Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System

Dr. Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Wisconsin at Madison, Madison, WI
https://tsung-wei-huang.github.io/
Static Timing Analysis (STA)

• A key step in the overall design flow
  • Verify the timing behavior of a circuit
  • Make sure signals can arrive on time

• Analyze the worst-case scenario
  • Before clock: setup time constraint
    • Latest required arrival time
  • After clock: hold time constraint
    • Earliest required arrival time
Challenge: Modern Circuits are Very Big

STA Runtime vs Circuit Size

STA runtime grows exponentially

Why Do We Need to Parallelize STA?

• Advances the runtime performance to a new level

10-100x speed-up over manycore CPUs

Time (minutes) to speed up a circuit timing analysis algorithm
CPU-parallel STA Algorithms

• Levelization-based vs task-parallel STA\textsuperscript{1}

![Diagram showing parallel and levelized STA algorithms with levelization-based vs task-parallel STA vs hierarchical STA]


Task parallelism allows timing propagation to flow more naturally with the circuit structure.
Levelization-based vs Task-parallel STA

• **OpenTimer v1**: levelization-based parallel timing propagation\(^1\)
  • Implemented using OpenMP “parallel_for” primitive

• **OpenTimer v2**: task-parallel timing propagation\(^2\)
  • Implemented using Taskflow ([https://taskflow.github.io/](https://taskflow.github.io/))

\(^1\): Tsung-Wei Huang and Martin Wong, "OpenTimer: A High-Performance Timing Analysis Tool," *IEEE/ACM ICCAD*, 2015

Overhead of Task-parallel STA

- Task-parallel STA involves two runtime components
  - Build a task dependency graph (TDG) – *often done in sequential*
  - Run the built TDG – *actual parallelization*

- Large circuits induce big TDGs
  - >10M tasks and >10M dependencies

- Big TDG has a big scheduling cost
  - 500–1000us for scheduling a task
  - Dependency breaking
  - Dynamic load balancing
  - Worker notification
  - …

Runtime Breakdown of Task-parallel STA

- Build Graph: 52%
- Run Graph: 48%
Need for a TDG Partitioning Algorithm

• In practice, task-parallel STA saturates at 8–16 cores
  • No need of a TDG of 10M tasks and 10M dependencies

• We can partition a large TDG into a smaller version to
  • Minimize TDG construction time (static overhead)
  • Minimize TDG scheduling overhead (dynamic overhead)
Challenges of TDG Partitioning

- TDG partitioning is very different from circuit graph partitioning
  - Circuit graph partitioning targets minimizing “cut”
  - TDG partitioning targets reducing the graph size without impacting too much its original task parallelism

- TDG partitioning has other constraints to worry about …
  - Cannot introduce too much time on TDG partitioning
  - Cannot introduce cyclic task dependencies
  - Cannot introduce too much sequential parallelism
**G-PASTA: GPU-parallel TDG Partitioner**

---

**Algorithm 1: G-PASTA partitioning kernel**

1. /* Step 1: assign f_pid for current-level tasks by d_pid */
2. parallel for each thread gid { /* gid < Rsize */
3.     cur = handle[Roffset + gid];
4.     cur_pid = d_pid[cur];
5.     if (atomicAdd(pid_cnt[cur_pid], 1) < Ps) then
6.         f_pid[cur] = cur_pid;
7.     else then
8.         new_pid = atomicAdd(max_pid, 1) + 1;
9.         f_pid[cur] = new_pid;
10.        pid_cnt[new_pid] ++;
11.    }
12. /* Step 2: assign d_pid and release dependencies for neighbors */
13. parallel for each thread gid {
14.     for each n in neighbors of cur
15.         /* cycle_free_clustering_algorithm */
16.         atomicMax(d_pid[n], f_pid[cur]);
17.         if (atomicSub(dep_cnt[n], 1) == 1) then
18.             Woffset = atomicAdd(Wsize, 1);
19.                 handle[Roffset + Rsize + Woffset] = n;
20.     }

---

Raw TDG vs Partitioned TDG

• Baseline TDG partitioner: GDCA¹

<table>
<thead>
<tr>
<th>TDG runtime after partitioning (ms/speed-up)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDCA</td>
</tr>
<tr>
<td>3.1 (1.5x)</td>
</tr>
<tr>
<td>16.0 (1.5x)</td>
</tr>
<tr>
<td>21.2 (1.5x)</td>
</tr>
<tr>
<td>153.0 (1.7x)</td>
</tr>
<tr>
<td>175.2 (1.7x)</td>
</tr>
<tr>
<td>193.5 (1.8x)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partitioning runtime (ms/speed-up)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDCA</td>
</tr>
<tr>
<td>7.3</td>
</tr>
<tr>
<td>49.4</td>
</tr>
<tr>
<td>70.7</td>
</tr>
<tr>
<td>727.9</td>
</tr>
<tr>
<td>856.8</td>
</tr>
<tr>
<td>986.9</td>
</tr>
</tbody>
</table>

Why CPU-GPU Heterogeneous STA?

• CPU-based parallelism does not scale beyond 16 threads
  • CPU has limited thread count and memory bandwidth
  • Strong scalability is limited to the circuit structure itself
    • Amdahl's law: https://en.wikipedia.org/wiki/Amdahl’s_law

• Modern STA workloads exhibit a big volume of data parallelism
  • Millions of dates to analyze (in parallel)
  • Hundreds of timing quantifies to propagate through millions of data
Our Research on GPU-accelerated STA

- GPU-based graph analysis (ICCAD’20)
- GPU-based path analysis (DAC’21)
- GPU-based CPPR (ICCAD’21)
GPU-accelerated Critical Path Generation

• GPU-friendly data structure for representing a critical path

Path suffix: \(<e_{14}>\) + Path prefix: \(<e_3, e_8, e_{11}>\) = Path: \(<e_3, e_8, e_{11}, e_{14}>\)
GPU-parallel Suffix/Prefix Tree Building

GPU-based suffix tree construction

(a) STA Graph.

GPU-based prefix tree construction

(b) Shortest path forest.

GPU-accelerated Path-based Analysis

- Example speed-up on a large design, leon2 (1.6M gates)
  - **611x speed-up** over 1 CPU and **44x** over 40 CPUs
  - Evaluated on an Nvidia RTX 3090 GPU

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Pins</th>
<th>#Gates</th>
<th>#Arcs</th>
<th>OpenTimer Runtime</th>
<th>Our Algorithm #MDL=10</th>
<th>Our Algorithm #MDL=15</th>
<th>Our Algorithm #MDL=20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Runtime</td>
<td>Speed-up</td>
<td>Runtime</td>
<td>Speed-up</td>
<td>Runtime</td>
<td>Speed-up</td>
<td>Runtime</td>
</tr>
<tr>
<td>leon2</td>
<td>4328255</td>
<td>1616399</td>
<td>7984262</td>
<td>2875783</td>
<td>4708.36</td>
<td>611×</td>
<td>5295.49ms</td>
</tr>
<tr>
<td>leon3mp</td>
<td>3376821</td>
<td>1247725</td>
<td>6277562</td>
<td>1217886</td>
<td>5520.85</td>
<td>221×</td>
<td>7091.79ms</td>
</tr>
<tr>
<td>netcard</td>
<td>3999174</td>
<td>1496719</td>
<td>7404006</td>
<td>752188</td>
<td>2050.60</td>
<td>367×</td>
<td>2475.90ms</td>
</tr>
<tr>
<td>vga_lcd</td>
<td>397809</td>
<td>139529</td>
<td>756631</td>
<td>53204</td>
<td>682.94</td>
<td>77.9×</td>
<td>683.04ms</td>
</tr>
<tr>
<td>vga_lcd_iccad</td>
<td>679258</td>
<td>259067</td>
<td>1243041</td>
<td>66582</td>
<td>720.40</td>
<td>92.4×</td>
<td>754.35ms</td>
</tr>
<tr>
<td>b19_iccad</td>
<td>782914</td>
<td>255278</td>
<td>1576198</td>
<td>402645</td>
<td>2144.67</td>
<td>188×</td>
<td>2948.94ms</td>
</tr>
<tr>
<td>des_perf_ispd</td>
<td>371587</td>
<td>138878</td>
<td>697145</td>
<td>24120</td>
<td>763.79</td>
<td>31.6×</td>
<td>766.31ms</td>
</tr>
<tr>
<td>edit_dist_ispd</td>
<td>416609</td>
<td>147650</td>
<td>799167</td>
<td>614043</td>
<td>1818.49</td>
<td>338×</td>
<td>2475.12ms</td>
</tr>
<tr>
<td>mgc_edit_dist</td>
<td>450354</td>
<td>161692</td>
<td>852615</td>
<td>694014</td>
<td>1463.61</td>
<td>474×</td>
<td>1485.65ms</td>
</tr>
<tr>
<td>mgc_matric_mult</td>
<td>492568</td>
<td>171282</td>
<td>948154</td>
<td>214980</td>
<td>994.67</td>
<td>216×</td>
<td>1075.90ms</td>
</tr>
</tbody>
</table>

#include <taskflow/taskflow.hpp>
int main()
{
    tf::Taskflow taskflow;
    tf::Executor executor;
    auto [A, B, C, D] = taskflow.emplace(
        [] () { std::cout << "TaskA\n"; },
        [] () { std::cout << "TaskB\n"; },
        [] () { std::cout << "TaskC\n"; },
        [] () { std::cout << "TaskD\n"; }
    );
    A.precede(B, C);
    D.succeed(B, C);
    executor.run(taskflow).wait();
    return 0;
}

// live: https://godbolt.org/z/j8hx3xnnx

Control Taskflow Graph (CTFG) Model

// CTFG goes beyond the limitation of traditional DAG-based models
auto cond_1 = taskflow.emplace([](){ return run_B() ? 0 : 1; }); // 0: is the index of B
auto cond_2 = taskflow.emplace([](){ return run_G() ? 0 : 1; }); // 0: is the index of G
auto cond_3 = taskflow.emplace([](){ return loop() ? 0 : 1; }); // 0: is the index of cond_3
cond_1.precede(B, E); // cycle
cond_2.precede(G, H); // if-else
cond_3.precede(cond_3, L); // loop

Very difficult for existing DAG-based systems to express an efficient overlap between tasks and control flow …
Dynamic Task Graph in Taskflow

// Live: https://godbolt.org/z/j76ThGbWK

tf::Executor executor;
auto A = executor.silent_dependent_async([](){
    std::cout << "TaskA\n";
});
auto B = executor.silent_dependent_async([](){
    std::cout << "TaskB\n";
}, A);
auto C = executor.silent_dependent_async([](){
    std::cout << "TaskC\n";
}, A);
auto [D, Fu] = executor.dependent_async([](){
    std::cout << "TaskD\n";
}, B, C);

Fu.wait();
Everything is Composable in Taskflow

- **End-to-end parallelism in one graph**
  - Task, dependency, control flow all together
  - Scheduling with whole-graph optimization
  - Efficient overlap among heterogeneous tasks

- **Largely improved productivity!**

---

Industrial use-case of productivity improvement using Taskflow

Reddit: [https://www.reddit.com/r/cpp/](https://www.reddit.com/r/cpp/) [under taskflow]

I've migrated [https://ossia.io](https://ossia.io) from TBB flow graph to taskflow a couple weeks ago. Net +8% of throughput on the graph processing itself, and took only a couple hours to do the change. Also don't have to fight with building the TBB libraries for 30 different platforms and configurations since it's header only.
Taskflow is Being Used by Many Projects
Our NSF POSE Project\(^1\): Sustainability

- Create a sustainable Taskflow application ecosystem


---

\(^1\): “POSE: Phase I: Toward a Task-Parallel Programming Ecosystem for Modern Scientific Computing,” $298K, 09/15/2022—08/31/2023, NSF POSE, TI-2229304