HPC Report
Parallel Borůvka algorithm

Elia Gatti, Matteo Bitussi
University of Trento

2024/2025

1 Abstract

Boruvka's algorithm is a greedy technique for determining the minimum spanning tree in an undirected graph. This report proposes a parallel implementation of the method, using a hybrid technique that combines MPI and OpenMP. The implementation has undergone testing on the unitn cluster using various graphs and configurations to verify its accuracy and scalability capabilities. Our testing focused on dense graphs, as these are the graphs where parallelism has a greater impact. The parallel version had respectable scaling performance, even if we identified potential areas for further possible enhancement.

2 Background

2.1 Boruvka algorithm

Boruvka's algorithm is a greedy method for finding the least spanning tree in an undirected graph [1]. It is named for its creator, Otakar Boruvka, who published it in 1926 as a technique of developing electricity networks for his nation. The algorithm starts by finding the smallest edge value for each vertex in the graph and selecting it as part of the MST. Then, for each subgraph, the minimum weight edge to another subgraph is found and added to the MST. This second step is repeated until there are no subgraphs left. Using this method, the algorithm is guaranteed to discover the shortest spanning tree. The worst-case time complexity of the algorithm is $O(E \log V)$, where E represents the number of edges and V represents the number of vertices in the graph. In cases where two edges have equal weights, a specific criterion must be applied to prevent the formation of cycles by ensuring consistent edge selection across different trees.

2.2 Parallel Metrics

In parallel computing, evaluating performance involves several key metrics:

Speedup quantifies the performance gain of a parallel program compared to its serial counterpart. It is calculated as the ratio of the serial execution time (T_s) to the parallel execution time (T_p):

$$\text{Speedup} = \frac{T_s}{T_p} \tag{1}$$

Efficiency measures how effectively the resources of a parallel implementation are utilized. It is defined as the speedup divided by the number of processors used (p):

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{T_s}{p \times T_p} \tag{2}$$

Where T_s is the serial execution time and T_p is the parallel execution time [2].

Scalability assesses a system's capacity to handle increased workloads by adding more processors. There are two primary types:

- **Strong Scalability** (or strong scaling) examines how the solution time varies with the number of processors for a fixed total problem size. Ideal strong

scaling means that doubling the number of processors halves the execution time [3].

- **Weak Scalability** (or weak scaling) evaluates how the solution time changes as both the problem size and the number of processors increase proportionally, keeping the workload per processor constant. The goal is to maintain consistent execution time as the system scales [3].

3 Design and Implementation

3.1 Assumptions

Due to technical difficulties in managing a situation where there is an uneven number of vertexes assigned to the processes, we assumed that the input graph of the algorithm has a number of vertexes that can be equally divided along all processes. This does not pose a limit to the input graphs, as it is sufficient to change the number of processes in a way that is a multiple of the number of vertexes in the input.

3.2 The Parallel Version

The algorithm accepts as input a graph in a standard format.

To parallelise Borůvka's algorithm, we employed a vertex-distribution approach, assigning an equal number of vertices to each process. In the first phase, each process identifies the lowest-weight edge for each of its assigned vertices and then communicates this information to all other processes. Simultaneously, each process receives the corresponding information from all other processes. This all-to-all communication is crucial, as each process requires global knowledge of the edge selections made by all other processes.

The subsequent phase mirrors the first: each process determines the minimum-weight edge connecting each of its vertices to a distinct subtree. This information is then shared with all other processes, which reciprocate by sending their own values. The minimum spanning tree (MST) is then updated by selecting the lowest-weight edge to connect each distinct component, leveraging the minimum-weight edge information from every subtree. All processes end up at the same MST update decision.

This process is repeated until no subtrees remain.

While this approach involves some redundant computations across processes, it was chosen to mitigate the substantial communication overhead that would result from centralised computation and subsequent data distribution to all processes, particularly for large datasets. Each process requires complete knowledge of the graph's current state to make informed decisions regarding minimum-weight edge selection.

In cases where multiple edges share the same minimum weight, a consistent decision logic is applied across all processes. This logic selects the edge with the lowest vertex IDs.

3.2.1 Implementation Details

The implementation utilises a hybrid MPI (multiprocessing) and OpenMP (multithreading) approach, with the option to disable the OpenMP component.

The main process reads the graph from an input file before forking child processes. The graph is stored as a triangular adjacency matrix, exploiting the undirected nature of the graph to conserve memory. A separate minimum spanning tree (MST) connection matrix is also maintained. Although an adjacency matrix was selected for its simplicity of implementation, we acknowledge that it is not the most memory-efficient representation, particularly for large, sparse graphs. Alternative representations, such as adjacency lists, could be explored for memory optimisation.

Each process is responsible for an equal number of vertices.

In the first iteration, each process identifies the minimum-weight edge for each of its assigned vertices and stores these edges in a list. This list is then shared with all other processes using an MPI_Allgather operation, ensuring that every process possesses the complete set of minimum-weight edges. Based on this global minimum-weight edge information, each process updates its local MST representation.

The second phase is similar. Each process iterates through its assigned vertices, searching for the minimum-weight edge connecting to a different subtree. The identified edge weights are stored in an array. Then, for each root of the subtrees, the minimum-weight edge among the values held by the current process is determined and stored in a separate array. This array contains one entry per subtree root. An MPI_Allreduce operation is then performed on this array, resulting in an array containing the minimum-weight edge for each subtree root (i.e., the minimum-weight edge connecting each subtree). Based on this information, each process updates its local MST.

3.2.2 Memory considerations

As introduced in the previous section, our algorithm has a pretty high memory consumption given by how it stores the graph; the space used increases polynomially with a degree of 2 with the number of vertices in the processes, as the matrix is large $|V|^2/2$ and it is used only in half. More specifically with $O(2 \cdot \frac{|V|^2}{2})$ as there are two matrices, which is equal to $O(|V|^2)$. There are also some arrays used to remember the number of processes, but they are already included in the big O. Another consideration to make is the maximum value a weight can have, as, based on that, the matrix will require more or less space. We decided to use the *unsigned short* type in C, which uses 2 bytes and has a max value of 65.535 minus one value that has to be used as a special value. Depending on the specific graph as input, the type would need to be changed, as the max weight value might not fit the unsigned short. Note that the space considered is **for a single process**; this means that the more processes are used, the more memory is needed.

3.2.3 OpenMP

The parallel implementation was enhanced with OpenMP multithreading applied to all suitable for loops. This approach leverages multiple threads within each process to improve performance. Initially, extensive OpenMP pragmas were inserted throughout the iterative sections of the algorithm. However, during the final stages of development, it was discovered that some of these parallelizations introduced errors. Consequently, the scope of OpenMP usage was reduced to a smaller, verified subset of the program. The code is designed in a way that it is scalable even without OpenMP, but its performance is improved even more by using it.

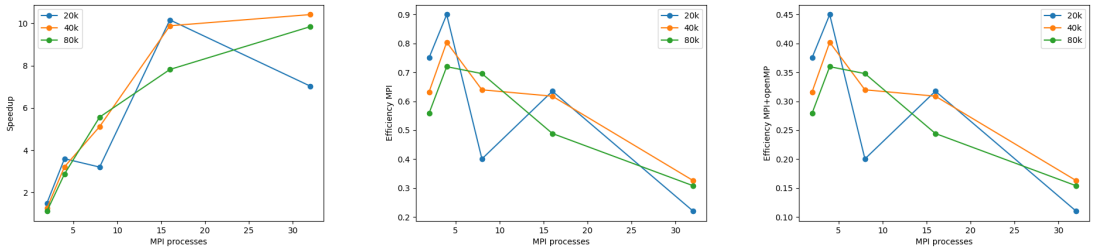
3.3 Data Dependencies

For the analysis of data dependencies, we examined the code snippet shown in Figure 1. This code is used to find the minimum-weight edges for each node of the graph managed by each process. The existing implementation already includes OpenMP pragmas to parallelise the iterations of the outer loop. In this data dependence analysis, we will consider the effects of parallelising both loops, as if we were adding a pragma parallel for the inner loop as well.

Starting with the outer loop, we observe that the variables declared within it, `local_min_weight` and `local_min_id`, are reset at the beginning of each iteration. Therefore, considering only the outer loop, we find no relevant data dependencies. Each thread has its own private copy of these two variables, and no iteration depends on the other. Furthermore, if only the outer loop is parallelised, the inner loop is executed serially by a single thread, eliminating the possibility of race conditions.

However, if we consider parallelising the inner loop, we notice that `local_min_weight` and `local_min_id` become shared variables among the threads. This introduces potential dependencies because these variables are both read and written. The table summarises some of the key dependencies within the analysed code snippet. The "Loop carried?" column indicates whether the dependency is carried across iterations of the outer loop (i) or the inner loop (j).

Memory Location	Earlier Statement			Later Statement			Loop carried?	Dependency Type
	Line	Iteration(s)	Access	Line	Iteration(s)	Access		
local_min_weight	430	i, j	Write	429	i, j+1	Read	no (i), yes(j)	Flow
local_min_weight	430	i, j	Write	430	i, j+1	Write	no (i), yes(j)	Output
weight	428	i, j	Write	430	i, j	Read	no	Flow
local_min_id	431	i, j	Write	430	i, j+1	Write	no (i), yes(j)	Output
local_min_id	431	i, j	Write	435	i, j+1	Read	no (i), yes (j)	Flow
j	426	i, j	Write	428	i, j	Read	no	Flow
j	426	i, j	Write	431	i, j	Read	no	Flow



(a) Speedup (b) Efficiency MPI (c) Efficiency MPI+openMP

```c
static void find_local_minimum_edges(int vertex_per_process, uint16_t** graph, int* lightest_edges) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = MY_NODES_FROM; i < MY_NODES_TO; i++) {
        uint16_t local_min_weight = MAX_EDGE_VALUE;
        int local_min_id = -1;

        // First find minimum weight without SIMD
        for (int j = 0; j < NODE_COUNT; j++) {
            if (i == j) continue;
            uint16_t weight = get_from_matrix(graph, i, j);
            if (weight != MAX_EDGE_VALUE && weight < local_min_weight) {
                local_min_weight = weight;
                local_min_id = j;
            }
        }

        lightest_edges[i - MY_NODES_FROM] = local_min_id;
    }
}
```

Figure 1: Code Snippet analysed for data dependencies

4 Results

4.1 Benchmark dataset

The performance of the algorithm was evaluated using randomly generated graphs. A custom program was developed to generate these graphs, allowing control over the number of nodes (n) and edges (m). To efficiently generate large graphs, the implementation utilises *OpenMP*. The generated graphs are guaranteed to be connected, and edge weights are assigned randomly. Three dense graphs were created for evaluation, containing 20.000, 40.000, and 80.000 nodes, respectively. "Dense" in this context refers to a high concentration of edges, approximately following the relationship $m = \frac{n^2}{4}$.

The graph generation program, written in C, is provided in the file `graph_generator.c`. A shell script, `generate_graph.sh`, automates the graph generation process for a given array of node counts. These custom graphs were necessary because existing datasets did not meet our specific requirements, namely the desired number of nodes, density, and connectivity.

The serial version of the algorithm used in the following comparison is simply the parallel version with 1 process, with the MPI and OpenMP directives removed. Although there are other serial versions, we felt that they would not be a fair comparison to our implementation because we know it can be made more efficient.

4.2 Setup parameters

The benchmark suite consists of five test suites, each containing three tests, one for each graph size (resulting in a total of 15 tests). Each test suite varies in the number of MPI processes and OpenMP threads used, maintaining a ratio of 1 MPI process to 2 OpenMP threads. The number of MPI processes used were 2, 4, 8, 16, and 32.

Within each test suite, the tests were run in a specific order: the serial implementation was executed first, followed by the parallel version. This order was chosen so that both the serial and parallel runs within a test would utilise the same compute node, minimising the impact of potential unknown factors and improving the reliability of the measurements.

The entire testing process was automated by a benchmark bash script. Furthermore, a Makefile was developed to automate compilation, job submission to the PBS queue, log file cleanup, active job monitoring, and, if necessary, job deletion from the PBS queue.

```
#!/bin/bash
#PBS -l select=1:ncpus=NCPUS:mem=MEMGB
#PBS -l walltime=2:00:00
#PBS -q short_cpuQ
```

Figure 2: PBS command used for each of the 5 test-suites

4.3 PBS directives

Each test used similar PBS directives. A single compute node was allocated, and the total number of CPUs (*ncpus*) was set to the product of the number of MPI processes and the number of OpenMP threads per MPI process. A wall time of approximately two hours was used due to the time required to read each graph. This read time is attributed to the graph representation size (1.38GB for 20,000 nodes, 5.8GB for 40,000 nodes, and 24GB for 80,000 nodes) and the cluster's relatively slow performance during intensive I/O operations. The allocated memory was sufficient to hold the graph representation in memory.

4.4 Performance evaluation

The images 1(a), 1(b), 1(c) show the performances of the parallel program during our simulations. We conducted the simulations multiple times and calculated the average of the outcomes. The speedup graph 1(a) illustrates that after 16 MPI cores, the rate of speedup start to decrease. The efficiency calculated considering only the cores fully dedicated to MPI can be seen in figure 1(b).

The efficiency decreases significantly as the number of utilised MPI processes increases. This rapid decline likely stems from several contributing factors. These include the current flat topology of the MPI processes, which contrasts with a more efficient tree-like structure that could minimise message passing as the process count increases. Performance may also be impacted by loops that iterate over all nodes; for large node counts, these loops can become a bottleneck. Finally, other, as yet unidentified, parts of the code may not scale efficiently, further contributing to the performance degradation with increasing core counts.

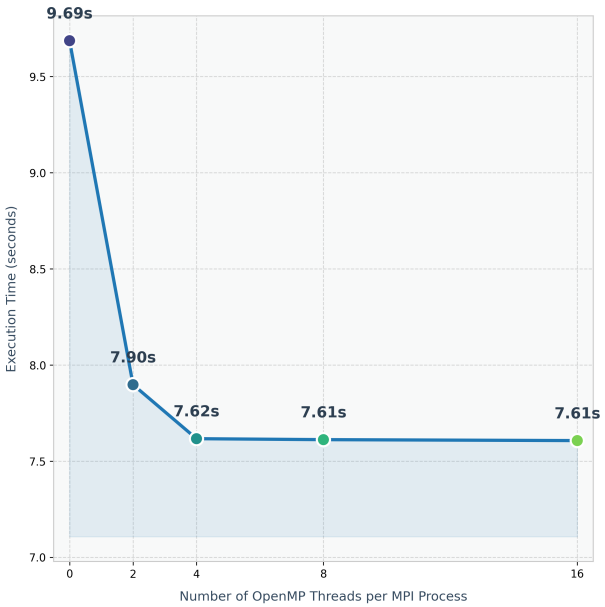


Figure 3: OpenMP scaling on 8 MPI processes

Additionally, we chose to compute the efficiency in terms of the total number of cores utilised, including those utilised by openMP as well as those fully dedicated to MPI processes. This obviously diminishes efficiency, as seen in figure 1(c). The majority of the work is performed by the MPI process; hence, the MPI cores are utilised, whereas OpenMP threads are instantiated just during the execution of OpenMP for loops, contributing less to the overall calculation. We believe that the efficiency that only takes into account MPI cores is more representative of the computation's true efficiency than this final representation.

Figure 3 illustrates the scaling behaviour of OpenMP with respect to the number of threads allocated per task. This particular experiment used a graph of 40,000 nodes and 8 MPI processes. As the figure shows, there is no significant performance gain beyond four threads. It appears that the computational benefits of additional threads are offset by the overhead of thread creation and result aggregation. Theoretically, if we increase again the number of threads used, we are going to face a time loss.

5 Limitations and future works

The majority of the limits encountered are given by some implementation decision we made to ease the implementation. One of these is the quantity of messages that the MPI processes send, which occasionally result in more data being sent than is required. Another issue that might be resolved is the quantity of processes and divisible vertices. Additionally, by increasing the code's efficiency, there may be room for improvement in terms of scalability. Finally, testing revealed a significant (10-20%) performance variability between runs with identical PBS directives and parameters on the HPC cluster. This inconsistency likely stems from factors such as resource contention (e.g., concurrent jobs on the same node) or the specific hardware assigned to the job (e.g., older machines).

References

[1] A. Mariano, A. Proenca, and C. Da Silva Sousa, "A generic and highly efficient parallel variant of boruvka's algorithm," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 610–617.

[2] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 408–423, 1989.

[3] PDC Center for High Performance Computing, "Scalability: Strong and weak scaling," https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/, 2018.