Parallel And-Inverter Graph Simulation Using a Task-graph Computing System

Elmir Dzaka, Dian-Lun Lin, Tsung-Wei Huang
Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT

***Abstract*—Acquiring significant speedup in gate-level simulation has proven challenging due to limitations such as synchronization and partition overhead. As a result, serial event-driven simulation remains the industry standard despite its slow runtime performance. This paper presents the utilization of Taskflow, a task-graph computing system, to effectively enhance the speedup of gate-level simulation. Taskflow provides solutions to challenges faced in previous attempts at parallelizing gate-level simulation, such as scalable pipelines, conditional tasking, and heterogeneous work stealing. The focus of the paper is on improving speedup within and-inverter graphs, which are used to represent structural implementations of circuits at the gate-level. Experimental results demonstrate significant speedup within and-inverter graph benchmarks.**

***Index Terms*—gate-level simulation, and-inverter graph, scalable pipeline, event-driven simulation, Taskflow**

I. Introduction

THE study of gate-level simulation (GLS) is a crucial step in the design verification process of an integrated circuit (IC). Research in functional verification indicates that design verification can consume up to 70% of the overall time spent in product development [2]. After logic synthesis is applied to a chip, a gate-level simulation (GLS) is performed. A GLS provides a netlist view of all gates within the circuit. The netlist view contains a detailed summary depicting all gates with their wiring and timing behaviors. One example of this view is an And-Inverter Graph (AIG), which are commonly used in design verification due to their universality in representing any boolean logic generated by a netlist [28]. GLS algorithms can be used to model AIGs and examine the wiring and timing behavior within them. This information is crucial for various aspects of the VLSI design process such as verifying dynamic circuit behavior, fault testing, and performance and power validation. Due to the importance of GLS, recent research has focused on areas such as accelerated simulation using GPUs [29] and minimizing failure through fault simulation [19].

However, there are significant limitations to current implementations of GLS that hinder performance during this crucial step. Current commercial implementations of GLS are event-driven, meaning that larger circuits result in higher costs and lower performing chips [9]. As circuits increase in size, more ”events” are created for GLS to process. These events include gate delays and wire delays, which are sub-optimally processed in single-core simulation, which result in longer runtimes. In the past, distributed parallel simulation has been proposed as a solution to improve speedup and performance [1], [6]. However, prior attempts to introduce parallelism to GLS have been unsuccessful due to issues such as difficulty in partitioning and load balancing, overhead between partitions, synchronization overhead imposed by the distributed environment, and lack of concurrency in the design process [1]. To address these limitations, this paper introduces the use of a light-weight task graph computing system (TGCS) called Taskflow [15]. Taskflow overcomes these limitations and introduces parallelism to GLS design by doing the following:

- *Multi-Core Simulations* - Previous attempts to introduce parallel simulations to GLS was met with a lack of concurrency in design [1]. Taskflow addresses this limitations by treating the netlist generated by GLS as a directed acyclic graph (DAG) and enabling distributed parallel simulations using features such as heterogeneous work-stealing [22], in-graph control flow, and an expressive programming model [15]. The dynamic task parallelism enabled by heterogeneous work-stealing adapts thread workers to the GLS netlist, allowing for parallelism in any netlist. Additionally, Taskflow's conditional tasking model allows for end-to-end parallelism in almost any DAG file, including those generated by GLS. These features are possible due to Taskflow's implementation of modern C++ closures, which enable efficient parallel simulations without overhead [15].
- *Linear Chain Partitioning Algorithm* - To partition efficiently, researchers have proposed using a hierarchy-based approach for circuit design [3]. However, this method is not applicable to GLS

due to the need for serial partitioning. In order to partition a GLS netlist correctly, this paper presents a new partitioning algorithm that identifies linear chains of gates on-the-fly. These linear chains are processed in a serial manner, allowing for the introduction of dynamic parallelism to GLS through the use of a task-parallel pipeline.
- *Scalable Pipeline* - Taskflow offers composable graph building blocks that enable efficient implementations of parallel algorithms based on design input. This feature can be used to create a scalable pipeline (also known as a task-parallel pipeline), where pipes can be assigned dynamically rather than instantiated at construction time. This approach allows for linear chains to be constructed on-the-fly for any GLS input, making the pipeline adaptable to different designs.

To conduct this research, the AIG simulation algorithm was tested on 18 different circuits represented in DAG files, each with its own logic that needed to be synthesized and optimized. The evaluation environment utilized three variables for testing: the number of testbenches, the number of threads, and the number of cycles. These variables were adjusted during the benchmarking of the AIG simulation algorithm. The results indicate that increasing parallelism through task-parallel pipelining with Taskflow can significantly improve the speedup of the AIG simulation algorithm, reaching up to 79x in some cases. This reduction in synthesis and verification times can greatly benefit gate-level simulation and design verification as a whole.

II. Motivations

This paper aims to address the need for introducing parallelism to GLS as chip design continues to include increasingly larger circuits. The research objective is to advance very large-scale integration (VLSI) design by incorporating heterogeneous parallelism in order to enhance speedup and decrease runtime. Unlike traditional loop-parallel computing problems, many computer-aided design (CAD) algorithms exhibit complex control flows that can benefit from strategic task graph decomposition and heterogeneous parallelism [15]. These advantages can also be applied to GLS, as GLS also involves complex control flows that have yet to fully leverage the potential of heterogeneous parallelism using a traditional TGCS.

III. Related Research

A. Using GPU Architecture

The utilization of Graphics Processing Units (GPUs) as a tool to increase speedup through parallelization has been increasingly popular in recent years [29]. GPUs are used as parallel processing hardware platforms to parallelize the design simulation, resulting in improved speedup. Despite their potential for acceleration, GPUs also present several challenges including communication overhead between the CPU and GPU, inefficient task scheduling, and suboptimal memory access patterns [1], [29].

The benefit of using a TGCS like Taskflow as opposed to GPUs is that it prevents communication overhead between a host and device (CPU/GPU). Additionally, Taskflow features efficient task scheduling which results in an increase in speedup without the need for communication overhead.

B. Multi-core Simulations

Previous attempts to introduce parallelism in GLS design have been made through multi-core machines [1], [20]. The main concept behind multi-core parallelism is to divide the netlist generated by GLS into sub-circuits that can be propagated simultaneously. However, direct applications of multi-core parallelism in GLS design have been unsuccessful due to issues such as unbalanced partitioning, lack of concurrency in the design partitions, and communication and synchronization overhead [1]. Alternative ideas have been proposed to address some of these challenges, such as using a prediction-based model to minimize communication overhead [1], or using temporal parallelism instead of spatial parallelism to avoid design partitions altogether [20]. Despite these methods showing some improvement in GLS design, they only solve part of the problem. Therefore, event-driven simulation remains the most widely used technique for design verification [20].

IV. Implementation Details

This section discusses the four stages of implementation, *baseline breadth-first search (BFS)*, *basic taskflow program*, *linear-chain partitioning algorithm*, and *scalable pipeline*.

A. Baseline BFS

To evaluate the parallel effectiveness of Taskflow, a basic Breadth-First Search (BFS) algorithm was implemented in C++ to represent a serial GLS simulation. BFS was chosen as the baseline because it is a widely used and highly effective method for traversing graphs with cycles. In this scenario, the BFS algorithm traverses all successor gates from a starting node until all gates have been successfully traversed.

During the simulation, the program reads the input stimuli (gates) provided by the GLS algorithm, as well as user-defined inputs such as the *number of cycles*, *number of testbenches*, and *number of threads*. In this case, the GLS algorithm is the baseline BFS algorithm. The program verifies the design by setting stimuli into an output DOT (graph description language) file and checking the output for correctness (as demonstrated in Algorithm 1). The baseline implementation is a single-threaded GLS simulation, which is then compared to a multi-threaded Taskflow implementation. A timer is used to measure various amounts of cycles and testbenches for benchmarking the algorithm. The time measurements are then recorded in a table for comparison and used to calculate the speedup of the implementation. An example of a table generated by the BFS implementation can be seen in Table I.

Algorithm 1 Verifying Design

```
Require: $c \geq 0$, $b \geq 0$, $g \geq 0$
  for $b$ in testbenches do
      for $c$ in cycles do
          while $g$ do                    ▷ while a gate exists
              set_stimulus    ▷ GLS simulation algorithm
```

B. Taskflow

1) Logic Behind Using a TGCS: To incorporate heterogeneous parallelism into GLS design, a Taskflow program was implemented. The concept behind this approach is that a design netlist can be represented as a directed acyclic graph, where each gate is a node that can be propagated from one end to another. This representation allows a Task-Graph Computing System (TGCS) to create a task for each gate, enabling it to be run through multiple worker threads to enable parallelism.

2) Why Taskflow?: Taskflow provides unique solutions to the known challenges of introducing parallelism to design verification (as discussed in Section III.B) that other TGCS programs do not offer, such as conditional tasking and heterogeneous work-stealing [12], [15]. These features allow for minimal communication overhead between design partitions while maintaining concurrency within the design partitions. To address the challenge of unbalanced partitioning, Taskflow utilizes a graph-based design instead of the loop-based design commonly used in many TGCS programs. This feature allows Taskflow to effectively parallelize and balance partitions with the aid of a custom partitioning algorithm and a scalable pipeline, which this paper introduces. More importantly, Taskflow has been successfully applied to accelerate many circuit design problems [21], [10], [16], [17], [26], [27], [8], [4], [30], [5], [7],

3) Taskflow GLS Algorithm: Similar to the baseline, Taskflow takes the input stimuli from the input DAG file and iterates through each gate. Since the input graph is directed, the program iterates through levels (as shown in Figure 1) to ensure that each gate is properly accessed. This method requires tracking the dependencies of each gate to ensure that the correct output is generated. For each gate, a new Taskflow task is created. Once all gates have been successfully iterated over, the dependencies are rebuilt for each task (as demonstrated in Algorithm 2). The dependencies are rebuilt at the end of the algorithm once all tasks are successfully generated.

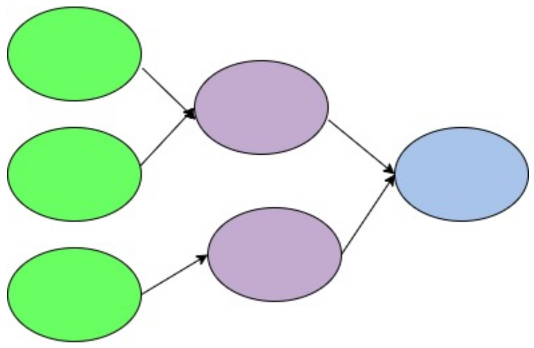


Fig. 1: An example directed-acyclic graph that shows levels by color.

Algorithm 2 GLS simulation Using Taskflow

```
  for $l$ in levels do
      for $g$ in gate do
          for $t$ in testbenches do
              Emplace Task in Taskflow
          track gate dependencies
  build task dependencies
```

C. Linear-Chain Partitioning Algorithm

In order to create a scalable pipeline, a partitioning algorithm is necessary that can partition a graph into subgraphs with minimal overhead. To achieve this, a custom algorithm was developed that scans through edges and vertices on-the-fly to detect linear-chains. A linear-chain is a set of nodes that are serial, meaning a node only has one predecessor or successor in a directed acyclic graph. It is important to note that each node can only be part of one linear-chain, otherwise there would be duplicate nodes in the netlist. An example of this procedure can be found in Fig. 2. Once a linear-chain is detected, the process of building a scalable pipeline on-the-fly can begin.

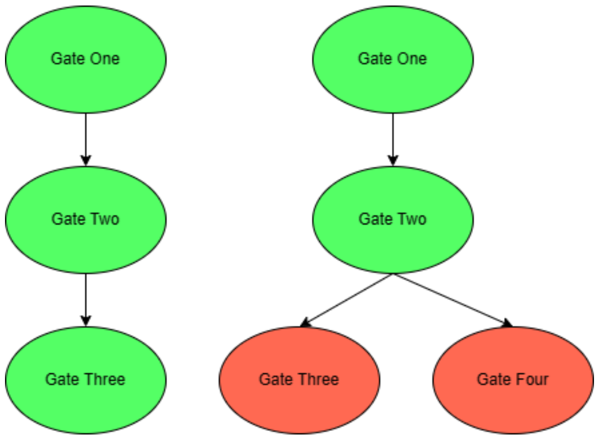


Fig. 2: An example showing a correct linear-chain vs. an incorrect linear-chain.

Algorithm 3 Custom Linear-chain Partitioning Algorithm

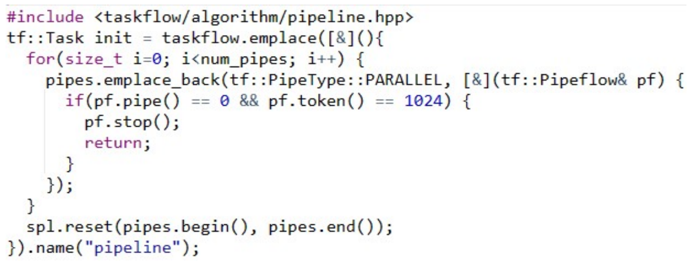
```
while successor count == 1 do                ▷ While serial
    if successor has only 1 predecessor then
        if gate not in any linear chain then
            add to linear chain
```

D. *Scalable Pipeline*

1) What is a Scalable Pipeline?: A scalable pipeline is a task-parallel software pipeline that allows for dynamic assignment of pipes using range iterators (as shown in Fig. 3) [**?**]. The level of parallelism is depicted by the number of lines (rows) in the pipeline. This type of pipeline allows for user-defined variables to be constructed on-the-fly, unlike other pipelines that instantiate all pipes at runtime. For a GLS simulation, a scalable pipeline is crucial as it allows for pipes to be reset through successive runs. Each run represents a different linear chain of varying gate size. It is also important to note that each gate is represented as a pipe in a scalable pipeline.

2) Constructing the Scalable Pipeline: The Taskflow GLS algorithm starts by scanning through each gate and checking whether a linear chain exists using a custom partitioning algorithm (as demonstrated in Algorithm 3). Once a linear chain is detected, a pipeline is constructed on-the-fly (as demonstrated in Algorithm 4). The number of pipes is dependent on the number of gates inside a single linear chain. Since multiple testbenches can exist from a single stimuli, it is imperative that the pipeline keeps track of how many testbenches the program has. The number of testbenches is a fixed number that is defined by user input, which the pipeline divides into groups called *batches*. This procedure allows for each pipe to run a separate batch, allowing for multiple testbenches to be run concurrently. Therefore, maximum parallelism is defined by dividing the number of testbenches by the number of gates within a linear chain (as illustrated in Fig. 4).

In Taskflow, a pipeline is represented as a single task, which enables any serial chain (linear chain) to be parallelized in conjunction with other gates. If a gate is determined not to be a part of a linear chain, the gate is run through the default Taskflow GLS algorithm, as described in Section IV.B.



```
#include <taskflow/algorithm/pipeline.hpp>
tf::Task init = taskflow.emplace([&](){
  for(size_t i=0; i<num_pipes; i++) {
    pipes.emplace_back(tf::PipeType::PARALLEL, [&](tf::Pipeflow& pf) {
      if(pf.pipe() == 0 && pf.token() == 1024) {
        pf.stop();
        return;
      }
    });
  }
  spl.reset(pipes.begin(), pipes.end());
}).name("pipeline");
```

Fig. 3: A sample code snippet of a task-parallel pipeline constructed by Taskflow [15].

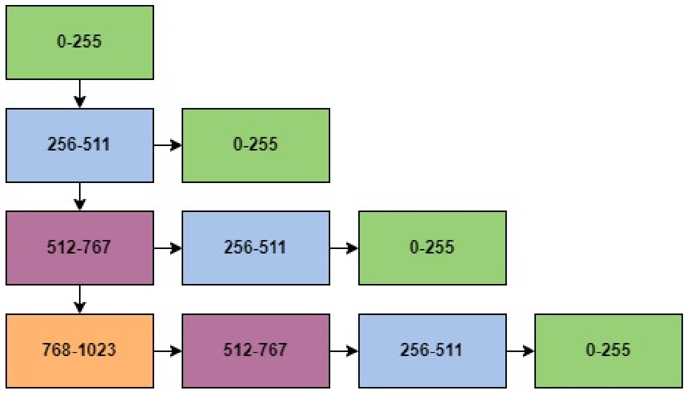


Fig. 4: An example scalable pipeline with 4 gates and 1024 testbenches.

Algorithm 4 Scalable Pipeline Algorithm

```
for gate in linear chain do emplace pipe
    for line in lines do                 ▷ pipeline lines
        simulate pipe
```

V. Results

We implemented this work on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon CPU cores at 2.00 GHz, 4 GeForce RTX 2080 GPUs, and 256 GB RAM. The program is compiled using g++ with C++17 standard with -O3 enabled. Each value is an average of five runs. Five runs was selected since the runtime deviation between each run was negligent (standard deviation close to 1). Circuit benchmarks are derived from real-world industrial designs in [18], [11].

A. Experimental Setup

TABLE I: Linear scalability of the serial baseline implementation

Testbenches	100 Cycles	1000 Cycles	10000 Cycles
1	0.004s	0.05s	0.3054s
10	0.048s	0.2989s	2.72s
100	0.31s	2.66s	25.65s

TABLE II: Multi-core analysis of Taskflow

Testbenches	1 Thread	4 Threads	8 Threads	Speedup(8)
100	56.39s	21.32s	15.63s	3.6
1000	477.42s	134.26	74.153s	6.44

TABLE III: Benchmark metrics

Benchmark	#Edges	#Nodes	#Linear Chains
sim01	12	11	2
sim02	12	11	2
sim03	12	11	2
sim04	1	3	0
sim05	25	16	2
sim06	10716	6451	1048
sim07	19069	9637	195
sim08	12	11	2
sim09	6695	3588	123
sim10	1433	754	1
sim11	6	7	2
sim12	18729	9364	1
sim13	166763	88411	3330

B. Experimental Analysis

1) Serial Baseline: The baseline implementation was designed to simulate a typical event-driven scenario. As per the data in Table 1, it was observed that the runtime increased in a linear manner in accordance with the number of cycles. Furthermore, as the number of testbenches and cycles increased, the runtime also increased significantly, as expected.

TABLE IV: Runtime comparison between scalable pipeline and baseline using 16 threads and two batches

Benchmark	Testbenches	Baseline(s)	Pipeline(s)	Speedup
sim01	10k	<1s	<1s	-
sim01	30k	<1s	<1s	-
sim01	65k	<1s	<1s	-
sim02	10k	<1s	<1s	-
sim02	30k	<1s	<1s	-
sim02	65k	<1s	<1s	-
sim03	10k	<1s	<1s	-
sim03	30k	<1s	<1s	-
sim03	65k	<1s	<1s	-
sim04	10k	<1s	<1s	-
sim04	30k	<1s	<1s	-
sim04	65k	<1s	<1s	-
sim05	10k	<1s	<1s	-
sim05	30k	<1s	<1s	-
sim05	65k	<1s	<1s	-
sim06	10k	1.74	<1s	13x
sim06	30k	10.2	<1s	30x
sim06	65k	21.4	<1s	30x
sim07	10k	2.82	<1s	28x
sim07	30k	13.3	<1s	49x
sim07	65k	35.2	<1s	66x
sim08	10k	<1s	<1s	-
sim08	30k	<1s	<1s	-
sim08	65k	<1s	<1s	-
sim09	10k	<1s	<1s	-
sim09	30k	2.56	<1s	18x
sim09	65k	7.5	<1s	28x
sim10	10k	<1s	<1s	-
sim10	30k	<1s	<1s	-
sim10	65k	<1s	<1s	-
sim11	10k	<1s	<1s	-
sim11	30k	<1s	<1s	-
sim11	65k	<1s	<1s	-
sim12	10k	2.79	<1s	28x
sim12	30k	14.98	<1s	60x
sim12	65k	35.46	<1s	69x
sim13	10k	60.92	<1s	79x
sim13	30k	217.43	3.76	57.8x
sim13	65k	558.13	8.35	66.8x

2) Scalable Pipeline Speedup: Table 3 illustrates the metrics used to evaluate the performance of the GLS implementations. The benchmarks selected for testing were diverse in terms of size, in order to comprehensively evaluate the GLS algorithm. It is worth noting that even though some benchmarks may have the same number of edges and vertices, their graph structure may not be identical. These benchmarks were chosen to also cover edge cases. Additionally, the table tracks the number of linear chains within each benchmark, as determined by the custom partitioning algorithm.

Table 4 demonstrates significant speedup in all cases when compared to the serial baseline GLS algorithm. It is important to note that the speedup for small cases was not considered since the runtime was deemed too

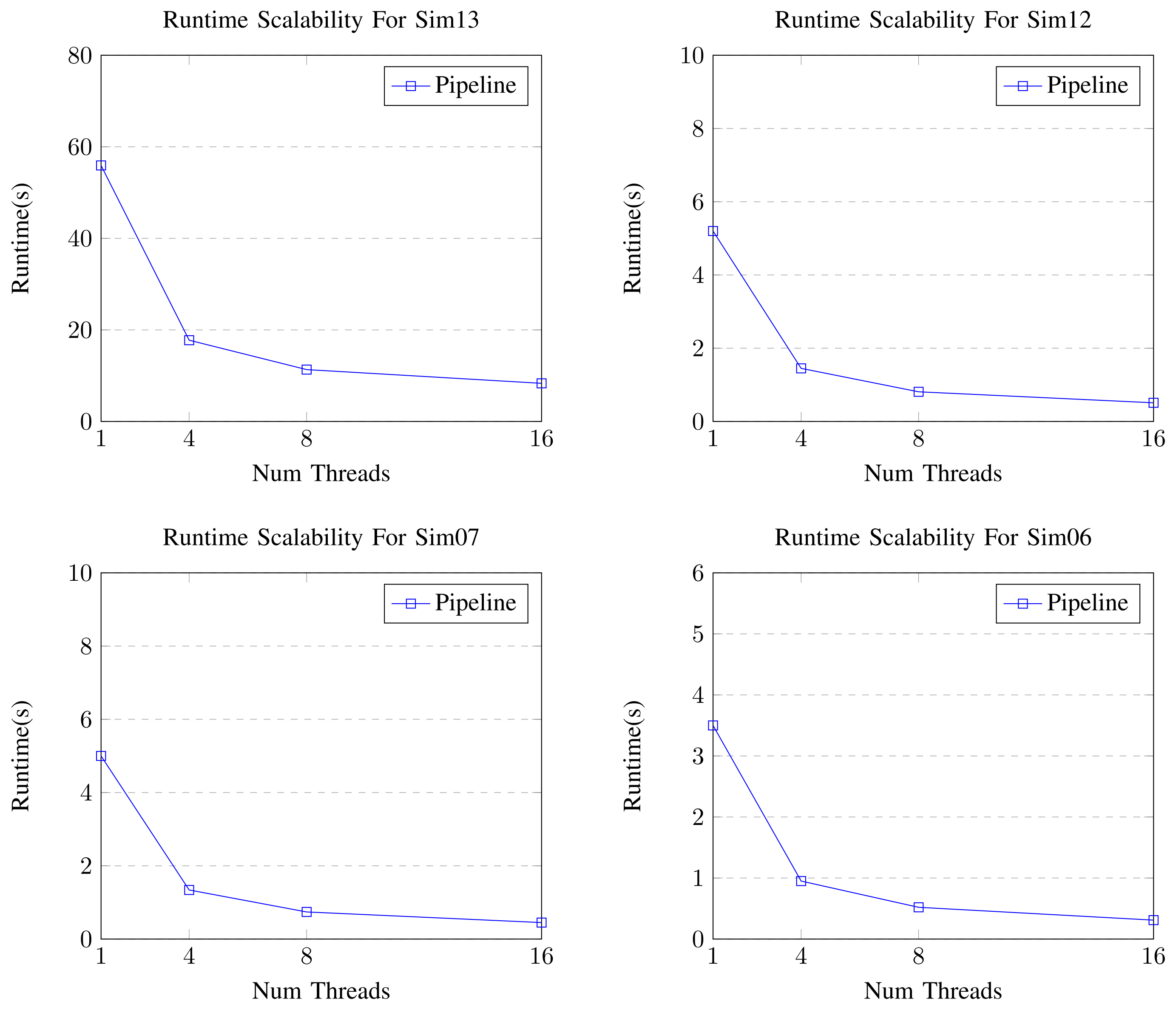


Fig. 5: Runtime Scalability of Four Largest Benchmarks Using a Scalable Pipeline

insignificant to be relevant. In larger cases, the speedup is shown to be between 13x and 79x. Additionally, it can be seen that as the benchmark and testbench size increases, the speedup between the baseline and pipeline implementation also increases. Given that circuit size is rapidly increasing over time, it is expected that using Taskflow for GLS will significantly reduce the time required for design verification.

3) Runtime Scalability Using a Scalable Pipeline: Figure 5 offers a detailed examination at the runtime scalability of the scalable pipeline implemented by Taskflow. For the four largest benchmarks, the runtime scalability remained relatively consistent across each benchmark. As the number of threads increased, the runtime was shortened in all cases. The most substantial improvement in performance was observed between 1-4 threads. While using more than four threads does result in further reduction of runtime, the speedup is considered insignificant when taking into account the overhead costs.

VI. Conclusion

In this paper, we proposed the use of a task-graph computing system named Taskflow to enable heterogeneous parallelism in gate-level design. Our approach is validated using and-inverter graphs, which serve as a structural representation of gate-level designs. We explore the concept of event-driven simulation and the requirement for parallelism in GLS design. We have developed a GLS algorithm that significantly improves speedup by up to 79x. The speedup is achieved due to parallel techniques such as conditional tasking, heterogeneous work stealing, and scalable pipelines. We have also compared this new parallel GLS algorithm to a serial baseline implementation to demonstrate the superiority of our approach over the standard industry implementation. For future work, we plan to leverage the new CUDA Graph execution model [23], [24], [25] or distributed computing to handle large-scale designs [13], [14].

Acknowledgment

We are grateful for the support of National Science Foundation (NSF) grants, CCF-2126672, CCF-2144523 (CAREER), OAC-2209957, and TI-2229304. Also, the authors would like to thank the reviewers for their constructive comments on improving this paper.

References

[1] Tariq Bashir Ahmad and Maciej Ciesielski. An approach to multi-core functional gate-level simulation minimizing synchronization and communication overheads. In *2013 14th International Workshop on Microprocessor Test and Verification*, pages 77–82, 2013.

[2] T Anderson and R Bhagat. Tackling functional verification for virtual components. *Integrated System Design*, 12:26–31, 2000.

[3] Kai-hui Chang and Chris Browy. Parallel logic simulation: Myth or reality? *Computer*, 45(4):67–73, 2011.

[4] Cheng-Hsiang Chiu and Tsung-Wei Huang. Composing Pipeline Parallelism Using Control Taskflow Graph. In *ACM HPDC*, page 283–284, 2022.

[5] Cheng-Hsiang Chiu and Tsung-Wei Huang. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results. In *ACM/IEEE DAC*, page 1388–1389, 2022.

[6] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[7] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM DATE*, 2023.

[8] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. Heterocppr: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism. In *IEEE/ACM ICCAD*, pages 1–9, 2021.

[9] Gary D Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. Springer Science & Business Media, 2007.

[10] Tsung-Wei Huang. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM ICCAD*, 2020.

[11] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD*, 40(4):776–789, 2021.

[12] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. Cpp-taskflow: Fast task-based parallel programming using modern c++. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 974–983. IEEE, 2019.

[13] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. DtCraft: A distributed execution engine for compute-intensive applications. In *IEEE/ACM ICCAD*, pages 757–765, 2017.

[14] Tsung-Wei Huang, Chun-Xun Lin, and Martin D. F. Wong. DtCraft: A High-Performance Distributed Execution Engine at Scale. *IEEE TCAD*, 38(6):1070–1083, 2019.

[15] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320, 2022.

[16] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE TCAD*, 41(5):1448–1452, 2022.

[17] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. Cpp-taskflow: A general-purpose parallel task programming system at scale. *IEEE TCAD*, 40(8):1687–1700, 2021.

[18] Tsung-Wei Huang and Martin Wong. OpenTimer: A high-performance timing analysis tool. In *IEEE/ACM ICCAD*, pages 895–902, 2015.

[19] Endri Kaja, Nicolas Gerlin, Mounika Vaddeboina, Luis Rivas, Sebastian Prebeck, Zhao Han, Keerthikumara Devarajegowda, and Wolfgang Ecker. Towards fault simulation at mixed register-transfer/gate-level models. In *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2021.

[20] Dusung Kim, Maciej Ciesielski, Kyuho Shim, and Seiyang Yang. Temporal parallel simulation: A fast gate-level hdl simulation using higher level models. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[21] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin D. F. Wong. A modern c++ parallel task programming library. In *ACM Multimedia Conference*, page 2284–2287, 2019.

[22] Chun-Xun Lin, Tsung-Wei Huang, and Martin D. F. Wong. An efficient work-stealing scheduler for task dependency graph. In *2020 IEEE ICPADS*, pages 64–71, 2020.

[23] Dian-Lun Lin and Tsung-Wei Huang. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE HPEC*, pages 1–7, 2020.

[24] Dian-Lun Lin and Tsung-Wei Huang. Efficient GPU Computation using Task Graph Parallelism. In *EuroPar*, 2021.

[25] Dian-Lun Lin and Tsung-Wei Huang. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE TPDS*, 33(11):3041–3052, 2022.

[26] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM ICPP*, 2023.

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

[28] Cunxi Yu, Maciej Ciesielski, and Alan Mishchenko. Fast algebraic rewriting based on and-inverter graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1907–1911, 2017.

[29] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. Opportunities for rtl and gate level simulation using gpus. Association for Computing Machinery, 2020.

[30] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM ASP-DAC*, pages 190–195, 2022.