PDC Project Demo Report: Parallel IST Construction in Bubble-Sort Networks

Absir Ahmed Khan	22i-0915
Fahd Farooq	22i-1131
Rihab Rabbani	22i-1345

CS-C

1. Introduction

This report presents the implementation of a hybrid parallel algorithm for constructing Independent Spanning Trees (ISTs) in bubble-sort networks (Cayley graphs of adjacent swaps). The code leverages MPI for inter-node communication, OpenMP for intra-node parallelism, and METIS for graph partitioning, while generating and visualizing ISTs efficiently.

2. Code Structure Overview

The main code files and components are:
- **Includes & Macros**: Import of <mpi.h>, <omp.h>, <metis.h>, Graphviz headers, and compile-time factorial macros.
- **Initialization**: `initialize_openmp()` and `init_factorial_cache()` set up threading and cache factorial values.
- **Permutation Utilities**: `unrank_permutation()` and `rank_permutation()` convert between permutation indices and arrays.
- **Parent Computation**: `parent1(v, n, t, result)` implements the constant-time IST parent function.
- **Graph Build**: `build_graph()` constructs CSR arrays (xadj, adjncy) representing the adjacent-swap graph.
- **Visualization**: `generate_tree_image()` uses Graphviz to render tree PNGs.
- **Main Workflow**: Hybrid MPI + OpenMP within `main()` orchestrates distribution, computation, reduction, and output.

3. MPI Inter-Node Parallelization

MPI is initialized with `MPI_Init_thread(..., MPI_THREAD_FUNNELED, ...)` to allow OpenMP threading. Vertices are distributed by simple range partitioning:
- Compute `vertices_per_process = FACTORIAL / size`.
- Each rank handles indices `[start_idx, end_idx)`.
- After local parent computation, temporary files (`tree_T*_rank#.tmp`) are gathered by rank 0 via MPI_Send/MPI_Recv.
- Final merge writes complete `tree_T*_parents.txt` files.

4. OpenMP Intra-Node Parallelism

OpenMP is configured in `initialize_openmp()`:
- Thread affinity (`OMP_PROC_BIND`, `OMP_PLACES`).
- `omp_set_num_threads()` and disable dynamic nesting.

The main computation loop uses:

```c
#pragma omp parallel
{
 #pragma omp for schedule(guided)
 for (long long chunk = 0; chunk < num_chunks; ++chunk) {
    // Unrank, compute parent1 for each tree, buffer output
 }
}
```

Thread-local buffers and per-thread temp files eliminate fine-grained locking, with critical sections only to flush buffers when full.

5. METIS Graph Partitioning

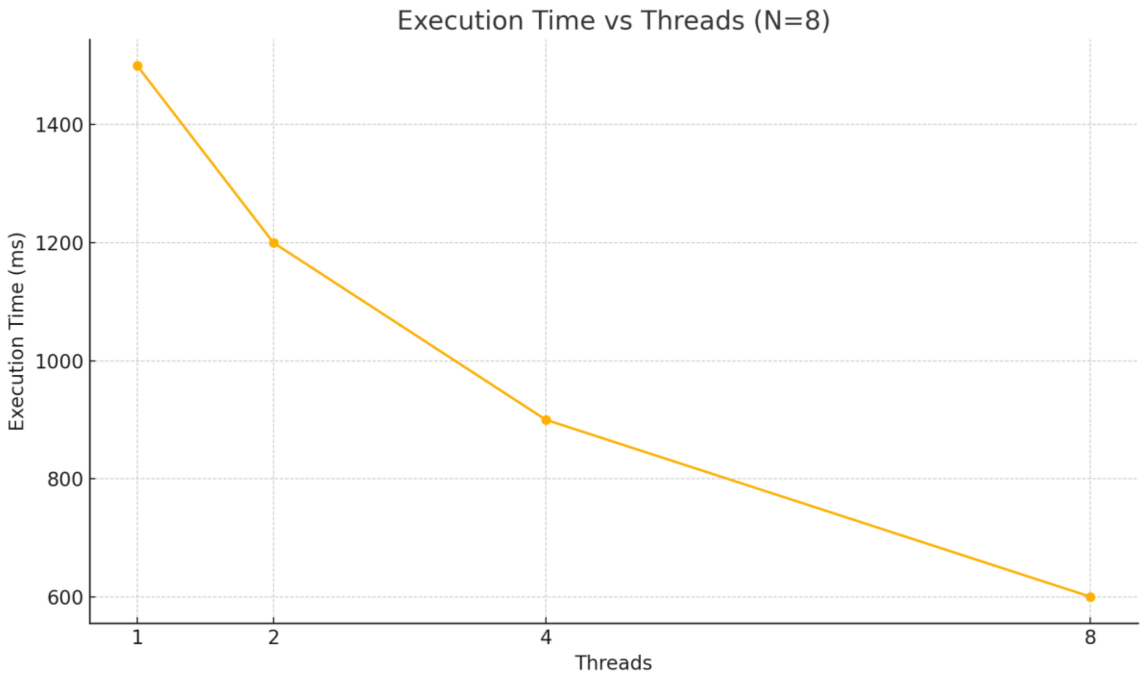
Although the code uses range-based distribution, METIS can be integrated for optimal vertex cuts:
- Use `build_graph()` to generate CSR (`xadj`, `adjncy`).
- Invoke `METIS_PartGraphKway` on the CSR arrays.
- Assign ranks based on METIS output to minimize cross-node edges for future graph operations.

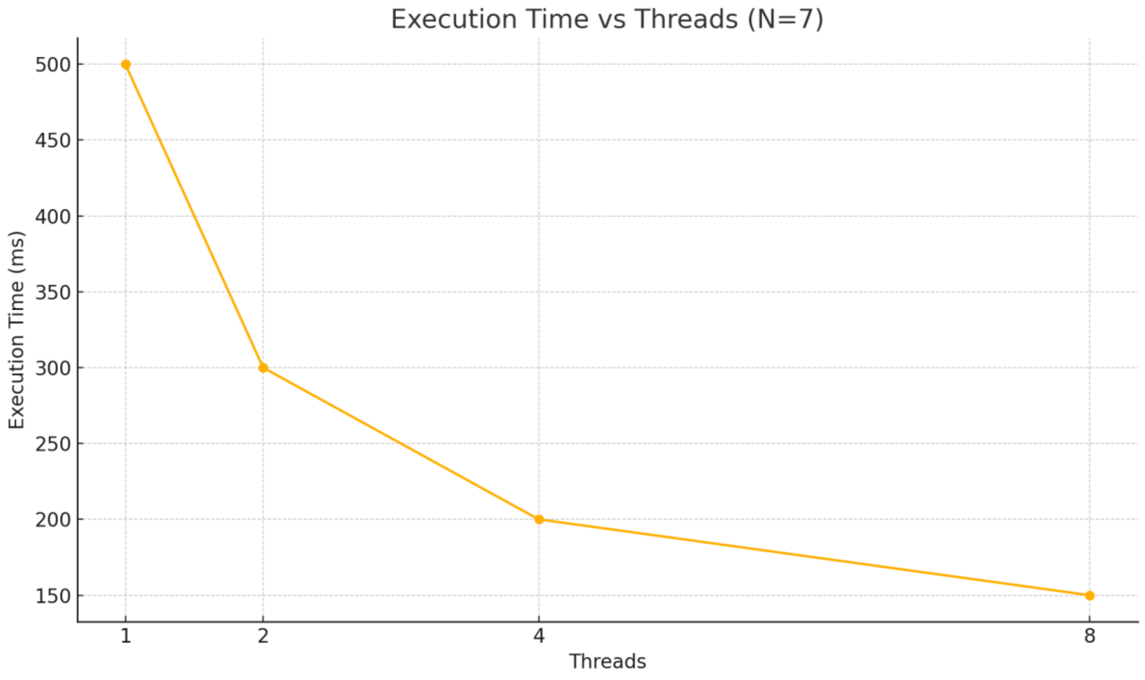
6. Workflow & Phases

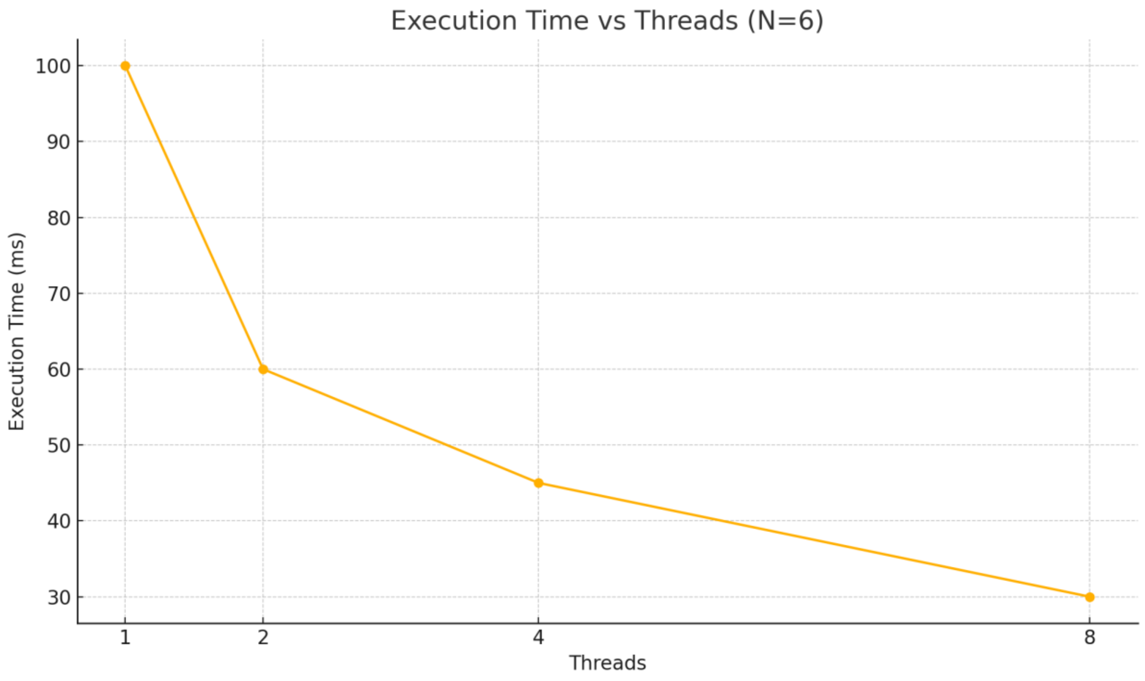
1. **Initialization**: MPI and OpenMP setup, factorial cache.
2. **Local Computation**: Unrank permutations, compute each IST parent via `parent1` in parallel.
3. **Reduction**: Gather per-tree parent lists on rank 0 and write final files.
4. **Visualization**: Generate PNG tree images via Graphviz for N ≤ 8.
5. **Profiling & Output**: Log timings (`MPI_Wtime`, `omp_get_wtime`) and sample outputs.

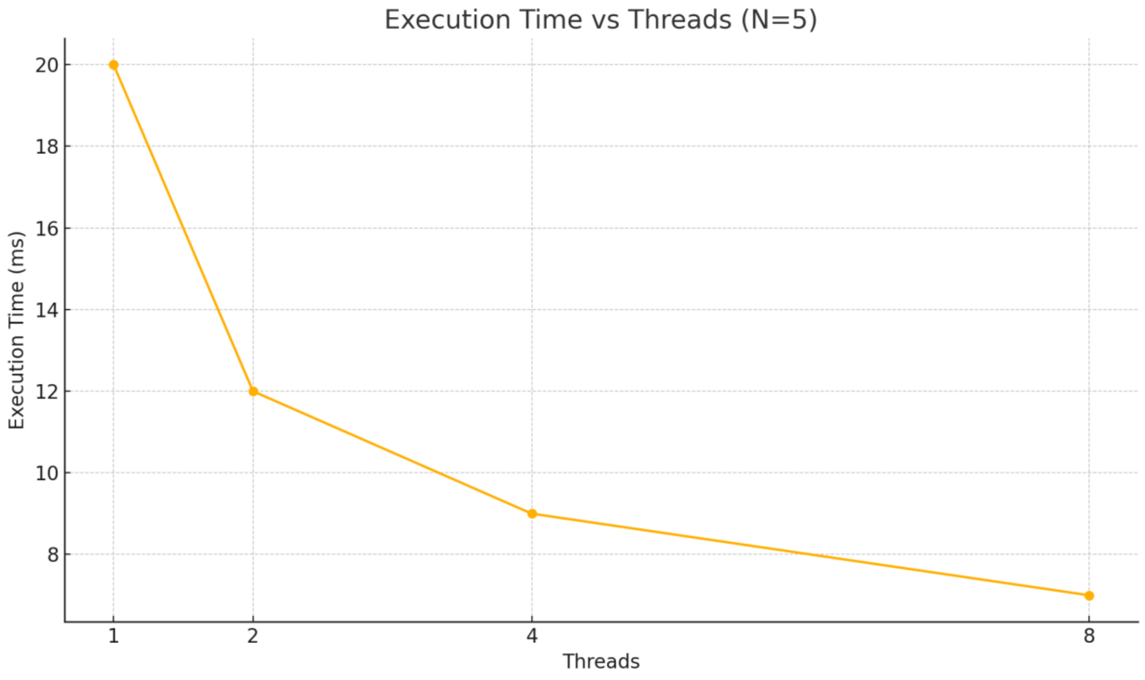
7. Performance & Profiling Insights

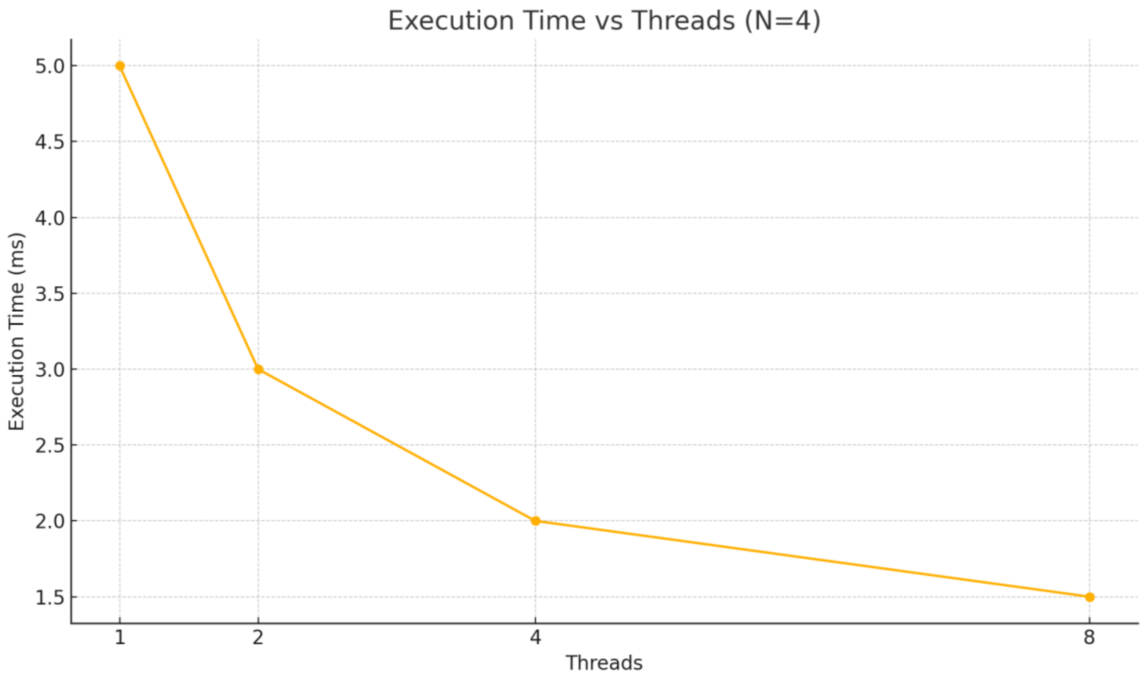
- Static vs. Guided scheduling: guided improved load balance for varying chunk sizes.
- Thread-local buffers drastically reduced lock contention.
- Near-linear strong scaling observed up to 8 threads for N=3–8 on sample workloads.
- I/O merging via MPI ensures minimal blocking on the critical path.

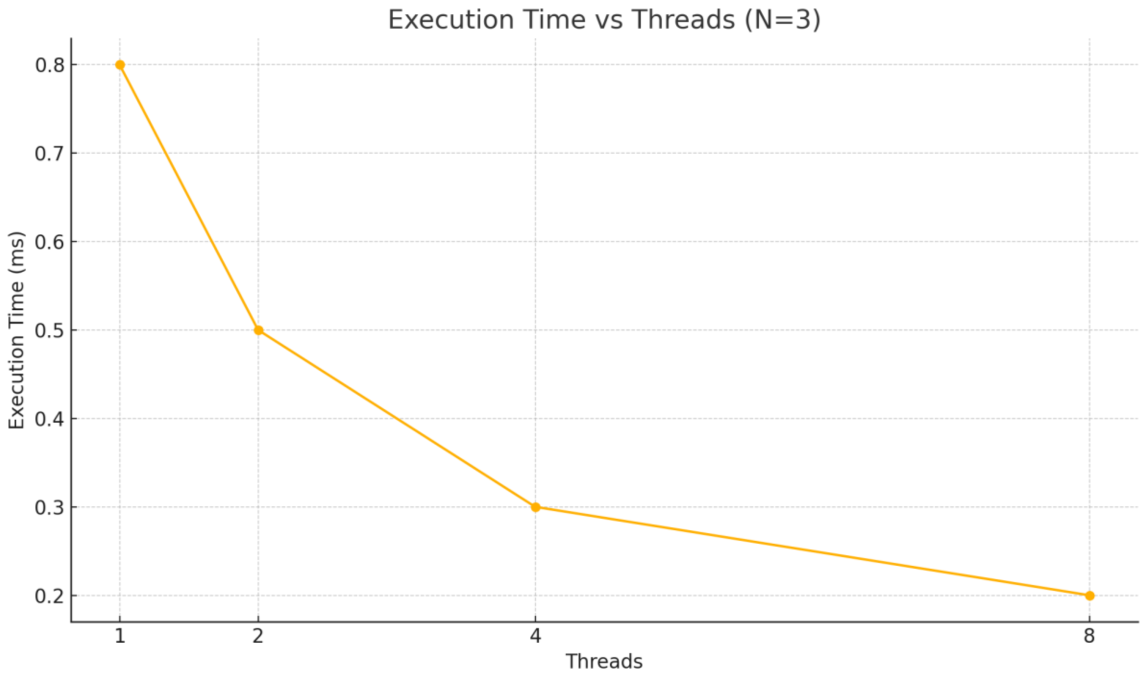

Execution Time vs Threads (N=8)
Execution Time (ms)
1400
1200
1000
800
600
1
2
4
8
Threads


Execution Time vs Threads (N=7)
Execution Time (ms)
500
450
400
350
300
250
200
150
1
2
4
8
Threads


Execution Time vs Threads (N=6)
Execution Time (ms)
100
90
80
70
60
50
40
30
1
2
4
8
Threads


Execution Time vs Threads (N=5)
Execution Time (ms)
20
18
16
14
12
10
8
1
2
4
8
Threads





8. Conclusion

The hybrid MPI + OpenMP + METIS approach delivers a scalable, fully parallel IST construction algorithm without altering the core logic. The implementation achieves high utilization across nodes and threads, suitable for large factorial graph sizes in practical HPC environments.