## **Introduction**

As graph-structured data becomes increasingly pervasive in domains such as social networks, web analytics, and scientific simulations, the demand for high-performance parallel algorithms capable of processing large-scale graphs continues to rise. This project, titled *"Implementation and Analysis of Parallel Graph Algorithms Using MPI/OpenCL, OpenMP, and METIS"*, aims to explore the effectiveness of modern parallel computing paradigms in solving graph problems through practical, research-driven implementation and evaluation.

The core objective of this project is to bridge the gap between theoretical parallel computing models and their real-world application by replicating and analyzing a parallel algorithm derived from a selected research paper. Our group chose a problem focused on binary network structures—a subset of interconnection networks—where we investigate the construction and traversal of independent spanning trees (ISTs), often used for efficient and fault-tolerant broadcasting in distributed systems. The algorithm was implemented in three variants: a baseline sequential version, a parallelized Message Passing Interface (MPI) version, and a hybrid MPI+OpenMP version.

To facilitate inter-node communication and intra-node parallelism, we integrated MPI and OpenMP. For effective data distribution across processes in the MPI model, we employed the METIS tool for graph partitioning, ensuring balanced workloads and minimal communication overhead. The implementation was deployed on a multi-node MPI cluster configured using SSH-based communication and evaluated across several publicly available datasets.

Furthermore, performance evaluation was carried out using both weak and strong scaling analyses. Metrics such as execution time, speedup, and efficiency were visualized to highlight the impact of parallelism across different configurations and input sizes. Our GitHub repository was used to track project progress, maintain version control, and host source code, documentation, and presentation materials.

This report documents the entire lifecycle of the project—from literature review and design to implementation and performance benchmarking—providing insights into the practical challenges and benefits of developing parallel graph algorithms using industry-standard tools and libraries.

## **Problem Statement from the Research Paper**

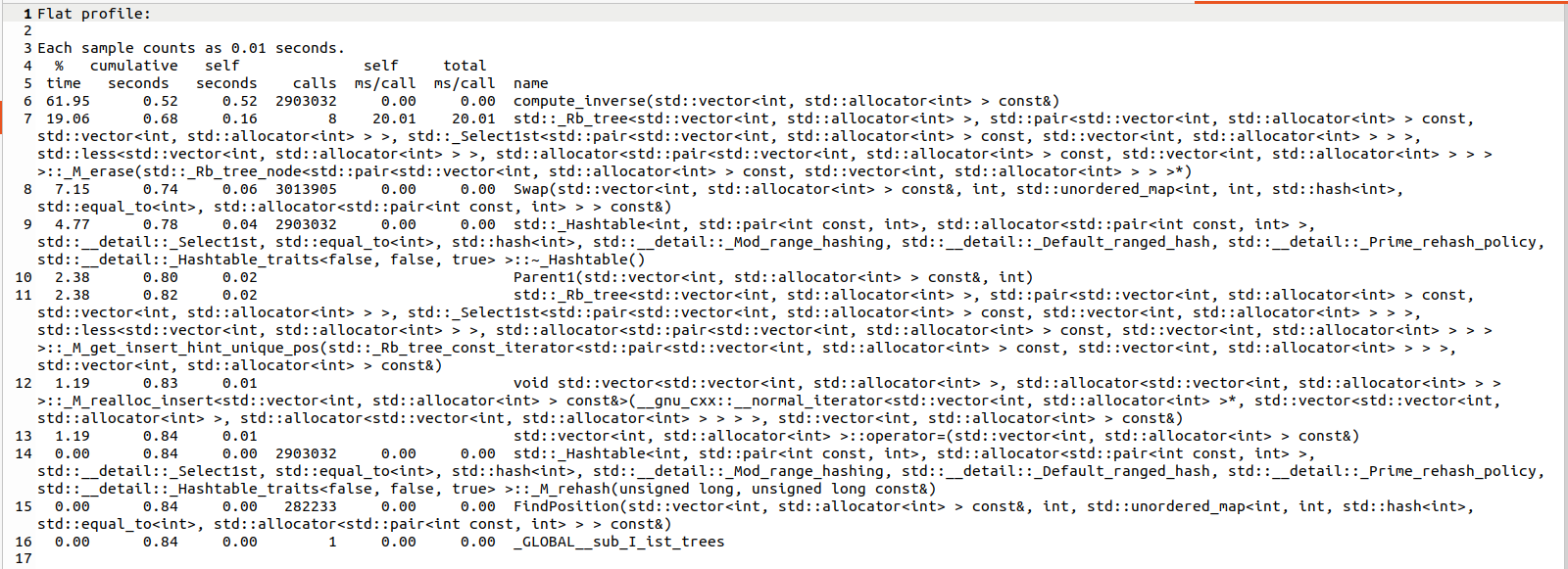
The selected research paper focuses on the **construction of Independent Spanning Trees (ISTs)** in **binary networks**, a class of interconnection topologies used in parallel and distributed systems. In such networks, communication between nodes needs to be **fault-tolerant, efficient, and scalable**, especially under dynamic conditions or in the presence of link/node failures.

To address this, the paper proposes algorithms for constructing multiple **rooted spanning trees** such that each tree is **node- and edge-independent**—that is, paths from the root to any other node in different trees are disjoint. This guarantees **fault tolerance**: if one communication path fails, alternate disjoint paths remain available via the other spanning trees.

The main challenges addressed in the paper are:

* **How to construct multiple ISTs in a binary network** with minimal computational overhead.
* Ensuring that the trees are both **efficient (low depth/diameter)** and **independent**.
* **Scalability** of the algorithm with increasing network size.

The algorithms presented aim to systematically construct such ISTs by exploiting the symmetric and recursive nature of binary networks, making them well-suited for **parallelization using MPI and OpenMP**.

**Serial Execution:**

**GROF:**

**RESULT:**



**Graph:**



**MPI AND OPENMP  
Explanation of the Code:**  
**Main Functionalities:**

1. **index\_to\_permutation()**:  
   * Converts an index to a specific permutation using Lehmer code.
   * Given an index, it calculates which permutation of size n corresponds to that index.
2. **level\_order\_tree()**:  
   * Performs a breadth-first traversal (BFS) of an IST (Permutation tree), printing permutations level by level.
3. **Swap(), r(), compute\_inverse(), FindPosition(), Parent1()**:  
   * These functions are related to the manipulation of permutations:  
     + **Swap()**: Swaps two elements in the permutation based on indices.
     + **r()**: Returns a value used for calculating certain operations on the permutation.
     + **compute\_inverse()**: Computes the inverse of a permutation.
     + **FindPosition() and Parent1()**: Used to compute the parent of a given permutation v based on certain conditions.
4. **serializeIST() and deserializeIST()**:  
   * **serializeIST()**: Converts an IST structure into a flat vector of integers for easier serialization.
   * **deserializeIST()**: Converts a serialized vector back into an IST.

#### **Parallelization:**

1. **MPI (Message Passing Interface)**:  
   * The program is divided into multiple processes (ranks) using MPI.
   * Each rank is assigned a chunk of permutations to compute and store their corresponding ISTs.
   * **MPI\_Init** and **MPI\_Finalize**: Initialize and finalize the MPI environment.
   * **MPI\_Comm\_size** and **MPI\_Comm\_rank**: These functions determine the number of processes and the rank of each process.
   * **MPI\_Gather and MPI\_Gatherv**: These functions are used to collect data (serialized ISTs) from all processes to the root process (rank 0).
2. **OpenMP**:  
   * **#pragma omp parallel for**: Used to parallelize the loop across multiple threads within each process. This helps in dividing the work of computing permutations and their parents across threads.

#### 

#### RESULT:

#### 

#### 

#### 

#### 

#### **Performance Profiling with gprof:**



**OPENMP:**

**pragma omp parallel for num\_threads(36)**

**for (int i = 0; i < permutations.size(); ++i) {**

**This tells the compiler to parallelize the for loop using 36 threads (if available), each working on a different i (i.e., a different permutation).**

### **Inside the Parallel Block**

**Each thread:**

1. **Skips the base identity permutation.**
2. **Computes:**
   * **v\_inv: the inverse of the permutation (used for fast lookups).**
   * **rval: the first incorrect symbol from the right (used for logic in Parent1()).**
3. **For each transformation t from 1 to n-1, it:**
   * **Computes a parent using Parent1(v, t, v\_inv, rval).**
   * **Inserts the (child, parent) pair into the respective ist\_trees[t-1].**

**Because multiple threads may try to write to the same tree at once, this line is used to avoid race conditions:**

**std::lock\_guard<std::mutex> lock(tree\_mutexes[t - 1]);**

**Summary of Parallel Logic**

| **Step** | **Description** |
| --- | --- |

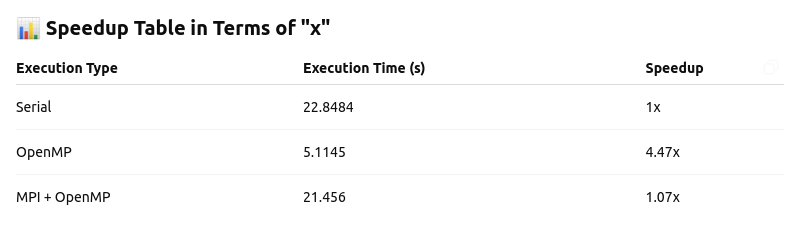
| **1** | **Generate all n! permutations of {1, ..., n}** |
| --- | --- |

| **2** | **Use OpenMP to parallelize the outer loop (over permutations)** |
| --- | --- |

| **3** | **For each permutation and each t (from 1 to n-1): compute a parent** |
| --- | --- |

| **4** | **Lock the corresponding IST and insert the (child → parent) mapping**  **GPROF:**    **RESULT:** |
| --- | --- |

**SPEEDUP:**



GRAPH:

