Project Proposal - 15618 Spring 2025

Meghna Jain (meghnaj) and Raveena Gupta (raveenag)

TITLE: Parallel Static Time Analysis

URL: https://mjain00.github.io/parallel-sta/

SUMMARY: Our project focuses on performing static time analysis on circuits to identify if they violate setup and hold time constraints. In this project, we'll leverage task-based parallelism to efficiently provide fast and through analysis on complex circuitry.

BACKGROUND: Our project builds on parallelizing static time analysis to efficiently analyze any time violations in circuits. The application, STA, is used to evaluate the correctness and performance of circuits. The tool takes as input a netlist of gates (AND, OR, NOT) and the flip-flops between them. It then uses cell delay models to calculate gate delays. These gate delays include timing constraints like clock period, setup time, and hold time. The setup time ensures that the signal arrives before the clock edge by a specified margin. The hold time ensures that the data signal remains stable after the clock edge. If the slack of a path is negative, there's a timing violation.

The algorithm is as follows:

1. Parse Netlist
2. Generate graphs to model the circuit (DAG) where nodes are gates and the edges are delay
3. Topological Sorting: The DAG is sorted in topological order so that the nodes (gates) are processed after all their dependencies. The topological sorting algorithm is done using Kahn's algorithm or DFS that traverses through the entire circuit path.
4. Forward Pass: After storing the nodes, the algorithm calculates the arrival time at each node based on the delays of gate
5. Backward Pass: The required time is calculated starting from the output nodes and working backwards. This ensures that the output meets the timing constraints relative to the clock signal.
6. Slack Calculation: The difference between the required arrival time and the arrival time. A negative slack indicates a timing violation.

It's a compute intensive application, especially for a large circuitry because it requires each path to be traversed in a sequential fashion with respect to its data dependencies (components that come before it). Additionally, each path has a different quantity of combinational logic, requiring more computations depending on the traversed path.

Although this algorithm is extremely data dependent, this algorithm can still benefit from parallelism. We can have multiple independent nodes (paths that are independent of each other) processed at the same time.

We can have task-based parallelism where we can divide the path layers into individual tasks that can be executed in parallel. Additionally, we can create a task-graph where each task is a node and edges are dependencies. This will minimize idle time and improve performance. These are all avenues of parallelization our group will look at.

CHALLENGE:

Sequential Dependencies:

- The above algorithm is difficult to parallelize because of sequential dependencies. In static time analysis, the arrival time of a node depends on all its predecessors. Thus, the slack values cannot be determined until the arrival times are received. Parallelizing this requires synchronization.

Workload Imbalance:

- Additionally, multiple parallel tasks read/write shared data (same paths), which can cause cache contention and race conditions. Additionally, some paths might have significantly more gates, which can cause uneven workload distribution.

Memory and synchronization:

- Besides finding the independent sections of the circuit, we also need to find a way to store the task graphs in memory in order to use temporal or local locality. For the sequential version of the algorithm, we would traverse the graph using depth first search (DFS). In the parallel case, we can create a task for each of the branches, and continue running DFS on them. There may be a high communication to computation ratio because a major chunk of the algorithm depends on synchronization of the tasks and sharing the timing values from different paths.
- We will be using a shared memory system, which means that we will have to ensure that updates to shared data are thread safe and free of race conditions. Additionally, mapping a graph to particular threads or cores in order to use temporal or spatial locality is not a trivial task. We would need to ensure that the tasks are divided up in such a way that each thread only has to work on its own section of the graph, and also ensure that it does not “disturb” other sections of the graph until absolutely necessary.

RESOURCES: The resources we will use are:

1. GHC machines to run our code.
2. Paper 1 (Primary): Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In Proceedings of the 2024 International Symposium on Physical Design (ISPD

'24). Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3626184.3635278
3. Paper 2: K. E. Murray and V. Betz, "Tatum: Parallel Timing Analysis for Faster Design Cycles and Improved Optimization," 2018 International Conference on Field-Programmable Technology (FPT), Naha, Japan, 2018, pp. 110-117, doi: 10.1109/FPT.2018.00026. keywords: {Timing;Graphics processing units;Optimization;Kernel;Field programmable gate arrays;Acceleration;Tools;Static Timing Analysis (STA);Parallel Algorithms;GPU;multi-core CPU;FPGA;Computer-Aided Design (CAD)} https://ieeexplore.ieee.org/document/8742253
4. Starter code: we have found the OpenTimer library which is an open source implementation of a different paper than what we found. We will use the library as inspiration for the sequential code.

GOALS AND DELIVERABLES:

Plan to achieve:
In this project, our team plans to significantly speed up the STA algorithm by finding avenues of parallelism. Similar to the assignments in class, there are multiple ways to parallelize this algorithm which we want to experiment with. We want to build on each iteration of our parallel algorithm to see the tradeoffs between performance and accuracy. To begin, we should have a robust sequential code working that traverses paths and finds any timing violation. Then, our team plans to work on at least two separate algorithms:

1. OpenMP
    a. We want to create a graph that can parallelize independent paths (paths with no dependencies) together. This involves creating tasks using BFS, and then going down the branches to calculate time for previous layers, have a synchronization barrier, and parallelize the next components.
2. Task Graph Parallelism
    a. We want to build on OpenMP and create a task graph that has nodes as tasks and edges as dependencies. This gets rid of synchronization barriers as we'll have a task queue where processors will steal tasks that are ready. Although we can't give a specific speedup, we assume that it'll have better performance because we're not iteratively waiting for the previous task to complete.

Hope to achieve:
1. An algorithm our group wanted to explore was using MPI to communicate components or paths that are shared. This requires heavy synchronization and is a tradeoff that needs to be explored.

2. Creating a CUDA kernel that does STA and finding avenues of speedup to see if more processors can bring more speedup.
3. Clusters of tasks Parallelism
    i. We want to partition the circuit in a manner that we have clusters of similar, independent tasks together. The research paper suggests that small insignificant paths have overhead in terms of computation and granularity, so it's imperative to cluster tasks in a manner that is computationally efficient on each processor.

What we hope to learn from this project:

From this project, we want to understand how the STA algorithm's performance can be improved using different parallelization techniques. Our team wants to explore the potential benefits of parallelism for a large circuit and understand which methods provide the best speedup.

The project will compare our OpenMP parallel implementation, Graph Parallelism, and other algorithms and investigate the tradeoffs between performance and the overhead of each technique. Additionally, we want to determine the scalability of these techniques. Some questions we want answered are:

- Do some circuits have better performance with the sequential algorithm due to overhead?
- Is the parallelization more efficient and evident with larger circuits?
- How is slack impacted? How does problem size impact synchronization and overhead?
- How does parallelism impact the correctness in STA? What are the bottlenecks?
- How do we create a task graph that considers data-dependencies?

Our performance goals are divided into speedup, scalability, efficiency, and correctness, which are the tradeoffs we want to see with each strategy.

PLATFORM CHOICE: The platform we have chosen for this project is a multi-core CPU environment using OpenMP. This setup uses a shared address space, which is good because we expect there to be a lot of communication and sharing of data between tasks. While this will require careful synchronization, it can be more efficient because it avoids duplicates of the data and is simpler than using a distributed memory model in this scenario. OpenMP also allows us to dynamically allocate tasks to different threads, as well as define dependencies between the data.

SCHEDULE:

Week	Dates	Tasks
0	3/24 - 3/30	● Plan the outline of the project. ● Set up repository and gather data
1	3/31 - 4/6	● Get sequential code running ● Benchmark the sequential algorithm
2	4/7 - 4/13	● Implement parallel static time analysis using OpenMP - part 1 from paper ● Run experiments to analyse performance ● Complete Milestone report
3	4/14 - 4/20	● Implement the timing updates using task graphs - overcoming synchronization challenges posed by OpenMP - part 2 from paper
4	4/21 - 4/27	● Stretch goal: Implement the CPU-parallel partitioning algorithm - algorithm from paper ● Realistic goal: Complete the task graph implementation
5	4/28 - 5/2	● Final Poster Session