An Efficient Implementation of Parallel Breadth-first Search

Pao-I Chen
University of Wisconsin at Madison
Madison, Wisconsin, USA

Tsung-Wei Huang
University of Wisconsin at Madison
Madison, Wisconsin, USA

Abstract

Breadth-first search (BFS) is a fundamental building block of many graph algorithms, such as shortest path finding, network flow analysis, and connected component detection. As the graph size continues to increase, efficient implementations of parallel BFS across multiple cores are critical to the design of scalable graph algorithms. In this paper, we introduce an efficient implementation of the popular bi-directional BFS (BD-BFS) algorithm. Evaluating on the Speedcode platform [3], we can achieve 38.07% throughput performance improvement (3.01B/s vs 2.18B/s) over the conventional BD-BFS implementation.

1 Parallel Breadth-first Search

Breadth-first search (BFS) is a fundamental graph traversal algorithm widely used in diverse applications such as social network analysis, shortest path computation, and web crawling [20]. BFS starts at a given node, often called the *root* in the tree, and explores all its neighboring nodes at the present depth level before moving on to nodes at the next depth level, and so on. The search process continues until all nodes have been explored or the desired target is found. As a result, BFS ensures that nodes are explored *level by level*, making it ideal for finding the shortest path in an unweighted graph. The time complexity of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges [19].

State-of-the-art parallel BFS algorithms [5, 7–18, 21–52, 56, 60–83, 85, 86, 88–92] are typically implemented via a *frontier*-based framework, as outlined in Algorithm 1. The algorithm maintains two separate queues to track the current and the next frontiers during the search process. Starting from a source s in the current frontier queue Q, the algorithm visits all 1-hop neighbors of s and stores visited neighbors in the next frontier queue Q_{next}. In the subsequent rounds, the algorithm iterates between Q and Q_{next}, visit all vertices in Q, and generate Q_{next} by identifying all unvisited neighbors reachable within one hop from the current frontiers. Neighbor searches are parallelized across multiple threads, with concurrent access to Q_{next} safeguarded by atomic *compare-and-swap* (CAS) operations.

Algorithm 1: Parallel Breadth-First Search (BFS)

```
   Input  : Graph G = (V, E), source vertex s
   Output : Distance array dist[|V|], initialized to ∞
1  dist[s] ← 0;
2  Q ← {s};
3  while Q ≠ ∅ do
4  |  Q_next ← ∅;
5  |  foreach u ∈ Q in parallel do
6  |  |  foreach v ∈ Neighbors(u) do
7  |  |  |  if AtomicCAS(dist[v], ∞, dist[u] + 1) then
8  |  |  |  |  Add v to Q_next;
9  |  |  |  end
10 |  |  end
11 |  end
12 |  Q ← Q_next;
13 end
```

Among various parallel BFS algorithms, *bi-directional BFS* (BD-BFS) [6] has demonstrated superior performance over existing methods. When the frontier size is large, instead of generating the next frontier from the current frontier in a *top-down* manner, BD-BFS processes all vertices in parallel and determine for each unexplored vertex if one of its neighbors is in the current frontier queue. If so, the vertex is added to the next frontiers, and all the rest of the neighbor exploration can be skipped. This idea is referred to as *bottom-up* search as it let unexplored vertices identify their parent frontiers in a decentralized fashion. On dense graphs such as social networks, existing results show that BD-BFS is very effective in improving performance by saving work [6].

As shown in Algorithm 2, when the frontier size is large, BD-BFS switches from a *top-down* step, where the next frontier is generated from the current one, to this *bottom-up* step, where unexplored vertices identify their parents from the current frontiers. The bottom-up step process all vertices in parallel and let each unexplored vertex check whether any of its neighbors belong to the current frontier queue. If such a neighbor is found, the vertex is added to the next frontier, skipping further neighbor exploration (line 7). This

FCPC '25, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1446-7/2025/03
https://doi.org/10.1145/3711708.3723443

Algorithm 2: Bottom-up Step

```
   Input  : Current frontier queue Q
   Output : Next frontier queue Q_next
1  Q_next ← ∅;
2  foreach u ∈ V in parallel do
3      foreach v ∈ Neighbors(u) do
4          if v ∈ Q then
5              dist[u] ← dist[v] + 1;
6              Add u to Q_next;
7              break;
8          end
9      end
10 end
```

bottom-up search allows unexplored vertices to identify their parent frontiers in a decentralized manner, which is particularly suitable for parallelization. More importantly, once the parent is identified, no more search is needed, saving a lot of redundant work.

2 Problems of Bi-directional BFS

While BD-BFS offers a great speed-up over conventional BFS, it relies on carefully tuned parameters to balance top-down and bottom-up steps. As shown in Figure 1, this tuning process involves five parameters, (1) n_f (the number of frontiers), (2) m_f (the number of edges to check from frontiers), (3) m_u (the number of edges to check from unexplored vertices), and (4) two user-defined thresholds, C_{TB} and C_{BT}, to decide when to switch from top-down to bottom-up steps and vice versa. Depending on the progress, computing m_f and m_u may be expensive as it requires a parallel reduction to sum up the total number of edges from frontiers. Moreover, implementing the control algorithm correctly is challenging, especially when combined with library-level parameter tuning, such as chunk size adjustments in OpenMP [84].

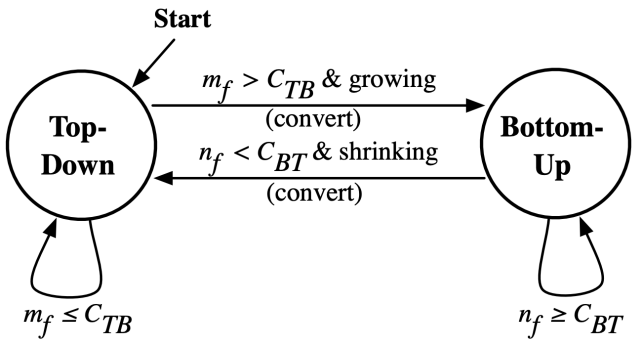

Figure 1: BD-BFS relies on carefully tuned parameters to achieve optimal performance by balancing top-down and bottom-up steps [6].

3 The Proposed Implementation

Instead of maintaining five different parameters (n_f, m_f, m_u, C_{TB}, C_{BT}) to decide when to perform bottom-up and top-down steps, we keep track of *unexplored vertices* that allows us to more precisely decide the workload of the bottom-up step and avoid redundant traversal on explored vertices. Specifically, we maintain a remainder queue, R, of unexplored vertices and only update R during the bottom-up step, as we found in experiments only a few bottom-up steps are needed. With R, we can roughly estimate the number of unexplored edges as $R \times \alpha$, where α is the average number of edges per vertex ($|E|/|V|$). Figure 2 illustrates our idea.

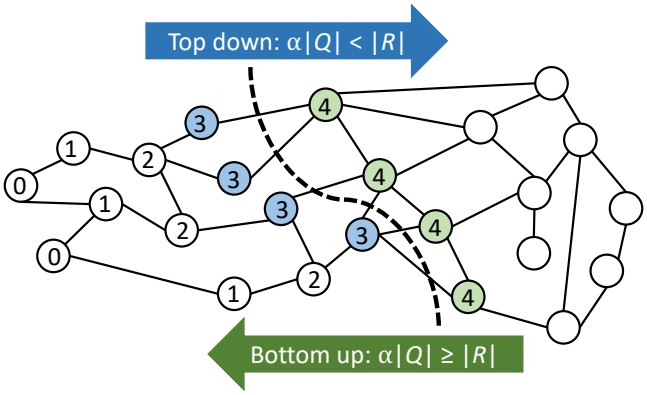

Figure 2: Our algorithm follows a straightforward control flow by directly estimating the workload of the top-down and bottom-up steps. This is done by comparing the number of frontier scans ($\alpha \times |Q|$) with the number of remainder scans (R).

Algorithm 3 presents our implementation. As long as the frontier queue Q and the remainder queue R are not empty, we iteratively identify the next frontiers using either bottom-up step (lines 9:19) or top-down step (lines 22:28). With the information of Q, our control algorithm is a simple comparison between "$|R|$" and "$|Q| \times \alpha$" (line 8). Furthermore, Q provides a very optimistic upper-bound on the number of vertices to traverse in the bottom-up step, to which a parallel traversal can be applied. Note that this design is different from the bitmap data structure in [6], which requires thread safety and still stores/visits all vertices in a bit vector.

3.1 Optimization Details

To further optimize performance, we have incorporated several strategies, summarized below:

- We only perform parallel traversal (line 9 and line 22) when the queue size is greater than 32. This allows us to avoid unnecessary threading overhead when the vertex parallelism is limited.

Algorithm 3: Proposed Algorithm

Input : Graph $G = (V, E)$, source vertex s, current frontier queue Q, current remainder queue R, next frontier queue Q_{next}, next remainder queue R_{next}
Output : Distance array $dist[|V|]$, initialized to ∞

```
1  R ← V − {s};
2  Q ← {s};
3  dist[s] ← 0;
4  α ← |E|/|V|;
5  while |Q| and |R| do
6      Q_next ← ∅;
7      R_next ← ∅;
8      if |R| < |Q| × α then
9          foreach u ∈ R in parallel do
10             if dist[u] ≠ ∞ then
11                 bottom_step(u);
12                 if dist[u] = ∞ then
13                     Add u to R_next;
14                 end
15                 else
16                     Add u to Q_next;
17                 end
18             end
19         end
20     end
21     else
22         foreach u ∈ Q in parallel do
23             foreach v ∈ Neighbors(u) do
24                 if AtomicCAS(dist[v], ∞, dist[u] + 1)
                    then
25                     Add v to Q_next;
26                 end
27             end
28         end
29     end
30     Q ← Q_next;
31     R ← R_next;
32 end
```

- We only initialize R at the first bottom-up step when it is needed. This laziness allows us to avoid unnecessary initialization when bottom-up step never participates in the search.
- We parallelize the bottom-up step using the dynamic loop scheduling algorithm and top-down step using the static loop scheduling algorithm. This is because bottom-up step has an early break (line 7 in Algorithm 2), which can cause workload imbalanced.
- We keep per-thread storage for Q, Q_{next}, R, and R_{next} to minimize the contention when multiple threads try to add vertices to the same queue. Per-worker queues are merged only at the end of the parallel-for execution.

We also experimented with different chunk sizes for parallel-for algorithms. For the dynamic loop scheduling algorithm, we use a chunk size of 32, while for the static loop scheduling algorithm, we use a chunk size of 4.

4 Experimental Results

We implemented our algorithm using OpenMP [84], Taskflow [45, 50], and C++ Thread [1] to evaluate our performance under different libraries. We use OpenMP to implement BD-BFS as our baseline, following the original reference code by the author in [2].

Table 1 compares the performance of runtime (ms) and throughput (edges/s) between our algorithm and BD-BFS across eight large graphs on the Speedcode platform [3]. For this particular contest, "Ours (OpenMP)" is the submission version, which achieved the best overall performance in both runtime (48.31 ms) and throughput (3.01B edges/s), as measured by geometric mean [1]. The Taskflow-powered implementation achieved comparable performance to OpenMP, with a runtime score of 53.02 ms and a throughput score of 2.75B edges/s. We attribute the performance difference to the library overhead, including the dynamic work-stealing [73] and task execution costs. That being said, both outperform the baseline BD-BFS, justifying the effectiveness of our algorithm and the efficiency of our implementations. Unfortunately, the "hard-coded" solution using C++ Thread underperforms library-based implementation. We attribute this to the lack of more adaptive scheduling strategies, such as guided scheduling in OpenMP and work-stealing scheduling in Taskflow.

5 Conclusion

In this paper, we introduced a simple yet efficient implementation to parallelize the bi-directional BFS algorithm. Evaluating on the Speedcode platform [3], we can achieve 38.07% throughput performance improvement (3.01B/s vs 2.18B/s) over the conventional BD-BFS implementation. Inspired by our success of GPU-accelerated graph applications [53–55, 57–59, 87], we plan to extend our BD-BFS algorithm to GPU. The source of our implementation is available on the benchmark folder under the Taskflow repository [4].

Acknowledgments

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141. We also appreciate reviewers' comments on improving this manuscript.

[1] While our solution placed second in the final evaluation, the first-place solution prioritized speed at the cost of thread safety, leading to data race.

Table 1: Overall performance comparison between our algorithm (implemented in three parallel programming libraries, C++ thread [1], Taskflow [50], and OpenMP [84]) and the baseline BD-BFS [6]. Time is measured in milliseconds. All results are collected from the Speedcode platform [3]. "Ours (OpenMP)" is the submission version for FCPC'25 Contest.

			Reference	BD-BFS (OpenMP)		Ours (C++ Thread)		Ours (Taskflow)		Ours (OpenMP)	
	|V|	|E|	Time	Time	edges/s	Time	edges/s	Time	edges/s	Time	edges/s
Collaboration Network 1	1.1M	113M	470	5.21	21.20B	4.28	25.82B	4.07	27.09B	3.90	28.31B
Road Network 1	22.1M	30M	3310	210	283.03M	640	90.74M	160	355.93M	160	356.41M
Road Network 2	87M	112.9M	14670	720	199.15M	760	285.76M	560	387.62M	530	407.30M
Social Network	4.9M	85.8M	1060	20.04	4.19B	13.30	6.31B	17.59	4.77B	13.57	6.19B
Synthetic Dense	10M	1B	11870	43.45	22.61B	40.17	24.40B	41.80	23.44B	40.51	24.19B
Synthetic Sparse	10M	40M	1620	130	293.54M	450	86.44M	90.08	435.21M	85.40	459.06M
Web Graph	6.6M	300M	2860	24.92	11.81B	16.44	17.90B	19.90	14.79B	17.38	16.94B
kNN Graph	24.9M	158M	2100	180	876.23M	320	476.68M	130	1.22B	110	1.42B
Score (Geomean)			2042.57	66.87	2.18B	84.64	1.72B	53.02	2.75B	48.31	3.01B

References

[1] 2025. C++ Thread. https://en.cppreference.com/w/cpp/thread/thread

[2] 2025. GAP Benchmark Suite. https://github.com/sbeamer/gapbs

[3] 2025. Speedcode BFS Benchmark. https://speedcode.org/ide/contest.html?ppopp_test_bfs_v1

[4] 2025. Taskflow Github. https://taskflow.github.io/

[5] Anshul Agarwal and David A Bader. 2010. A scalable hybrid parallel breadth-first search algorithm on multicore CPU and GPU architectures. *High Performance Computing and Simulation (HPCS), 2010 International Conference on* (2010), 1–7.

[6] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah). Article 12, 10 pages.

[7] Aydin Buluc and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. *SC'11: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2011), 1–12.

[8] Che Chang, Cheng-Hsiang Chiu, Boyang Zhang, and Tsung-Wei Huang. 2024. Incremental Critical Path Generation for Dynamic Graphs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[9] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE DAC*.

[10] Che Chang, Boyang Zhang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. PathGen: An Efficient Parallel Critical Path Generation Algorithm. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[11] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[12] Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *ACM ICPP*. 565–575.

[13] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[14] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*.

[15] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2024. An Experimental Study of Dynamic Task Graph Parallelism for Large-Scale Circuit Analysis Workloads. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[16] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*.

[17] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[18] Cheng-Hsiang Chiu, Chedi Morchdi, Yi Zhou, Boyang Zhang, Che Chang, and Tsung-Wei Huang. 2024. Reinforcement Learning-generated Topological Order for Dynamic Task Graph Scheduling. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[20] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1 (1959), 269–271.

[21] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*.

[22] Serhan Gener, Sahil Hassan, Liangliang Chang, Chaitali Chakrabarti, Tsung-Wei Huang, Umit Ograss, , and Ali Akoglu. 2025. A Unified Portable and Programmable Framework for Task-Based Execution and Dynamic Resource Management on Heterogeneous Systems. In *ACM International Workshop on Extreme Heterogeneity Solutions (ExHET)*.

[23] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*.

[24] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).

[25] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[26] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*.

[27] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[28] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[29] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. GPU-accelerated Static Timing Analysis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[30] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*.

[31] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[32] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).

[33] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[34] Zizheng Guo, Zuodong Zhang, Wuxi Li, Tsung-Wei Huang, Xizhe Shi, Yufan Du, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. HeteroExcept: Heterogeneous Engine for General Timing Path Exception Analysis. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[35] Pawan Harish and P J Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. *International Conference on High Performance Computing* (2007), 197–208.

[36] Sungpack Hong, Teddy Oguntebi, and Kunle Olukotun. 2011. An efficient parallel graph coloring algorithm for multi-core architectures. *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), 320–330.

[37] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[38] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*.

[39] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[40] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[41] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[42] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[43] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*.

[44] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*.

[45] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[46] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*.

[47] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[48] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[49] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test (DAT)* (2021).

[50] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).

[51] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2022).

[52] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*.

[53] Tsung-Wei Huang, Hong-Yan Su, and Tsung-Yi Ho. 2011. Progressive network-flow based power-aware broadcast addressing for pin-constrained digital microfluidic biochips. In *ACM/IEEE Design Automation Conference (DAC)*. 741–746.

[54] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[55] Tsung-Wei Huang and Martin Wong. 2016. UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2016).

[56] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*.

[57] Tsung-Wei Huang, P.-C. Wu, and Martin Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM ICCAD*.

[58] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Route: An Ultra-Fast Incremental Maze Routing Algorithm. In *ACM System Level Interconnect Prediction Workshop (SLIP)*. 1–8.

[59] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE/ACM ICCAD*.

[60] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*.

[61] Shui Jiang, Yi-Hua Chung, Chih-Chun Chang, Tsung-Yi Ho, and Tsung-Wei Huang. 2025. BQSim: GPU-accelerated Batch Quantum Circuit Simulation using Decision Diagram. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[62] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[63] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*.

[64] Jiang, Shui and Fu, Rongliang and Burgholzer, Lukas and Wille, Robert and Ho, Tsung-Yi and Huang, Tsung-Wei. 2024. FlatDD: A High-Performance Quantum Circuit Simulator using Decision Diagram and Flat Array. In *ACM ICPP*. 388–399.

[65] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*.

[66] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[67] T.-Y. Lai, Tsung-Wei Huang, , and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[68] Wan-Luan Lee, Shui Jiang, Dian-Lun Lin, Che Chang, Boyang Zhang, Yi-Hua Chung, Ulf Schlichtmann, Tsung-Yi Ho, , and Tsung-Wei Huang. 2025. iG-kway: Incremental k-way Graph Partitioning on GPU. In *ACM/IEEE Design Automation Conference (DAC)*.

[69] Wan-Luan Lee, Dian-Lun Lin, Cheng-Hsiang Chiu, Ulf Schlichtmann, and Tsung-Wei Huang. 2025. HyperG: Multilevel GPU-Accelerated k-way Hypergraph Partitioner. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[70] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*.

[71] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*.

[72] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[73] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.

[74] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*.

[75] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[76] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.

[77] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).

[78] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

[79] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing (ICPP)*.

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

[81] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

[82] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[83] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*.

[84] OpenMP Architecture Review Board. 2021. *OpenMP Application Programming Interface Version 5.2*. https://www.openmp.org/specifications/ Accessed: January 27, 2025.

[85] R Pearce, M Gokhale, and N Amato. 2010. Multithreaded graph traversal, partitioning, and layout. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA)* (2010), 103–114.

[86] Jie Tong, Liangliang Chang, Umit Yusuf Ogras, and Tsung-Wei Huang. 2024. BatchSim: Parallel RTL Simulation using Inter-cycle Batching and Task Graph Parallelism. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[87] Sheng-Han Yeh, Jia-Wen Chang, Tsung-Wei Huang, Shang-Tsung Yu, and Tsung-Yi Ho. 2014. Voltage-Aware Chip-Level Design for Reliability-Driven Pin-Constrained EWOD Chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 33, 9 (2014), 1302–1315.

[88] Andy Yoo, Edmond Chow, Keith Henderson, Mahidhar T Rajan, and William C McLendon. 2005. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. *Proceedings of the ACM/IEEE SC 2005 Conference (SC'05)* (2005), 25.

[89] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[90] Boyang Zhang, Che Chang, Cheng-Hsiang Chiu, Dian-Lun Lin, Yang Sui, Chih-Chun Chang, Yi-Hua Chung, Wan-Luan Lee, Zizheng Guo, Yibo Lin, and Tsung-Wei Huang. 2025. iTAP: An Incremental Task Graph Partitioner for Task-parallel Static Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[91] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

[92] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.