Parallel Algorithms and Programming
Week 6

Kishore Kothapalli

October 22, 2008

Chapter 6

Parallel Graph Algorithms

6.1 Introduction

Graphs play an imoprtant role in Computer Science given the ability to model several problems as graph-theoretic problems. In the sequential setting also, efficient algorithms for graph problems continues to be an active area of research and pedagogy. In this chapter, we study algorithms for some of the fundamental graph problems such as finding the connected components of a given graph, to find an MST of a given weighted graph, and a shortest paths problem.

Unless otherwise specified, we deal with undirected graphs denoted $G = (V, E)$ with $|V| = n$ and $|E| = m$. There are two ways to represent graphs: one using an $n \times n$ matrix called the adjacency matrix, A, and two using adjacency lists where the adjacency list of a node v consists of all the neighbours of v in a linked list.

6.2 Connected Components

The connected components problem can be defined as follows. Given an undirected graph $G = (V, E)$ partition the vertex set into disjoint sets $V_1, V_2, \cdots$, so that for every pair of vertices u, v in V_i, there exists a path from u to v. Clearly, this partition forms equivalence classes under the realtion of having a path.

The algorithm we study resembles the familiar Union-Find data structure. The idea is as follows. Suppose we have an initial set of trees where vertices in each tree are in the same connected component. Two trees T_1 and T_2 are combined together to form a bigger tree if there exists vertices $v \in T_1$ and $w \in T_2$ so that $vw \in E$. This process results in what we call as *super-verties*. For the next iteration, we can take the graph to be graph with the vertex set being the super-vertices and the edge set so that there is an edge $v_s w_s$ whenever there exists a pair of vertices $v \in v_s$ and $w \in w_s$ so that $vw \in E$. Notice that we are talking of a sequence of graphs, one for each iteration where the vertices of G_{i+1} are the super-vertices from G_i. Doing this repeatedly will result in the required partition of V.

That completes the intuitive description of the algorithm. However, several implementation details remain. How do we arrange these supervertices? How to represent the graph in the algorithm? How to build the graph of super-vertices as the algorithm proceedes? How many iterations are required?

In the following, let us try to address these questions. Firstly, for simplicity let us represent the input graph as its adjacency matrix. Initially, the adjacency matrix is an $n \times n$ matrix. If n_k super-vertices remain after the kth iteration, then the adjacency matrix for the next iteration, A_{k+1} is an $n_k \times n_k$ matrix. To populate this matrix, we just have to see if for two vertices of U and W in G_k, there exists a pair so that $u \in U$ and $w \in W$ and $uw \in E$. Then, $A_{k+1}(V, W) = 1$ and 0 otherwise. If we assume any reasonable concurrent write model, this step is easy to implement.

For the above step, one further detail is required. It should be easy to check if v and w are in the same supervertex after the k iteration. For this, we require that every node $v \in V$ be given a label so that if two vertices are in the same super-vertex then they have the same label. Initially, since each vertex is treated as a super-vertex, the label is same as the node label itself. But as we combine two trees, the labels have to updated. So. let us assume that the label of a node will be the index of the smallest node in the tree. To find this, one can simply make the node of the smallest label as the root of the tree and use pointer jumping as we combine trees so that the label can be set correctly.

Given the above details, let us now develop the entire algorithm. We have to describe how to start with an initial set of trees. For this, let us define a function $C : V \rightarrow V$ on the vertex set of the graph G as follows. We let $C(v) = \min\{w | A(v, w) = 1\}$, i.e., the smallest numbered neighbour of v. We can view this function also defining a set of directed trees (plus a cycle) on V with the edge set given by $(v, C(v))$. Some properties of this definition one can show immediately are:

- The forest of trees thus created indeed partitions V into sets $V_1, V_2, \cdots, V_j$ so that all the vertices in each partition are in the same connected component.
- Each cycle in the forest is either a self-loop or of length 2,

The above properties are easy to show by virtue of construction. We use the $C()$ function to initially come up with the required set of trees. The entire algorithm is now given below.

Algorithm ConnectedComponents(G)
Let $G_0(V_0, E_0) = G$
$A = A_0, i = 1$
for each vertex $v \in V_0$ do in parallel
set $C(v) = \min\{w | A(v, w) = 1\}$, if no such w exists, then set $C(v) = v$.
While $n_k > 0$ do
Shrink the set of directed trees obtained using C to a star using pointer doubling
Define the set of roots of each star with at least two vertices as the set of supervertices, V_i
Compute the adjacency matrix A_i corresponding to vertex set V_1
set $A = A_i$ and set $i = i + 1$
end-while
Compute the component number for each vertex by reversing the process of shrinking.

An example is given below to explain the algorithm. **Example.** The graph is shown in Figure 6.1 along with the set of trees obtained by the C function. The shrinking of the trees and the resulting stars are shown in Figure 6.2. The adjacency matrix for the next iteration is also shown in 6.2.

□

6.3 All Pairs Shortest Paths

We now study some parallel algoritms for shortest path related problems in directed graphs. First, we study an algorithm for computing the transitive closure of a directed graph.

6.3.1 Transitive Closure

Suppose A represents the adjacency matrix of a directed graph $G = (V, E)$. The transitive closure problem asks for a matrix B such that $B(i, j) = 1$ if and only if either $i = j$ or if there exists a directed path from i to j in G.

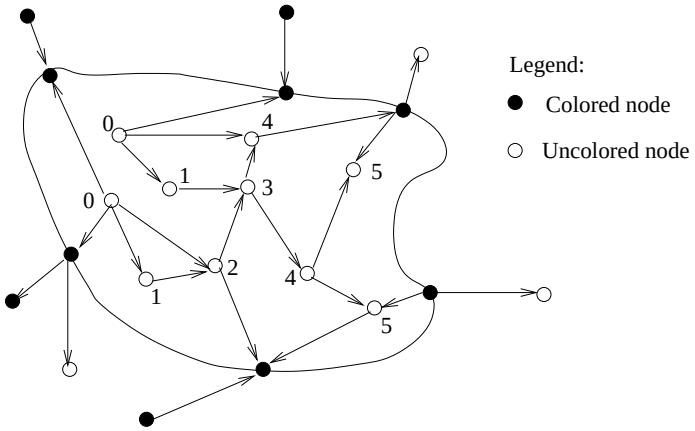
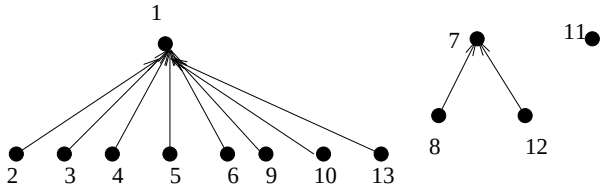


Figure 6.1: An example graph and its initial set of trees



A_1	1	2	3
1	0	1	0
2	1	0	0
3	0	0	0

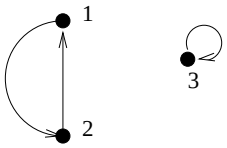


Figure 6.2: The working of the connected component algorithm.

The computation of transitive closure is greatly simplified if one solves the problem of matrix multiplication in parallel efficiently. We know that matrix multiplication can be solved in $O(\log n)$ time using $O(n^3)$ operations for two $n \times n$ operations. However, if the entries of the matrices being multiplied are from a ring, then matrix multiplication of two $n \times n$ matrices can be performed in $O(\log n)$ time but using only $O(n^{2.376})$ operations. We denote this by $M(n)$. A special case of the matrix multiplication algorithms is multiplication of two Boolean matrices. In the CRCW model, this can be done in $O(1)$ time using a total of $O(n^3)$ operations.

To compute the transitive closure, we first observe that the required matrix is $B = (I + A)^{2^{\lceil n \rceil}}$ where I is the identity matrix of size n. The claim can be shown by induction on the shortest path length between i and j. If $i = j$ then the addition of I ensures that $B(i, j) = 1$. For $i \neq j$, we claim that $(I + A)^r$ will have a 1 where the shortest path between i and j has a length of r. This is because of the fact that if i and j have a path of length r, then there exists a vertex k such that i and k have a path of length $r - 1$ and k to j has a path of length 1. This ensures that $(I + A)^r$ will have the (i, j)th entry as 1.

So we just have to find out how to compute a matrix power. This is also easy as we can compute A^k by repeated squaring. Hence we have:

Lemma 6.3.1 *The transitive closure of a directed graph G can be computed in $O(\log n)$ time using $O(n^3 \log n)$ operations on the CRCW PRAM or in $O(\log^2 n)$ time using $O(M(n) \log n)$ operations on the CREW PRAM.*

6.3.2 All Pairs Shortest Paths

The problem is to compute for every pair of vertices u and v in a given weighted directed graph G the weight of the shortest directed path from u to v. There is a straight-forward reduction from all-pairs-shortest-paths to matrix multiplication. Replace the $+$ operation by $\min$ and the $\times$ operation by $+$ in matrix multiplication. Under this replacement, compute A^n where A is the weighted adjacency matrix of G. Thus, we have:

Lemma 6.3.2 *APSP in $O(\log^2 n)$ time, $O(n^3 \log n)$ operations.*

6.4 Minimum Spanning Tree

A spanning tree of a graph $G = (V, E)$ is a subgraph $T = (V_T s, E_T)$ such that $V_T = V$ and $E_T \subseteq E$. In the case of a weighted graph G with a weight function $w : E \rightarrow \mathbb{R}^+$, a minimum spanning tree is a spanning tree that has the lowest possible weight where the weight of a tree is defined as the sum of the weights of its edges. Spanning trees and minimum spanning trees arise in many applications in graph theory. As we design an algorithm for obtaining an MST of a given weighted undirected graph G, we assume that the edge weights are all distinct.

The following lemma aids us in the design of an MST in the PRAM model.

Lemma 6.4.1 *Let $G = (V, E)$ be a weighted undirected graph. Let $C(v)$ for a vertex $v \in V$ denote the neighbor of v with the least cost edge incident on v. Then,:*

- *All the edges $(v, C(v))$ belong to the MST.*
- *The function $C : V \rightarrow V$ defines a pseudoforest such that any directed cycle has a length of two.*

Proof. For 1, suppose that for some vertex v, the edge $(v, C(v))$ is not in the MST, T. Let the path between v and $C(v)$ in T be $v\ v_1\ v_2 \cdots\ v_k\ C(v)$. Adding the edge $(v, C(v))$ to this path yields a cycle. In this cycle, removing the edge (v, v_1) creates another spanning tree T' whose weight is smaller than T. So T is not an MST.

For 2, notice that if the pseudoforest defined by C has a cycle of more than 2 edges, then the tree T defined by the edges of the form $(v, C(v))$ also has a cycle. Since T is a MST and all these edges are part of the MST, this implies that no cycles of length more than two can exist. Self-loops are also not possible as $(v, C(v)) \in E(G)$. □

The above lemma can be generalized for a partition of vertices as follows.

Lemma 6.4.2 *Let $V = \cup_i V_i$ be a partition of V. Let $G_i = (V_i, E_i)$ be the subgraph induced by V_i. Let e_i^* be the edge of minimum weight in $E(G) \cap V_i \times V \setminus V_i$, that connects a vertex in V_i to the rest of the graph. Then all such edges e_i belong to an MST of G.*

Using the above ideas, we now present an algorithm given by Sollin for MST in the PRAM model.

6.4.1 Sollin's Algorithm

This algorithm is similar to that of the algorithm for connected components we saw earlier. The essential difference is the definition of the function C and its implementation.

Algorithm ConnectedComponents(G)
1. Let $G_0(V_0, E_0) = G$, and $W_0 = W$, $n_0 = n$, and $k = 1$.
2. While $n_k > 1$ do
   /* Define C values */
   for each vertex $v \in V_k$ do in parallel
      Set $C(v) = u$ where $W_k(v, u) = \min\{W_k(v, u)\}$.
      Mark $(v, C(v))$.
   Shrink the set of directed trees obtained using C to a rooted star using pointer doubling
   /* Update for the next iteration */ set $k = k + 1$, n_k to the number of rooted stars, W_k to be the weights of the graph of n_k vertices induced by viewing each star as a single vertex.
   end-while
3. Output the marked edges. Notice that the marked edges have to be translated to their original label.

In the following, we study the runtime of the above algorithm and some implementation issues. One of the difficult issues in this algorithm is to construct the matrix W_k from the matrix W_{k-1}. The central operation is to find for two roots $r_1, r_2 \in V_{k-1}$ the weight of the edges between r_1 and r_2 in W_k. The required weight is $\min\{W_{k-1}(u, v) | C(u) = r_1, C(v) = r_2\}$. To compute this, we proceed as follows.

Group the vertices of w_{k-1} according to the definition of C and for each group find the minima with respect to every vertex in the group. This can be seen as resulting in a matrix of size $n_{k-1} \times n_k$. The step involves sorting according to C values and the weights in W_{k-1}. Now, perform a similar kind of grouping on the rows of the $n_{k-1} \times n_k$ matrix similarly. It can be then seen that these operations can be done in $O(\log n_{k-1})$ time using a total of $O(n_{k-1} \log n_{k-1})$ operations.

Finally, we have that a MST of an undirected graph can be comptued in $O(\log^2 n)$ time using $O(n^2)$ operations on a CREW PRAM.