Distributed Computing (2021) 34:413–436
https://doi.org/10.1007/s00446-020-00388-x

Concurrent disjoint set union

Siddhartha V. Jayanti[1] · Robert E. Tarjan[2]

Received: 14 November 2016 / Accepted: 22 December 2020 / Published online: 23 April 2021

Abstract
We develop and analyze concurrent algorithms for the disjoint set union ("union-find") problem in the shared memory, asynchronous multiprocessor model of computation, with CAS (compare and swap) or DCAS (double compare and swap) as the synchronization primitive. We give a deterministic bounded wait-free algorithm that uses DCAS and has a total work bound of $O\left(m \cdot \left(\log\left(\frac{np}{m}+1\right) + \alpha\left(n, \frac{m}{np}\right)\right)\right)$ for a problem with n elements and m operations solved by p processes, where α is a functional inverse of Ackermann's function. We give two randomized algorithms that use only CAS and have the same work bound in expectation. The analysis of the second randomized algorithm is valid even if the scheduler is adversarial. Our DCAS and randomized algorithms take $O(\log n)$ steps per operation, worst-case for the DCAS algorithm, high-probability for the randomized algorithms. Our work and step bounds grow only logarithmically with p, making our algorithms truly scalable. We prove that for a class of symmetric algorithms that includes ours, no better step or work bound is possible. Our work is theoretical, but Alistarh et al (In search of the fastest concurrent union-find algorithm, 2019), Dhulipala et al (A framework for static and incremental parallel graph connectivity algorithms, 2020) and Hong et al (Exploring the design space of static and incremental graph connectivity algorithms on gpus, 2020) have implemented some of our algorithms on CPUs and GPUs and experimented with them. On many realistic data sets, our algorithms run as fast or faster than all others.

Keywords Algorithms · Data structures · Concurrent · Union find · Linearizable · Wait free

1 Introduction

As data sets get bigger and bigger, it becomes more and more important to harness the potential of parallelism to solve computational problems—even linear time is too slow. In the late twentieth century, many beautiful and efficient algorithms were developed in the PRAM (parallel random access machine) model, which assumes a memory shared among many synchronized processors. In practice, however, synchronization is expensive or may not be possible. A weaker model that has attracted much attention in the distributed systems community is the APRAM (asynchronous parallel random access machine) model, in which a common memory is shared among many unsynchronized processors. In the most general version of this model, any processor can be arbitrarily slow compared to any other.

Obtaining efficiency bounds in the APRAM model is extremely challenging: the use of locks, for example, seems to make it impossible to guarantee efficiency, since one process could set a lock and then go to sleep indefinitely, blocking progress by any other process that needs access to the same resource. To overcome this problem, systems researchers have invented synchronization primitives that do not use locks, notably CAS (compare and swap) [18], transactional memory [20], and others. These primitives allow at least the possibility of obtaining good efficiency bounds for asynchronous concurrent algorithms. Yet, except for "embarrassingly parallel" computations, this possibility is almost

This paper is a combination and revision of [28] and part of [29]. Research at Princeton University partially supported by the Princeton University Innovation Fund for Industrial Collaborations and gifts from Microsoft. Part of this work was done at the first AlgoPARC workshop, funded by NSF grant CCF-1745331. The first author is also grateful for funding received through the NDSEG fellowship given by the Department of Defense.

✉ Robert E. Tarjan
ret@princeton.edu

Siddhartha V. Jayanti
jayanti@mit.edu

1 Computer Science and Artificial Intelligence Laboratory (CSAIL), Massachusetts Institute of Technology, Cambridge, MA, USA

2 Department of Computer Science, Princeton University, Princeton, NJ, USA

unrealized. Indeed, we know of only one example of a concurrent data structure (other than our work, to be described) for which a work bound without a term at least linear in the number of processes has been obtained. This is an implementation by Ellen and Woefel [11] of a fetch-and-increment object.

An important problem in data structures that could benefit from an efficient concurrent algorithm is *disjoint set union*, also known as the *union-find* problem. The simplest version of this problem requires maintaining a collection of disjoint sets, each containing a unique element called its *leader*, under two operations:

find(x): return the leader of the set containing element x.

unite(x, y): if elements x and y are in different sets, unite these sets into a single set and designate some element in the new set to be its leader; otherwise, do nothing.

Each initial set is a singleton, whose leader is its only element. Note that the implementation is free to choose the leader of each new set produced by a *unite*. This freedom simplifies concurrent implementation, as we discuss in Sect. 4. Other versions of the problem add operations for initializing singleton sets and for maintaining and retrieving information about the sets such as names or sizes. We study the simplest version but comment on extensions in Sect. 9.

Applications of sequential disjoint set union include storage allocation in compilers [32], finding minimum spanning trees using Kruskal's algorithm [31], maintaining the connected components of an undirected graph under edge additions [10,17,44], testing percolation [40], finding loops and dominators in flow graphs [12,42,43], and finding strong components in directed graphs. Some of these applications, notably finding connected components [24,27,33,37,39,41] and finding strong components, are on immense graphs and could potentially benefit from the use of concurrency to speed up the computation. For example, model checking requires finding strong components in huge, implicitly defined directed graphs [4,6,46]. There are sequential linear-time strong components algorithms [38,41], but these may not be fast enough for this application. The sequential algorithms use depth-first search [41], which apparently cannot be efficiently parallelized [36]. If one had an efficient concurrent disjoint set union algorithm one could use it in combination with breadth-first search to potentially speed up model checking. This application, described to the second author by Leslie Lamport, was the original motivation for our work.

The classical sequential solution to the disjoint set union problem is the *compressed tree* data structure [13,15,22, 44,45]. With appropriate tree linking and path compaction rules, m operations on sets containing a total of n elements take $O(m\,\alpha(n, m/n))$ time [16,44,45], where α is a functional inverse of Ackermann's function, defined in Sect. 3. Three linking rules that suffice are linking by size [44], linking by rank [45], and linking by random index [16]; three compaction rules that suffice are compression [16,44,45], splitting [16,45], and halving [16,45].

Perhaps surprisingly, there has been almost no previous research on wait-free concurrent disjoint set union. We have found only one such effort, that of Anderson and Woll [3]. Their work contains a number of significant ideas that are the genesis of our results, but it has many flaws that reveal the subtlety of the problem. We use their concurrency model. In one of our linking algorithms we use DCAS (double compare and swap) as a synchronization primitive, whereas they used only the weaker CAS (compare and swap) primitive.

Anderson and Woll considered an alternative formulation of the problem in which sets do not have leaders and the two operations are *same-set*(x, y), which returns true if x and y are in the same set and false otherwise, and *unite*(x, y), which combines the sets containing x and y into a single set if these sets are different. (We discuss *same-set* further in Sect. 4.) They attempted to develop an efficient concurrent solution that combines linking by rank with a concurrent version of path halving. They claimed a bound of $O(m \cdot (p + \alpha(m, 1))$ on the total work, where p is the number of processors. (They did not treat n as a separate parameter.). Their linking method can produce paths of $\Omega(p)$ nodes of equal rank. The $O(mp)$ term in their work bound accounts for such paths. Their proof of their upper bound is not correct, because they did not consider interference among different processes doing halving on intersecting paths. Whether or not their bound is correct, it is easy to show that their algorithm can take $\Omega(np)$ work to do $n-1$ unite operations, compared to the $O(n\alpha(n, 1))$ time required by one process. Thus in the worst case their work bound gives essentially no speedup.

Anderson and Woll also claimed a work bound of $O(m \cdot (\alpha(m, 1) + \log^* p))$ for a synchronous PRAM algorithm that uses deterministic coin tossing [9] to break up long paths of equal-rank nodes. They provided no details of this algorithm and no proof of the work bound. We think that their bound is incorrect and that the work bound of their algorithm is $\Omega(n \log p)$, since it is easy to construct sets of operations that do linking by rank exactly but such that concurrent finds with halving take $\Omega(\log p)$ steps per find, even on a PRAM. See Sect. 8. Deterministic coin tossing is a good idea, however: Using it along with the ideas in the present paper and a new one we have developed deterministic set union algorithm in the APRAM model using only CAS for synchronization, at the cost of a multiplicative $\log^* p$ factor in the work bound. See Sect. 9

In this paper we apply the ideas of Anderson and Woll and some additional ones to develop several efficient concurrent algorithms for disjoint set union. We give three concurrent implementations of *unite*, one deterministic and the other two randomized. The deterministic method uses DCAS to do linking by rank. The randomized methods use only CAS: one does linking by random index, the other does randomized linking by rank. We also give two concurrent implementations of path splitting, *one-try* and *two-try splitting*. The former is simpler, but we are able to prove slightly better bounds for the latter, bounds that we think are tight for the problem.

We prove that any of our linking methods in combination with one-try splitting does set union in $O\left(m \cdot \left(\log\left(\frac{np^2}{m}+1\right)+\alpha\left(n, \frac{m}{np^2}\right)\right)\right)$ work, and in combination with two-try splitting in $O\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n, \frac{m}{np}\right)\right)\right)$ work. Each set operation takes $O(\log n)$ steps. These bounds are worst-case for deterministic linking and high-probability for randomized linking. The $O(\log n)$ step bound per operation holds even without path splitting; without splitting, the work bound is $O(m \log n)$. The work and step bounds for randomized linking by rank hold even for an adversarial scheduler, provided that scheduling is based only on information sent to the scheduler, or we allow a form of CAS that writes a random bit. The work and step bounds for linking by random index hold provided that the randomization is independent of the order in which the unite operations are executed, or, more precisely, independent of the "linearization order" of the unite operations. (We define linearization order in Sect. 2.) We also show that $\Omega\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n, \frac{m}{np}\right)\right)\right)$ work is needed in the worst case for any algorithm satisfying a symmetry assumption, which implies that our work bound for two-try splitting is best possible for such algorithms.

Our work is theoretical, but [1,10,17] have implemented some of our algorithms on CPUs and GPUs and experimented with them. On many realistic data sets, our algorithms run as fast or faster than all others.

The remainder of our paper contains 8 sections. Section 2 describes our concurrency model. Section 3 describes the compressed tree data structure and sequential algorithms for disjoint set union. Section 4 presents concurrent linking by index, a special case of which is concurrent linking by random index, and one-try and two-try splitting. Section 5 presents preliminary versions of deterministic and randomized linking by rank. These versions rely on some simplifying assumptions that we eliminate in Sect. 6. Section 7 gives upper bounds on the total work of our algorithms. Section 8 presents lower bounds. Section 9 contains some final remarks and open problems.

2 Concurrency model

Our concurrency model is the same as that of Anderson and Woll: a shared memory multiprocessor, otherwise known as an asynchronous random-access machine (APRAM). We assume that p processes run concurrently but asynchronously, each doing a different set operation. Each process has a private memory. In addition, all processes have access to a shared memory that supports concurrent reads but not concurrent writes.

To provide synchronization of writes to shared memory, we use the *compare and swap* primitive CAS(x, y, z). Given the address x of a block of shared memory and two values y and z, this operation tests whether block x holds value y; if so, it stores value z in block x (overwriting y) and returns true; if not, it returns false. We also consider the two-block extension DCAS(u, v, w, x, y, z). Given the addresses u and x of two blocks of shared memory and four values v, w, y, and z, this operation tests whether block u holds value v and block x holds value y; if both are true, it stores value w in block u and value z in block x and returns true; if not, it returns false. These operations are atomic: once one starts, it completes before any other operation can read, write, CAS, or DCAS the affected block or blocks. Although both CAS and DCAS return a value indicating success or failure, many of our algorithms do not actually use these values.

In one version of our randomized linking algorithm we use the following randomized version of CAS: atomic operation CAS$(x$, y, \$) tests whether the value of x is y and, if so, sets the value of x equal to true or false, each with probability $1/2$. Such a randomized atomic write operation has been used in algorithms for achieving consensus [7].

Many current hardware designs include CAS as an instruction; DCAS was supported on the Motorola 68030 [35] but not on any current hardware, as far as we know. As we demonstrate in Sect. 5, it is straightforward to implement linking by rank using DCAS, but much harder using only CAS.

We study concurrent algorithms for disjoint set union that are *linearizable* [23] and *bounded wait-free* [18]. To be linearizable means that (i) the outcome of a concurrent execution is the same as if each set operation were executed instantaneously at some distinct time (its linearization time) during its actual execution and (ii) the sequential execution sequence given by the linearization times is correct; that is, all find operations produce answers that are correct at their linearization times. The linearization times define a total order of the operations, called the *linearization order*. Although we focus on linearizable algorithms, some applications of disjoint set union may not require linearizability for correctness. We briefly discuss this issue in Sect. 9, and leave further investigation as an open problem.

To be bounded wait-free means that every operation finishes in a bounded number of its own steps. The *total work* done by a concurrent solution is the total number of steps done by all processes to complete all operations.

Two weaker progress properties than bounded wait-freedom are *wait-freedom* and *lock-freedom* [21]. A concurrent solution is wait-free if every process is guaranteed to finish. It is lock-free if every operation can execute its next step when it chooses to do so, and at least one process is guaranteed to finish its operation. In general a lock-free solution need not be wait-free, and a wait-free solution need not be bounded wait-free. In our version of disjoint set union, the number of elements is fixed, which makes it easy to guarantee bounded wait-freedom. This remains true if we add an operation that allows the creation of singleton sets containing new elements, as long as the total number of set operations is bounded. If we allow an unbounded number of singleton sets to be created, then our solutions are no longer wait-free, but they remain lock-free. In this case there are no meaningful work bounds.

3 Data structure and sequential algorithms

Our concurrent disjoint set union algorithms use the same data structure as the best sequential algorithms: a *compressed forest*. This forest contains one rooted tree per set, whose nodes are the elements of the set and whose root is the set leader. Each node x has a pointer $x.p$, to its parent if it has a parent or to itself if it is a root.

In this section we explain the sequential set union algorithms. We present pseudo-code for the *same-set* and *find* procedures as Algorithm 1, and procedures for *unite* and its helper-method *link* as Algorithm 2. The sequential algorithm pseudo-code we present is not optimized for brevity. Rather, we take care to present pseudo-code that is as similar to the forthcoming concurrent algorithms as possible, thereby highlighting the key observations and distinctions that arise in the concurrent code.

The sequential algorithm for *find*(x) follows parent pointers from x until reaching a node u that points to itself, optionally *compacts* the find path (the path of ancestors from x to u) by replacing the parent of one or more nodes on the find path by a proper ancestor of its parent, and returning u. *Naïve find* does no compaction. Three good compaction rules are *compression*, *splitting*, and *halving*. Compression replaces the parent of every node on the find path by the root u. Splitting replaces the parent of every node on the find path by its grandparent. Halving replaces the parent of every other node on the find path by its grandparent, starting with x. Figure 1 illustrates how these algorithms restructure a path of nodes when *find* is called on node 8, the bottom-most node.

The sequential implementation of *same-set*(x, y) does *find*(x) and *find*(y), returning the roots u and v of the trees containing x and y, respectively, and returns true if $u = v$, false otherwise. Algorithm 1 is the pseudo-code for *same-set* and the variations of *find*.

```
procedure same-set($x, y$)
    $u \leftarrow find(x)$
    $v \leftarrow find(y)$
    return $u = v$

procedure findNaïve($x$)
    $u \leftarrow x$; $v \leftarrow u.p$
    while $v \neq u$ do
        $u \leftarrow v$; $v \leftarrow u.p$
    return $v$

procedure findCompress($x$)
    $root \leftarrow findNaïve(x)$
    $u \leftarrow x$
    while $u \neq root$ do
        $v \leftarrow u.p$; $u.p \leftarrow root$; $u \leftarrow v$
    return $root$
procedure findSplit($x$)
    $u \leftarrow x$; $v \leftarrow u.p$; $w \leftarrow v.p$
    while $v \neq w$ do
        $u.p \leftarrow w$; $u \leftarrow v$; $v \leftarrow u.p$; $w \leftarrow v.p$
    return $v$

procedure findHalve($x$)
    $u \leftarrow x$; $v \leftarrow u.p$; $w \leftarrow v.p$
    while $v \neq w$ do
        $u.p \leftarrow w$; $u \leftarrow w$; $v \leftarrow u.p$; $w \leftarrow v.p$
    return $v$
```

Algorithm 1: Sequential same-set algorithm with alternative implementations of find. The pseudo-code is written to match the forthcoming concurrent version as closely as possible, so that the key differences are more clear.

```
procedure unite($x, y$)
    $u \leftarrow find(x)$
    $v \leftarrow find(y)$
    if $u \neq v$ then $link(u, v)$

procedure linkByRank($u, v$)
    $r \leftarrow u.r$; $s \leftarrow v.r$
    if $r < s$ then $u.p \leftarrow v$
    else if $r > s$ then $v.p \leftarrow u$
    else
        $v.r \leftarrow v.r + 1$
        $u.p \leftarrow v$
procedure linkByIndex($u, v$)
    if $u < v$ then $u.p \leftarrow v$
    else $v.p \leftarrow u$

procedure linkBySize($u, v$)
    if $u.size \leq v.size$ then
        $u.p \leftarrow v$
        $v.size \leftarrow v.size + u.size$
    else
        $v.p \leftarrow u$
        $u.size \leftarrow u.size + v.size$
```

Algorithm 2: Sequential unite algorithm, with multiple implementations of link. The pseudo-code is written to match the forthcoming concurrent version as closely as possible so that the key differences are clear.

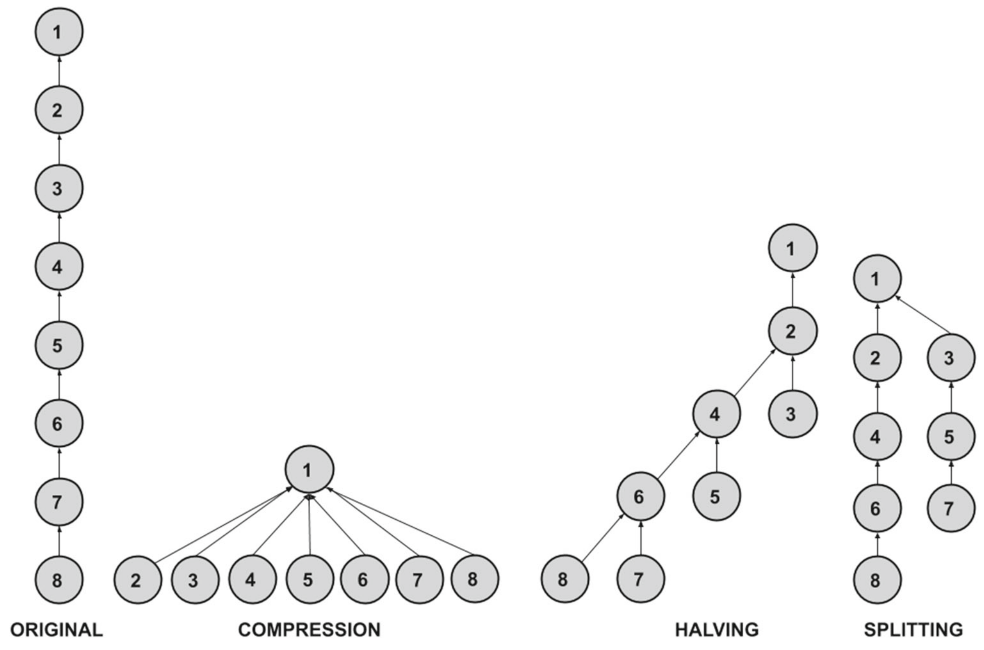

Fig. 1 The results of running *find*(8) on the original path with the three different types of compaction: compression links all the nodes on the find path directly to the root, thereby "compressing" the path; halving links alternating nodes on the find path to their grandparents, thereby creating a path of "half" the length with nodes hanging off; splitting links every node on the find path to its grandparent, thereby "splitting" one path in two

The sequential implementation of *unite*(x, y) does *find*(x) and *find*(y), returning the roots u and v of the trees containing x and y, respectively, and tests whether $u = v$. If $u \neq v$, it *links* u and v by making one the parent of the other. Three good linking rules are *linking by size*, *linking by rank*, and *linking by random index*. *Linking by size* maintains the $size$ (number of nodes) of each tree in its root, and makes the root of the tree of larger size the parent of the other, breaking a tie arbitrarily. *Linking by rank* maintains a non-negative integer *rank* for each root, initially zero, and makes the root of larger rank the parent of the other, breaking a tie by adding one to the rank of one of the roots. In the pseudo-code, we use $u.r$ to represent node u's rank. *Linking by index* chooses a fixed total order of the nodes and makes the root of larger index the parent of the other. *Linking by random index* is the special case of linking by index that chooses the total order of nodes uniformly at random. Algorithm 2 is the pseudo-code for *unite* and the variations of *link*.

Linking by size, rank, or random index combined with naïve find, compression, splitting or halving gives an algorithm that takes $O(\log n)$ time for an operation on a set or sets containing n elements, worst-case for deterministic linking, high-probability for linking by random index. Use of compaction improves the amortized time per operation: any combination of compression, splitting, or halving with linking by size, rank, or random index gives an algorithm that takes $O(m \cdot \alpha(n, m/n))$ time to do m operations on sets containing a total of n elements. The bound is worst-case for linking by size or rank, average-case for linking by randomized index. Here α is a functional inverse of Ackermann's function defined as follows. We recursively define $A_k(n)$ for non-negative integers k and n as follows:

$$A_0(n) = n + 1;\ A_k(0) = A_{k-1}(1) \text{ if } k > 0;$$
$$A_k(n) = A_{k-1}(A_k(n-1)) \text{ if } k > 0 \text{ and } n > 0.$$

For a non-negative integer n and non-negative real-valued d,

$$\alpha(n, d) = \min\{k > 0 \mid A_k(\lfloor d \rfloor) > n\}$$

Lemma 1 *$A_k(n) < \min\{A_{k+1}(n), A_k(n+1)\}$, i.e., $A_k(n)$ is strictly increasing in k and n.*

Proof The proof is by double induction on k and n. $A_0(n) = n+1 < n+2 = A_0(n+1)$, and $A_0(0) = 1 < 2 = A_1(0)$. Let $k > 0$. Suppose the lemma holds for $k' < k$ and all n. Then $A_k(0) < A_k(0) + 1 = A_0(A_k(0)) \leq A_{k-1}(A_k(0)) = A_k(1) = A_{k+1}(0)$. Thus the lemma also holds for k and $n = 0$. Let $k > 0$ and $n > 0$. Suppose the lemma holds for $k' < k$ and all n, and for k and $n - 1$. Then $A_k(n) < A_k(n) + 1 = A_0(A_k(n)) A_{k-1}(A_k(n)) = A_k(n+1)$, and $A_k(n) = A_k(A_0(n-1)) < A_k(A_{k+1}(n-1)) = A_{k+1}(n)$. □

Corollary 1 *$\alpha(n, d)$ is non-decreasing in n and non-increasing in d.*

Our goal is to extend at least one sequential set union algorithm to the concurrent model of Sect. 2 and to obtain an almost-linear work bound that grows sublinearly with p, the number of processes. For convenience in stating bounds, we assume that $2 \leq p \leq n \leq m$, and that there is at least one unite of different elements. We denote the base-two logarithm by lg.

4 Concurrent linking and splitting

Concurrency significantly complicates the implementation of the set operations. One complication is that processes can interfere with each other by trying to update the same field at the same time, requiring our algorithms to be robust to such interference. Consider doing unites concurrently. To do *unite*(x,y), we can start as in the sequential case by finding the roots u and v of the trees containing x and y, respectively. Then we can try to link u and v by doing a CAS to make v the parent of u or vice-versa. But we must allow for the possibility that the CAS can fail, for example if it tries to make v the parent of u but in the meantime some other process makes another node the parent of u. If this happens we must retry the unite. When retrying, we start the new finds at u and v rather than at x and y, to avoid revisiting nodes. Anderson and Woll [3] proposed this method; the following pseudocode implements it. Method *link*(u, v), to be defined, tries to make one of two roots u and v the parent of the other.

```
1: procedure unite($x$, $y$)
2:     $u \leftarrow find(x)$; $v \leftarrow find(y)^*$
3:     while $u \neq v$ do
4:         $link(u, v)^*$
5:         $u \leftarrow find(u)$; $v \leftarrow find(v)^*$
```

Algorithm 3: : Concurrent unite algorithm.

In this and subsequent implementations, asterisks denote linearization points. The linearization point of a unite is the linearization point of the successful link if there is one, or the linearization point of the last find if no link is successful.

Concurrency also imposes constraints on the linking rule. We need to prevent concurrent links from creating a cycle of parent pointers other than a loop at a root. For example, three concurrent links might make v the parent of u, w the parent of v, and u the parent of w. The simplest way to prevent such cycles is to do linking by index, which we can implement using CAS. We denote the total order of nodes by "$<$". The following pseudocode implements linking by index:

```
1: procedure link($u$, $v$)
2:     if $u < v$ then CAS($u.p$, $u$, $v$)*
3:     else CAS($v.p$, $v$, $u$)*
```

Algorithm 4: : Concurrent linking by index algorithm.

The linearization point of the link is its CAS. A link is *successful* if its CAS returns true. For any total order, linking by index guarantees acyclicity. *Linking by random index* is the special case of linking by index that chooses the total order uniformly at random.

With this implementation of link, a link can succeed even though the new parent itself becomes a child of another node at the same time. Fortunately this affects neither correctness nor efficiency. We could prevent this anomaly by using DCAS to do links, which allows us to guarantee that the new parent remains a root. But this has two drawbacks. First, it uses DCAS, whereas our goal is to use only CAS if possible. Second, if all links are done using DCAS, the total work can be linear in p, as we discuss in Sect. 5.1.

Next we consider finds. Concurrent naïve finds do not interfere with each other, since such finds do not change the data structure. Thus we can do such finds exactly as in the sequential case. The following pseudocode implements concurrent naïve find:

```
1: procedure find($x$)
2:     $u \leftarrow x$; $v \leftarrow u.p^*$
3:     while $v \neq u$ do
4:         $u \leftarrow v$; $v \leftarrow u.p^*$
5:     return $u$
```

Algorithm 5: : Concurrent Naïve find algorithm.

The linearization point of a find is the last update of v.

Concurrent finds with compaction *can* interfere with each other. Consider a sequential find with splitting. Let u be the current node visited by the find. One step of the find consists of setting $v = u.p$; setting $w = v.p$; and, if $v \neq w$, replacing $u.p$ by w and then setting $u = v$. Steps continue until $v = w$, when the find finishes by returning v. The only update to the data structure in a step is the replacement of $u.p$ by w. We obtain a concurrent version of splitting by using CAS($u.p$, v, w) to do the update. The following pseudocode implements this method, which we originally presented in [28] and which is based on Anderson and Woll's version of find with halving:

```
1: procedure find($x$)
2:     $u \leftarrow x$; $v \leftarrow u.p$; $w \leftarrow v.p^*$
3:     while $v \neq w$ do
4:         CAS($u.p$, $v$, $w$); $u \leftarrow v$; $v \leftarrow u.p$; $w \leftarrow v.p^*$
5:     return $v$
```

Algorithm 6: : Concurrent Find with One-Try Splitting algorithm.

The linearization point of a find is the last update of w. We call this method *one-try splitting* because it tries once to update $u.p$ and then changes the current node from u to v, whether or not the update of $u.p$ has succeeded.

Concurrent splits can produce anomalies that are not possible if splits are sequential, as a simple example shows. (See Fig. 2) Suppose a, b, c, d, e is a path in a tree built by linking by index, and that four processes, 1, 2, 3, and 4 begin concurrent finds with one-try splitting starting at a, a, b, and

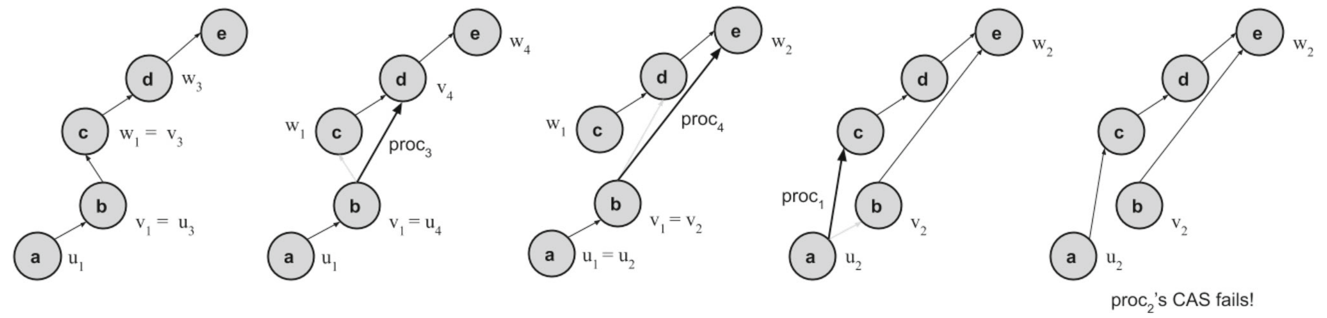

Fig. 2 Interference in concurrent splitting: process 1 updates a's parent to a node that is not its ancestor; process 2's CAS fails. These difficulties do not occur in the sequential setting

b, respectively. We denote the local variables of process i by u_i, v_i, w_i. First, process 1 sets $u_1 = a, v_1 = a.p = b$, and $w_1 = b.p = c$. Second, process 3 sets $u_3 = b, v_3 = c, w_3 = d$, and replaces $b.p$ by d. Third, process 4 sets $u_4 = b, v_4 = d, w_4 = e$, and replaces $b.p$ by e. Fourth, process 2 sets $u_2 = a, v_2 = b$, and $w_2 = d$. Fifth, process 1 replaces $a.p$ by c. Sixth, process 2 attempts to replace $a.p$ by d but fails, because process 1 changed $a.p$ after process 2 read it. Observe that just before process 1 replaces $a.p$ by c, c is not an ancestor of a, even though it was when process 1 read it. This threatens correctness. Furthermore, even though the failure of process 2 to update $a.p$ guarantees that $a.p$ has changed since process 2 read it, the new value of $a.p$, namely c, is not an ancestor of the current grandparent of a, namely e, violating a property used in the analysis of sequential splitting. Finally, even though the new parent c of a is higher in index than the old parent b of a (as we prove in Theorem 1), the new grandparent d of a is *lower* than the old grandparent e of a.

Fortunately, correctness requires only a weak property of compaction, one that holds for one-try splitting and many other methods. We introduce an analytic tool called the *union forest* in order to explain the property. We assume that if a compaction changes the parent $w.p$ of a node w by a CAS, $w.p \neq w$ just before the change; that is, w is not a root. Equivalently, only a link can change the parent of a root. Suppose we do linking by index. Consider a fixed history, i.e. a concurrent execution of several *unite, same-set*, and *find* operations by different processes up to some time t. For this fixed history, the *union forest* is the set of trees such that the parent of a node w is the *first* value, other than w, that $w.p$ takes on during the history; if $w.p = w$ throughout the history, then w is a root in the union forest.

Claim The union forest is a forest.

Proof Since linking is by index, when a link changes the parent of a root w from w to z, $z > w$. Hence the union forest contains no cycles of parent pointers other than loops. Thus the union forest is indeed a forest. □

We call a compaction method *valid* if it visits nodes on a single path in the union forest, each vertex visit takes $O(1)$ steps, each replacement of a parent w by another node z (of which there may be none) is such that z is a proper ancestor of w in the union forest, and the linearization point of the find doing the compaction is the last read of a parent that returns the node itself. The parent update requirements are only with respect to the fixed union forest, *not* with respect to the dynamically changing actual forest maintained by the data structure. In particular, although find with splitting can change the parent $w.p$ of a node w to a non-ancestor of w in the actual forest (see Fig. 2), it cannot do so in the union forest. Indeed, splitting is valid.

The following theorem states the correctness of linking by index with finds that do valid compactions.

Theorem 1 *Any disjoint set union algorithm that does linking by index in combination with finds that do valid compaction is linearizable. The parent of any non-root node has higher index than the node, and the parents define a set of trees that partition the nodes into the correct disjoint sets. Furthermore each set operation stops in $O(h)$ steps, where h is the height (maximum number of edges on a path) of the union tree, so the algorithm is bounded wait-free.*

Proof An induction on the number of parent changes using the transitivity of "<" shows that the parent of any node never has smaller index than the node. This implies that the only cycles are loops at roots. Parent changes done during compactions do not change the node partition defined by the trees. A link that makes v the parent of u must be such that u is a root before the link, $u < v$, and u and v are in the trees containing the two nodes x and y that are the inputs to the unite that does the link. It follows by induction on the number of parent changes that at all times the trees correctly partition the nodes: a find cannot change this partition, and a link unites the trees containing the nodes that are the inputs to the corresponding unite. Correctness of the linearization points follows in a straightforward way by induction on the number of parent changes: When a find reads the parent of a root, that root at that moment is the leader of the set containing the input to the find; when a unite does a link, the partition remains correct; when a test "$u \neq v$" in unite returns false, the inputs to the unite are in the same set.

Since the nodes visited during a find are on a single path in the union forest, and each node visit takes $O(1)$ steps, each find stops in $O(h)$ steps. (Our assumption that there is at least one unite of different elements implies $h > 0$.) The nodes visited during a unite are on two paths in the union forest. Consider the node visits in the order they occur. Each node visit takes $O(1)$ steps, but a node can be visited many times. This can only happen while it is a root; once it becomes a child, it can only be visited once more (as the input to a find). Consider the nodes u and v just before an execution of the test "$u \neq v$" in unite. Each of u and v was a root at some time during the find that computed it. If the test "$u \neq v$" succeeds, whichever of u and v is smaller in the total order will be a child after the next link (whether or not the link succeeds). Suppose without loss of generality it is u. We charge the next visits to u and v to u becoming a child. There are at most $2h$ such events. It follows that the total number of node visits during the unite, and hence the total number of steps, is $O(h)$. □

Having dealt with correctness, we discuss concurrent compaction in more detail. The monotonicity of parents (each new parent is higher in index than the old one) allows us to extend the analysis of sequential splitting to one-try splitting, although the extension is not straightforward. On the other hand, the analysis of sequential halving relies on monotonicity of grandparents, which fails in the concurrent setting, as our example above shows. Anderson and Woll [3] claimed a good work bound for their concurrent version of halving, but they overlooked the problem of non-monotonicity. We see no way to get a good work bound for their method.

Even though we can prove good efficiency bounds for one-try splitting, we can prove slightly better bounds for a related compaction method that tries to change each parent pointer twice instead of once. We call this method *two-try splitting*. The following pseudocode implements find with two-try splitting:

```
1: procedure find(x)
2:     u ← x; v ← u.p; w ← v.p*
3:     while v ≠ w do
4:         CAS(u.p, v, w); v ← u.p; w ← v.p
5:         CAS(u.p, v, w); u ← v; v ← u.p; w ← v.p*
6:     return v
```

Algorithm 7: : Concurrent Find with Two-Try Splitting algorithm.

The linearization point of a find is the last assignment to w. If every attempted parent change succeeds, the effect of a single two-try split is to replace the parent of every other node on the find path by its great-grandparent. This splits the original path into two paths, each containing half the nodes on the original path, but the split is different from that produced by one-try splitting: if the nodes on the original path are numbered consecutively from 1, the latter produces a path of nodes 1, 3, 5, 7... and another path of nodes 2, 4, 6, 8... ; the former produces a path of nodes 1, 4, 5, 8, 9... and another path of nodes 2, 3, 6, 7, 10, 11...

A variant that has the same work bounds as two-try splitting is *conditional two-try splitting*, in which the second try occurs only if the first one fails. We omit a detailed discussion of this variant, since its pseudocode is a bit longer and it is unclear whether avoiding extra parent changes improves efficiency.

Both one-try and two-try splitting are valid compaction methods, so Theorem 1 holds for both of them.

We conclude this section by presenting Anderson and Woll's concurrent implementation of *same-set*, which gives an extension of our algorithms to their formulation of the problem. It is easy to do *same-set*(x, y) in the sequential setting: find the root u of the tree containing x, find the root v of the tree containing y, and test whether $u = v$. As Anderson and Woll observed, this does not suffice in the concurrent setting, because u might no longer be a root when the equality test occurs, possibly invalidating the test. Their solution has three cases. If $u = v$, return true: x and y are in the same tree when the test occurs, and remain in the same tree. If $u \neq v$, test whether u is still a root. If so, return false: x and y were in different trees when v was computed, since u and v were different roots. If not, redo the computation: do new finds from u and v, and repeat the test or tests. The following pseudocode implements this method:

```
1: procedure same-set(x, y)
2:     u ← find(x); v ← find(y)*
3:     while u ≠ v do
4:         w ← u.p
5:         if u = w then return false
6:         u ← find(u); v ← find(v)*
7:     return true
```

Algorithm 8: : Concurrent same-set algorithm.

The linearization point of a same-set is the last assignment to v. All our analyses of find and unite extend to include *same-set* as an allowed operation.

5 Concurrent linking by rank

To obtain a good work bound, we combine one-try or two-try splitting with a good linking method. Linking by random index is one such method, but our analysis of it assumes that the scheduling of CAS instructions is independent of the random node order. This assumption is questionable; if it fails, the work bound becomes much worse as a function of p, as we show in Sect. 8. To overcome this, we develop two concurrent versions of linking by rank, one deterministic and one randomized, both of which have good work bounds. To simplify our descriptions, we assume for the moment that the rank and parent of a node can be stored in a single block of memory that is updatable by one CAS instruction. In Sect. 6 we show how to eliminate this assumption.

Both of our versions of linking by rank are refinements of a generic method. The generic method links roots of different ranks using CAS, and links roots of the same rank using method *elink*, to be defined. The rank of node u is $u.r$, initially zero. The following pseudocode implements the generic method:

```
1: procedure link($u$, $v$)
2:     $r \leftarrow u.r$; $s \leftarrow v.r$
3:     if $r < s$ then CAS($(u.p, u.r), (u, r), (v, r)$)*
4:     else if $r > s$ then CAS($(v.p, v.r), (v, s), (u, s)$)*
5:     else elink($u, v, r$)*
```

Algorithm 9: : Concurrent linking by rank algorithm.

Given two roots u and v with ranks r and s, respectively, this method compares r to s. If $r < s$, it uses a CAS to make v the parent of u while guaranteeing that neither the parent nor the rank of u changes in the meantime. If $r > s$, it proceeds symmetrically. If $r = s$, it does an *elink* to link u and v. A link is successful if its CAS returns true or its *elink* is successful, in which case the linearization point of the link is its CAS or that of its *elink*. Our two versions of linking by rank differ only in their implementation of *elink*.

5.1 Linking by rank via DCAS

A simple way to do *elink*(u, v, r) is to use a DCAS to make v the parent of u and increment the rank of v while guaranteeing that the ranks and parents of u and v do not change in the meantime. The following pseudocode implements this idea:

```
1: procedure elink($u$, $v$, $r$)
2:     DCAS($(u.p, u.r), (u, r), (v, r), (v.p, v.r), (v, r), (v, r+1)$)*
```

Algorithm 10: : Concurrent linking by DCAS algorithm.

An elink is successful if its DCAS returns true, in which case the linearization point of the elink is its DCAS.

Our first version of linking by rank uses this implementation of elink. The rank of a node can never decrease, and can increase only while the node is a root. It follows that the rank of a child is always strictly less than that of its parent. Linking by rank is an implicit form of linking by index: the successful links respect any total order consistent with the final ranks of nodes. Thus Theorem 1 holds for this method.

The following lemma and theorem extend known bounds on sequential linking by rank [45] to linking by rank via DCAS:

Lemma 2 *With linking by rank via DCAS, the sum of ranks is at most $n - 1$, the number of nodes of rank k is at most $n/2^k$, and the maximum rank and the height of the union forest are at most* $\lg n$.

Proof For a node to increase in rank by 1, it must be a root, and another root must become its child at the same time. It follows that the number of rank increments, and hence the sum of ranks, is at most $n - 1$, one per root that becomes a child. An induction on k shows that at most $n/2^k$ nodes can ever attain rank k. The bounds on the maximum rank and the height of the union forest follow, since no node can have rank exceeding $\lg n$. □

Theorem 2 *Linking by rank via DCAS in combination with any valid compaction method maintains the invariant that the parents define a set of trees that partition the nodes into the correct disjoint sets, and the rank of a child is less than that of its parent. Furthermore each set operation stops in $O(\log n)$ steps, so the algorithm is bounded wait-free.*

Proof The first half of the theorem follows by induction on the number of steps as in the proof of the first half of Theorem 1. A find takes $O(\log n)$ steps by the argument in the proof of Theorem 1, since the height of the union forest is $O(\log n)$ by Lemma 2. We prove the bound for unites by an extension of the argument in the proof of Theorem 1. The nodes visited during a unite are on two paths in the union forest, and on each path they are visited in increasing order by rank. Each node visit takes O(1) steps, but roots can be visited many times. We charge each repeated visit to a root either to a root becoming a child or to a root increasing in rank. Consider the nodes u and v just before an execution of the test "$u \neq v$" in unite. Each of u and v was a root at some time during the find that computed it. Suppose the test "$u \neq v$" succeeds. The next execution of link sets r to the rank of u and s to the rank of v. If $r < s$, then after the CAS either the rank of u has increased or u has become a child, whether or not the CAS succeeds. We charge the next visits to u and v to the rank increase of u or to u becoming a child. The symmetric argument applies if $r > s$. If $r = s$, at least one of u and v has increased in rank or become a child after the elink. We charge the next visits to u and v to whichever of these events has occurred. There are at most $2 \lg n$ roots that become children and at most $2 \lg n$ rank increases by Lemma 2, since for each of the two paths in the union forest the rank increases sum to at most $\lg n$. It follows that the total number of node visits during the unite, and hence the total number of steps, is $O(\log n)$. □

The efficiency of this linking method (though not its correctness) depends critically on using CAS to link nodes of different ranks, reserving DCAS for the equal-rank case. An attempted link of equal-rank nodes u and v using DCAS fails only if some other process makes u or v a non-root, or increases the rank of u or v. In the proof of Theorem 2 we charge extra node visits resulting from the failure of the DCAS to whichever of these events occurs. If we were to use DCAS to try to make a node v the parent of a node u of lower rank, the DCAS could fail because another process made v the parent of another node w. This changes neither the parent nor rank of u, nor of v, leaving us with no event to charge for extra node visits. In the worst case, $O(n)$ such links could produce $\Omega(pn)$ failures, resulting in total work linear in p. Using CAS to link nodes of different ranks eliminates these failures. Although we can avoid such interference in the disjoint set union problem as we have defined it, this is much harder to do in some extensions of the problem, as we discuss in Sect. 9.

5.2 Randomized linking by rank

To link equal-rank nodes using CAS, we need to do the parent change and the rank increment separately. The question is which one to do first. Making this decision randomly gives an approximation to linking by rank that produces few enough rank ties that we are able to get good work bounds. Since this method allows rank ties, we use linking by index to break such ties, in order to prevent the creation of non-trivial cycles of parent pointers. Assume that "<" is an arbitrary total order of the nodes. To link two equal-rank roots u and v such that $u < v$, we flip a fair coin. If it comes up heads, we attempt to make v the parent of u; if it comes up tails, we attempt to increase the rank of u. The following pseudocode implements this idea. Random Boolean method *flip* returns true with probability $1/2$ and false otherwise, independent of all other flips.

```
1: procedure elink($u, v, r$)
2:    if $u < v$ then
3:       if flip then CAS(($u.p, u.r$), ($u, r$), ($v, r$))*
4:       else CAS(($u.p, u.r$), ($u, r$), ($u, r + 1$))
5:    else
6:       if flip then CAS(($v.p, v.r$), ($v, r$), ($u, r$))*
7:       else CAS(($v.p, v.r$), ($v, r$), ($v, r + 1$))
```

Algorithm 11: : Concurrent randomized linking by rank algorithm.

An elink is successful if it does a CAS that changes a parent pointer, in which case the linearization point of the elink is its CAS.

Our second version of linking by rank uses this implementation of elink. Observe that the CAS done after a flip is almost the same whether the flip returns true or false, the only difference being the updated field (parent or rank, respectively). In our analysis we shall assume that the success or failure of the CAS following a flip is independent of the outcome of the flip. In Sect. 6 we describe how to modify the implementation to eliminate the need for this independence assumption. Randomized linking by rank is an implicit form of linking by index: links respect the total order defined by final node ranks with ties broken by "<" .

Lemma 3 *With randomized linking by rank,* (i) *any node x has $O(1)$ ancestors of the same rank, in expectation;* (ii) *the sum of ranks is at most n in expectation and $n + O(n^{1/2})$ with probability $1 - 1/n^c$ for any constant $c > 0$, where the constant factor in the "O" depends on c;* (iii) *the expected number of nodes of rank at least k is at most $n/2^k$, and with probability at least $1 - n/2^k$, all nodes have rank less than k;* (iv) *the maximum rank is at most $\lg n + 3$ in expectation and is at most $(c + 1) \lg n$ with probability at least $1 - 1/n^c$ for any positive constant c;* (v) *the depth of the union forest is at most $3 \lg n + 9$ in expectation and $O(\log n)$ with probability at least $1 - 1/n^c$ for any constant $c > 0$, where the constant factor inside the "O" depends on c; and* (vi) *for a large enough constant c and any $k > 0$, the expected number of nodes of rank less than k and height at least ck in the union forest is at most $n/2^k$.*

Proof Consider only flips that result in successful CAS operations. Each such flip produces a rank increment with probability $\frac{1}{2}$; otherwise, it makes a root into a child.

(i) The probability that a node has k ancestors of the same rank is at most $1/2^{k-1}$. Summing gives the bound.

(ii) There are at most $n - 1$ flips that make a root into a child. The sum of ranks, which is the number of rank increments, is thus at most the number of heads in a sequence of coin flips containing at most n tails, which is at most n in expectation, and at most $n + O(n^{1/2})$ with probability $1 - n^c$ for any constant $c > 0$ by a Chernoff bound [8], with the constant factor in the "O" depending on c.

(iii) The rank of a given node is at least k with probability at most $1/2^k$. The expected number of nodes of rank at least k is thus at most $n/2^k$. By a union bound, all nodes have rank less than k with probability at least $1 - n/2^k$.

(iv) For $c > 0$, the probability that the maximum rank is at least $\lg n + c$ is at most $n/2^{\lg n + c} = 1/2^c$. It follows that the expected maximum rank is at most $\lg n + \sum_{i=1}^{\infty} i/2^i \lg n + 2 \lg n + 3$, and the probability that the maximum rank exceeds $(c + 1)\lg n$ is at most $1/2^{c \lg n} = 1/n^c$.

(v) A node gains a proper ancestor of the same rank with probability at most $1/2$. Thus the expected number of proper ancestors of the same rank as that of a given node is at most $\sum_{i=1}^{\infty} i/2^i = 2$, which implies by (ii) that the expected depth of the union forest is at most $3\lg n + 9$. Let $c > 0$. By (ii) the maximum node rank is at most $(c + 3)\lg n$ with probability at least $1 - 1/n^{c+2}$. For any node, the probability that it has at least $(b + c + 3)\lg n$ proper ancestors in the union forest is at most the probability that a sequence of fair coin flips containing at most $(c + 3) \lg n$ heads contains at least $b\lg n$ tails. By a Chernoff bound, for b sufficiently large, this probability is at most $1 - 1/n^{c+2}$. The probability that at least one of the n nodes has more than $(b + c + 3)\lg n$ proper ancestors in the union forest is at most $2n/n^{c+2} \le 1/n^c$.

(vi) Let x be any node. We claim that for some $c > 0$ the probability that x has an ancestor y of rank less than k such that the path from x to y contains ck edges is at most $1/2^k$. Part (vi) follows from the claim by a union bound. To prove the claim, consider the edges on the path from x to y in the union forest. Call such an edge *good* if its ends have different ranks and *bad* otherwise. Each edge has probability at least $1/2$ of being good, independent of the status of all other edges. The claim follows by a Chernoff bound. □

Theorem 3 *Randomized linking by rank in combination with any valid compaction method maintains the invariant that the parents define a set of trees that partition the nodes into the correct disjoint sets. The parent of any non-root node has rank no less than that of the node, and if the ranks are equal, the parent has larger index. Each set operation stops in* $O(\log n)$ *steps with probability* $1-1/n^c$ *for any* $c > 0$*, where the constant factor in the "O" depends on c. The algorithm is bounded wait-free.*

Proof Except for the fact that the algorithm is bounded wait-free, the theorem follows from parts (iv) and (v) of Lemma 3 by a proof like those of Theorems 1 and 2.

To prove that the algorithm is bounded wait-free, we observe that for a node to increase in rank some larger node must have the same rank. It follows by induction that the i^{th} largest node has rank at most $i - 1$, so the maximum rank is at most $n - 1$. □

5.3 Linking by random index

With an appropriate definition of rank, Lemma 3 and Theorem 3 hold for linking by random index, under a strong independence assumption. We define the rank of node x to be $\lg n - \lg(n - x + 1)$. Thus node n has rank $\lg n$, nodes $n - 1$ and $n - 2$ have rank $\lg n - 1$, and so on. The rank of a child is no greater than that of its parent. We use these ranks only in the analysis; the implementation of the algorithm does not use them.

We assume that the random node order is independent of the linearization of the unite operations. More precisely, we assume that the node order and linearization are generated together in the following way. The implementation maintains a set U of unordered pairs $\{u, v\}$ that are candidates for linking, initially empty, and a partial order P of the nodes, initially empty, that is a total order on the nodes of any set defined by the links done so far, and that leaves any two nodes in different sets unordered. To do *link*(u, v), a process adds the unordered pair $\{u, v\}$ to U.

The scheduler sequentially removes pairs from U in arbitrary order. When removing a pair $\{u, v\}$ from U, the scheduler performs three actions. First, it modifies P by merging the total orders of the sets containing u and v, with each possible merged order equally likely. Second, if $u < v$ it sets $u.p = v$, if $v < u$ it sets $v.p = u$. This unites the sets containing u and v. The link corresponding to the pair $\{u, v\}$ succeeds. Third, the scheduler deletes from U all other pairs containing u or v. Each link corresponding to such a pair fails. When a pair is deleted from U, the process that added the pair to U proceeds with its next operations, which are the recomputing of its u and v.

The updating of P maintains the invariant that the total order of the nodes in any set defined by the links done so far is uniformly random. If there is more than one set after all unites have been done, we can extend the final partial order to a total order by merging the total orders on the final sets, with each possible merged order equally likely. The result is a uniformly random permutation of the nodes, equivalent to a uniformly random numbering of the nodes from 1 to n. Thus this implementation does linking by random index, subject to the restriction imposed on the scheduler. The execution does not change if we initially number the nodes uniformly at random but reveal to the scheduler only the total order within each set formed so far.

This implementation restricts the behavior of linking by random index in at least two different ways: in the actual implementation, the CAS operation for a link is defined by an ordered pair, not an unordered pair, so the scheduler gets information about the node order before it needs to decide which such CAS to do next. Also, in the actual implementation a link can make a root the child of a non-root, which cannot happen with the scheduler constraint. We think the latter restriction is inconsequential, but the former is significant. We thus view our analysis of linking by random index as suggestive, not definitive.

Lemma 4 *With linking by random index, if the scheduler restriction holds, then parts* (i), (v), *and* (vi) *of Lemma 3 are true, as well as the following strengthened versions of parts* (ii), (iii), *and* (iv): (ii) *the sum of ranks is at most n,*

(iii) *the number of nodes of rank at least k is at most* $n/2^k$, *and* (iv) *the maximum node rank is at most* $\lg n$.

Proof Parts (ii), (iii), and (iv) are immediate from the definition of ranks. Parts (i), (v), and (vi) follow as in the proof of Lemma 3 from the following claim: □

(*) Given a node x, each successive ancestor of x in the union forest has probability at least $1/2$ of having higher rank than its parent, independent for each ancestor.

To prove (*), consider a node x. Given an execution of the algorithm, we modify the linearization by delaying links uniting sets not containing x until such a set is a subset of a set about to be united with x. That is, let S be the current set containing x, let $\{u, v\}$ be the next pair deleted from U with u but not v in S, and let S' be the current set containing v. We modify the schedule to delay all the links forming S' until just before the link of u and v. This does not change the steps done by the execution, only their linearization order. We generate the partial order P by generating a numbering of the nodes incrementally and revealing to the scheduler only the total order within each set constructed so far. Initially we assign a number uniformly at random to x. Subsequently when the scheduler removes a pair $\{u, v\}$ from U, if u and/or v is unnumbered we assign it a number chosen uniformly at random from the numbers not yet assigned.

Suppose the scheduler removes a pair $\{u, v\}$ from P with u the root of the tree containing x, and v in another tree, and that the corresponding link succeeds. Let S and S', respectively, be the sets containing u and v just before $\{u, v\}$ is removed from U. Just before the links forming S' are done, the only numbered nodes are those in S, and u has the largest number. Among the numbers larger than that of u, at least half have rank larger than that of u. When S' is formed, its nodes are numbered uniformly at random from among the unassigned numbers. Given that a number assigned to a node in S' is larger than that of u, it has probability at least $1/2$ of being larger than the rank of u. Since v has largest number among the nodes in S', if the link makes v the parent of u the rank of v is greater than that of u with probability at least $1/2$. The claim (*) follows by induction on the number of steps.

Theorem 4 *Theorem 3 holds for linking by random index if the scheduler restriction holds.*

Proof The theorem follows from parts (iv) and (v) of Lemma 4 in the same way that Theorem 3 follows from parts (iv) and (v) of Lemma 3. □

6 Indirection and helping

The algorithms in Sects. 5.1 and 5.2 require that CAS (and DCAS in 5.1) support testing and updating of storage blocks able to store both the parent and the rank of a node. In this section we present two ways to eliminate the need for blocks to contain multiple fields. Both methods increase the number of steps per operation, but by at most a constant factor. We also discuss how to modify the implementation of the randomized linking algorithm of Sect. 5.2 to eliminate the need for an unrealistic independence assumption in its analysis.

The first method to reduce the block size, proposed by Anderson and Woll [3] is to use *indirection*. Specifically, each node contains only one field, a pointer to a *ledger* that contains the parent and rank of the node. To do a link via a CAS, a process creates a new ledger containing the updated information for the node being linked and then uses a CAS to attempt to replace the old ledger of the node by the new one. A link via a DCAS is similar, except that the process creates two new ledgers and uses a DCAS to replace the old ledgers of the two affected nodes. Parent updates done by splitting are done directly on the appropriate ledgers, without allocating new ones. The ledger method requires a way to allocate ledgers, and care must be taken to avoid the reuse of ledgers. The algorithm of Sect. 5.1 needs at most $3n - 2 + 2p = O(n)$ ledgers, one per initial set plus at most two per successful link plus at most two per process. The algorithm of Sect. 5.2 needs $O(n)$ ledgers with high probability. If ledgers are used to implement randomized linking by rank, the independence assumption needed by the analysis becomes much weaker and quite realistic: the success or failure of a CAS can depend on all inputs to the CAS, in particular the ledger addresses, but not on the contents of the ledgers.

Allocating ledgers efficiently is itself a challenging problem, which Anderson and Woll ignored. One way to do it is to use the concurrent fetch and increment method of Ellen and Woelfel [11]. If ledgers are allocated individually, the number of steps to allocate a ledger is $O(\log p)$. If ledgers are allocated in groups of $O(\log p)$, the amortized time per allocation is $O(1)$ and the number of ledgers used will be $O(n)$ provided $p = O(n/\log n)$. If we are willing to use $O(n + p^2)$ memory, hazard pointers [34] and related techniques [2] can be used.

The second method is *helping*, as described for example in [19]. The idea is to allow processes to complete the tasks of other processes. We number the processes from 1 to p. Each node u has an extra field, *u.process*, which can hold a process number or 0, and is initially 0. Each process has a *descriptor* in which it records a sequence of steps it wants to perform. To update a node, a process writes appropriate instructions into its descriptor and then does a CAS or DCAS to write its process number into the *process* field of the affected node or nodes. Any other process that wants to update a node containing a non-zero process number must first execute the instructions in the corresponding descriptor. When the last instruction is executed, the process number in the affected node or nodes is reset to 0, allowing further updates to the node or nodes. As long as the number of instructions needed to do an update is bounded by a constant, the use of descriptors increases the total work by only a constant factor.

In using helping to link nodes of equal rank, we have to solve the *ABA problem*: A helping process, having completed the instructions in a descriptor, resets the process number in the relevant node to zero, but in the meantime the process being helped has reset the process number to zero, initiated a new update, and set the process number to its own number again. The helping process has no way to detect that a new update has been initiated by the process that initiated the old one. In our application, we can solve the ABA problem by using the monotonicity of ranks. This requires that a CAS be able to update the rank and the process number of a node as an atomic operation. Since ranks are small and in any realistic application $p \ll n$, we think this is a reasonable assumption.

The deterministic algorithm of Sect. 5.1 does links using helping as follows. Each descriptor contains two nodes and a rank. To link root x of rank r to root y, a process, say process i, writes x, y, and r into its descriptor. If y has rank greater than r, it uses a CAS to write i into node x while verifying that the process number of x is 0 and the rank of x is r. If y has rank r, it uses a DCAS to write i into both x and y while verifying that the process numbers of x and y are 0 and the ranks of x and y are r. A process wanting to update a node that finds a non-zero process number i in the node reads the corresponding descriptor. Suppose the descriptor contains nodes x and y and rank r. The process sets $z = y.p$ and does a CAS to set the parent of x to z while verifying that x was a root before the update. It then tests whether y is a root of rank r. If so, it does a CAS to change the rank of y to $r + 1$ and the process number of y to 0 while verifying that the rank and process number of y were r and i before the update.

This method uses a couple of optimizations. It does not reset the process number of a node that becomes a non-root, since no subsequent link will try to change its parent or rank. Instead of making y the parent of x, it makes $y.p$ the parent of x. The reason to do the link this way is that some other process can make y a non-root just before process i does its CAS or DCAS to write i into x, or into x and y. If this happens, $y.p$ will have rank greater than r when x becomes its child, preserving the invariant that ranks strictly increase from child to parent. If this does not happen and the rank of y is r, the helping process adds one to the rank of y and resets the process number of y to 0.

The randomized algorithm of Sect. 5.2 does helping using descriptors containing a node y, a rank r, and a *flag* whose value is null, true, or false. A flag of true indicates that y should become the parent of the root containing the process number of the descriptor; a flag of false indicates that the rank of this node should be changed from r to $r + 1$. If process i wants to link root x of rank r to root y, it writes y and r into its descriptor. If y has rank greater than r, it sets the flag to true; if the rank of y equals r, it flips a fair coin and sets the flag correspondingly. Then it uses a CAS to set the process number of x equal to i while verifying that the rank and process number of x were r and 0 before the update. A process wanting to update a node x that finds a non-zero process number i in x reads the corresponding descriptor. If the flag is true it does a CAS to set the parent of x to y while verifying that x was a root before the update. If the flag is false it does a CAS to set the rank and process number of x to $r + 1$ and 0 while verifying that they were r and i before the update.

The analysis of this method relies on the same independence assumption as the method using ledgers: scheduling decisions are independent of the contents of descriptors. A variant of the method is to set the flag *after* the process number of the descriptor is written into x: the first step of a helping process is to change the flag from null to true or false using the randomized CAS operation mentioned in Sect. 2. If this operation is available, no independence assumption is needed. Algorithms 13 and 14 in Appendix A contain the pseudo-code for an implementation using randomized CAS. Appendix A also contains a detailed line-by-line explanation of the implementation. This implementation of randomized linking by rank satisfies the Anderson-Woll requirement that a randomized algorithm be efficient even if the scheduler knows the outcome of previous random choices. We think, though, that it is reasonable to assume that the scheduler makes its decisions only on the basis of the inputs to the CAS operations, or that it cannot read the private memories of the processes. If either of these assumptions hold, we do not need randomized CAS.

7 Upper bounds

The results of Sects. 5 and 6 give us the following theorem:

Theorem 5 *With any of the three linking methods of Sect. 5 combined with any valid compaction method, the total work is $O(m \log n)$. This bound is worst-case for the deterministic linking method, high-probability for the randomized methods. If randomized linking by rank is implemented as described in Sect. 6, the bound is valid even for an adversarial scheduler.*

Proof The theorem is immediate from the results of Sects. 5 and 6. □

The use of splitting instead of naïve find improves the total work bounds significantly if $p \ll n$. We show this by extending the analysis of sequential splitting [16,45] to one-try and two-try splitting.

We define the *density* d of a set union problem instance to be $m/(np)$ if splitting is two-try, $m/(np^2)$ if splitting is one-try. We shall obtain a bound of $O(m \cdot (\alpha(n, d) + \log(1 + 1/d)))$ on the total work if either kind of splitting is used in combination with any of the three linking methods. The main obstacle we encounter in extending the sequential analysis to the concurrent setting is accounting for unsuccessful CAS operations. Accounting for such operations adds the logarithmic term to the work bound.

We call a problem instance *sparse* if $d < 1$ and *dense* otherwise. The logarithmic term in the work bound dominates only in sparse instances. We start with the analysis of dense instances, which is simpler than that of sparse ones.

We call a child a *zero child* if its rank is the same as that of its parent. Zero children only exist if a randomized linking method is used.

With deterministic linking by rank or linking by random index, ranks are at most $\lg n$. With randomized linking by rank, they are at most $n - 1$, although large ranks occur with exponentially small probability (Lemma 3 part (iii)).

7.1 The dense case

Throughout this section we assume $d \geq 1$.

Lemma 5 *The number of finds is $O(m)$, worst-case unless randomized linking by rank is used, in which case the bound is with high probability.*

Proof There are at most two finds per unite plus at most two per process per root that increases in rank or becomes a child, for a total of $O(m + np) = O(m)$. For randomized linking, this bound follows from part (ii) of Lemmas 3 and 4 and is high-probability for randomized linking by rank, worst case for linking by random index. □

We call a node *low* if its rank is less than d and *high* otherwise. All nodes have rank at most $n - 1$. During a find, a *visit* to a node is an iteration of the find loop in which the node is the value of u. (See the pseudocode in Sect. 4.)

Lemma 6 *The number of visits to low nodes during finds is $O(m)$, worst-case if linking is deterministic, expected if randomized.*

Proof Consider three successive visits to low nodes during a find, to u, v, and w. Let I be the interval of time between the visits of u and w. We claim that at least one of the following events occurs during I: u or v becomes a child, $u.p.r$ increases, or u or v loses an ancestor of the same rank. The number of such events for fixed u and v is $O(d)$: a node only becomes a child once, its parental rank can increase at most d times before it exceeds d and its parent is not low; a node has $O(1)$ ancestors of the same rank in expectation by part (i) of Lemmas 3 and 4. We charge the visit to u to the corresponding event (or any such event if there is more than one). Each event is charged for at most $2p$ visits, at most two per process. (The factor of two comes from the two nodes associated with a visit, the node itself and the next node visited.) Summing over all nodes, we obtain a bound of $O(npd) = O(m)$ on visits to low nodes.

Suppose the claim is false. Then u and v are children when u is visited.

After the CAS following the visit to u, the parent of u has changed; after the CAS following the visit to v, the parent of v has changed. If either u or v is a zero child when u is visited, at least one of them becomes a non-zero child or loses an ancestor of the same rank during I. Thus neither u nor v is a zero child when u is visited. But then the rank of the parent of u increases by the time the CAS after the visit to u finishes, making the claim true. □

Bounding visits to high nodes is more complicated. For each high child x, we measure the progress of compaction by keeping track of an increasing function of the rank of the parent of x, called the *count* of x. We define counts using Ackermann's function. Our formulation is an extension of that of Kozen [30]. We define the *level* $x.a$ of a high node x, and the *index* $x.b$ and *count* $x.c$ of a high child x, as follows:

$$x.a = \min \{k | A_k(x.r) > x.p.r\};$$
$$x.b = \max \{i | A_{x.a}(i) \leq x.p.r\};$$
$$x.c = x.r \cdot x.a + x.b.$$

We bound the range of levels, indices, and counts by using the properties of Ackermann's function:

Lemma 7 *If x is a high node, $0 \leq x.a \leq \alpha(n, d)$ and $x.a = 0$ if and only if $x.r = x.p.r$. If x is a high child, $0 \leq x.b < x.r$ and $0 \leq x.c < (\alpha(n, d) + 1)x.r$. The values of $x.a$ and $x.c$ never decrease, and if $x.a$ or $x.b$ increases, $x.c$ increases by at least as much.*

Proof Since $A_0(x.r) = x.r + 1, x.a = 0$ if and only if $x.r = x.p.r$, and $x.a \geq 0$ if it is defined. If x is a high node, $A_{\alpha(n,d)}(x.r) \geq A_{\alpha(n,d)}(\lfloor d \rfloor) > n > x.p.r$. Thus $x.a$ is defined and is at most $\alpha(n, d)$. (Here for randomized linking by rank we use the assumption that all ranks are less than n.) Suppose x is a high child. If $x.a = 0$, $x.b = x.r - 1$ since $x.r \geq d \geq 1$. If $x.a > 0$, $A_{x.a}(0) = A_{x.a-1}(1) \leq A_{x.a}(x.r) \leq x.p.r$, so $x.b$ is defined. Since $A_{x.a}(x.r) > x.p.r$, $x.b < x.r$. The bounds on $x.c$ follow from those on $x.a$ and $x.b$. While x is a root, $x.a = 0$. Once x is a child, $x.r$ is constant and $x.p.r$ cannot decrease, so $x.a$ cannot decrease by Lemma 1. While $x.a$ is constant, $x.b$ cannot decrease for the same reason. If $x.a$ increases by one, $x.b$ can decrease by at most $x.r - 1$, resulting in an increase of at least one in $x.c$. If $x.a$ increases by at least k, $x.c$ increases by at least $(k - 1)x.r + 1$. □

Lemma 8 *The sum of the counts of all high children is $O(n\alpha(n, d))$, worst-case unless linking is randomized by rank, in which case the bound is high-probability.*

Proof By Lemma 7, the sum of the counts of high children is $O(\alpha(n, d))$ times the sum of the ranks of all nodes. By Lemmas 2 and 4, the sum of ranks is less than n for deterministic linking by rank and linking by random index. For randomized linking by rank, it is $O(n)$ with high probability by Lemma 3. □

The following lemma is the key to the analysis of splitting.

Lemma 9 *Consider a time t at which u is a high child whose parent v is also a (high) child. Let w the parent of v at time t, and let $u.a$, $v.a$, and $w.r$ be the levels of u and v and the rank of w at time t, respectively. Suppose that at time t or later the parent of u changes from v to a node x of rank at least $w.r$. If $v.a > u.a$, the parent change increases $u.a$ and $u.c$ by at least $v.a - u.a$; if $v.a = u.a$, the parent change increases $u.c$ by at least 1 or causes u to lose an ancestor of the same rank.*

Proof Let $u.r$ and $v.r$ be the ranks of u and v at time t, respectively. Let $x.r$ be the rank of x when it becomes the parent of u. Since $A_{v.a-1}(u.r) < A_{v.a-1}(v.r) \le w.r \le x.r$, the level of u after the parent change is at least $v.a$. If $v.a > u.a$, the parent change increases the level and hence the count of x by at least $v.a - u.a$ by Lemma 7. Suppose $v.a = u.a$. If $u.a = 0$, the parent change causes u to lose v as an ancestor. Suppose $u.a > 0$. Since $A_{u.a}(u.b + 1) = A_{u.a-1}(A_{u.a}(u.i)) \le A_{u.a-1}(v.r) \le w.r \le x.r$, the parent change increases either the level or the index of u and hence increases the count of u. □

To count visits to high nodes, we use a credit argument. One credit pays for one high-node visit. We allocate a certain number of credits to each find when it starts, and additional credits when high nodes increase in count or lose ancestors of the same rank. We show via a *credit invariant* that these credits suffice to pay for all the high-node visits. A bound on the total number of credits gives a bound on the number of high-node visits.

We begin by analyzing two-try splitting: even though it is more complicated than one-try splitting, its analysis is simpler. We call a find *active* while it is being executed. When a find starts, we allocate it $\alpha(n, d) + 1$ credits. When the count of a high child increases by k, we allocate $2k$ credits to each active find, for a total of at most $2pk$. When a high child loses an ancestor of the same rank, we allocate one credit to each active find, for a total of at most p.

Lemma 10 *With two-try splitting, the number of allocated credits is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average-case if randomized.*

Proof By Lemma 5, the number of credits allocated to finds when they start is $O(m\alpha(n, d))$. By Lemma 8, the number of credits allocated to finds as a result of increases in count is $O(np\alpha(n, d)) = O(m\alpha(n, d))$. By Lemmas 3 and 4, the expected number of credits allocated to finds as a result of nodes losing ancestors of the same rank is $O(np) = O(m)$. □

Lemma 11 *With two-try splitting, just after a high node u is visited by a find, the find has at least $u.a$ credits.*

Proof We prove the lemma by induction on the number of high-node visits done by a find. When the find starts, it has $\alpha(n, d) + 1$ credits. The first visit costs one, leaving $\alpha(n, d)$, which is enough to make the lemma true just after this visit. Suppose the lemma holds just after u is visited, and let v be the next node visited. We denote by unprimed and primed variables their values just after the visit to u and just before the visit to v, respectively. The lemma holds after the visit to v provided that the find accrues at least $v.a' - u.a + 1$ credits between the visits to u and v. To show that this happens, we need the following crucial inequality, which follows from Lemma 9:

(*) $u.a' \ge v.a$

To prove (*), we refer to the implementation of two-try splitting. Let t be the first time $u.p = v$. Time t is after the visit to u, since $u.p$ changes between the first and second times that the find sets its variable v after the visit to u, as a result of the first CAS during the visit to u succeeding or failing. Let w be the parent of v at time t. Consider the change to $u.p$ resulting from the second CAS after the visit to u. This change satisfies the hypothesis of Lemma 8, since the new parent of u must have been the parent of v at time t or later. By Lemma 9, just after this change to $u.p$, the level of u is at least the level of v at time t. Since levels are non-decreasing, (*) holds.

Between the visits to u and v, the find accrues at least $2(u.a' - u.a + v.a' - v.a) = (v.a' - u.a) + (u.a' - u.a) + (v.a' - v.a) + (u.a' - v.a)$ credits as a result of level increases. Each of the last three terms is non-negative, the last one by (*). Thus the find accrues at least $v.a' - u.a + 1$ credits between the visits, unless the levels of u and v are equal and unchanging between the visits. Suppose the levels of u and v are equal and unchanging between the visits. By Lemma 9, the find accrues at least one credit when the parent of u changes from v. □

Lemma 12 *With two-try splitting, the number of visits to high nodes is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average-case if randomized.*

Proof The lemma is immediate from Lemmas 10 and 11. □

Now we extend the analysis to one-try splitting. The proof of Lemma 11 fails for one-try splitting, because a CAS done by one process, say process 1, can fail as a result of a successful CAS done by another process, say process 2, that sets its value of v *before* process 1's most recent high-node visit. That is, time t in the proof of Lemma 9 can precede the visit. This invalidates the use of Lemma 9 in the proof.

To overcome this problem, we allocate additional credits to node count increases, and we allow active finds to shift some of their credits to the other active finds. Specifically, when a find starts, we allocate it $\alpha(n, d) + 1$ *normal* credits. When a high child loses an ancestor of the same rank, we allocate one normal credit to each active find. When the count of a high node increases by k, we allocate $2k$ normal credits and $2k(p-1)$ *extra* credits to each active find. When a CAS in a find succeeds, we shift a $1/(p-1)$ fraction of the find's extra credits to each other active find. Shifted extra credits become normal; that is, we shift a credit at most once.

Lemma 13 *With one-try splitting, the number of allocated credits is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average-case if randomized.*

Proof The bound holds for normal credits by the proof of Lemma 10. By Lemma 8, the number of extra credits allocated to finds as a result of increases in count is $O(np^2\alpha(n, d)) = O(m\alpha(n, d))$ since $d = m/(np^2)$. □

Lemma 14 *With one-try splitting, just after a high node u is visited by a find, the find has at least u.a normal credits.*

Proof The proof is an extension of that of Lemma 11. Consider a find, say find 1. The credits allocated to the find when it starts make the lemma true just after its first high-node visit. Suppose the lemma holds just after find 1 visits u, and let v be the next node it visits. We consider three cases. If the CAS during the visit of find 1 to u succeeds, the lemma holds just after the visit to v by an argument like that in the proof of Lemma 11. (This case does not use shifted credits.) Suppose this CAS fails, because a CAS done by another find, say find 2, changes $u.p$ from v to another value. Let t be the last time that find 2 set its variable v before its successful CAS. If t is after find 1 visits u, the lemma holds just after the visit to v by an argument like that in the proof of Lemma 11, again without the use of shifted credits.

The third, new case is if t precedes the visit of find 1 to u. Let t', t'', and t''' be the times find 1 visits u, find 2 does its CAS, and find 1 visits v, respectively. We denote by unprimed, primed, double-primed, and triple-primed values their values at times t, t', t'', and t''', respectively. Applying Lemma 8 to time t and the successful CAS of find 2 gives $u.a'' \geq$ v.a; and, if $u.a \leq v.a$, the count of u increases by at least 1 or u loses an ancestor of the same rank when find 2 does its CAS.

At time t', find 1 has at least $u.a'$ normal credits by the induction hypothesis. Between times t' and t''', it accrues at least $2(u.a''' - u.a' + v.a''' - v.a')$ normal credits. Between times t and t'', find 2 accrues at least $2(p-1)(u.a'' - u.a + v.a'' - v.a) \geq 2(p-1)(u.a' - u.a + v.a' - v.a)$ extra credits, of which at least $2(u.a' - u.a + v.a' - v.a)$ are shifted to find 1 and become normal at time t'': find 1 is active at t'' since its CAS fails as a result of the CAS by find 2 succeeding. Thus between t' and t''' find 1 accrues at least $2(u.a''' - u.a + v.a''' - v.a) \geq (v.a''' - u.a') + (u.a''' - u.a) + (v.a''' - v.a) + (u.a'' - v.a)$ normal credits. Since $u.a'' \geq v.a$, this is at least $v.a''' - u.a' + 1$, enough to make the lemma true for the visit to v, unless u and v have equal and unchanging levels from t to t''', in which case find 1 accrues a normal credit when find 2 does its CAS. □

Lemma 15 *With one-try splitting, the number of visits to high nodes is $O(m\alpha(n, d))$, worst-case if linking is deterministic, average if randomized.*

Proof The lemma is immediate from Lemmas 13 and 14. □

7.2 The sparse case

In this section we modify the analysis of Sect. 7.1 to handle sparse instances. Throughout this section we assume $d < 1$. We need to change the definition of low and high nodes, add an additional node type, *middle*, and (for the purpose of the analysis only) redefine the ranks of nodes.

Let $l = \lg(1 + 1/d)$. Since $d < 1$, $l > 1$. A node is *low* if its rank is less than l and its height is less than cl, where c is the constant in part (vi) of Lemmas 3 and 4; *middle* if its rank is less than l but its height is at least cl, and *high* if its rank is at least l. Middle nodes can exist only if linking is randomized.

Lemma 16 *The number of non-low nodes is at most $2nd$, as is the sum of the ranks of such nodes. This bound is worst-case if linking is deterministic, average-case if randomized.*

Proof By part (vi) of Lemmas 3 and 4, the expected number of middle nodes is at most $n/2^l \leq n/2^{\lg(1/d)} = nd$ if linking is randomized. (It is zero if not.). By Lemma 2 or part (iii) of Lemma 3 or 4 depending on the linking method, the number of high nodes is also at most $n/2^l \leq nd$, worst-case if linking is deterministic or by randomized index, average-case if by randomized rank. The bound on the sum of ranks follows from the node bound by the argument in the proof of Lemma 2 if linking is deterministic, by that in the proof of part (ii) of Lemma 3 or 4 if randomized. □

Lemma 17 *The number of finds that visit at least one non-low node is $O(m)$, worst-case if linking is deterministic, average-case if randomized.*

Proof Consider the finds during unites that visit at least one non-low node. At most two per unite also visit a low node.

Of those that visit only non-low nodes, there are at most two per unite plus at most $2p$ per non-low node that becomes a child or has a rank increase, two per process doing a unite while the event in question takes place. By Lemma 16, the number of such finds is $O(npd) = O(m)$. □

Lemma 18 *The number of visits to low nodes is $O(ml)$, worst-case.*

Proof The analysis of node visits in the proof of Theorem 2 restricted to nodes of rank less than l and height less than cl gives a bound of $O(l)$ low-node visits for each find and unite. □

Lemma 19 *If linking is randomized, the expected number of visits to middle nodes is $O(ml)$.*

Proof By Lemma 17, the expected number of finds that visit middle nodes is $O(m)$. During such a find, each visit to a middle node except the last two is followed by a middle node losing a child of the same rank or the parent of a middle node x increasing in rank. The latter can only happen l times before x has a parent that is not a middle node; subsequently, x can only be the last middle node visited during a find. We charge each visit to a middle node other than the last two of a find to the corresponding event. The charge per event is at most p, and the expected number of events is at most ndl rank increases and $O(nd)$ losses of same-rank ancestors, the latter by part (i) of Lemma 3 or 4. Such events account for $O(npdl) = O(ml)$ visits. Adding the last two per find gives the lemma. □

To count visits to high nodes, we define the *effective rank* of a high node x to be $x.er = x.r - l + 1$. We define levels of high nodes and indexes and counts of high children, using effective ranks in place of ranks. Since the effective rank of a high node is at least one, levels of high nodes and indices and counts of high children are well-defined. We allocate credits exactly as in Sect. 7.1. Lemmas 7 and 9 remain true. By Lemma 16, the sum of counts of high children is $O(nd\alpha(n,d))$, worst-case if linking is deterministic, high-probability if randomized. We allocate credits exactly as in Sect. 5.1. Lemmas 11 and 14 remain true. If splitting is two-try, the number of allocated credits is $O((m + ndp)\alpha(n,d)) = O(m\alpha(n,d))$ since $d = m/(np)$; if splitting is one-try, it is $O((m + ndp^2)\alpha(n,d)) = O(m\alpha(n,d))$ since $d = m/(np^2)$. We conclude that Lemmas 10, 12, 13, and 15 hold in the sparse case (with the new definition of a high node).

7.3 The total work bound

Combining the results of Sects. 7.1 and 7.2, we obtain the following theorem:

Theorem 6 *With any of the three linking methods of Sect. 5 and either one-try or two-try splitting, the total work is $O(m(\alpha(n,d)+\log(1+1/d)))$, worst-case if linking is deterministic, average-case if randomized, where $d = m/(np^2)$ if splitting is one-try, $d = m/(np)$ if splitting is two-try.*

Proof The theorem follows from Lemmas 6, 12, 15, 18, and 19. □

8 Lower bounds

In this section, we derive lower bounds on the worst-case and amortized efficiency of set union algorithms. In the first subsection, we prove lower bounds on the work efficiency of the algorithms described in this paper by explicitly providing worst-case executions—both the operations and the adversarial schedules. At a high level, our executions are constructed by the following observations and steps. For each algorithm, we describe operations that build a tree of logarithmic height using *unite* operations. We observe that *shadowing schedules* in which all processes are scheduled in lock-step while performing the same expensive *find* operations result in worst-case behavior. We apply a shadowing schedule to processes performing a *find* on the deepest node in the aforementioned tree to prove that the logarithmic term in our upper bounds is tight. Then, we combine the idea of shadowing schedules with previous sequential lower bounds of Tarjan et al. and Fredman et al. [14,45] to show that the inverse-Ackermann term in our upper bounds is tight. Our algorithmic lower bounds section proves that our amortized upper bound analyses are tight when find operations are done with two-try splitting.

In the second subsection, we show general lower bounds that apply to the concurrent set union problem. First, we prove that, in the worst-case, any concurrent set-union algorithm must do at least $\Omega(\log \min\{n, p\})$ work in expectation for a single operation. When $p = n^{\omega(\frac{1}{\log\log n})}$, this lower bound is stronger than the sequential lower bound of $\Omega\left(\frac{\log n}{\log\log n}\right)$ given by Fredman and Saks [14] in the cell probe model. It also shows a separation in work complexity between the sequential and concurrent versions of the set-union problem, since Blum [5] presented an algorithm that does at most $O\left(\frac{\log n}{\log\log n}\right)$ work per operation in the sequential setting. Furthermore, whenever $\log p = \Theta(\log n)$, i.e. when $p = n^{\epsilon}$, this lower bound establishes that randomized linking with any form of compaction yields an algorithm with optimal expected work per operation. Finally, we generalize the worst-case lower bound using shadowing schedules to show that our algorithm obtained by combining randomized linking with two-try splitting is optimal amongst a class of *symmetric algorithms* that includes all known algorithms for the concurrent disjoint set union problem.

8.1 Algorithmic lower bounds

In order to prove the tightness of the inverse-Ackermann term in our upper bounds, we recall a sequential cell probe lower bound on the set union problem given by Fredman and Saks.

Lemma 20 ([14]) *Let $\mathcal{A}$ be any randomized algorithm that solves the sequential set union problem. For any fixed number of nodes n, and any $M \geq n$, there is a sequence of operations σ_M, that makes $\mathcal{A}$ perform $\Omega(M\alpha(n, M/n))$ expected work.*

We use Lemma 20 to establish a concurrent lower bound.

Lemma 21 *Let $\mathcal{A}$ be any of the algorithms we have described for concurrent set-union. There is some sequence of m operations using p processes on n nodes that requires $\Omega\left(m \cdot \alpha\left(n, \frac{m}{np}\right)\right)$ work in expectation.*

Proof Any concurrent algorithm is also a sequential algorithm if it is run by a single process. So, for any given $M \geq n$, we can take a worst-case sequence σ_M of operations from Lemma 20. That is, a single process running the sequence of operations σ_M will perform $\Omega(M\alpha(n, M/n))$ work in expectation. In the remainder of the proof, we use shadowing schedules, in which processes run in lock-step with each other and thereby do not gain locally from any compaction attempts of other processes, to get the lower bound.

We consider two cases for m:

Case 1: If $m \geq np$, then we choose $M = m/p \geq n$. If each of the p processes runs σ_M and is scheduled in lock-step (so that the processes all walk up find sequences together and do not benefit from each other's compaction attempts), then the total number of operations is $pM = m$ and the total amount of work is $\Omega(pM\alpha(n, M/n)) = \Omega\left(m\alpha\left(n, \frac{m}{np}\right)\right)$.

Case 2: If $m < np$, we choose $M = n$. Then, σ_M performed by a single process takes $\Omega(n\alpha(n, 1))$ expected work. We observe that $m/n < p$ and assign m/n processes the operation sequence σ_M, thus assigning m operations. If the processes are scheduled in lock-step, the total expected work performed by them is $\Omega(m/n \cdot n\alpha(n, 1)) = \Omega\left(m\alpha\left(n, \frac{m}{np}\right)\right)$.

We can also show that the logarithmic term $\log(\frac{np}{m} + 1)$ is an amortized lower bound for all our algorithms. The schedule that builds binomial trees with a single process and makes all the processes shadow each other up the longest branch of these trees yields the lower bound.

Lemma 22 *For randomized linking by rank and linking by DCAS, regardless of what type of compaction is used in find operations, and for any positive integer $k \in [1, n]$, there is a sequence of $k - 1$ unite operations that will build a tree with k nodes with height $\Omega(\log k)$.*

Proof For simplicity, we initially assume that k is a power of 2. Let B_j be the binomial tree of height j. All the nodes are initially in singleton trees, i.e. B_0 trees. We proceed in $\lg k$ rounds. In round r, we start with $\frac{k}{2^{r-1}}$ trees of type B_{r-1}, and simply unite their roots pairwise. After $\lg k$ rounds we end up with a single tree of type $B_{\lg k}$. If k were not a power of 2, we perform the above procedure with the largest power of 2 less than k and simply unite the remaining nodes to the root of the main tree. □

A slightly more complex construction allows us to prove a similar lemma for linking by index.

Lemma 23 *For randomized linking by index, regardless of what type of compaction is used in find operations, and for any positive integer $k \in [1, n]$, there is a sequence of $k - 1$ unite operations by a single process that will build a tree with k nodes in which the depth of a uniformly randomly picked node is $\Omega(\log k)$ in expectation.*

Proof The proof is constructive. The construction of these trees is inspired by *binomial trees*, and is done in multiple *rounds* such that each round fully finishes before the next round starts. Without loss of generality let k be a power of 2, as otherwise we could just use the greatest power of 2 less than k in the following construction. Initially, we let the nodes be in singleton trees $T_{1,1}, \ldots, T_{k,1}$. In each round we will combine pairs of trees, and each tree T will have a designated node $\nu(T)$. In the initial trees the designated node is the only node. In the first round we combine pairs of trees by performing

$$unite(\nu(T_{1,1}), \nu(T_{2,1})), unite(\nu(T_{3,1}), \nu(T_{4,1})), \ldots, unite(\nu(T_{k-1,1}), \nu(T_{k,1}))$$

to produce tree $T_{1,2}, \ldots, T_{k/2,2}$. The designated node $\nu(T_{i,2})$ is chosen to be one of the designated nodes of the subtrees that formed $T_{i,2}$. We call this process of picking the new designated nodes as a subset of the old ones *refining*. The subsequent rounds are done similarly by combining pairs of trees from the previous round and refining designated nodes, until only the tree $T_{1,\lg k}$ remains.

We now make the following observations about this process:

(1) All trees $T_{i,r}$ of a given round r have the same number of nodes 2^r.
(2) A designated node always has depth at most 2. (This follows from the way *find* does compactions.)

(3) A node of depth δ in any of the trees $T_{i,r}$ has at most $\left(\frac{1}{2}\right)^{\delta} \cdot |T_{i,r}|$ successors.

The links in the rounds raise the depth of half the nodes due to (1), and the compactions in the *find* operations of the rounds reduce the average depth of a node in the forest by at most $\frac{1}{4}$ due to (2) and (3). Thus, each round increases the average depth of a node in the forest by at least $\frac{1}{2} - \frac{1}{4} = \frac{1}{4}$. Since there are $\log(k)$ rounds, the proof is complete. □

Combining the previous lemmas yields our best algorithmic lower bound result.

Lemma 24 *Let $\mathcal{A}$ be a concurrent disjoint set union algorithm obtained by combining linking by DCAS, randomized linking by rank, or linking by random index with find with no compaction, one-try splitting, or two-try splitting. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m \log\left(\frac{np}{m}+1\right)\right)$ work. The bound holds in expectation for the linking by random index algorithm even under the independence assumption.*

Proof We prove the theorem for linking by the DCAS and randomized linking by rank algorithms first. The lower bound is non-trivial only when $m/p < n$. In this case, we describe a particular sequence of operations and schedule that performs the requisite work. Divide the nodes into m/p groups of size $n/(m/p) = np/m$. Lemma 22 allows us to link each group of nodes into a tree of height $\Omega\left(m \log\left(\frac{np}{m}+1\right)\right)$. For each such tree, perform *find*(x) on the deepest node x of that tree simultaneously with each of the processes. Now consider the schedule in which processes shadow each other in all the finds. In this schedule, each process does $\Omega\left(m \log\left(\frac{np}{m}+1\right)\right)$ work per find, and one find per group. The total number of operations is $m/p \times p = m$, and the total amount of work is $\Omega\left(m \log\left(\frac{np}{m}+1\right)\right)$.

In the case of the linking by random index algorithm under the independence assumption, we modify the above argument by replacing the use of Lemma 22 with Lemma 23, and performing *find*(x) on a uniformly randomly picked node x in the tree. □

Combining the previous lemmas yields our best algorithmic lower bound result.

Theorem 7 *Let $\mathcal{A}$ be a concurrent disjoint set union algorithm obtained by combining linking by DCAS, randomized linking by rank, or linking by random index with find with no compaction, one-try splitting, or two-try splitting. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m\left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n,\frac{m}{np}\right)\right)\right)$ work. The bound holds in expectation for the linking by random index algorithm even under the independence assumption.*

Proof Combine the results of Lemmas 21 and 24. □

As the final algorithmic lower bound, we prove that the independence assumption we have been using to analyze the linking by random index algorithm is indeed necessary. In particular, we present a super-logarithmic work lower bound for the algorithm if the independence assumption does not hold.

Lemma 25 *Concurrent set union via the linking by random index algorithm performs $\Omega(m\sqrt{p})$ expected work to do $m = n\sqrt{p}$ operations if $\sqrt{p} \leq n$, regardless of which compaction rule the find operations use.*

Proof We will show an explicit example. Assume $\sqrt{p} < n$, and pick a set S of $\sqrt{p}$ nodes. Let $p/2$ processes attempt to do *unite*(x, y) where each pair of x, y in S is tried by at least one process. The scheduler can wait to see the outcomes of the node comparisons and decide to schedule the processes so that the nodes of S get linked into a linear path of length $\sqrt{p}$. If the remaining $p/2$ processes all perform *find*(x) where x is chosen randomly from S, and are scheduled in lock-step, they will perform, in expectation, $\Omega(\sqrt{p})$ work each, since the expected depth of x is $\sqrt{p}/2$.

Performing the same process on each of the $\left\lfloor \frac{n}{\sqrt{p}} \right\rfloor$ sets of nodes leads to $\Omega(np)$ work to do $m = n\sqrt{p}$ operations. The average operation takes $\Omega(\sqrt{p})$ work. □

8.2 Problem lower bounds

In this subsection, we prove that any concurrent set-union algorithm must do $\Omega(\log \min\{n, p\})$ work for a single operation in the worst case. Furthermore, we build on the worst-case lower bound to show an $\Omega(\log(np/m+1))$ amortized work lower bound for all *symmetric algorithms*, where we say an algorithm is symmetric if:

(1) The algorithm's code for the *unite* and *find* procedures does not use process ids.
(2) The algorithm does not use the return values of CAS operations.

All our algorithms and all algorithms known to us can be made symmetric without effecting the upper bound analyses of the algorithms. For instance, this can be done if we assume that all CAS operations return false. This does not effect the correctness of our algorithms since we only use the return value of a CAS operation in the *unite* operation to determine if a link has been successful. However, if we do not perform this check atomically, our algorithms remain correct, since a process that has successfully united two trees together will realize this shortly afterwards when its u and v pointers meet at the root of the united tree. The work efficiency analysis increases by at most a factor of two, because we can always imagine the case where processes work in pairs (p, q), and each pair performs operations together and are always scheduled in lock-step. In this case, if p ever performs a successful link CAS$(u.p, u, v)$ and returns, then q's attempt to perform the same link will fail; thus q will only return after it traverses the whole tree and discovers that some other process has already finished the link it wanted to do. The modification we propose to symmetrize the algorithm will simply make p do the same work as q.

Our lower bounds make use of a result on a problem called "wake-up". The *wake-up* problem on k processes asks for a wait-free algorithm with two properties: (i) every process returns a boolean value and at least one process returns true, and (ii) a process may return true only if every process has already executed at least one step. The following lemma is a lower bound on the complexity of wake-up that follows straightfowardly from Jayanti's lower bound in [26].

Lemma 26 ([25,26]) *For any k process wake-up algorithm that uses variables supporting read, write, and CAS, there is a schedule in which some process performs $\Omega(\log k)$ steps in expectation.*

We solve wake-up via set-union to get our lower bounds below.

Lemma 27 *The reduction (below) solves the wake-up problem for k processes using a disjoint set union instance with $k + 1$ nodes, in which each process executes one unite and two find operations.*

Proof Let $q_1, \ldots, q_k$ be the k processes and let the nodes be labelled $0, \ldots, k$. The reduction below correctly solves wake-up because:

```
1: procedure WakeUp
2:     unite(j − 1, j); x ← find(0); y ← find(k); return x = y
```

Reduction : q_j's code in wake-up solution.

(i) the last process to complete *unite* finds that the leaders of nodes 0 and k are the same and thus returns true, and (ii) no process returns true before all processes have completed *unite*, since no leader of k can be the same as any leader of 0 until they are in the same set, i.e. until the last of the *unite* operations is linearized. □

Theorem 8 *Let $\mathcal{A}$ be a linearizable wait-free concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega(\log \min\{n, p\})$ work in expectation.*

Proof Instantiate Lemma 27 with $k = \min\{n - 1, p\}$. The most expensive of the three set union operations of the process that performs the most work in the adversarial schedule of Lemma 26 must do $\Omega(\log \min\{n, p\})$ expected work. □

Corollary 2 *The disjoint set union algorithm obtained by combining randomized linking with any form of find described in this paper gives an algorithm with optimal worst-case work per operation up to constant factors when $\log p = \Theta(\log n)$.*

Remark 1 Theorem 2 shows that our set union algorithms with randomized linking have optimal work per operation when $p = n^{\varepsilon}$ for constant ε.

Remark 2 Theorem 8 establishes a separation in worst-case work complexity between sequential and concurrent set-union when $p = n^{\omega(\frac{1}{\log\log n})}$ since Blum's sequential set-union algorithm has a worst-case work complexity of $O(\frac{\log n}{\log\log n})$ [5].

Lemma 28 *Let $\mathcal{A}$ be a linearizable wait-free symmetric concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega(m \log(np/m + 1))$ work.*

Proof Divide the n nodes into $g = \frac{m}{p}$ groups of size $k + 1$, where $k = \frac{np}{8m}$ (disregard any additional nodes); label the groups $G_1, \ldots, G_g$. Note that $m \geq p$ and $m \geq \frac{n}{2}$, so $k \leq \frac{p}{4}$. We divide $\frac{p}{2}$ (out of the p) processes into two sets $A = \{q_1, \ldots, q_k\}$ and $B = \{q_{k+1}, \ldots, q_{p/2}\}$. Note that $|B| \geq \frac{p}{4}$ and $|A \cup B| = \frac{p}{2}$.

Consider running the wake-up algorithm of Lemma 27 on the k processes in A using the $k+1$ nodes in G_1. By Lemma 26 there is a schedule σ_1 in which some process q_i performs $\Omega(\log k)$ steps. Assign to each process in B the same sequence of three set union operations that q_i performs, and define schedule σ_1' to be the schedule σ_1, with the processes of B interleaved in to run in lockstep with q_i. In this schedule, the processes $q_1, \ldots, q_{p/2}$ perform $\Omega(p \log k)$ work to do p set union operations. Repeating this procedure on each group G_j produces schedules σ_j', each of which performs $\Omega(p \log k)$ work. Therefore, in the concatenated schedule of $\sigma_1'\sigma_2' \cdots \sigma_g'$, the processes $q_1, \ldots, q_{p/2}$ perform a total of $\Omega(gp \log(k+1)) = \Omega(m \log(np/m+1))$ work to do a total of $gp = m$ operations. □

Theorem 9 *Let $\mathcal{A}$ be any linearizable wait-free symmetric concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m\left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n, \frac{m}{np}\right)\right)\right)$ work in expectation.*

Proof Combine the results of Lemma 21, whose argument applies to all symmetric algorithms, with Lemma 28. □

Remark 3 Theorem 9 shows that the set union algorithm obtained by combining randomized linking with two-try splitting has optimal amortized work efficiency amongst all symmetric algorithms (up to a constant factor).

The ideas behind our collection of upper and lower bounds lead us to make the following conjecture about the expected work complexity of concurrent disjoint set union.

Conjecture 1 The expected work complexity of the concurrent set union problem is

$$\Theta\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n, \frac{m}{np}\right)\right)\right).$$

In light of Theorem 6, which shows that randomized linking with two-try splitting satisfies the conjectured upper bound, a refutation of Conjecture 1 would imply a more efficient algorithm than randomized linking with two-try splitting. On the other hand, a demonstration of the conjecture would involve proving a universal lower bound; namely, showing that Theorem 9 holds for *all* algorithms (as opposed to only symmetric ones). While this paper proves a universal lower bound on the worst-case complexity of a single operation, it does not prove any universal lower bounds on the total work complexity of m operations. The only such lower bound that is known for the problem is exponentially weaker than the conjectured one. We state this bound, by Jayanti et al., below.

Theorem 10 ([29]) *Let $\mathcal{A}$ be any linearizable wait-free concurrent disjoint set union algorithm using read, write, and CAS. There is a schedule of m operations on n nodes by p processes that forces $\mathcal{A}$ to perform $\Omega\left(m\left(\log\log\left(\frac{np}{m}+1\right)+\alpha\left(n, \frac{m}{n}\right)\right)\right)$ work in expectation.*

9 Remarks and open problems

We have presented three linking methods and two splitting methods for concurrent disjoint set union. With any of the linking methods, with or without compaction, the number of steps per operation is $O(\log n)$, worst-case if linking is deterministic, high-probability if randomized. With any of the linking methods and either of the splitting methods, the total work is $O(m(\alpha(n, d)+\log(1+1/d)))$, worst-case if linking is deterministic, average-case if randomized, where the problem density d is $m/(np^2)$ if splitting is one-try, $m/(np)$ if splitting is two-try. No matter what the density, the cost of concurrency is at most a factor of $\log p$, making our algorithms truly scalable. The proofs of the bounds for linking by random index require assuming that the scheduler is non-adversarial, as discussed in Sect. 5.3. The bounds differ for the two splitting methods only for a narrow range of densities: if $m/n = O(1)$ or $m/n = \Omega(p^2)$, the bounds are the same; if $m/n = \omega(1)$ and $m/n = o(p^2)$, the bounds differ by a factor of at most $\log p$.

The $O(\log n)$ step bound is tight for all our algorithms. The work bounds for splitting are almost tight: any symmetric algorithm (as defined in Sect. 8) has a work bound of $\Omega\left(m \cdot \left(\log\left(\frac{np}{m}+1\right)+\alpha\left(n, \frac{m}{np}\right)\right)\right)$. We conjecture that the same lower bound can be shown for asymmetric algorithms also (Conjecture 1), but leave the proof or refutation of this statement as an open problem.

Our results leave open the question of whether there is an efficient *deterministic* algorithm that uses only CAS: our deterministic algorithm uses DCAS. Recently we have developed a surprisingly simple algorithm that answers this question positively. The algorithm combines two ideas: the use of *latent links*, which represent unites started but not finished, and *deterministic coin tossing* [9], which provides a deterministic way to break ties. The worst-case and amortized time bounds for finds are the same as those in the present paper; the bounds for unites are larger by a factor of $\lg^* n$, reducible to $\lg^* p$. We shall describe this result in a forthcoming paper.

In some applications of disjoint set union, such as computing flow graph information [42,43] each set has a name or some other associated value, such as the number of elements in the set. We can extend the compressed tree data structure to support set values by storing these in the set roots. In the sequential setting, it is easy to update set value information in $O(1)$ time during a link. But in the concurrent setting, updating the value in the new root during a link requires a DCAS or some more-complicated implementation using CAS. Updating root values using DCAS invalidates our analysis. Consider n singleton sets $\{1\}, \{2\}, \ldots, \{n\}$. Suppose $p = n$, and unite$(1, n)$, unite$(2, n)$,..., unite$(n-1, n)$ are performed concurrently using linking by rank via DCAS. Assume the tie-breaking total order is numeric. At most one link will succeed initially, say the link of 1 and n. After this link, all nodes except n will still have rank 0, and n will have rank 1. The algorithm of Sect. 5.1 does all the remaining links concurrently using CAS, since none affects node n. But if each such link needs to update the value in node n, the remaining links must be done one-at-a-time, resulting in overall work $\Omega(np)$.

We think this problem can be overcome, and that the concurrent set union problem with set values can be solved in a work bound that is quasi-linear in m and poly-logarithmic in p. But doing so may well require relaxing the linearization requirement: instead of continuing to try to link each node i and n, suppose the algorithm does a different set of links to reduce the contention. For example, the algorithm could link roots in pairs, then the remaining roots in pairs, and so on. The set resulting from all the links would be the same, but the intermediate sets would not correspond to any linearization of the original unites. Even though it violates linearization, such an algorithm might suffice in many if not all applications.

An algorithm of this kind needs a mechanism to restructure the links. We think some sort of binary tree structure, like the one used by Ellen and Woelfel [11] in their fetch-and-increment algorithm but more dynamic, may suffice. We leave open the development of this idea or some other idea to solve the problem of concurrent sets with values.

Although our results are for a shared memory model, we think they will fruitfully extend to a distributed-memory, message-passing setting.

Our work is theoretical, but others [1,10,17] have implemented some of our algorithms on CPUs and GPUs and experimented with them. On many realistic data sets, our algorithms run as fast or faster than all others.

Acknowledgements We thank Prasad Jayanti for helpful discussions. We also thank Philipp Woelfel, the anonymous PODC 2016 and 2019 reviewers, and the journal reviewers for their helpful comments.

A: Our algorithm with randomized compare-and-swap

There are several ways of implementing randomized linking by rank using randomized CAS. We present what we think is the clearest and most concise implementation below. Our implementation uses a compressed tree structure, just as do the other sequential and concurrent algorithms we have presented, but with a small modification discussed below. The forest contains one rooted tree per set, whose nodes are the elements of the set and whose root is the set leader. We assume that the nodes have indices 1 through n and thus can be compared via '$<$'. Each node x has a field $x.p$ to store the address of a *parent* node, a field $x.r$ to store a non-negative integer *rank*, and a field $x.b$ to store a single *root-bit* signifying whether or not x is the root of its tree. Initially, $x.p = x$, $x.r = 0$, and $x.b = 1$. The rank is never more than n, and thus needs only $\lceil \log n \rceil$ bits of storage. We shall assume that all three fields of a node are stored in a single word in memory, as a triple $x.f = [x.p, x.r, x.b]$. The memory words are of size $2\lceil \log n \rceil + 1 = O(\log n)$ in this representation. In fact, with high probability the maximum rank is $O(\log n)$, and each node requires only $\log n + O(\log \log n)$ bits.

A note about our representation: If the root-bit $x.b$ is set to 1, x is a root, in which case our implementation ignores the value of the parent field $x.p$. In this way our representation differs from the classic representation, in which the parent field $x.p = x$ for a root node x. This fact is crucial to understanding the pseudo-code.

In the pseudo-code, \$ represents a random bit, i.e. a value with $Bernoulli(1/2)$ distribution.

Algorithm 13 is the pseudo-code for unite and link. The implementation of the *find*(x) procedure used in the code is discussed in the next subsection. To do *unite*(x, y), we start as in the sequential case by finding the roots u and v of the trees containing x and y, respectively (Line 2). If $u = v$, then x and y are already in the same set and nothing needs to be done (Line 3). Otherwise, x and y are in different trees, so we can try to link u and v by doing a CAS to make v the parent of u, or vice-versa (Line 4). We mark the *link* on Line 4 and subsequent linearization points with an asterisk in the code. Further explanation of the pivotal *link* procedure is in the next paragraph. But we must allow for the possibility of the CAS failing, which can happen for example if it tries to make v the parent of u but in the meantime some other process makes another node the parent of u. Notably, the CAS fails if the other process does exactly the same CAS and makes v the parent of u. A solution that works in either case is to simply continue walking up the tree from the present u and v (Line 5), until the paths intersect or another attempt at linking is necessary. This method was first proposed by Anderson and Woll [3] and was subsequently used by Jayanti and Tarjan [28].

```
1:  procedure unite$(x, y)$
2:      $u \leftarrow find(x)$; $v \leftarrow find(y)$*
3:      while $u \neq v$ do
4:          $link(u, v)$*
5:          $u \leftarrow find(u)$; $v \leftarrow find(v)$*

6:  procedure link$(u, v)$
7:      $[u_p, r, u_b] \leftarrow u.f$
8:      $[v_p, s, v_b] \leftarrow v.f$
9:      if $r < s$ then CAS$(u.f, [u_p, r, 1], [v, r, 0])$*
10:     else if $s < r$ then CAS$(v.f, [v_p, s, 1], [u, s, 0])$*
11:     else
12:         if $u < v$ then CAS$(u.f, [u_p, r, 1], [v, r+1, \$])$*
13:         else CAS$(v.f, [v_p, s, 1], [u, s+1, \$])$*
```

Algorithm 13: : Pseudo-code to unite x and y.

An explanation of our Linking heuristic: Link initially reads the fields of u and v (Lines 7-8). If r, the rank of u, is less than s, the rank of v, the link attempts to change the parent of u to v and the root-bit of u to 0 ("not a root"), while leaving the rank of u unchanged (Line 9). If s is less than r, the link proceeds symmetrically (Line 10). The interesting case is $r = s$: if $u < v$ (check on Line 12 succeeds), the algorithm tries to change the parent of u to v *and* increment the rank of u, *and* set the root-bit of u *randomly* (Line 12). This case deviates from our presentation in Algorithm 11, so we now explain why the algorithm tries to change *both* the parent and rank fields. If the update succeeds and the root-bit, $u.b$, gets set to 1, then u continues to be a root, and thus the parent field is disregarded by the algorithm, so it does not matter that it was changed to v. On the other hand, if the update succeeds and root-bit gets set to 0, then u is no longer a root, and the rank field is subsequently disregarded by the algorithm, so it does not matter that it was changed to $r + 1$. Therefore, although syntactically the algorithm changes both the parent *and* the rank fields, semantically only the parent *or* the rank changes. Thus the implementation matches the idea in Algorithm 11. Line 13 acts symmetrically in the case that $u.r = v.r$ and $v < u$.

Note: for the purpose of the analysis only, we think of the rank as *not* incremented if Line 12 or 13 makes a root a non-root.

A.1: The find procedure

We present the two different implementations of *find*(x), namely *naïve* and *two-try splitting* in Algorithm 14. These procedures reproduce Algorithm 5 and Algorithm 7 using the node representation with three fields (parent, rank, and root-bit) per node.

An explanation of Naïve find: Naïve find uses an auxiliary variable u (Line 2) that follows parent pointers up the tree (Line 4) until it reaches a root (Line 3). *find*(x) returns this node as the leader (Line 5). The linearization point of this procedure is the time at which u is discovered to be a root.

An explanation of the two-try splitting find pseudo-code: Find uses an auxiliary variable u to walk up the tree (Line 7). If u is not a root (Line 8), its parent v, and grand-parent $v.p = w$ are read (Line 9). If v is a root, Find has succeeded and can return v (Line 10); otherwise, an improvement is attempted on Line 11. A successful CAS on Line 11 changes only the parent of u, without modifying the other fields of u. After a second attempt to improve the same node u (Lines 12-14), u is replaced by its parent v, in order to keep walking up the tree (Line 15). Finally, if u becomes the root (Line 8), it is returned on Line 16. The linearization point of the procedure is the time when the root-bit of the returned node is read, since this is the time when the returned node is surely the root of the tree.

```
1:  procedure find($x$)
2:      $u \leftarrow x$
3:      while not $u.b^*$ do
4:          $u \leftarrow u.p$
5:      return $u$

6:  procedure find($x$)
7:      $u \leftarrow x$;
8:      while not $u.b^*$ do
9:          $[v, r, u_b] \leftarrow u.f$; $[w, s, v_b] \leftarrow v.f^*$
10:         if $v_b$ then return $v$
11:         CAS($u.f$, $[v, r, 0]$, $[w, r, 0]$)
12:         $[v, r, u_b] \leftarrow u.f$; $[w, s, v_b] \leftarrow v.f^*$
13:         if $v_b$ then return $v$
14:         CAS($u.f$, $[v, r, 0]$, $[w, r, 0]$)
15:         $u \leftarrow v$
16:     return $u$
```

Algorithm 14: : Find algorithms naïve and two-try splitting, respectively

Note: Removing the second attempt to improve u (Lines 12-14) from the pseudo-code of find with two-try splitting produces pseudo-code for find with one-try splitting.

References

1. Alistarh, D., Fedorov, A., Koval, N.: In search of the fastest concurrent union-find algorithm (2019)
2. Aghazadeh, Z., Golab, W., Woelfel, P.: Making objects writable. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14, pp. 385–395. Association for Computing Machinery, New York
3. Anderson, R.J., Woll, H.: Wait-free parallel algorithms for the union-find problem. In: Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA, pp. 370–380 (1991)
4. Bloemen, V.: On-The-Fly Parallel Decomposition of Strongly Connected Components. Master's thesis, University of Twente (2015)
5. Blum, N.: On the single-operation worst-case time complexity of the disjoint set union problem. In: Annual Symposium on Theoretical Aspects of Computer Science, STACS 1985 (1985)
6. Bloemen, V., Laarman, A., van de Pol, J.: Multi-core on-the-fly scc decomposition. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, page to appear (2016)
7. Chor, B., Israeli, A., Li, M.: On processor coordination using asynchronous hardware. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '87, pp. 86–97. Association for Computing Machinery, New York (1987)
8. Chung, F., Lu, L.: Concentration Inequalities and Martingale Inequalities: A Survey, pp. 79–127. Internet Mathematics (2005)
9. Cole, R., Vishkin, U.: Deterministic coin tossing with applications to optimal parallel list ranking. Inf. Control **70**(1), 32–53 (1986)
10. Dhulipala, L., Hong, C., Shun, J.: Connectit: a framework for static and incremental parallel graph connectivity algorithms (2020)

11. Ellen, F., Woelfel, P.: An optimal implementation of fetch-and-increment. In: Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205, DISC 2013, pp. 284–298. Springer, New York (2013)
12. Fraczak, W., Georgiadis, L., Miller, A., Tarjan, R.E.: Finding dominators via disjoint set union. J. Discrete Algor. **23**, 2–20 (2013)
13. Fischer, M. J.: Efficiency of equivalence algorithms. In: Complexity of Computer Computations, pp. 153–167. Springer, New York (1972)
14. Fredman, M.L., Saks, M.E.: The cell probe complexity of dynamic data structures. In: Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14–17, 1989, Seattle, Washigton, USA, pp. 345–354 (1989)
15. Galler, B.A., Fisher, M.J.: An improved equivalence algorithm. Commun. ACM **7**(5), 301–303 (1964)
16. Goel, A., Khanna, S., Larkin, D.H., Tarjan, R.E.: Disjoint set union with randomized linking. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5–7, 2014, pp. 1005–1017 (2014)
17. Hong, C., Dhulipala, L., Shun, J.: Exploring the design space of static and incremental graph connectivity algorithms on gpus. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (2020)
18. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
19. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Proceedings of the 16th International Conference on Distributed Computing, DISC '02, pp. 265–279, London, UK. Springer, New York (2002)
20. Herlihy, M., Eliot, J., Moss, B.: Transactional memory:architectural support for lock-free data structures. SIGARCH Comput. Archit. News, **21**(2), 289–300 (1993)
21. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
22. Hopcroft, J.E., Ullman, J.D.: Set merging algorithms. SIAM J. Comput. **2**(4), 294–303 (1973)
23. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
24. Halperin, S., Zwick, U.: Optimal randomized erew pram algorithms for finding spanning forests. J. Algor. **39**(1), 1–46 (2001)
25. Jayanti, P.: A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In: Distributed Computing, 12th International Symposium, DISC '98, Andros, Greece, September 24–26, 1998, Proceedings, pp. 216–230 (1998)
26. Jayanti, P.: A time complexity lower bound for randomized implementations of some shared objects. In: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, pp. 201–210. ACM, New York (1998)
27. Johnson, D.B., Metaxas, P.: Connected components ino (log3/2 n) parallel time for the crew pram. J Comput Syst Sci **54**(2), 227–242 (1997)
28. Jayanti, S.V., Tarjan, R.E.: A randomized concurrent algorithm for disjoint set union. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC '16, pp. 75–82. ACM, New York (2016)
29. Jayanti, S., Tarjan, R.E., Boix-Adserà, E.: Randomized concurrent set union and generalized wake-up. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC '19, pp. 187–196. Association for Computing Machinery, New York (2019)
30. Kozen, D.C.: The Design and Analysis of Algorithms. Springer, Berlin, Heidelberg (1992)
31. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Am. Math. Soc. **7**(1), 48–50 (1956)
32. Lattner, C., Adve, V.: Automatic pool allocation for disjoint data structures. SIGPLAN Not. **38**(2 supplement), 13–24 (2002)
33. Liu, S., Tarjan, R.E.: Simple concurrent labeling algorithms for connected components. In: Fineman, J.T., Mitzenmacher, M. (Eds.) 2nd Symposium on Simplicity in Algorithms (SOSA 2019), Volume 69 of OpenAccess Series in Informatics (OASIcs), pp. 3:1–3:20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2018)
34. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
35. Motorolla. Programmer's reference manual. https://www.nxp.com/files-static/archives/doc/ref_manual/M68000PRM.pdf. Accessed 07 Sept 2018
36. Reif, J.: Depth first search is inherently sequential. Inform. Process. Lett. **1**(2), 229–234 (1985)
37. Rastogi, V., Machanavajjhala, A., Chitnis, L., Das Sarma, A.: Finding connected components on map-reduce in logarithmic rounds. In: Proceedings - International Conference on Data Engineering 03 (2012)
38. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Comput. Math. Appl. **7**(1), 67–72 (1981)
39. Shiloach, Y., Vishkin, U.: An o(logn) parallel connectivity algorithm. J. Algor. **3**(1), 57–67 (1982)
40. Sedgewick, R., Wayne, K.: Algorithms, 4th edn. Addison-Wesley, London (2011)
41. Tarjan, R.: Depth first search and linear graph algorithms. Siam J. Comput. **1**(2), (1972)
42. Tarjan, R.: Testing flow graph reducibility. In: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, pp. 96–107 (1973)
43. Tarjan, R.: Finding dominators in directed graphs. SIAM J. Comput. **3**(1), 62–89 (1974)
44. Robert Endre Tarjan: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
45. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM **31**(2), 245–281 (1984)
46. Wolper, P.: Lectures on formal methods and performance analysis. In: Brinksma, H., Holger, K., Joost-Pieter (Eds.) Lectures on Formal Methods and Performance Analysis. Chapter Constructing Automata from Temporal Logic Formulas: A Tutorial, pp. 261–277. Springer, New York (2002)