Parallel Kruskal's Algorithm

High Performance Computing for Data Science Project 2024/2025

1st Alberto Messa
University of Trento
38123 Povo TN, Italy

2nd Enrico Tenuti
University of Trento
38123 Povo TN, Italy

***Abstract*—Minimum Spanning Trees (MST) are a fundamental concept in graph theory with applications in network design, clustering, and optimization. This report presents a parallel implementation of Kruskal's algorithm for finding the MST of a graph, designed to handle large datasets efficiently using parallel processing. We discuss the theoretical foundations of Kruskal's algorithm, the challenges of parallelization, and the performance improvements achieved. We propose two parallel solutions using the MPI API [1], focusing on the parallelization of the edge sorting step and making use of the Disjoint Set Union (DSU) data structure for efficient cycle detection. Experimental results demonstrate a significant speedup and scalability, giving the non-parallelizable nature of the Kruskal's algorithm. The parallel implementation is evaluated on various graph sizes and densities, showcasing its effectiveness in reducing computation time. We also discuss the limitations of our approach and potential future work to further enhance the performance of parallel MST algorithms. Our findings contribute to the understanding of parallel algorithms in graph theory and their practical applications in data science and high-performance computing.**

The complete source code is available at: https://github.com/MessaAlberto/HPC_Kruskal_Parallelization.git

I. Introduction

Finding the minimum spanning tree (MST) of a connected, undirected, weighted graph is a fundamental problem in computer science with applications in network design, clustering, and optimization. Kruskal's algorithm [2] is a widely taught greedy approach that solves the MST problem by sorting edges in non-decreasing order of their weights and adding them to the MST while avoiding cycles. The algorithm is efficient, especially when implemented with a union-find data structure to manage connected components.

However, the traditional sequential implementation of Kruskal's algorithm can be a bottleneck for large graphs. This report presents a parallel implementation that leverages modern parallel processing techniques to improve efficiency and scalability. By distributing the workload across multiple processors, we aim to reduce computation time and enhance performance on large-scale datasets.

In the realm of parallel MST algorithms, several approaches have been proposed to improve Kruskal's algorithm. These include parallelizing the edge sorting step, processing edges concurrently, and optimizing union-find operations. Additionally, alternative algorithms like Boruvka's algorithm, which repeatedly merges components using minimum-weight edges, offer different parallelization opportunities. While Boruvka's algorithm is effective in certain scenarios, this report focuses on parallelizing Kruskal's algorithm due to its widespread use and well-established theoretical foundation. The report is structured as follows:

- **Section II**: This section provides a comprehensive overview of related works in the field, discussing the theoretical background of Kruskal's algorithm and emphasizing the importance and implementation details of the union-find data structure, which is crucial for the algorithm's efficiency. With that, other Kruskal alternatives are breifly listed, in order to understand the state of the art of MST algorithms.
- **Section III**: Here, we describe our methodological approach to parallelizing Kruskal's algorithm. The section outlines the main building blocks of our code structure and explains in detail how we transitioned from a traditional sequential implementation to an efficient parallel version.
- **Section IV**: This section delves into the technical aspects of our implementation, offering code references and an in-depth comparison between two different parallelization strategies that were considered and evaluated.
- **Section V**: In this section, we present the experimental results obtained from our tests, highlighting the performance improvements and scalability benefits achieved through parallelization, supported by relevant data and analysis.
- **Section VIII**: The report concludes with a summary of our findings and a discussion of potential directions for future research and further enhancements to the parallel Kruskal's algorithm.

II. Related Works

A. Different Algorithms for MST

Kruskal's algorithm is one of the classical greedy approaches for computing the Minimum Spanning Tree (MST). It operates by first sorting the edges by weight and then adding them to the MST if they do not form a cycle. Two other well-established algorithms are Prim's and Borůvka's,

each with distinct properties that affect their suitability for parallelization.

Prim's algorithm [3] builds the MST incrementally by expanding from a starting node, always selecting the minimum-weight edge that connects the current tree to an unvisited node. Its reliance on a global priority queue and dynamic graph traversal makes it inherently sequential and less practical for parallelization.

Borůvka's algorithm [4], on the other hand, offers a more parallel-friendly structure. At each iteration, it selects the lightest outgoing edge from each connected component in parallel and merges components accordingly. Since the operations on different components are independent, the algorithm scales well in parallel environments and reduce the number of components exponentially over iterations.

B. Challenges in paralleling Kruskal's Algorithm

Kruskal's algorithm poses three main challenges for parallelization. First, the *edge sorting step*, with $O(E \log E)$ complexity, is a global operation that must be completed before processing can begin, limiting concurrency. Some approaches address these challenges by parallelizing only selected steps. For example, Katsigiannis et al. [5] propose a method where the main thread processes edges sequentially, while helper threads discard cycle-forming edges in parallel, reducing synchronization overhead. Second, the *disjoint set union (DSU)* structure requires global consistency: concurrent `find` and `union` operations across threads or processes can lead to incorrect results if not properly synchronized. Third, *concurrent access* to shared structures introduces race conditions and synchronization overhead. Alternatives like local MST construction and merging mitigate some issues but require additional coordination and reapplication of DSU logic.

C. Union-Find Data Structure

The DSU is a critical component in Kruskal's algorithm, as it efficiently manages the connected components of the graph during the MST construction process. Its primary functionalities include determining the root of a set (using the `Find` operation) and merging two sets (using the `Union` operation). In the context of Kruskal's algorithm, the DSU is used to check whether adding an edge to the MST would form a cycle. This is achieved by verifying if the two vertices of the edge belong to the same set. If they do, the edge is skipped; otherwise, the sets are merged, and the edge is added to the MST. The DSU's efficiency, particularly with the optimizations of path compression and union by rank, ensures that these operations are performed in nearly constant time, even for large graphs. This makes the DSU indispensable for maintaining the algorithm's overall efficiency and scalability, especially when dealing with dense graphs or parallel implementations where multiple edges are processed concurrently.

1) Core Concepts: The DSU maintains two key arrays:

Algorithm 1 Disjoint Set Union (DSU)

Initialize:
for each vertex v **do**
    $parent[v] \leftarrow v$
    $rank[v] \leftarrow 0$
end for
procedure FIND(v)
    if $parent[v] \neq v$ **then**
        $parent[v] \leftarrow$ FIND($parent[v]$)
    end if
    return $parent[v]$
end procedure
procedure UNION(u, v)
    $root_u \leftarrow$ FIND(u)
    $root_v \leftarrow$ FIND(v)
    if $root_u = root_v$ **then**
        return
    end if
    if $rank[root_u] < rank[root_v]$ **then**
        $parent[root_u] \leftarrow root_v$
    else if $rank[root_u] > rank[root_v]$ **then**
        $parent[root_v] \leftarrow root_u$
    else
        $parent[root_v] \leftarrow root_u$
        $rank[root_u] \leftarrow rank[root_u] + 1$
    end if
end procedure

- **Parent Array**: This array keeps track of the parent of each element. Initially, each element is its own parent, representing individual sets.
- **Rank Array**: This array stores the rank (or depth) of the trees representing the sets. It is used to optimize the union operation by ensuring that smaller trees are always attached under larger trees.

2) Optimizations: Two critical optimizations make DSU highly efficient:

- **Path Compression**: During the `Find` operation, the path from a node to the root is flattened by directly connecting each node to the root. This reduces the depth of the tree, making future queries faster.
- **Union by Rank**: During the `Union` operation, the root of the smaller tree is attached to the root of the larger tree. If both trees have the same rank, one root is arbitrarily chosen, and its rank is incremented.

3) Time Complexity: The DSU operations are extremely efficient due to the combined effect of path compression and union by rank. The amortized time complexity for both `Find` and `Union` operations is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. For all practical purposes, $\alpha(n)$ grows so slowly that it can be considered a constant for any reasonable input size.

4) Applications: The DSU is widely used in various applications, including:

- **Graph Algorithms**: Kruskal's algorithm for MST, cycle detection, and connected components.
- **Dynamic Connectivity**: Maintaining connectivity information in dynamic graphs.
- **Clustering**: Grouping elements into clusters based on connectivity or similarity.

The DSU's efficiency and versatility make it a fundamental data structure in computer science, particularly in graph theory and algorithm design.

III. Methodology and Implementation

It is important to take in consideration that at the beginning of the computation, the graph is generated and entirely held by `rank 0`. This process is responsible for distributing the edge set to the remaining processes according to the chosen strategy. The graph is not replicated globally; instead, only `rank 0` initially has access to the complete edge list.

A. Sequential Kruskal's Algorithm

The sequential version of Kruskal's algorithm can be described by the following pseudocode:

Algorithm 2 Sequential Kruskal's Algorithm

Input: $G = (V, E)$, $|V| = n$, $|E| = m$
Output: MST $\subseteq E$
Sort E such that $\forall e_i, e_j \in E, w(e_i) \leq w(e_j)$
Initialize DSU: $\forall v \in V, \text{parent}[v] \leftarrow v, \text{rank}[v] \leftarrow 0$
MST $\leftarrow \emptyset$
for $e = (u, v) \in E$ (in sorted order) **do**
    if Find$(u) \neq$ Find(v) **then**
        MST $\leftarrow$ MST $\cup \{e\}$
        Union(u, v)
    end if
end for
return MST

The algorithm begins by sorting the edges in non-decreasing order of their weights. It then initializes the DSU structure, where each vertex is its own parent. The algorithm iterates through the sorted edges, checking if adding an edge would form a cycle using the `Find` operation. If it does not form a cycle, the edge is added to the MST, and the sets are merged using the `Union` operation. The time complexity of the sequential Kruskal's algorithm is dominated by the edge sorting step, which is $O(m \log m)$, where m is the number of edges. The union-find operations are nearly constant time due to the optimizations discussed earlier.

B. Naive Parallelization Attempt and Design Motivation

A straightforward approach to parallelizing Kruskal's algorithm would consist of equally dividing the edge set among the processes, letting each process compute a local MST independently (including sorting and DSU operations), and then performing a series of *tree-based reductions*. At each reduction level, two local MSTs would be merged, sorted again, and processed with Kruskal's algorithm to compute a new MST. This process would repeat until only one global MST remains.

Although this strategy is conceptually simple, it introduces significant inefficiencies. Each merge step would require re-sorting the combined edge lists and reapplying union-find operations, incurring unnecessary computation and communication overhead. Moreover, repeatedly merging partial MSTs disrupts the sorted structure of the edges, negating the benefit of initial sorting.

To overcome these limitations, we designed a strategy where each process operates on a subset of edges that are globally ordered by weight. Specifically, processes with lower ranks are assigned lighter edges than those with higher ranks. This guarantees that any MST edge computed by a lower-ranked process is lighter than any edge held by a higher-ranked process. As a result, during merging, the receiving process can safely discard most of the incoming edges without needing to re-sort or reapply Kruskal's algorithm entirely. This approach significantly reduces redundancy and ensures that the merging step consists only of adding the few remaining non-cyclic edges using the DSU structure.

C. Parallel Sorting Strategies

We've implemented two parallel strategies for Kruskal's algorithm; in both strategies, the key idea is to distribute edges across processes *in a globally ordered way*, so that each process handles a subset of edges lighter than those managed by higher-ranked processes. Each process then independently computes a local MST using Kruskal's algorithm on its assigned edges. During the merging phase, each process receives the MST from the next higher-ranked process and integrates only the missing edges using union-find operations—without any need for further sorting.

This design achieves three important goals:

- **No re-sorting is required** after merging.
- **Global edge order is preserved** through careful distribution.
- **Efficient merging** avoids redundant comparisons and edge duplication.

Since sorting the edge set is the most performance-critical part of the parallel Kruskal implementation, we focus on two distinct parallel sorting strategies. Both aim to produce a global ordering of edges across processes to enable efficient local MST computations and merging without re-sorting.

D. Strategy 1: `pivotSort` – Parallel Quicksort-Based Distribution

This approach assumes a near-uniform distribution of edge weights, which is reasonable for large random graphs. The steps are:

1) **Pivot computation:** `Rank 0` computes pivot values by dividing the global weight range into P intervals, where P is the number of processes.
2) **Edge assignment:** Each edge is assigned to a process based on the interval its weight falls into.

3) **Data distribution:** Each process receives its corresponding subset of edges from `rank 0`.
4) **Local computation:** Each process sorts its edges and runs Kruskal's algorithm independently to compute its local MST.

This strategy requires only one round of communication and has low overhead during edge distribution. However, the workload may become unbalanced if the edge weights are not uniformly distributed.

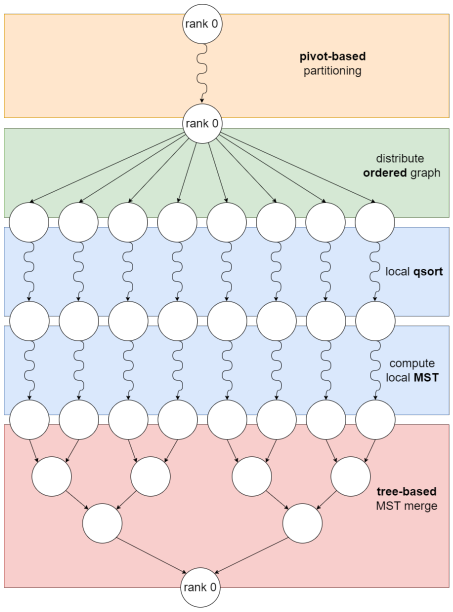

Fig. 1. Bottleneck of `pivotSort`: Centralized sorting workload on `rank 0`, which must reorder the entire edge set before distribution.

E. Strategy 2: `mergeSort` – Global Parallel Mergesort

This approach performs a fully parallel sort of the edge set before MST construction. The steps are:

1) **Initial distribution:** `Rank 0` divides the unsorted edge set and distributes it to all processes.
2) **Local sorting:** Each process sorts its subset of edges locally.
3) **Parallel merging:** A binary tree-like merging is performed across processes to obtain a fully sorted global edge list.
4) **Final redistribution:** The sorted list is then partitioned such that each process receives the lightest available edges according to its rank.
5) **Local MST computation:** Each process runs Kruskal's algorithm on its assigned subset.

While this approach involves more communication rounds, it offers better load balancing due to the initial uniform edge distribution.

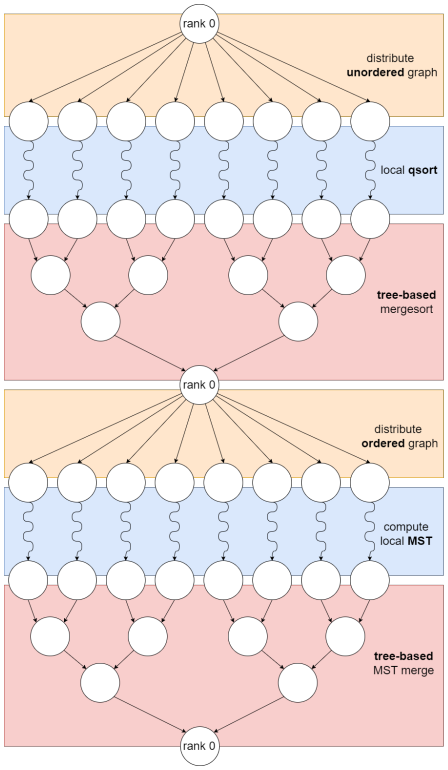

Fig. 2. Bottleneck of `mergeSort`: High communication overhead during parallel merging phases across all ranks.

F. Bottlenecks of the Two Strategies

The bottlenecks described in the following section are visually summarized in Figures 1 and 2, which illustrate the critical phases limiting scalability for each strategy.

The main bottleneck of `pivotSort` lies in the sorting step performed entirely by `rank 0`, which must reorder the entire edge set before distributing the subsets to other processes. This centralization can limit scalability and increase the computational load on a single process.

On the other hand, `mergeSort` distributes the sorting workload more evenly across processes but suffers from high communication overhead during the multiple parallel merging phases required to obtain a globally sorted edge list.

IV. Implementation Details

This section provides an in-depth explanation of the implementation of the parallel Kruskal's algorithm. We describe the key components of the implementation, focusing on the parallelization strategies, data structures, and communication patterns used to achieve efficient execution.

A. Graph Representation and Dimensionality Handling

The graph is represented as an array of edges, where each edge is a tuple (u, v, w), with u and v being the vertices connected by the edge and w being the weight of the edge. This representation is the minimal one and allows for efficient storage and processing of edges. The graph is built of nodes that have at least one edge, as the algorithm requires a

connected graph to find the MST and a disconnected graph would not yield a valid MST. In the initial implementation the graph was read from a file, which specified the number of vertices V and edges E, followed by the list of edges. For storage purposes, the final implementation builds a graph at the start of the program which is then used to find the MST and in the end it gets discarded. This choice is made for the fact that we had not enough storage space to store the graph so, in order to get a bigger graph, we had to keep it in RAM and never store the graph on disk. The weights of the edges are randomly generated between 1 and the maximum weight, which is a parameter given by the dimension of the graph and the number of vertices, the maximum weight is stored in a variable called `max_wgt`, and is computed as `INT32_MAX / (V - 1)`. This choice ensures that the sum of the edge weights in the worst-case scenario—i.e., a spanning tree with $(V-1)$ edges—does not exceed the maximum representable value of a 32-bit signed integer, thus preventing integer overflow. The graph generation is done by indicating the number of vertices and the density of the graph, which is the ratio between the number of edges and the maximum number of edges that can be present in a complete graph with V vertices.

```
typedef struct Edge {
    int u;
    int v;
    int weight;
} Edge;

MPI_Datatype MPI_EDGE;

void create_mpi_edge_type() {
  int block_lengths[3] = {1, 1, 1};
  MPI_Datatype types[3] = {MPI_INT, MPI_INT,
      MPI_INT};
  MPI_Aint offsets[3];

  offsets[0] = offsetof(Edge, u);
  offsets[1] = offsetof(Edge, v);
  offsets[2] = offsetof(Edge, weight);

  MPI_Type_create_struct(3, block_lengths,
      offsets, types, &MPI_EDGE);
  MPI_Type_commit(&MPI_EDGE);
}
```

B. Active Processes Handling

In our implementation, we handle the active processes dynamically. The number of active processes is determined at runtime and there is a cycle that increments the number of active processes until it reaches the maximum number of processes. In order to get all representative results, we use the MPI groups to create a new communicator that contains only the active processes. This allows us to run the algorithm with different numbers of processes, starting from 2 up to the maximum number of processes available. The rank is used to determine which edges are assigned to each process, ensuring that each process receives a unique subset of edges. This strategy prevents to use the MPI_COMM_WORLD communicator, which would include all processes, and instead creates a new communicator that can be used in the subsequent steps with the MPI_Scatterv and MPI_Gatherv functions and prevent to communicate with inactive processes.

```
for (int j = 2; j <= nproc; j *= 2) {
    // other code...

    MPI_Comm active_comm = MPI_COMM_NULL;
    MPI_Group world_group, active_group;
    MPI_Comm_group(MPI_COMM_WORLD, &
        world_group);

    if (rank < j) {
      // Create the active group
      int ranks[j];
      for (int k = 0; k < j; k++) ranks[k]
          = k;
      MPI_Group_incl(world_group, j, ranks
          , &active_group);
    } else {
      MPI_Group_excl(world_group, 1, &rank
          , &active_group);
    }

    MPI_Comm_create(MPI_COMM_WORLD,
        active_group, &active_comm);

    if (active_comm == MPI_COMM_NULL) {
      // Processes not in the active group
      continue;
    }
    // Perform the parallel Kruskal's
        algorithm...
}
```

C. Parallel Edge Sorting

As discussed previously, edge sorting is a fundamental step in Kruskal's algorithm. Our parallel implementation offers two sorting strategies—`pivotSort` and `mergeSort`—each designed to distribute the sorting workload efficiently while ensuring a globally ordered edge list.

1) Complexity of `pivotSort`: The `pivotSort` method begins with the root process (rank 0) computing pivot values to divide the edge list into P subsets, where P is the number of processes. This partitioning step has a linear complexity of $O(m)$, where m is the number of edges.

Each process then performs a local sort on its assigned subset. Assuming an even distribution, the sorting step has a per-process complexity of $O\left(\frac{m}{P}\log\frac{m}{P}\right)$. The strategy involves a one-to-many communication from the root process to all others, incurring an additional $O(m)$ communication cost, which is influenced by the underlying network's bandwidth and latency.

The dominant cost lies in the local sorting phase, yielding a total complexity of $O\left(\frac{m}{P}\log\frac{m}{P}\right)$ per process, plus the communication overhead.

2) Complexity of `mergeSort`: In `mergeSort`, edges are first evenly distributed among processes, which takes $O(m)$ communication time. Each process locally sorts its portion in $O\left(\frac{m}{P}\log\frac{m}{P}\right)$ time.

Following local sorting, a parallel hierarchical merging is performed. At each of the $\log P$ levels, sorted subsets are combined, each step costing $O(m)$. This results in a merging complexity of $O(m \log P)$.

Therefore, the overall complexity becomes:

$$O\left(\frac{m}{P} \log \frac{m}{P} + m \log P\right)$$

accounting for both sorting and merging stages.

3) Comparison of `pivotSort` and `mergeSort`: The two approaches diverge significantly in their merging mechanisms. `pivotSort` eliminates the need for hierarchical merging by relying on a well-balanced pivot partitioning and assumes uniform edge distribution. In contrast, `mergeSort` provides a globally sorted edge list at the cost of higher communication and merging overhead.

For graphs with uniformly distributed edge weights, `pivotSort` tends to be more efficient. However, in cases with skewed weight distributions, `mergeSort` achieves better load balancing.

D. Local MST Construction

Once edges are sorted and distributed, each process constructs its local minimum spanning tree (MST). This is achieved by iterating over the local edge list in $O\left(\frac{m}{P}\right)$ time and performing union-find operations, each with an amortized cost of $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function—negligible in practical terms.

Consequently, the complexity per process for local MST construction is:

$$O\left(\frac{m}{P} \cdot \alpha(n)\right)$$

Global MST merging follows. Each process contributes its local MST, requiring $O(P)$ communication steps. If each local MST contains approximately $\frac{n}{P}$ edges, merging them in a tree-based fashion entails:

$$O\left(\frac{n}{P} \cdot \alpha(n)\right)$$

per level, over $\log P$ levels.

Combining both local construction and global merging, the full parallel complexity of Kruskal's algorithm becomes:

$$O\left(\frac{m}{P} \cdot \alpha(n) + \frac{n}{P} \cdot \alpha(n) \cdot \log P\right)$$

This formulation underscores the algorithm's scalability, with communication overhead increasing logarithmically with the number of processes.

E. Communication Techniques for Large Datasets

Handling `INT_MAX` Limitations in MPI. MPI communication functions have certain limitations when handling large datasets. While basic functions such as `MPI_Send` and `MPI_Recv` support a variety of datatypes—including user-defined structures like the `MPI_EDGE` type used in our implementation—they are still constrained by the number of elements that can be communicated in a single call. This becomes particularly problematic when using collective communication functions such as `MPI_Scatter`, `MPI_Gather`.

The function `MPI_Scatterv`, for instance, allows sending variable-sized chunks of data to each process, controlled via support arrays for counts and displacements. However, these functions internally rely on 32-bit integers to represent the count of elements per process. As such, they are limited to sending no more than `INT_MAX` elements (2,147,483,647). This poses a problem in our context when distributing large graphs with more than `INT_MAX` edges.

To address this, we adopted different strategies depending on the communication scheme used, as described in the following subsections.

Chunked `MPI_Scatterv` in MergeSort. As described earlier, in the MergeSort-based approach the unordered graph is initially distributed to the processes, locally sorted, then merged into a globally sorted structure, and finally redistributed. This second distribution step is critical for the correctness of the algorithm, as it depends on the global order of the edges.

When the graph size is below the `INT_MAX` limit, the redistribution is straightforward. Once the number of elements per process has been computed:

```
int64_t base_size = E / nproc;
int64_t extra = E % nproc;
int64_t offset = 0;

for (int i = 0; i < nproc; i++) {
  send_counts[i] = base_size + (int64_t)(i <
      extra ? 1 : 0);
  displs[i] = offset;
  offset += send_counts[i];
}
```

The redistribution can then be performed with a single call to `MPI_Scatterv`:

```
MPI_Scatterv(graph, send_counts_int,
    displs_int, MPI_EDGE, *local_graph,
              send_counts_int[rank], MPI_EDGE,
                  0, active_comm);
```

When the total number of edges exceeds `INT_MAX`, the 32-bit integer constraint of `MPI_Scatterv` makes a single distribution infeasible. To address this, the edges are distributed in multiple *batches*, each fitting within the per-call integer limit (see Figure 3).

The overall process works as follows:

1) The root process first computes the number of edges each process should receive, based on a balanced partitioning of the total graph.
2) A secondary index is built to segment the graph per-process in a way compatible with batching: each process will receive its assigned edges in multiple smaller pieces.
3) The root constructs a reordered copy of the graph, organized as a sequence of batches, each containing at most `INT_MAX` elements in total.
4) For each batch:
   - A local `MPI_Scatterv` is performed to send a slice of the graph to each process. Counts and

displacements are computed per batch and cast to 32-bit integers.
- Each process appends the received chunk to its local buffer.

Pseudocode sketch (rank 0):

Algorithm 3 Chunked distribution of edges when $E >$ `INT_MAX`

```
if rank = 0 then
    Compute number of edges for each process: send_counts
    Compute displacements: displs
    Compute per-process upper bounds: batch_index
    Initialize ordered_graph
    for each batch do
        for each process do
            Copy a segment of graph to ordered_graph ▷ respect per-process batch size
        end for
    end for
end if
for each batch do
    Compute 32-bit send_counts_int and displs_int
    MPI_SCATTERV(current batch of ordered_graph)
    Append received edges to local_graph
end for
```

This scheme guarantees that:
- No call to `MPI_Scatterv` exceeds the 32-bit count limitation.
- Each process receives exactly its assigned edges across batches.
- Communication is orderly and scalable even for very large graphs.

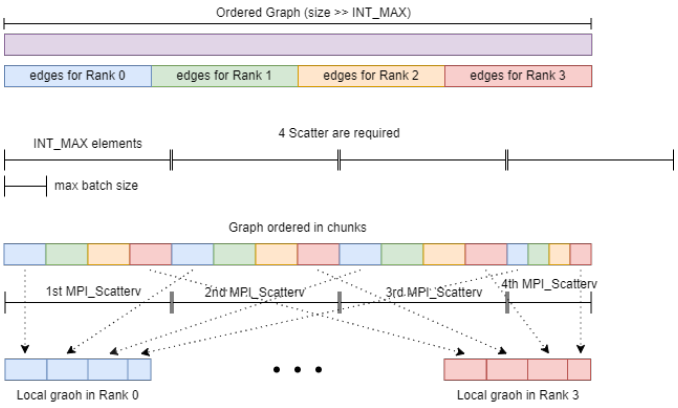

Fig. 3. Illustration of the chunked `MPI_Scatterv` process when the number of edges E exceeds `INT_MAX`. Each color represents a batch sent across processes.

Multiple `MPI_Send/Recv` in PivotSort In the PivotSort version, the distribution of the graph—partially sorted based on pivot values—is handled exclusively through `MPI_Send` and `MPI_Recv` communications between the root process and the others. Only the process with rank 0 has access to the full graph, and it performs the pivot-based sorting alone. This is a partial sort with a time complexity of $O(E)$, which is faster than a complete sort. However, it requires more memory, as the root must temporarily store the subset of edges destined for each process in separate buffers. Once the sorting is done, the root process iterates over each recipient process: it first sends the number of edges to be received, then both sender and receiver enter a loop to send and receive the data in chunks of at most `INT_MAX` elements per `MPI_Send`. Using `MPI_Scatterv` was not suitable in this context, as it would have required first copying the entire pivot-sorted graph into a contiguous buffer, then broadcasting the edge counts to each process, and finally calling the scatter function. This approach is more complex and was empirically shown to be slightly slower than the simpler `MPI_Send/Recv` strategy.

V. Results

A. Hardware

All experiments were conducted on the High-Performance Computing (HPC) cluster maintained by the University of Trento [6]. The system is designed for large-scale scientific workloads and is managed via PBS Professional (version 19.1), a job scheduling system optimized for parallel execution.

The cluster configuration at the time of testing included:
- **142 CPU nodes**, providing a total of **7,674 CPU cores**.
- **10 GPU nodes**, equipped with NVIDIA hardware for a combined total of **48,128 CUDA cores**.
- **2 frontend (head) nodes** for user interaction and job submission.
- **65 TB of RAM**, distributed across the nodes.
- **Theoretical peak performance**: 478.1 TFLOPs overall, with 422.7 TFLOPs from CPUs and 55.4 TFLOPs from GPUs.

The nodes are interconnected using a **10 Gb/s Ethernet network**. Some nodes additionally support **Infiniband** or **Omni-Path** technologies to ensure low-latency, high-bandwidth communication, particularly beneficial for MPI-based parallel applications.

User home directories and software environments are hosted on a shared storage system, which is mirrored to a backup site for redundancy. All compute nodes run **Linux CentOS 7** to maintain consistency across the cluster.

B. Job Submission and Resource Constraints

The experiments were submitted to the HPC cluster using the following PBS job script directives:

```
#!/bin/bash

#PBS -l select=8:ncpus=64:mem=128gb
#PBS -q short_cpuQ
#PBS -l walltime=06:00:00
```

```
module load mpich-3.2

DIR=${PBS_O_WORKDIR}

mpiexec -n 512 "${DIR}/bin/kruskal" $V
```

In this configuration, the job requests 8 nodes with 64 CPU cores and 128 GB of RAM each, totalling 512 CPU cores. The job is submitted to the `short_cpuQ` queue with a maximum walltime of 6 hours.

Considering the HPC cluster's network infrastructure, the inter-node bandwidth (10 Gb/s Ethernet, with some nodes having Infiniband or Omni-Path) plays a critical role in parallel scalability. Since the requested resources exceed the capacity of a single node (which can have up to 96 CPU cores and 1 TB of RAM), the workload is distributed across multiple nodes. This distribution inevitably involves network communication overhead, which can become a bottleneck, especially for data-intensive operations requiring frequent synchronization or data exchange.

Moreover, the dataset size used in the experiments approaches the upper limit of what can be handled on the cluster. In particular, the root process (rank 0) stores multiple copies of the graph during the pivot sorting phase — up to three times the graph size — which demands significant memory. This factor constrained the maximum graph size used, as exceeding 1 TB of memory on a single node would be unfeasible.

These resource and network considerations must be taken into account when interpreting the performance results and scalability behavior of the implemented parallel algorithm.

C. MPI Parallelization Results

In the table below, we can see the comparison in average execution time of the sequential and parallel implementations of Kruskal's algorithm. The results we are going to provide are the one obtained with the `pivotSort` strategy, which is the one that performed better in our tests. The table presents on rows the number of processes used and on columns the number of edges in the graph. We are going to consider 100% density graphs, or complete graphs, which are the ones that take the longest time to compute the MST.

nproc	1024	2048	4096	8192	16384	32768	65536
1	0.15	0.71	2.66	11.4	53.75	201.19	854.57
2	0.09	0.40	1.55	6.69	30.70	115.45	482.40
4	0.05	0.23	0.89	3.90	17.56	65.21	281.69
8	0.04	0.14	0.59	2.50	11.11	41.70	178.18
16	0.03	0.10	0.45	1.86	8.10	30.53	131.30
32	0.02	0.09	0.38	1.56	6.64	25.55	106.47
64	0.02	0.07	0.41	1.46	6.41	22.76	96.80

TABLE I
EXECUTION TIME (IN SECONDS) FOR DIFFERENT NUMBERS OF PROCESSES AND GRAPH SIZES (IN NODES)

The table shows the execution time in seconds for different numbers of processes and graph sizes. The results indicate that the parallel implementation scales quite well with the number of processes, as the execution time decreases significantly when using more processes. The total time increases quadratically with the number of nodes, as expected, since the number of edges in a complete graph is $E = \frac{V(V-1)}{2}$, where V is the number of nodes.

D. Algorithm Evaluations

To assess the effectiveness of our parallel implementation, we rely on two standard performance indicators commonly used in parallel computing: *speedup* and *parallel efficiency*.

- **Speedup (S)** measures the ratio between the execution time of the best serial implementation (T_1) and the execution time using p parallel processes (T_p). It is defined as:
$$S(p) = \frac{T_1}{T_p}$$
A speedup close to the number of processes indicates that the parallel algorithm scales well and makes effective use of the available resources.
- **Efficiency (E)** indicates how well the computational resources are utilized during parallel execution. It is computed as the speedup divided by the number of processes:
$$E(p) = \frac{S(p)}{p} = \frac{T_1}{p \cdot T_p}$$
Efficiency values close to 1 imply minimal overhead and good scalability, while lower values highlight suboptimal performance, potentially due to communication costs or workload imbalance.

These two metrics provide a clear picture of the algorithm's scalability and help identify performance bottlenecks as the number of processes increases.

E. Speedup and Efficiency Results

The speedup graph (Figure 4) illustrates the speedup achieved by our parallel implementation of Kruskal's algorithm across different numbers of processes and graph sizes. The efficiency graph shows how well the computational resources are utilized during the execution.

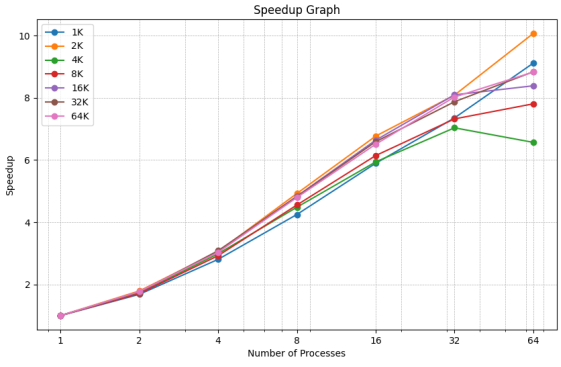

Fig. 4. Speedup Results for Parallel `pivotSort` Kruskal's Algorithm

We can see some little inconsistencies in the speedup and efficiency values, especially for larger graphs. This is likely due to the increased communication overhead as the number of processes increases, which can lead to diminishing returns in performance. However, overall, the results indicate that our parallel implementation scales well with the number of processes, achieving significant speedup and maintaining reasonable efficiency. The cluster load and the network bandwidth also play a crucial role in the performance of the parallel implementation, as they can affect the communication overhead and the overall execution time.

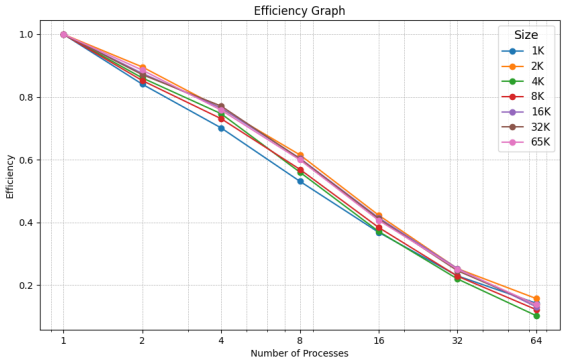

Fig. 5. Efficiency Results for Parallel `pivotSort` Kruskal's Algorithm

The efficiency graph shows that the efficiency decreases almost linearly as the number of processes increases: this is expected, as the bottleneck of the algorithm is the sorting step, which cannot be parallelized perfectly. The communication overhead also increases with the number of processes, which contributes to the decrease in efficiency.

VI. Density and Graph Size Impact

The execution time increases, as expected, with the number of edges, as the algorithm has to process more edges and perform more union-find operations. However, the parallel implementation scales well with the number of processes, achieving significant speedup and maintaining reasonable efficiency.

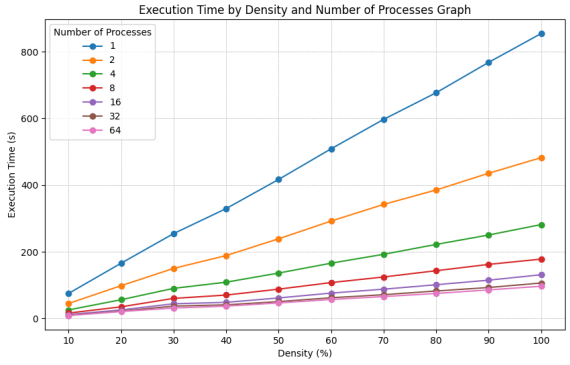

Fig. 6. Execution Time for Different Graph Densities

VII. Time comparison between `pivotSort` and `mergeSort`

From the graph in Figure 7, we can see that the `pivotSort` strategy is slightly faster than the `mergeSort` strategy for all numbers of processes, and both algorithms share a similar trend in execution time. The `pivotSort` strategy is more efficient because it performs a partial sort of the edges based on pivot values, which reduces the amount of data that needs to be communicated between processes. The `mergeSort` strategy, on the other hand, requires a complete sort of the edges and a redistribution, which increases the communication overhead and execution time.

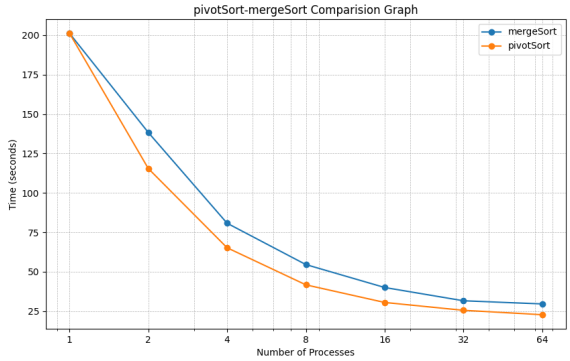

Fig. 7. Execution Time Comparison between `pivotSort` and `mergeSort` Strategies

VIII. Conclusion

The development of a parallel implementation of Kruskal's algorithm proved to be particularly challenging, especially in overcoming the sorting bottleneck that limits scalability. Despite these difficulties, the algorithmic strategy we designed is innovative and effective in addressing the key challenges of parallel MST computation.

Experimental results show that the parallel implementation achieves significant speedup with a nearly linear decrease in efficiency as the number of processes grows. This behavior is consistent with the inherent bottlenecks in sorting and communication overhead. Nevertheless, the efficiency remains relatively high, demonstrating good scalability and effective resource utilization, especially for large and dense graphs.

The pivotSort strategy proved more efficient than mergeSort, thanks to reduced communication and sorting demands, highlighting practical benefits for distributed MST computations.

Overall, this work confirms that Kruskal's algorithm can be effectively parallelized using MPI, offering meaningful performance improvements and identifying the key challenges for further scalability enhancements.

The complete source code is available at: https://github.com/MessaAlberto/HPC_Kruskal_Parallelization.git

IX. References

[1] Open MPI Community, "Open MPI: Open Source High Performance Computing," https://www.open-mpi.org/doc/, accessed May 27, 2025.

[2] Wikipedia contributors, "Kruskal's algorithm," *Wikipedia, The Free Encyclopedia*, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kruskal%27s_algorithm&oldid=1290906756.

[3] GeeksforGeeks, "Prim's Minimum Spanning Tree (MST) – Greedy Algo 5," Available: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/.

[4] GeeksforGeeks, "Borůvka's Algorithm – Greedy Algo 9," Available: https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/.

[5] A. Katsigiannis, N. Anastopoulos, K. Nikas and N. Koziris, "An Approach to Parallelize Kruskal's Algorithm Using Helper Threads," in *Proc. IEEE Int. Parallel and Distributed Processing Symposium Workshops*, 2012, pp. 1601–1610. doi: 10.1109/IPDPSW.2012.201.

[6] University of Trento, "Cluster HPC Architecture," Internal knowledge base article, available at: https://servicedesk.unitn.it/sd/en/kb-article/cluster-hpc-architecture?id=unitrento_v2_kb_article&sysparm_article=KB0010434. (Requires UniTN authentication)