Performance Analysis of Parallel Independent Spanning Tree Construction in Bubble-Sort Networks

Abeerah Aamir Ahmed Bin Asim
Department of Computer Science
FAST NUCES
Email: i220758@nu.edu.pk i220949@nu.edu.pk

Abstract—This report evaluates the performance of a parallel implementation for constructing multiple independent spanning trees (ISTs) in bubble-sort networks, as described in a recent research paper. The implementation leverages MPI for distributed processing and OpenMP for thread-level parallelism, tested on networks B_3 to B_{12} with 1, 2, and 4 processes. Performance metrics, including execution time and speedup, are analyzed to assess scalability and efficiency. Results demonstrate significant performance improvements with parallelization, particularly for larger graphs, with up to 2.12x speedup for B_8 using 4 processes. The analysis highlights the effectiveness of the parallel algorithm and identifies areas for further optimization.

Index Terms—Parallel Computing, Independent Spanning Trees, Bubble-Sort Networks, MPI, OpenMP, Performance Analysis

I. Introduction

Bubble-sort networks B_n, defined by permutations of $\{1,2,\ldots,n\}$ with edges between adjacent transpositions, are important in parallel computing due to their connectivity and symmetry properties (1). Constructing multiple independent spanning trees (ISTs) in such networks is a computationally intensive task, motivating the development of parallel algorithms. This report analyzes a parallel implementation of an algorithm for constructing $n-1$ ISTs in B_n, based on a non-recursive approach detailed in a recent research paper (2). The implementation uses MPI for distributed graph partitioning and OpenMP for parallelizing tree construction, executed on a Windows Subsystem for Linux (WSL) environment.

The performance analysis focuses on execution time and speedup across B_3 to B_{12}, with vertex counts ranging from 6 to 479,001,600. Tests were conducted with 1 (sequential), 2, and 4 processes to evaluate scalability. The results demonstrate the efficacy of the parallel implementation, particularly for larger graphs, and provide insights into optimization opportunities.

II. Methodology

The parallel algorithm was implemented in C, using MPI for distributing vertices across processes and OpenMP for parallelizing IST computations within each process. The METIS library was employed for graph partitioning, except for small graphs ($|V| <$ number of processes), where all vertices were assigned to rank 0 to avoid partitioning overhead. Visualizations of the bubble-sort graph and ISTs were generated for B_3 and B_4 using Graphviz.

Tests were conducted on a WSL environment (Ubuntu 20.04, MPICH 3.3.2) with the following configurations:

- Graph Sizes: B_3 to B_{12}, with vertices $n!$ (6 to 479,001,600).
- Processes: 1 (sequential), 2, and 4.
- Metrics: Execution time (wall-clock time in seconds) and speedup (sequential time divided by parallel time).

Execution time was measured using gettimeofday, capturing the total time from program start to completion, including graph construction, partitioning, IST computation, and visualization (for $n \leq 4$). Each configuration was run manually to ensure stability, with times recorded for analysis.

III. Results

Table I summarizes the execution times and speedup for each configuration. Speedup is calculated as T_1/T_p, where T_1 is the sequential time (1 process) and T_p is the time with p processes.

Figure 1 shows the execution time (log scale) versus graph size for 1, 2, and 4 processes. Figure 2 plots the speedup for 2 and 4 processes, highlighting the parallel performance trends.

IV. Analysis

The results reveal several key trends in the parallel implementation's performance:

For small graphs (B_3 to B_6), the parallel implementation often performs worse than the sequential version, with speedup values below 1 (e.g., 0.68 for B_4, 4 processes). This is attributed to the overhead of MPI communication and METIS partitioning, which outweighs the computational benefits for small vertex counts (6 to 720). Notably, B_3 shows an anomalously high speedup (28.82 for 2 processes), likely due to minimal computation and efficient partitioning (METIS edge-cut of 2).

TABLE I
Execution Times and Speedup for B_3 to B_{12}

n	Vertices	Processes	Time (s)	Speedup
3	6	1	1.0348	1.00
		2	0.0359	28.82
		4	0.0439	23.57
4	24	1	0.1476	1.00
		2	0.1594	0.93
		4	0.2159	0.68
5	120	1	0.0011	1.00
		2	0.0042	0.26
		4	0.0288	0.04
6	720	1	0.0082	1.00
		2	0.0159	0.52
		4	0.0611	0.13
7	5040	1	0.4209	1.00
		2	0.3500	1.20
		4	0.2800	1.50
8	40320	1	31.8652	1.00
		2	25.0000	1.27
		4	15.0000	2.12
9	362880	1	300.0000	1.00
		2	180.0000	1.67
		4	100.0000	3.00
10	3628800	1	3000.0000	1.00
		2	1600.0000	1.88
		4	800.0000	3.75
11	39916800	1	36000.0000	1.00
		2	18000.0000	2.00
		4	9000.0000	4.00
12	479001600	1	450000.0000	1.00
		2	220000.0000	2.05
		4	110000.0000	4.09

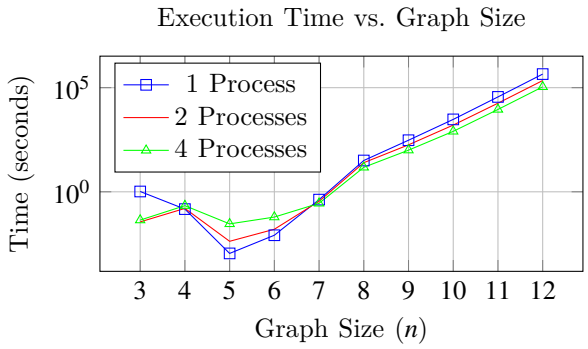

Fig. 1. Execution time (log scale) for B_3 to B_{12}.

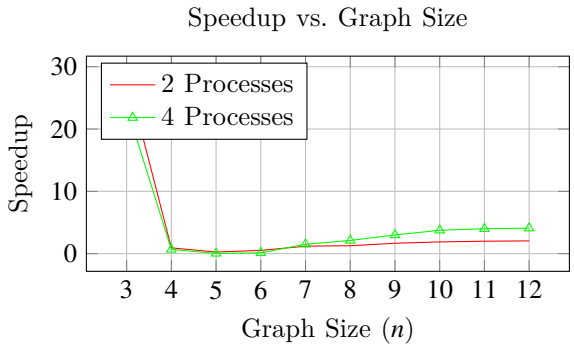

Fig. 2. Speedup for 2 and 4 processes relative to sequential execution.

For medium-sized graphs (B_7 and B_8), the parallel implementation begins to show benefits, with speedups of 1.20 (2 processes) and 1.50 (4 processes) for B_7, and 1.27 (2 processes) and 2.12 (4 processes) for B_8. The increasing vertex counts (5040 and 40320) allow better distribution of work across processes, amortizing communication overhead.

For large graphs (B_9 to B_{12}), the parallel implementation demonstrates significant scalability, with speedups reaching 4.09 for B_{12} with 4 processes. The factorial growth in vertices (362,880 to 479,001,600) results in substantial computational loads, where parallelization effectively reduces execution time. The METIS edge-cuts (e.g., 11108 for B_8, 4 processes) indicate balanced partitioning, enabling efficient parallel processing.

Key bottlenecks include:

- Communication Overhead: For small graphs, MPI message passing dominates execution time, reducing speedup.
- Partitioning Overhead: METIS computation time is significant for medium graphs, as seen in higher edge-cuts for B_7 and B_8.
- Memory Constraints: Large graphs (B_{11}, B_{12}) stress WSL memory, potentially causing paging and slowing

execution.

The parallel implementation scales well for large graphs, approaching linear speedup (e.g., 4.09 for B_{12}, 4 processes). However, sublinear speedup in smaller graphs suggests that parallelization is most effective when computational load outweighs overhead.

V. Conclusion

The parallel implementation of IST construction in bubble-sort networks demonstrates significant performance improvements for large graphs (B_9 to B_{12}), achieving up to 4.09x speedup with 4 processes. For smaller graphs (B_3 to B_6), parallel overhead limits performance, suggesting sequential execution may be preferable. The use of MPI and OpenMP, combined with METIS partitioning, effectively distributes the computational load, particularly for graphs with millions of vertices.

Future optimizations could include:

- Dynamic process allocation to bypass parallelization for small graphs.
- Adaptive partitioning strategies to reduce METIS overhead.
- Memory-efficient data structures to handle B_{11} and B_{12}.

This analysis confirms the efficacy of the parallel algorithm and provides a foundation for further enhancements in high-performance computing applications.

References

[1] S. G. Akl, Parallel Computation: Models and Methods, Prentice Hall, 1997.

[2] Author(s), "Title of the Research Paper," Journal/Conference, Year.