A Task-Parallel Pipeline Programming Model with Token Dependency

Cheng-Hsiang Chiu(✉), Wan-Luan Lee, Boyang Zhang, Yi-Hua Chung, Che Chang, and Tsung-Wei Huang(✉)

Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, USA
`{chenghsiang.chiu,wanluan.lee,bzhang523,yihua.chung, cchang289,tsung-wei.huang}@wisc.edu`

Abstract. Task-parallel pipeline framework explores pipeline parallelism in applications and is critical in many parallel and heterogeneous areas, such as VLSI static timing analysis and data similarity search. However, existing solutions only deal with certain types of applications in which data dependency exists between preceding data and succeeding data in a forward direction. Some applications, such as video encoding, exhibit data dependency in both forward and backward directions and cannot be processed with existing solutions. To address the limitation, we introduce a token dependency-aware pipeline framework. Our framework associates each data element with a token as its identifier, supports explicit definitions of forward and backward token dependency with an expressive programming model, resolves token dependency using simple data structures, and schedules tokens with lightweight atomic counters. We have evaluated the framework on applications that exhibit both forward and backward token dependency. For example, our framework is 8.6% faster than PARSEC's implementation in x.264 video encoding applications.

1 Introduction

Task-parallel pipeline framework (TPF) explores pipeline parallelism in applications and plays a critical role in parallel and heterogeneous computing workloads, such as static timing analysis [3,8,13–18,22–24,28,32], data similarity search workload (ferret benchmark [4,5]), quantum circuit simulation [26], and others [7,25,27,29,35,36]. TPF models a pipeline application as a *task graph* that describes a function call as a task and a functional dependency as an edge. Through task graph scheduling, pipeline parallelism arises when multiple tasks are scheduled and executed concurrently once the dependency constraints are met. As a result, recently Chiu et al. introduced the state-of-the-art task-parallel pipeline framework, Pipeflow [13].

Although Pipeflow has demonstrated good runtime performance [13], we find out Pipeflow only deals with specific applications in which data dependencies exist between preceding data and succeeding data in a forward direction. However, some applications exhibit data dependency from succeeding data back to

https://doi.org/10.1007/978-3-031-97196-9_1

preceding data. For instance, in video encoding applications, frames would reference encoded frames to reduce stream size for online transmission [34]. In real-world applications, three frame types are employed, intra (I) frame, predicted (P) frame, and bi-directional (B) frame. I frames are encoded independently without reference to other frames. P frames require references from a preceding I frame. B frames require references from both a preceding and a succeeding (future) I or P frames. Figure 1 illustrates such frame dependency in a video encoding application. The presence of B frames introduces bi-directional dependency, where the encoding of a frame relies on information from both past and future frames. This characteristic poses a significant challenge for existing pipeline frameworks, including Pipeflow, which primarily focus on uni-directional dependency. As a result, Pipeflow cannot effectively schedule the encoding of B frames, limiting its applicability in real-world video encoding scenarios.

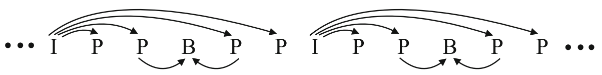


Fig. 1. A sample dependency diagram in a video encoding application of an x.264 standard. Edges denote the dependencies between two frames. I denotes frames, P denotes predicted, B denotes bi-directional frames.

To handle the data dependency in both forward and backward directions, the most common way is to reorder the execution order of data using low-level synchronization primitive, *condition variable* [1], and then feed the reordered data to the pipeline framework as PRASEC does [4,5]. However, we notice three limitations of this approach: 1) Manipulating *condition variable* requires a deep understanding of this low-level synchronization primitive from users and is error-prone when dependency is intricate. 2) The approach is not an end-to-end implementation as users need to additionally reorder the data outside the original pipeline application. 3) The implementation could encounter deadlock when the data dependency is complex and insufficient threads are spawned.

To overcome the limitations, we have associated each data element with a token as its identifier and introduced a new task-parallel pipeline framework with token dependency enabled on top of Pipeflow [13]. We summarize our technical contributions as follows:

– **New Programming Model.** We have proposed a new programming model for applications to explicitly define generalized bi-directional token dependency. With our programming model, applications can easily specify the token dependency with a single and intuitive API and do not need to touch low-level synchronization primitives.
– **New Scheduling Algorithm.** We have proposed a new scheduling algorithm to support our new programming model. Our scheduling algorithm leverages simple data structures to efficiently determine the execution order of tokens and to avoid potential deadlock when dealing with intricate token dependency and insufficient threads are spawned.

- **End-to-end Implementation.** We have integrated the step of reordering tokens into the pipeline to achieve an end-to-end implementation. With our seamless integration, users can eliminate the need for external token reordering mechanisms, simplifying the overall system design while providing an end-to-end solution for scheduling tokens with bi-directional dependency within the pipeline itself.

We have evaluated the framework on applications that exhibit both forward and backward token dependency. For example, our framework is 8.6% faster than PARSEC's implementation in x.264 video encoding applications.

2 Background

Due to space constraints, we will focus on token dependency and the state-of-the-art Pipeflow programming framework [13], rather than providing a comprehensive discussion of pipeline frameworks. For a broader discussion of other pipeline frameworks, readers are referred to [13].

2.1 Token Dependency

Token dependency constrains the order in which tokens should execute in the pipeline. A dependency exists between token t_1 and t_2 in which t_1 must complete before t_2 can begin. We categorize token dependency into two types: forward token dependency (FTD), which refers to dependency connecting from preceding to succeeding token, and backward token dependency (BTD), which refers to dependency in the opposite direction. Figure 2(a) shows a diagram in which all dependencies are FTDs, which are implicitly assumed in existing task-parallel pipeline framework. Since all dependencies are FTDs, there is no need to reorder the tokens to get the correct execution order. Figure 2(b) shows a diagram combining both FTDs and BTDs. As BTDs exist, we need to reorder the tokens to get the correct execution order. For example, token 16 pointing to 7 and 12 are BTDs, we need to reorder 16 before 7 and 12.

To get the correct execution order of tokens when BTDs exist, existing frameworks adopt the *condition variable* primitive, such as PARSEC's pthread implementation, to first determine the execution order and then flow the reordered tokens through the pipeline. Figure 3 illustrates PARSEC's implementation using C++. Every token has its own condition variable `cv` and a mutex `mutex` and is handled by one thread. A token will wait on the `cv` of its dependent token until that dependent token finishes. For example, token 7 has a dependent token 16, meaning token 7 must wait until 16 finishes. 7 will acquire 16's `mutex` and wait on 16's `cv` until 16 finishes. Once token 16 finishes, token 7 can start the execution and then notify all the other waiting tokens that token 7 has finished.

The implementation of using *condition variable* is able to reorder the tokens whenever BTDs exist. However, we notice three challenges: 1) Manipulating *condition variable* requires a deep understanding of this low-level synchronization

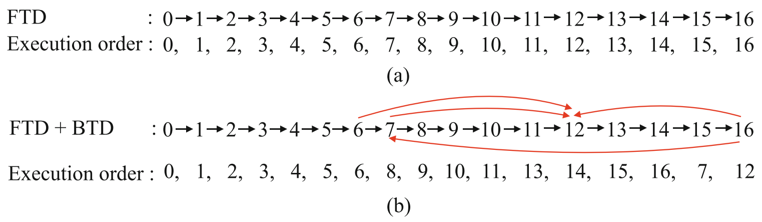
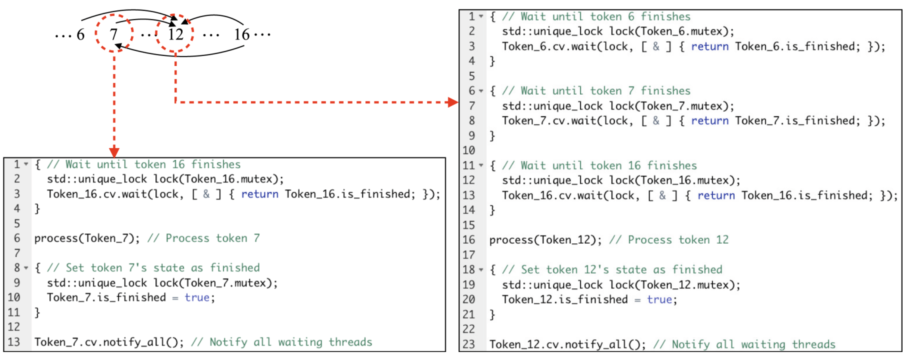

FTD : 0→1→2→3→4→5→6→7→8→9→10→11→12→13→14→15→16
Execution order : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
(a)
FTD + BTD : 0→1→2→3→4→5→6→7→8→9→10→11→12→13→14→15→16
Execution order : 0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15, 16, 7, 12
(b)

Fig. 2. (a) The diagram of all FTDs and the corresponding execution order of tokens. (b) The diagram of both FTDs and BTDs and its corresponding execution order of tokens. Red edges pointing from token 6 and 7 to 12 are FTDs, and those from 16 to 7 and 12 are BTDs. Black edges are implicit dependencies and red ones are explicit dependencies. Execution order denotes the order in which the tokens should be executed. (Color figure online)



```
{ // Wait until token 16 finishes
  std::unique_lock lock(Token_16.mutex);
  Token_16.cv.wait(lock, [ & ] { return Token_16.is_finished; });
}

process(Token_7); // Process token 7

{ // Set token 7's state as finished
  std::unique_lock lock(Token_7.mutex);
  Token_7.is_finished = true;
}

Token_7.cv.notify_all(); // Notify all waiting threads
```

```
{ // Wait until token 6 finishes
  std::unique_lock lock(Token_6.mutex);
  Token_6.cv.wait(lock, [ & ] { return Token_6.is_finished; });
}

{ // Wait until token 7 finishes
  std::unique_lock lock(Token_7.mutex);
  Token_7.cv.wait(lock, [ & ] { return Token_7.is_finished; });
}

{ // Wait until token 16 finishes
  std::unique_lock lock(Token_16.mutex);
  Token_16.cv.wait(lock, [ & ] { return Token_16.is_finished; });
}

process(Token_12); // Process token 12

{ // Set token 12's state as finished
  std::unique_lock lock(Token_12.mutex);
  Token_12.is_finished = true;
}

Token_12.cv.notify_all(); // Notify all waiting threads
```

Fig. 3. PARSEC's implementation of Fig. 2(b) to reorder tokens using the *condition variable* primitive.

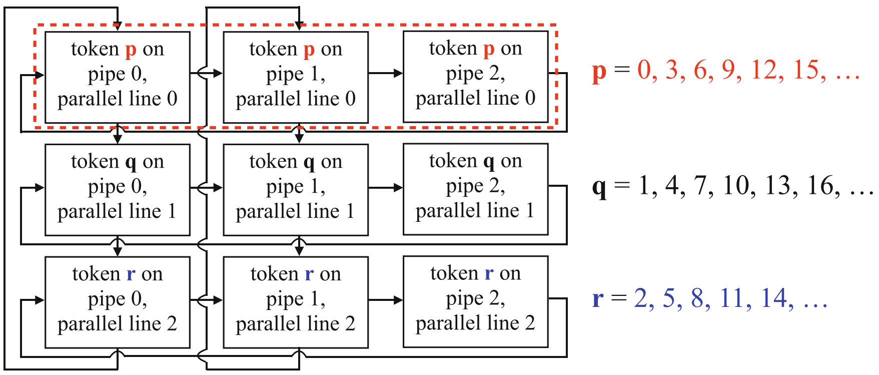
primitive from users. This can be particularly challenging for users, especially when dealing with complex token dependency, such as those involving tokens with multiple forward and backward dependencies (e.g., token 12 in Fig. 3). 2) This solution is not an end-to-end implementation as users require manual token reordering outside the core pipeline execution logic. This introduces additional complexity and increases the risk of errors in the overall system. 3) The reliance on low-level synchronization primitives can increase the risk of deadlocks, especially in resource-constrained environments (insufficient threads) with complex dependency graphs. For instance, consider a scenario where token 6 has backward dependencies on tokens 1 through 5, and only 5 threads are available. These 5 threads are all waiting for token 6 to complete, a deadlock occurs, as no thread is available to process token 6. As a result, we need a solution that is able to avoid deadlock when application's token dependency is complex and insufficient threads are spawned.

2.2 Pipeflow: Task-Parallel Pipeline Framework

Pipeflow [13] represents a state-of-the-art task-parallel pipeline framework, offering significant advancements over prior work such as oneTBB [2]. By decoupling task scheduling from data abstraction, Pipeflow introduces an efficient scheduling algorithm that optimizes pipeline execution. Furthermore, Pipeflow provides an expressive programming model, simplifying the process of defining complex pipeline applications for developers.

Pipeflow models pipeline applications as circular task graphs, as depicted in Fig. 4. This representation facilitates the scheduling of tasks across multiple parallel execution units. For instance, in a scenario with three parallel execution lines, Pipeflow can effectively schedule token 0 on parallel line 0, token 1 on parallel line 1, and token 2 on parallel line 2 and so on, adhering to the execution order obtained by the dependency illustrated in Fig. 2(a).

Figure 4 illustrates the concept of parallel execution within the Pipeflow framework. In this example, the first and second pipeline stages (or pipes) are serial pipes, meaning tokens within these stages must execute sequentially. For instance, token q in pipe 0 and parallel line 1 cannot commence execution until token p in pipe 0 and parallel line 0 has completed. In contrast, the third pipe is a parallel pipe. This allows for concurrent processing of tokens within this stage. For example, token p in pipe 2 and parallel line 0 can execute concurrently with token q in pipe 2 and parallel line 1.



token p on pipe 0, parallel line 0
token p on pipe 1, parallel line 0
token p on pipe 2, parallel line 0
p = 0, 3, 6, 9, 12, 15, …
token q on pipe 0, parallel line 1
token q on pipe 1, parallel line 1
token q on pipe 2, parallel line 1
q = 1, 4, 7, 10, 13, 16, …
token r on pipe 0, parallel line 2
token r on pipe 1, parallel line 2
token r on pipe 2, parallel line 2
r = 2, 5, 8, 11, 14, …

Fig. 4. Pipeflow's circular task graph of an application in which every token is processed by a chain of 3 pipes (in the red dashed rectangle, referred to a `parallel line`) and up to 3 tokens can be processed concurrently. Edges denote dependencies.

3 Proposed Framework

We introduce a new task-parallel pipeline framework with token dependency enabled shown in Fig. 5. At a high level, we first determine the execution order of tokens and then flow the ordered tokens through the circular task graph. We provide an expressive programming model for users to explicitly express generalized bi-directional token dependency and provide a pipeline scheduling framework to support our programming model.

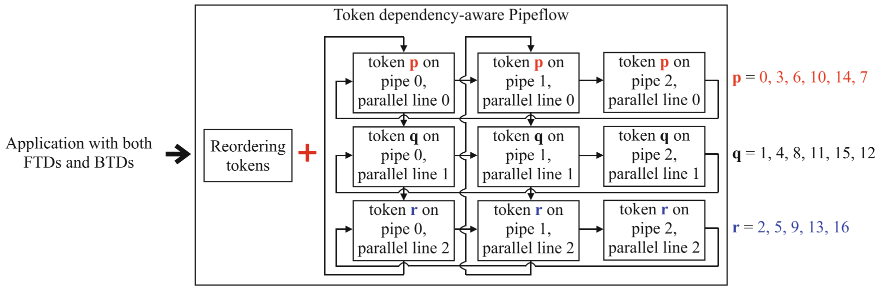

Token dependency-aware Pipeflow
Application with both FTDs and BTDs
Reordering tokens
token p on pipe 0, parallel line 0
token p on pipe 1, parallel line 0
token p on pipe 2, parallel line 0
token q on pipe 0, parallel line 1
token q on pipe 1, parallel line 1
token q on pipe 2, parallel line 1
token r on pipe 0, parallel line 2
token r on pipe 1, parallel line 2
token r on pipe 2, parallel line 2
p = 0, 3, 6, 10, 14, 7
q = 1, 4, 8, 11, 15, 12
r = 2, 5, 9, 13, 16

Fig. 5. Overview of token dependency-aware Pipeflow running an application with both FTDs and BTDs, as illustrated in Fig. 2(b). After determining the execution order of tokens, our framework schedules token $0, 3, 6, 10, 14$, and 7 on parallel line 0 and so on.

3.1 Programming Model

We extend the Pipeflow framework [13] by introducing a novel programming model that explicitly supports bi-directional token dependency. Our new programming model leverages a two-tiered approach. Firstly, it retains Pipeflow's core structure for defining the pipeline structure. Secondly, we introduce a mechanism for explicitly defining token dependency within this framework. Listing 1.1 exemplifies an application of a serial-serial-parallel pipeline structure with token dependency in Fig. 2(b). To define the pipeline structure, we follow Pipeflow's programming model. We use the API `Pipeline` to instantiate an object `pl` and define the pipeline structure (line 5). In the API, we specify the number of parallel lines (line 4) and the abstract function of every pipe. For every pipe, we define the pipe type and a pipe callable using `Pipe`. We use `PipeType::SERIAL` to specify the type for the first pipe (line 5) and the second pipe (line 30), and `PipeType::PARALLEL` for the third pipe (line 36). The pipe callable takes an argument `pf` (line 6, 31, and 37) which is used to query the status of a token that is executing the pipe callable. In this example, the first pipe stores a `float` in the buffer `buffer` (line 21 and 25), the second pipe stores a `string` in `buffer` (line 33), and the third pipe prints the value (line 39).

The second part of our new programming model is to specify the token dependency. To achieve an end-to-end implementation, we integrate this step into the pipeline by explicitly specifying the token dependency at the first pipe before the tokens flows to the pipes. To specify token dependency, we first use the number returned by `Pipeflow::num_deferrals` to start defining the token dependency (line 11). Initially, all tokens have zero `num_deferrals`. Then we specify token 12's three dependencies (line 13:15) and token 7's dependency (line 18) using `Pipeflow::defer`, respectively. For tokens that do not have dependencies (line 21) or tokens whose execution orders are determined can resume execution (line 25), we define the corresponding function. Finally, we call `run` to submit the object `pl` to a runtime and execute it (line 43).

```
1  std::variant<float, std::string> data_type;
2  std::array<data_type, number_lines> buffer;
3
4  Pipeline pl(3,
5    Pipe{PipeType::SERIAL, // Define the first pipe
6      [&](Pipeflow& pf) {
7        if ( pf.token() == 100) { // Stop when 100 tokens are done
8          pf.stop();
9        }
10       else {
11         if (pf.num_deferrals() == 0) {
12           if (pf.token() == 12) { // Specify token 12's dependencies
13             pf.defer(6);
14             pf.defer(7);
15             pf.defer(16);
16           }
17           else if (pf.token() == 7) { // Specify token 7's dependency
18             pf.defer(16);
19           }
20           else {
21             buffer[pf.line()] = 0.0f; // Save a float in buffer
22           }
23         }
24         else {
25           buffer[pf.line()] = 0.0f; // Save a float in buffer
26         }
27       }
28     }
29   },
30   Pipe{PipeType::SERIAL, // Define the second pipe
31     [&](Pipeflow& pf) {
32       // Save a string in buffer
33       buffer[pf.line()] = make_string(std::get<0>(buffer[pf.line()]));
34     }
35   },
36   Pipe{PipeType::PARALLEL, // Define the third pipe
37     [&](Pipeflow& pf) {
38       // Print the string stored in buffer
39       std::cout << std::get<1>(buffer[pf.line()]);
40     }
41   }
42 );
43 pl.run();  // Execute the pipeline
```

Listing 1.1. The implementation of a pipeline application consisting of 3 parallel lines and 3 pipes with a serial-serial-parallel type. The token dependency for the application is shown in Fig. 2(b). Assume the first pipe stores a float in `buffer`, the second pipe stores a string in `buffer`, and the third pipe prints the value.

In contrast to existing approaches, such as PARSEC [4,5], our framework eliminates the need for low-level synchronization primitives like *condition variables*, significantly simplifying dependency management. This not only reduces development complexity but also improves developer productivity by minimizing the risk of errors. Our framework introduces a concise and intuitive API for defining dependency, `pf.defer`. For example, specifying the dependencies for token 12 requires only three lines of code in Listing 1.1, compared to the 23 lines of code required in the PARSEC implementation (see Fig. 3) for handling same dependencies. This significant reduction in code complexity simplifies debugging and maintenance, particularly in scenarios involving complex dependency graphs.

3.2 Scheduling Algorithm

To support our programming model, we design a new scheduling algorithm which includes two components: 1) Determining the correct execution order of tokens. 2) Scheduling the reordered tokens in the pipeline.

1) **Determining the Correct Execution Order of Tokens**. The first part of our scheduling algorithm is to determine the correct execution order of tokens. The idea is to defer the execution of a token with unresolved token dependency and save the token until its dependency is resolved. Once that token becomes ready, we run it as soon as possible in order to resolve possible dependency for other tokens. To realize the idea, we use three data structures,

– `deferred_tokens` (`DT`): An associative container that stores deferred tokens and their respective dependencies. For example, in Fig. 2(b), token 7 has one dependent token 16, meaning 16 must reorder before 7. We consider token 7 as a deferred token and represent the relationship as the entry {key:7, value:16} in `DT`.
– `token_dependencies` (`TD`): Another associative container that stores the reverse mapping of dependencies. For example, for the dependency between token 7 and 16 in Fig. 2(b) `TD` stores {key:16, value:7}, allowing for efficient identification of tokens that depend on a given token. This enables rapid updates to `DT` when a dependency is resolved. Specifically, once token 16's order is determined later, we can quickly use the value obtained at `TD[16]`, which is token 7, to locate and remove the entry at `DT[7]`.
– `ready_tokens` (`RT`): A queue that stores tokens whose dependencies have been resolved and are ready for execution.

Figure 6 visualizes how we use the three data structures to determine the correct execution order of tokens in Fig. 2(b). In (a), token 0 to 6 do not have BTDs and we put them in the `EST` list in order. In (b), token 7 has a dependent token 16 because of `7.defer(16)`. We first insert {key:7, value: 16} in `DT`, meaning 7 needs to be reordered after 16. Then we insert {key:16, value:7} in `TD` in order to locate 7 in `DT` quickly. `EST` does not have 7 as its' execution order is not yet decided. In (c), token 8 to 11 do not have BTDs and appear in `EST`. Token 12 has three dependencies, token 6, 7, and 16. As token 6's order has been determined in (a), we only insert {key:12, value:$\{7, 16\}$} in `DT` and then update `TD` to reflect the new dependency. In (d), token 13 to 15 do not have BTDs and appear in

EST. In (e), token 16 does not have BTDs and appears in EST. As TD[16] exists, we use the value of that entry, token 7 and 12, to locate the corresponding entry in DT. Then we resolve 16's related dependency by deleting 16 from DT[7] and DT[12]. As a result, DT[7] is empty, meaning 7's order can be determined and we insert it in RT. In (f), RT is not empty and we append token 7 in EST. Next, TD[7] has token 12, and we directly use 12 to locate DT[12] and delete 7 from that entry. As a result, DT[12] is empty, meaning 12's order can be determined and be inserted in RT. In (g), we find 12 in RT and then append it in EST. In the end, we obtain the correct execution order of tokens as shown in Fig. 2(b) using only three simple data structures, DT, TD, and RT.

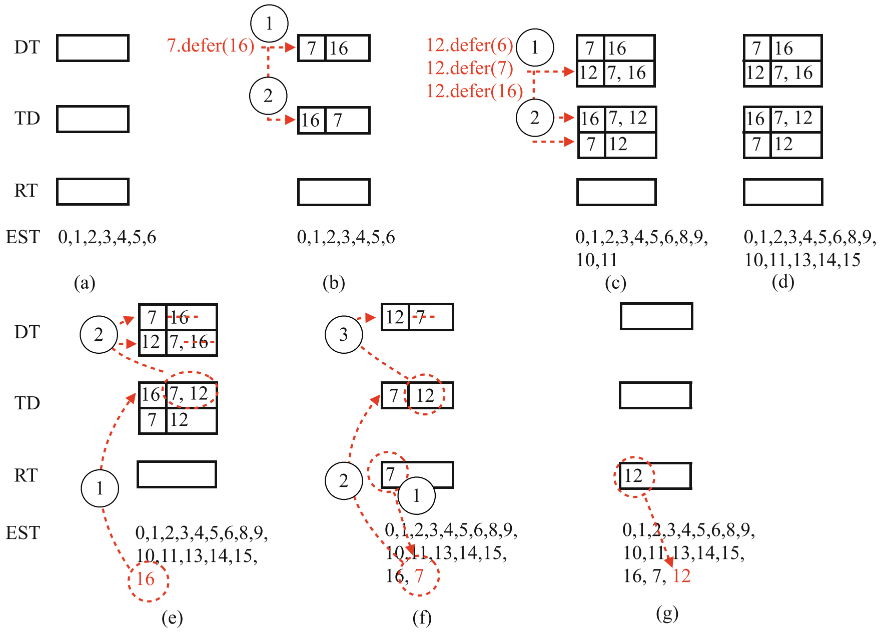


Fig. 6. Visualization of how DT, TD, and RT determine the correct execution order of tokens with the token dependency in Fig. 2(b). EST denotes the execution order of tokens and is used for illustration. We simplify `pf.defer(16)` in line 15 in Listing 1.1 to `7.defer(16)` for explanation purposes. The encircled numbers denote the operation sequence in each sub-figure.

2) **Scheduling Reordered Tokens in the Pipeline**. After determining the correct execution order of tokens, we schedule the reordered tokens over the circular task graph as Pipeflow does [13]. We place one token per parallel line and schedule all tokens in a circular way across all parallel lines. As a result, we schedule the reordered tokens as shown in Fig. 5. For example, we schedule token $0, 3, 6, 10, 14, 7$ over parallel line 0 and so on.

3.3 Pseudocode

To implement the proposed algorithm discussed in Sect. 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) determines the execution order of tokens at the first pipe, 2) deals with one scheduling token per parallel line, and 3) decides which adjacent task to run on its next parallel line and pipe. Algorithm 1 implements such a task. When a task is to be scheduled, we must know which pipe and which parallel line for the token to run at. Each task owns an object `pf` of a specific line `l` (line 1). Once a token is done at a pipe, there are two cases for its corresponding task to proceed: 1) for a parallel pipe, the task moves to the next pipe at the same parallel line; 2) for a serial pipe, the task additionally checks if it can move to the next parallel line. For example, in Fig. 5 when a thread finishes token p at pipe 1 and parallel line 0, we check whether the thread goes to the pipe 2 at the same parallel line 0 or pipe 1 at the adjacent parallel line 1. To carry out such a task 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 (serial) or 1 (parallel) based on the corresponding pipe types. At the first pipe (line 3), a task either takes a ready token (i.e., a token whose order has been determined) (line 4:6) or a new token (line 7:9), and then invokes the pipe callable on that token (line 10). Then, we increment the number of processed tokens `num_tokens` by one if the task processes a new token (line 11:12).

When a task finds the token has dependency (line 13:19), we call `check_dependents` (line 14, defined in Algorithm 2) to keep valid dependents and remove invalid ones. Invalid dependents refer to tokens whose order has been determined. If the token still has dependents after the first check (line 15:17), we construct it as a deferred token (line 16, defined in Algorithm 3) and reiterate the task with another token (line 17). If the token has no dependent, we reiterate the task with the same token (line 18:19). After a token finishes at the first pipe, we call `resolve_token_dependencies` (line 21, defined in Algorithm 4) to resolve its associated token dependency up to `longest_deferral` (lines 20:21). `longest_deferral` keeps track of the biggest deferred token ID and is used to avoid redundant invocations of `resolve_token_dependencies`. For example, there is no need to invoke `resolve_token_dependencies` for applications that do not exhibit BTDs, such as Fig. 2(a). At the first pipe (line 3:21), we perform the above operations. For other pipes, we simply invoke the pipe callables (line 22:23). After the pipe callable returns, we call `schedule_tasks(pf)` (line 24, defined in Algorithm 5) to determine the next possible tasks to run.

Algorithm 2 shows the implementation of `check_dependents` in line 14 in Algorithm 1 to check if a token has valid dependents. We increment the number of deferrals of that token to track how many times this token has been deferred (line 1). We iterate the token's dependents to check the validity for two cases (line 2:10). Firstly, the dependent whose ID is bigger than the number of processed

Algorithm 1: build_tasks(l)

```
    global: $tasks$: an array of tasks
    global: $ready\_tokens$: a queue of ready tokens
    global: $join\_counters$: a two-dimensional array of join counters
    global: $longest\_deferral$: an integer of longest deferral
    global: $num\_tokens$: the number of processed tokens
    Input: $l$: a parallel line id
 1  $pf \leftarrow tasks[l]$;
 2  AtomicStore($join\_counters[pf.line][pf.pipe], join\_counter\_of\_pf.type$);
 3  if $pf.pipe == 0$ then
 4      if $ready\_tokens.empty() == false$ then
 5          $pf.token \leftarrow ready\_tokens.front()$;
 6          $ready\_tokens.pop()$;
 7      else
 8          $pf.token \leftarrow num\_tokens$;
 9          $pf.num\_deferrals \leftarrow 0$;
10      invoke_pipe_callable($pf$);
11      if $pf.token == num\_tokens$ then
12          Increment($num\_tokens$);
13      if $pf.dependents.empty() == false$ then
14          check_dependents($pf$);
15          if $pf.dependents.empty() == false$ then
16              construct_deferred_tokens($pf$);
17              goto Line 2;
18          else
19              goto Line 10;
20      if $pf.token \leq longest\_deferral$ then
21          resolve_token_dependencies($pf$);
22  else
23      invoke_pipe_callable($pf$);
24  schedule_tasks($pf$);
```

tokens (`num_tokens`) is a future token and thus is valid. We insert `pf.token` in `token_dependencies[dep]` (line 4) and update `longest_deferral` (line 5). Secondly, the dependent that is a deferred token is valid, and we insert the corresponding entry in `token_dependencies` (line 7:8). The remaining dependents are invalid and are removed from the token's dependents (line 9:10).

Algorithm 3 shows the construction of a deferred token in line 16 in Algorithm 1. For a deferred token, we insert the key-value pair in `deferred_tokens`, where the key is the token ID, and the value includes the token's ID, `num_deferrals`, and its `dependents`.

Algorithm 4 implements `resolve_token_dependencies` in line 21 in Algorithm 1. When a token whose order has been determined at the first pipe and has an entry in `token_dependencies`, we need to resolve the associated token dependency (line 1:6). We iterate over every element (`deferred_token`) of the entry in `token_dependencies[pf.token]` and remove the token from `deferred_token`'s

Algorithm 2: check_dependents(pf)

```
    global: token_dependencies: a hashmap of a token and the deferred tokens
    global: longest_deferral: an integer of longest deferral
    global: num_tokens: the number of processed tokens
    Input: pf: a pipeflow object
 1  Increment($pf.num\_deferrals$);
 2  for $dep \in pf.dependents$ do
 3      if $num\_tokens \leq dep$ then
 4          $token\_dependencies[dep].push(pf.token)$;
 5          $longest\_deferral \leftarrow max(longest\_deferral, dep)$;
 6      else
 7          if $dep \in deferred\_tokens$ then
 8              $token\_dependencies[dep].push(pf.token)$;
 9          else
10              $pf.dependents.erase(dep)$;
```

Algorithm 3: construct_deferred_tokens(pf)

```
    global: deferred_tokens: a hashmap of a token and its deferred object
    Input: pf: a pipeflow object
 1  $deferred\_tokens[pf.token].token \leftarrow pf.token$;
 2  $deferred\_tokens[pf.token].num\_deferrals \leftarrow pf.num\_deferrals$;
 3  $deferred\_tokens[pf.token].dependents \leftarrow pf.dependents$;
```

dependents (line 3). If `deferred_token` does not have any dependent left, it is no longer a deferred token and becomes ready. We insert `deferred_token` in `ready_tokens` and remove it from `deferred_tokens`.

Algorithm 4: resolve_token_dependencies(pf)

```
    global: token_depencencies: a hashmap of a token and its related deferred
            tokens
    global: deferred_tokens: a hashmap of a token and its deferred object
    global: ready_tokens: a queue of ready tokens
    Input: pf: a pipeflow object
 1  if $pf.token \in token\_dependencies$ then
 2      for $deferred\_token \in token\_dependencies[pf.token]$ do
 3          $deferred\_token.dependents.erase(pf.token)$;
 4          if $deferred\_token.dependents.empty() == true$ then
 5              $ready\_tokens.push(deferred\_token)$;
 6              $deferred\_tokens.erase(deferred\_token)$;
```

Algorithm 5 shows how we schedule tasks when a token finishes at a pipe in line 24 in Algorithm 1. We update variables (line 1:4) and define an array to track next tasks (line 5). We update the join counters based on the pipe type and determine the next possible tasks to run (line 6:9). When the join counter of a pipe reaches zero, we bookmark this pipe as a task to run next (line 7 and line 9). If two next tasks exist (line 10), the current task informs the scheduler to call a worker thread to run the task at the next parallel line (line 11) and reiterates itself on the next pipe (line 12). The idea here is to facilitate data locality as applications tend to deal with the next pipe as soon as possible. If only one task exists, the current task directly runs the next task with the updated `pf` object (line 13:16).

Algorithm 5: schedule_tasks(pf)

```
global: $num\_pipes$: the number of pipes
global: $num\_lines$: the number of parallel lines
global: $join\_counters$: a two-dimensional array of join counters
Input: $pf$: a pipeflow object
1  $curr\_pipe \leftarrow pf.pipe$;
2  $next\_pipe \leftarrow (pf.pipe + 1)\% num\_pipes$;
3  $next\_line \leftarrow (pf.line + 1)\% num\_lines$;
4  $pf.pipe \leftarrow next\_pipe$;
5  $next\_tasks = \{\}$;
6  if $curr\_pipe$ is $SERIAL$ and
    $AtomicDecrement(join\_counters[next\_line][curr\_pipe]) == 0$ then
7      $next\_tasks$.insert(1);
8  if $AtomicDecrement(join\_counters[pf.line][next\_pipe]) == 0$ then
9      $next\_tasks$.insert(0);
10 if $next\_tasks.size() == 2$ then
11     call_scheduler($tasks[next\_line]$);
12     goto Line 2 in Algorithm 1;
13 if $next\_tasks.size() == 1$ then
14     if $next\_tasks[0] == 1$ then
15         $pf \leftarrow tasks[next\_line]$;
16     goto Line 2 in Algorithm 1;
```

4 Experiments

We implemented our framework using C++20 and evaluated the runtime performance on a x.264 application. We compiled all programs using g++11.4 with -std=c++20 and -O3 enabled. We ran all the experiments on a Ubuntu Linux 22.04 machine with 20 Intel i5-13500 CPU cores at 4.8 GHz and 128 GB RAM. All data is an average of ten runs.

4.1 Real-World X.264 Application

We evaluated our framework with x.264 applications and demonstrated the advantages of our approach to effectively handle generalized token dependency.

The goal of x.264 is to generate H.264-compatible video streams [4,5]. In H.264 standard, there are three types of video frames, I, P, and B frames. To encode frames, different frame types would reference either preceding or succeeding frames. I frames do not reference other frames, P frames reference one preceding I frame, and B frames reference one preceding and one succeeding I or P frame. Figure 1 illustrates a sample dependency diagram between these three frames. The backward dependency of B frames cannot be easily handled using existing pipeline frameworks without using `condition variable` to first reorder the tokens as PARSEC [4,5] does. We modified the x.264 benchmark from [4] by duplicating the frames to a bigger benchmark to evaluate the performance under different frame sizes and thread sizes. In addition, we added more `B` frames in the modified benchmark to further mimic the H.264 standard.

We considered PARSEC's pthread implementation [4] as the baseline because PARSEC is the popular pipeline benchmark for many applications, such as ferret and x.264. To apply pipeline parallelism to x.264 applications, PARSEC assigns a thread to each frame. Every frame has a *condition variable* associated with its dependency. If a frame has unresolved dependency, its condition variable will wait until its dependency is resolved. When the frame becomes ready, its condition variable will broadcast the frame's readiness to any other frames that are waiting for the frame to finish. We implemented PARSEC's condition variable solution using C++. Figure 3 illustrates the implementation in which we use token 12 to simulate a B frame. This fine-grained control requires a high familiarity with low-level synchronization primitives from developers and is error-prone. With our framework, applications can directly specify the frame dependencies at the first pipe. Besides, PARSEC's solution could lead to deadlock when insufficient threads are spawned as discussed in Sect. 2.1. Our framework can avoid deadlock regardless of the thread counts.

Figure 7 compares the frame reordering time between our framework and PARSEC with up to 2 million frames and using up to 20 threads. We find out that the gap between our solution and PARSEC increases as the frame sizes grow. For example, when using 8 threads, the gap increases from 0.7 s at 1 million frames (2^{20}) to 1.4 s at 2 million frames (2^{21}). For the largest 2 million frame sizes, our framework is consistently faster than the baseline. For example, ours is 8.2%, 8.6%, 5.4%, 6.8%, 7.8%, and 5% faster than PARSEC when using 8, 10, 12, 14, 16, and 20 threads, respectively. We also notice that all of the plots show one trend that our framework outperforms PARSEC regardless of the frame sizes and thread sizes. For example, when running 1 million frames we are 0.7 s faster with both 8 and 16 threads; when using 16 threads, we are 8%, 8%, and 7.8% faster running 0.5, 1, and 2 millions frames, respectively. We attribute the observations to the reasons: 1) PARSEC uses fine-grained low-level *condition variable* to address the bi-directional frame dependencies. When a frame has unresolved dependency, the thread that processes the frame has to wait until the dependency is resolved. When a frame finishes, the thread needs to broadcast the completeness of the frame to other waiting frames. This mechanism that puts waiting threads to sleep and wakes up threads to resume

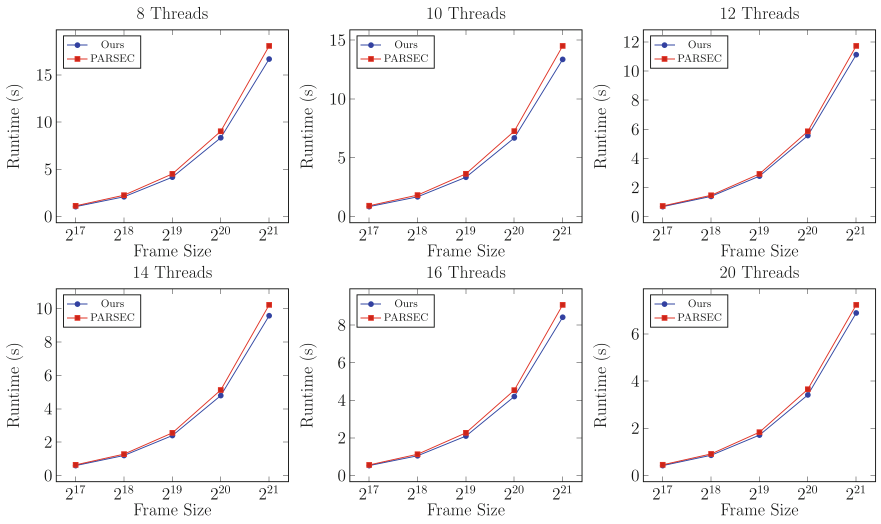


Fig. 7. Runtime comparison between our framework and PARSEC at different frame and thread sizes.

the operations causes overheads. 2) Our solution does not wait for the frame but stores a deferred frame in `deferred_token` and continues to schedule other frames without waiting. This design could avoid the overheads that *condition variable* brings. As a result, our framework demonstrates the runtime advantages over the baseline in all cases regardless of the frame sizes and thread sizes.

5 Conclusion

In this paper, we have introduced a new task-parallel pipeline programming framework on top of the state-of-the-art Pipeflow to explore pipeline parallelism in applications with token dependency. We have introduced an expressive programming model for applications to explicitly specify generalized token dependency. We have introduced a simple yet efficient scheduling algorithm to reorder tokens and schedule reordered tokens in the pipeline. We have evaluated the performance of our framework on an x.264 video encoding application. For example, our framework is 8.6% faster than PARSEC's implementation. We have integrated Pipeflow into the open-source task-parallel programming system, Taskflow [22] to benefit the HPC community. Our future plans are to 1) apply the framework to more different types of applications and bring interdisciplinary ideas to the parallel computing community, 2) extend token dependency-aware Pipeflow to task-parallel GPU computing platforms [6,10,11,25,27,29–31,35] and distributed environment [19–21], and 3) leverage machine learning techniques to further improve the scheduling performance [9,12,33].

Acknowledgement. This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

1. C++ condition variable. https://en.cppreference.com/w/cpp/thread/condition_variable
2. Intel oneTBB. https://github.com/oneapi-src/oneTBB
3. OpenTimer. https://github.com/OpenTimer/OpenTimer
4. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 72–81 (2008)
5. Bienia, C., Li, K.: Scaling of the parsec benchmark inputs. In: International Conference on Parallel Architectures and Compilation Techniques (PACT) (2010)
6. Chang, C., et al.: PathGen: an efficient parallel critical path generation algorithm (2025)
7. Chang, C.C., Zhang, B., Huang, T.W.: GSAP: a GPU-accelerated stochastic graph partitioner. In: ACM ICPP, pp. 565–575 (2024)
8. Chiu, C.H., Huang, T.W.: Efficient timing propagation with simultaneous structural and pipeline parallelisms: late breaking results. In: ACM/IEEE DAC, p. 1388–1389 (2022)
9. Chiu, C.H., Huang, T.W.: An experimental study of dynamic task graph parallelism for large-scale circuit analysis workloads (2024)
10. Chiu, C.H., Lin, D.L., Huang, T.W.: An experimental study of SYCL task graph parallelism for large-scale machine learning workloads. In: Euro-Par Workshop (2022)
11. Chiu, C.H., Lin, D.L., Huang, T.W.: Programming dynamic task parallelism for heterogeneous EDA algorithms. In: IEEE/ACM ICCAD (2023)
12. Chiu, C.H., Morchdi, C., Zhou, Y., Zhang, B., Chang, C., Huang, T.W.: Reinforcement learning-generated topological order for dynamic task graph scheduling (2024)
13. Chiu, C.H., Xiong, Z., Guo, Z., Huang, T.W., Lin, Y.: An efficient task-parallel pipeline programming framework. In: ACM International Conference on High-Performance Computing in Asia-Pacific Region (HPC Asia) (2024)
14. Guo, G., Huang, T.W., Lin, C.X., Wong, M.: An efficient critical path generation algorithm considering extensive path constraints. In: ACM/IEEE DAC, pp. 1–6 (2020)
15. Guo, G., Huang, T.W., Lin, Y., Wong, M.: GPU-accelerated critical path generation with path constraints. In: IEEE/ACM ICCAD, pp. 1–9 (2021)
16. Guo, Z., Huang, T.W., Lin, Y.: HeteroCPPR: accelerating common path pessimism removal with heterogeneous CPU-GPU parallelism. In: IEEE/ACM ICCAD, pp. 1–9 (2021)
17. Huang, T.W., Lin, C.X., Guo, G., Wong, M.D.F.: Cpp-Taskflow: fast task-based parallel programming using modern C++, pp. 974–983 (2019)
18. Huang, T.W., Guo, G., Lin, C.X., Wong, M.: OpenTimer v2: a new parallel incremental timing analysis engine. In: IEEE TCAD, pp. 776–789 (2021)
19. Huang, T.W., Lin, C.X., Guo, G., Wong, M.D.F.: A general-purpose distributed programming system using data-parallel streams. In: ACM MM, pp. 1360–1363 (2018)

20. Huang, T.W., Lin, C.X., Wong, M.D.F.: DtCraft: a distributed execution engine for compute-intensive applications. In: IEEE/ACM ICCAD, pp. 757–765 (2017)
21. Huang, T.W., Lin, C.X., Wong, M.D.F.: DtCraft: a high-performance distributed execution engine at scale. IEEE ICAD **38**(6), 1070–1083 (2019)
22. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: a lightweight parallel and heterogeneous task graph computing system. In: IEEE TPDS, pp. 1303–1320 (2022)
23. Huang, T.W., Wong, M.D.F.: OpenTimer: a high-performance timing analysis tool. In: IEEE/ACM ICCAD, pp. 895–902 (2015)
24. Huang, T.W., Wu, P.C., Wong, M.D.F.: Fast path-based timing analysis for CPPR. In: IEEE/ACM ICCAD, pp. 596–599 (2014)
25. Jiang, S., Huang, T.W., Yu, B., Ho, T.Y.: SNICIT: accelerating sparse neural network inference via compression at inference time on GPU. In: ACM ICPP (2023)
26. Jiang, S., et al.: FlatDD: a high-performance quantum circuit simulator using decision diagram and flat array. In: ACM ICPP, pp. 388–399 (2024)
27. Lee, W.L., Lin, D.L., Chiu, C.H., Schlichtmann, U., Huang, T.W.: Hyperg: multilevel gpu-accelerated k-way hypergraph partitioner (2025)
28. Lin, C.X., Huang, T.W., Wong, M.D.F.: An efficient work-stealing scheduler for task dependency graph. In: IEEE ICPADS, pp. 64–71 (2020)
29. Lin, D.L., Huang, T.W.: A novel inference algorithm for large sparse neural network using task graph parallelism. In: IEEE HPEC, pp. 1–7 (2020)
30. Lin, D.L., Huang, T.W.: Efficient GPU computation using task graph parallelism. In: Euro-Par (2021)
31. Lin, D.L., Huang, T.W.: Accelerating large sparse neural network inference using GPU task graph parallelism. IEEE TPDS **33**(11), 3041–3052 (2022)
32. Lin, S., Guo, G., Huang, T.W., Sheng, W., Young, E., Wong, M.: G-PASTA: GPU accelerated partitioning algorithm for static timing analysis. In: ACM/IEEE DAC (2024)
33. Morchdi, C., Chiu, C.H., Zhou, Y., Huang, T.W.: A resource-efficient task scheduling system using reinforcement learning. In: IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC) (2024)
34. Reed, E., Chen, N., Johnson, R.E.: Expressing pipeline parallelism using TBB constructs, pp. 133–138 (2011)
35. Zhang, B., et al.: Tap: an incremental task graph partitioner for task-parallel static timing analysis (2025)
36. Zhang, B., et al.: G-PASTA: GPU accelerated partitioning algorithm for static timing analysis. In: ACM/IEEE DAC (2024)