Parallel Graph Traversal with a Web API Using Blocking Queue

Erik Saule

Goal: Implement a breadth-first search (BFS) traversal algorithm in C++ that interacts with a web-based graph server, utilizing a blocking queue pattern for efficient parallel processing. 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.

1 The Blocking Queue Approach

The blocking queue is a concurrent programming pattern that provides an efficient way to distribute work across multiple threads. This approach is particularly well-suited for graph traversal where the workload is unpredictable and I/O-bound operations (like web API calls) dominate the processing time.

1.1 What is a Blocking Queue?

A blocking queue is a thread-safe data structure that:

- Allows multiple producer and consumer threads to safely access the queue
- Blocks consumer threads when the queue is empty until new items are available
- Efficiently coordinates work distribution with minimal overhead
- Provides natural load balancing among worker threads

1.2 Implementation Details

For our graph traversal task, the blocking queue implementation consists of:

- A thread-safe queue that manages work items
- Synchronization mechanisms to coordinate access to the queue
- A way to signal when processing is complete

1.3 Parallel BFS Algorithm with Blocking Queue

Here's how the BFS algorithm works using the blocking queue approach:

1. Initialize a blocking queue and add the starting node with its depth information
2. Create a pool of worker threads
3. Each worker thread:
    - Pulls nodes from the queue (blocking if none available)
    - Processes the node by fetching its neighbors from the web API

- For each unvisited neighbor, marks it as visited and adds it to the queue
- Continues until the queue is empty and all threads are idle

4. When all threads finish, return the list of visited nodes

1.4 Thread Coordination Example

To illustrate how the blocking queue coordinates work, consider a BFS traversal starting from "Tom Hanks" with depth 4:

- Initially, the queue contains only "Tom Hanks" with depth 0
- Multiple worker threads are started, but only one finds work (Tom Hanks) while others wait
- The thread processing "Tom Hanks" finds several neighbors (movies and actors)
- As neighbors are added to the queue, waiting threads wake up and start processing them
- Threads dynamically process nodes as they become available, regardless of their depth level
- When a thread finishes processing a node, it immediately retrieves the next available node from the queue
- This continues until all nodes within depth 4 are processed

The blocking queue approach requires no explicit coordination of which thread processes which nodes - the queue handles this automatically, allowing faster threads to process more nodes.

2 Programming Task

Your task is to implement a parallel BFS algorithm using the blocking queue approach described above. Follow these steps:

1. Create a thread-safe blocking queue class that:
    - Has methods to add and remove items
    - Blocks threads when the queue is empty
    - Provides a mechanism to signal when no more items will be added
2. Implement the BFS algorithm:
    - Initialize data structures (visited set, result list)
    - Add the starting node to the queue
    - Create multiple worker threads
    - Have each thread process nodes from the queue until completion
    - HINT: Keep track of how many threads are currently working (to be able to detect algorithm termination)
3. Ensure thread safety:
    - Protect access to shared data structures
    - Properly synchronize thread termination
    - Handle race conditions when checking and updating the visited set
4. Determine an appropriate number of worker threads:
    - Use a fixed number of worker threads (e.g. 8) to handle parallelism efficiently while managing shared resources and ensuring that threads remain active during execution.

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"/4 on Centaurus where node name = "Tom Hanks" and distance to crawl = 4.
- Measure execution time for different depths and graph sizes.
- Compare the performance with different numbers of worker threads.

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.