Graph Query: Path Between Nodes
Technical Documentation

Darius Vincent Ardales
STDISCM - S13

I. Representation of the Graph

Table 1. Adjacency List

Node (string)	Adjacent Nodes (vector<string>)
a	[‘b’, ‘c’]
b	[‘a’, ‘c’]
c	[‘b’, ‘a’]

The graph is represented as an adjacency list in C++ using the standard libraries map and vector. The graph is defined as **Graph = map<string, vector<string>>;** wherein **Graph** is a type alias for a map, which is a collection of key-value pairs. The key is a **string** which represents a node in the graph. The value is a **vector<string>**, where each vector holds a list of strings representing adjacent nodes. A visual representation of the adjacency list is shown in Table 1.

II. The Thread Pool Class

```
private:
    vector<thread> workers;
    queue<function<void()>> tasks;
    mutex queueMutex;
    condition_variable condition;
    bool stop;
};
```

Figure 1. ThreadPool Private Members

In the ThreadPool class, the private members, as seen in Figure 1, are used to manage the workers, task queue, synchronization, and stopping mechanism. Here are its uses:

- **vector<thread> workers:** holds all the worker threads.
- **queue<function<void()>> tasks:** holds the tasks to be executed by the worker threads.
- **mutex queueMutex:** protects the tasks queue. It ensures that only one thread can access the queue at a time.

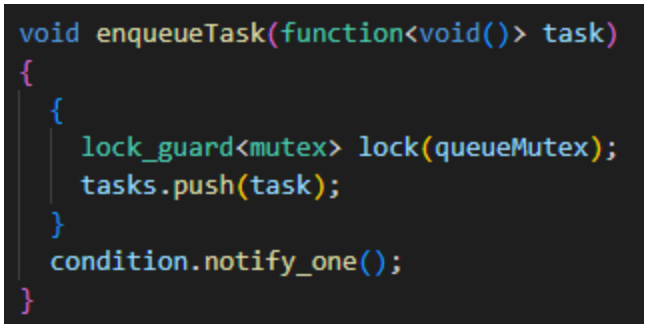
- **condition_variable condition:** used to synchronize the workers. It allows the threads to wait for tasks to become available in the task queue.
- **bool stop:** indicates whether the thread pool should stop accepting new tasks and terminate the workers.

```
ThreadPool(size_t numThreads) : stop(false)
{
  for (size_t i = 0; i < numThreads; i++)
  {
    workers.emplace_back([this]()
                         {
              while (true) {
                  function<void()> task;
                  {
                      unique_lock<mutex> lock(this->queueMutex);
                      this->condition.wait(lock, [this]() { return this->stop || !this->tasks.empty(); });
                      if (this->stop && this->tasks.empty())
                          return;
                      task = move(this->tasks.front());
                      this->tasks.pop();
                  }
                  task();
              } });
  }
}
```

Figure 2. ThreadPool Constructor

The ThreadPool Constructor, as shown in Figure 2, takes the number of worker threads to create and initializes the member variable stop to false (indicating that the pool is running). After that, the loop **for (size_t i = 0; i < numThreads; i++) { … }** creates the specified number of worker threads. Inside the loop, there is a **workers.emplace_back([this]() { ... });** line wherein we create a new thread and add it to the **workers** vector. The lambda expression **[this]()** means that the lambda "captures" the **this** (the class instance) pointer by value. This allows the lambda to access all the member variables and methods of the ThreadPool object (such as **queueMutex**, **tasks**, **condition**, and **stop**). This lambda also takes no arguments.

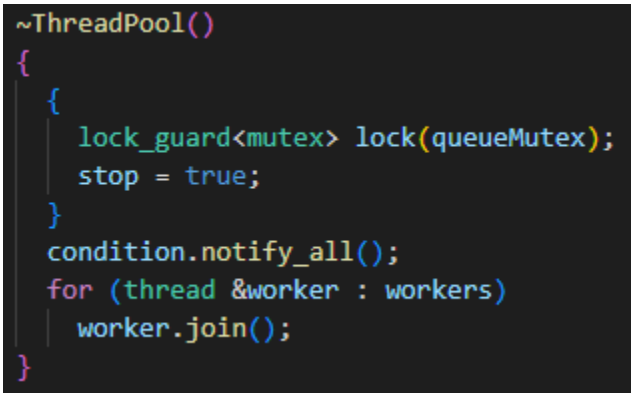
Inside the lambda, a **while (true)** loop is used so that each worker thread continuously acquires a task. Inside each thread, the **unique_lock<mutex> lock(this->queueMutex)** locks the shared **queueMutex** to protect access to the task queue. **unique_lock** is used instead of **lock_guard** for manual locking and unlocking via the condition variable. The code **this->condition.wait(lock, [this]() { return this->stop || !this->tasks.empty(); })** makes the thread wait and will only be unlocked if **stop** becomes true (indicating shutdown), or there is at least one task in the tasks queue. Once unlocked, it proceeds to the critical section of the code and encounters an if block. For the if block, if the thread pool has been told to stop and there are no remaining tasks, the worker thread exits. Otherwise, **task = move(this->tasks.front())** gets the task from the front of the queue and stores it inside **task**. The task is then removed from the queue using **this->tasks.pop()**. After all of that, the task is executed via **task()**.



```
void enqueueTask(function<void()> task)
{
  {
    lock_guard<mutex> lock(queueMutex);
    tasks.push(task);
  }
  condition.notify_one();
}
```

Figure 3. Enqueuing Tasks Function

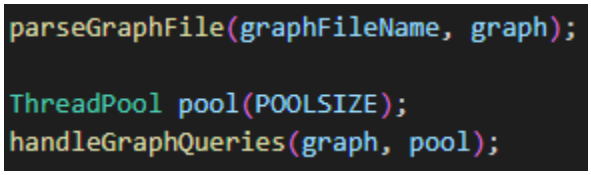
When enqueuing tasks for the ThreadPool to execute, the **enqueueTask** function, as seen in Figure 3, is used. The function takes a **function<void()>** object as its task. Then, it first locks the task queue (queueMutex) using a **lock_guard** to add the task safely. This means that **this->condition.wait()** in any of the worker threads will just keep waiting until the code goes out of the scope of **lock_guard**. After pushing the task into the tasks queue, it calls **condition.notify_one()** to wake up one waiting worker thread so that it can process the newly added task.



```
~ThreadPool()
{
  {
    lock_guard<mutex> lock(queueMutex);
    stop = true;
  }
  condition.notify_all();
  for (thread &worker : workers)
    worker.join();
}
```

Figure 4. ThreadPool Destructor

Finally, the destructor, as seen in Figure 4, is just to ensure that the thread pool is properly cleaned up and that all threads are safely terminated when the ThreadPool object is destroyed. This is done by first locking the queue and setting stop to true. Then, the destructor calls **condition.notify_all()** to wake up any threads that might be waiting. All threads are then joined to ensure that the program does not exit until every worker has finished.



```
parseGraphFile(graphFileName, graph);

ThreadPool pool(POOLSIZE);
handleGraphQueries(graph, pool);
```

Figure 5. Initialize ThreadPool

Now that the explanation for its implementation is done, we will initialize a **ThreadPool** object called **pool**, which will have **POOLSIZE** threads. **POOLSIZE** is set to **80** as an arbitrary choice, but mainly because my machine has 8 cores, and I multiplied it by 10. The queries will be handled through **handleGraphQueries(graph, pool);** wherein we give both the graph and the thread pool.

III. Design of Parallel "node" Query Algorithm

The node query algorithm basically divides the adjacency list into batches, and each batch is checked by the worker threads inside the ThreadPool in parallel. This allows for faster checking if a node is in the graph or not.

```
// Check if a node exists in the graph.
void checkNode(const string &node, const Graph &graph, ThreadPool &pool)
{
  int total = graph.size();
  int numTasks = min(POOLSIZE, total);
  atomic<bool> found(false);
  atomic<int> tasksRemaining(numTasks);
  mutex cvMutex;
  condition_variable cv;

  // Calculate how many nodes each task should check.
  int batchSize = (total + numTasks - 1) / numTasks;
```

Figure 6. checkNode Initial Variables

The node query algorithm starts by determining the total number of nodes in the graph and is stored in **total** as seen in Figure 6. Then, the task division is also determined by the **numTasks** variable, wherein it is the minimum of the **POOLSIZE** (number of threads in the thread pool) and **total** (number of nodes). After that, there are also atomic flags and mutex locks. The **atomic<bool> found(false)** is used to signal if a certain thread finds the target node. The **atomic<int> tasksRemaining(numTasks)** variable keeps track of how many tasks are still in progress. Each task will decrement this counter when it finishes its work. The mutex **cvMutex** is used to safely modify shared variables (like **tasksRemaining**), and the condition variable **cv** is used to notify the main thread when all tasks have been completed. Lastly, **batchSize** is the number of nodes each task will handle. This is calculated by dividing the total number of nodes by the number of tasks (**numTasks**).

```
// Vector of iterators that will mark the boundaries between tasks.
vector<Graph::const_iterator> boundaries;
auto it = graph.begin();
boundaries.push_back(it);
// Set boundaries by advancing the iterator in steps of batchSize.
for (int i = 0; i < numTasks; i++)
{
  // For the last task, ensure we go up to graph.end()
  if (i == numTasks - 1)
  {
    boundaries.push_back(graph.end());
    break;
  }
  advance(it, batchSize);
  boundaries.push_back(it);
}
```

Figure 7. checkNode Batch Boundaries using Iterator

Instead of copying the keys into a vector, boundaries are created for each task using iterators, as seen in Figure 7. **boundaries** is a vector that stores iterators, marking the start and end of each batch. The first boundary is set to **graph.begin()**, indicating the start of the graph. Then, for each task (except the last one), we move the iterator forward by **batchSize** steps using **advance(it, batchSize)**. Each advance call adjusts the iterator to the correct position for the current task. For the last task, we make sure to set the final boundary to **graph.end()**, ensuring that the last task processes all remaining nodes.

```
for (int i = 0; i < numTasks; i++)
{
  auto startIt = boundaries[i];
  auto endIt = boundaries[i + 1];
  pool.enqueueTask([&, startIt, endIt]()
                   {
    for (auto iter = startIt; iter != endIt && !found.load(); iter++) {
      if (iter->first == node) {
        found = true;
        break;
      }
    }
    if (--tasksRemaining == 0) {
      lock_guard<mutex> lock(cvMutex);
      cv.notify_one();
    } });
}
```

Figure 8. checkNode Enqueue Tasks to ThreadPool

Now, onto the fun part which is shown in Figure 8. This loop enqueues **numTasks** tasks, each of which will handle a batch of the graph. For each task, we capture **startIt** and **endIt** (iterators to the boundaries) in the lambda function. These iterators define the segment of the graph (batch) that each task will process. Each task iterates over its assigned segment or batch of the graph (from **startIt** to **endIt**). Then, for each node in the batch, it compares the key (**iter->first**) with the target node. If a match is found, the **found** flag is set to true, and the loop breaks, terminating the search for that task.

After completing its work, each task decrements **tasksRemaining**. Once the last task finishes (i.e., **tasksRemaining** becomes 0), it notifies the main thread using the condition variable **cv.notify_one()**.

```
// Wait until all tasks are finished.
{
  unique_lock<mutex> lock(cvMutex);
  cv.wait(lock, [&]()
          { return tasksRemaining.load() == 0; });
}
```

Figure 9. checkNode Wait for Main Thread

The main thread waits for all tasks to finish by using the condition variable **cv.wait()**, as seen in Figure 9. It will block until **tasksRemaining** reaches zero, indicating that all tasks have completed their work.

IV. Design of Parallel “edge” Query Algorithm

The edge query algorithm is almost the same as the node query algorithm. The only difference is that the task being enqueued to the ThreadPool is slightly more complex than just simply checking for the target node. This time, we have two target nodes, namely **src** and **dest,** wherein these two nodes signify an edge. To find the **src**, it is exactly the same as the node query algorithm wherein we divide the adjacency list into batches, and each batch is checked by the worker threads inside the ThreadPool in parallel. However, once the **src** is found, the task immediately goes inside that **src** and checks if it has the **dest** as one of its adjacent neighbors. **found** will be marked true if a neighbor is equal to **dest**. Otherwise, it shall be false. The codes in Figure 6, Figure 7, and Figure 9 can also be found in the edge query algorithm. The only difference is in how Figure 8, or the task, is enqueued, which will be explained below.

```
for (int i = 0; i < numTasks; i++)
{
  auto startIt = boundaries[i];
  auto endIt = boundaries[i + 1];
  pool.enqueueTask([&, startIt, endIt]()
                   {
    for (auto iter = startIt; iter != endIt && !found.load(); iter++)
    {
      if (iter->first == src)
      {
        // If src is found, check adjacency list for dest
        for (const auto &neighbor : iter->second)
        {
          if (neighbor == dest)
          {
            found = true;
            break;
          }
        }
      }
      if (found.load()) break;
    }
    if (--tasksRemaining == 0)
    {
      lock_guard<mutex> lock(cvMutex);
      cv.notify_one();
    } });
}
```

Figure 10. checkEdge Enqueue Tasks to ThreadPool

For each batch (defined by a pair of iterators, **startIt** and **endIt**), we enqueue a task in the thread pool, as seen in Figure 10. Then, the lambda captures **startIt** and **endIt** by value and everything else by reference (using **[&, startIt, endIt]**). Within each task, each node in that batch is checked if the key **iter->first** matches **src**. If **src** is found, It then iterates through that node's adjacency list stored in **iter->second** to see if any neighbor equals **dest**. If the desired edge is found, it sets the atomic flag **found** to true. The loop is guarded by **!found.load()**, so if one task finds the edge, other tasks will eventually break out of their loops to avoid unnecessary work.

After finishing its search, each task decrements the **tasksRemaining** counter. When the last task finishes, the task acquires **cvMutex** and calls **cv.notify_one()** to wake up the main thread.

V. Design of Parallel “path” Query Algorithm

The main idea behind the design of the parallel path query algorithm using breadth-first search is to search the graph level by level in parallel. Meaning that we first explore all neighbors of the **src** (level 1), then all unvisited neighbors of those nodes (level 2), and so on. This guarantees that when we first encounter the destination (**dest**), it means that, theoretically, the shortest path is found from **src** to **dest.** The current level being explored in parallel is what we call a **frontier**. Thus, the **nextFrontier** is the set of nodes that have been discovered (visited) at the current level. These are the nodes that will be processed next.

```
// Find and display a path from src to dest using Breadth-First Search (BFS).
void bfsPath(const string &src, const string &dest, const Graph &graph, ThreadPool &pool)
{
  map<string, bool> visited;
  map<string, string> parent;
  for (const auto &entry : graph)
    visited[entry.first] = false;
  visited[src] = true;

  vector<string> frontier = {src};
  atomic<bool> found(false);
```

Figure 11. bfsPath Initialization

The **visited** map keeps track of whether a node has been visited. Initially, every node is marked false (unvisited), and then the source (**src**) is marked as visited. The **parent** map will store the parent of each node, which is the node from which it was discovered. It is used later to reconstruct the path once the destination is found. Then, the **frontier** is initialized to contain only the source node. This represents the starting point of our BFS or Level 1. Finally, an atomic boolean **found** is used to indicate if the destination (**dest**) has been discovered in the process. Since multiple threads update this flag, it is atomic for thread safety. This strategy can also be found in **checkNode** and **checkEdge**.

```
// Process BFS level by level.
while (!frontier.empty() && !found.load())
{
  int total = frontier.size();
  int numTasks = min(POOLSIZE, total);
  atomic<int> tasksRemaining(numTasks);
  // Each task will record its discovered nodes as (neighbor, parent) pairs.
  vector<vector<pair<string, string>>> localCandidates(numTasks);
  mutex cvMutex;
  condition_variable cv;
  int batchSize = (total + numTasks - 1) / numTasks;
```

Figure 12. BFS Loop per Frontier (Level by level)

As can be seen in Figure 12, the loop runs as long as there are nodes in the current **frontier** and the destination has not been found. Also, the algorithm for dividing the current frontier into batches so that threads can work on each batch in parallel is also the same as how the tasks are divided into batches in **checkNode** and **checkEdge**. The only difference is the **localCandidates** vector of vectors. The purpose of this is so that each task will fill its own local vector with pairs of (neighbor, parent) for the neighbors it discovers. The goal of doing this is to **minimize lock contention** because tasks do not need to update a shared data structure for each neighbor they discover (and it maybe less hassle to implement synchronization for updating the **parent** vector in parallel).

```
// Launch tasks to process batches of the current frontier.
for (int i = 0; i < numTasks; i++)
{
  int start = i * batchSize;
  int end = min(start + batchSize, total);
  pool.enqueueTask([&, i, start, end]()
                   {
            vector<pair<string, string>> candidates;
            for (int j = start; j < end && !found.load(); j++)
            {
                string node = frontier[j];
                // Process all neighbors of this node.
                for (const auto &neighbor : graph.at(node))
                {
                    if (found.load()) break;  // Early exit if dest already found.
                    // Collect the candidate without locking.
                    candidates.push_back({ neighbor, node });
                    if (neighbor == dest)
                        found = true;
                }
            }
            localCandidates[i] = move(candidates);
            if (--tasksRemaining == 0)
            {
                lock_guard<mutex> lock(cvMutex);
                cv.notify_one();
            } });
}
```

Figure 13. Tasks to Process the Current Frontier

So now we move on to the actual part with action in the parallel BFS implementation, as seen in Figure 13, which is processing batches of the current frontier. Here, the **frontier** is split into batches based on **start** and **end** indices. For each batch, a task is enqueued into the thread pool. The lambda captures the partition indices (**i, start, end**) by value and other variables by reference.

Inside the task, a local vector **candidates** is created to store discovered neighbors and the corresponding parent for the batch. Then, the task loops through its batch of the **frontier**, and for each node, it processes all of its neighbors through **for (const auto &neighbor : graph.at(node))**. If a neighbor equals the destination (**dest**), the **found** flag is set to true. Each discovered **neighbor** is also added to the local candidates along with the current node as its parent. After processing, the task moves its local **candidates** vector into the corresponding slot in **localCandidates**. The task decrements the **tasksRemaining** counter. If it's the last task to finish (counter becomes 0), it notifies the waiting main thread, just like how it worked in **checkNode** and **checkEdge**.

```
// Wait for all tasks for this level to complete.
{
  unique_lock<mutex> lock(cvMutex);
  cv.wait(lock, [&]()
          { return tasksRemaining.load() == 0; });
}

// Merge the local candidate lists and update visited and parent.
vector<string> nextFrontier;
for (const auto &candList : localCandidates)
{
  for (const auto &p : candList)
  {
    const string &neighbor = p.first;
    const string &par = p.second;
    if (!visited[neighbor])
    {
      visited[neighbor] = true;
      parent[neighbor] = par;
      nextFrontier.push_back(neighbor);
    }
  }
}
frontier = move(nextFrontier);
```

Figure 14. Wait for all Tasks to Complete at Current Level and Merge Results

In Figure 14, the main thread waits on the condition variable until all tasks for the current level have finished processing. Once all tasks are complete, the local candidate lists (each a list of (**neighbor, parent**) pairs) are merged. For each candidate, if the **neighbor** has not been visited, it is marked visited, its **parent** is recorded, and it is added to the **nextFrontier**. Finally, the current **frontier** is replaced with the **nextFrontier,** ready for processing the next level.

```
// Reconstruct and display the path if found.
if (found.load())
{
  vector<string> path;
  string current = dest;
  while (current != src)
  {
    path.push_back(current);
    current = parent[current];
  }
  path.push_back(src);
  reverse(path.begin(), path.end());

  cout << "Path: ";
  for (size_t i = 0; i < path.size() - 1; i++)
    cout << path[i] << " -> ";
  cout << path.back() << endl;
}
else
{
  cout << "No path from " << src << " to " << dest << endl;
}
```

Figure 15. Path Reconstruction

Finally, as you can see in Figure 15, if the path is found, the path will be printed out using the **parent** map and this implementation is the same as how it is implemented in the sequential version. Starting from the destination, it uses the **parent** map to trace back to the source. Then, the collected path is reversed so that it starts from **src** and ends at **dest**. The path is then printed in a readable format.