Wait-Free Parallel Disjoint-Set Data Structure for CPUs and GPUs

Manas Thakur (CS13D023)
Mohammad Shuaib (CS13M029)
Nizamudheen Ahmed (CS14S014)
Shashidhar G. (CS14S022)

1 Introduction

In this work we strive towards implementing and comparing the parallel disjoint-set data structure on CPUs and GPUs. The disjoint set data structure is inherently fast (bound by the Ackermann function), but we find scope for parallelization and try to utilize it in order to speedup the connectivity-checking for large graphs. We proceed by designing a correct concurrent algorithm for the union-find data structure; then implement it on the x86 architecture in C++, as well as for GPUs using the CUDA language. We also compare the two concurrent implementations with the sequential one. The results verify that though the union-find data structure is inherently fast, it can still be supplied as a parallel library to take advantage of the underlying multicore or GPU architecture for getting an even faster implementation for large social-network graphs.

1.1 Disjoint-Set Data Structure

The disjoint-set data structure is used to partitioning of a set. It is useful for checking the connectedness in large graphs and for various other applications. For example, it can be used to detect the people forming groups or circles on social networks; it is the standard way to form a minimum spanning tree in weighted undirected graphs; and can be used to determine whether two vertices in a graph are in the same connected component.

The data structure used to represent disjoint sets is widely called the union-find data structure; based on the two primary operations it supports. Actually there are standard operations agreed to be supported by the union-find data structure: makeSet, find, and union. The makeSet operation creates a node consisting of the parent and rank information for a vertex [2]. Each disjoint set in the union-find data structure is represented by a particular node, called the *representative* of that set. The find operation returns the representative of the set to which a node belongs. The union operation merges two sets headed by their representatives, so that all the elements of that set have the same representative after performing the union operation. The concept of rank can be used to decide the next representative (called *union by rank*).

In this project, we have completely omitted the concept of union by rank. We reckon that for checking the connectedness (the application used), the concept of rank does not alter the set to which a node belongs to, at the end. As the inputs have nodes as integers, we use the node IDs to decide which way to merge two disjoint sets while taking their union.

Earlier it was believed that the complexity of operations in the union-find data structure [3] is bound by the function log^*n, which grows very slowly. However, later it was shown to be bound by the Ackermann function [4], which is among the slowest growing functions known till date.

1.2 Concurrent Union-Find

There are several challenges one faces while writing a correct and efficient concurrent algorithm for the union-find data structure. As several threads may be performing merge operations on the same connected component, it needs to be made sure that they are following the conventions without disturbing the correctness aspect. For example, when a thread decides to change the representative of a node, it has to take care of two facts: the new representative may itself have attached itself to another node, or someone else may have attached itself to it. In the former case, we don't run into trouble as later if we find the representative of the older node, it will actually lead to the latest one by traversing up the tree. In the second case, however, the rank of the new representative may have been altered by the other subtree trying to attach. Here we argue that the implementation does not actually change the set to which a node belongs, and so it is also acceptable.

It needs to be taken care that the older node (which is being attached to a newer representative) itself has not been attached to some other set by another thread. For this, we compare the node as well as its rank with the older values, and perform a Compare&Swap (CAS) operation on it if the conditions were satisfied. Otherwise, the union is retried (wait-freedom is proved in the results section).

1.3 CUDA Programming in GPUs

CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and computing model created by NVIDIA for GPUs. Using CUDA, GPUs can be used for general-purpose programming too, apart

```
find(x) {
   //Copy x's parent into par
   par = x.parent;
   //Move up the tree until par == x
   while(par != x) {
      /*Take grandparent for
              path-compression*/
      gpar = par.parent;
          /*Try to perform path-
              compression*/
      CAS(x.parent, par, gpar);
          //Move up by changing pointers
      x = par;
      par = x.parent;
   }
   return x;     // Return the parent
}
```

Figure 1. Parallel find algorithm

```
union (x, y) {
   while(true) {
      p_x = find(x); p_y = find(y);
      if(x < y) {
         /*y needs to be merged into x*/
         if(CAS(y.parent, p_y, p_x)) {
            return;
         }
      }
      else {
         if(CAS(x.parent, p_x, p_y)) {
            return;
      }
   }
}
```

Figure 2. Parallel union algorithm

from traditional graphics rendering work. The CUDA platform is highly useful for programs in which a large number of threads need to be executed instead of a single, fast thread. We use CUDA for implementing the GPU version of union-find data structure, so that the union operations are performed on the GPU device instead of the traditional x86 architecture. The CPU and GPU versions are compared in later sections.

The parallelism in the algorithm is extracted by throwing more threads and to do independent operations in parallel. CUDA operates on data parallelism, and the way to efficiently extract parallelism is to construct the data-structure so that each thread accesses different data and the data dependency is taken care of.

2 Approach and Implementation

We wrote a parallel algorithm for performing the union and find operations in parallel, so that all points of concern discussed in Section 1.2 are taken care. Then we implement it using C++ for x86 architectures, and using CUDA for GPUs.

2.1 The Algorithm

The wait-free parallel algorithms for find and union operations are given in figures 1 and 2, respectively.

2.2 Data Representation

The Input graph is given as a list of edge(u,v). Adjacency Matrix and Adjacency List are the popular representation for the graph. But these representation will become inefficient if the input graph is Sparse. We need a better method to represent the nodes in the graph. For representing Sparse Graph we have used three Arrays. The 'NodeArray' array is an array of node elements. The size of 'NodeArray' is the number of nodes in the graph. The second array is the 'BucketTable' which will be a Structure of two elements. First element is the node's ID and the second element is an index into the 'NodeArray' array. The third array is the 'HashTable'. Each array element in the 'HashTable' is a pair of elements. Every node is Hashed into the HashTable by using some Hash() function. 'HashTable' will be a closed Hashing implementation of Nodes. Every entry in the HashTable is like a bucket of set of nodes. Each nodes in this bucket has the same Hash value. The first element 'a' in the 'HashTable' entry will be an index into the 'BucketTable', which tells that the bucket starts from index 'a' on BucketTable. The second element of the 'HashTable' will be the size of the bucket.

For searching the Node details of a node n, first Hash the value of Node ID and lets say the Hashed entry in 'HashTable' is (4, 3). This means that node n will be present between the index 4 and (4 + 3) of the 'BucketTable'. (i.e BucketTable[4 to 7] is a bucket where node n belongs to). Now search the BucketTable from 4 to 7 where you will find an entry in BucketTable as (c, d) where c == n. Here d becomes the index into the NodeArray where the node 'n' is stored. As soon as this is known, we get the details of a node. The representation is sparse in the sense that even if there are 100 nodes from 0 to 10000, instead of storing 10000 * sizeof(Node) we are storing:

100*sizeof(Node)+sizeof(HashTable) +
sizeof(BucketTable)*100

= 100 * sizeof(Node) + sizeof(HashTable) + 2 * 100
If Hash() is a map from 10000 => 100,
then, sizeof(HashTable) = 100 * sizeof(Node) + 100+200
= 100 * sizeof(Node) + 300;

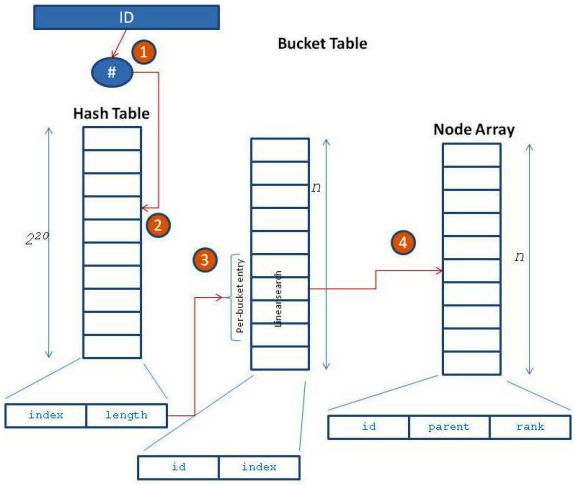

Figure 3. The hashing mechanism used in the implementation

Table I
TEST SETUP

Architecture	Features	Tools
x86	Intel(R) Xeon(R) CPU E5-	g++ 4.4.5, pthreads
	2670 0 @ 2.60GHz, 8 Cores	
GPU	Tesla M2070,	nvcc 5.5, cuda-
	Global Mem: 1.2 GB	memcheck, nvprof

which is very small when compared to the original representation of the nodes. The whole hash-map structure is illustrated in Figure 3.

3 EVALUATION AND RESULTS

The evaluation setup is described in Section 3.1, and the results along with graphs are given in Section 3.2. The analysis of results is done in Section 3.3.

3.1 Evaluation Setup

3.1.1 Test application: The application we have used is finding the connected components of a graph. The graphs used are described in Table 2. The correctness of the data structure is established by comparing the outputs of sequential and parallel versions of the application.

Figure 4 illustrates the flow of computation in the test application. The green colored blocks are executed in parallel, the maroon blocks are executed on x86, and blue ones on the GPU.

Table II
GRAPH LISTING

Stanford Graph_Name	Number of nodes	Number of edges
FB	4, 039	88, 234
soc-Epinions1	75, 879	508, 837
twitter	81, 306	1, 768, 149
soc-LiveJournals	4, 847, 571	68, 993, 773

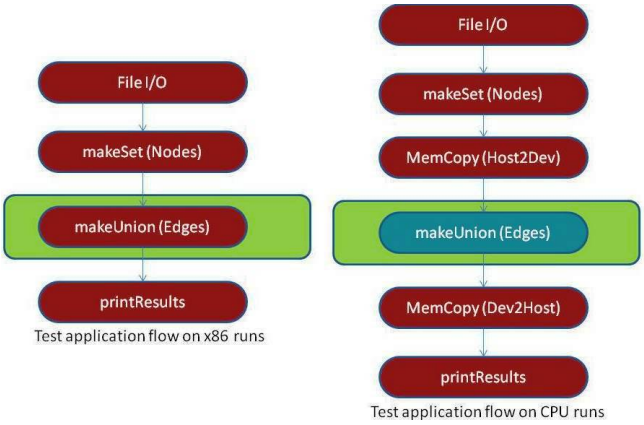

Figure 4. Flow of computation in the test application

3.2 Results

Table 4 shows the time measurements obtained in our experiments. We measure the time taken by makeUnion block for x86 experiments, and memCopy(Dev2Host), makeUnion and memCopy(DEv2Host) in case of GPU experiments.

3.3 Discussion

The newer two-level has table based implementation gives better performance than the traditional node-based implementation even in the sequential union-find data structure. This is because it utilizes the spatial locality better as compared to the traditional technique, in which memory for nodes is allocated on-the-fly. However, there is an exception to this fact in the LiveJournals graph.

As the actual job to be performed per thread in the union-find data structure is very simple, we observed that having a very large number of threads in the x86 implementation takes more time because of the bottleneck due to thread management over the computation that needs to be performed. For example, creating 8K threads takes more time than creating 4K threads (by varying the number of maximum jobs per thread) in the LiveJournals plot. The parallel implementation in x86 outweighs the sequential implementation when the graph is very large, thus leading to better performance due to parallelism over the overhead of thread management.

As can be seen from the results, the CUDA-based implementation takes much greater time than the x86 implementation. To understand this, we did profiling using the nvprof tool, and observed that the memCopy from device to host and vice-versa is actually not a big overhead and does not take much share of the required time. Thus, the other two reasons behind this may be either the atomic CAS operation or the divergence problem. To get a clearer picture, we compared the latency of atomic CAS and load-store operation, and found that they are approximately the same. To confirm the culprit as control divergence, we removed the concept of union-by-rank (and just used the node id's

Table III
IMPLEMENTATION DESCRIPTION

Implementation	Description
Sequential node-based	C++ based conventional union-find, where a structure represents a node. The makeSet operation adds this node to a std::map.
Sequential table-based	C++ based union-find data structure represented in the format described in Section 2.2.
Parallel x86	C++ based parallel version of the union-find data structure represented in the format described in Section 2.2.
Parallel CUDA	C based CUDA version of the union-find data structure represented in the format described in Section 2.2.

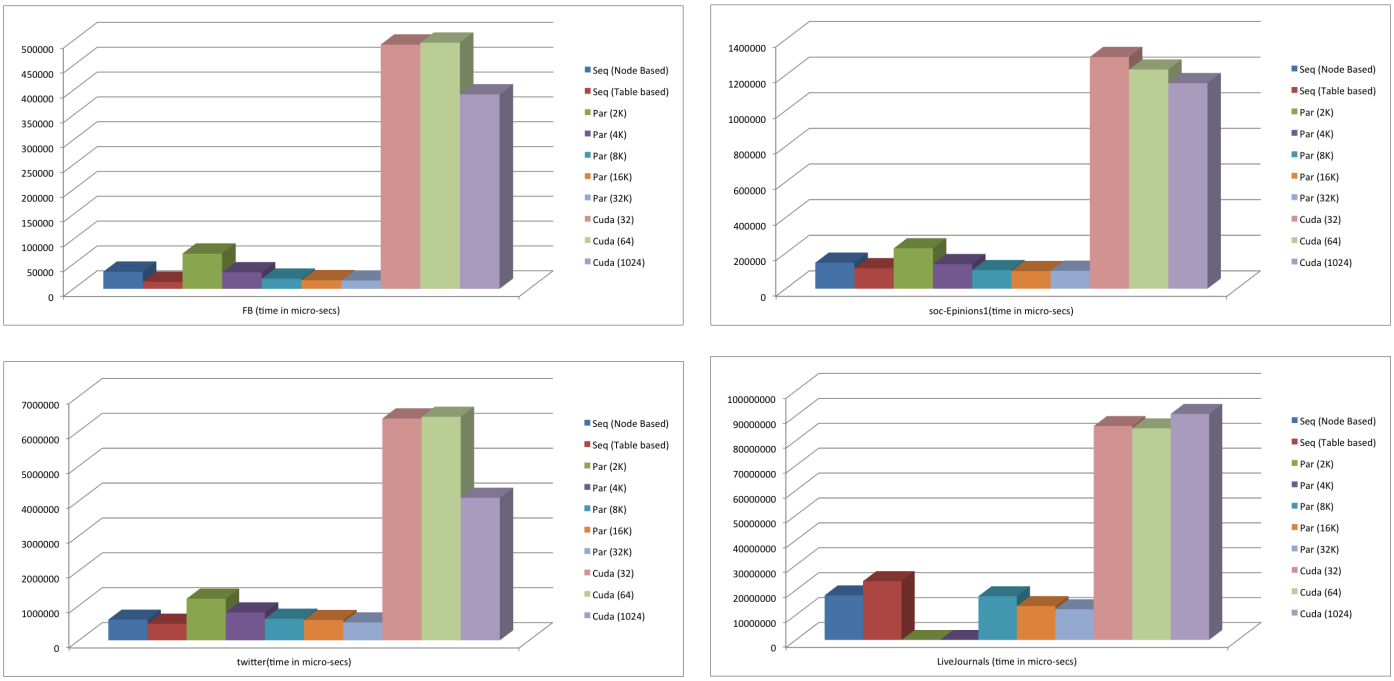

Figure 5. Plots for the time taken for various applications

instead), which apparently led to removing a couple of branches, and improved the performance three-folds. Hence the primary reason for CUDA taking more time is certified to be the divergence problem.

4. CONCLUSION AND FUTURE SCOPE

We implemented the union-find data structure in four different ways in this project. We devised a new way for efficiently managing the hash structure to get the node given its id, in constant time. The executions were tested on various graphs obtained by the Stanford Graph Library, and execution time was noted in each case. We observe and conclude that the parallel version on x86 beats the sequential one in most cases, but the CUDA version takes more time due to the problem of control divergence.

In future we plan to look towards reducing the time required due to control divergence. For this we are aiming at optimizing both the data representation and the operations. We are trying to construct a work queue in prior, and then create threads to perform the actual operations required by the application.

Table IV
TIME TAKEN BY DIFFERENT IMPLEMENTATIONS (IN MICROSECONDS). DNR STANDS FOR DID NOT RUN, BECAUSE THE NUMBER OF THREADS WAS REALLY HUGE. * MEANS JOBS PER THREAD.

Implementation	FB	soc-E	twitter	soc-LJ
Seq. (node)	34268	145459	591140	17899981
Seq. (table)	14108	114558	476047	23599170
Parallel (2K *)	70870	226404	1192637	DNR
Parallel (4K *)	33031	137635	799962	DNR
Parallel (8K *)	20352	104264	618007	17570802
Parallel (16K *)	17090	98724	583805	13594297
Parallel (32K *)	16553	100611	513364	12318298
CUDA (32 *)	491539	1300871	6360657	85884201
CUDA (64 *)	495520	1230461	6411095	84960618
CUDA (1024 *)	391221	1153976	4086961	90770348

5 REFERENCES

[1] Richard J. Anderson and Heather Woll, "Wait-Free Parallel Algorithms for the Union-Find Problem", *23rd ACM Symposium on Theory of Computing*

[2] Thomas Cormen et al., Data Structures for Disjoint Sets in *Introduction to Algorithms*, II Edition. Prentice Hall India, pp. 440-462

[3] Z Galil and GF Italiano, “Data Structures and algorithms for disjoint set union problems”, *ACM Computing Surveys (CSUR), 1991*

[4] R. E. Tarjan, “Efficiency of a good but not linear disjoint set union algorithm”, *Journal of the ACM, 22:215–225, 1975*

[5] Stanford Large Network Dataset Collection - SNAP. Internet: http://www.snap.stanford.edu/data/, *May 5, 2014*