A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers

Fredrik Manne and Md. Mostofa Ali Patwary*

Abstract

The Union-Find algorithm is used for maintaining a number of non-overlapping sets from a finite universe of elements. The algorithm has applications in a number of areas including the computation of spanning trees and in image processing.

Although the algorithm is inherently sequential there has been some previous efforts at constructing parallel implementations. These have mainly focused on shared memory computers. In this paper we present the first scalable parallel implementation of the Union-Find algorithm suitable for distributed memory computers. Our new parallel algorithm is based on an observation of how the Find part of the sequential algorithm can be executed more efficiently.

We show the efficiency of our implementation through a series of tests to compute spanning forests of very large graphs.

1 Introduction

The disjoint-set data structure is used for maintaining a number of non-overlapping sets consisting of elements from a finite universe. Its uses include among other, image decompositions, the computation of connected components and minimum spanning trees in graphs, and is also taught in most algorithm courses. The algorithm for implementing this data structure is often referred to as the Union-Find algorithm.

More formally, let U be a collection of n distinct elements and let S_i denote a set of elements from U. Two sets $\{S_1, S_2\}$ are disjoint if $S_1 \cap S_2 = \emptyset$. A disjoint set data structure maintains a collection $\{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets selected from U. Each set is identified by a representative x, which is usually some member of the set. The two main operations are then to *find* which set a given element belongs to by locating its representative element and also to create a new set from the *union* of two existing sets.

The underlying data structure of each set is typically a rooted tree where the element in the root vertex is the representative of the set. Using the two

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway, {Fredrik.Manne, Mostofa.Patwary}@ii.uib.no

techniques *Union-by-rank* and *path compression* the running time of any combination of m Union and Find operations is $O(n\alpha(m,n))$ where α is the very slowly growing inverse Ackerman function [4].

From a theoretical point of view using the Union-Find algorithm is close to optimal. However, for very large problem instances such as those that appear in scientific computing this might still be too slow or it might even be that the problem is too large to fit in the memory of one processor. One recent application that makes use of the Union-Find algorithm is a new algorithm for computing Hessian matrices using substitution methods [7]. Hence, designing parallel algorithms is necessary to keep up with the very large problem instances that appear in scientific computing.

The first such effort was by Cybenko et al. [5] who presented an algorithm using the Union-Find algorithm for computing the connected components of a graph and gave implementations both for shared memory and distributed memory computers. The distributed memory algorithm duplicates the vertex set and then partitions the edge set among the processors. Each processor then computes a spanning forest using its local edges. In $\log p$ steps, where p is the number of processors, these forests are then merged until one processor has the complete solution. However, the experimental results from this algorithm were not promising and showed that for a fixed size problem the running time increased with the number of processors used.

Anderson and Woll also presented a parallel Union-Find algorithm using *wait-free* objects suitable for shared memory computers [1]. They also showed that parallel algorithms using concurrent Union-operations risk creating unbalanced trees. However, they did not produce any experimental results for their algorithm.

We note that there exists an extensive literature on designing parallel algorithms for computing a spanning forest or the connected components of a graph. However, up until the recent paper by Bader and Cong [3] such efforts had failed to give speedup on arbitrary graphs. In [3] the authors present a novel scalable parallel algorithm for computing spanning forests on a shared memory computer.

Focusing on distributed memory computers is of importance since these have better scalability than shared memory computers and thus the largest systems tend to be of this type. However, their higher latency makes distributed memory computers more dependent on aggregating sequential work through the exploitation of locality.

The current work presents a new parallel Union-Find algorithm for distributed memory computers. The algorithm operates in two stages. In the first stage each processor performs local computations in order to reduce the number of edges that need to be considered for inclusion in the final spanning tree. This is similar to the approach used in [5], however, we use a sequential Union-Find algorithm for this stage instead of BFS. Thus when we start the second parallel stage each processor has a Union-Find type forest structure that spans each local component.

In the second stage we merge these structures across processors to obtain a global solution. In both the sequential and the parallel stage we make use of a novel observation on how the Union-Find algorithm can be implemented. This allows both for a faster sequential algorithm and also to reduce the amount of communication in the second stage.

To show the feasibility and efficiency of our algorithm we have implemented several variations of it on a parallel computer using C++ and MPI and performed tests to compute spanning trees of very large graphs using up to 40 processors. Our results show that the algorithm scales well both for real world graphs and also for small-world graphs.

The rest of the paper is organized as follows. In Section 2 we briefly explain the sequential algorithm and also how this can be optimized. In Section 3 we describe our new parallel algorithm, before giving experimental results in Section 4.

2 The Sequential Algorithm

In the following we first outline the standard sequential Union-Find algorithm. We then point out how it is possible to speed up the algorithm by paying attention to the rank values. This is something that we will make use of when designing our parallel algorithm.

The standard data structure for implementing the Union-Find algorithm is a forest where each tree represents a connected set. To implement the forest each element x has a pointer $p(x)$ initially set to x. Thus each x starts as a set by itself. The two operations used on the sets are then $Find(x)$ and $Union(x, y)$ where x and y are distinct elements. $Find(x)$ returns the root of the tree that x belongs to. This is done by following pointers starting from x. $Union(x, y)$ merges the two trees that x and y belong to. This is achieved by making one of the roots of x and y point to the other. With these operations the connected components of a graph $G(V, E)$ can be computed as shown in Algorithm 1.

Algorithm 1 The sequential Union-Find algorithm

```
$S = \emptyset$
for each $x \in V$ do
    $p(x) = x$
for each $(x, y) \in E$ do
    if $Find(x) \neq Find(y)$ then
        $Union(x, y)$
        $S = S \cup \{(x, y)\}$
```

When the algorithm terminates the vertices of each tree will consist of a connected component and the set of edges in S define a spanning forest on G.

There are two standard techniques for speeding up the Union-Find algorithm. The first is Union-by-rank. Here each vertex is initially given a rank of 0. If two sets are to be merged where the root elements are of equal rank then the rank of the root element of the combined set will be increased by one. In all other Union operations the root with the lowest rank will be set to point to the root with the higher rank while all ranks remain unchanged. Note that this ensures that the parent of a vertex x will always have higher rank than the vertex x itself.

The second technique is path compression. In its simplest form, following any Find operation, all traversed vertices will be set to point to the root. This has the effect of compressing the path and making subsequent Find operations using any of these vertices faster. Note that even when using path compression the rank values will still be strictly increasing when moving upwards in a tree.

Using the techniques of Union-by-rank and path compression the running time of any combination of m Union and Find operations is $O(n\alpha(m,n))$ where α is the very slowly growing inverse Ackerman function [4].

We now consider how it is possible to implement the Union-Find algorithm in a more efficient way. It is straight forward to see that one can speed up Algorithm 1 by storing the results of the two Find operations and use these as input to the ensuing Union operation which then only has to determine which of the two root vertices should point to the other.

Our observation is that in certain cases one can use the rank values to terminate the Find operation before reaching the root. Let the rank of a vertex z be denoted by $rank(z)$. Consider two vertices x and y belonging to different sets with roots r_x and r_y respectively where $rank(r_x) < rank(r_y)$. If we find r_x before r_y then it is possible to terminate the search for r_y as soon as we reach an ancestor z of y where $rank(z) = rank(r_x)$. This follows since the rank function is strictly increasing and we must therefor have $rank(r_y) > rank(r_x)$ implying that $r_y \neq r_x$. At this point it is possible to join the two sets by setting $p(r_x) = p(z)$. Note that this will neither violate the rank property nor will it increase the asymptotic time bound of the algorithm. However, if we perform $Find(y)$ before $Find(x)$ we will not be able to terminate early. We therefore suggest that instead of doing the two Find operations separately, that one instead performs them in an interleaved fashion by always continuing the search from the vertex with the lowest current rank. In this way the Find operation can terminate as soon as one reaches the root with the smallest rank. We label this as the *zigzag* Find operation as opposed to the *classical* Find operation.

The zigzag Find operation can also be used to terminate the search early when the vertices x and y belong to the same set. Let z be their lowest common ancestor. Then at some stage of the zigzag Find operation the current ancestors of x and y will both be equal to z. At this point it is clear that x and y belong to the same set and the search can stop.

3 The Parallel Algorithm

In the following we outline our new parallel Union-Find algorithm. We assume a partitioning of both the vertices and the edges of G into p sets each, $V = \{V_0, V_1, \ldots, V_{p-1}\}$ and $E = \{E_0, E_1, \ldots, E_{p-1}\}$ with the pair (V_i, E_i) being allocated to processor i, $0 \leq i < p$. If $v \in V_i$ (or $e \in E_i$) processor i *owns* v (or e) and v (or e) is *local* to processor i.

Any processor i that has a local edge $(v, w) \in E_i$ such that it does not own vertex v will create a *ghost vertex* v' as a substitution for v. We denote the set of ghost vertices of processor i by V_i'. Thus an edge allocated to processor i can either be between two vertices in V_i, between a vertex in V_i and a vertex in V_i', or between two vertices in V_i'. We denote the set of edges adjacent to at least one ghost vertex by E_i'.

The algorithm operates in two stages. In the first stage each processor performs local computations without any communication in order to reduce the number of edges that need to be considered for the second final parallel stage. Due to space considerations we only outline the steps of the algorithm and neither give pseudo-code nor a formal proof that the algorithm is correct.

Stage 1. Reducing the input size

Initially in Stage 1 each processor i computes a spanning forest T_i for its local vertices V_i using the local edges $E_i - E_i'$. This is done using a sequential Find-Union algorithm. It is then clear that T_i can be extended to a global spanning forest for G.

Next, we compute a subset T_i' of E_i' such that $T_i \cup T_i'$ form a spanning forest for $V_i \cup V_i'$. Due to space considerations we omit the details of how T_i' can be computed efficiently but note that this can be done without destroying the structure of T_i. The remaining problem is now to select a subset of the edges in T_i' so as to compute a global spanning forest for G.

Stage 2. Calculating the final spanning forest

The underlying data structure for this part of the algorithm is the same as for the sequential Union-Find algorithm, only that we now allow trees to span across several processors. Thus a vertex v can set $p(v)$ to point to a vertex on a different processors other than its own. The pointer $p(v)$ will in this case contain information about which processor owns the vertex being pointed to, its local index on that processor, and also have a lower bound on its rank. Each ghost vertex v' will initially set $rank(v') = 0$ and $p(v') = v$. Thus the connectivity of v' is initially handled through the processor that owns v. For the local vertices the initial $p()$ values are as given from the computation of T_i.

We define the *local* root $l(v)$ as the last vertex on the Find-path of v that is stored on the same processor as v. If in addition $l(v)$ has $p(l(v)) = l(v)$ then $l(v)$ is also a *global* root.

In the second stage of the algorithm processor i iterates through each edge $(v, w) \in T_i'$ to determine if this edge should be part of the final spanning forest or not. This is done by issuing a Find-Union query (FU) for each edge. A FU-query can either be resolved internally by the processor or it might have to be sent to other processors before an answer is returned. To avoid a large number of small messages a processor will process several of its edges before sending and receiving queries. A computation phase will then consist of first generating new FU-queries for a predefined number of edges in T_i' and then to handle incoming queries. Any new messages to be sent will be put in a queue and transmitted in the ensuing communication phase. Note that a processor might have to continue processing incoming queries after it has finished processing all edges in T'_i.

In the following we describe how the FU-queries are handled. A FU-query contains information about the edge (v, w) in question and also to which processor it belongs. In addition the FU-query contains two vertices a and b such that a and b are on the Find-paths of v and w respectively. The query also contains information about the rank of a and b and if either a or b is a global root. Initially $a = v$ and $b = w$.

When a processor receives (or initiates) a FU-query it is always the case that it owns at least one of a and b. Assume that this is a, we then label a as the *current* vertex. Then a is first replaced by $p(l(a))$. There are now three different ways to determine if (v, w) should be part of the spanning forest or not: *i)* If $a = b$ then v and w have a common ancestor and the edge should be discarded. *ii)* If $a \neq b$, $p(a) = a$, and $rank(a) < rank(b)$ then $p(a)$ can be set to b and thus including (v, w) in the spanning forest. *iii)* If $a \neq b$, $rank(a) = rank(b)$, $p(a) = a$, while b is marked as also being a global root then $p(a)$ can be set to b while a message is sent to b to increase its rank by one.

To avoid that a and b concurrently sets each other as parents in Case *iii)* we associate a unique random number $r()$ with each vertex. Thus we must also have $r(a) < r(b)$ before we set $p(a) = b$.

If a processor i reaches a decision on the current edge (v, w), it will send a message to the owner of the edge about the outcome. Otherwise processor i will forward the updated FU-query to a processor j (where $j \neq i$) such that j owns at least one of a and b.

In the following we outline two different ways in which the FU-queries can be handled. The difference lies mainly in the associated communication pattern and reflects the classical as opposed to the zigzag Union-Find operation as outlined in Section 2.

In the classical parallel Union-Find algorithm a is initially set as the current vertex. Then while $a \neq p(a)$ the query is forwarded to $p(a)$. When the query reaches a global root, in this case a, then if b is marked as also being a global root, rules *i)* through *iii)* are applied. If these result in a decision such that the edge is either discarded or $p(a)$ is set to b then the query is terminated and a message is sent back to the processor owning the edge in question. Otherwise, the query is forwarded to b where the process is repeated (but now with b as the current vertex).

In the parallel zigzag algorithm a processor that initiates or receives a FU-query will always check all three cases after first updating the current vertex z with $l(z)$. If none of these apply the query is forwarded to the processor j which owns the one of a and b marked with the lowest rank and if $rank(a) = rank(b)$ the one with lowest r value. Note that if v and w are initially in the same set then a query will always be answered as soon as it reaches the processor that owns the lowest common ancestor of v and w. Similarly, if v and w are in different sets the query will be answered as soon as the query reaches the global root with lowest rank.

Since FU-queries are handled concurrently it is conceivable that a vertex $z \in \{a, b\}$ has seized to be a global root when it receives a message to increase its rank (if Case *iii)* has been applied). To ensure the monotonicity of ranks z then checks, starting with $w = p(z)$, that $rank(w)$ is strictly greater than the updated rank of z. If not we increase $rank(w)$ by one and repeat this for $p(w)$. Note that this process can lead to extra communication.

Similarly as for the algorithm in [1] it is possible that unbalanced trees are created with both parallel communication schemes. This can happen if trees with the same rank are merged concurrently such that one hangs of the other.

When a processor i receives a message that one of its edges (v, w) is to be part of the spanning forest it is possible to initiate a path compression operation between processors. On processor i this would entail to set $l(v)$ (and $l(w)$) to point to the new root which would then also have to be included in the return message. Since there could be several such incoming messages for $l(v)$ and these could arrive in an arbitrary order we must first check that the rank of the new root is larger than the rank that i has stored for $p(l(v))$ before performing the compression. If this is the case then it is possible to continue the compression by sending a message to $p(l(v))$ about the new root. We label these schemes as either 1-level or full path compression.

4 Experiments

For our experiments we have used a Cray XT4 distributed memory parallel machine with AMD Opteron quad-core 2.3 GHz processors where each group of four cores share 4 GB of memory. The algorithms have been implemented in C++ using the MPI message-passing library. We have performed experiments using both graphs taken from real application as well as on different types of synthetic graphs. In particular we have used application graphs from areas such as linear programming, medical science, structural engineering, civil engineering, and automotive industry [6, 8]. We have also used small-world graphs as well as random graphs generated by the GTGraph package [2].

Table 1 give properties of the graphs. The first nine rows contains information about the application graphs while the final two rows give information about the small-world graphs. The first 5 columns gives structural properties about the graphs while the last two columns show the time in seconds for computing a spanning forest using Depth First Search (DFS) and the sequential zigzag algorithm (ZZ). We have also used two random graphs both containing one million vertices and respectively, 50 and 100 million edges. Note that all of these graphs only contains one component. Thus the spanning forest will always be a tree.

Our first results concern the different sequential algorithms for computing a spanning forest. As is evident from Table 1 the zigzag algorithm outperformed the DFS algorithm. A comparison of the different sequential Union-Find algorithms on the real world graphs is shown in the upper left quadrant of Figure 1. All timings have been normalized relative to the slowest algorithm, the classical algorithm (CL) using path compression (W). As can be seen, removing the path compression (O) decreases the running time. Also, switching to the

Name	$\lvert V\rvert$	$\lvert E\rvert$	Max Deg	Avg Deg	DFS	ZZ
m_t1	97578	4827996	236	98.95	0.12	0.06
cranksg2	63838	7042510	3422	220.64	0.15	0.03
inline_1	503712	18156315	842	72.09	0.57	0.26
ldoor	952203	22785136	76	47.86	0.71	0.47
af_shell10	1508065	25582130	34	33.93	1.04	0.37
boneS10	914898	27276762	80	59.63	0.86	0.38
bone010	986703	35339811	80	71.63	1.05	0.47
audi	943695	38354076	344	81.28	1.20	0.33
spal_004	321696	45429789	6140	282.44	1.33	0.66
rmat1	377823	30696982	8109	162.49	2.07	1.34
rmat2	504817	40870608	10468	161.92	2.71	1.81

Table 1: Properties of the application graphs

zigzag algorithm (ZZ) improves the running time further, giving approximately a 50% decrease in the running time compared to the classical algorithm with path compression. To help explain these results we have tabulated the number of "parent chasing" operations on the form $z = p(z)$. These show that the zigzag algorithm only executes about 10% as many such operations as the classical algorithm. However, this does not translate to an equivalent speed up due to the added complexity of the zigzag algorithm.

The performance results for the synthetic graphs give an even more pronounced improvement when using the zigzag algorithms. For these graphs both zigzag algorithms outperforms both classical algorithms and the zigzag algorithm without path compression gives an improvement in running time of close to 60% compared to the classical algorithm with path compression.

Next, we present the results for the parallel algorithms. For these experiments we have used the Mondrian hypergraph partitioning tool [9] for partitioning vertices and edges to processors. For most graphs this has the effect of increasing locality and thus enabling to reduce the size of T'_i in Stage 1. In our experiments $T' = \cup_i T'_i$ contained between 0.1% and 0.5 % of the total number of edges for the application graphs, between 1 % and 6 % for the small-world graphs, and between 2 % and 36 % for the random graphs. As one would expect these numbers increase with the number of processors.

In our experiments we have compared using either the classical or the zigzag algorithm, both for the sequential computation in Stage 1 and also for the parallel computation in Stage 2. We note that in all experiments we have only used level-1 path compression in the parallel algorithms as using full compression, without exception, slowed down the algorithms.

How the improvements from the sequential zigzag algorithm are carried into the parallel algorithm can be seen in the upper right and lower left quadrant of Figure 1. Here we show the result of combining different parallel algorithms with different sequential ones when using 4 and 8 processors. All timings have again been normalized to the slowest algorithm, the parallel classical algorithm (P-

CL) with the sequential classical algorithm (S-CL), and using path compression (W). Replacing the parallel classical algorithm with the parallel zigzag algorithm while keeping the sequential algorithm fixed gives an improvement of about 5% when using 4 processors. This increases to 14% when using 8 processors, and to about 30% when using 40 processors. This reflects how the running time of Stage 2 of the algorithms becomes more important for the total running time as the number of processors are increased.

The total number of sent and forwarded FU-queries is reduced by between 50% and 60% when switching from the parallel classical to the parallel zigzag algorithm. Thus this gives an upper limit on the possible gain that one can obtain from the parallel zigzag algorithm over the parallel classical algorithm.

When keeping the parallel zigzag algorithm fixed and replacing the sequential algorithm in Step 1 we get a similar effect as we did when comparing the sequential algorithms, although this effect is dampened as the number of processors is increased and Step 1 takes less of the overall running time.

The figure in the lower right corner shows the speedup on three large matrices when using the best combination of algorithms, the sequential and parallel zigzag algorithm. As can be seen the algorithm scales well up to 32 processors at which point the communication in Stage 2 dominates the algorithm and causes a slowdown. Similar experiments for the small-world graphs showed a more moderate speedup peaking at about a factor of four when using 16 processors. The random graphs did not obtain speedup beyond 8 processors and even for this configuration the running time was still slightly slower than for the best sequential algorithm.

To conclude we note that the zigzag Union-Find algorithm achieves considerable savings compared to the classical algorithm both for the sequential and the parallel case. However, our parallel implementation did not achieve speedup for the random graphs, as was the case for the shared memory implementation in [3]. This is mainly due to the poor locality of such graphs.

References

[1] R. J. Anderson and H. Woll, *Wait-free parallel algorithms for the union-find problem*, in Proceedings of the twenty-third annual ACM symposium on Theory of computing (STOC 91), 1991, pp. 370–380.

[2] D. A. Bader and K. Madduri, *GTGraph: A synthetic graph generator suite.* http://www.cc.gatech.edu/~kamesh/GTgraph, 2006.

[3] D. J. Bader and G. Cong, *A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps)*, Journal of Parallel and Distributed Computing, 65 (2005), pp. 994–1006.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, second ed., 2001.

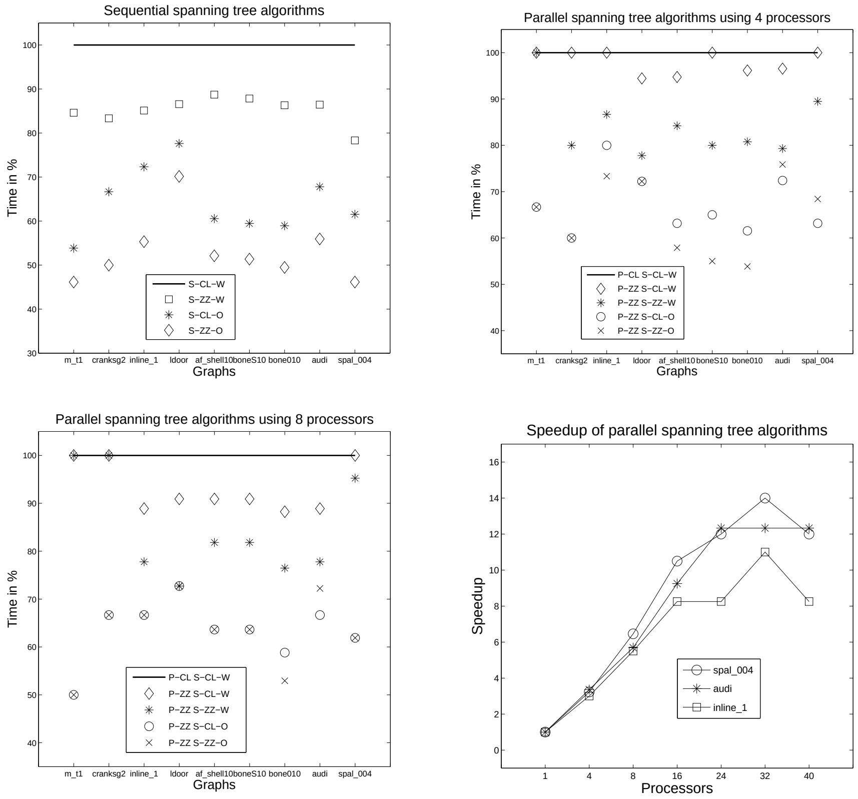

Figure 1: Performance results: S - Sequential algorithm, P- Parallel algorithm, CL - Classical Union-Find, ZZ - zigzag Union-Find, W - With path compression, O - Without path compression.

[5] G. Cybenko, T. G. Allen, and J. E. Polito, *Practical parallel algorithms for transitive closure and clustering*, International Journal of Parallel Computing, 17 (1988), pp. 403–423.

[6] T. A. Davis, *University of Florida sparse matrix collection.* Submitted to ACM Transactions on Mathematical Software.

[7] A. H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen, *New acyclic and star coloring algorithms with applications to computing hessians*, SIAM Journal on Scientific Computing, 29 (2007), pp. 515–535.

[8] J. Koster, *Parasol matrices.* http://www.parallab.uib.no/projects/parasol/data.

[9] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.