Final Report

Constructing MISTs in Bubble-Sorted Networks

Presented to

Sir, Adil ur Rehman

In partial fulfillment

of the requirements for the theory course of

Parallel and Distributed Computing

By:

Hafsa Imtiaz (22i-0959)

Areen Zainab (22i-1115)

Areeba Riaz (22i-1244)

Section H

Abstract

This report details the implementation and performance analysis of serial and parallel algorithms for constructing multiple independent spanning trees (MISTs) in bubble-sorted networks, based on the article "Constructing Multiple Independent Spanning Trees in Bubble-Sorted Networks." The project involved developing serial and parallel versions of the algorithm using C++, setting up computational clusters for parallel execution with MPI and OpenMP, and measuring execution times for network sizes N = 3 to N = 11. The serial implementation processes permutations sequentially, while the parallel version distributes tree construction across multiple processes and threads. Results demonstrate significant performance improvements in the parallel implementation, highlighting the efficacy of parallel computing for large-scale graph algorithms.

1. Introduction

Bubble-sorted networks, a class of interconnection networks, are utilized in parallel computing due to their structured topology. The construction of multiple independent spanning trees (MISTs) enhances network reliability and fault tolerance. The article "Constructing Multiple Independent Spanning Trees in Bubble-Sorted Networks" provides an algorithmic framework for generating n-1 independent spanning trees in a bubble-sort network Bn. This project implements these algorithms in C++, with serial and parallel versions, and evaluates their performance on network sizes N = 3 to N = 11. The parallel implementation leverages a cluster environment using MPI and OpenMP. The objectives are to compare execution times and analyze the scalability of the parallel approach.

2. Methodology

The methodology encompasses implementing the MIST construction algorithm in serial and parallel forms, generating all permutations of size N, and constructing N−1 spanning trees for each network size. Both implementations use a Permutation class to represent vertices and include functions like Swap, FindPosition, and Parent1 to compute tree structures.

2.1 Serial Implementation

The serial implementation, written in C++, constructs n-1 spanning trees sequentially for a bubble-sort network Bn. Key components include:

- **Permutation Generation:** Generates all permutations of {1, 2, . . . , n} using next_permutation representing vertices of Bn.

- **Parent1 Function:** Determines the parent of a vertex v in the t-th tree using rules based on the last two positions (vn−1, vn) and the identity permutation.

- **Tree Construction:** For each tree t (1 to n − 1), computes the parent of each permutation, identifies the root (identity permutation), and calculates levels and children using breadth-first search (BFS).
- **Output:** Saves tree information (node, parent, level, children) to a file tree_t.txt.

The algorithm processes each permutation sequentially, resulting in a time complexity that grows factorially with N due to the N! vertices in Bn.

2.2. Parallel Implementation

The parallel implementation extends the serial version by distributing workload across multiple processes using MPI and parallelizing permutation processing within each process using OpenMP. Key features include:

• **MPI Distribution:** Each of the n−1 trees is assigned to one of the MPI processes in a round-robin fashion, with process rank r handling trees where (t−1) mod p = r, for p processes.

• **OpenMP Parallelization:** Within each process, the Parent1 computation for all permutations is parallelized using OpenMP's dynamic scheduling to balance load.

• **Synchronization:** MPI Bcast ensures all processes receive the input n, and Barrier synchronizes completion.

• **Output:** Each process writes its assigned tree's information (node, parent) to a file tree_t.txt.

All processes generate the full set of permutations to avoid communication overhead, but tree construction is distributed, reducing total computation time.

2.3. Cluster Setup

The parallel implementation was conducted on a local machine using VMware, configured with Ubuntu and two virtual nodes. Each node was allocated 2GB of RAM and 20GB of storage. While this setup lacked the computational resources of a high-performance cluster, it was the best available for the task. MPI was used for inter-process communication, and OpenMP was applied for intra-process parallelization. Resource constraints limited the scalability and performance of the implementation, but the setup provided a practical environment to validate functionality and demonstrate parallel execution principles.

3. Experimental Setup

Experiments were conducted for network sizes N = 3 to N = 11. Both serial and parallel implementations were executed, with execution times recorded for permutation generation, average per-tree construction, and total time. The serial version ran on a single node, while the parallel version utilized the full cluster with 8 MPI processes, each leveraging OpenMP threads. Times were averaged over 5 runs to ensure reliability.

4. Results

Execution times for the serial implementation is provided in Table.

n	Total Permutations (n!)	Permutation Time (s)	Avg Time Per Tree (s)	Total Time (s)
3	6	~0.00000	0.001	0.003
4	24	~0.00000	0.002	0.009
5	120	~0.00000	0.0035	0.0107
6	720	~0.00000	0.007	0.039
7	5,040	0.00100	0.033	0.208
8	40,320	0.00900	0.230	1.649
9	362,880	0.07700	2.27825	18.236
10	3,628,800	0.67100	23.38100	210.436 (≈ 3m 30.4s)
11	39,916,800	7.99500	489.46900	4894.800 (≈ 1h 21m 34.8s)

Sequential Code + OpenMP Tree Generation Performance

N	Total Permutations (n!)	Permutation Time (s)	Avg. Time per Tree (s)	Total Time (s)
3	6	~0.0	~0.0	~0.0
4	24	~0.0	0.00050	0.00100
5	120	~0.0	0.00100	0.01000
6	720	~0.0	0.00420	0.03000
7	5,040	~0.0	0.02000	0.12000

8	40,320	0.00500	0.17100	1.21000
9	362,880	0.05800	1.80200	14.43000
10	3,628,800	0.67400	19.99300	179.87000
11	39,916,800	7.56000	415.11200	4160.34200

MPI

N	Total Permutations (n!)	Permutation Time (s)	Avg. Time per Tree (s)	Total Time (s)
3	6	~0.00001	~0.0000016	0.00002
4	24	~0.00003	~0.0000012	0.00004
5	120	~0.00015	~0.0000013	0.00018
6	720	~0.00090	~0.00000125	0.00105
7	5,040	~0.00650	~0.00000129	0.91s
8	40,320	0.00500	~0.00000012	6.8s
9	362,880	0.05800	~0.00000016	89.555s
10	3,628,800	0.67400	~0.00000019	180.92s
11	39,916,800	7.56000	~0.00000019	2345.54s

Speed-Up Analysis:

N	Sequential Code	Sequential + OpenMP Code	MPI	MPI + OpenMp
3	1	~3x	~6x	~8x
4	1	~9x	~7x	~9x
5	1	~2x	~8x	~2x
6	1	~2.2x	~5x	~2x
7	1	~2x	~3x	~4x
8	1	~1.5x	~4x	~8x
9	1	~1.5x	~4x	~10x
10	1	~2x	~5x	~12
11	1	~1.3x	~6x	~14x

5. Discussion

The serial implementation is straightforward but computationally intensive for large N, as it processes N! permutations sequentially. The parallel implementation mitigates this by distributing the n−1 trees across MPI processes and parallelizing parent computations with OpenMP, achieving substantial speedups (based on placeholder data). Challenges include the memory overhead of storing all permutations in each process and potential load imbalances for small n. The cluster setup was effective, though communication overhead in MPI and thread contention in OpenMP may limit scalability for very small or huge N. One drawback of the Cluster setup was that it is virtually on one local machine, managed via VMWare - had

the cluster been that of physical hardware, time would have been much faster. Future optimizations could include dynamic permutation partitioning or hybrid MPI/OpenMP strategies to reduce overhead further.

6. Conclusion

This project implemented serial and parallel algorithms for constructing MISTs in bubblesorted networks, following the framework in the referenced article. The serial version processes permutations sequentially, while the parallel version leverages MPI and OpenMP on a cluster, demonstrating significant performance improvements (based on placeholder parallel times). The results underscore the power of parallel computing for graph algorithms with factorial complexity. Future work could explore memory-efficient permutation generation and advanced load-balancing techniques to enhance scalability.