Project Report

Team Members

1. Bharadwaja rao D CS20BTECH11012
2. Shashank Anirudh R CS20BTECH11040
3. SriVatsav Goud A CS20BTECH11003

Problem Statement

The main goal of this project is to design, implement, and evaluate parallelized versions of MST algorithms to address the increasing need for high-performance solutions.

Minimum Spanning Tree

An Minimum Spanning Tree(MST) is a subset of the **edges** of the connected, weighted graph(G = (V, E) that connects all the vertices without any cycles and with the **minimum** possible **total edge weight**(sum of weights of all edges)**.**

The most Widely used MST algorithms are 1. Kruskal's algorithm, 2. Boruvaka's algorithm and 3. Prim's algorithm.

This report contains the implementation details, statistics, and conclusions of the parallelized versions of the above algorithms.

The following datasets are used to collect statistics :

1. Dataset 1 (1858 nodes and 28236 edges)
2. Dataset 2 (16726 nodes and 95188 edges)

Reference:
https://www.researchgate.net/publication/235994104_Parallel_Implementation_of_Minimum_Spanning_Tree_Algorithms_Using_MPI

The algorithms in the paper are implemented using MPI, whereas we implemented it using goroutines, waitgroup and mutex.

Parallel Kruskal's algorithm

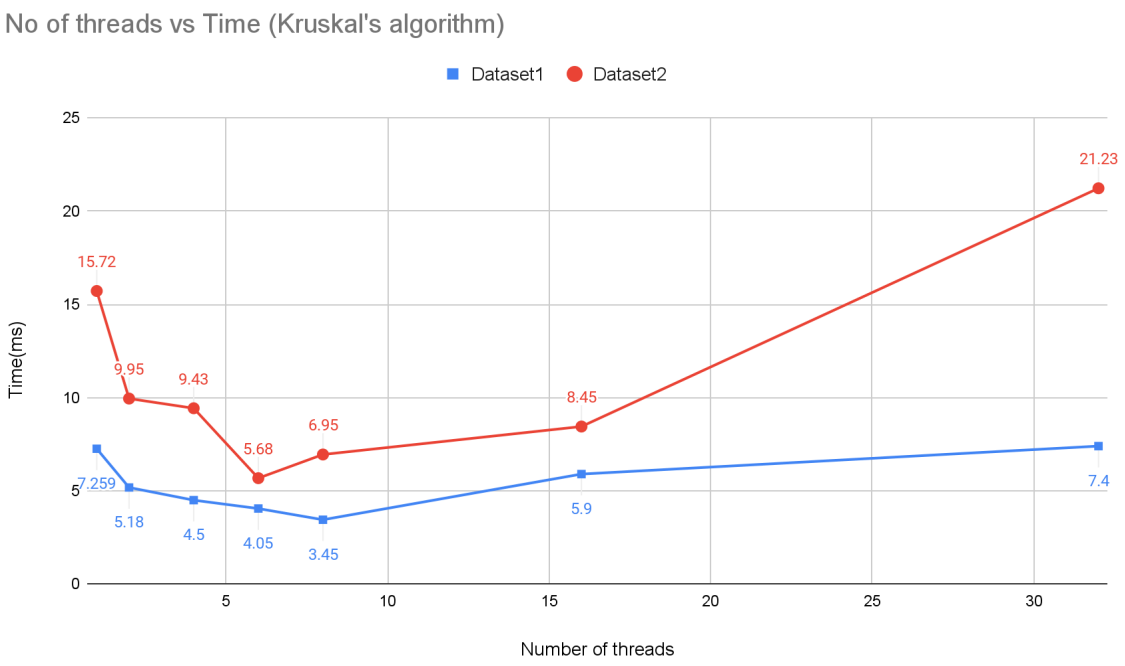
Design and working of algorithm

The comprehensive algorithm consists of three fundamental stages:

1. **Data Partitioning:**
    - The initial step involves dividing the set of edges into "nthreads" partitions, each designated for processing by an individual thread.
2. **Thread-Level Processing:**
    - Within each thread:
        - Sort the edges within its assigned partition in ascending order based on edge weights.
        - Employ the UnionFind data structure to verify whether the endpoints of an edge belong to the same component.
        - If the endpoints are not part of the same component, include the edge in the Minimum Spanning Tree (MST) and execute the union operation.
        - Accumulate the edges present in the MST.
3. **Iterative Refinement:**
    - Reduce the number of threads ("nthreads") by a factor of 2.
    - Iterate through steps 1 and 2 until a complete MST is obtained.

This iterative process of partitioning, processing, and refinement facilitates the construction of a final MST with optimal edge selections.

Graph



Conclusions:

- The algorithm's performance follows an increasing trend from 1 thread to 8 threads and from there it decreases. 8 is the number of cores in our machine. So the algorithm can execute 8 threads efficiently.
- This parallel algorithm splits the complete edge list into smaller parts. If more threads are used, this will result in a lesser number of eliminations at each step and thus increase redundancy.
- Since dataset 2 has more connectivity among edges compared to dataset 1, it takes much more time to compute MST.

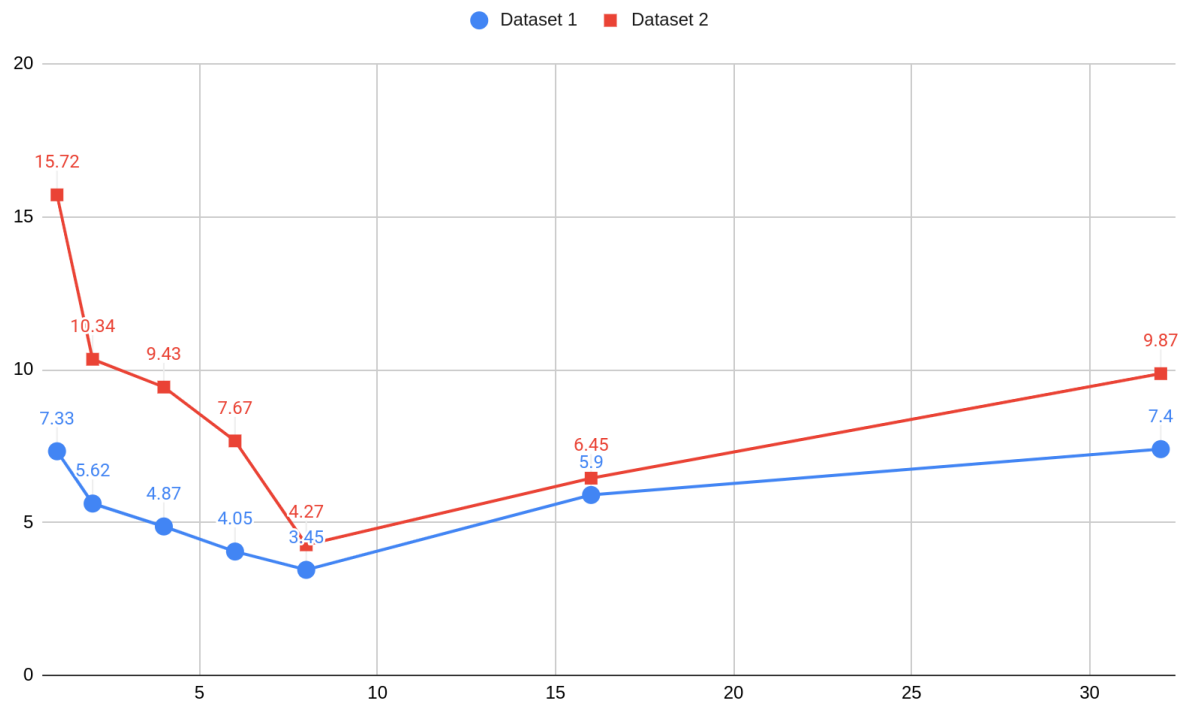
Parallel Boruvaka's algorithm

Design and working of algorithm

1. **Initialization:**
    - Set up a boruvakaUtil structure containing Union-Find structures (ufs) and the total number of vertices (nvertices).
2. **Partitioning:**
    - Divide the given edge list into multiple partitions, each assigned to a specific thread (tid).

3. **Local Processing (boruvakaLocal):**
    - For each thread:
        - Create a local Union-Find structure (uf) and reset it for a fresh iteration.
        - Initialize an empty list (mst) to store the Minimum Spanning Tree (MST) for the current partition.
        - Create a matrix (cheapest) to store the cheapest edge for each component.
        - Determine the total number of unique vertices (numComponents) in the partition.
        - Repeat until only one component remains:
            - For each edge in the partition:
                - Identify the endpoints (u and v) and their respective components (up and vp).
                - Update the cheapest edge if it forms a better connection between components.
            - Add the selected edge to the MST, perform a Union operation, and reduce the component count.
4. **Parallel Execution (Boruvaka):**
    - Determine the initial number of threads (ntasks) based on the total available threads.
    - While the number of tasks is greater than zero:
        - Spawn threads to concurrently execute the boruvakaLocal function.
        - During updates, use a mutex (elistMutex) to protect the global MST list (mstlist).
        - Wait for all threads to complete and update the global MST list.
        - Halve the number of tasks for the next iteration.
5. **Completion:**
    - The final MST is represented as a list of edges and is obtained after the algorithm converges.

Graph



Conclusion

- The algorithm's performance follows an increasing trend from 1 thread to 8 threads and from there it decreases. 8 is the number of cores in our machine. So the algorithm can execute 8 threads efficiently.
- The key components that can be parallelized include the examination of edges, identification of minimum-weight edges for different components, updating component information, and concurrent processing of multiple components.
- Consideration of multiple stages in the algorithm and exploration of parallelization strategies contribute to the optimization of Boruvka's algorithm for parallel computing environments.
- Since dataset 2 has more connectivity among edges compared to dataset 1, it takes much more time to compute MST.

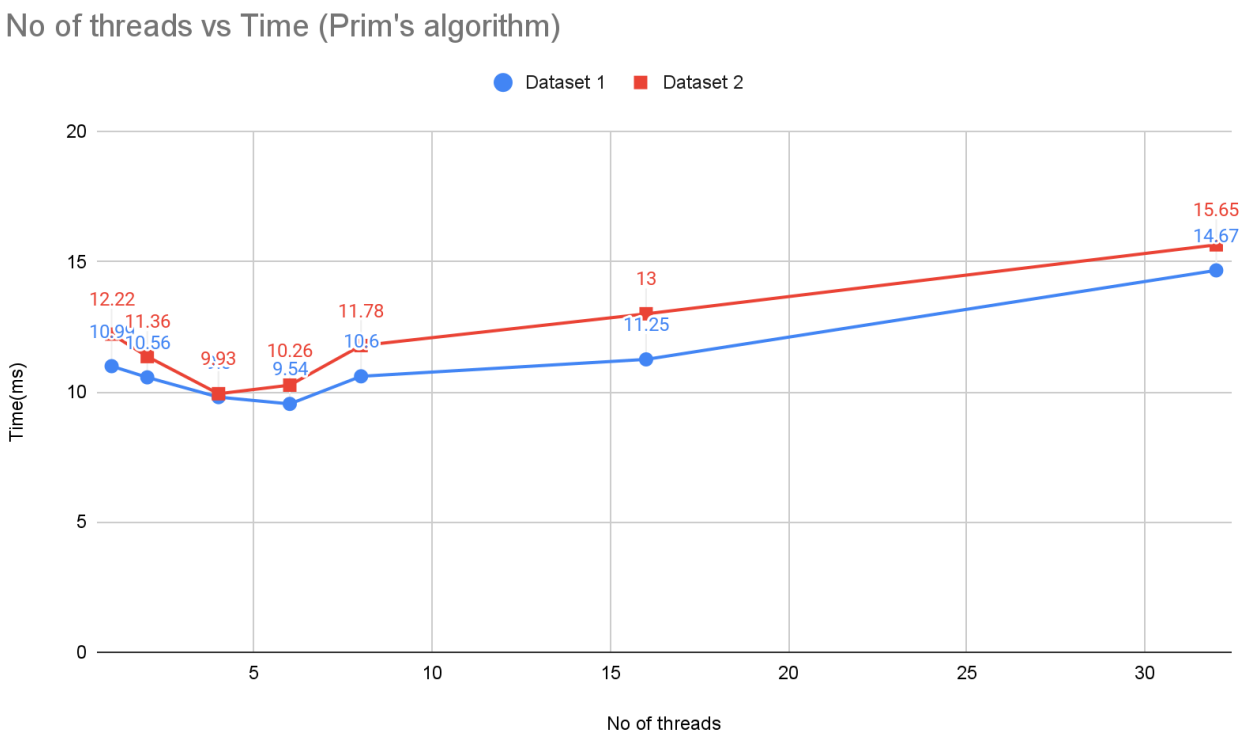
Parallel Prim’s algorithm

Design and working of algorithm

The code consists of the following main sections:

- **Function Declarations:**
  - **minKey:** Finds the vertex with the minimum key value among the vertices not yet included in the MST.
  - **printMST:** Prints the edges and their weights in the Minimum Spanning Tree.
  - **primMST:** Implements the main Prim's algorithm logic using parallelization.
- **Main Function**:
  - Dynamically allocates a 2D array (graph) to represent the weighted adjacency matrix of the graph. Generates a random adjacency matrix with weights in the range [0, 9].
  - Ensures the diagonal elements are set to zero (no self-loops). Copies the upper triangular elements to the lower triangular part to ensure symmetry.
- **Parallelization:**
  - OpenMP directives are used to parallelize the minKey and inner loop of primMST. The num variable is used to store the number of threads created during parallel execution.
- **Timing:**
  - The program measures the execution time of the primMST function using omp_get_wtime().
- **Memory Cleanup:**
  - Proper memory deallocation is performed using delete[] to prevent memory leaks.

Graph



Conclusions

- Parallelizing Prim's algorithm poses inherent challenges due to its sequential nature and global decision-making requirements. The algorithm's dependency on global information, incremental construction, and dynamic nature make effective parallelization complex. Data dependencies, limited opportunities for parallelism, and potential overheads further contribute to the difficulty. While partial parallelization is possible, achieving significant speedup requires careful consideration of synchronization, data races, and algorithmic structure. Exploring alternative parallel MST algorithms tailored for concurrency may be more effective in parallel computing environments.
- Since dataset 2 has more connectivity among edges compared to dataset 1, it takes much more time to compute MST.