Programming.* 253–271.
[17] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Degign Automation Conference (DAC).* 1–6.
[18] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 1–9.
[19] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-Accelerated Static Timing Analysis. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD).*
[20] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *ACM/IEEE Design Automation Conference (DAC).* 715–720.
[21] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 1–9.
[22] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE TCAD* (2023).
[23] Ralf Hoffmann, Matthias Korch, and Thomas Rauber. 2004. Performance Evaluation of Task Pools Based on Hardware Synchronization. In *ACM Supercomputing.* 44–44.
[24] Renato B. Hoffmann, Junior Loff, Dalvan Griebler, and Luiz G. Fernandes. 2022. OpenMP as Runtime for Providing High-Level Stream Parallelism on Multi-Cores. In *The Journal of Supercomputing.* 7655–7676.
[25] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High Performance Extreme Computing Conference (HPEC).* 1–2.
[26] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).* 776–789.
[27] Tsung-Wei Huang and Leslie Hwang. 2022. Task-Parallel Programming with Constrained Parallelism. In *IEEE High Performance Extreme Computing Conference (HPEC).* 1–7.
[28] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2018. A General-Purpose Distributed Programming System Using Data-Parallel Streams. In *ACM International Conference on Multimedia (MM).* 1360–1363.

[29] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 974–983.

[30] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2019. Essential Building Blocks for Creating an Open-Source EDA Project. In *ACM/IEEE DAC*. Article 78, 4 pages.

[31] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. 2017. DtCraft: A distributed execution engine for compute-intensive applications. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 757–765.

[32] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE DAC*. Article 229.

[33] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 1303–1320.

[34] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. 2021. Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 40, 8 (2021), 1687–1700.

[35] Tsung-Wei Huang and Martin D. F. Wong. 2015. Accelerated Path-Based Timing Analysis with MapReduce. In *ACM International Symposium on Physical Design (ISPD)*. 103–110.

[36] Tsung-Wei Huang and Martin D. F. Wong. 2015. On fast timing closure: speeding up incremental path-based timing analysis with mapreduce. In *ACM/IEEE International Workshop on System Level Interconnect Prediction (SLIP)*. 1–6.

[37] Tsung-Wei Huang and Martin D. F. Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 895–902.

[38] Tsung-Wei Huang and Martin D. F. Wong. 2016. UI-Timer 1.0: An Ultrafast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 35, 11 (2016), 1862–1875.

[39] Tsung-Wei Huang, Martin D. F. Wong, Debjit Sinha, Kerim Kalafala, and Natesan Venkateswaran. 2016. A distributed timing analysis framework for large designs. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[40] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 596–599.

[41] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 758–765.

[42] Tsung-Wei Huang Huang, Chun-Xun Lin, and Martin D. F. Wong. 2019. DtCraft: A High-Performance Distributed Execution Engine at Scale. *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS (ICCAD)* 38, 6 (2019), 1070–1083.

[43] Shui Jiang, Tsung-Wei Huang, Bei Yu, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*.

[44] Shiu Jiang, Tsung-Wei Huangand, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE HPEC*. 1–7.

[45] Andrew B. Kahng. 2018. Reducing Time and Effort in IC Implementation: A Roadmap of Challenges and Solutions. In *ACM Design Automation Conference (DAC)*.

[46] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *International Conference on Partitioned Global Address Space Programming Models (PGAS)*. 1–11.

[47] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. 2013. Load-Balanced Pipeline Parallelism. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12.

[48] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE DAC*. Article 108, 6 pages.

[49] Tin-Yin Lai, Tsung-Wei Huang, and Martin D. F. Wong. 2017. LibAbs: An Efficient and Accurate Timing Macro-Modeling Algorithm for Large Hierarchical Designs. In *ACM/IEEE Design Automation Conference (DAC)*.

[50] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. In *ACM Transactions on Parallel Computing*. 1–42.

[51] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The Design of a Task Parallel Library. In *ACM OOPSLA*. 227–241.

[52] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. In *The Journal of Supercomputing*. 244–257.

[53] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM International Conference on Multimedia (MM)*. 2284–2287.

[54] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin D. F. Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI (GLVLSI)*. 183–188.

[55] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[56] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation Using Task Graph Parallelism. In *European Conference on Parallel Processing (Euro-Par)*.

[57] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism. *IEEE IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33, 11 (2022), 3041–3052.

[58] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2023. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *International Conference on Parallel Processing (ICPP)*. 1–12.

[59] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*.

[60] Junior Loff, Renato Barreto Hoffmann, Dalvan Griebler, and Luiz Gustavo Fernandes. 2021. High-Level Stream and Data Parallelism in C++ for Multi-Cores. In *Brazilian Symposium on Programming Languages (SBLP)*. 41–48.

[61] Aristeidis Mastoras and Thomas R. Gross. 2018. Understanding parallelization Tradeoffs for Linear Pipelines. In *International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. 1–10.

[62] Aristeidis Mastoras and Thomas R. Gross. 2018. Unifying Fixed Code Mapping, Communication, Synchronization and Scheduling Algorithms for Efficient and Scalable Loop Pipelining. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*. 2136–2149.

[63] Aristeidis Mastoras and Thomas R. Gross. 2019. Efficient and Scalable Execution of Fine-Grained Dynamic Linear Pipelines. In *ACM Transactions on Architecture and Code Optimization (TACO)*. 1–26.

[64] Aristeidis Mastoras and Thomas R. Gross. 2019. Load-balancing for load-imbalanced fine-grained linear pipelines. In *Parallel Computing*. 2136–2149.

[65] Aristeidis Mastoras and Albert-Jan N. Yzelman. 2023. Studying the expressiveness and performance of parallelization abstractions for linear pipelines. *Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 29–38.

[66] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. 2009. Load Balancing Using Wok-Stealing for Pipeline Parallelism in Emerging Applications. In *ACM International Conference on Supercomputing (ICS)*. 517–518.

[67] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 105–118.

[68] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. 2008. Parallel-Stage Decoupled Software Pipelining. In *IEEE/ACM international symposium on Code generation and optimization (CGO)*. 114–123.

[69] Ram Rangan, Neil Vachharajani, Guilherme Ottoni, and David I. August. 2008. Performance Scalability of Decoupled Software Pipelining. In *ACM Transactions on Architecture and Code Optimization (TACO)*. 1–25.

[70] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. 2011. Dynamic Fine-Grain Schedulig of Pipeline Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 22–32.

[71] Tao Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 189–203.

[72] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. 2010. Feedback-Directed Pipeline Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 147–156.

[73] Adriano Vogel, Dalvan Griebler, and Luiz Gustavo Fernandes. 2021. Providing high-level self-adaptive abstractions for stream parallelism on multicores. In *Journal of Software: Practice and Experience*. 1194–1217.

[74] Adriano Vogel, Gabriele Mencagli, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. Towards On-the-fly Self-Adaptation of Stream Parallel Patterns. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 89–93.

[75] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.