2012 IEEE 26th International Parallel and Distributed Processing Symposium

Multi-core spanning forest algorithms using the disjoint-set data structure

Md. Mostofa Ali Patwary* Peder Refsnes** Fredrik Manne**

***Abstract*—We present new multi-core algorithms for computing spanning forests and connected components of large sparse graphs. The algorithms are based on the use of the disjoint-set data structure. When compared with the previous best algorithms for these problems our algorithms are appealing for several reasons: Extensive experiments using up to 40 threads on several different types of graphs show that they scale better. Also, the new algorithms do not make use of any hardware specific routines, and thus are highly portable. Finally, the algorithms are quite simple and easy to implement.**

***Keywords*-Multi-core, spanning forrest, connected components, Disjoint sets**

I. Introduction

We consider the problem of computing a spanning tree for an undirected graph $G(V, E)$. This is a fundamental problem in computer science as it is a building block for many graph algorithms. The spanning tree problem falls into the category of graph problems where there is very little work compared to the amount of data. Lately, these types of problems have received considerable interest as one wants to explore the structure of the very large graphs that are becoming available from sources such as web-graphs and various social network graphs. One example of the interest in these types of problems is the recent introduction of the Graph 500 initiative [1],[2]. This classifies the performance of parallel computers based on their performance when solving specific graph problems where the ratio between computation and data is low.

In sequential computation a spanning tree is easily found using a graph traversal method such as depth-first-search or breadth-first-search. Both of these methods will traverse each component of G in a connected manner. It is also possible to compute a spanning tree by maintaining partial non-overlapping trees and then adding edges to the solution as long as they do not introduce a cycle. From a parallel point of view it is advantageous to be able to work concurrently on unconnected parts of the graph. Thus many of the algorithms with best asymptotic running time on the PRAM model employs techniques of this type.

However, in terms of developing a practical algorithm these strategies has so far not been successful. In fact, for shared memory computers the only algorithm that has been shown to give speedup when applied to arbitrary graphs is based on performing a connected parallel depth first search [3].

*EECS Department, Northwestern University, Evanston, IL 60208, USA, m-patwary@northwestern.edu

**Department of Informatics, University of Bergen, N-5020 Bergen, Norway, fredrikm@ii.uib.no, Peder.Refsnes@gmail.com

In this paper we present two new parallel algorithms based on the Union-Find algorithm. The Union-Find algorithm maintains a collection of disjoint sets where each set represents connected vertices. The classical sequential approach to implementing Union-Find algorithms is to use the two techniques known as Union-by-rank and path compression obtaining a running time of $O(m\alpha(m, n))$ for computing a spanning tree on a graph containing n vertices and m edges [4], where α is the very slow growing inverse Ackermann function. Our algorithms on the other hand are based on a different scheme known as Rem's algorithm [5]. In an extensive study we recently showed that even though the asymptotic running time of this algorithm is suboptimal, in practice it is still substantially faster than other implementations [6].

Compared to the algorithm in [3] our solution offers the following advantages: 1) It computes a spanning forest and not just one spanning tree. 2) It scales better with increasing number of processors. 3) The edges of the graph can be processed in any order and therefore does not rely on the input being given in terms of edge lists for each vertex. 4) The algorithm is simpler to implement

We also note that as our algorithm is implemented in OpenMP it is therefore highly portable and does not rely on any low level hardware specific primitives.

The rest of the paper is organized as follows. In Section II we summarize previous work on developing parallel algorithms for this problem. Then in Section III we explain how Union-Find algorithms work and in particular Rem's algorithm. Our new parallel algorithms are given in Section IV and Section V contains experiments using these. Finally, we conclude in Section VI.

II. Previous Work

There exists a large number of PRAM algorithms for computing connected components and spanning trees, for a survey see the paper by Bader and Cong [3]. This paper also lists several previous efforts at designing practical parallel algorithms for these problems and concludes that none of these show speedup on arbitrary graphs, although some show modest speedup on particular graphs. The paper then goes on to present a new parallel algorithm suitable for shared memory computers for computing the spanning tree of a connected graph. This algorithm is based on a parallel depth first search, where one processor first sequentially computes a stub spanning tree on a small part of the graph. Following this, the processors will start from different nodes of the stub tree and perform a depth first traversal until the entire graph

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.79

has been covered. To ensure load balance a processor that runs out of vertices can steal work from other processors. Through a number of experiments they show that the algorithm scales well on different types of graphs. They also compare their algorithm with an implementation of the Shiloach-Vishkin algorithm [7] for computing connected components. However, in most cases this algorithm did not give better running time than the sequential algorithm. All algorithms were implemented using POSIX threads and software-based barriers.

We note that the Shiloach-Vishkin algorithm is based on the general principle of maintaining a (dynamic) set of disjoint connected components each represented by a rooted tree using parent pointers. Initially each vertex is a component by itself. Then during the processing of the edges, if an edge is found to connect two components, one component is *grafted* (i.e. merged) with the other by setting the parent pointer of the root node to point to some vertex in the other component. In addition, the algorithm performs *pointer jumping* steps where a parent pointer is set to point higher up in its own tree, in this way shortening the distance from the vertex to the root.

Although not referenced in [3], we note that prior to this publication Cybenko et al. [8] presented different types of spanning tree algorithms based on parallelizing a UNION-FIND algorithm. A UNION-FIND algorithm performs the same type of operations as the Shiloach-Vishkin algorithm, but a grafting operation is now called a UNION operation and pointer jumping is referred to as a *compression* operation. We outline the details in Section III. For shared memory computers they present an algorithm based on using a critical section for the UNION operation. This ensures that there is never any contention between the processors as to which one updates a particular parent pointer. The algorithm works by partitioning the edges among the processors who then work concurrently on the same global directed forest data structure. Only when a processor needs to perform a UNION operation will it enter into the critical section to do so. In [8] it is also shown that different types of compression operations can be performed concurrently without using a critical section or locks. The algorithm was implemented using the two techniques of Union-by-size and path compression. Experiments showed that it scaled well when applied to random graphs. It is mentioned that one can use one lock for every node instead of a critical section, although this approach was not implemented.

There are some differences between the algorithm by Bader and Cong and the shared memory algorithm by Cybenko et al. that are worth noting. As implemented, the Bader-Cong algorithm requires a neighbor list representation where each edge is represented twice (i.e. in both directions). Also, this algorithm will only compute a spanning tree of a *connected* graph. The algorithm by Cybenko et al. on the other hand, can compute a spanning forest of a disconnected graph and only requires a (possibly unordered) list of the edges.

Cybenko et al. also present a distributed memory algorithm where the edges are partitioned 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 due to communication costs and showed that for a fixed size problem the running time increased with the number of processors used.

We note that Anderson and Woll presented a parallel UNION-FIND algorithm using wait-free objects suitable for shared memory computers [9]. In doing so they showed that parallel algorithms using concurrent UNION operations risk creating unbalanced trees. However, they did not produce any experimental results for their algorithm. This algorithm relies on the low level operation of *Compare&Swap* to achieve correctness. The Compare&Swap operation lets one processor atomically compare the value in one memory location and depending on the outcome the value can be swapped with another value. Although some processors offers hardware support for this operation these are in general not portable between different architectures. It is also not included in parallel libraries such as PTHREADS or OpenMP.

Finally, we point out that we have proposed and implemented a UNION-FIND algorithm suitable for distributed memory parallel computers [10]. This algorithm maintains distributed disjoint sets. Initially it partitions the vertices of the graph among the processors with each edge being stored with one of its endpoints. Similarly to [8], each processor first computes a local spanning tree on its assigned vertices. The edges that span vertices allocated to different processors are then processed in a final stage to compute the complete solution. This stage employs a parallel algorithm similar to Rem's algorithm. The owner of each edge issues a single query that will travel upwards on the respective trees of the two initial vertices. The query will terminate once it can determine if the two trees are already connected. If this is not the case, the query will perform the merge operation itself. Experiments showed that this algorithm scaled well although it should be noted that this was dependent on the graph being partitioned across the processors as to preserve locality.

III. UNION-FIND ALGORITHMS

In the following we explain in more detail how UNION-FIND algorithms are used for computing connected components of a simple undirected graph $G(V, E)$. We also explain different ways in which such algorithms can be implemented. Common to all of these algorithms is that they operate on a collection of disjoint sets containing all vertices of G as elements. Each set consists of a rooted tree using a parent function $p()$. The root element of each set is referred to as the representative element of the set. The basic outline of a UNION-FIND algorithm for computing a spanning forest is shown in Algorithm 1.

Here the operation MAKESET(x) creates a set containing only the vertex x, while the FIND(x) operation returns the representative element of the set that vertex x belongs to. Finally, the UNION(x, y) operation merges the two sets containing vertices x and y. The resulting set of edges S is a minimal subset of E such that S is a spanning forest of G. Within S, vertices in the same connected component of G will

Algorithm 1 The UNION-FIND algorithm

```
1: S ← ∅
2: for each x ∈ V do
3:    MAKESET(x)
4: for each (x, y) ∈ E do
5:    if FIND(x) ≠ FIND(y) then
6:       UNION(x, y)
7:       S ← S ∪ {(x, y)}
```

belong to the same set and will have the same representative element.

There exists several ways of enhancing Algorithm 1 so that it runs faster. The most well known of these is to use the two techniques known as Union-by-rank and path compression. Union-by-rank specifies that when merging two sets one should set the pointer of the root element of the "smaller" tree to point to the root of the "larger" tree. Size is here measured by a *rank* value associated with the root element. This value starts out as zero and is only increased by one when two sets of equal rank are merged. In path compression the vertices traversed during a FIND operation (i.e. the FIND path) are traversed a second time after the root has been found, setting all parent pointers to point directly to the root. The running time of Algorithm 1 using both of these techniques is $O(m\alpha(m, n))$ for any combination of m MAKESET, UNION, and FIND operations on n elements [4]. Here α is the very slowly growing inverse Ackermann function.

Other ways of implementing Algorithm 1 include the UNION technique Union-by-size where the tree containing the fewest vertices is set to point to the root of the other tree. Instead of path compression one can use *path-splitting* where every node on the FIND-path is set to point to its grandparent, the effect being that the FIND-path is split into two disjoint paths, both hanging off the root. Finally, in *path-halving* this process of pointing to a grandparent is only applied to every other node on the FIND-path.

Combining either Union-by-size or Union-by-rank with one of the three mentioned compression techniques will still maintain the running time of $O(m\alpha(m, n))$.

In a recent experimental study we compared all suggested ways of implementing Algorithm 1 (a total of 63 different algorithms) to find which was the most efficient. The conclusion from this was that a somewhat forgotten algorithm known as Rem's algorithm [5] with a sub-optimal worst case running time of $O(m \log_{2+m/n} n)$ [4] was the best. On a large collection of test graphs the improvement given by Rem's algorithm when compared to using Union-by-rank and path compression ranged from 38% to 63% with an average of 49%.

In Rem's algorithm the UNION operation is done by setting a lower numbered node to point to a higher numbered one. Here the number used is typically the associated index (numbered 1 through n) of the vertex. We denote this way of performing the UNION operation as Union-by-index. Also, the two FIND operations are executed in an interleaved fashion instead of done separately. Thus when two pointers are moving upwards along their respective FIND-path, only the one pointing to the node with the lowered number parent will be moved. In this way, if the two nodes are already in the same tree, it is possible to stop the search as soon as the two pointers both reach their lowest common ancestor z. Compared to the traditional FIND operation this save two traversals from z to the root element.

In the case where the two nodes are not in the same tree, Rem's algorithm can also save some processing by performing the UNION operation before the pointers r_x and r_y reach their respective roots. This can happen if r_x is a root and $r_x < p(r_y)$. Then the two sets must be disjoint and the algorithm will set $p(r_x) = p(r_y)$ thus merging the sets. Note that this does not violate the increasing parent pointer property.

Rem's algorithm was originally presented using a compression technique known as *splicing*. In the case when r_x is to be moved to $p(r_x)$, let $z = p(r_x)$, then the value of $p(r_x)$ is set to $p(r_y)$ before r_x is set to point to z. Thus, following the operation the subtree originally pointed to by r_x is now a sibling of r_y. This neither compromises the increasing parent property (because $p(r_x) < p(r_y)$) nor invalidates the set structures (because the two sets will have been merged when the operation ends.) This also takes care of a possible final UNION operation once r_x (or r_y) reaches the root of its subtree. The effect of splicing is that each new parent has a higher value than the value of the old parent, thus hopefully compressing the tree. The algorithm for processing one edge is given as Algorithm 2.

Algorithm 2 Rem's algorithm

```
1:  r_x ← x, r_y ← y
2:  while p(r_x) ≠ p(r_y) do
3:     if p(r_x) < p(r_y) then
4:        if r_x = p(r_x) then
5:           p(r_x) ← p(r_y), break
6:        z ← p(r_x), p(r_x) ← p(r_y), r_x ← z
7:     else
8:        if r_y = p(r_y) then
9:           p(r_y) ← p(r_x), break
10:       z ← p(r_y), p(r_y) ← p(r_x), r_y ← z
```

IV. PARALLELIZING REM'S ALGORITHM

In the following we describe how one can run Rem's algorithm in parallel on a shared memory system. In doing so we make the assumption that memory read/write operations are atomic and any operations issued concurrently by different processors will be executed in some unknown sequential order unless specific constructs are used to ensure an ordering. However, two dependent operations issued by the same processor will always be applied in the same order as they are issued. This is in accordance with the memory model when using the atomic directive in OpenMP [11].

A. Using Locks

As was suggested by Cybenko et al. it is possible to implement a parallel UNION-FIND algorithm using a separate

lock for each vertex. A processor wishing to change any variable associated with a vertex during a UNION operation would then have to acquire the vertex's lock before doing so. However, when using Union-by-rank or Union-by-size this gives rise to an additional problem. Consider a UNION operation that sets $p(x) = y$. Then one might also have to update the associated (rank or size) value of y. To ensure correctness, subsequent actions by other processors must have knowledge about both the pointer change and the value update before any new operations are performed on either x or y. Thus when performing a Union-by-rank or Union-by-size operation it is necessary to lock both x and y before changing their values. The only exception to this is a Union-by-rank operation where no rank values is changed. Then it is sufficient to only lock the vertex which pointer is being set.

Since any processor acquires multiple locks sequentially it is possible that acquiring two locks will cause a deadlock unless every processor always acquires locks in a global predetermined order. Even then it is possible that a sequence of processors where each holds one lock each are waiting for each other and with only one processor being able to execute a UNION operation. Note that this is not an issue when using Union-by-index as is done in Rem's algorithm. This follows since the vertex being pointed to is not modified and thus it is only necessary to acquire one lock to execute a UNION operation.

Thus a processor that wishes to perform a UNION operation in this setting will first attempt to acquire the necessary lock. Once this has been achieved the processor will test if this vertex is still a root element. If this is the case the processor will set the parent pointer and release the lock. On the other hand if some other processor has altered the parent pointer so that the node is no longer a root, the processor will release the lock and continue executing the algorithm from its current position.

Algorithm 3 shows how lines 4 and 5 of Rem's algorithm would be implemented using OpenMP locks. Here the variable lock_array contains n locks, one for each vertex, while the variable success is a boolean variable used to indicate if the variable r_x is pointing to is still a root once the lock has been obtained. If this is not the case the processor will continue executing the regular algorithm from its current position.

Algorithm 3 Using locking in Rem's algorithm

```
1: if r_x = p(r_x) then
2:    omp_set_lock(lock_array(r_x))
3:    success = FALSE
4:    if r_x = p(r_x) then
5:       p(r_x) = p(r_y)
6:       success = TRUE
7:    omp_unset_lock(lock_array(r_x))
8:    if success then
9:       break;
```

Lines 8 and 9 of Rem's algorithm would have to modified similarly.

As was noted by Cybenko et al. performing compression operations does not require the use of locks. Intuitively this follows from the fact that each tree will always remain connected even if one compression operation undoes another one.

B. A New Lock-Free Approach

Since locks give rise to overhead and can possibly limit speedup we next consider how the UNION-FIND algorithm can be parallelized using a lock-free strategy.

To do so consider a parallel UNION-FIND algorithm that does not use locks when performing UNION operations. Then when a processor sets $p(v) = w$ in a UNION operation it is possible that this will be undone by another processor that either performs a UNION or a compression operation that sets $p(v) = z$ where $z \neq w$. Moreover, if one is using Union-by-Rank or Union-by-Size it is possible that a sequence of UNION operations can create a cycle among the parent pointers. To see this, consider three nodes v, w, and x where each of these is the root of a tree and of equal rank (or size) k. Further, let the tie breaking scheme be such that the nodes are ordered as $v < w < x$ and assume that the parent pointer is set in a UNION operation before the rank (or size) value is updated. Let t_0, t_1, and t_2 be three processors such that t_0 is trying to merge the sets that v and w belongs to, while t_1 and t_2 are both trying to merge the sets that w and x belongs to. Then the following order of operations will create a cycle involving the parent pointers of w and x: Each processor first traverses to the roots of the two component it is trying to merge. Then t_0 reads and compares the rank values of v and w (both are of rank k but $v < w$). Following this t_1 reads and compares the rank values of w and x and sets $p(w) = x$ but does not yet increase the rank value of x. Then t_0 *both* sets $p(v) = w$ and increases the rank of w to $k + 1$. Finally, t_2 reads and compares the rank values of w and x setting $p(x) = w$. At this stage a cycle has been created. A similar sequence of operations could also create a cycle if rank values are updated before parent pointers are set.

Note that this situation cannot occur if one is using Union-by-index. This follows since then the precedence of vertices is not changed during the execution of the algorithm. Thus in an algorithm that is using Union-by-index (such as the Rem algorithm) but not using locks it might be that some components are not merged in the final solution as they should have been due to some parent pointer being overwritten during the execution of the algorithm.

To get around this problem we suggest that each processor locally stores the edges that gave rise to UNION operations. When all processors have finished the regular algorithm each processor checks that both endpoints of each of its stored edges are in the same component. If this is not the case the edge is marked for later processing. When all edges have been checked there are two possibilities for how to proceed. Either one could iteratively run the parallel algorithm again until all edges are in the same component. Or if there are very few edges that requires reprocessing one could handle these sequentially.

The advantage of parallelizing the UNION-FIND algorithm in this way is that it is only necessary to syncronize the processors twice in each round, first following the execution of the regular algorithm and then after the verification stage. Also, assuming that UNION operations are evenly distributed between the processors then one would expect that each processor would have to check about n/p edges. This should be compare with the initial algorithm where each processor must process m/p edges. Thus if $n \ll m$ the running time will still be dominated by the regular algorithm. Also, if the height of the tree is reasonably low then we would expect the running time of the two stages to be $O(m/p)$ and $O(n/p)$, respectively.

One way to reduce the running time of the verification step is to instead of storing each edge x, y that gave rise to a UNION operation, to instead store the root vertex s whose parent pointer is actually being set along with the value t it is being set to. Once it has been verified that the two stored vertices are in the same component for every edge that gave rise to a UNION operation it follows (by induction) that x and y are also in the same component. The advantage of storing s and t (instead of x and y) is that one would expect that these are both closer and higher up in their respective components than x and y. We note that in this approach care must be taken so that one is not storing a value that some other processor might have changed. Thus in the code given below we are not storing the value that is being pointed to, but rather its child which is held in a local variable.

Algorithm 4 outlines one complete iteration of the algorithm that is executed by each processor. Each processor starts with a list L of edges such that $|L| = m/p$. At the end of the algorithm the marked edges could either be fed back to the algorithm as a new L or one could process each list of marked edges sequentially.

We note that following the original algorithm it is possible to rebalance the number of edges that each processor must verify, however it is unclear if the the cost of this will offset any gain.

Although the verification scheme is intended for computing a spanning tree, as given it can only guarantee to compute the connected components of G (represented by the parent pointers). The reason for this is that it is possible that the algorithm adds too many edges to the solution. This can happen if two processors are trying to merge the same components simultaneously. Then both processors might believe that they were successful with their UNION operation and (locally) add their edge to the solution. To determine if this is the case one could count, once the algorithm ends, how many vertices are roots and then compare this with the total number of edges that gave rise to UNION operations. However, this would only tell if too many edges had been added to the solution, and not which edges could be removed. One way to purge the solution of unwanted edges would be to run the locking algorithm described in Section IV-A only using as input the edges that the verification algorithm returned.

Algorithm 4 Using verification in Rem's algorithm

```
1:  U = ∅
2:  for each x, y ∈ L do
3:    r_x ← x, r_y ← y
4:    while p(r_x) ≠ p(r_y) do
5:      if p(r_x) < p(r_y) then
6:        if r_x = p(r_x) then
7:          p(r_x) ← p(r_y)
8:          store r_x and r_y in U, break
9:        z ← p(r_x), p(r_x) ← p(r_y), r_x ← z
10:     else
11:       if r_y = p(r_y) then
12:         p(r_y) ← p(r_x)
13:         store r_x and r_y in U, break
14:       z ← p(r_y), p(r_y) ← p(r_x), r_y ← z
15: Global barrier
16: for each x, y ∈ U do
17:   r_x ← x, r_y ← y
18:   while p(r_x) ≠ p(r_y) do
19:     if p(r_x) < p(r_y) then
20:       if p(r_x) = r_x then
21:         mark x, y for further processing, break
22:       r_x = p(r_x)
23:     else
24:       if p(r_y) = r_y then
25:         mark x, y for further processing, break
26:       r_y = p(r_y)
```

V. EXPERIMENTS

For the experiments we used a Dell computer running GNU/Linux and equipped with four 2.00 GHz Intel Xeon E7-4850 processors with a total of 128 GB memory. Each processor has ten cores, each which can run two threads using hyperthreading. Thus the entire configuration can run a maximum of 80 threads. All algorithms were implemented in C using OpenMP and compiled with gcc using the -O3 flag.

Our testbed consists of 22 graphs. Ten of them are real-world graphs drawn from various *scientific computing* (sc) applications from the University of Florida Sparse Matrix Collection. The remaining 12 graphs are synthetically generated using the R-MAT algorithm [12]. By combining the four input parameters of the R-MAT algorithm in various ways (the sum of the parameters needs to be equal to one), it is possible to generate graphs with varying properties. We generated three types of graphs:
(i) *Erdös-Renyi random* (er) graphs, using the set of parameters $(0.25, 0.25, 0.25, 0.25)$;
(ii) *small-world type 1* (g) graphs, using the set of parameters $(0.45, 0.15, 0.15, 0.25)$;
(iii) *small-world type 2* (b) graphs, using the set of parameters $(0.55, 0.15, 0.15, 0.15)$.
These three graph types vary widely in terms of *degree distribution* of vertices and *density of local subgraphs* and represent a wide spectrum of input types. The er graphs have *normal* degree distribution, whereas the g and b graphs contain many dense local subgraphs. The g and b graphs differ

TABLE I
STRUCTURAL PROPERTIES OF THE SCIENTIFIC COMPUTING (SC) GRAPHS IN THE TESTBED. $|V|$ AND $|E|$ ARE GIVEN IN MILLIONS.

Name	$\|V\|$	$\|E\|$	Max Deg	Avg Deg	Comp
sc1 (inline_1)	0.5	18.2	842	72	1
sc2 (ldoor)	1.0	22.8	76	48	1
sc3 (delaunay_n23)	8,4	25.2	28	6	1
sc4 (bone010)	1.0	35.3	80	72	2
sc5 (audikw_1)	0.9	38.4	344	81	1
sc6 (delaunay_n24)	16.8	50.3	26	6	1
sc7 (hollywood-2009)	1.1	56,4	11,467	99	44,507
sc8 (kron_g500-logn21)	2.1	91.0	213,904	87	553,159
sc9 (rgg_n_2_24_s0)	16.8	132.6	40	16	2
sc10 (nlpkkt240)	28.0	373.2	27	27	1

TABLE II
STRUCTURAL PROPERTIES OF THE RMAT-RANDOM (ER), RMAT-G, AND RMAT-B GRAPHS IN THE TESTBED. $|V|$ AND $|E|$ ARE GIVEN IN MILLIONS

Name	$\|V\|$	$\|E\|$	Max Deg	Avg Deg	Comp
er1	2.1	16.8	102	16	891
er2	4.2	33.6	109	16	1,828
er3	8.4	67.1	123	16	5,050
er4	16.8	134.2	138	16	12,061
g1	2.1	16.8	1,069	16	137,898
g2	4.2	33.5	1,251	16	297,570
g3	8.4	67.1	1,739	16	726,599
g4	16.8	134.2	1,814	16	1,456,228
b1	2.1	16.6	14,066	16	730,989
b2	4.2	33.3	20,607	16	1,540,016
b3	8.4	66.7	31,594	16	3,348,760
b4	16.8	133.7	42,662	16	6,807,765

primarily in the magnitude of maximum vertex degree they contain, the b graphs have much larger maximum degree and also more components.

For structural properties of the test sets see tables 1 and 2. Note that the numbers in the $|V|$ and $|E|$ columns are given in millions. The sequential running times of Rem's algorithm on these graph ranged from 0.11 (sc1) to 3.28 (er4) seconds.

We first note that in all of our experiments when using the verification algorithm we never encountered a situation where the verification step found some edge which had to be reprocessed. This is most likely due to the ratio between the number of UNION operations and the number of edges. It seems highly unlikely that two UNION operations will happen in such an interleaved fashion that one will cancel the other. Also, we did not see speedup from using more threads than the number of cores. Thus we only show number using up to 40 threads.

The first set of experiments investigates the scalability of the locking algorithm and the verification algorithm on the graphs from scientific computing (sc). The results are shown in Figure 1. For these graphs the locking algorithm gave an average maximal speedup of 20.5 with a range from 12.9 to 29.0, while the average maximal speedup for the verification algorithm was 14.4 with a range from 9.7 to 29.2. A trend that we observed when comparing the curves of individual graphs is that the verification algorithm seems to scale better when using up to approximately 16 threads while the locking algorithm scales further, thus giving a higher peak scalability. To see this effect we point to Figure 3 where the two lines labeled "ver" and "lock" shows the behaviour of the verification and the locking algorithm on the problems sc1 and sc5.

The results for the er, g, and b graphs can be found in Figure 2. Here the numbers for the er graphs are slightly better than those for the b graphs, who in turn are slightly better than the g graphs. However, the difference in performance between the different types of RMAT graphs is fairly small. The ranges in maximal speedup of all of these were from 5.2 to 14.2. The overall average for the locking algorithm was 13.9 while it was 9.1 for the verification algorithm. Thus, the sc graphs gave better speedup than the synthetically generated graphs.

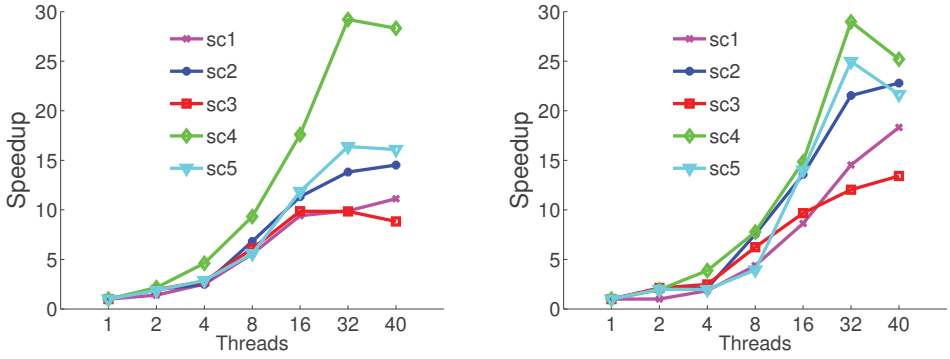
As was explained in Section IV-B it is possible that the verification algorithm will perform more UNION operations than what is needed. Although this does happen in the experiments, the relative occurrence of such operations is very small compared to the size of the graphs. The relative fraction of such occurrences divided by the number of vertices in each graph gives on average 2.6510^{-4} for 32 threads. Still, this shows that the verification algorithm only returns the connected components and not a spanning tree. As was suggested in Section IV-B we then tried to run the locking algorithm only on the edges that gave rise to UNION operations. However, this took on average 29% of the total time to run the locking algorithm for the whole graph on the same configuration.

We have run some of the graphs using the algorithm by Bader and Cong and also using their implementation of the Shiloach-Vishkin algorithm. However, since the Bader-Cong algorithm is a spanning tree algorithm it can only be used on connected graphs. On these instances we obtained speedup that was considerably lower than that of the locking and verification algorithms. As an example we show in Figure 3 the speedup obtained for the sc1 and sc5 graphs. We note that the speedup that we obtained for the Bader-Cong algorithm is considerably lower than what was reported in [3]. For most instances four threads were needed to match the sequential algorithm. We believe that this is partly due to the fact that we are using a faster sequential algorithm (Rem's algorithm) than what was previously used (depth-first-search).

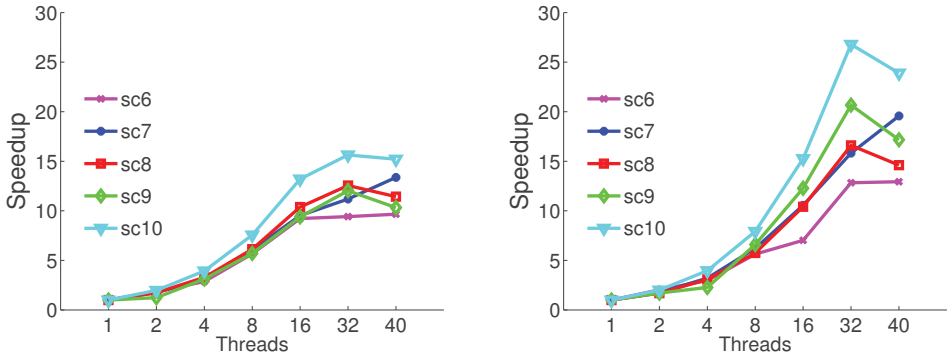
VI. CONCLUDING REMARKS

We have presented two new parallel algorithms suitable for shared memory computers for computing spanning forests. As the experiments show they offer good speedup on a large selection of test problems. Moreover, the algorithms are quite simple and only make use of standard directives in OpenMP and are thus easily portable. Comparisons with the algorithms from [3] indicate that the presented algorithms scale better. However, we note that some care should be taken when

Fig. 1. Scalability results on the algorithms using scientific computing (sc) graphs. Left column: Verification algorithm. Right column: Locking algorithm.



(a) Scientific computing (sc) graphs - set 1



(b) Scientific computing (sc) graphs - set 2

interpreting these results. The code from [3] is slightly old and has not been tuned for the current hardware platform. Thus it is conceivable that it could have been made to run faster. Still, this does not change what we believe to be one of the main advantages of our algorithms: Apart from scalability, they are simple and easy to implement.

Although the speedup given by the verification algorithm did not reach as high as that of the locking algorithm we still believe that the general idea of computing a solution without any constraints and then verifying that it is correct is something that can be used for developing parallel algorithms for other problems. We have previously used this when developing graph coloring algorithms for shared memory computers [13], [14].

One of the more common uses of the UNION-FIND algorithm is in Kruskal's algorithm for computing a minimum weight spanning tree. Since this relies on the edges of the graph being processed by increasing weight it is clear that the presented algorithm cannot be used for this purpose. However, it should be possible to obtain a reasonable approximation to the minimum weight spanning tree by letting each processor process its own edges by increasing weight.

Acknowledgements: The authors would like to thank David Bader and Guojing Cong for making their code available and also in assistance in running it.

REFERENCES

[1] R. C. Murphy, J. Berry, W. McLendon, B. Hendrickson, D. Gregor, and A. Lumsdaine, "DFS: A simple to write yet difficult to execute benchmark," in *Proceedings of the IEEE International Symposium on Workload Characterizations (IISWC06)*, 2006.

[2] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," Cray User's Group (CUG), 2010.

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

[4] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *J. ACM*, vol. 31, no. 2, pp. 245–281, 1984.

[5] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[6] M. M. A. Patwary, J. R. S. Blair, and F. Manne, "Experiments on union-find algorithms for the disjoint-set data structure," in *Proceedings of SEA 2010*, vol. 6067. Lecture Notes in Computer Science, Springer, 2010, pp. 411–423.

[7] Y. Shiloach and U. Vishkin, "An O($\log n$) parallel connectivity algorithm," *J. Alg.*, vol. 3, no. 1, pp. 57–67, 1982.

[8] G. Cybenko, T. G. Allen, and J. E. Polito, "Practical parallel algorithms for transitive closure and clustering," *Internat. J. Par. Comput.*, vol. 17, pp. 403–423, 1988.

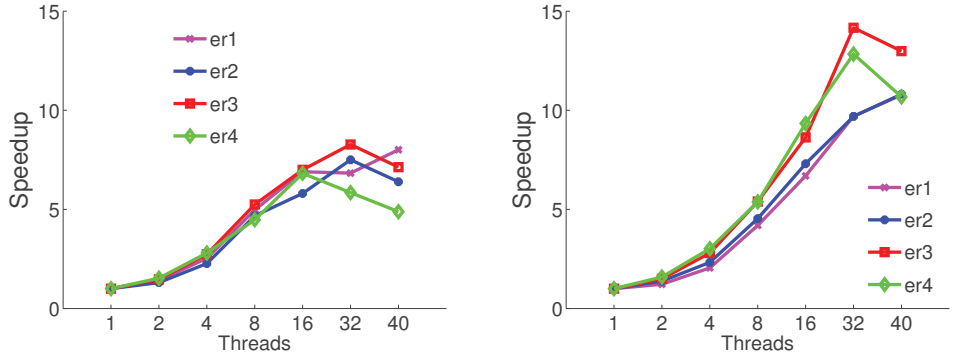
[9] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *Proceedings of the twenty-third annual ACM symposium on Theory of computing (STOC 91)*, 1991, pp. 370–380.

[10] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," vol. 6067. Lecture Notes in Computer Science, Springer, 2010, pp. 186–195.

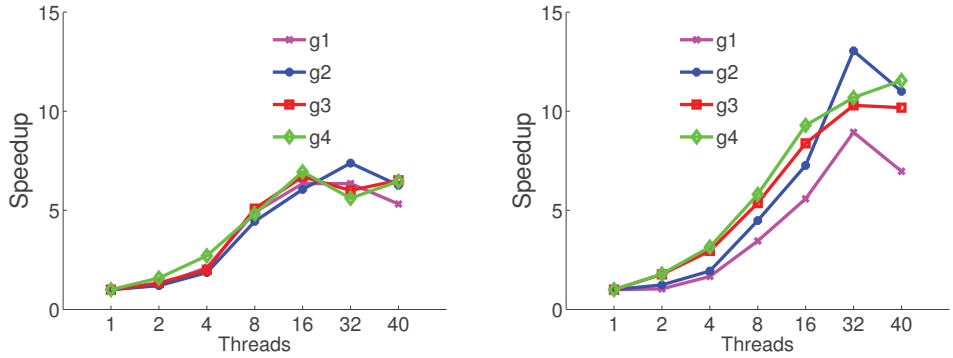
[11] "OpenMP specifications." [Online]. Available: http://openmp.org/wp/openmp-specifications/

[12] D. Chakrabarti and C. Faloutsos, "Graph mining: Laws, generators, and algorithms," *ACM Comput. Surv.*, vol. 38, no. 1, p. 2, 2006.

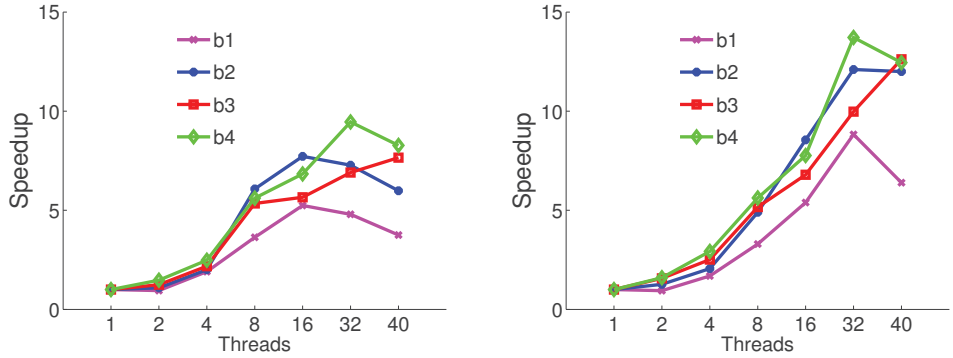
Fig. 2. Scalability results on the algorithms using synthetic graphs. Left column: Verification algorithm. Right column: Locking algorithm.



(a) RMAT-ER (er) graphs



(b) RMAT-G (g) graphs



(c) RMAT-G (g) graphs

[13] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, vol. 12, pp. 1131–1146, 2000.

[14] M. M. A. Patwary, A. H. Gebremedhin, and A. Pothen, "New multi-threaded ordering and coloring algorithms for multicore architectures," in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, ser. Euro-Par'11. Springer-Verlag, 2011, pp. 250–262.

Fig. 3. Comparisons of all four algorithms on sc1 (left) and sc2 (right).