Parallel Graph Traversal with a Web API

Erik Saule

Goal: Implement a breadth-first search (BFS) traversal algorithm in C++ that interacts with a web-based graph server, utilizing multithreading for parallel node expansion at each level. The program should take a starting node and a traversal depth as input, query the server for neighboring nodes in parallel, and return all nodes reachable within the given depth. The solution should implement a "Level by Level Approach" as explained below.

1 Programming

For reference, here is an overview of breadth-first search (BFS) from Geeks for Geeks:

Breadth-First Search (BFS) - GeeksforGeeks

Here's a step-by-step explanation of how BFS should work with multithreading:

- Start from the given source node.
- Instead of a queue that would be typical in BFS, use a vector for each level of the BFS to explore nodes level by level. (So likely a vector of vectors)
- Keep track of visited nodes to avoid cycles.
- Expand nodes up to the given traversal depth.
- Utilize multiple threads to fetch neighbors of nodes at the same level concurrently.
- Return all nodes visited within the depth limit.

You may start from the BFS code that you wrote in a previous assignment. Though it is likely that your previous submission does not follow a level by level structure. To simplify your work, we provide you a sequential implementation of level-by-level BFS. You can start your parallel implementation from that sequential implementation.

2 TODO: Parallel Level by Level Approach

The level-by-level approach ensures that all nodes at a given depth are fully expanded before proceeding to the next depth level. This method is well-suited for parallel execution because nodes at the same level can be processed independently and concurrently.

Determining the Number of Threads Per Level: Your code should determine the number of threads dynamically based on the number of nodes at that level. The approach follows these key principles:

- A **fixed maximum** number of threads is set to avoid excessive thread creation, which could overload the system, leading to server crashes.

- If there are fewer nodes than the maximum thread count at the current level, each node is assigned to its own thread.
- If there are more nodes than the maximum thread count at the current level, the nodes are evenly distributed among the available threads to ensure efficient load balancing.
- Each thread fetches neighbors for its assigned nodes and stores the results for the next level.
- A **mutex** is used to synchronize updates to shared structures like the visited set to prevent race conditions.

Role of Mutex in Multithreading:

Mutex (short for mutual exclusion) is used in the program to ensure safe access to shared data structures when multiple threads are executing concurrently. It helps prevent race conditions, where two or more threads attempt to read or modify a shared resource at the same time, leading to unpredictable behavior. In this implementation:

- A mutex is used to protect the **visited set**, ensuring that a node is added only once even if multiple threads encounter it simultaneously.
- The mutex also synchronizes updates to the **next level node storage**, preventing conflicts when multiple threads attempt to append new nodes at the same time.
- Without a mutex, there is a risk of duplicate nodes in traversal or memory corruption due to concurrent modifications.

2.1 Example Execution

Assume we start with a root node and set the depth to 2, with a maximum of 8 threads available. Below is how the BFS expands:

Level 0 (Start)

- Nodes to process: 1
- Threads used: 1 (only one node, so one thread is enough)
- Next Level: 3 nodes

Level 1

- Nodes to process: 3
- Threads used: 3 (one per node since there are fewer than 8 nodes)
- Next Level: 4 nodes

Level 2

- Nodes to process: 24
- Threads used: 8 (since 24 nodes exceed the maximum of 8, nodes are distributed among available threads)
- Each thread processes approximately an equal number of nodes.
- Threads execute in parallel, fetching neighbors for their assigned nodes.
- Once all threads complete execution, the BFS advances to the next level.

3 Interacting with the Web API

The graph server has been set up and is accessible at:

`http://hollywood-graph-crawler.bridgesuncc.org/neighbors/`

The API provides a single endpoint:

- **GET /neighbors/{node}**: Returns a JSON response containing all immediate neighbors of the given node.

Example API Call:

```
curl -s http://[ENDPOINT]/neighbors/Tom%20Hanks
```

Example Response:

```
{
    "neighbors": ["Forrest_Gump", "Saving_Private_Ryan", "Cast_Away"],
    "node": "Tom_Hanks"
}
```

Dataset Information. In case you are curious, the graph used in this assignment is built from the Bridges Actor/Movie Dataset. This dataset contains actors and the movies they have acted in, with edges representing the relationship between them. You can learn more about this dataset at: `https://bridgesuncc.github.io/tutorials/Data_WikiDataActor.html`

Though, the actual source of the dataset is irrelevant to your work.

4 Benchmarking and Testing

Your program should handle graphs of various sizes efficiently. Test your implementation by running it with different nodes and depths.

TODO: Evaluate the performance of your BFS implementation.

- Run the program with different starting nodes and traversal depths.
- Test for "Tom Hanks"/3 on Centaurus where node name = "Tom Hanks" and distance to crawl = 3.
- Measure execution time for different depths and graph sizes.

5 Submission

TODO: Submit an archive containing:

- Your C++ source code.
- A Makefile for compiling the code.
- A README explaining how to run the program.
- Example output logs.
- Timings with a sequential version and a parallel version.