Parallel Community Detection Using MPI and OpenMP

1 Introduction

Community detection in large graphs is a critical problem in various domains, such as social network analysis, biology, and communication systems. The label propagation algorithm is an efficient method for identifying communities in large-scale networks. However, the sequential implementation becomes slow as the size of the graph increases. In this project, we explore the parallelization of the label propagation algorithm using both **MPI (Message Passing Interface)** and **OpenMP (Open Multi-Processing)**. The goal is to improve the performance of the algorithm by distributing the graph across multiple processes (using MPI) and parallelizing the label update operation within each process (using OpenMP).

2 Problem Statement

The problem addressed in this project is the community detection in large-scale graphs using the label propagation algorithm. The challenge is to efficiently detect communities in graphs with thousands or millions of nodes and edges by parallelizing both the graph partitioning (using MPI) and the label propagation process (using OpenMP).

3 Methodology

3.1 Graph Generation

The graph is generated using the **Erdős–Rényi model**, which is a random graph model where each edge is present with a fixed probability p. The graph consists of N nodes, and the adjacency matrix is populated based on the probability of edge creation. Each node is assigned an initial label that is unique.

3.2 Graph Partitioning with MPI

The graph is partitioned into subgraphs, where each MPI process handles a portion of the graph. This partitioning ensures that the graph is divided across the available processes for parallel computation. Each process works on its subgraph, with boundaries defined by the nodes that are shared between adjacent subgraphs.

- **MPI Communication** is used to handle the exchange of boundary node labels between processes. This ensures that each process has up-to-date label information from its neighboring processes.

3.3 Label Propagation with OpenMP

Each process uses **OpenMP** to parallelize the label propagation within its subgraph. The label propagation works by updating each node's label based on the most frequent label of its neighboring nodes. Each process propagates its labels and then communicates boundary nodes to ensure that the propagation continues across subgraphs.

- **OpenMP Parallelization** is used within each process to update the labels of the nodes in parallel, speeding up the process of label propagation.

3.4 Convergence Check

The algorithm iterates for a maximum number of iterations or until convergence is reached. Convergence occurs when no node's label changes during an iteration. This is checked by comparing the current and previous labels for each node.

3.5 Performance Evaluation

The performance of the parallel implementation is measured by the total execution time. The program records the time taken to complete the label propagation process using **MPI** for communication and **OpenMP** for parallel processing within each process. Execution time is measured using `MPI_Wtime()`, which provides the wall clock time across all MPI processes.

4 Parallelization Strategy

4.1 MPI for Inter-process Communication

MPI is used to manage the distribution of the graph across processes and the communication of boundary node labels between them. Each process handles a subset of the nodes and sends/receives boundary labels to/from neighboring processes.

- **MPI Functions Used:**
  - `MPI_Bcast` to broadcast data (e.g., graph structure) from rank 0 to all other processes.
  - `MPI_Isend` and `MPI_Irecv` for non-blocking communication of boundary labels between processes.
  - `MPI_Gather` to collect the final labels at rank 0 after all iterations.

4.2 OpenMP for Intra-process Parallelism

Within each MPI process, OpenMP is used to parallelize the label propagation step. The propagation process updates the labels for all nodes in the subgraph simultaneously, using multiple threads.

- **OpenMP Parallelization:**
  - The #pragma `omp parallel for` directive is used to parallelize the loop that updates the labels of each node.
  - The number of threads is controlled by setting the `OMP_NUM_THREADS` environment variable, which specifies how many threads to use within each process.

5 Results

The program runs for a maximum number of iterations, checking for convergence after each iteration. The final labels are gathered from all processes at rank 0, and the execution time is measured.

At the end of the program, rank 0 will print:

- **Final Node Labels:** Each node's community assignment.
- **Total Execution Time:** The time taken for the entire algorithm to complete.

6 Code Snippets

Below are some important sections of the code.

6.1 Graph Partitioning:

```
1  Graph* partitionGraph(Graph *graph, int numProcesses, int
       processRank) {
2      int nodesPerProcess = graph->numVertices / numProcesses;
3      Graph *subgraph = malloc(sizeof(Graph));
4      subgraph->numVertices = nodesPerProcess;
5      subgraph->adjMatrix = malloc(nodesPerProcess * sizeof(int *));
6      subgraph->labels = malloc(nodesPerProcess * sizeof(int));
7
8      for (int i = processRank * nodesPerProcess; i < (processRank +
           1) * nodesPerProcess; i++) {
9          subgraph->adjMatrix[i - processRank * nodesPerProcess] =
               graph->adjMatrix[i];
10         subgraph->labels[i - processRank * nodesPerProcess] = graph
               ->labels[i];
11     }
```

```
12
13      return subgraph;
14  }
```

6.2 Label Propagation (OpenMP Parallelization):

```
1  #pragma omp parallel for shared(subgraph)
2  for (int i = 0; i < subgraph->numVertices; i++) {
3      int *neighborLabels = (int*) calloc(subgraph->numVertices,
           sizeof(int));
4
5      // Count the frequency of labels among neighbors
6      for (int j = 0; j < subgraph->numVertices; j++) {
7          if (subgraph->adjMatrix[i][j] == 1) {  // If there's an edge
               between node i and node j
8              neighborLabels[subgraph->labels[j]]++;  // Increment the
                   count for the label of node j
9          }
10     }
11
12     // Find the most frequent label among the neighbors
13     int maxCount = 0;
14     int mostFrequentLabel = subgraph->labels[i];  // Default to the
           current label
15     for (int j = 0; j < subgraph->numVertices; j++) {
16         if (neighborLabels[j] > maxCount) {
17             maxCount = neighborLabels[j];
18             mostFrequentLabel = j;
19         }
20     }
21
22     newLabels[i] = mostFrequentLabel;
23     free(neighborLabels);
24 }
```

7 Performance Evaluation

The performance of the parallel implementation is evaluated by measuring the total execution time. The execution time is measured using `MPI_Wtime()`, and it is printed from rank 0.

- **MPI Functions for Time Measurement:**
  - `MPI_Wtime()` is used to record the start and end times of the execution.
  - The total execution time is printed from rank 0 after the label propagation is completed.

8 Performance Evaluation Tables

Here are the tables for evaluating performance across different numbers of MPI processes and OpenMP threads. These tables are used to track the total execution time for various configurations of MPI processes and OpenMP threads.

8.1 Table 1: Total Execution Time vs Number of MPI Processes

Number of MPI Processes	Total Execution Time (s)	Speedup
2	2.154149	1.0
4	2.091330	1.03
8	2.078033	1.01
16	2.063530	1.01

8.2 Table 2: Total Execution Time vs Number of OpenMP Threads

Number of OpenMP Threads	Total Execution Time (s)	Speedup
1	2.396753	1.0
2	2.214473	1.08
4	2.137946	1.04
8	2.085974	1.02
16	2.033086	1.03

9 Conclusion

This project demonstrates how to efficiently perform community detection using the label propagation algorithm by leveraging both **MPI** for distributed parallelism and **OpenMP** for parallelism within each process. The hybrid parallelization significantly improves performance, especially for large graphs. The program successfully partitions the graph, propagates labels using parallel processing, and exchanges boundary labels between processes. The execution time is measured, and the final community labels are printed at rank 0.

10 Future Work

Future improvements could include:

- **Dynamic Load Balancing:** Implement dynamic partitioning of the graph based on node degrees to ensure more balanced work distribution.
- **Optimization of Communication:** Minimize the frequency of boundary communication or use non-blocking communication to reduce overhead.