Parallel and Sequential Implementations of PSAIIM Algorithm for Social Network Influence Analysis

Group Member:

Name: Saeed Ud Din Ahmad 22i-0938

Name: Zohaib Khan 22i-0946

Name: Saad Abrar 22i-1238

1. Introduction

Social networks play a crucial role in the dissemination of information. Identifying influential users who can effectively spread ideas, opinions, or products is key in domains such as viral marketing and epidemic control. Given the size of modern networks, parallel processing is essential for scalable influence analysis. This project implements and analyzes a state-of-the-art parallel algorithm PSAIIM for identifying influential users.

Why PSAIIM?

Because in traditional Influence User finding only considers the degree of the hat node(user) but in this updated Algo PSAIIM there are other semantics also considered in finding the top k influence maximum nodes.

2. Objectives

The primary objective is to implement the PSAIIM algorithm in four configurations and compare their performance:
1. Sequential
2. MPI
3. OpenMP + MPI
4. OpenMP + MPI + METIS

The goal is to analyze scalability, speedup, and efficiency.

3. Methodology

The project was implemented in C++ using four different paradigms:
- Sequential: Direct implementation of the PSAIIM algorithm.
- MPI: Distributed nodes and edge lists across processes.
- MPI + OpenMP: Shared memory parallelism within each process.
- MPI + OpenMP + METIS: Preprocessed graphs with METIS for optimized load balancing.

Core Algorithm Steps:
1. Community-Aware Centrality Computation
2. Influence Propagation via CAC
3. Seed Candidate Selection
4. Seed Pruning using BFS Tree Influence

4. Implementation Details

4.1 Parallelization Strategy

- MPI:

Each MPI process handles a separate partition of the input graph. Communication occurs during synchronization of influence values.

- Hybrid Model:

The combination of MPI (inter-node) and OpenMP (intra-node) improves performance by optimizing processor-level and thread-level parallelism.

4.2 Graph Partitioning using SCC and CAC

We used the HIGGS Twitter Dataset, which consists of multiple edge files capturing social, mention, reply, and retweet networks. We focused on the higgs-social_network.edgelist for the primary graph and performed graph

Partitioning as follows:

- **SCC (Strongly Connected Components):**

SCCs identify tightly linked subgraphs where each node is reachable from every another node in the component. These subgraphs were ideal for isolated parallel processing as they inherently contain high locality.

- **CAC (Community-Aware Clustering):**

CAC detects communities based on user interaction frequency and behavior similarity. This approach aligns well with social influence propagation, though Overlaps between communities require synchronization between MPI processes.

4.3 Dataset Handling

• Dataset: Higgs Twitter network (SNAP repository), containing 456k nodes and over 14 million edges.

We utilized the following files from the HIGGS Twitter dataset:

- higgs-social_network.edgelist – structural edges
- higgs-mention_network.edgelist, higgs-reply_network.edgelist,
- higgs-retweet_network.edgelist – behavioral interactions
- higgs-reply_network.edgelist – temporal information of activity

4.4 Technical Workflow Steps (1–7)

1. **Load Data**
   - Loads Higgs Twitter dataset including:
     - social, retweet, reply, mention networks
2. **Initialize Graph**
     - Adds directed, weighted edges and assigns interest vectors.
     - Initializes up to 500,000 nodes.
3. **Detect Communities**
   - Uses DFS to find
     - Strongly Connected Components (SCCs)
     - Single-node CACs (Connected Acyclic Components)
     -
4. **Calculating Influence Power**
   - Weights:
     - α_retweet = 0.50
     - α_comment = 0.35
     - α_mention = 0.15
     - Damping factor = 0.85
5. **Select Seed Candidates**
   - Compares influence power to neighborhood influence.
6. **Select Seeds**
   - Selects top-k seeds maximizing spread.
   - Build Influence-BFS trees for candidates.

4.5 Complexity

- **Time Complexity:**

O(((k * m) / p) + n)

where:

  - k = PPR iterations
  - m = number of edges
  - n = number of nodes
  - p = threads

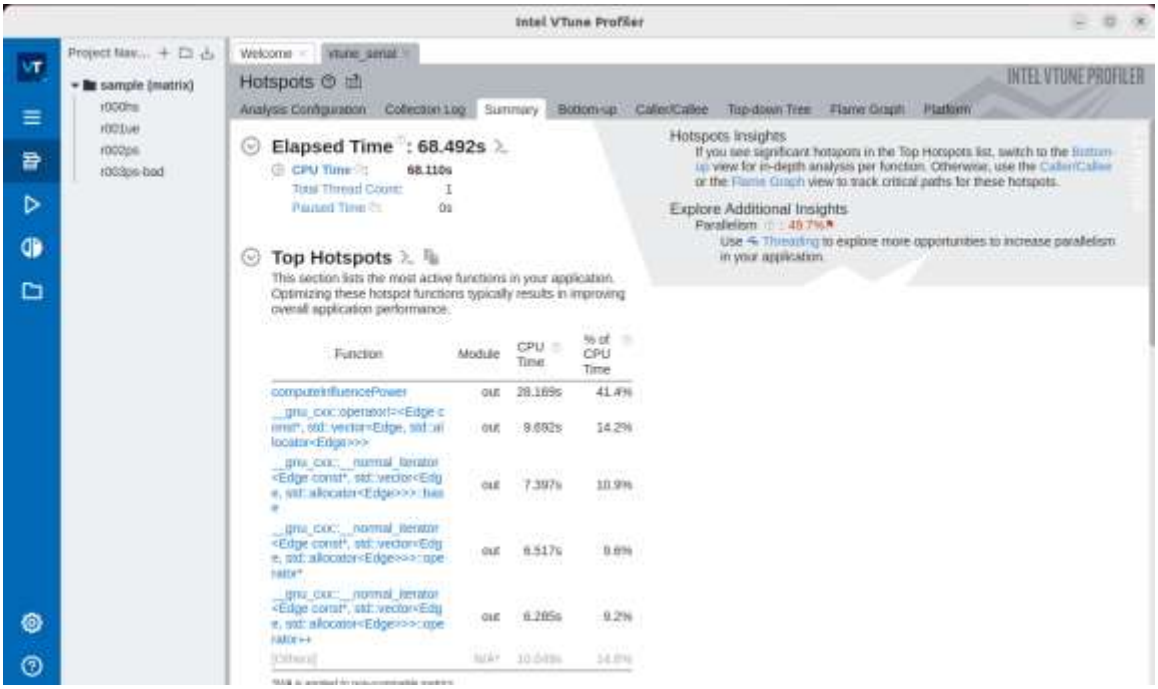
- **Space Complexity:**

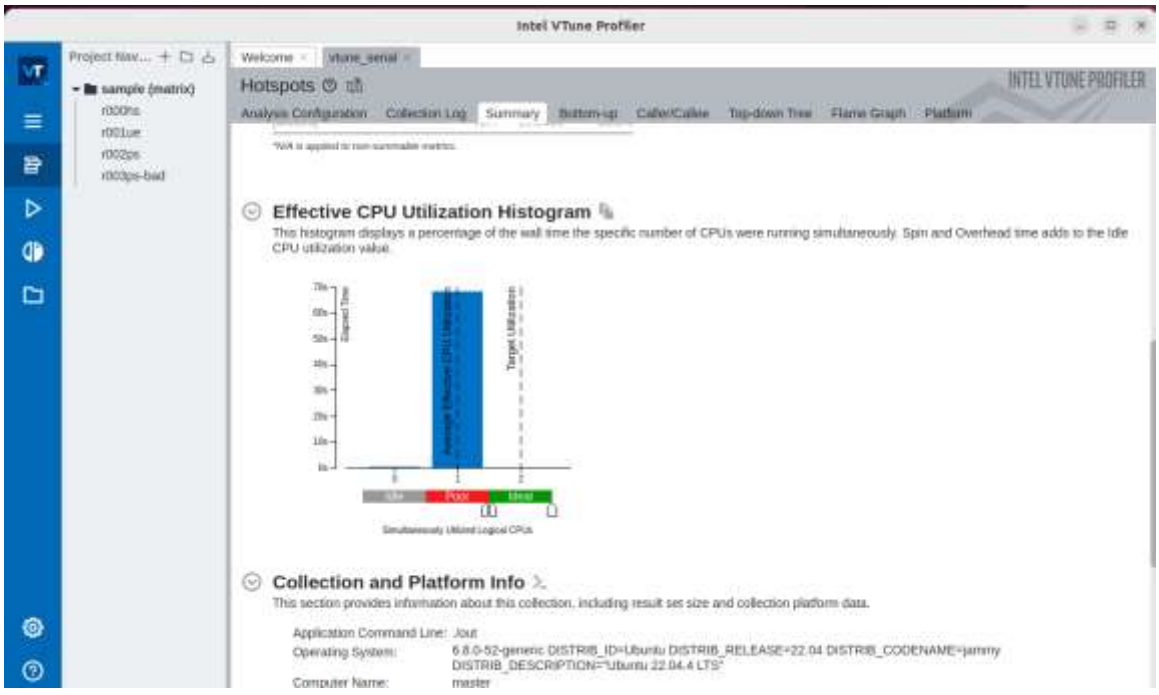
O(max(n, m))

5. Results and Performance Analysis

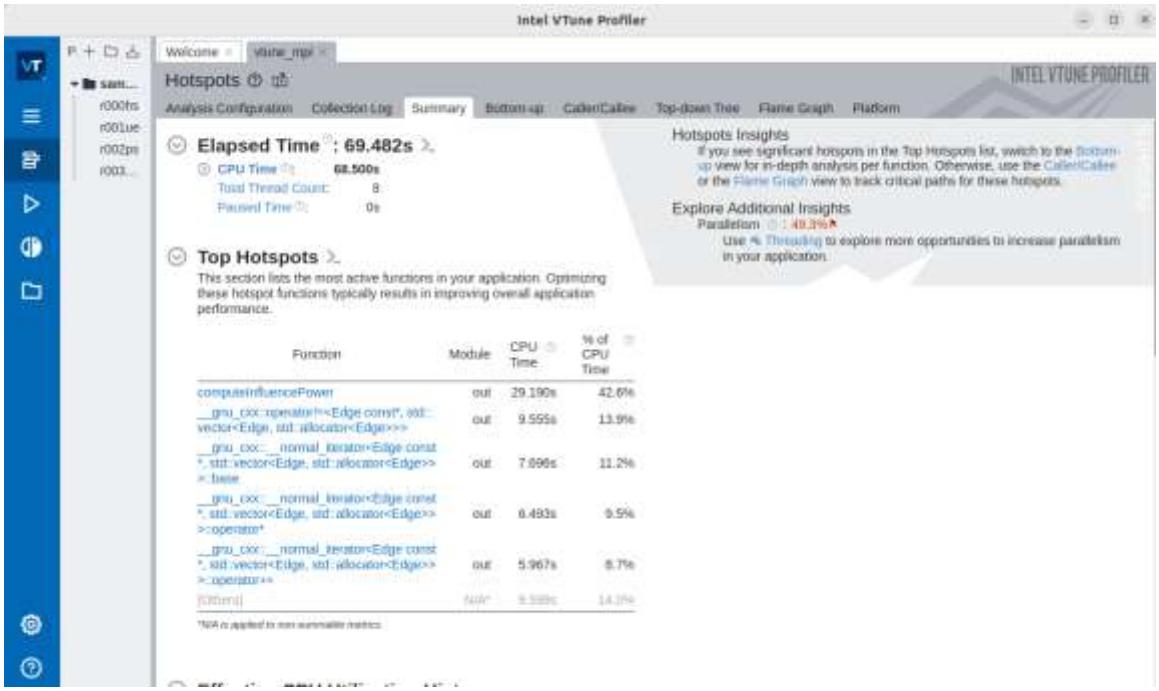
So, I used vTune for Performance Analysis and all Codes are tested or now on 10,000 Nodes

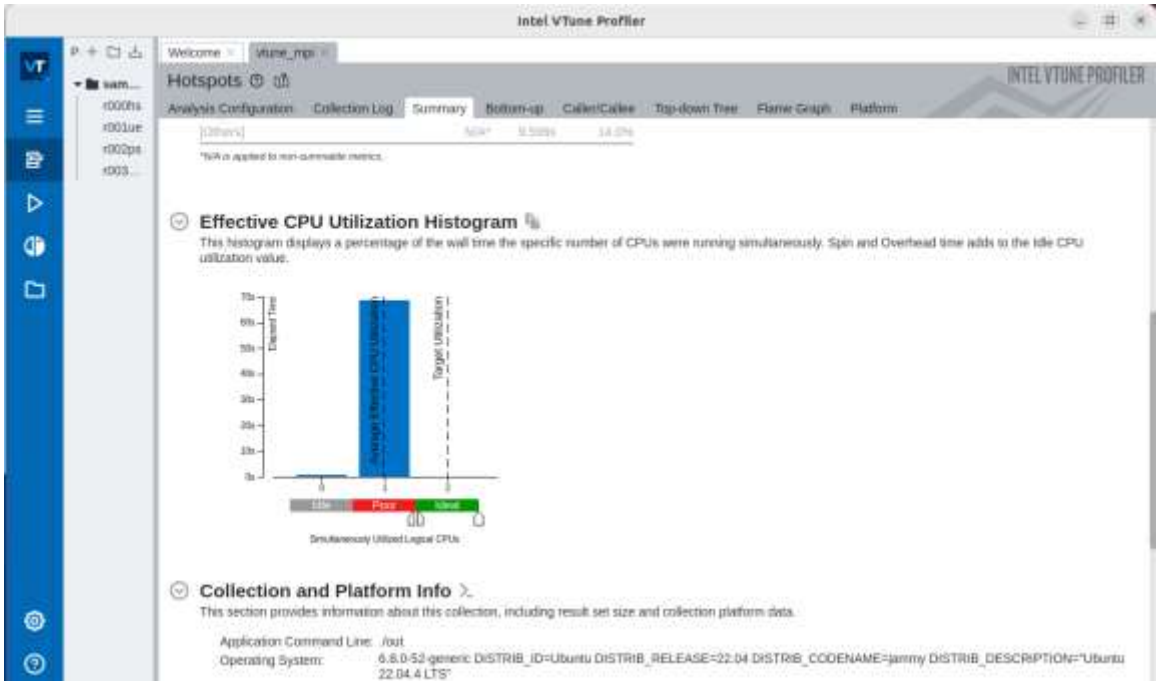
- **Sequential Code:**



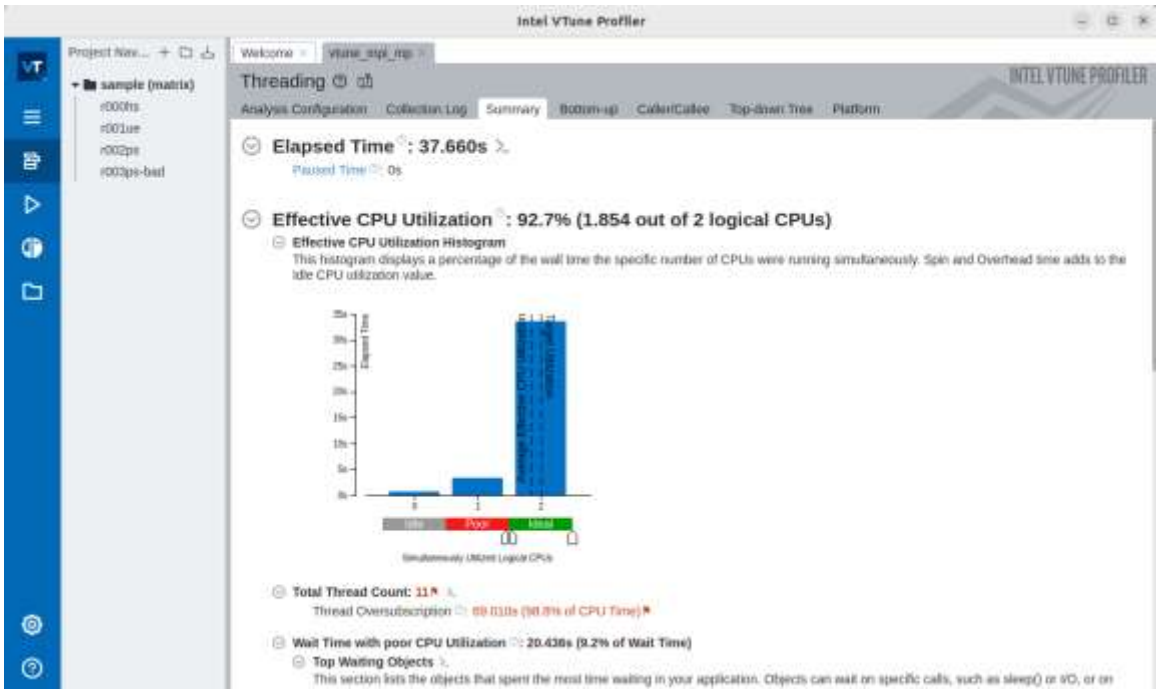


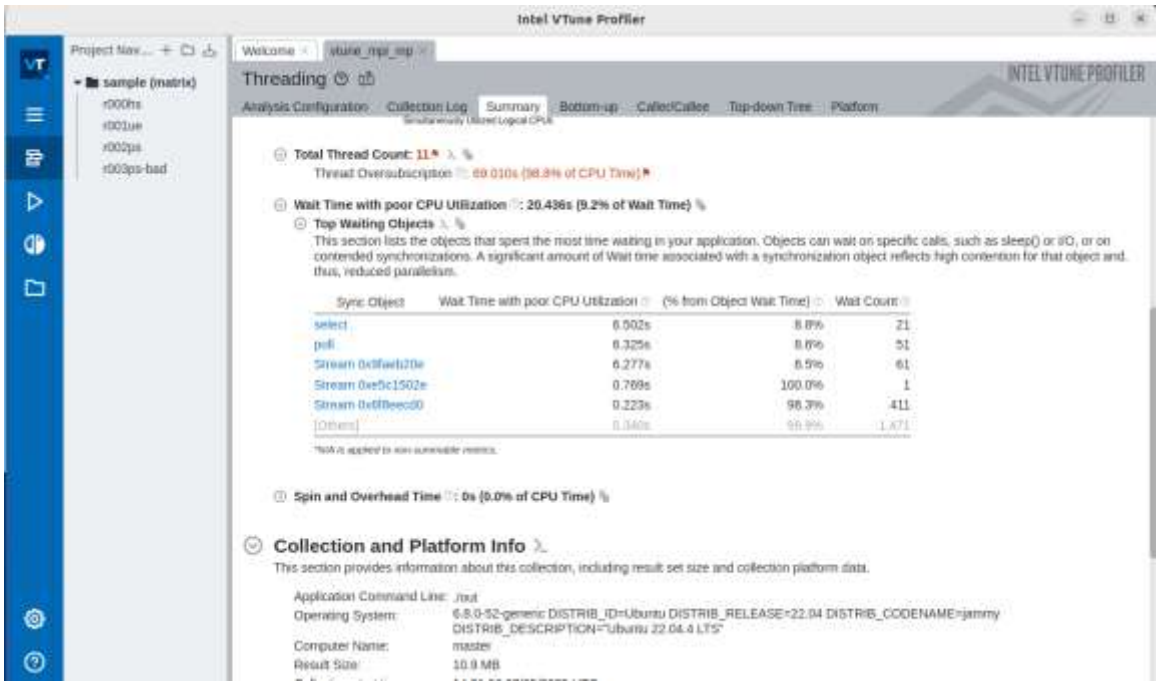
- **MPI**


Intel VTune Profiler
Welcome
vtune_mpi
Hotspots
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Elapsed Time : 69.482s
CPU Time 68.500s
Total Thread Count: 8
Paused Time 0s
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.
Explore Additional Insights
Parallelism 49.3%
Use Threading to explore more opportunities to increase parallelism in your application.
Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.
Function
Module
CPU Time
% of CPU Time
computeInfluencePower
out
29.190s
42.6%
__gnu_cxx::operator!=<Edge const*, std::vector<Edge, std::allocator<Edge>>>
out
9.555s
13.9%
__gnu_cxx::__normal_iterator<Edge const*, std::vector<Edge, std::allocator<Edge>>>::base
out
7.696s
11.2%
__gnu_cxx::__normal_iterator<Edge const*, std::vector<Edge, std::allocator<Edge>>>::operator*
out
6.493s
9.5%
__gnu_cxx::__normal_iterator<Edge const*, std::vector<Edge, std::allocator<Edge>>>::operator++
out
5.967s
8.7%


Intel VTune Profiler
Welcome
vtune_mpi
Hotspots
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.
Elapsed Time
Target Utilization
Idle
Poor
Ideal
Simultaneously Utilized Logical CPUs
Collection and Platform Info
This section provides information about this collection, including result set size and collection platform data.
Application Command Line: ./out
Operating System: 6.8.0-52-generic DISTRIB_ID=Ubuntu DISTRIB_RELEASE=22.04 DISTRIB_CODENAME=jammy DISTRIB_DESCRIPTION="Ubuntu 22.04.4 LTS"

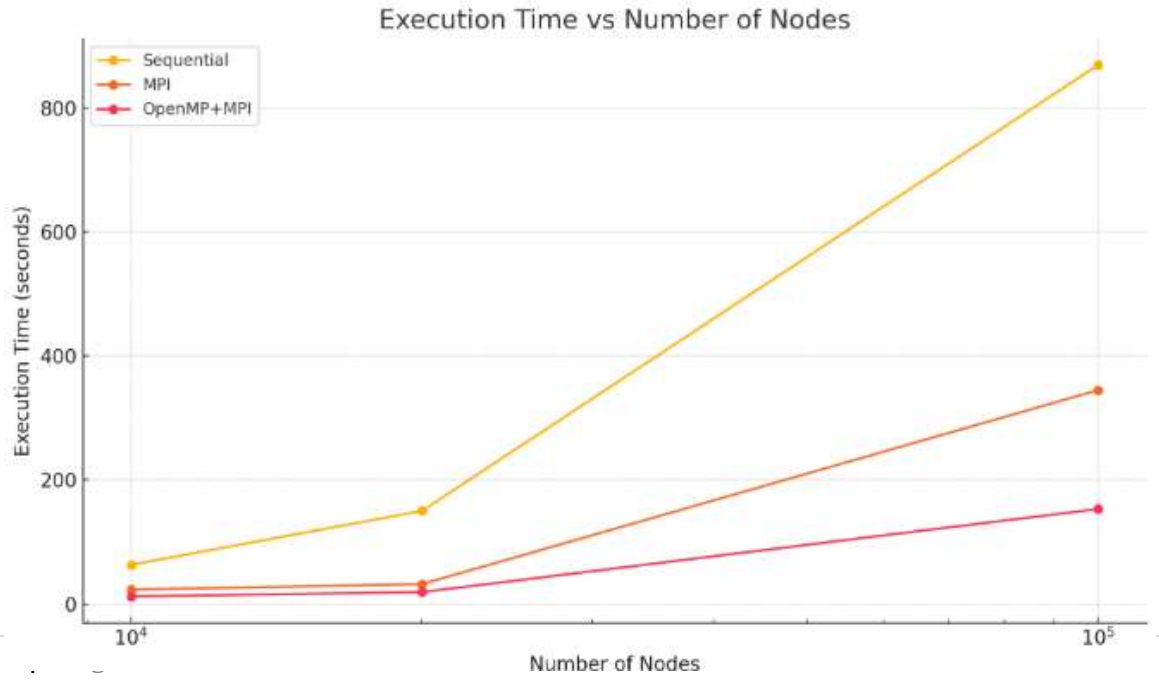
- **OpenMP + MPI**


Intel VTune Profiler
Welcome
vtune_mpi_mp
Threading
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Platform
sample (matrix)
r000ts
r001ue
r002ps
r003ps-bad
Elapsed Time: 37.660s
Paused Time: 0s
Effective CPU Utilization: 92.7% (1.854 out of 2 logical CPUs)
Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.
Elapsed Time
Idle
Poor
Ideal
Total Thread Count: 11
Thread Oversubscription
Wait Time with poor CPU Utilization: 20.438s (9.2% of Wait Time)
Top Waiting Objects
This section lists the objects that spent the most time waiting in your application. Objects can wait on specific calls, such as sleep() or I/O, or on



Configuration	Dataset Size	Execution Time
Sequential	10000	68.492 seconds
MPI	10000	69.48 seconds
OpenMP + MPI	10000	37.660 seconds

The configuration with OpenMP + MPI outperforms other configurations by a major difference because of its CPU level Optimization and Efficient use of MPI.

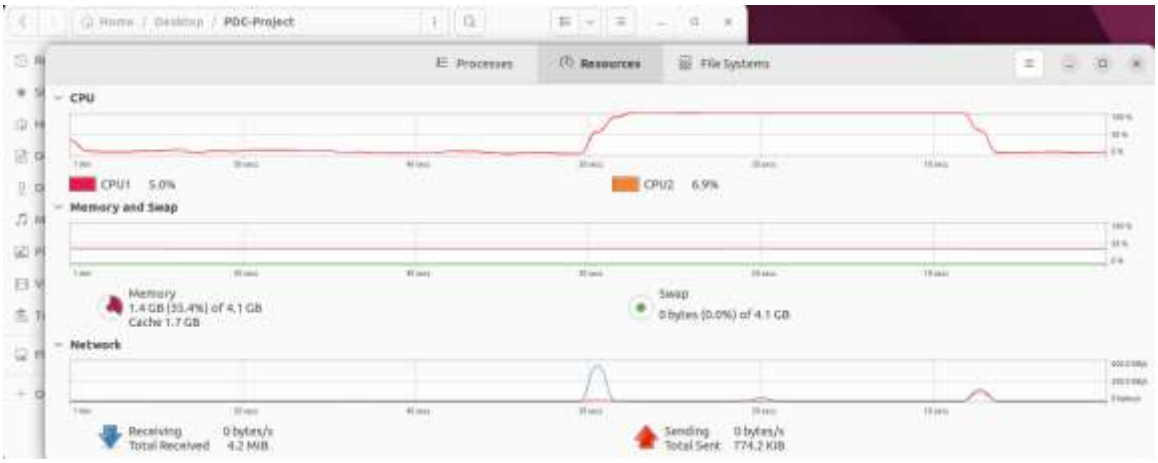


6. Clustering

The hybrid Model Processing on another node

- **Simple MPI configuration:**

- **Hybrid version OpenMP+MPI:**



7. Conclusion

The PSAIIM algorithm can be effectively parallelized. The MPI-based versions performed significantly better than the sequential baseline. The addition of OpenMP provided intra-process acceleration so decreasing the execution time.

8. Future Work

- -Implement the GPU-based version using OpenCL.

9. References

- Mnasri, W., et al. (2021). Parallel social behavior-based algorithm for identification of influential users in social networks. SN Applied Sciences.
  Paper Link