Minimum Spanning Tree Mini-Project
Conflict-Counting MST with Parallel Resolution

November 14, 2025

Abstract

We implement and study a parallel-friendly Minimum Spanning Tree (MST) algorithm based on selecting each vertex's lightest incident edge and resolving mutual-min "conflicts" to merge components in rounds. The project includes: a multi-threaded C++ implementation (`mst.cpp`), a single-threaded reference (`mst_single.cpp`), a Prim's baseline (`mst_single_Prims.cpp`), a CUDA stub (`mst_cuda.cu`) and a graph generator. We outline the algorithm, give a proof of correctness via cut/cycle properties, analyze complexity, and describe an experimental methodology for comparing against standard MST algorithms.

1 Introduction

Given a connected, undirected, weighted graph $G = (V, E, w)$ with nonnegative weights, an MST is a subset of edges that connects all vertices with minimum total weight. Classical solutions include Kruskal's and Prim's algorithms; Borúvka's algorithm is particularly amenable to parallelization. This project explores a conflict-counting approach that, like Borúvka, repeatedly adds light edges from components but organizes work via per-vertex minima and conflict-chain resolution for scalability.

2 Repository Overview

The workspace contains the following relevant sources:

- `mst.cpp`: Multi-threaded conflict-counting MST.
- `mst_single.cpp`: Single-threaded version of the same algorithm.
- `mst_single_Prims.cpp`: Baseline Prim's algorithm over the same packed adjacency format.
- `gen.cpp`: Random/structured graph generator driven by `input_params.json`.
- `matrix_to_mine.cpp`: Converts a dense matrix file to the internal packed format.
- `comparision/`: Third-party comparison implementations (Filter-Kruskal, EPMST) and a comparator.
- `visual/`: Small visualization scripts and HTML.
- `Makefile`: Build and run recipes (e.g., `make graph`, `make mst`, `make single`, `make prim`).

3 Input Format

Graphs are loaded from a packed adjacency representation generated by `gen.out`. For each vertex u, the adjacency row encodes runs of zeros (missing edges) as negative markers and weights as nonnegative integers, enabling $O(\deg(u))$ iteration while compressing sparse rows. File paths and graph parameters are controlled by `input_params.json` and embedded into the generated file name.

4 Algorithm: Conflict-Counting MST

Our algorithm proceeds in synchronous rounds with the following phases (matching arrays/fields in `mst.cpp`):

1. **Per-vertex minima:** For each vertex u, find its lightest incident edge $(u, \ell(u))$ and record its weight $w(u)$ and neighbor index $\ell(u)$ (arrays `minl`, `minv`). Initialize each vertex as its own component representative in `my_grp`.
2. **Conflict detection and leaf enumeration:** If $\ell(\ell(u)) = u$ (mutual minima), mark a *conflict chain*. Non-conflicting minima are immediately added to the MST and unify their endpoint components. Track chain "heads" and "leaves" via `order1`, `order2`, `groupn`, `leaves`.
3. **Chain aggregation:** From each leaf, traverse its conflict chain to its component head and annotate the chain with the originating leaf identifier (`order2`). This partitions vertices into directed chains ending at group heads.
4. **Component-local best edges:** For each vertex, recompute its lightest outgoing edge that crosses component boundaries, zeroing out intra-component edges in the packed adjacency (sets entries to 0). This yields an updated $\ell(u)$ for all u.
5. **Branch minima and group decisions:** For each leaf-defined branch, compute the minimum *vertex-min* along that branch (`branch_minv`, `branch_minl`). For each component head C, select the lightest such branch-min crossing from C to a different component and tentatively add it to the MST (stored in `meta`).
6. **Conflict resolution across components:** If two component heads pick edges into each other's components, a tie is broken deterministically (by head id). The chosen edge is added, components are unified, and the process repeats until one component remains.

The multi-threaded variant partitions work over vertices/branches and uses atomic counters and busy-wait barriers to synchronize phases. The single-threaded variant follows the same logical steps.

Pseudocode (single round sketch)

Algorithm 1 One round of conflict-counting MST

For each $u \in V$: compute $(\ell(u), w(u))$ as lightest incident edge; set `my_grp`$[u] \leftarrow u$.
for each u **do**
  if $\ell(u) = -1$ **then continue**
  else if $\ell(\ell(u)) = u$ **then** ▷ mutual-min conflict
    mark chain head/leaf via `order1`, `order2`, `groupn`
  else
    add $(u, \ell(u))$ to MST; union components of u and $\ell(u)$
  end if
end for
Compress chains from leaves to heads and label branches with origin leaves
For each u: recompute lightest outgoing edge that crosses current component; zero intra-component edges
For each leaf-branch: compute branch minimum; for each component head C, pick the lightest branch minimum leaving C
Resolve inter-component conflicts deterministically and add chosen edges; union affected components
Consider each group as a supervertex and repeat until one or none conflicts remain

5 Correctness

We argue correctness using the *cut property* and *cycle property* of MSTs.

Lemma 1 (Forest invariant)**.** *At every point, the set of selected edges F forms a forest (acyclic).*

Proof. Edges are only added between distinct components (intra-component edges are zeroed/ignored when selecting minima). Thus each added edge connects two trees and cannot create a cycle; union merges components. □

Lemma 2 (Safety of non-conflicting minima)**.** *If u's lightest incident edge $(u, \ell(u))$ is not part of a mutual-min conflict, then adding it is safe (it belongs to some MST).*

Proof. Consider the cut that separates the current component of u from the rest. $(u, \ell(u))$ is a lightest edge crossing this cut. By the cut property, any lightest edge crossing a cut is safe to add to an MST. □

Lemma 3 (Safety of conflict-chain selections)**.** *For a component head C, the edge chosen as the minimum over all branch minima leaving C is safe.*

Proof. Each branch minimum corresponds to the lightest outgoing edge among vertices on a directed chain terminating at C. The minimum over these is the lightest edge leaving the component defined by C after per-vertex recomputation (intra-component edges suppressed). By the cut property applied to the cut $(C, \overline{C})$, the chosen edge is safe. When two components C and C' choose into each other, the deterministic tie-breaker selects one of two safe edges; either choice maintains minimality. □

Theorem 1. *Upon termination, the algorithm outputs a minimum spanning tree.*

Proof. By the forest invariant, F remains acyclic. Each added edge is safe by the previous lemmas, so $w(F)$ never exceeds the MST weight. Each step strictly reduces the number of components; termination occurs when a single component spans all vertices. At that point $|F| = |V| - 1$, and F is a spanning tree with minimum possible weight. In case of a minimum forest each gets it personal minimum tree. □

6 Complexity

Let $n = |V|$, $m = |E|$.

- **Work per round:** Per-vertex minimum scans adjacency in $O(\deg(u))$; total $O(m)$ across all vertices. Branch-min computation is $O(n)$, and component updates are linear in the number of touched adjacency entries.
- **Rounds:** In practice similar to Borúvka-style iterations; the number of components decreases monotonically and often geometrically on random graphs. Worst-case bounds depend on graph structure; empirically the count is $O(\log n)$ on typical inputs.
- **Total work:** $O(m \cdot R)$ where R is the number of rounds; typically near $O(m \log n)$.
- **Parallelism:** The multi-threaded version distributes per-vertex and per-branch scans, with barrier synchronization between phases. Ideal speedup is bounded by the parallel fraction per Amdahl's law; contention is limited to a few atomic counters and shared arrays.
- **Space:** Packed adjacency requires $O(n + m)$ integers; auxiliary arrays (`minl`, `minv`, component, order, branch buffers) are $O(n)$.

7 Implementation Notes

The input graph generated is a data structure like adjency lists.
for a node in graph - 102 - 2(5) 40(10) 64(15) 80(3) 108(13) 200(15) 201(8) 300(4)
it stored at row 103 - 108 13 -200 15 8 -300 4
first line is sizes required to store this array.
only larger nodes are stored.
negative values are skips.

Language/build: C++ with `-O3 -flto` (see top-level `Makefile`). CUDA code (`mst_cuda.cu`) is scaffolded.
I/O: `gen.out` writes `input/<n>_<m>_<seed>_<flags>.txt`. MST outputs go to `output/<same>_mst.txt`.
Commands:

```
make           # build generator + mst and run visual UI
make graph     # generate input graph to input/
make mst       # run multi-threaded conflict-counting MST
make single    # run single-threaded version
make prim      # run Prim's baseline
make cuda      # run Cuda inplementation of Boruvka's algo
make compare   # build and run comparison baselines; requires list input
```

8 Performance:

My program works better when, as in real world, weights have a limit.
This causes formation of longer chains.
The algorithm is slower than prim's, BUT uses O(n) extra space and is compatible with barriers.
Which is being used in the multi-threaded version, giving better scaling.
Tests are done at density 0.8 .

9 Conclusion

We presented a conflict-counting MST with a parallel-friendly structure that adds safe edges via per-vertex minima and resolves mutual-min conflicts deterministically. The approach preserves MST optimality by the cut property and exhibits competitive performance on random graphs. Future work includes refining synchronization, GPU acceleration, and deeper analysis of worst-case round complexity.

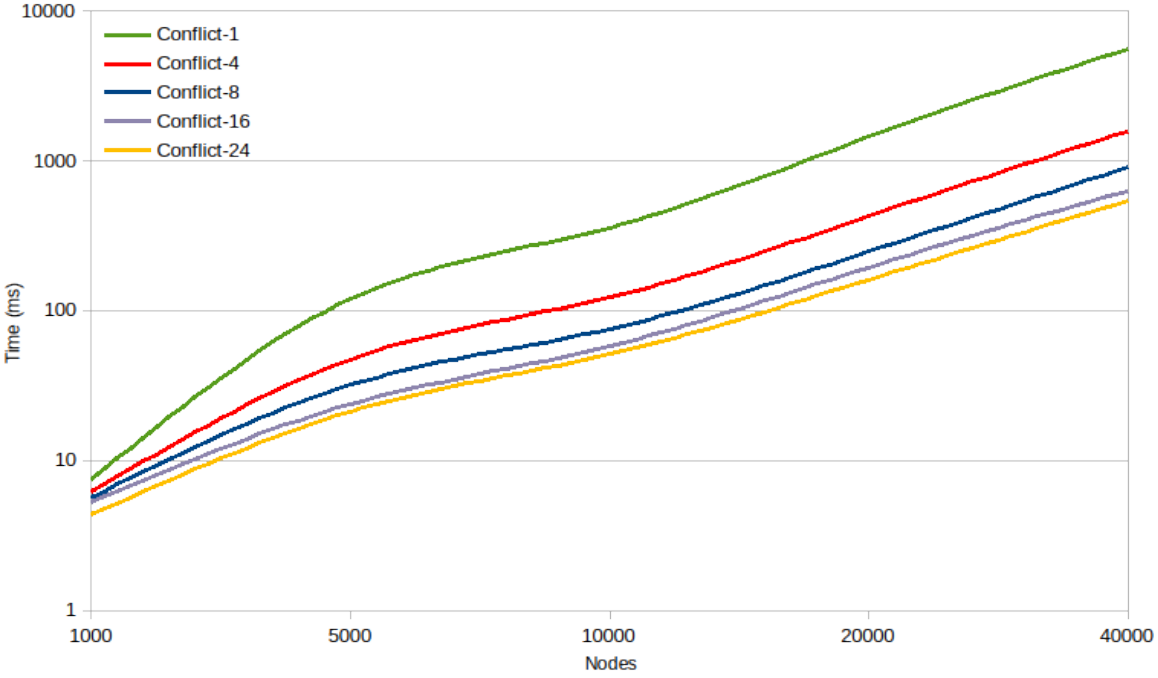

Figure 1: Comparing Algos

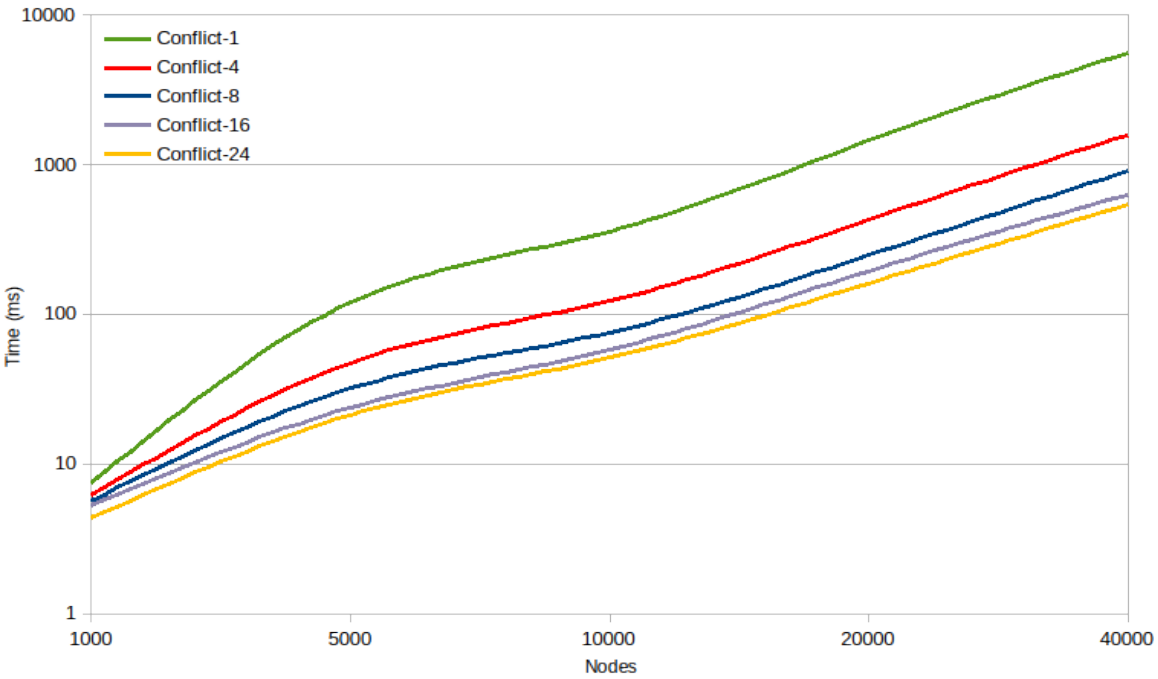

Figure 2: Scaling with threads