Asynchronous Hybrid MPI+OpenMP Label Propagation on Erdős–Rényi Graphs

Ahmed Essawi
aae2552
AhmedEssawi05
ahmedessawi23@gmail.com

Abstract

We present a hybrid MPI+OpenMP implementation of an asynchronous label propagation algorithm for community detection on Erdős–Rényi random graphs. Our method distributes the graph rows via MPI, performs neighbor-label counting in parallel with OpenMP, and guarantees exact convergence by updating and broadcasting one node at a time. We evaluate scalability across MPI processes and OpenMP threads, and discuss extensions to stochastic block models for nontrivial community structure.

1 Introduction

Label propagation is a simple, parameter-light method for community detection: each node adopts the most frequent label among its neighbors. Traditional synchronous algorithms often collapse to a single label on random graphs. We implement an *asynchronous* scheme—updating nodes one at a time and broadcasting each change immediately—using MPI for inter-process communication and OpenMP for intra-process speedup. The algorithm terminates only when no label changes globally.

2 Problem Statement

Given a random graph $G(n, p)$ with $n = 10{,}000$ nodes and edge probability $p = 10^{-4}$, we partition nodes into communities by iteratively reassigning each node's label to the most frequent label among its neighbors, looping until global label stability.

3 Methodology

3.1 Graph Generation

MPI rank 0 generates the full $n \times n$ adjacency matrix for $G(n, p)$:

$$A_{ij} = \begin{cases} 1, & \text{with probability } p, \\ 0, & \text{otherwise.} \end{cases}$$

A fixed seed ensures reproducibility.

3.2 Data Distribution

To distribute memory and work:

- Compute per-rank row counts `countsRows`$[r]$ and displacements `displsRows`$[r]$ so that each of the P MPI ranks receives $\lceil n/P \rceil$ consecutive rows.
- Use `MPI_Scatterv` to send each rank its block of rows (stored in `localAdj`).

3.3 Label Initialization

- Each node's initial label equals its node ID.
- Rank 0 initializes `globalLabels`$[0 \ldots n-1] = 0 \ldots n-1$ and broadcasts via `MPI_Bcast`.
- Each rank copies its segment into `localLabels`.

4 Algorithmic Complexity and Overheads

We analyze the computational and communication costs of our asynchronous hybrid algorithm.

4.1 Time Complexity

Each global update processes one node g:

- **Neighbor scanning:** An $O(n)$ loop over n possible neighbors, parallelized across T OpenMP threads, yields $O(n/T)$ work per thread.
- **Histogram reduction:** Building the neighbor-label histogram requires n atomic increments, also $O(n)$.
- **Broadcast:** We issue one `MPI_Bcast` of size 1 per node, for n broadcasts per pass.

Thus each pass costs $O(n^2/T)$ compute plus n broadcasts.

The number of passes to convergence, P_c, is typically small (3–5) on sparse ER graphs. Overall compute time is

$$O\Big(P_c \frac{n^2}{T}\Big).$$

4.2 Communication Overheads

Two major costs:

- `MPI_Scatterv`: a single collective to distribute the adjacency matrix rows. Cost $O(n^2/P)$ per rank.
- `MPI_Bcast` per node update: n broadcasts, each with latency α and negligible bandwidth (1 integer). Total cost $O(n\alpha)$ per pass.
- `MPI_Allreduce` of a boolean per pass: cost $O(\log P \cdot \alpha)$.

For large n and moderate P, the dominant overhead shifts from neighbor scanning to the n-fold broadcast latency.

4.3 Memory Footprint

Each rank stores:

- Adjacency block of size `countsRows`$[r] \times n$ ($\approx n^2/P$ entries).
- Global label array of size n.
- Local label array of size n/P.

Total memory per rank: $O(n^2/P + n)$, which scales inversely with P.

5 Code Snippets and Explanations

5.1 Scatter and Initialization

```
// Compute per-rank row counts and displacements
total = numNodes; P = nprocs;
base = total/P; rem = total%P;
for(r=0,off=0;r<P;r++){
  countsRows[r] = base + (r<rem);
  displsRows[r] = off;
  off += countsRows[r];
}
// Distribute adjacency rows
MPI_Scatterv(fullAdj, sendCounts, sendDispls, MPI_INT,
             localAdj, localN*numNodes, MPI_INT,
             0, MPI_COMM_WORLD);
// Broadcast initial labels
MPI_Bcast(globalLabels, numNodes, MPI_INT, 0, MPI_COMM_WORLD);
```

5.2 Asynchronous Node Update

```
// Owner rank updates node g
int *counts = calloc(numNodes, sizeof(int));
int rowOff = (g - displsRows[rank]) * numNodes;
#pragma omp parallel for
for(int j = 0; j < numNodes; j++){
  if(localAdj[rowOff + j]){
    #pragma omp atomic
    counts[ globalLabels[j] ]++;
  }
}
// Select and apply best label
int best = localLabels[i], top = 0;
for(int k = 0; k < numNodes; k++){
  if(counts[k] > top){
    top = counts[k];
    best = k;
  }
}
free(counts);
if(best != localLabels[i]){
  localChanged = 1;
  localLabels[i] = best;
  newLabel = best;
}
globalLabels[g] = newLabel;
// Broadcast updated label
MPI_Bcast(&newLabel, 1, MPI_INT, owner, MPI_COMM_WORLD);
```

6 Performance Evaluation Tables

6.1 Execution Time vs MPI Processes (2304 Nodes)

MPI Processes	Exec. Time (s)	Speedup
2	0.044902	—
4	0.047141	0.95
8	0.048242	0.98
16	0.050893	0.95
32	0.053763	0.95
48	0.057791	0.93

Analysis of MPI Scalability Adding more MPI processes at fixed problem size incurs significant overhead because each of the n node updates issues a separate MPI broadcast.

Despite each broadcast carrying only a single integer, each one incurs non-negligible latency. As the number of processes grows, the per-broadcast latency increases (larger collective trees), while the computational work per rank shrinks. Consequently, the communication cost dominates the decreasing per-rank compute cost, leading to longer overall runtimes when scaling to many MPI processes.

6.2 Execution Time vs OpenMP Threads (2304 Nodes, 48 Processes)

OMP Threads	Exec. Time (s)	Speedup
1	0.058039	1.00
2	0.205695	0.28
4	0.500661	0.41
8	1.053608	0.48
16	2.159725	0.49
32	4.371074	0.49

Analysis of OpenMP Scalability When running with 48 MPI processes on only 2 304 nodes, each rank is responsible for just 48 nodes. At one thread, the single-threaded neighbor scans incur no OpenMP overhead, so you see the lowest time (0.058s). As you increase the thread count:

- **Parallel overhead dominates.** Each node update launches an OpenMP region and uses atomic increments to build a small histogram. With only 48 nodes and 2304 neighbor checks per rank, the overhead of thread creation, synchronization, and atomics quickly exceeds any work saved by splitting the loop.
- **Oversubscription.** You already have 48 MPI ranks saturating the available cores. Adding more threads per rank forces context-switching between threads, further slowing down each neighbor-scan.
- **Diminishing returns (negative scaling).** Beyond 1 thread, you never have enough work per thread to amortize the cost of spawning and synchronizing. Hence the execution time rises roughly linearly with thread count, from 0.058s at 1 thread to over 4s at 32 threads.

In practice, in this small per-rank problem regime you should stick to `OMP_NUM_THREADS=1`, or reduce the number of MPI ranks so that each has more work and thread-level parallelism can pay off.

7 Conclusion

We have analyzed the algorithmic complexity of an asynchronous hybrid MPI+OpenMP label propagation. The compute cost per pass is $O(P_c n^2/T)$, while communication is domi-

nated by n per-node broadcasts costing $O(n\alpha)$. Memory scales as $O(n^2/P+n)$. This analysis highlights the broadcast latency as a key bottleneck and guides future optimizations.