15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Graphs II **Date:** October 4th
Scribe: Kanat Tangwongsan

12.1 Today

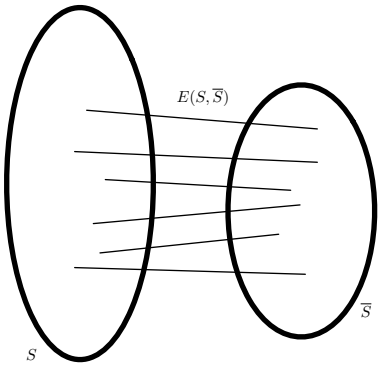
- Minimum Spanning Tree
- Breadth-first search (BFS)
- Reachability (in a dense graph)

So far what we have seen in the course followed the pattern: there is a sequential algorithm X for problem Y, and here is how to parallelize it efficiently. There are, however, problems that people don't know how to solve efficiently in parallel. Graph problems are typically harder than what we looked at thus far. For example, we don't know of any work efficient single-source shortest-path algorithm. We know that DFS is **P**-complete, and we are unable to solve any **P**-complete in poly-logarithmic depth yet.

For max-flow, it is easy to get an $O(n^2)$-depth work-efficient algorithm. The algorithm is based on the preflow-push method. The challenge is to reduce the depth and maintain work efficiency.

12.2 Minimum Spanning Tree

Given an undirected, connected graph $G = (V, E)$ and a weight function, $w : E \to \mathbb{R}_+$, the minimum spanning tree problem is to find the spanning tree on G that minimizes the sum of edge weights. Well-known sequential algorithms for this problem include Kruskal's algorithm, Prim's algorithm, and Boruv̊ka's algorithm. Throughout this discussion, we assume edge weights are unique. To devise an efficient parallel algorithm for this problem, we recall the following lemma about minimum spanning trees.



A cut $(S, \overline{S})$ along with the edges across the cut $E(S, \overline{S})$.

Lemma 12.2.1 *For any $S \subseteq V$, let $E(S, \overline{S})$ denote the set of edges with one endpoint in S and the other in $\overline{S}$. That is, $E(S, \overline{S})$ is precisely the set of edges going across the cut $(S, \overline{S})$. Then for any non-empty $S \subset V$ with $S \neq V$, the edges in $E(S, \overline{S})$ with the smallest weight belongs to the spanning tree.*

The proof is a simple exercise. Curious readers can try to prove it or consult a standard textbook (e.g., CLRS).

12.2.1 Applying Graph Contraction

The crucial idea for getting this to work is the lemma we just stated above—i.e., to realize that for a given node or supernode (contracted group of nodes), the minimum edge out of that node must belong to the minimum spanning tree. Thus we can use an algorithm similar to the connectivity algorithm, but when we hook the edges in the tree, we always pick edges that are the minimum incident on a node, instead of picking random edges. Assuming we have concurrent write with priority (when multiple writes occur, the processor with the lowest ID wins), we can sketch an algorithm as follows.

1. Sort the edges in ascending order of weights.
2. Repeat until a single node remains: (each of the following steps is carried out in parallel)
   (a) Randomly flip coins to assign nodes to parents and children.
   (b) Each edge writes the weight it has to both the endpoints.
   (c) Every node checks if the edge that "wins" the write is a parent-parent edge. If so, undo the write.
   (d) Contract and remove self-edges.

Claim 12.2.2 *In each round the algorithm removes $n/4$ nodes in expectation, where n is the number of nodes at the beginning of that round.*

Proof: Let X_v be the indicator random variable for "the node v is removed." We define X_v for all nodes v in the current graph. We know that v is removed if and only v is a tail (parent) and the other end of the winning edge is a node with a heads (child), so $\mathbf{E}[X_v] = 1/4$. It follows that the expected number of nodes removed is $\mathbf{E}[\sum_v X_v] = \sum_v \mathbf{E}[X_v] = n/4$, where n is number of nodes in the graph at the beginning of the round. ■

The claim implies that the algorithm here has expected work $O(m \log n)$ and expected depth $O(\log n)$.

Can we do it without concurrent write with priority? One way to determine the edge to use would be to use the adjacency array representation for the graph. Then for each node we just do a `min` on the weights and that gives us the edge to use. The problem with this is that the adjacency list representation needs to be updated when doing a contraction. We need to be able to merge multiple arrays into one (in arbitrary order). This can be accomplished using scans. Suppose we want to merge arrays of length $i_1, i_2, \ldots, i_k$. One way to do this is to make the j-th element of the ℓ-th array goes to offset $j + \sum_{w=1}^{\ell-1} i_w$ in the new array. Following this scheme, we do a scan on

$[i_1, \ldots, i_k]$. This gives us the offsets at which an array has to start writing to, and we start writing in parallel.

12.3 Breadth First Search (BFS)

We will discuss a parallel algorithm for performing breadth-first search in work $O(|E| + |V|)$ and depth $O(w)$, where w is the "width" of the graph. Consider the graph in Figure 12.3.1.

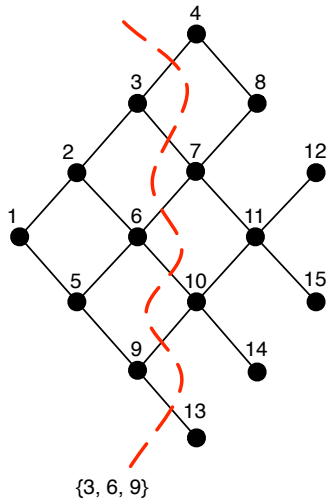


Figure 12.3.1: Sample graph

A parallel breadth-first search works much like the serial algorithms. Pick a starting node in the graph (say 1) as the root of the tree, and put this initial node into a set called the frontier set {1}. As we add a node to the frontier set we mark it as having been visited.

For each node in the frontier set, add the edges that connect that node to any other node which has not yet been visited ((1,2) and (1, 5)) to the tree, and then form a new frontier set with these new nodes {2, 5}, marking them as having been visited. Repeat the process until all nodes have been visited.

In the serial case this works fine, because in the next step node 6 will get connected to either node 2 or node 5 but not both, since one or the other will come across it first, add the edge to the tree, put the node into the frontier set and mark it as visited. When the other node encounters it will see that it has been visited and not use it.

In the parallel case both node 2 and node 5 could try to add the edge to 6, thus putting node 6 into the frontier set twice. This would quickly cause the algorithm to explode exponentially. Thus, we now have to be careful about how we add nodes to the frontier set, in order to avoid duplicates.

The basic algorithm goes as follows. Represent the graph as an adjacency array—each node has a list of nodes that it's connected to. Then create the frontier set which represent the nodes that we have just discovered. The initial frontier set is just the root node.

Mark all the nodes in the frontier set as having been visited. In parallel, grab all the nodes that are connected to nodes in the frontier set, remove any duplicates, remove any nodes that have already visited, and call this the new frontier set. Repeat this process until the frontier set becomes empty.

In the example above we would have as an initial frontier set {1}. We would then grab {2 5} and remove duplicates and nodes that have been visited (none in this case). Then we would grab everything connected to 2 or 5, {3 6 6 9}, remove duplicates to get {3 6 9}, mark these as visited and then grab {4 7 7 10 10 13}. To remove duplicates we can simply use an auxiliary array of length $|V|$, which each element writes its index into. So for example, the array {4 7 7 10 10 13} would write 0 into location 4, write 1 and 2 into location 7, and so forth. Now each elements reads the value back from the same location and if the index is not its own, it drops out. For example, either 1 or 2 will be written into location 7 and one of them will read the other's index back, and therefore drop out.

12.4 Shortest Paths and Reachability

As we discussed earlier, no one knows of a work-efficient single-source shortest-path algorithm. Even though Dijkstra runs in $O(m + n \log n)$, the best parallel algorithm has pretty much the same work as the all-pairs case. The basic idea is to represent the graph a suitable matrix. Raising it to the appropriate power gives the solution. For example, let $M = A + I$ where A is the Boolean adjacency matrix representation of a graph G. We know that $(M^n)_{i,j}$ counts the number of paths of length $\leq n$ from i to j. This immediately gives a parallel algorithm for the reachability problem with work $O(m_n \cdot \log n)$. Here m_n denotes the work for matrix multiplication. We will probably get into more details next lecture.