# Task F – Simple Graph

// Graph.hpp

A graph is composed of vertices and edges.

Each vertex has a property and a unique id.

Since we read the graph from a file we can't ensure that the nodes ids are contiguous [0..N]

If we adopt an std::vector we'll have many empty spots with no property/inexistent node

So we suppose that nodes ids range can be non contiguous

We can use a linked list to represent vertices which is convenient for adding nodes O(1) back or front but linear in time to retrieve or delete a node.

Another possibility is a hash table which has constant time O(1) for lookup and insert. However, it takes more memory to perform these operations in an average constant time

PGX uses large amount of data (Millions of vertices) and hash table is better than most STL data structures (http://scottmeyers.blogspot.de/2015/09/should-you-be-using-something-instead.html)

Choosing std::unordered\_map because we are concerned about performance than memory footprint.

In addition to that STL doesn't use contiguous space to allocate memory for the std::unordered\_map using standard allocator but it does for vector.

Vector is better for dividing it into chunks because it has a RAI pointer but hash table doesn’t have a forward pointer which can be solved using bucket inside the hash table

Memory usage for 40M nodes using GCC 4.4 unordered\_map is about 1.5GB.

which is huge and not optimal for old systems but also we can think about parallelism and how we can divide a graph into sub graphs for a better space consumption and performance.

Concerning the edges representation, we can use an adjacent list or an edge matrix.

The adjacent list can be implemented by combining a list of lists

or a hash table of list. The later representation uses a two-dimension array where edges[i][v] is a Boolean which represent if there is an edge from i to v.

Operations to be performed on this data structure are find adjacent nodes, find if there is an edge from I to V.

With hash table it can be constant but with linkedlist of list we may reach a quadratic time.

//GraphAlgorithm.hpp

is\_weakly\_connected:

we need to verify that for each pair of vertex (u, w) that there is a path.

suppose we have N vertex; we check for each vertex has path with N-1 other vertices

which N\*(N-1) but since u, w order is not important because it's an undirected graph we have to find N/2\*(N-1) paths

Total execution time will be then at LEAST O(N^2\*(N + E)) where O(N+E) refer to a DFS or BFS to find a path

between u and w

We notice that a lot of work is redundant like 0->1->2->3 we look for different path

0: (0,1) (0, 2) (0,3)

1: (1, 2) (1, 3)

2: (2,3)

From the example we can deduct a new strategy to verify if it's weakly connected because

(1,2) && (1,3) is included in (1,3) path

(0,1) && (0, 2) && (1,3) are included in (0,3) path

so if we find a path from (0,3) we have proved that is weakly connected.

Thus, for a graph G(V,E) we choose a random "u" vertex and try to visit all nodes from that point

if we visit |v| vertices. It means it's weakly connected. this can be done in linear time

O(|V|+|E|)

is\_fully\_connected:

It refers to a complete graph, since we are representing our edges in a matrix.

IF each row has N-1 edges a part from self-edge it means the graph is fully connected where N is number of vertices

Another way to do it is while loading the graph we calculate the number of edges we have without counting self-edges and store it as a class member ..

The later approach has a constant time O(1) because we know that we will compare it to N\*(N-1) while the first one is O(N) in time

GetShortestDistances: Algorithm of my choice

This algorithm allows us to determine the shortest distance in term of number of edges from a start node to all nodes within the graph.

Some nodes may be unreachable will have infinite value (UNIT64 MAX)

To do so, we can run the Bread First Search N-1 Time and get the distances we are looking for. This approach cost for a graph(V, E)

O(|V|(|V|+|E|) so it has a quadratic time.

From the approach, we can deduct that there is a repetitive work, which is exploring the graph.

On the node side, the minimal distance from this node to a start node is the minimum of all paths heading to this node.

With a Breath First Search, we explore adjacent nodes before going deeper and we go over all possible paths. So each time at a given node, we update the distance of its adjacent nodes by taking the mininum.

Then, we are going from quadratic to linear time O(|V|+|E|) and O(|V|) in time complexity.

--- Parallelism

I thought first about executing each separate task within a thread but the issue here (as Siegfried stated) we may have millions of threads running. And the CPU may not be able to create that much threads.

Then, I used a constant number of thread each time using the available cores within a computer. This time, work is balanced between available threads.

Again, there was another optimisation, as Siegfried stated, is to use one thread pool running in the background and reuse it for all algorithms that we may want to run.

The advantage here is we only create threads once and notify them when a job is available.

-When using stack/queue each available thread look for an element within the data structure to consume and push alongside if necessary.

-When using a hash table, and to perform a traversal of this data structure I used the bucket\_count to know how many buckets are within this hash table then divide it to available threads. So each thread will be doing a traversal but for determined window of buckets.

---Thread pool

Our thread pool contains X threads running in the background where X represents how many threads can be executed at the same time.

The current API allow to add jobs to a concurrent queue by passing the reference of the ThreadPool. Since multiple jobs are related to one algorithm, I thought of monitoring the number of job added/finished. But this approach is limited if the Thread Pool has multi-threaded access. Then, I added a tag for each algorithm, so that we know when an algorithm has finished executing.

Drawbacks of our implementation and what we could do better

* While the hash table is convenient for our use cases. We could do better, if we could read/write simultaneously we could combine reading the graph and running algorithms.

(We can’t because since hash table might rehash after an insert/delete and iterator become invalidated ..)

* If we have information about the graph, like density, directed vs undirected, algorithms to perform. We could improve data structures currently used.
* We could decide which version parallel vs sequential algorithm to use based on the graph’s nature (nodes size, edges, available threads...) Parallel algorithms have a constant overhead due to creation of threads. This enhacement would need to profile each algorithm based on defined criterias.
* We could choose whether to create manually thread or use the std::async
* (Work balance in DFS) While implementing the parallel DFS few options are available about how to manage this concurrency. In this implementation each non waiting thread does push ALL adjacent nodes to a stack, we could push ALL - 1 adjacent nodes to the stack for the waitings threads and keep one for the current thread and do it again ..

//Testing

For testing, I started with sample examples in the task

Then tried edges cases for each algorithm

Finally, benchmarked the time execution for small and large graph (up to 1000 nodes) between sequential and parallel algorithm

Parallel algorithms are efficient starting for a graph of 50 nodes.