Parallel Minimum Spanning Tree Data Structures and Algorithms
Connor Clayton and Jacqui Fashimpaur

**Boruvka’s Algorithm**
This is a common algorithm for finding the minimum spanning tree. Boruvka’s algorithm initially considers each vertex to be its own component, and then proceeds in three stages until there is only one component remaining:

1. **Find-min**: for each component, find the minimum weight edge leaving that component. Add edges to the MST.
2. **Connect-components**: using the edges found in find-min, identify which components are now connected and group them together.
3. **Compact-graph**: condense each connected component into a single supervertex.

Once only one component remains, all vertices have been connected and the MST has been found.

**Prim’s Algorithm**
This is a common algorithm for finding the minimum spanning tree.

**Minimum Spanning Trees**
A Minimum Spanning Tree (MST) of a weighted, undirected, connected graph is an acyclic subset of the edges such that all vertices are connected, and the total edge weight is the minimum possible. There are many algorithms for finding an MST, but all rely on the property that for any subset of the MST, the lowest-weight edge intersecting this component will be in the final MST.

**Flexible Adjacency Lists**
This data structure stores multiple adjacency lists at each vertex as a linked list. To merge two vertices, one simply needs to point one vertex’s flexible adjacency list to the other. This is useful for Boruvka’s algorithm in that it shifts work away from compact-graph, which is usually the most costly step.

**Approach to Parallelization**
The basic parallelizable unit in Boruvka’s is the connected component. We organized our algorithms into distinct stages which capitalize on independent components. We also implemented Flexible Adjacency Lists to remove large sequential portions of explicit merging.

**Results**
Runtime for the standard implementation increased modestly with graph density, while that of the merging implementations increased dramatically. The FAL implementation out-performed all others due to large speedups in compact-graph. Overall speedup was moderate within implementations due to section dependencies and load-balancing challenges.

**Reference**