ChatGPT

Parallel BFS and DFS in C++ with OpenMP

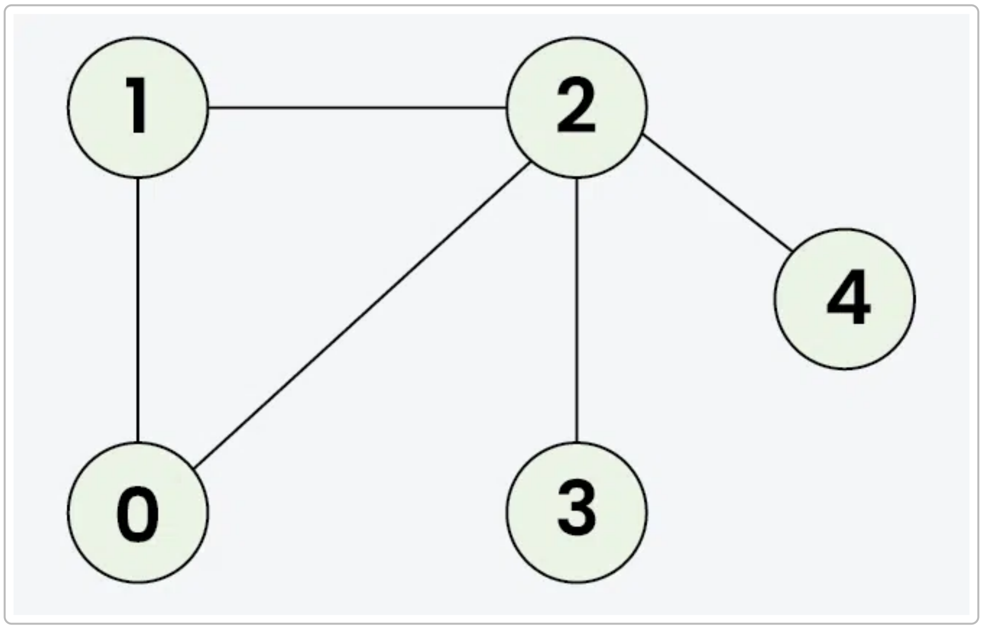
Graph Representation (Adjacency List)

In C++, a common way to store a graph is with an **adjacency list**, often using a `vector<vector<int>>`. Here `adj[i]` is a list of all neighbors of node `i`. For example, after reading the number of vertices `V`, we can do:

```
vector<vector<int>> adj(V);
// For each edge (u, v) read from input:
adj[u].push_back(v);
adj[v].push_back(u);  // if the graph is undirected
```

The above code shows that each edge is added to the lists of both nodes. In other words, `adj[i]` holds exactly the nodes adjacent to `i` [1]. This structure makes it easy to loop over all neighbors of any node when we do BFS or DFS.

Breadth-First Search (BFS) – Concept & Algorithm



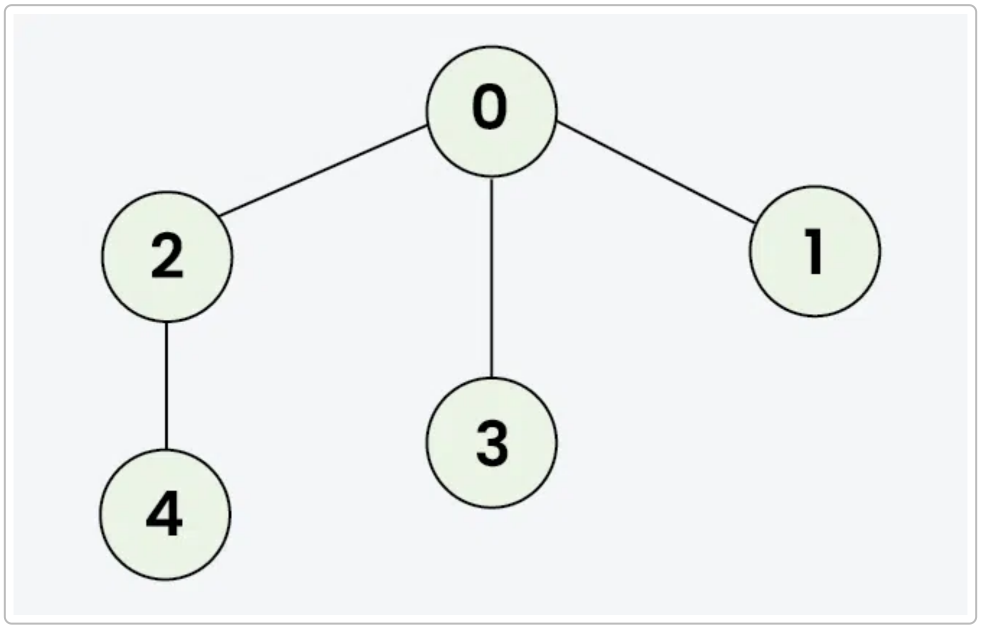
Breadth-First Search (BFS) explores a graph level by level, starting from a chosen start node. In BFS, we first visit the start node, then all its neighbors, then neighbors-of-neighbors, and so on. We use a **queue** to process nodes in the order they were discovered. Concretely:

- **Step 1:** Mark the start node as visited and enqueue it.
- **Step 2:** While the queue is not empty:
- Dequeue the next node `u`. Print or process `u`.
- For each neighbor `v` of `u`: if `v` is not yet visited, mark it visited and enqueue it.

This ensures we visit nodes in order of their distance from the start. For example, in the graph above starting at node `0`, BFS might visit nodes in the order `[0, 1, 2, 3, 4]`, visiting all neighbors of `0` before moving deeper [2]. In code, a `visited` array (e.g. `vector<bool> visited(V,false)`) is essential to avoid revisiting nodes, since graphs may contain cycles [3].

Key points about BFS:
- Uses a queue (`std::queue<int>`).
- Visits nodes “breadth-first” (by distance layers).
- Relies on a `visited[]` array to prevent processing a node more than once [4].

Depth-First Search (DFS) – Concept & Algorithm



Depth-First Search (DFS) explores a graph by going as deep as possible along each branch before backtracking. Starting from the root, DFS picks one neighbor, recursively explores all of its descendants, and only then backtracks to explore other neighbors. In a graph with cycles, we again use a `visited` array to avoid infinite loops. For example, in the graph above, one DFS order from node `0` could be `[0, 2, 4, 3, 1]`, depending on neighbor order.

A typical DFS procedure is:

- **Step 1:** Visit the start node, mark it `visited`, and print it.
- **Step 2:** For each neighbor `v` of this node: if `v` is not visited, recursively DFS on `v`.

This is very similar to a preorder traversal of a tree [5]. The recursion (stack) naturally goes deep first. As with BFS, we use a boolean `visited[]` array to avoid revisiting the same vertex [5].

Key points about DFS:
- Uses recursion (or an explicit stack).
- Visits nodes "depth-first" (go deep before backtracking).
- Also needs a `visited[]` array for graphs with cycles [5].

Parallel BFS in C++ with OpenMP

In a sequential BFS, we would loop over the queue one node at a time. To parallelize, we let multiple threads process different nodes of the current frontier simultaneously. A sketch of a parallel BFS loop might look like:

```
vector<bool> visited(V, false);
queue<int> q;
visited[start] = true;
q.push(start);

#pragma omp parallel    // start a parallel region
{
    while (!q.empty()) {
        int layerSize = q.size();  // number of nodes at current level

        #pragma omp for     // divide the next for-loop among threads
        for (int i = 0; i < layerSize; i++) {
            int u;
            #pragma omp critical
            {
                u = q.front(); q.pop();   // safely pop from queue
            }
            cout << u << " ";  // process node u

            // Enqueue neighbors in parallel
            for (int v : adj[u]) {
                if (!visited[v]) {
                    visited[v] = true;
                    #pragma omp critical
                    {
                        q.push(v);  // safely push into queue
                    }
                }
            }
        }
        // Implicit barrier: all threads synchronize here before checking while condition again
    }
}
```

- We mark the start node visited and enqueue it.
- Inside `#pragma omp parallel`, all threads repeatedly share work from the queue. We get the current queue size, then use `#pragma omp for` so that each thread handles a different index `i`.
- Popping from (and pushing to) the shared `std::queue` must be inside a **critical section** (`#pragma omp critical`) to avoid race conditions [6]. Only one thread pops or pushes at a time.
- After popping `u`, each thread processes `u` (for example, prints it). Then it loops over neighbors `v`. If `v` was not visited, we mark it and push it onto the queue (inside another critical section).
- We use the shared `visited[]` array to ensure no node is enqueued twice.

This parallelization allows multiple nodes in the same BFS level to be handled concurrently. In practice, however, the reliance on `#pragma omp critical` for queue operations means the queue itself remains sequential, which can limit speedup. Still, the work of exploring neighbors (`for (int v : adj[u])`) can happen in parallel across threads, and heavy processing inside the loop can benefit from multiple threads.

Parallel DFS in C++ with OpenMP

A parallel DFS is done by letting different branches of the search tree run in parallel. We typically combine `#pragma omp parallel`, `#pragma omp single`, and `#pragma omp task`. For example:

```
vector<bool> visited(V,false);

// Recursive DFS function using tasks
void dfs_task(int u, vector<bool>& visited) {
    visited[u] = true;
    cout << u << " ";  // process u

    // For each neighbor, spawn a new task if not visited
    for (int v : adj[u]) {
        if (!visited[v]) {
            #pragma omp task shared(visited)
            {
                dfs_task(v, visited);
            }
        }
    }
    // Implicit taskwait at end of function ensures child tasks complete before returning
}

int main() {
    // ... (read graph and start) ...
    #pragma omp parallel
    #pragma omp single  // ensure only one thread starts the DFS
    {
        dfs_task(start, visited);
    }
    // At this point, the DFS is complete.
}
```

Explanation:
- We use `#pragma omp parallel` + `#pragma omp single` so that one thread begins the DFS (other threads wait). The `single` directive means “only one thread executes the next block” [7].
- In `dfs_task(u)`, we mark `u` visited and print it. Then for each neighbor `v` we check `visited[v]`. If not visited, we do `#pragma omp task` and recursively call `dfs_task(v)`. This creates a new **task** that can run in parallel with other tasks [8].
- Each recursive task will similarly spawn further tasks for its neighbors. The OpenMP runtime will schedule these tasks on available threads, exploring different branches of the DFS tree in parallel.
- We rely on OpenMP’s implicit `taskwait` at function exit to ensure that all child tasks of a node complete before that function returns.
- As with BFS, we must be careful with shared data. The `visited[]` array is shared, so in theory two tasks could check-and-mark the same neighbor at the same time. In robust code, one might use a critical section or atomic operation when writing `visited[v] = true` to avoid a race. Here we assume either the graph has no duplicate edges or accept a small race (common in simple examples). In production code, protecting `visited[v] = true` with `#pragma omp critical` or `atomic` would be safer.

OpenMP Directives Explained

OpenMP uses special `#pragma omp` directives to control parallelism:

- `#pragma omp parallel`: Starts a parallel region; the code block after this will be executed by multiple threads simultaneously.
- `#pragma omp for` (often combined as `#pragma omp parallel for`): Splits the iterations of a for-loop among the threads. In BFS above, `#pragma omp for` divided the loop over the queue among threads [9]. In general, `parallel for` tells OpenMP to divide loop iterations among threads so they run in parallel.
- `#pragma omp single`: Specifies that a block of code should be executed by only one thread. In our DFS example, it ensures only one thread calls the DFS function initially [7]. Other threads skip the single region.
- `#pragma omp task`: Defines a task – a unit of work that can be executed by any thread. When a thread encounters `#pragma omp task`, it packages that block of code as a new task that can run in parallel with other tasks [8]. This is useful for recursive or irregular workloads like DFS.
- `#pragma omp critical`: Marks a section of code that must be executed by only one thread at a time. It prevents race conditions on shared data [6]. For example, queue operations in parallel BFS are placed inside a `critical` section so that only one thread pops or pushes at once.

Each directive helps coordinate threads. For instance, without `critical`, two threads might pop from the queue simultaneously and corrupt it; with `critical`, they take turns. However, excessive use of `critical` or `single` can limit parallel speedup, because they force serialization of parts of the code.

Input and Output

Typically, in `main()` we read the graph from the user like this: first read the number of vertices `N` and edges `M`. Then read `M` pairs of nodes `(u, v)`, and for each do:

```
adj[u].push_back(v);
adj[v].push_back(u);
```

to build the adjacency list [1]. Next, read the start node index. We create a `visited` array of size `N` initialized to `false`. Then we call our traversal functions, e.g. `parallelBFS(adj, start)` and the parallel DFS routine shown above. Each traversal prints nodes as it visits them (using `cout << node << " "`), which will appear in the correct BFS or DFS order. For example, for the graph above starting at node `0`, a BFS might print `0 1 2 3 4` while a DFS might print `0 2 4 3 1` (depending on neighbor order). The `visited` checks ensure we don't print any node more than once and that the output sequence truly reflects the intended BFS or DFS order.

---

[1] **Graph and its representations | GeeksforGeeks**
https://www.geeksforgeeks.org/graph-and-its-representations/

[2] [3] [4] **Breadth First Search or BFS for a Graph | GeeksforGeeks**
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[5] **Depth First Search or DFS for a Graph | GeeksforGeeks**
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

[6] **#pragma omp critical [explained with example]**
https://iq.opengenus.org/pragma-omp-critical/

[7] **#pragma omp single**
https://www.ibm.com/docs/SSLTBW_2.4.0/com.ibm.zos.v2r4.cbclx01/prag_omp_single.htm

[8] **#pragma omp task**
https://www.ibm.com/docs/en/zos/2.4.0?topic=processing-pragma-omp-task

[9] **OpenMP Directives | Microsoft Learn**
https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170