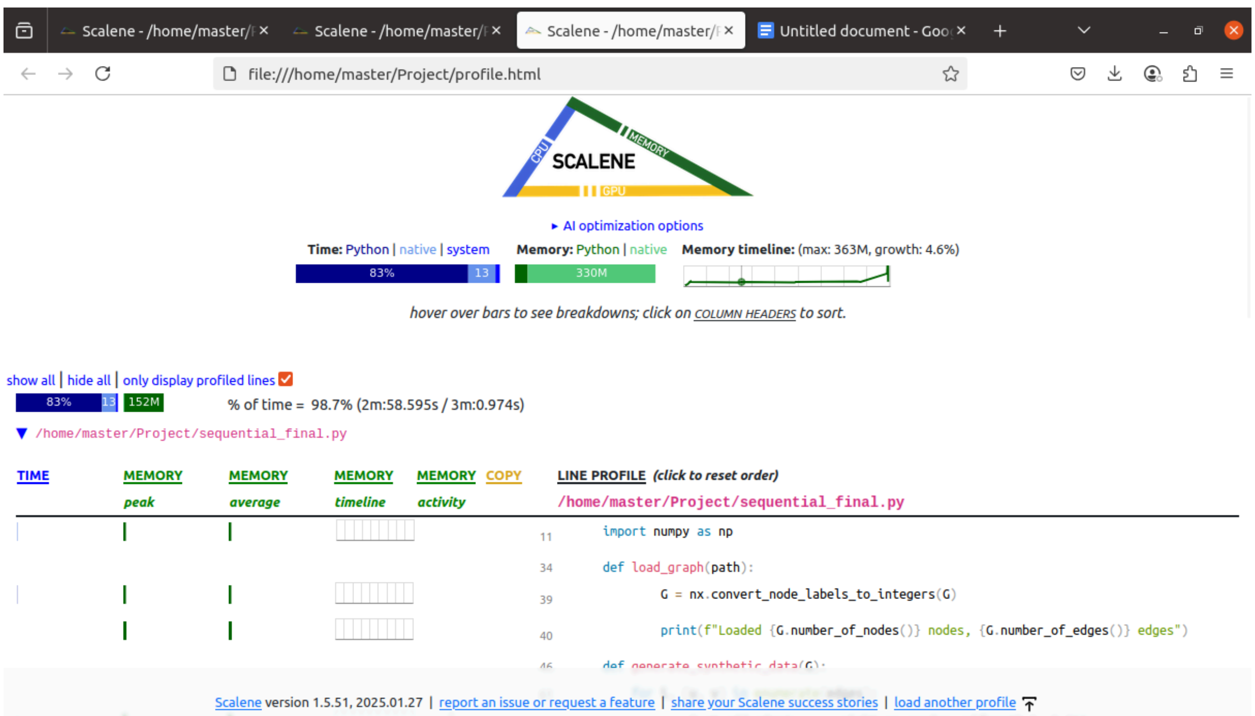
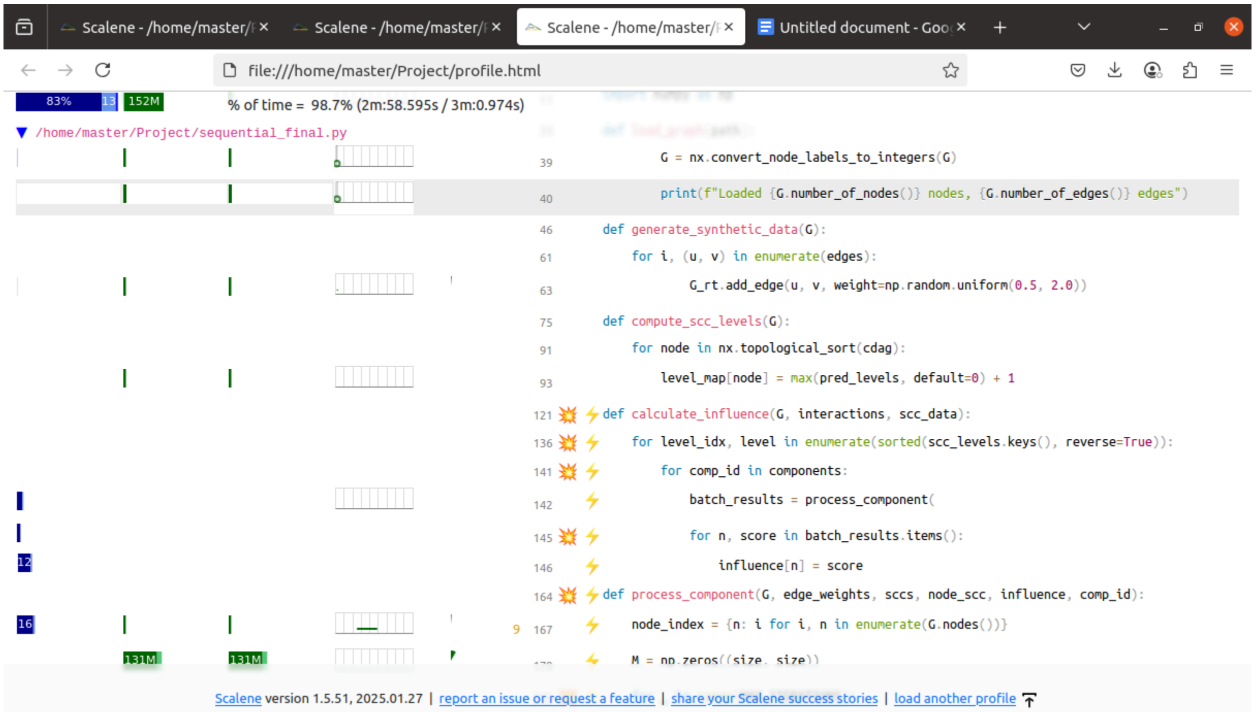
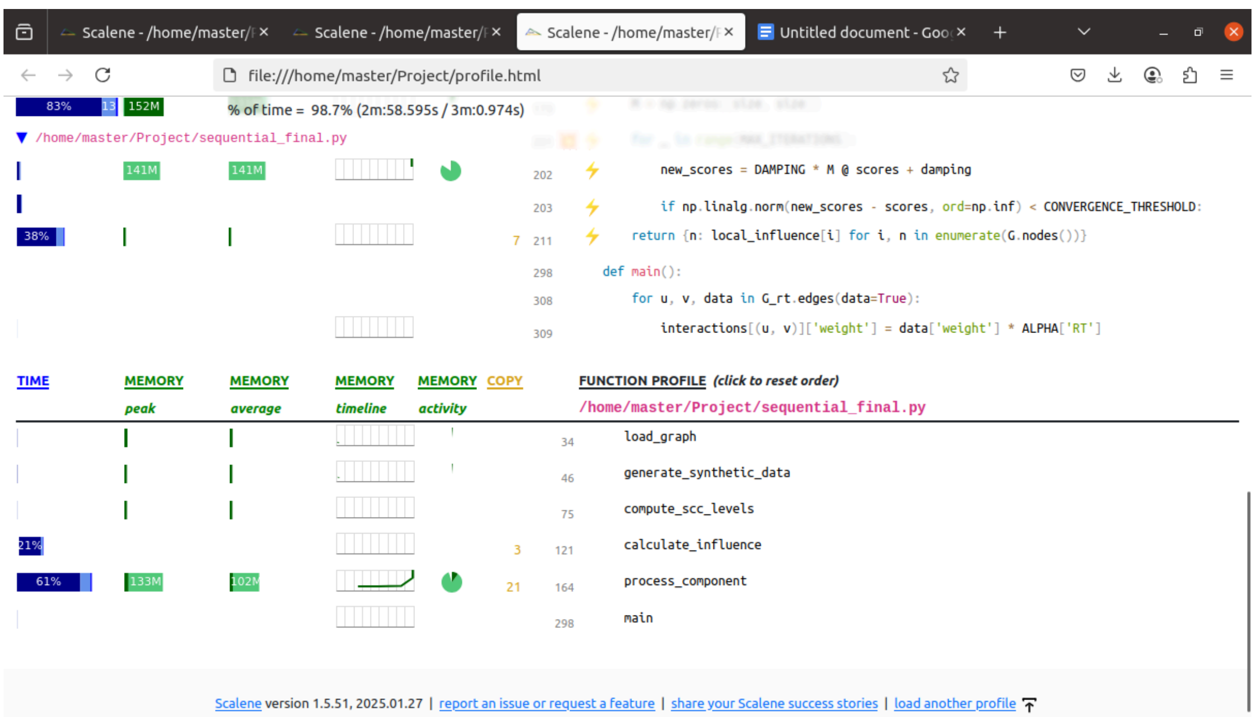
PDC PROJECT REPORT

Scalene Analysis (Best for deep analysis)
Used for all 3 (Time + memory + I/O)

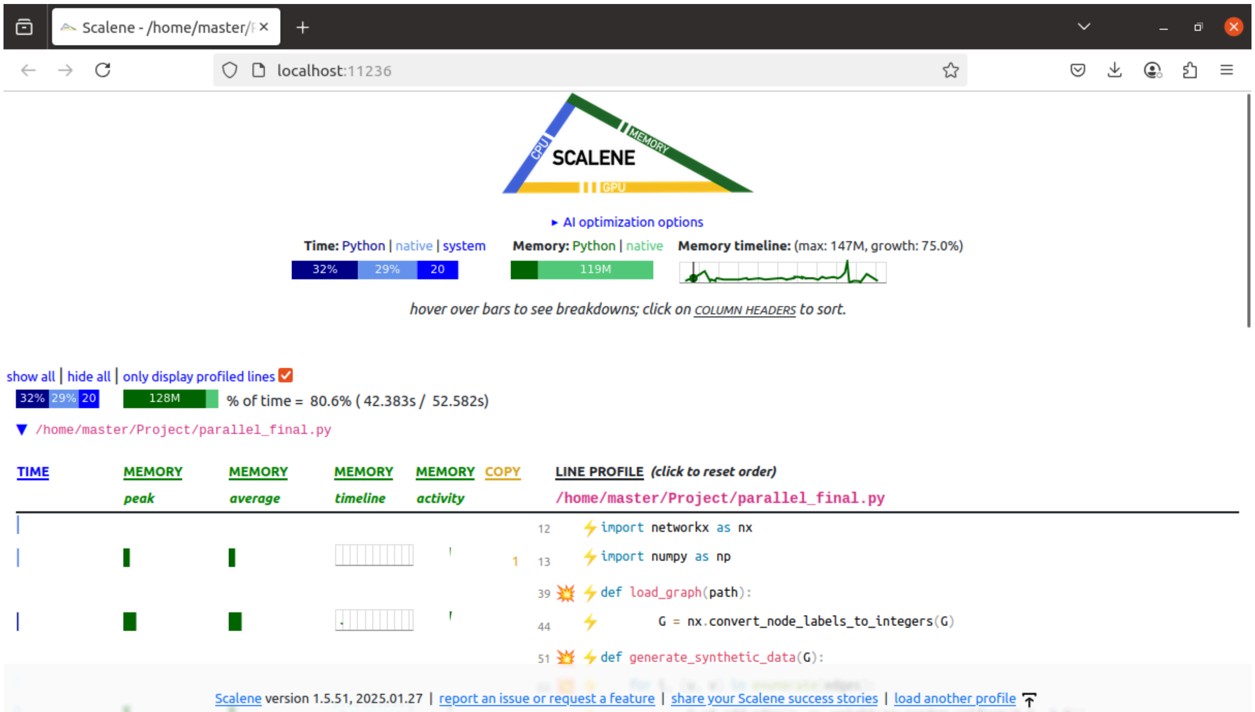
Sequential:

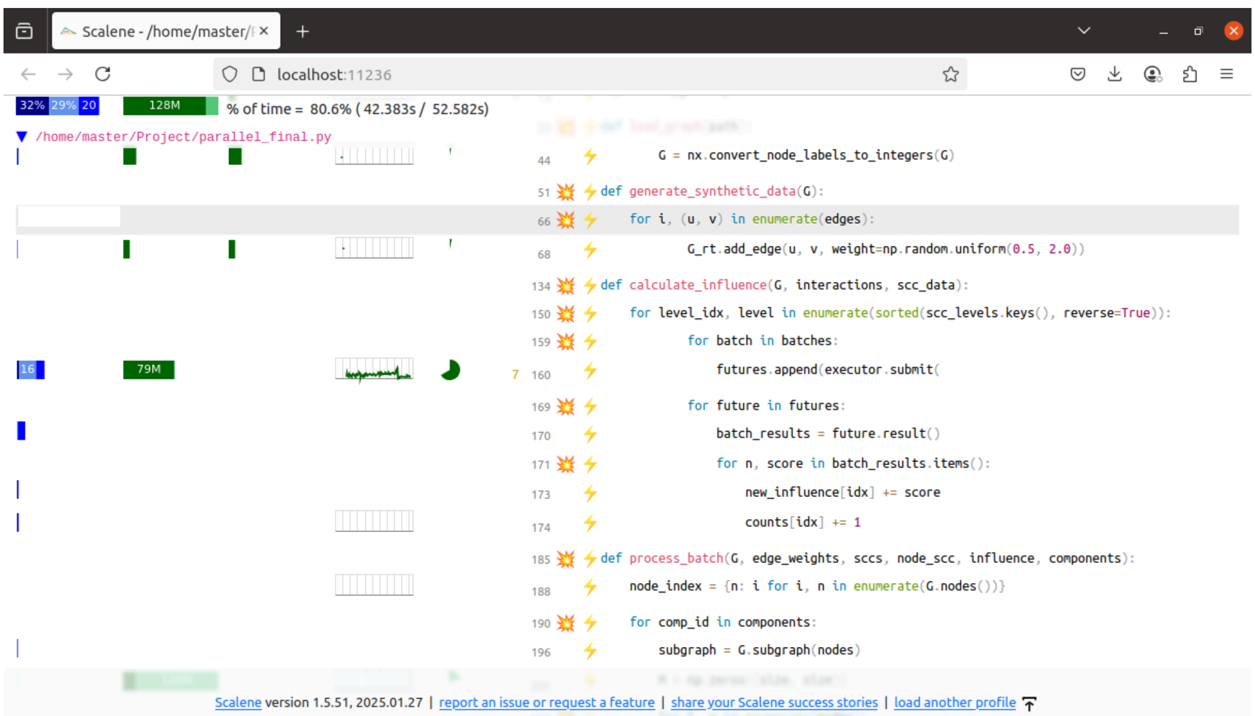


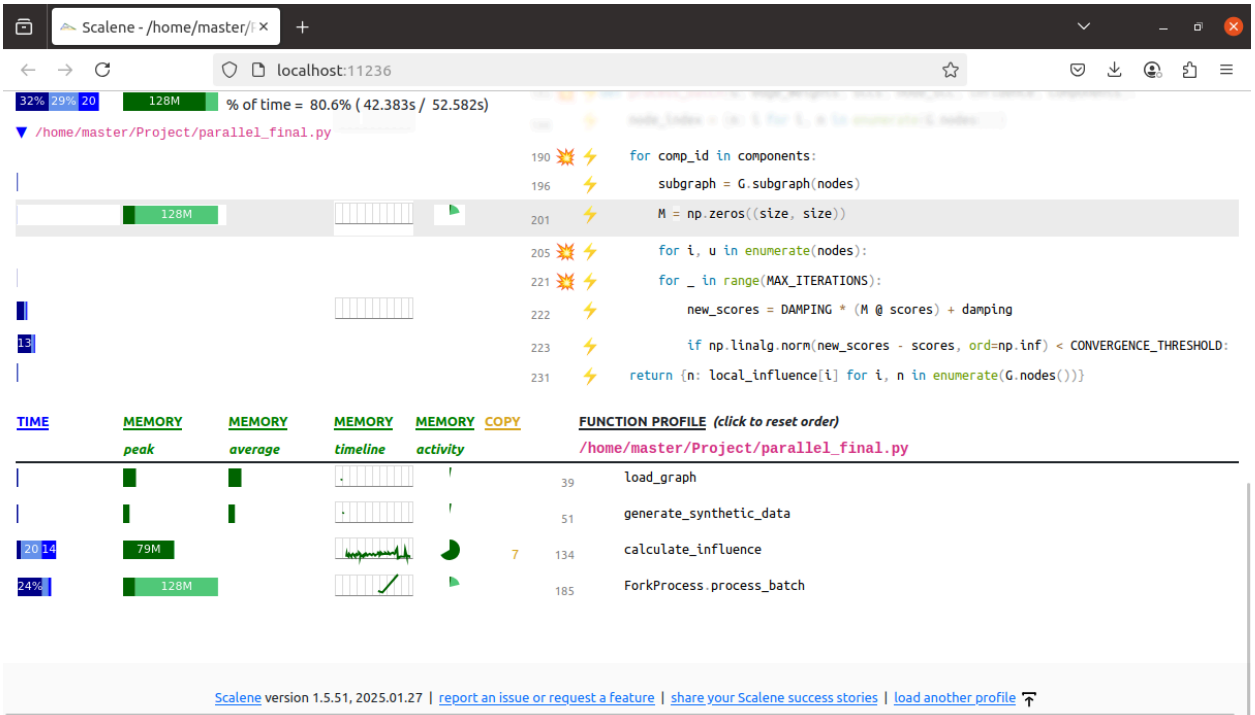

Scalene - /home/master/
Untitled document - Goo
file:///home/master/Project/profile.html
83% 13 152M
% of time = 98.7% (2m:58.595s / 3m:0.974s)
/home/master/Project/sequential_final.py
39 G = nx.convert_node_labels_to_integers(G)
40 print(f"Loaded {G.number_of_nodes()} nodes, {G.number_of_edges()} edges")
46 def generate_synthetic_data(G):
61 for i, (u, v) in enumerate(edges):
63 G_rt.add_edge(u, v, weight=np.random.uniform(0.5, 2.0))
75 def compute_scc_levels(G):
91 for node in nx.topological_sort(cdag):
93 level_map[node] = max(pred_levels, default=0) + 1
121 def calculate_influence(G, interactions, scc_data):
136 for level_idx, level in enumerate(sorted(scc_levels.keys(), reverse=True)):
141 for comp_id in components:
142 batch_results = process_component(
145 for n, score in batch_results.items():
146 influence[n] = score
164 def process_component(G, edge_weights, sccs, node_scc, influence, comp_id):
16 9 167 node_index = {n: i for i, n in enumerate(G.nodes())}
131M 131M M = np.zeros((size, size))
Scalene version 1.5.51, 2025.01.27 | report an issue or request a feature | share your Scalene success stories | load another profile


Scalene - /home/master/
Untitled document - Goo
file:///home/master/Project/profile.html
83% 13 152M
% of time = 98.7% (2m:58.595s / 3m:0.974s)
/home/master/Project/sequential_final.py
141M 141M 202 new_scores = DAMPING * M @ scores + damping
203 if np.linalg.norm(new_scores - scores, ord=np.inf) < CONVERGENCE_THRESHOLD:
38% 7 211 return {n: local_influence[i] for i, n in enumerate(G.nodes())}
298 def main():
308 for u, v, data in G_rt.edges(data=True):
309 interactions[(u, v)]['weight'] = data['weight'] * ALPHA['RT']
TIME
MEMORY peak
MEMORY average
MEMORY timeline
MEMORY activity
COPY
FUNCTION PROFILE (click to reset order)
/home/master/Project/sequential_final.py
34 load_graph
46 generate_synthetic_data
75 compute_scc_levels
21% 3 121 calculate_influence
61% 133M 102M 21 164 process_component
298 main
Scalene version 1.5.51, 2025.01.27 | report an issue or request a feature | share your Scalene success stories | load another profile

Parallel:


Scalene - /home/master/
localhost:11236
SCALENE
CPU
MEMORY
GPU
▸ AI optimization options
Time: Python | native | system
32% 29% 20
Memory: Python | native
119M
Memory timeline: (max: 147M, growth: 75.0%)
hover over bars to see breakdowns; click on COLUMN HEADERS to sort.
show all | hide all | only display profiled lines
32% 29% 20
128M
% of time = 80.6% (42.383s / 52.582s)
▼ /home/master/Project/parallel_final.py
TIME
MEMORY peak
MEMORY average
MEMORY timeline
MEMORY activity
COPY
LINE PROFILE (click to reset order)
/home/master/Project/parallel_final.py
12 import networkx as nx
1 13 import numpy as np
39 def load_graph(path):
44 G = nx.convert_node_labels_to_integers(G)
51 def generate_synthetic_data(G):
Scalene version 1.5.51, 2025.01.27 | report an issue or request a feature | share your Scalene success stories | load another profile


Scalene - /home/master/
localhost:11236
32% 29% 20
128M
% of time = 80.6% (42.383s / 52.582s)
▼ /home/master/Project/parallel_final.py
44 G = nx.convert_node_labels_to_integers(G)
51 def generate_synthetic_data(G):
66 for i, (u, v) in enumerate(edges):
68 G_rt.add_edge(u, v, weight=np.random.uniform(0.5, 2.0))
134 def calculate_influence(G, interactions, scc_data):
150 for level_idx, level in enumerate(sorted(scc_levels.keys(), reverse=True)):
159 for batch in batches:
16
79M
7 160 futures.append(executor.submit(
169 for future in futures:
170 batch_results = future.result()
171 for n, score in batch_results.items():
173 new_influence[idx] += score
174 counts[idx] += 1
185 def process_batch(G, edge_weights, sccs, node_scc, influence, components):
188 node_index = {n: i for i, n in enumerate(G.nodes())}
190 for comp_id in components:
196 subgraph = G.subgraph(nodes)
Scalene version 1.5.51, 2025.01.27 | report an issue or request a feature | share your Scalene success stories | load another profile



Time comparison:

Sequential : Approx 114 sec

Parallel: Approx 22 sec

MPI : Approx 12 sec

MPI+(OpenMP) : Approx 10 sec

Speedup:

Speedup = Parallel Time / Sequential Time

Speedup Parallel = 114 / 22 ≈ 5.18

Speedup MPI = 114 / 12 ≈ 9.5

Speedup for MPI + OpenMP (Hybrid) ≈   114 / 10 ≈ 11.4

Code Explanaton:

1) file(Sequential.py) :

This Python script implements a **sequential influence maximization algorithm** (called PSAIIM) on a **peer-to-peer (P2P) network graph** from the `p2p-Gnutella04` dataset. It removes parallelism to keep things simple and focuses on understanding how influence spreads in a social-like network using synthetic interaction data.

🔶 **Step-by-Step Breakdown:**

1. Graph Loading:

The script loads a directed graph from a text file (`p2p-Gnutella04.txt`). Each line represents an edge, indicating a connection from one node to another.

2. Synthetic Interaction Generation:

To simulate real-world social interactions, the script generates **three types of interactions** randomly over the network edges:

- **Retweet (RT)** – 30% chance
- **Reply (RE)** – 20% chance
- **Mention (MT)** – 10% chance
  Each interaction is given a random weight to reflect how strong or frequent it is.

3. Community Detection (SCC Levels):

The graph is analyzed to find **Strongly Connected Components (SCCs)**. These are clusters of nodes where each node can reach every other node in the same component.
These components are organized into **levels** using a **condensed DAG (Directed Acyclic Graph)** formed by collapsing SCCs into single nodes, and then topologically sorting them. This helps in processing influence from higher (source) communities down to lower ones.

4. Influence Calculation:

The core idea is that nodes in a network influence each other based on the structure and the interactions. For each SCC:

- A **transition matrix** is built to model how influence flows between nodes based on edge weights.
- Then, **power iteration** is used to compute influence scores for the nodes in each SCC.
- The influence is **updated level by level**, ensuring higher-level nodes propagate their influence downward.

5. Most Influential Nodes per Community:

For the **top 10 largest communities**, the script identifies the **top 3 most influential nodes** based on the computed influence scores.

6. Seed Selection:

The algorithm selects a set of **top-k nodes (default 10)** to act as **"seeds"** for influence spread. It uses a **community-aware greedy strategy**, considering:

- High influence scores globally
- Top nodes within the largest communities
  This ensures that the selected nodes are both influential and well-distributed across the network structure.

2) file(parallel.py) :

This file aims to:

1. **Analyze the influence of nodes** in a P2P network graph.
2. **Detect communities (SCCs)** in the graph.
3. **Calculate influence scores** within and across these communities.
4. **Select seed nodes** with high influence for diffusion or propagation tasks (like information spread).

Overall Workflow

3) **Load Graph**: Reads a `.txt` file representing the directed graph (`p2p-Gnutella04.txt`) using NetworkX.
4) **Generate Interaction Types**:
   a) Randomly simulates 3 types of interactions on the edges:

i) Retweets (RT)

ii) Replies (RE)

iii) Mentions (MT)

b) Assigns weights to these interactions for influence modeling.

5) **Community Detection via SCCs**:

a) Computes **Strongly Connected Components (SCCs)** of the graph.

b) Converts them into a **condensed DAG** to model levels (hierarchies of communities).

c) Each SCC is assigned a "level" based on topological sorting.

6) **Influence Computation**:

a) Influence scores are initialized with slight randomness and normalized.

b) The algorithm goes through SCC levels in reverse order.

c) For each SCC, it calculates influence via **power iteration** using a transition matrix (similar to PageRank).

d) This is done **in parallel using multiple CPU cores** for efficiency.

7) **Show Influential Nodes**:

   a) From the largest communities, it selects and prints the **top 3 most influential nodes** in each using their computed scores.

8) **Seed Selection**:

   a) Picks k (e.g., 10) nodes with the **highest influence scores**, distributed across different communities, ensuring diversity.

3) file (mpi.py):

What This Code Accomplishes:

- **Distributed Graph Processing**: Each MPI rank loads and partitions the graph, and works on a subgraph.
- **Threaded Processing**: Within each MPI process, edge operations and influence calculations are parallelized with threads.
- **Synthetic Modeling**: Three synthetic edge types simulate different interaction modes (RT/RE/MT).
- **PageRank-Like Influence Scoring**: Uses a weighted version of PageRank within each SCC level.
- **Seed Selection**: Top-k influential nodes per partition are selected and gathered at root.

📌 **Suggestions for Improvement or Extension:**

9) **File Loading Optimization**:

a) All nodes currently load the full graph (`G_full = load_graph(...)`). This could be memory-inefficient for large graphs.

b) **Alternative**: Only rank 0 loads and partitions, then scatters subgraphs to other ranks.

10) **Edge Storage**:

a) Edge weights are recomputed every time from scratch.

b) **Optimization**: Persist or cache synthetic edge graphs and precomputed weights to avoid recomputation on each run.

11) **Influence Propagation Between Partitions**:

a) Influence calculation is done per partition only.

b) **Limitation**: Inter-partition influence is not accounted for (important if cross-partition edges exist).

c) **Fix**: Implement **ghost nodes** or **boundary synchronization** between partitions.

12) **Logging Improvements**:

a) Current `print()` usage is clean but mixed.

b) You could consider using the `logging` module with rank-specific prefixes for consistency and better control.

13) **Scalability Analysis**:

a) Include code for measuring:

i) Speedup per MPI process count

ii) Breakdown of computation vs communication time

14) **Testing on p2p-Gnutella04**:

a) Ensure file format matches expected edge list (directed). You may want to validate the dataset shape before full runs.

4) file (mpi_open-mp.py):

This program performs influence maximization on a graph (like a social network) using parallel processing. It identifies the most "influential" nodes in the network, i.e., the best starting points to spread information.

Key Technologies Used

- MPI (`mpi4py`): Distributes work across multiple processes (machines or cores).
- OpenMP-style threading (`ThreadPoolExecutor`): Uses threads for faster processing within each MPI process.
- METIS: Optional, for smart graph partitioning.
- NetworkX: For graph operations (loading, SCCs, etc.).

Step-by-Step Breakdown

1. Initialization
    - Uses MPI to determine which process you are (rank).
    - Uses available CPU cores to decide how many threads to run.
2. Graph Loading
    - Loads a real-world graph (`p2p-Gnutella04.txt`) into memory using NetworkX.
    - All MPI processes load the full graph.
3. Graph Partitioning
    - If METIS is available, partitions the graph evenly across MPI processes for load balancing.
    - Otherwise, it splits nodes manually based on index.
4. Synthetic Data Generation
    - Creates three interaction graphs (`RT`, `RE`, `MT`) with randomly weighted edges.
    - This simulates different types of relationships in the network.
5. Community Detection
    - Computes Strongly Connected Components (SCCs).
    - Organizes these SCCs into levels, where level 1 has no incoming edges from other SCCs.

6. Edge Weight Computation

    - Precomputes edge weights by combining the weights from RT, RE, and MT graphs using predefined importance factors (ALPHA).

7. Influence Calculation

    - Performs influence propagation per SCC level using a matrix-based iterative method (like PageRank).

    - Processes SCCs from highest to lowest level to respect the direction of influence.

8. Seed Node Selection

    - Picks top `k=10` nodes with the highest influence scores within each MPI process.

9. Result Gathering

    - Master node (rank 0) gathers all top seeds from workers and prints them.

10. Execution Time

- Prints total execution time at the end.
