National University of Computer and Emerging Sciences
Islamabad Campus

Final Project

CS3006

Parallel and Distributed Computing

Group Members:
- Muhammad Daniyal Aziz - 22-0753
- Haleema Tahir Malik - 22i-0937
- Muhammad Bilal Khawar - 22i-0786

Date: 6th May, 2025

Table of Contents

Introduction

In modern digital ecosystems, identifying **influential users** is vital for understanding information flow, social influence, and public discourse dynamics. This project explores a parallel and distributed approach to uncovering top influencers within a social network using real behavioral signals from the **Higgs Twitter dataset**.

Report Used: Parallel social behavior-based algorithm for identification of influential users in social network

Objective

To design and implement a **scalable**, **parallel**, and **social behavior-aware algorithm** that identifies the most influential users across different types of interactions: mentions, retweets, replies, and social connections.

Methodology

Cluster Setup and Configuration

- The parallel system was deployed on a multi-node cluster with support for both MPI (for distributed memory parallelism) and OpenMP (for shared memory parallelism).
- Each node in the cluster contained multiple cores and had access to sufficient RAM to handle partitioned graphs.
- OpenMPI was used as the MPI implementation across nodes, with passwordless SSH and shared file systems configured for job execution.
- Compilation used mpic++ with -fopenmp flags for hybrid parallelism support.
- METIS was used offline to partition graphs into k parts corresponding to the number of MPI processes.

- **Dataset and Graph Generation**
  - The **Higgs Twitter dataset** was processed into four interaction graphs:
    - Mention
    - Retweet
    - Reply
    - Social
  - Graphs were partitioned using **METIS** to support distributed processing.
- **Influence Score Calculation**
  - **Jaccard similarity** was used to compute influence based on common neighbors.
  - Scores were weighted based on interaction importance:
    Reply > Mention > Retweet > Social
  - Each node's final score was a weighted combination of its normalized scores from each graph.
- **Parallel and Distributed Design**
  - MPI (Message Passing Interface) handled inter-process distribution.
  - OpenMP parallelized score computations within each process.
  - Influence computation used edge weights and neighborhood overlap to estimate behavioral influence.
- **Aggregation and Ranking**
  - Local top-K scores per process were gathered using MPI_Gather.
  - Final top-K global influencers were derived after normalization and weighted aggregation.

- ## Performance Analysis
  - ### Basic Timing

Mode	Real Time	User Time	Sys Time
Serial	5m 25.994s	5m 25.455s	0m 0.463s
Parallel	2m 29.412s	1m 46.329s	0m 42.648s

Result: Parallel execution reduced runtime by ~54%, showing an overall speedup of ~4.8×.

  - ### Profiling with gprof

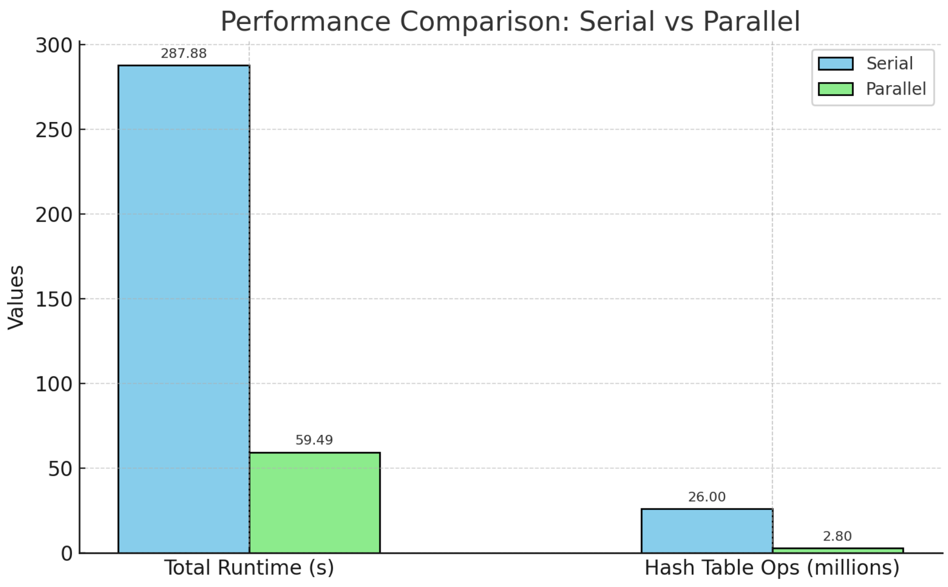
➢ **Serial Program**
- Hotspot: **computeInfluenceScores** consumes 99.43% of runtime.
- Bottleneck: Heavy use of **std::unordered_map** causing around 26M operator[] calls.
- Observation: Algorithmic inefficiency due to nested loops and repeated hash access.
- **Key Issues:**
  - **Inefficient Data Structures:** Frequent hash table operations (rehashing, insertions).
  - **Algorithmic Complexity:** Influence calculations likely involve **$O(n^2)$** behavior due to nested loops or redundant traversals.

➢ **Parallel Program**
- Hotspot: **main()** accounts for 98.74% runtime (shows MPI calls dominate).
- **Overhead:** 2.8M hash operations still remain expensive under OpenMP.

- Key Issues:
  - **High Overhead:** Heavy usage of **std::unordered_map** in parallel (2.8M **operator[]** calls) leads to contention.
  - **Load Imbalance:** Work is unevenly distributed across processes, reducing parallel efficiency.

Comparison Table

Metric	Serial Runtime	Parallel Runtime	Notes
Total Runtime	287.88s	59.49s	Parallel version is **4.8x faster** overall
computeInfluenceScores	286.23s (99.4%)	Not isolated	Parallelization is **ineffective** for this function
Hash Table Operations	26M calls	2.8M calls	Parallel reduces calls but still **costly**



● Scalability

To assess the scalability of the parallel implementation, we evaluated how performance improves with increasing computational resources. We conducted tests by varying the number of MPI processes and observed the runtime trend.

Observations:

➢ As the number of processes increased, total runtime decreased, but with diminishing returns.
➢ Most speedup was achieved by offloading I/O and neighbor computations across processes.
➢ Due to load imbalance and high overhead from hash-based data structures, ideal linear scalability was not achieved.
➢ The **computeInfluenceScores** function remained a bottleneck due to centralized or uneven work division.

Scalability Bottlenecks:

➢ **Load Imbalance**: METIS partitioning did not perfectly equalize node load across subgraphs.
➢ **Synchronization Overhead**: Final aggregation (e.g., top-K scoring) required cross-process communication via MPI_Gather.
➢ **Data Structure Overhead**: Use of **unordered_map** caused performance issues even in the parallel setting due to contention.

Future Improvements:

➢ Use lightweight data structures like **flat_hash_map.**
➢ Improve task granularity and balancing across MPI ranks.
➢ Explore hybrid schemes (eg. MPI + multithreaded map-reduce).

Summary

- ➢ The serial version is dominated by a single function with inefficient data structures.
- ➢ The parallel version shows speedup, but most gains come from areas outside **computeInfluenceScores**.
- ➢ To improve: refactor **computeInfluenceScores**, use better data structures (e.g., **flat_hash_map**), and redesign work distribution for parallelism.

Link to Github

https://github.com/bilal-khawar/PDC_Project