Conflict Based Algorithm to Parallelize Minimum Spanning Tree

Prof. Sathya Peri

Gadekar S.B. | Siddhant Godbole

What is a Minimum Spanning Tree ?

- Minimum Spanning Tree (MST): A minimum spanning tree is a set of edges connecting all graph nodes with no cycles and the smallest total connection cost possible.
- Goal: Using Multi-threading approach on MST algorithm to improve performance for large and dense graphs.

Sequential MST Algorithms:

- Prim's Algorithm O E log V): Grows the MST from a starting node by repeatedly adding the smallest edge to an unvisited node.
- Kruskal's Algorithm O(E log E): Sorts edges and adds the smallest edge that does not form a cycle (uses Union-Find data structure).
- Borůvka's Algorithm O(E log V): Adds the smallest edge from each connected component, merging them iteratively until all vertices are connected. It is highly parallelizable due to independent component processing.

Challenges in Sequential Computation:

- High *time complexity*.
- Poor scalability with large and dense graphs.

Conflict Based Approach Towards MST

- Consideration: The minimum outgoing edge of a vertex "u" is its external edge with minimum weight resolved by index value the destination vertex "v".
- Consideration: A group is a **Acyclic connected component**.
- If we look at an MST of a connected graph with n vertices and pick the lightest edge leaving each vertex, we get n edges. But an MST only has n -1 edges, so one of those chosen edges must appear twice—once from each of its endpoints.
- **Conflict** : In an undirected connected graph with n vertices, we pick the minimum-weight outgoing edge for each vertex. If two different vertices choose the same edge as their minimum outgoing edge, we say that edge is a **conflict**.

Algorithm

1. Get minimum edge of each vertex/super-vertex(set of vertices).
2. Get conflicts, These define the group. Number of groups = Number of conflicts.
3. Groups are now considered as super-vertex.
4. Repeat steps 1 to 3, until number of conflicts become <= 1.

Implementation Details

Phase 1: Initial Minimum Edge Selection

Objective: Each vertex independently finds its minimum weight outgoing edge

- Parallel Strategy: Work-stealing approach using atomic counter
- Synchronization: Barrier synchronization using atomic counters and condition variables

Phase 2: Conflict Detection & Initial Tree Formation

Objective: Identify mutual edge selections (conflicts) and build initial forest

- Conflict Detection: If vertex u selects v AND v selects u → creates conflict group
  - Greater ID becomes group head to ensure deterministic resolution, add this edge once.
- Direct Edge Addition: Non-conflicting edges added directly to MST
- Leaf Identification: Mark endpoints of conflict chains for chain traversal

Phase 3: Conflict Chain Traversal

Objective: Map hierarchical structure of conflict chains from leaf nodes

- Traverse backward from each leaf through the conflict chain, establishes order1 (forward) and order2 (backward) pointers for chain navigation

Phase 4: Intra-Group Edge Elimination

Objective: Update minimum edges by eliminating edges within same group

- Path Compression: Aggressively compress Union-Find paths for O(α(n)) lookup
- Edge Invalidation: Mark intra-group edges as 0 (used) in adjacency structure
- Recomputation: Find new minimum cross-group edges for each vertex

Phase 5: Iterative Supervertex Merging

Multi-step iterative process until single component remains:

Step 5.1: Branch Minimum Calculation

- For each conflict chain (from leaf to head), find minimum outgoing edge

Step 5.2: Group-Level Edge Selection

- Aggregate branch minimums to find best outgoing edge per group head

Step 5.3: Inter-Group Conflict Resolution

- Detect Conflicts: Check if two groups mutually select each other
- Resolution Strategy: Higher group head ID "wins" & store the lower ID group
- Edge Addition: Add selected edges to MST and update group membership

Step 5.4: Prepare Next Iteration

- Group Membership Update: Path compression on newly merged groups
- Edge Revalidation: For branches whose minimum edge was used:
  - Traverse the chain
  - Recompute minimum cross-group edges

Loop Termination: Repeat Phase 5 until conflict ≤ 1

Single connected component.

Parallelization Strategy:

Thread Coordination lock-free:

- Work Distribution: Atomic fetch-and-add for dynamic load balancing
- Synchronization Points: barrier points using atomic counters and busy-wait loops for fast reactions
- Lock-Free contention using atomic operations for shared counters

Key Optimizations:

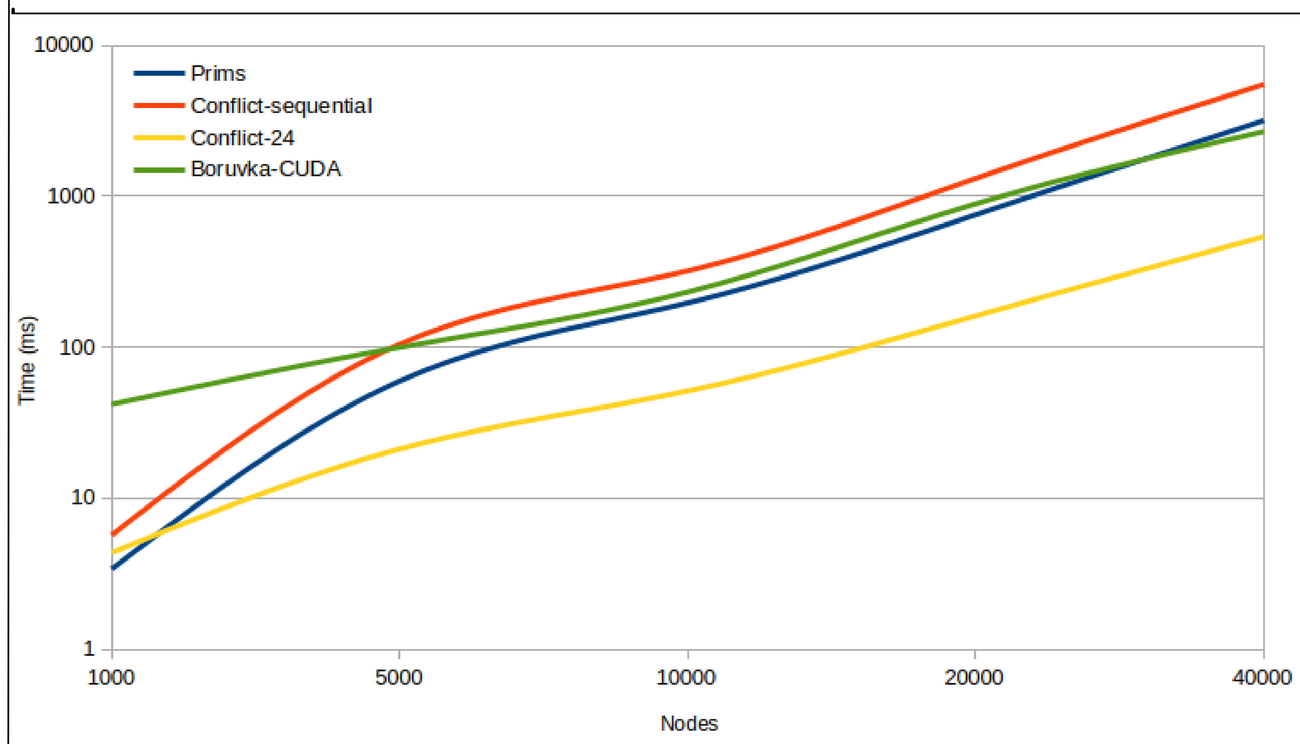
- **Compressed Adjacency Storage**: Uses negative indices to represent gaps, reducing memory
- Path Compression: Ensures nearly constant-time group lookups
- Early Termination: Checks conflict count at multiple points to exit early

Advantages:

- Fewer redundant edge comparisons through conflict resolution
- Efficient for graphs with localized connectivity patterns
- Scalable Synchronization: Uses fine-grained atomic operations instead of coarse-grained locks, minimizing contention and allowing better multi-core scaling compared to mutex-heavy implementations.

Trade-offs

- More complex bookkeeping (chains, groups, metadata)
- Multiple barrier synchronizations in iterative phase
- Performance **sensitive** to conflict distribution in graph

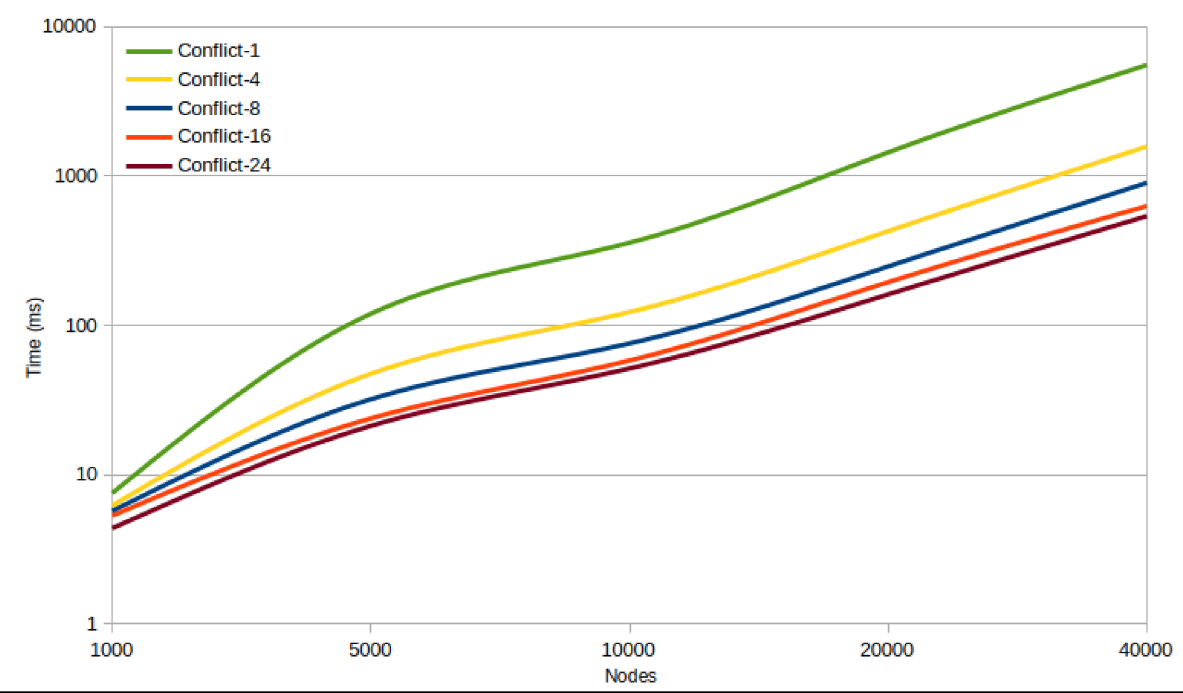


Comparing Our Algorithm with:

- prims-seq
- conflict-seq
- conflict-24
- boruvka-CUDA

Comparing Our Algorithm on various thread counts:

- conflict-1
- conflict-4
- conflict-8
- conflict-16
- conflict-24



Performance Observations:

Graph 1: Single-Threaded Comparison (1K-40K Nodes)

- Competitive baseline performance: Our conflict-based algorithm maintains execution times comparable to classical MST algorithms (Prim's, Kruskal's, Borůvka's) in single-threaded Design.
- Consistent scaling: Algorithm shows relatively flat execution time across different graph sizes (1K to 40K vertices), indicating good scalability characteristics

Graph 2: Multi-Threaded Scaling (2-24 Threads)

- Superior parallel speedup: Our algorithm achieves the best speedup among all tested algorithms, reaching approximately 7-9x faster than sequential execution at optimal thread counts.
- Outperforms Borůvka's: Despite Borůvka's being considered highly parallelizable, our approach demonstrates superior or comparable multi-threaded performance with simpler synchronization mechanisms.