Distributed Data Structures and Algorithms for Disjoint Sets in Computing Connected Components of Huge Network

Wing Ning Li, CSCE Dept. University of Arkansas, Fayetteville, AR 72701
wingning@uark.edu

Thomas A. J. Schweiger, Acxiom Corporation, Fayetteville, AR 72701
Tom.Schweiger@acxiom.com

Abstract
A network or graph consists of a set of nodes joined by edges. A set of nodes are form a connected component when every vertex has a path to every other node in the set. The connected components of a network can be computed efficiently using depth-first search or disjoint-sets data structures. The disjoint sets approach has the advantages of easily answering queries whether two nodes are in the same connected component and efficiently maintaining the connected components as new edges are added. For huge network with hundreds of millions or billions of nodes, the disjoint set data structures may be too large to be held in memory by a single processor. To solve this problem, we propose a scheme to partition and distribute the disjoint set data structures among multiple processors in a computing grid.

1. Introduction
A set of nodes are form a connected component when every vertex has a path to every other node in the set. The connected components of a network can be computed efficiently using depth-first search or disjoint-sets data structures [1]. The disjoint sets approach has the advantages of easily answering queries whether two nodes are in the same connected component and efficiently maintaining the connected components as new edges are added. This latter feature allows the network view to be dynamic, and simplifies maintenance of the disjoint set as the network evolves over time.

For huge network with hundreds of millions or billions of nodes, the disjoint set data structures may be too large to be held in memory by a single processor. For instance, many 32-bit based processor systems have the maximum size of virtual memory of four gigabytes (4GB) and user applications may use up to two gigabytes of memory space for all heap, stack, and global stores. Programs that require more than 4GB memory space cannot be compiled and run. To find the connected components for a network with two billions nodes, storing the disjoint set as an array of four-byte integers, requires eight gigabytes of memory.

To address this problem, we propose a scheme to partition and distribute the disjoint set in a computing grid so that each processor holds a manageable subset of the data structures. Reconsidering the above example, one could partition the 8 GB integer array into eight subsets. Each array subset is allocated to a processor in a computing grid with adequate pooled resources to manage the problem in memory, say eight processors. In this case, each processor requires 1GB of memory, well below architecture limits.

In general, the fewer the sub arrays the better our scheme performs. A sophisticated partitioning algorithm should take into account the configuration of the available resources. A processor with larger main memory should get a larger piece of the array than a processor with fewer resources.

Three basic operations are required to determine the connected components of a network: MAKE-SET(x), UNION(x,y), and FIND-SET(x) [1], where x and y denote set elements or objects (these operations will be described in detail later). If, by chance, the nodes in all the connected components happen to be allocated to the same partition (that is, no edge connects nodes allocated to different partitions), then a trivial solution is to let each processor construct the disjoint set for its sub array, and only these three operations are needed.

In practice, however, many edges will connect nodes allocated to different partitions. Therefore, in addition to the MAKE-SET(x), UNION(x,y), and FIND-SET(x), additional operations are needed which we will introduce.

The remainder of our paper is organized as follows. In section two we review disjoint sets data structures and algorithms and we introduce basic definitions and terminology. In section three we provide examples to illustrate the key ideas and principle operations of our distributed algorithm. In section four we present our
algorithm. Finally we offer concluding remarks and suggest future work.

2. Preliminaries

A network is an undirected graph of nodes which for our purposes are numbered sequentially from 1 to \( n \). Once \( n \) is given, the set of nodes is implicitly understood. An edge is a connection between a pair of nodes, written as \((x, y)\), where \( x \) and \( y \) are integers between 1 and \( n \) that denote the connected nodes. Thus a network is uniquely specified by \( n \) and its set of edges.

Two nodes may not be directly connected by an edge, but may be connected by an indirect path. In graph theoretic term, the path from node \( u \) to node \( v \) is the sequence of nodes starting at \( u \) and ending at \( v \) for which each pair of consecutive nodes in the sequence is connected by an edge. A node \( u \) is said to be reachable from \( v \) if the there exists a path between them. For undirected graphs this property is reflexive; if \( v \) is reachable from \( u \), \( u \) is also reachable from \( v \). An graph is said to be fully connected if every pair of nodes is reachable. If a graph is not fully connected, it is said to be comprised of connected components. Connected components are fully-connected, non-overlapping subgraphs of the larger network.

We find connected components using a disjoint-set data strucure -- a collection of disjoint dynamic sets. Each disjoint set in the collection represents a member of the graph. Disjoint-set data structures support the operations \( \text{MAKE-SET}(x) \), \( \text{UNION}(x,y) \), and \( \text{FIND-SET}(x) \). \( \text{MAKE-SET}(x) \) creates a new set whose only member is \( x \). \( \text{UNION}(x,y) \) unites the dynamic sets that contain \( x \) and \( y \) respectively into a new set that is their union (a standard set operation). \( \text{FIND-SET}(x) \) returns a “pointer” to the representative of the unique set that contains \( x \).

In a most efficient implementation of disjoint sets, rooted trees represent sets, with each tree node containing one member and each tree representing one set. By using two heuristics—“union by rank” and “path compression”—we can achieve the asymptotically fastest disjoint-set data structure known [1].

In [1], the following pseudocode is presented to implements disjoint-set data structure. The parent of \( x \) is designated as \( p[x] \) and the rank of \( x \) as \( \text{rank}[x] \) in the code.

\begin{verbatim}
MAKE-SET(x)
1. p[x] = x
2. rank[x] = 0

UNION (x,y)
1. LINK(FIND-SET(x), FIND-SET(y))
\end{verbatim}

\begin{verbatim}
LINK(x,y)
1. if \( \text{rank}[x] > \text{rank}[y] \) then
2. \( p[y] = x \)
3. else
4. \( p[x] = y \)
5. if \( \text{rank}[x] = \text{rank}[y] \) then
6. \( \text{rank}[y] = \text{rank}[y] + 1 \)

FIND-SET(x)
1. if \( x \neq p[x] \) then
2. \( p[x] = \text{FIND-SET}(p[x]) \)
3. return \( p[x] \)
\end{verbatim}

Note that indentation is used to indicate block structure in the pseudo-code. The reader is referred to [1] for in depth discussion and amortized analysis of the algorithms.

For a disjoint set whose elements are integers from 1 to \( n \), rooted trees may be implemented by an integer array \( p[1..n] \), where \( p[k] \) contains the index of the array at which the parent of \( k \) is located. By using clever programming and the observation that rank values are needed only by the roots of the trees, we can store both the parent and rank information in the \( p[] \) array and eliminate the rank[] fields or array completely.

To use a disjoint-set data structure to compute the connected components of a given network sequentially, we initially place each node in its own set using the \( \text{MAKE-SET} \) operation. Thus each \( x \) is initially a “connected component”. Conceptually, this is a graph with nodes but no edged.

Next we begin to add edges to the graph. For each edge \((x,y)\), we test if \( x \) and \( y \) belong to different sets using the return values of \( \text{FIND-SET}(x) \) and \( \text{FIND-SET}(y) \). If so, the two sets are united to form a larger connected component using the \( \text{UNION}(x,y) \). Once all the edges are processed we construct the final connected components by determining which nodes are in the same set. The reader is referred to [1] for a thorough discussion of the algorithm.

As we explained in the introduction section, for huge network with billions of nodes this sequential approach becomes intractable due to the memory limitation imposed by processor architecture. To address this issue we use distributed multiple processors in a computing grid to gain access to a larger amount of memory. This requires a different view of the disjoint set data structure array. It must be partitioned and distributed among multiple processors in a computing grid in such a way that each processor can hold a subset of the data structure internally.

To begin, we partition the array \( p[1..n] \) into \( k \) sub arrays \( p[n_0..n_1] \), \( p[n_1+1..n_2] \), \( p[n_2+1..n_3] \), ..., \( p[n_{k-1}+1..n_k] \), where \( n_0 = 1 \) and \( n_k = n \). Within this scheme we recognize
two kinds of edges: local edges and global edges. An edge (x,y) is a local edge if x and y are in the same partition; that is, for some j, x and y are in the range \( n_{j-1}+1..n_j \). If the edge spans two partitions it is a global edge.

As we mentioned earlier, if all edges are local edges, then no connected components span partitions. In this case the solution is a trivial parallel and distributed algorithm and no additional operations are required. In practice, however, the network will partition such that there are many global edges.

One solution for finding connected components is to implement UNION and FIND-SET operations over distributed sub array with parent indices pointing back and forth between different processors. This solution is sequential in nature and very inefficient.

Instead, we propose first processing just the local edges. This gives the connected components that are found in the range \( n_{j-1}+1..n_j \) for each partition j. We call these connected components the local components of partition j. Local components are computed using the sequential algorithm on local edges.

Two local components on different partitions will belong to the same connected component if and only if a global edge connects a node in one local component to a node in the other local component. The next step of our algorithm examines the global edges so that we can merge local components on different partitions.

Note that our initial view of local edges may be insufficient to determine all the local components. For example, two local components on the same partition may belong to the same connected component, but insufficient information is available at this time to join them.

Consider the example of in Figure 1. The larger circles represent partitions or processors. Small circles represent nodes in the network. Lines connecting small circles are edges. Lines connecting small circles within a larger circle are local edges.

By inspection one can see that the network is fully connected -- there is only one connected component in the network. The eight nodes in each partition should form a local component. However, if we consider only local edges we derive one local component for the lower-left partition and four local components, each of which has two nodes, for the remaining partitions.

To see how local components on the same partition may be merged by considering global edges, consider the global edges of lower-left partition. It has four global edges, two of which connect to nodes in the upper-left partition and two to nodes in the lower-right partition. The global edges to the upper-left partition generate a path joining two local components in that partition – the second and fourth local components are joined via the path through global edges to the single local component in the lower left partition. Similarly, the global edges to the lower-right partition generate a path joining two local edges in that partition. Such a pathway has the same effect as adding a new local edge in the partition. In general, k global edges from a “current local component” to another partition generate paths equivalent to k-1 new local edges in the second partition.

Global edges not only generate paths that join local edges, they also generate new paths between local partitions on different partitions, and these in turn may generate additional paths between local edges in the next iteration. As we saw, the global edges from the local component in lower-left partition connect to the upper-left and lower-right partitions. Note how this generates a new pathway between the upper-left and lower-right partition that has the same effect as adding a new global edge in the network. In general, for k partitions reached by global edges leaving a “current local component”, k(k-1)/2 new global edges are generated for each pair of partitions.

Consider the example of in Figure 1. The larger circles represent partitions or processors. Small circles represent nodes in the network. Lines connecting small circles are edges. Lines connecting small circles within a larger circle are local edges.

By inspection one can see that the network is fully connected -- there is only one connected component in the network. The eight nodes in each partition should form a local component. However, if we consider only local edges we derive one local component for the lower-left partition and four local components, each of which has two nodes, for the remaining partitions.

To see how local components on the same partition may be merged by considering global edges, consider the global edges of lower-left partition. It has four global edges, two of which connect to nodes in the upper-left partition and two to nodes in the lower-right partition. The global edges to the upper-left partition generate a path joining two local components in that partition – the second and fourth local components are joined via the path through global edges to the single local component in the lower left partition. Similarly, the global edges to the lower-right partition generate a path joining two local edges in that partition. Such a pathway has the same effect as adding a new local edge in the partition. In general, k global edges from a “current local component” to another partition generate paths equivalent to k-1 new local edges in the second partition.

Global edges not only generate paths that join local edges, they also generate new paths between local partitions on different partitions, and these in turn may generate additional paths between local edges in the next iteration. As we saw, the global edges from the local component in lower-left partition connect to the upper-left and lower-right partitions. Note how this generates a new pathway between the upper-left and lower-right partition that has the same effect as adding a new global edge in the network. In general, for k partitions reached by global edges leaving a “current local component”, k(k-1)/2 new global edges are generated for each pair of partitions.
Figure 2 shows symbolically the new local and global edges generated by the pathways created by the global edges the lower left partition. The blue colored edges symbolize new local edges and the red colored edge symbolizes a new global edge.

Recognizing that adding such new edges creates path shortcuts is an important step that we use extensively. Our distributed algorithm has three steps that all run in parallel:

1. Each processor computes its current local components using disjoint-set operations.
2. Each processor computes new local edges and new global edges for other partitions based on its current local components and existing global edges.
3. Each processor transmits information about new local and global edges to the interested processors.

The three steps are repeated until no new local or global edges are created. The maximum number of repetitions is determined by the number of partitions.

3. Illustrative Examples.

Let us first consider the example network shown in Figure 3. The larger circles represent partitions, each a process on a grid host, and the smaller circles represent nodes in the network. Lines connecting small circles are edges. In this example all the edges are initially global edges; each vertex forms a locally connected component by itself. This completes step 1.

For step 2, the global edges radiating from the top partition generates new local edges for the bottom partition: (F, G) via B, and (G, H) and (H, I) via C. Simultaneously, the global edges radiating from the bottom partition generates new local edges for the top partitions: (A, B) via F, (B, C) via G, and (C, D) and (D, E) via I.

For step 3 the partitions transmit the new local edges.

Returning to step 1, the partitions process the new local edges using the disjoint set algorithm. The effect of the new local edges is to join the vertices of each partition into a single local component. A global edge is stored from each local component to any other local component, indicating they belong to the same connected component though residing in different partitions.

Now consider again the network shown in Figure 1. The results of completing the first iteration of the three steps are shown in Figure 4, with new local edges drawn in blue and new global edges drawn in red. Note that the topmost local component in the upper-right partition generates a new local edge for the upper-left partition and a new global edge between the first local component of the upper-left partition and the third local component of the lower-right partition.

![Figure 4: New edges generated after iteration one.](image)

We indicate by the bold line those global edges that need to be stored by the partitions to record which local components are connected to make connected components.

Figure 5 shows the local components after processing the new local edges generated in first iteration. Nodes having the same color belong to the same local component. Figure 5 also shows the new global edges generated by the first iteration. The heavy black and red lines are stored global edges. The heavy blue lines are new local edges.
The new edges after iteration two are shown as light blue colored lines in Figure 6. At this point, as expected, each processor has a single local component after processing the new local edges.

3. The Basic Algorithm.

Without loss of generality, we subdivide an array of size \( n \) into \( 2^k \) consecutive partitions of which each is assigned to a grid resource. Based on this partitioning, edges are classified into two types: the local edges and global edges.

The disjoint-set FIND-SET and UNION algorithms are used as before to process local edges. Global edges are used to generate new local edges for other partitions. Global edges are also used to generate new global edges for the next iteration of the same process. We have shown that after at most \( k \) iterations all local components are found. Hence, if the array is partitioned in two, one iteration is needed. If the array is partitioned in four, two iterations are needed; if the array is partitioned in eight, three iterations are needed; and so on. The final set of global edges link the connected components that are partitioned among the grid processors. A MPI implementation of the proposed algorithm is reported in [2]. The problems for which the proposed distributed algorithm solves are considered in [3,4,5].

5. Conclusions

A fast distributed algorithm was developed for finding the connected components of massive networks. The distributed disjoint set allows us to maintain the connected components information in memory. This property will be investigated and explored further to implement a connected components service and a transactional connected components computation where connected components information is updated as new edge pairs are introduced.

Acknowledgments

This research was supported in part by Acxiom Corporation through the Acxiom Laboratory for Applied Research.

References


