Composing Pipeline Parallelism using Control Taskflow Graph

Cheng-Hsiang Chiu
Tsung-Wei Huang
cheng-hsiang.chiu@utah.edu
tsung-wei.huang@utah.edu
Department of Electrical and Computer Engineering, University of Utah
Salt Lake City, Utah, USA

ABSTRACT

Graph-based propagation (GBP) is a common parallel pattern in many graph computing applications. Many GBP applications compose pipeline parallelism for each linear segment in the graph, where each task encapsulates a sequence of linearly dependent functions. This type of *task-parallel* pipeline parallelism is hard to express using mainstream programming frameworks (e.g., oneTBB) that count on data-parallel models to perform pipeline scheduling. In this paper, we introduce a new task-parallel method to compose pipeline parallelism in a GBP workload by leveraging the state-of-the-art control taskflow graph model. We demonstrate the promising performance of our method on a real circuit simulation workload.

CCS CONCEPTS

• **Theory of computation → Parallel computing models**; • **Software and its engineering → Scheduling**; **Multithreading**.

KEYWORDS

Parallel Pipeline

ACM Reference Format:
Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3502181.3533714

1 INTRODUCTION

Graph-based propagation (GBP) is a task graph-based parallel pattern to describe many graph computing applications. In GBP, each task encapsulates a sequence of linearly dependent functions and each edge denotes dependency between two tasks. Figure 1(a) gives an example. There are nine tasks `A` to `I` and each task encapsulates a sequence of five functions `f1` to `f5`. In a circuit simulation workload [2, 3], the nine dependent tasks model the computation on a circuit network, and each encapsulated function computes a certain timing quantity (e.g., slew, delay, arrival time) in a global graph database.

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06.
https://doi.org/10.1145/3502181.3533714

A typical GBP workload can exhibit many linear segments, such as the linear chain `B`→`F` in Figure 1(a). The functions in each linear segment can run in parallel using *pipeline parallelism*. For example, `f3` of `B`, `f2` of `C`, and `f1` of `D` can overlap in time. As there is no explicit dataflow between adjacent functions but task dependency, we refer to this type of parallelism as *task-parallel pipeline*. Task-parallel pipeline parallelism is not easy to express using existing pipeline programming models (PPMs) (e.g., oneTBB's `tbb::parallel_pipeline` [1]). There are two reasons: First, existing PPMs are *data-parallel* and count on explicit dataflow (e.g., `tbb::make_filter`) to perform pipeline scheduling. Second, existing PPMs are often *standalone* and lack composability with task graph parallelism that is essential for a GBP workload to explore both structural and pipeline parallelisms.

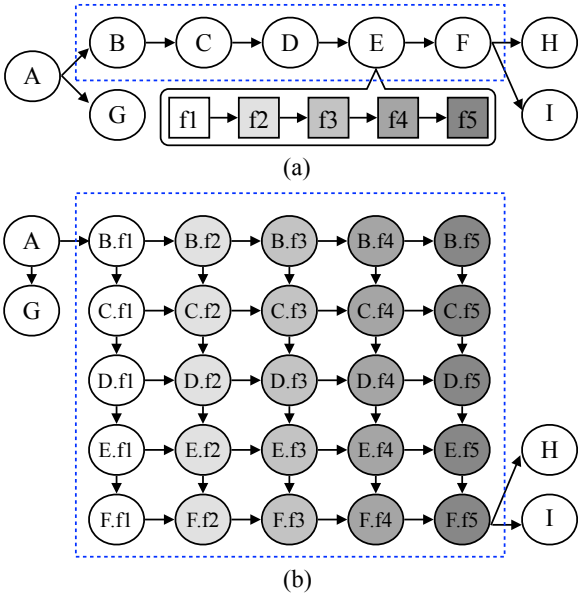

Figure 1: (a) A graph-based propagation example. Each task encapsulates five linearly dependent functions. (b) The flat task graph of unrolled pipeline parallelism in (a).

As a result, a common workaround is to unroll the task-parallel pipeline into a flat task graph, as shown in Figure 1(b). The unrolled task-parallel pipeline results in a total of 29 tasks and 44 dependencies. For large GBP problems with many segments and lengthy function sequences, this workaround becomes inefficient because the flat task graph grows in proportion to the product of

the segment length and the number of encapsulated functions in each task.

To overcome this challenge, we leverage the state-of-the-art *Control TaskFlow Graph* (CTFG) model [4] to express task-parallel pipeline in an end-to-end task graph with in-graph control flow. We have demonstrated the efficiency of our solution on a real circuit simulation workload [2] and improved runtime up to 31%.

2 THE PROPOSED METHOD

As introduced by the Taskflow system [4], a CTFG is a task dependency graph with in-graph control flow tasks, namely *condition tasks*, to describe end-to-end parallelism. A condition task is a callable that returns an integer index indicating the next successor task to execute, and it can support cycles for iterative tasking. In addition to condition tasks, a CTFG is composable, in which a task graph can be composed of modular and reusable blocks, namely *module tasks*.

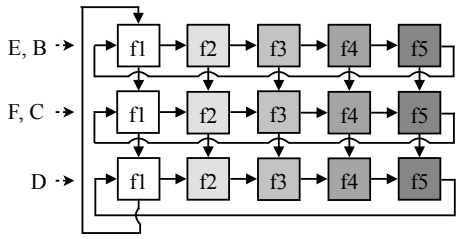

Figure 2: The pipeline diagram of Figure 1(b) running on three parallel lines in a circular fashion.

We express a task-parallel pipeline without unrolling by using condition and module tasks. Our idea is to consider each running task in a linear segment as a *scheduling token* and overlap different tokens across parallel lines in a *circular* fashion. In Figure 2, the pipeline propagates each token through five functions `f1`-`f5` and simultaneously processes up to three tokens at three parallel lines (maximum parallelism). When a token finishes a function, it clears its horizontal and vertical dependencies for the next function on the same line and the same function on the next line, respectively.

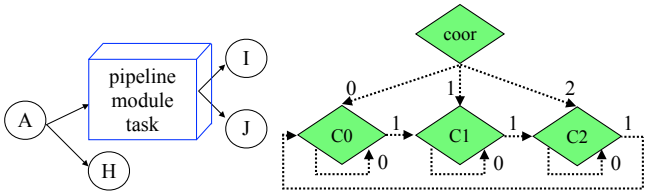

Figure 3: The control taskflow graph of Figure 2.

Figure 3 shows the CTFG of Figure 2. We use (1) a pipeline module task to compose the task-parallel pipeline and (2) four condition tasks to define one coordinator (`coor`) and three parallel lines (`C0`, `C1`, `C2`) in the module task. `coor` decides which line to run when the pipeline starts. Each one of `C0`-`C2` either returns `0` or `1`, indicating a horizontal or a vertical move. Since the pipeline runs in a circular manner, there is a dependency from `C2` to `C0`. Although the execution involves many scheduling tokens, our CTFG uses only four condition tasks instead of performing unrolling, as in Figure 1(b).

Table 1: Benchmark statistics and comparisons of graph size and performance between ours ("o") and the baseline ("b").

circuit	$\|G\|$	$\|G_{unroll}\|$	$\|G_o\|$	T_b	T_o
s526_1	2007	8327	4578	31ms	25ms
s526_2	2027	8459	4633	32ms	27ms
s526_3	1887	7439	4188	29ms	24ms
s526_4	2287	10375	5473	36ms	25ms
vga_lcd_1	897K	3.5M	1.9M	13s	11s
vga_lcd_2	897K	3.5M	1.9M	13s	11s
wb_dma_1	30K	113K	63K	432ms	354ms
wb_dma_2	32K	129K	71K	483ms	361ms

3 EXPERIMENTAL RESULTS

We evaluate the performance of our task-parallel pipeline using an actual circuit simulation workload [2]. We implement our method atop the Taskflow system [4] and compile it using clang++ v10.2 with `-std=c++17` and `-O2` on a Linux machine with Intel i7-9700K 8 Cores at 3.6 GHz and 32 GB RAM. All results are an average of five runs. We use OpenTimer [2] as the baseline, which implements the method in Figure 1(a).

In Table 1, $\|G\|$ represents the graph size of the derived task graph from each of the eight circuits (e.g., Figure 1(a)), and $\|G_{unroll}\|$ represents the unrolled versions (e.g., Figure 1(b)). The unrolled version is much larger because additional tasks and edges are constructed for expanding functions from a linear segment. For the biggest circuit `vga_lcd_1`, the unrolled graph size is `3.5M`, 4× bigger than its original task graph size. In fact, according to our experiments [2], the unrolled task graph has worse performance than its original version due to the overheads of constructing and scheduling too many tasks. Thus, we report here the runtime of the original task graph, denoted as T_b. With our pipeline, the task graph size increases by about 2× as a consequence of additional module and condition tasks and their dependencies. Nevertheless, the benefit is significant–we improve the runtime by 13–31%.

ACKNOWLEDGMENTS

The project is supported by two NSF grants CCF-2126672 and CCF-2144523 (CAREER).

REFERENCES

[1] Intel oneTBB. https://github.com/oneapi-src/oneTBB
[2] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine. In *IEEE TCAD*, Vol. 40. 776–789.
[3] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE IPDPS*. 974–983.
[4] 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 TPDS*, Vol. 33. 1303–1320.