# Parallel Minimum Spanning Tree Using Borůvka's Algorithm

*High Performance Computing for Data Science Project 2024/2025*

**Lucas Fernández**
University of Trento
l.fernandezb@studenti.unitn.it

**Abstract — The Minimum Spanning Tree (MST) problem, fundamental to graph theory, involves connecting all vertices of a graph with the minimum total edge weight, a task pivotal in fields such as network design and optimization. Among classical MST algorithms, Borůvka's algorithm is distinguished for its iterative approach and potential for parallelization. This work investigates a distributed-memory parallel implementation of Borůvka's algorithm using the Message Passing Interface (MPI) and Open Specification for Multi-Processing (OpenMP) framework, emphasizing scalability and performance evaluation. Through analysis of synthetic graphs, the study demonstrates near-linear speed up on sparse datasets while addressing the communication overhead inherent in dense graphs. These results underscore the algorithm's effectiveness in leveraging distributed architectures, offering insights into optimizing MST computation for large-scale applications.**

## 1 Introduction

Graphs stand as pivotal structures for encapsulating the complexity of relationships among diverse entities, rendering them indispensable in fields such as network analysis, computational geometry, clustering, and optimization. One of the most intensively studied problems in graph theory is the determination of a Minimum Spanning Tree (MST), which aims to identify a subset of edges that connects all vertices with the minimum total edge weight. The MST problem's significance transcends theoretical discussions, finding practical applications in diverse arenas, including network design, data clustering, and the optimization of resource allocation processes (Chung & Condon, 1996; Chazelle, 2000). By ensuring minimality of edge weights over a connected set of vertices, MST solutions often serve as foundational building blocks for more complex algorithms and systems.

Within the broader family of MST algorithms, Borůvka's algorithm occupies a special place due to its historical importance and the elegant simplicity of its approach. First proposed in 1926 by Otakar Borůvka to address the electrification of Moravia, the algorithm operates via an iterative process that begins by regarding each vertex as an individual component. Subsequently, the lightest edge connecting each component to a different component is selected, thereby merging those components into a larger connected structure. This divide-and-conquer process continues until a single connected component encompasses all vertices (Sanders & Schimek, 2023). Notably, Borůvka's method can detect multiple lightest edges in a single iteration if different components share identical minimal weights, effectively orchestrating parallel edge selections that naturally streamline the merging process. Over the years, this straightforward iterative paradigm has attracted attention for its potential to be parallelized on modern computing infrastructures, where harnessing concurrency is paramount to achieving high performance.

Despite the clear conceptual alignment between Borůvka's iterative merging steps and parallel execution, implementing the algorithm efficiently in distributed-memory environments poses nontrivial challenges. When thousands—or even millions—of edges and vertices are distributed across multiple computing nodes, each node must coordinate with the others to identify the globally lightest edges for each component. This coordination process may incur substantial overhead due to the need for synchronization and communication among nodes, especially as the size of the graph scales (Chung & Condon, 1996; Sanders & Schimek, 2023). In addition, managing load imbalance—where certain nodes carry out significantly more work than others—can degrade overall performance. Sparse graphs, which intrinsically have fewer edges per vertex, typically mitigate some of these communication and synchronization concerns, whereas dense graphs, with their high ratio of edges to vertices, often exacerbate them (Sanders & Schimek, 2023). Consequently, careful consideration must be given to data distribution strategies, communication pro-

tocols, and load-balancing techniques to optimize performance.

Recent studies underscore the viability of using Message Passing Interface (MPI) and hybrid methods such as MPI with OpenMP to exploit multiple levels of parallelism. By dividing the graph across nodes and then further leveraging shared-memory parallelism within each node, researchers have achieved remarkable speed-ups when tackling large-scale MST instances. In many cases, Borůvka's algorithm, with its iterative structure, emerges as particularly adaptable to these layered parallel strategies because each stage of the merge operation is naturally amenable to concurrent processing across distinct components (Chazelle, 2000; Sanders & Schimek, 2023). Moreover, contemporary high-performance computing systems, equipped with increasing numbers of cores and larger memory capacities, provide a compelling environment to investigate optimizations that might be infeasible on smaller-scale machines.

Against this backdrop, the present work endeavours to implement and systematically evaluate a distributed-memory version of Borůvka's algorithm using MPI and OpenMP. Our goal is to provide an in-depth analysis of how well the algorithm scales, how communication overhead influences performance, and how effectively different load-balancing schemes address imbalances. By testing our implementation on a range of graph datasets—from sparse to dense and from synthetic benchmarks to real-world networks—we offer insights into the interplay among computational efficiency, communication costs, and the architecture-dependent characteristics that shape performance outcomes. Ultimately, our findings seek to contribute to the broader discourse on optimizing parallel graph algorithms, with an emphasis on striking a balance between minimal computation time and acceptable resource utilization, thereby pushing the boundaries of large-scale, real-world applications (Chung & Condon, 1996; Sanders & Schimek, 2023).

# 2 Related Works

Borůvka's algorithm, introduced in 1926, stands as one of the earliest approaches to solving the Minimum Spanning Tree (MST) problem. Owing to its iterative design, where each connected component selects its lightest incident edge independently, Borůvka's algorithm exhibits a natural affinity for parallel execution (Chung & Condon, 1996). This characteristic has positioned it at the forefront of distributed-memory and shared-memory parallel computing research. Over time, numerous enhancements and variants have been proposed to optimize its performance, scalability, and robustness across different architectures and use cases.

## 2.1 Distributed-Memory Implementations

One of the foundational works in this domain was presented by Chung and Condon (Chung & Condon, 1996), who "pioneered" a parallel Borůvka implementation suited for distributed-memory systems. Their approach tackled critical issues such as minimizing communication overhead and ensuring synchronization efficiency among nodes. Experimental results demonstrated significant speedups for *sparse* graphs, reflecting the algorithm's capability to exploit concurrency effectively in such scenarios. However, the authors underscored that dense graphs often became a bottleneck due to the higher volume of inter-node communication, which curtailed potential performance gains.

Later, Sanders and Schimek (Sanders & Schimek, 2023) advanced these concepts by designing massively parallel MST algorithms that integrated both the Message Passing Interface (MPI) and OpenMP. Their hybrid approach targeted two levels of parallelism, across nodes and within each node—enabling the algorithm to scale up to thousands of cores. While they reported excellent throughput for sparse graphs, they also observed diminishing returns in dense graph contexts, primarily attributed to intensive synchronization demands and unbalanced workload distributions across the nodes.

## 2.2 Hybrid Approaches and Theoretical Advances

Beyond purely distributed strategies, hybrid approaches to parallel MST computation have emerged as a means to balance computational and communication workloads (Sanders & Schimek, 2023). By combining distributed-memory techniques (for inter-node parallelism) with shared-memory optimizations (for intra-node parallelism), researchers aim to mitigate communication delays and enhance local processing efficiency. In particular, two-level communication strategies have proven effective: a lightweight, node-level synchronization mechanism paired with collective operations like `MPI_Alltoallv` ensures that the overhead of global data exchanges is minimized. These techniques yield especially strong performance for graphs with intermediate densities, where careful scheduling and partitioning can prevent severe imbalances in edge distribution.

On the theoretical front, the work of Chazelle (Chazelle, 2000) introduced an MST algorithm with an inverse-Ackermann complexity, represent-

ing a notable milestone in the upper-bound analysis of MST computations. Although not directly parallelizable in a straightforward manner, Chazelle's insights helped frame the complexity boundaries for MST algorithms and laid the groundwork for more efficient parallel designs. Researchers continue to explore ways to integrate these theoretical breakthroughs into practical parallel implementations, seeking to minimize asymptotic complexity while maximizing real-world performance.

### 2.3 Applications of Parallel MST Algorithms

Parallel MST algorithms, inclusive of Borůvka's method, have found broad applicability in domains such as network optimization, clustering, and large-scale data analytics. In network design, for instance, MST-based solutions help determine minimal-latency paths by connecting crucial infrastructure points with the smallest possible aggregate cost (Chung & Condon, 1996; Sanders & Schimek, 2023). Similarly, in data clustering, MST-based techniques (particularly when combined with cut-based heuristics) can segment massive datasets into cohesive subgroups, each representing a tight-knit cluster. The rapid growth of data volume, velocity, and variety has only accentuated the importance of scalable MST algorithms, as they facilitate near-real-time analysis in fields ranging from bioinformatics to social network analysis.

### 2.4 Challenges in Parallel Implementations

Despite the advantages offered by parallel MST algorithms, multiple practical challenges continue to impede their deployment at very large scales:

1. **Communication Overhead:** Coordinating MST construction across geographically distributed nodes often involves *substantial* synchronization and data exchange, especially for *dense* graphs with numerous edges (Chung & Condon, 1996).

2. **Load Balancing:** Inconsistent edge or vertex distribution among processes can create hotspots where certain compute nodes work overtime while others remain underutilized. Effective partitioning or dynamic scheduling is therefore crucial to maintain high parallel efficiency (Sanders & Schimek, 2023).

3. **Memory Constraints:** With extremely large datasets, memory usage becomes a limiting factor. Even if the algorithm scales theoretically, practical constraints related to node-local memory and interconnect bandwidth can lead to performance bottlenecks (Sanders & Schimek, 2023; Chazelle, 2000).

4. **Algorithmic Complexity vs. Practical Performance:** While refined algorithms may offer asymptotic advantages, implementing them in a real-world, massively parallel environment requires *significant engineering effort* and may not always translate to clear runtime benefits.

Borůvka's algorithm and its derivatives demonstrate notable promise for solving MST problems at scale, leveraging both distributed and shared-memory paradigms to accelerate computations. However, researchers must carefully navigate the interplay between **communication overhead**, **load balancing**, and **memory limitations** to fully exploit modern high-performance computing systems. Ongoing innovations—ranging from hybrid architectures to theoretically grounded optimizations—continue to refine and expand the capabilities of parallel MST algorithms for large-scale, real-world applications.

## 3 Methodology and Implementation

This section outlines the methodology and implementation of Borůvka's algorithm in three variations: sequential, parallel (using MPI), and hybrid parallel (using MPI and OpenMP). Each version focuses on optimizing the algorithm for its respective computing environment while preserving the integrity of the Minimum Spanning Tree (MST) solution. Pseudocode and key data structures are provided to explain the workflow of each implementation.

### 3.1 Sequential Implementation

The sequential version of Borůvka's algorithm processes the entire graph in a single memory space. The algorithm iteratively identifies the lightest edge for each component and merges them to construct the MST. The workflow follows these steps:

1. Initialize disjoint sets for each vertex using a Union-Find structure.

2. Iteratively find the lightest edge for each component.

3. Merge components based on the identified edges.

**Key Features:**

- **Union-Find Structure:** Efficiently tracks component membership and supports union operations.

- **Cheapest Edge Array:** Maintains the minimum-weight edge for each component.

- **Edge Filtering:** Eliminates redundant edges to reduce computation.

**Algorithm 1** Sequential Borůvka MST Construction

```
Input: Graph G(V, E)
Initialize Union-Find structure
while more than one component exists do
    Find the cheapest edge for each component
    for each component do
        Add the cheapest edge to MST
        Merge components using Union-Find
    end for
end while
Output: Total weight of MST
```

## 3.2 Parallel Implementation Using MPI

The parallel version employs MPI to distribute computation across multiple processes in a distributed memory system. The dataset is divided among processes, and local MSTs are computed independently. These local results are then merged iteratively to form the final MST.

**Workflow:**

1. Process 0 reads the input graph and distributes edges among all processes.

2. Each process computes the local cheapest edges for its portion of the graph.

3. Process 0 gathers local results, determines the global cheapest edges, and broadcasts updates.

4. Repeat until all components are merged.

**Key Features:**

- **MPI Communication:** Efficient distribution and aggregation of edge data using `MPI_Scatterv` and `MPI_Gatherv`.

- **Union-Find Adaptation:** Maintains disjoint sets locally on each process and coordinates merges globally.

**Algorithm 2** MPI-based Parallel Borůvka MST Construction

```
Input: Graph G(V, E), distributed among P processes
Distribute edges using MPI_Scatterv
while more than one component exists do
    Compute local cheapest edges
    Gather results using MPI_Gatherv
    Determine global cheapest edges on Process 0
    Broadcast updated components to all processes
end while
Output: Total weight of MST
```

## 3.3 Hybrid MPI+OpenMP Implementation Details

To further exploit modern multicore architectures, we extended our parallel Borůvka's MST algorithm to a **hybrid** MPI+OpenMP implementation. In this approach, the coarse-grained parallelism continues to rely on MPI for distributing edge data and synchronizing the merging of partial MSTs, while the fine-grained parallelism uses OpenMP threads within each MPI process to speed up local edge processing. This hybrid design can reduce the overall number of MPI processes (minimizing global communication) while concurrently leveraging multiple cores on each node.

### 3.3.1 Key Modifications for Hybrid Parallelism

Listing 1 shows the relevant excerpts of the hybrid code. The main changes relative to the pure MPI version include:

- **OpenMP Thread Spawning** via `omp_set_num_threads()`, based on a command-line parameter.

- **Parallel Local Edge Processing**: The local loop that scans the assigned edge partitions is protected with `#pragma omp parallel for`, splitting the iteration range among multiple threads.

- **Thread-Safe Update of Best Edges**: Within the local loop, a `#pragma omp critical` section ensures atomic updates to the `bestEdge` array. This prevents data races but also becomes a contention point when many threads run concurrently. For large thread counts, reducing lock contention (e.g., using a private array per thread and merging them post-loop) can further improve performance. But on this work it was a clear constraint when executing with more than 2 threads.

```
[...]
#pragma omp parallel for private(i)
for (i = 0; i < localCount; i++) {
 int src = localEdges[i].src;
 int dst = localEdges[i].dst;
 int weight = localEdges[i].weight;

 int r1 = findRoot(src, parent);
 int r2 = findRoot(dst, parent);

 if (r1 != r2) {
  #pragma omp critical
  {
   if (weight < bestWeight[r1]) {
    bestWeight[r1] = weight;
    bestEdge[r1]   = localEdges[i];
   }
   if (weight < bestWeight[r2]) {
    bestWeight[r2] = weight;
    bestEdge[r2]   = localEdges[i];
   }
  }
 }
}
[...]
```

**Figure 1:** Excerpt from the Hybrid MPI+OpenMP Borůvka MST Implementation.

# 4 Experimental Evaluation

This section presents the experimental results obtained from the implementation of our parallel Borůvka's algorithm across multiple datasets of varying sizes and densities, focusing on the larger datasets, as the papers from which the experimentation is based used much smaller graphs or smaller degrees per node. The primary aim is to assess the performance and effectiveness of our approach, focusing on execution time, speedup, and efficiency metrics. By comparing the parallel implementation to a serial counterpart, we highlight how the algorithm scales when increasing both the number of MPI processes and OpenMP threads.

## 4.1 Hardware

To evaluate the performance of our Borůvka's algorithm, we used the HPC cluster at the University of Trento. This environment provides the necessary computational power to handle large-scale graph instances and to examine strong and weak scalability. The primary specifications of this environment are as follows:

- **Operating System of the nodes**: Linux CentOS 7
- **Number of nodes**: 126
- **CPU cores**: 6.092
- **CUDA cores**: 37.376
- **RAM**: 53 TB
- **Connectivity**: 10 Gb/s Ethernet interconnect with select nodes also featuring Infiniband at 40 Gb/s or Omnipath at 100 Gb/s

This computational environment provided the required processing power for handling large-scale graph instances and for examining scalability.

## 4.2 Experimental Setup and Parameters

For our evaluation, we considered four synthetic datasets with differing numbers of vertices, average degrees, and total edges. We measured performance under various hybrid MPI+OpenMP configurations:

- **Number of MPI Processes**: {1, 2, 4, 8, 16, 32}
- **Number of OpenMP Threads**: {1, 2, 4}

Only results from the hybrid implementation were recorded and used for evaluation, as the performance when using one cpu and one thread was better than the sequential and purely MPI execution performance.

Four datasets (summarized in Table **??**) were used to investigate the scalability and performance of our approach:

1. **Dataset A (Large, Sparse)**:
   - *Vertices*: 65.608.366
   - *Average Degree*: 57
   - *Total Edges*: 1.869.838.431
2. **Dataset B (Large, Sparser Variant)**:
   - *Vertices*: 65.608.366
   - *Average Degree*: 9,5
   - *Total Edges*: 311.639.739
3. **Dataset C (Smaller, Extremely Dense)**:
   - *Vertices*: 50.000
   - *Average Degree*: 42.750
   - *Total Edges*: 1.068.750.000
4. **Dataset D (Moderate in Size, Dense)**:
   - *Vertices*: 100.000
   - *Average Degree*: 38.000
   - *Total Edges*: 1.900.000.000

Datasets A and D can be viewed as "larger" in terms of vertices and/or edge count, while Datasets B and C illustrate scenarios where the total graph size is somewhat smaller (or sparser). All tests used a *hybrid MPI+OpenMP* parallelization with varying numbers of MPI processes and OpenMP threads. Details on hardware and the experimental setup appear in Section 4.1.

For complete experimental data on execution times, speedups, and efficiency under each configuration, see Tables 2 and 3 in the annex.

## 4.3 Performance Metrics

To evaluate performance, we recorded:

- **Execution Time**: The wall-clock time measured via `MPI_Wtime`, with the reported value being the maximum time among all MPI processes.
- **Speedup** $S$: Defined as

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}},$$

using the single-process, single-thread version as the baseline for $T_{\text{serial}}$, as this had a shorter execution time than the sequential code.

- **Efficiency** $E$:

$$E = \frac{S}{(\text{number of total cores})}.$$

This metric shows how effectively the algorithm utilizes the available compute resources.

## 4.4 Experimental Results

In Table 1, each row denotes a particular configuration of (*MPI Processes*, *OpenMP Threads*) alongside the measured execution time, speed-up, and the corresponding efficiency. For all the results, there are two tables 2  3 in the annex.

### 4.4.1 Execution Time

The execution time tends to decrease as we add more processes and threads; however, the *degree of improvement* varies across datasets:

- **Datasets A & D (Larger Cases)**. These more demanding cases reap the most noticeable absolute time reduction from higher concurrency, though overhead eventually reduces the incremental gains. As concurrency grows (e.g., 16 or 32 total parallel tasks), the overhead from synchronization and merges can become nontrivial.
- **Datasets B & C (Smaller / Different Densities)**. For Dataset B (same vertex count as A, but lower degree) we see significant *relative* speedups, but absolute times remain higher than expected for some configurations, likely due to suboptimal load balancing or communication overhead. Dataset C, despite fewer vertices, is extremely dense (42.7k average degree), so parallelism helps, but we see diminishing returns faster with high concurrency.

Figure 2 compares the *execution times* for Dataset A (large, sparse) and Dataset D (moderately sized, dense) under selected concurrency configurations. Although both benefit from parallelization, Dataset A exhibits faster absolute times with fewer MPI processes when compared to Dataset D, reflecting differences in total edge processing and communication patterns. Proving that the algorithm works better for a sparse graph, similar to the discoveries of Sanders & Schimek and Chung & Condon. For all the results, there are graphs representing them in the annex 8.

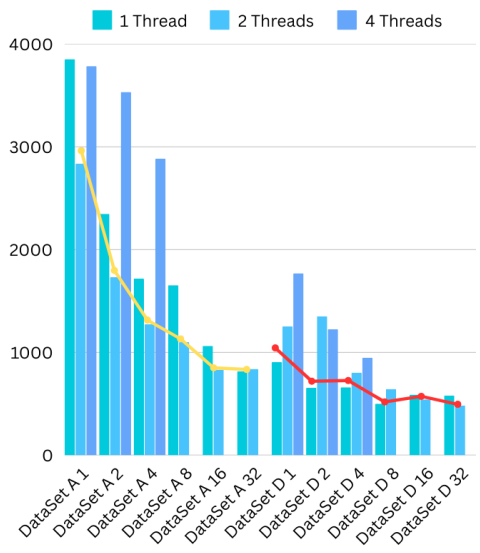


**Figure 2:** Execution Time Comparison for Dataset A (yellow line) and Dataset D (red line). Configurations along the x-axis reflect different (*processes*, *threads*).

### 4.4.2 Speedup

As shown in Tables 2  3 in the annex, speed-up initially grows when moving from 1 process to 2 or 4, then tapers off at larger concurrency levels due to communication overhead and synchronization inside the hybrid code. Also, when moving from one to two threads, there is an increment in some cases, but when moving to four and eight threads it tapers off. Both situations may be attributed to the following reasons:

**Table 1:** Extract of Execution Times, Speedup, and Efficiency for All Four Datasets Under Different MPI+OpenMP Configurations (*excerpted cells for brevity*).

| Dataset | Vertices | Edges | MPI Procs | OMP Threads | Time (m) | Speedup (Efficiency) |
|---|---|---|---|---|---|---|
| **A** | 65,6M | 1,87B | 1 | 1 | 3.8524 | 1.0000 (1,0000) |
| **A** | 65,6M | 1,87B | 1 | 2 | 2.8362 | 1.3583 (0,6792) |
| **A** | 65,6M | 1,87B | 8 | 2 | 1.0993 | 3.5045 (0,2190) |
| **A** | 65,6M | 1,87B | 16 | 2 | 0.8157 | 4.7228 (0,1476) |
| **A** | 65,6M | 1,87B | 32 | 2 | 0.8396 | 4.5885 (0,0717) |
| **B** | 65,6M | 0,31B | 1 | 1 | 689.7374 | 1.0000 (1,0000) |
| **B** | 65,6M | 0,31B | 1 | 2 | 501.7044 | 1.3748 (0,6874) |
| **B** | 65,6M | 0,31B | 8 | 1 | 295.7351 | 2.3323 (0,1458) |
| **B** | 65,6M | 0,31B | 16 | 2 | 232.2613 | 2.9697 (0,0938) |
| **B** | 65,6M | 0,31B | 32 | 2 | 252.6916 | 2.7296 (0,0426) |
| **C** | 50k | 1,07B | 1 | 1 | 498.1425 | 1.0000 (1,0000) |
| **C** | 50k | 1,07B | 1 | 2 | 714.3157 | 0.6974 (0,3487) |
| **C** | 50k | 1,07B | 8 | 2 | 335.7237 | 1.4838 (0,0927) |
| **C** | 50k | 1,07B | 16 | 1 | 268.0721 | 1.8582 (0,1161) |
| **C** | 50k | 1,07B | 32 | 2 | 311.3391 | 1.6000 (0,0250) |
| **D** | 100k | 1,90B | 1 | 1 | 905.0462 | 1.0000 (1,0000) |
| **D** | 100k | 1,90B | 1 | 2 | 1,253.0700 | 0.7223 (0,3611) |
| **D** | 100k | 1,90B | 2 | 1 | 655.8867 | 1.3799 (0,6899) |
| **D** | 100k | 1,90B | 16 | 2 | 529.9107 | 1.5617 (0,0488) |
| **D** | 100k | 1,90B | 32 | 2 | 482.7178 | 1.8749 (0,0293) |

- **Lock Contention and Communication**. Each process must merge partial MSTs in every iteration of Borůvka, and at high MPI ranks, calls to `MPI_Allreduce` and `MPI_Bcast` become more frequent. Additionally, in the hybrid code, updates to the `bestEdge` array are protected by `#pragma omp critical`, creating potential bottlenecks for large thread counts.
- **Dense Edges**. In Datasets C and D, the overhead from processing very large edge sets can limit how effectively additional cores accelerate the solution.

For all the results, there are graphs representing them in the annex 8.

### 4.4.3 Efficiency

Efficiency indicates what fraction of the *ideal speedup* (= number of cores) is realized. As expected, efficiency is highest when the problem size is large enough to distribute substantial work per task. Key trends include:

- **High Efficiency at Small Concurrency**. At low MPI ranks, parallel overhead is minimal, and the algorithm scales well.
- **Diminishing Returns at High Concurrency**. Both communication and lock contention reduce the fraction of time spent on *useful* MST computations. This is particularly evident in larger concurrency runs for Datasets B and C, where efficiency may drop below 10%.

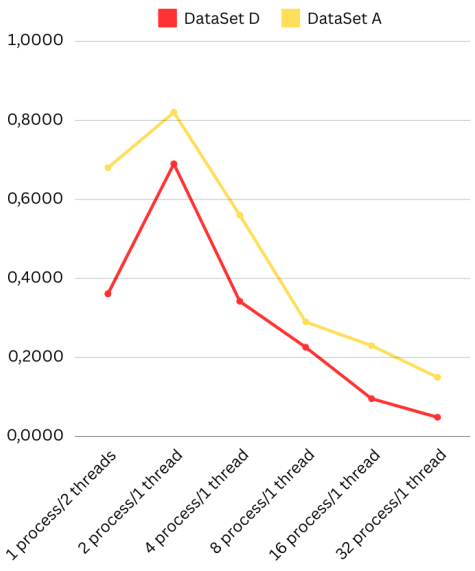


**Figure 3:** Efficiency Comparison for Dataset A (yellow line) and Dataset D (red line). Note the quicker drop in efficiency for the denser graph (Dataset D).

- **Comparisons Across Datasets**. Dataset A remains more efficient than B under similar concurrency, likely because the

average degree is higher in A, providing a bigger chunk of computation per process. Dataset C's extreme density also saturates concurrency more quickly.

**Dataset A vs. Dataset D (Efficiency).** Figure 3 highlights a direct efficiency comparison between Dataset A and Dataset D under the same (*processes*, *threads*) configurations. Dataset D, being denser, exhibits a more rapid decline in efficiency as we add more processes, presumably due to heavier synchronization costs per iteration of Borůvka's merging steps.

For all the results, there are graphs representing them in the annex 8.

### 4.5 Discussion

By examining both **smaller** and **larger** datasets with varying degrees of density, we see that:

- **Parallelism is Beneficial**. Even for smaller datasets, we can achieve improved runtimes when the concurrency level is matched appropriately.
- **Overhead Scales Non-linearly**. With extremely dense graphs (Datasets C & D), or high degrees of parallelism, both communication and thread-synchronization overhead can outweigh the benefits of additional tasks.
- **Load Balancing Remains Key**. Since we distribute edges evenly in the current implementation, graphs with skewed edge distributions or high degrees may experience suboptimal concurrency.

Overall, these experiments reinforce that a hybrid MPI+OpenMP Borůvka's algorithm is highly effective at reducing MST computation time for *large* or *dense* problems. However, the algorithm demonstrates typical HPC scaling trade-offs, yielding *diminishing returns* at high concurrency, especially for smaller datasets or extremely dense graphs.

## 5 Conclusion

In this paper, we have investigated a viable strategy for parallelizing Borůvka's algorithm using the MPI framework. Our primary objective was to harness modern high-performance computing resources to accelerate and scale Minimum Spanning Tree (MST) computations, particularly when handling large and complex graphs.

The experimental results clearly demonstrate that our parallel formulation substantially improves execution times compared to a serial version. We observe notable speedup and reasonable weak scalability for both large sparse graphs and moderately sized but highly dense graphs. These findings highlight the effectiveness of the parallel Borůvka's approach in distributed-memory settings where MST computations must handle vast numbers of vertices and edges.

However, our solution does exhibit limitations with strong scalability. The performance gains begin to plateau past a certain parallelism threshold, largely due to synchronization and communication overhead in merging partial MSTs. Despite these constraints, the approach still offers clear performance benefits for many real-world scenarios, making it a valuable strategy for addressing MST problems in high-throughput data environments.

## 6 Future Works

Although our current Borůvka MST implementation already uses a **hybrid MPI+OpenMP** approach (as shown in Section 3.3), there remain multiple opportunities to extend and optimize the solution further:

- **Refining Hybrid Concurrency.** While our code leverages both MPI for inter-process communication and OpenMP for on-node parallelism, the current design still relies on a `#pragma omp critical` directive to update best edges. Under large thread counts, this can become a bottleneck, as shown in the results. Future work could investigate lock-free or reduction-based strategies (e.g., private arrays in each thread, followed by a local merge) to reduce contention and improve scalability.
- **Exploration of Larger and More Diverse Graphs.** Our experiments covered a wide range of graph sizes and densities, but real-world MST applications often demand even greater scalability or handle specialized structures (e.g., social networks, sensor networks, or bioinformatics). Evaluating the hybrid algorithm on these varied graph types would help confirm its robustness and highlight additional tuning opportunities (e.g., node-level thread affinity, better partitioning strategies).
- **Communication Overhead Reduction.** As indicated in Section 4, merging partial MSTs and broadcasting updates across MPI processes remains the primary source of overhead at high concurrency levels. Reducing global synchronization frequencies, leveraging asynchronous communication, or implementing more advanced gather/scatter routines could further increase efficiency.

- **Load Balancing and Graph Partitioning.** Even though edges are distributed roughly evenly in our current approach, real-world graphs are rarely uniform. Adopting advanced partitioning or on-demand load balancing could prevent certain processes from becoming bottlenecks when the graph has skewed distributions or high-degree subregions.
- **Additional Hybrid Optimizations.** Beyond refining the existing MPI+OpenMP model, further hybrid extensions, such as dynamic switching between locking and lock-free updates based on thread counts, may unlock better scalability. Another possibility is combining OpenMP with specialized accelerators (e.g., GPUs) for the local edge processing phase, though this requires more complex data movement strategies.

In conclusion, the *hybrid* Borůvka's MST algorithm detailed in Section 3.3 demonstrates the value of a two-level parallel strategy, but addressing the challenges of *thread contention*, *communication overhead*, and *load imbalance* can push performance even further. We believe ongoing work in these directions will continue to expand the algorithm's usability and efficiency for large-scale, complex MST problems.

# 7 References


1. Chung, S., & Condon, A. (1996). Parallel Implementation of Borůvka's Minimum Spanning Tree Algorithm. *Proceedings of IPPS '96.*

2. Sanders, P., & Schimek, M. (2023). Engineering Massively Parallel MST Algorithms. *Institute of Theoretical Informatics, Karlsruhe Institute of Technology.*

3. Chazelle, B. (2000). A Minimum Spanning Tree Algorithm with Inverse-Ackermann Complexity. *Journal of the ACM.*

4. Lfb1206/HPC_Personal GitHub Repository. (n.d.). Retrieved January 10, 2025, from https://github.com/lfb1206/HPC_Personal

# 8 Annex

**Table 2:** Execution Times, Speedup, and Efficiency for All Four Datasets Under Different MPI+OpenMP Configurations (*excerpted cells for brevity*).

| Dataset | Vertices | Edges | MPI Procs | OMP Threads | Time (m) | Speedup (Efficiency) |
|---|---|---|---|---|---|---|
| **A** | 65,6M | 1,87B | 1 | 1 | 64,2069 | 1,0000 (1,0000) |
| **A** | 65,6M | 1,87B | 1 | 2 | 47,2700 | 1,3583 (0,6792) |
| **A** | 65,6M | 1,87B | 1 | 4 | 63,0807 | 1,0179 (0,2545) |
| **A** | 65,6M | 1,87B | 2 | 1 | 39,1344 | 1,6407 (0,8203) |
| **A** | 65,6M | 1,87B | 2 | 2 | 28,8617 | 2,2246 (0,5562) |
| **A** | 65,6M | 1,87B | 2 | 4 | 58,9030 | 1,0900 (0,1363) |
| **A** | 65,6M | 1,87B | 4 | 1 | 28,6288 | 2,2427 (0,5607) |
| **A** | 65,6M | 1,87B | 4 | 2 | 21,2096 | 3,0273 (0,3784) |
| **A** | 65,6M | 1,87B | 4 | 4 | 48,0668 | 1,3358 (0,0835) |
| **A** | 65,6M | 1,87B | 8 | 1 | 27,5521 | 2,3304 (0,2913) |
| **A** | 65,6M | 1,87B | 8 | 2 | 18,3212 | 3,5045 (0,2190) |
| **A** | 65,6M | 1,87B | 16 | 1 | 17,7261 | 3,6222 (0,2264) |
| **A** | 65,6M | 1,87B | 16 | 2 | 13,8650 | 4,6309 (0,1447) |
| **A** | 65,6M | 1,87B | 32 | 1 | 13,5951 | 4,7228 (0,1476) |
| **A** | 65,6M | 1,87B | 32 | 2 | 13,9930 | 4,5885 (0,0717) |
| **B** | 65,6M | 0,31B | 1 | 1 | 11,4956 | 1,0000 (1,0000) |
| **B** | 65,6M | 0,31B | 1 | 2 | 8,3617 | 1,3748 (0,6874) |
| **B** | 65,6M | 0,31B | 1 | 4 | 13,3947 | 0,8582 (0,2146) |
| **B** | 65,6M | 0,31B | 2 | 1 | 8,4399 | 1,3621 (0,6810) |
| **B** | 65,6M | 0,31B | 2 | 2 | 6,5558 | 1,7535 (0,4384) |
| **B** | 65,6M | 0,31B | 2 | 4 | 9,6198 | 1,1950 (0,1494) |
| **B** | 65,6M | 0,31B | 4 | 1 | 6,7385 | 1,7060 (0,4265) |
| **B** | 65,6M | 0,31B | 4 | 2 | 5,0855 | 2,2605 (0,2826) |
| **B** | 65,6M | 0,31B | 4 | 4 | 7,7429 | 1,4847 (0,0928) |
| **B** | 65,6M | 0,31B | 8 | 1 | 5,0099 | 2,2946 (0,2868) |
| **B** | 65,6M | 0,31B | 8 | 2 | 4,9289 | 2,3323 (0,1458) |
| **B** | 65,6M | 0,31B | 16 | 1 | 5,3690 | 2,1411 (0,1338) |
| **B** | 65,6M | 0,31B | 16 | 2 | 3,8710 | 2,9697 (0,0928) |
| **B** | 65,6M | 0,31B | 32 | 1 | 4,9026 | 2,3448 (0,0733) |
| **B** | 65,6M | 0,31B | 32 | 2 | 4,2115 | 2,7296 (0,0426) |

**Table 3:** Execution Times, Speedup, and Efficiency for All Four Datasets Under Different MPI+OpenMP Configurations (*excerpted cells for brevity*).

| Dataset | Vertices | Edges | MPI Procs | OMP Threads | Time (m) | Speedup (Efficiency) |
|---|---|---|---|---|---|---|
| **C** | 50k | 1,07B | 1 | 1 | 8,3024 | 1,0000 (1,0000) |
| **C** | 50k | 1,07B | 1 | 2 | 11,9053 | 0,6974 (0,3487) |
| **C** | 50k | 1,07B | 1 | 4 | 14,8641 | 0,5586 (0,1396) |
| **C** | 50k | 1,07B | 2 | 1 | 6,0954 | 1,3621 (0,6810) |
| **C** | 50k | 1,07B | 2 | 2 | 10,8592 | 0,7645 (0,1911) |
| **C** | 50k | 1,07B | 2 | 4 | 12,0343 | 0,6899 (0,0862) |
| **C** | 50k | 1,07B | 4 | 1 | 5,5954 | 1,4838 (0,3709) |
| **C** | 50k | 1,07B | 4 | 2 | 5,8431 | 1,4209 (0,1776) |
| **C** | 50k | 1,07B | 4 | 4 | 8,5653 | 0,9693 (0,0606) |
| **C** | 50k | 1,07B | 8 | 1 | 4,7095 | 1,7629 (0,2204) |
| **C** | 50k | 1,07B | 8 | 2 | 6,3178 | 1,3141 (0,0821) |
| **C** | 50k | 1,07B | 16 | 1 | 4,4679 | 1,8582 (0,1161) |
| **C** | 50k | 1,07B | 16 | 2 | 5,0365 | 1,6484 (0,0515) |
| **C** | 50k | 1,07B | 32 | 1 | 4,3185 | 1,9225 (0,0601) |
| **C** | 50k | 1,07B | 32 | 2 | 5,1890 | 1,6000 (0,0250) |
| **D** | 100k | 1,90B | 1 | 1 | 15,0841 | 1,0000 (1,0000) |
| **D** | 100k | 1,90B | 1 | 2 | 20,8845 | 0,7223 (0,3611) |
| **D** | 100k | 1,90B | 1 | 4 | 29,4927 | 0,5115 (0,1279) |
| **D** | 100k | 1,90B | 2 | 1 | 10,9314 | 1,3799 (0,6899) |
| **D** | 100k | 1,90B | 2 | 2 | 22,5389 | 0,6692 (0,1673) |
| **D** | 100k | 1,90B | 2 | 4 | 20,4108 | 0,7390 (0,0924) |
| **D** | 100k | 1,90B | 4 | 1 | 11,0219 | 1,3686 (0,3421) |
| **D** | 100k | 1,90B | 4 | 2 | 13,3725 | 1,1280 (0,1410) |
| **D** | 100k | 1,90B | 4 | 4 | 15,7989 | 0,9548 (0,0597) |
| **D** | 100k | 1,90B | 8 | 1 | 8,3447 | 1,8076 (0,2260) |
| **D** | 100k | 1,90B | 8 | 2 | 10,7217 | 1,4069 (0,0879) |
| **D** | 100k | 1,90B | 16 | 1 | 9,8363 | 1,5335 (0,0958) |
| **D** | 100k | 1,90B | 16 | 2 | 9,0485 | 1,6670 (0,0521) |
| **D** | 100k | 1,90B | 32 | 1 | 9,6588 | 1,5617 (0,0488) |
| **D** | 100k | 1,90B | 32 | 2 | 8,0453 | 1,8749 (0,0293) |

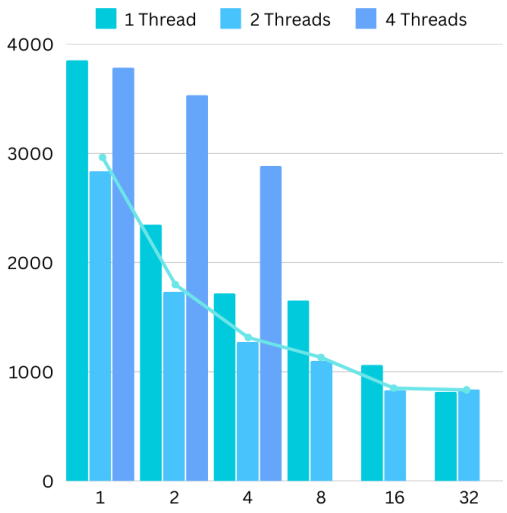


**Figure 4:** DataSet A Execution Times.

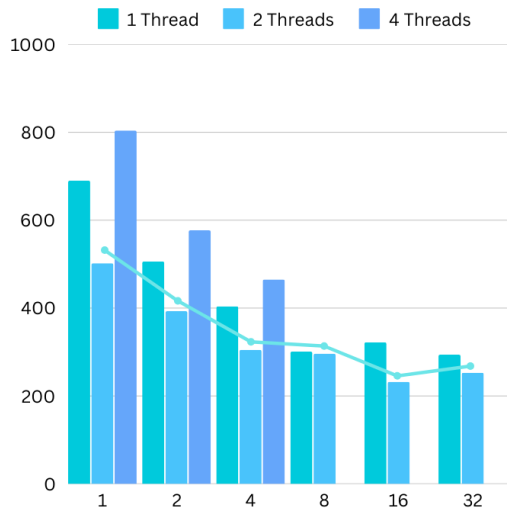


**Figure 5:** DataSet B Execution Times.

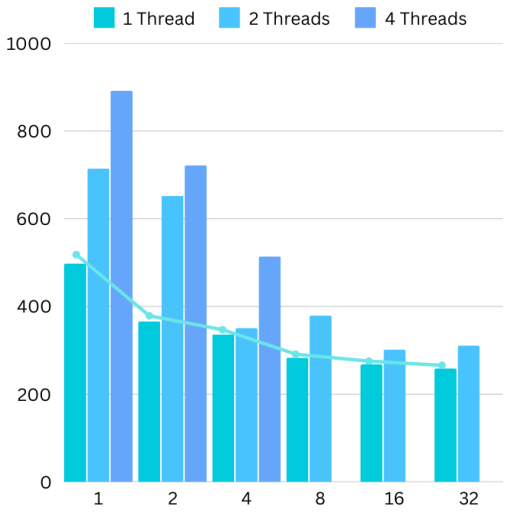


**Figure 6:** DataSet C Execution Times.

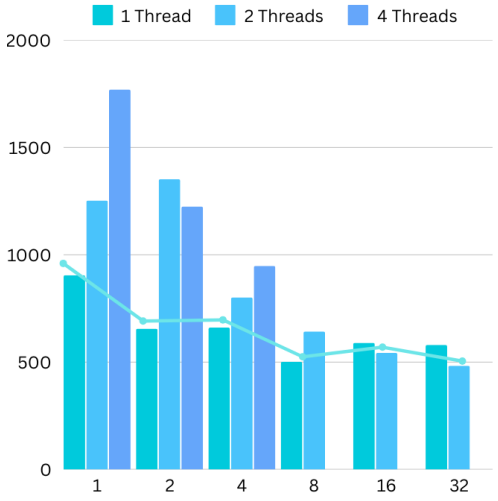


**Figure 7:** DataSet D Execution Times.

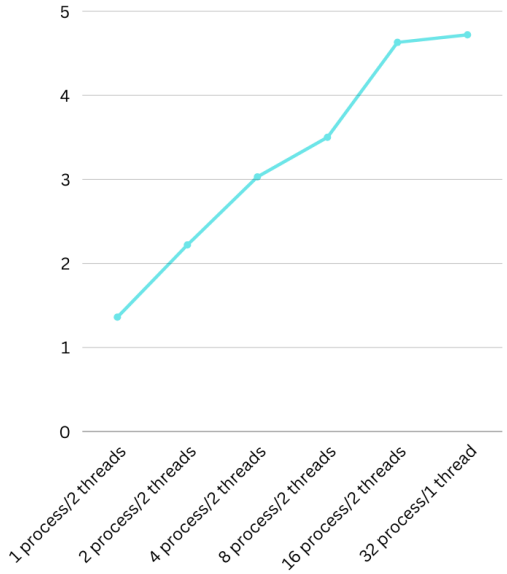


**Figure 8:** DataSet A Speed Up.

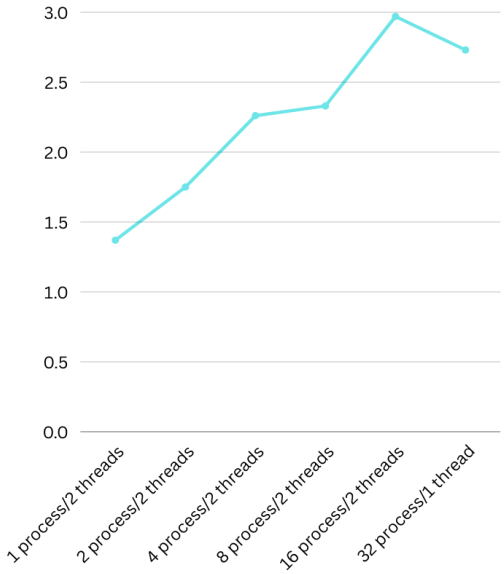


**Figure 9:** DataSet B Speed Up.

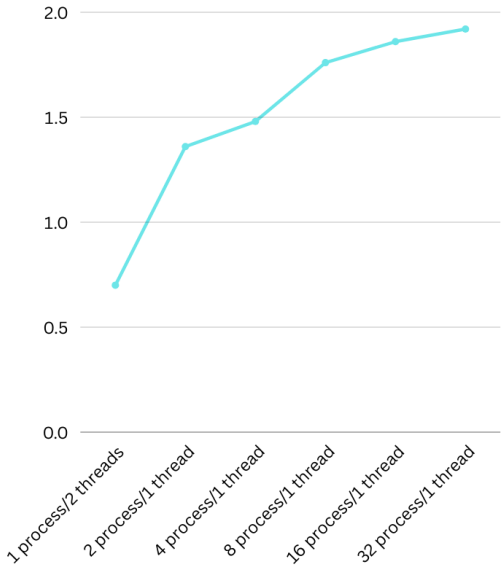


**Figure 10:** DataSet C Speed Up.

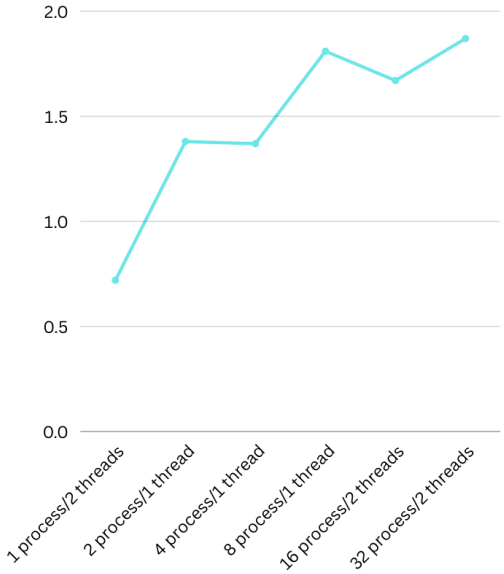


**Figure 11:** DataSet D Speed Up.

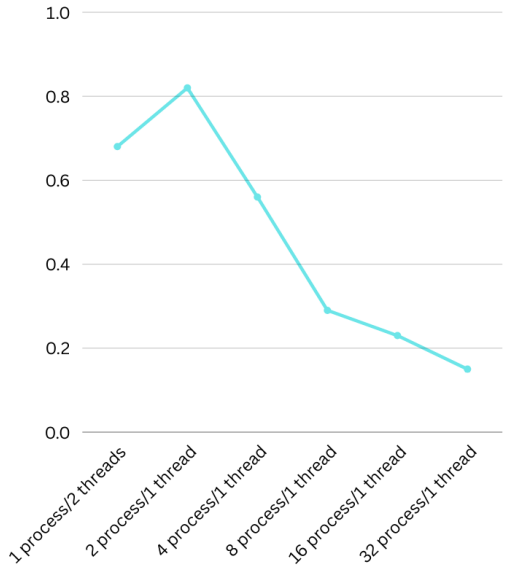


**Figure 12:** DataSet A Efficiency.

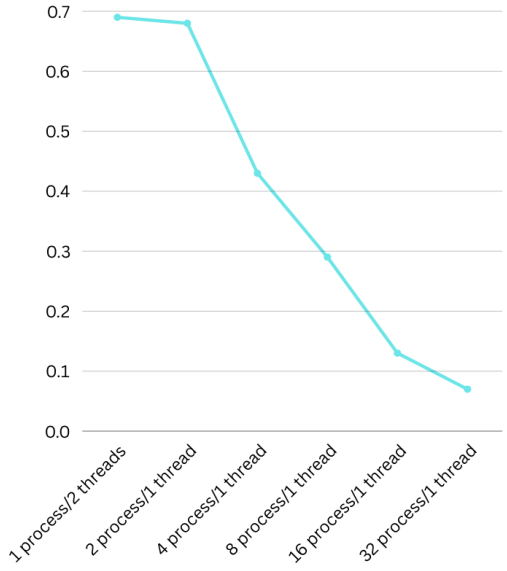


**Figure 13:** DataSet B Efficiency.

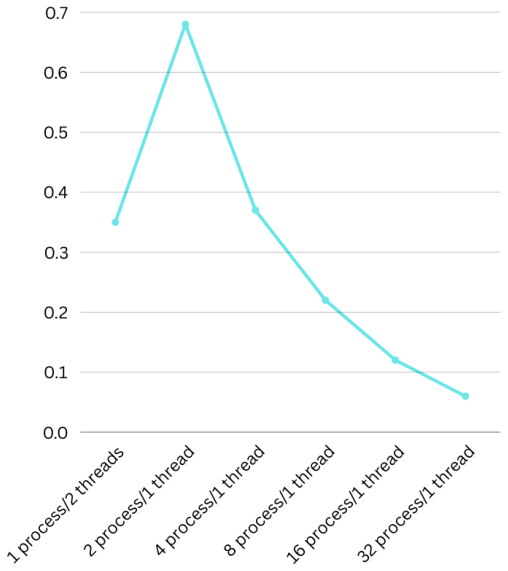


**Figure 14:** DataSet C Efficiency.

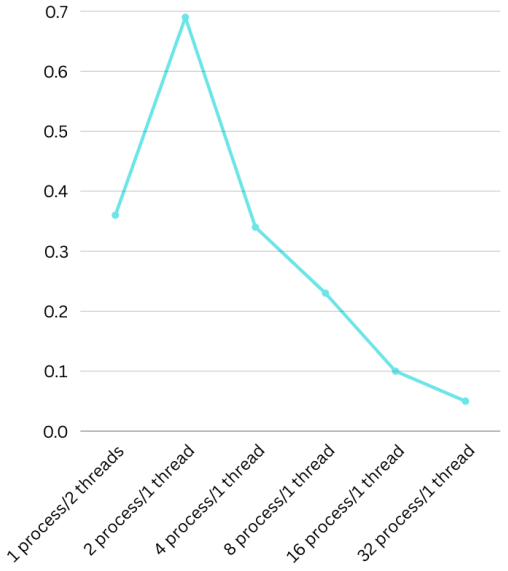


**Figure 15:** DataSet D Efficiency.