Parallel & Distributed Computing

Project Report

A parallel algorithm for constructing multiple independent spanning trees in bubble-sort networks

Group Members:
22i-1013 Mustafa Kamran
22i-0885 Malaika Afzal
22i-1074 Fatima Bakhsh

Section K

A parallel algorithm for constructing multiple independent spanning trees in bubble-sort networks

Overview

This project investigates a parallel algorithm to construct multiple independent spanning trees (ISTs) in bubble-sort networks, evaluating its performance under different implementations:

- **Sequential (Python)**
- **MPI-based Distributed (Python + METIS)**
- **Hybrid MPI + OpenMP (C++)**

We focus on:

- Performance and scalability comparisons
- Bottleneck identification
- Optimization using METIS for partitioning and TAU for profiling

1. Algorithm Implementations

1.1 Sequential (Python)

- **Time Complexity:** $O(n! \times n)$
- **Space Complexity:** $O(n! \times n)$
- **Key Functions:**
    - generate_bn(n)
    - parent1(v, t, n, inv, rpos)
    - build_trees(n)

Findings:

- Suitable for **n ≤ 4**
- Performance degrades factorially with increasing n
- Inefficient for larger graphs due to memory usage and computation time

1.2 MPI Implementation (Python + METIS)

- **Parallelism:** Distributed-memory model
- **Partitioning:** METIS used for graph-based load balancing
- **Key Functions:**
    - write_metis_file()
    - run_metis()
    - optimize_partitioning()

Findings:

- Optimal performance when **partitions (k) ≈ process count**
- Communication and inter-partition edges increase overhead
- Best for **medium-sized problems (n = 5–6)**

1.3 Hybrid MPI + OpenMP (C++)

- **Parallelism:** Combines distributed and shared-memory parallelism
- **Key Enhancements:**
    - OpenMP for intra-node parallelism
    - MPI for inter-node communication
    - TAU profiler used for identifying communication and cache bottlenecks

Findings:

- Best performance for **large n (≥ 7)**
- Reduced communication overhead and better memory locality
- Performance scales well with threads and processes

2. Experimental Configuration

Test Type	Parameters
Sequential	n = 2 to 7
MPI	n = 2 to 7, processes = 2–4, k = 2–4
Hybrid	n = 5–7, processes = 1–4, threads = 2–6, k = 3–5

3. Performance Metrics

- **Execution Time**
- **Speedup:** Speedup= T_1 / T_p
- **Efficiency:** Efficiency= Speedup / p
- **Partitioning Efficiency**
- **Cache Metrics (L1, L2, L3)**
- **Instruction & Cycle Counts**
- **Call Path Profiling (TAU)**

4. Scalability Analysis

4.1 Strong Scaling

- Measures improvement with more resources at constant problem size
- Efficiency drops with excess parallelism due to communication overhead

4.2 Weak Scaling

- Measures efficiency as problem size increases with resources
- Hybrid model shows better sustained performance

5. Implementation Comparison

Feature	Sequential	MPI	Hybrid (MPI + OpenMP)
Best Use Case	$n \leq 3$	$n = 4–6$	$n \geq 7$
Parallelism	None	Distributed	Hybrid
Scalability	✗	Moderate	High
Memory Usage	Very High	Moderate	Efficient
Communication Overhead	None	High	Low
Code Complexity	Low	Medium	High

6. Detailed Performance Analysis

6.1 Small Problems ($n < 4$)

- Sequential performs best
- Parallel overhead dominates in MPI and Hybrid

6.2 Medium Problems ($n = 4–6$)

- MPI shows 2x–4x speedup
- Hybrid reaches up to 6x speedup due to thread-level optimization

6.3 Large Problems ($n \geq 7$)

- Sequential is impractical

- MPI is memory-intensive
- Hybrid offers highest throughput and scalability

7. Optimization Insights

7.1 Partitioning

- **METIS** enhances load balancing
- Ideal when **k = number of MPI processes**
- Fewer inter-partition edges yield better performance

7.2 Threading & Memory

- OpenMP improves cache locality and reduces memory traffic
- Use 2–4 threads for medium n, 4–6 for large n
- Shared-memory use improves efficiency

7.3 Profiling (TAU)

- Identified hotspots in MPI communication
- Cache performance improves significantly with hybrid model
- Revealed optimal process/thread combinations

8. Recommendations

Problem Size	Recommended Implementation	Notes
$n \leq 3$	Sequential	Minimal overhead
$4 \leq n \leq 6$	MPI	Ensure good partitioning
$n \geq 7$	Hybrid (MPI + OpenMP)	Best resource utilization

Resource Configuration Tips:

- Match **MPI processes to partitions**
- Use **2–6 OpenMP threads per process**
- Optimize based on cache and core count

9. Conclusion

The **Hybrid MPI + OpenMP implementation** demonstrates superior scalability, memory efficiency, and runtime performance, especially for large-scale problems. The combination of METIS-based partitioning and TAU-guided tuning enables significant optimization across both computation and communication layers.

The correct choice of implementation depends on:

- Problem size
- Hardware architecture
- Available memory and cores

By carefully selecting configurations and tools, our algorithm can scale effectively from small systems to high-performance clusters.