# A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers

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

*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, in image processing, as well as in scientific computations.**

**Although the algorithm is inherently sequential there has been some previous efforts at constructing parallel implementations. These have mainly focused on shared memory computers. Here 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.**

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 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 $\alpha$ 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.

Early efforts at designing parallel Union-Find algorithms [5], [1] did not result in codes that gave speedup. In [3], Bader and Cong presented the first parallel algorithm for computing spanning forests that gave speedup on arbitrary graphs. However, this code did not employ the Union-Find structure and is only applicable for shared memory computers.

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 Stage 1 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 Stage 2 we merge these structures across processors to obtain a global solution. This is again carried out using a Union-Find type algorithm resulting in a data structure where components can span several processors. Thus a Find operation can now move between processors and result in parent pointers being set to off-processor nodes. Also, since multiple edges might be considered simultaneously, a Union operation might not always succeed as this could lead to cycles in the data-structure.

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 Stage 2. When determining if an edge $(v, w)$ should be part of the spanning tree in the classical sequential implementation, one follows parent pointers, first from $v$ and then from $w$. If the two root elements are different then the edge is added to the solution and one of the roots is set to point to the other. If the two root elements are identical the edge is discarded.

We show that by carefully traversing the data structure in a *zigzag* fashion, one can stop the search earlier. If the edge should not be part of the solution we stop the search as soon as we reach the lowest common ancestor of $v$ and $w$ and if the edge should be part of the solution we never search further than to the lowest root before merging the trees.

To show the feasibility and efficiency of our algorithm we have implemented several variations of it on a Cray XT4 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. 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].

Our first results concern the different sequential algorithms for computing a spanning forest. A comparison of the different sequential Union-Find algorithms on the real world graphs is shown in the upper left quadrant of Fig. 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 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 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 assigning vertices and edges to processors. For most graphs this has the effect of increasing locality and thus enabling to reduce the number of edges that have to be considered for Stage 2.

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. 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 Fig. 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 becomes more important for the total running time as the number of processors is increased.

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 lower right quadrant of Fig. 1 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.

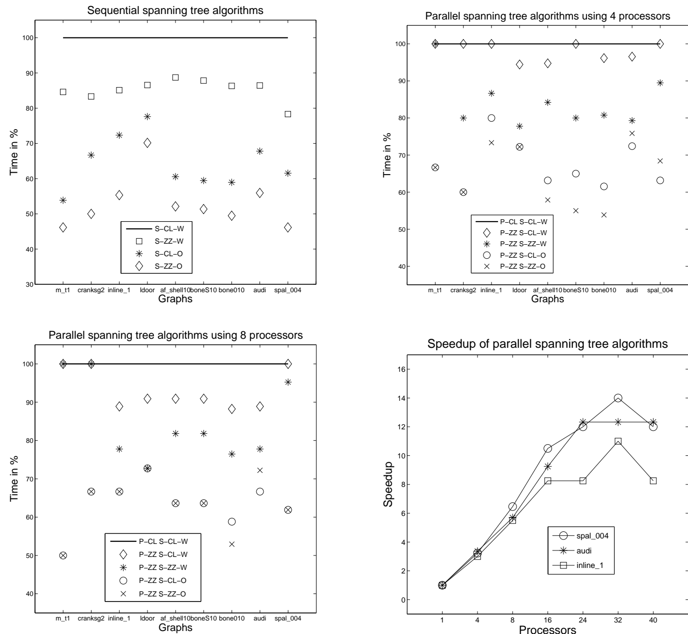


Fig. 1. Performance results: S - Sequential algorithm, P- Parallel algorithm, CL - Classical Union-Find, ZZ - zigzag Union-Find, W - With path compression, O - Without path compression.

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 beyond 8 processors and even for this configuration the running time was still slightly slower than for the best sequential algorithm. 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.

[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.