



A Parallel Solution to the Extended Set Union Problem with Unlimited Backtracking

M. Cristina Pinotti^{†,‡}, Vincenzo A. Crupi[†] and Sajal K. Das[‡]

[†] Ist. di Elaborazione dell'Informazione [‡] Department of Computer Sciences
 Consiglio Nazionale delle Ricerche University of North Texas
 56126 Pisa, Italy Denton, TX 76203-6886, U.S.A.

Abstract

In this paper, we study on the EREW-PRAM model a parallel solution to the extended set union problem with unlimited backtracking which maintains a dynamic partition Π of an n -element set S subject to the usual operations Find, Union, Backtrack and Restore as well as the new operations SetUnion, MultiUnion. The SetUnion operation is a special case of the commonly known Union operation aimed to unify two prespecified set-names, while MultiUnion operation deals with a batch of Union operations. A new data structure, called k -Parallel Union Find (or, k -PUF) trees, is introduced to represent a disjoint set in Π . The structure is defined for a wide range for the parameter k , but the more interesting results are achieved for $k = \frac{\log n}{\log \log n}$. In this case, using the k -PUF trees, both SetUnion and Restore operations are performed in constant parallel time requiring optimal work $O(k)$. This constant-time performance is not achievable parallelizing the existing data structures. Moreover, using $p = O(k)$ processors, MultiUnion for a batch of p operations is performed in $O(k)$ time, requiring optimal work $O(pk)$.

1 Introduction

The *set union* (also known as *union-find*) problem consists of maintaining an efficient internal representation of a dynamic partition Π of an n -element set S which is subject to a sequence of the following two operations:

- **Union**(a, b): combines into a new subset two disjoint subsets A and B containing, respectively, the elements a and b of the set S .
- **Find**(x): returns the name of the unique subset of S that currently contains the element x .

Recently, some variants of the set union problem have been considered where an individual or a sequence of unions can be *backtracked* [1, 5, 6, 8]. Backtracking arises in such problems as incremental logic programming memory management [7], and clustering in pattern analysis and recognition [4]. The more recent and flexible approach for the backtrack process is due to Apostolico et al.[1]. They introduced for the *set union problem with unlimited backtracking* two new operations – *Backtrack*(i) and *Restore*(e). *Backtrack*(i) undoes the last i unions not yet undone, for any positive integer i , while *Restore*(e) updates the edge e and a special set of its neighborhoods when a Union is required after a Backtrack operation. Their implementation takes worst-case time $O(\log n / \log \log n)$ for Union, Find and Restore operations and constant time for Backtrack(i),

for any i . This bound is tight compared with the lower-bound of a single operation worst-case time complexity in [2].

In this paper, we study on the EREW-PRAM (exclusive-read and exclusive-write, parallel random access machine) model the *extended set union problem with unlimited backtracking*, which maintains an efficient dynamic partition Π of an n -element set S subject to, besides the Union, Find and Backtrack operations, the following new operations:

- **Restore**(A, t): restores the configuration of the subset A of S at time t . (This is different from the Restore operation defined in [1].)
- **MultiUnion**[[$(a_1, b_1), \dots, (a_p, b_p)$]]: combines the $2p$ subsets (not necessarily all distinct) of S $A_1, B_1, \dots, A_p, B_p$ containing, respectively, the elements $a_1, b_1, \dots, a_p, b_p$.
- **SetUnion**(A, B): receives two subset-names A and B , rather than two elements of S , and combine their elements. If either the subset-name A or B is not present on the partition when the SetUnion is called, the more recent partition Π of S where both A and B are present is forced back by a suitable Backtrack operation.

Letting $k = \frac{\log n}{\log \log n}$, the procedure MultiUnion is performed in $O(k)$ time and optimal work $O(pk)$ using $p = O(k)$ processors; whereas the SetUnion and Restore operations require $O(k/p)$ time and optimal work $O(k)$ using $p = O(k)$ processors.

This paper is organized as follows. In Section 2, we survey the existing data structures [1] for the set union problem with unlimited backtracking, on which our new parallel data structure is based, and explain that no parallel constant time algorithm can be achieved for the Restore operation using the structure in [1]. In Section 3, the new parallel data structure, which is also well defined in the sequential computing context, is discussed. In Section 4 the algorithms for Find, Union and Restore as well as the new operations – SetUnion, and MultiUnion – are developed. Conclusions are offered in Section 5.

2 k -BUF trees

Apostolico et al. [1] represent a dynamic partition Π of a set S of n elements supporting Union, Find, Restore and Unlimited-Backtrack operations as a forest of k -Backtrack Union Find (k -BUF) trees such that there is one k -BUF tree for each disjoint set in Π . Each k -BUF tree, defined for any integer $2 \leq k \leq n$, and retaining the basic structure of

the k -Union Find tree introduced by Blum [2], is a rooted tree T such that:

1. The root has at least two sons and contains the set-name. The root of T is said to be *slim* if it has less than k sons, otherwise it is *fat*.
2. Each node v has a pointer to its father $p(v)$.
3. All leaves are at the same level and each leaf stores an element of S .
4. Each node v except the root has at least k sons. Assuming that the height of the leaves is equal to 1, a k -BUF tree, T , with n leaves has height $\leq \lceil \log_k n \rceil + 1$.
5. At least the k leftmost sons of each node v are linearly ordered in a doubly linked list.

Roughly speaking, the k -BUF trees grow and shrink under the action of the Union and Restore operations. In order to describe how to combine two trees, named T_A and T_B , let r_A and r_B be the roots of two such k -BUF trees and let us assume that $\text{height}(T_B) \leq \text{height}(T_A)$. Letting v be the node on the path from a leaf of T_A to r_A such that the subtree rooted at v has the same height as T_B , the unions can be classified into the following three types [2]:

Type 1: If root r_B is fat and $v \neq r_A$, then r_B becomes sibling of v . The arc $\langle r_B, p(v) \rangle$ is said the characteristic arc of a union of Type 1.

Type 2: If both roots r_A and r_B are fat and $v = r_A$, then r_A and r_B are made sons of a newly created root r . W.l.o.g., the new set will be referred to as A . Either the arc $\langle r_A, r \rangle$ or $\langle r_B, r \rangle$ is characteristic of a union of Type 2.

Type 3: This case covers all remaining possibilities. If root r_B is slim, then the sons of r_B become the rightmost sons of v and r_B is released. If root r_B is fat then, since $v = r_A$ and v is slim, the sons of r_A are made the rightmost sons of r_B , and r_A will be released. The leftmost arc among the redirected arcs, referred to as a *separator arc*, is characteristic of a union of Type 3.

Before completing the union description, let us add that each union is uniquely determined by an integer and a new union receives an ordinal number equal to 1 plus the ordinal number of the last valid union performed. Letting i_{\max} be the ordinal number of the last valid union performed, $\text{Backtrack}(i)$, which undoes the last i unions, is implemented just setting $i_{\max} = \max\{i_{\max} - i, 0\}$.

From now on, in the dynamic partition Π , a union is *valid* if it is not yet undone by backtracking operations, and *void* otherwise. Moreover, since a lazy approach is adopted for the backtracking process, a *void* union can be *persistent* if the arcs associated with it have not yet been removed, otherwise it is *dissolved*. Consequently, an arc can be live, dead, or cheating and a separator can be active or inactive. A *live arc* is a connection whose last union that handled it is still valid. A *dead arc* is a connection whose first union that created it is void. An arc is *cheating* when the first union that created it is still valid but the last union that handled it is void. So the cheating arcs must be redirected correctly, but not destroyed. Furthermore, a separator arc is *active* if the union which made it is valid, otherwise it is *inactive*. Similarly, a node on the k -BUF tree is called a *live apex* if it is a k -BUF root, and it is a *dead apex* (resp. *cheating apex*) if the outgoing arc is dead (resp. cheating). The status of the arc e is captured by storing on each arc $\text{first_union}(e)$, that is the first union that

created it; $\text{last_union}(e)$, the last union that manipulated it; and $\text{separate_union}(e)$, the union (if it exists) that made it a separator. Each separator s also keeps in $\text{label}(s)$ the set-name destroyed by the $\text{separate_union}(s)$.

Now, the $\text{Find}(x)$ operation return the set-name to which x belongs. Precisely, $\text{Find}(x)$ visits the path on the tree from the leaf containing x until an apex r is reached. If the apex is live or dead, the set-name is stored in r . If the apex is cheating, there is at least one inactive separator, named is , within k arcs to the left of the outgoing cheating arc. $\text{Find}(x)$ will return the set-name stored in $\text{label}(is)$.

Finally, the first task of $\text{Union}(a,b)$ consists of finding the roots of the trees storing a and b by $\text{Find}(a)$ and $\text{Find}(b)$. After the two apexes r_A and r_B are located, the Restore procedure is applied to detach their upgoing arcs from the remaining data structure, obtaining two separate k -BUF trees, T_A and T_B . Then, before using the integer $i_{\max} + 1$ as identifier of the new union (assuming that i_{\max} is the last valid union), such integer must be released by the old structure if it is in use. That is, the persistent Union with the same ordinal number is made void by applying Restore to its characteristic arc. Lastly, the melding of T_A and T_B is of Type 1, 2 or 3 and it is performed according to the rules described before.

It remains to analyze the *Restore* procedure. Recalling that the edges in every node of the k -BUF tree are grouped into clusters where a *cluster* is defined as a maximal set of consecutive sibling arcs whose last_union field is set to the same value, $\text{Restore}(e)$ removes all dead arcs from the cluster E to which the input arc e belongs, and partitions the remaining arcs in a certain number of smaller yet live clusters (i.e., clusters of live edges). For each edge $e \in E$, let $as(e)$ and $is(e)$ be, respectively, the rightmost active and rightmost inactive separators on the left of e . The Restore will move each arc e to a new root with the name $\text{label}(is(e))$ and will reset $\text{last_union}(e) = \text{separate_union}(as(e))$ or $\text{last_union}(e) = \text{first_union}(e)$ if the separator $as(e)$ is undefined.

Observe that Restore is accomplished in $O(k)$ time, while Find and Union require $O(\log_k n + k)$ time [1].

Let us justify why the preceding $\text{Restore}(e)$ procedure, cannot be accomplished in parallel in constant time on the EREW-PRAM model. To compute $is(e)$ and $as(e)$ for each $e \in E$, we should use two

suitable segmented broadcast which cannot be performed on the EREW-PRAM model faster than $O(\log k + \frac{k}{p})$ time where $|E| \leq k$.

3 New data structures for the extended set union problem

In this section, we present a modification of the k -BUF trees to handle the Restore procedure in constant parallel time. This has been the first motivation of our work.

On the EREW-PRAM model, we represent a dynamic partition Π of an n -elements set S subject to the extended set union problem with unlimited backtracking. The data structure is represented by a forest of k -Parallel Union Find (or k -PUF) trees, one for each disjoint set in Π , and by n Set-Stacks, one for each possible set-name in Π . Our structures consume $O(nk + n \log_k n)$ space, rather than $O(n)$ space required by the k -BUF tree since each Set-Stack has at most $\log_k n$ entries and each k -PUF node occupies $O(k)$ space.

Formally, for $2 \leq k \leq n$, a k -PUF tree is a rooted tree T such that:

1. The root has at least two sons and stores the name of the disjoint set in S represented by T .
2. Each node v except the root has at least k sons.
3. At least k leftmost sons of each node v are linearly ordered in a doubly linked list (the list of the root can contain fewer than k sons). Pointers to the head and tail of the son list at the node v are saved in $head(v)$ and $tail(v)$. When the nodes are moved from v to a new slim root u , the list of v is linked as the rightmost part of the list of u and $tail(u)$ is updated.
4. An array of size k , referred to as CH_v , is associated with each node v . For each j , where $1 \leq j \leq k$, the two fields $CH_v[j].node$ and $CH_v[j].ordinal$ store, respectively, the pointer to the j -th son of v from the left to the right and the ordinal number of the union that made such node a son of v . The arcs $\langle v, CH_v[j].node \rangle$, for $1 \leq j \leq k$, from a father to its sons will be referred to as the *downgoing* arcs of v .
5. Each node v stores the set-name assigned to it when it was created.
6. Each node v stores the pointer to its father $p(v)$. This arc, $e = \langle v, p(v) \rangle$, will be referred to as the *upgoing* arc of v . The attributes $first_union(e)$ and $last_union(e)$ are saved in the source node v of e . If e is a separator, besides the attributes $separate_union(e)$ and $label(e)$, a new field $old_father(e)$ is introduced which stores the pointer to the node which was left by e at the time of the $separate_union(e)$.
7. All the leaves are at the same level and each leaf contains an element of the set S on which the partition Π is defined.

In practice, each node v has a record with three logical components: downgoing arcs, upgoing arc and information on the node itself.

Moreover, let the *set-apex* of the set-name A be the highest node on the k -PUF tree created by a union not yet undone and containing the set-name A . If the set-name A is active, the set-apex of A is the root of a k -PUF tree.

For better understanding, let us explain that each time an upgoing arc moves into a new slim father, a downgoing arc is registered in the new father. Hence, for each node v there may be several persistent downgoing arcs – one for each node $p(v)$ which has been father of v . Each downgoing arc registers in its ordinal field the time when the node v was made son of $p(v)$. The fields, *ordinal*, of the two leftmost downgoing arcs of each internal node will always contain the same integer as they are created together by a union of Type 2. Since the nodes transferred to a new father are attached as the rightmost sons in their new father, the fields *ordinal* in CH will be in nondecreasing order scanning from left to right. By this ordering after a backtrack operation, the live sons of a node are always the leftmost among their siblings. It is noteworthy that only the first k leftmost sons must be recorded. Namely, we need to access quickly the sons if and only if they are moved (resp., moved back) to (resp., from) a new root. The above discussion can be summarized as follows.

INVARIANT 0 [downgoing arc consistency]: *Let v be a node in the forest of k -PUF trees. If v is not a leaf, let $m \leq k$ (only for the root the inequality $m < k$ holds) be the number of entries in CH_v . For each j , for $1 \leq$*

$j \leq m$, the node $CH_v[j].node$ becomes a son of v by the union with ordinal number $CH_v[j].ordinal$. Moreover, $CH_v[j].ordinal \leq CH_v[j+1].ordinal$ for $1 \leq j \leq m-1$, $head(v) = CH_v[1].node$ and $tail(v) = CH_v[m].node$.

Finally, when v becomes the i -th live son of node r , where $1 \leq i \leq m$, $last_union(v) = CH_r[i].ordinal$ and both arcs $\langle v, r \rangle$ and $\langle r, v \rangle$ are present at that moment. Moreover, let $e' = \langle r_A, w \rangle$ be a downgoing arc and let $w = CH_{r_A}[j].node$ for some j with $1 \leq j \leq k$. Then, e' is live if $CH_{r_A}[j].ordinal \leq i_{\max}$ holds and r_A is a set-apex. In the case, $CH_{r_A}[j].ordinal > i_{\max}$ and r_A is a set-apex, e' is said to be discardable.

As mentioned earlier, a Set-Stack is associated to each set-name. When a new root r of a k -PUF tree at height i containing the set-name A is created by the union with the ordinal number $ord(u)$, the pointer to r and the time stamp $ord(u)$ are saved respectively into $P_A[i].node$ and $P_A[i].ordinal$. If the set-name A is inactivated by the Union $ord(u)$ when it was stored in the root at height h of a given k -PUF, a new entry $P_A[h+1]$ is pushed on the top of P_A . Additionally, $P_A[h].node$ is copied in $P_A[h+1].node$ while $P_A[h+1].ordinal = ord(u)$. Therefore, if a set-name is inactive, the last two entries in its Set-Stack point to the same k -PUF node. Then, we have:

INVARIANT 1 [set-apex consistency]: *Let A be a set-name and h be the number of entries pushed onto the Set-Stack P_A . Then, $P_A[1].node$ points to a k -PUF leaf containing the singleton element A of S , while $P_A[j].ordinal < P_A[j+1].ordinal$ for each $1 < j \leq h-1$. Moreover, at any time i_{\max} , the set-apex r_A for the set A is determined as follows:*

1. if $P_A[h].ordinal \leq i_{\max}$ and $P_A[h-1].node = P_A[h].node$ then $r_A := P_A[h-1].node$.
2. if $P_A[h].ordinal \leq i_{\max}$ and $P_A[h-1].node \neq P_A[h].node$ then $r_A := P_A[h].node$.
3. if $P_A[j].ordinal \leq i_{\max} < P_A[j+1].ordinal$, where $1 \leq j < h$, then $r_A := P_A[j].node$.

Finally, P_A grows until the set-name A will be made inactive, and hence P_A has at most $\log_k n$ entries.

Moreover,

INVARIANT 2 [upgoing arc consistency]: *An upgoing arc e is dead if and only if $i_{\max} < first_union(e)$; cheating if and only if $first_union(e) \leq i_{\max} < last_union(e)$; and live if and only if $i_{\max} \geq last_union(e)$.*

Finally, recalling that there are three types of unions and that each union is uniquely determined by an integer as said in Section 2, when either the $Union(a,b)$ or the $SetUnion(A,B)$ is performed with ordinal number $ord(u)$, the pointers to the roots of the two trees T_A and T_B combined during the union are stored in the array $Unions$ at the position $ord(u)$.

INVARIANT 3 [union-numbering consistency]: *At any time i , $1 \leq i \leq i_{\max} \leq n-1$, there exists at most one union with ordinal number i stored in the array $Unions$. In the forest there are exactly two sibling arcs $e_1 = \langle v_1, w \rangle$ and $e_2 = \langle v_2, w \rangle$ such that $first_union(e_1) = first_union(e_2) = i$ and $CH_w[1].ordinal = CH_w[2].ordinal = i$, if i is the ordinal number of a union of Type 2. For a union of Type 1, there is a single upgoing arc $e = \langle v, w \rangle$*

with $\text{first_union}(e) = i$, and at most a downgoing arc $CH_w[j].\text{ordinal} = i$ for some j where $1 \leq j \leq k$ if e is one of the k leftmost sons of w . Finally, for a union of Type 3, there is at most one upgoing separator arc $e = \langle v, w \rangle$ with $\text{separate_field}(e) = i$. This implies that there are at most $k - 1$ upgoing arcs with the last union fields equal to i . Consequently, there are at most $k - 1$ subsequent downgoing arcs on the array CH_w storing i in their ordinal fields.

4 Parallel operations on k -PUF trees

In the following we devise a parallel implementation for the basic Find, Restore and Union operations as well as the two new operations SetUnion and MultiUnion.

4.1 Find operation

The Find(x) operation determines the set-name A to which x belongs, and then returns the node candidate to be the root of the k -PUF tree representing A , i.e. the candidate set-apex of A . Precisely, Find starts from the leaf containing x and follows the live upgoing arcs until an apex node, i.e. a node without upgoing arc or whose upgoing arc is dead or cheating, is traversed. Depending on the status of $\text{apex}(x)$, the following candidate set-apex is returned:

INVARIANT 4 [find consistency]: *If $\text{apex}(x)$ is dead or live (i.e., it contains the name of the set to which x belongs), $\text{apex}(x)$ itself is the set-apex candidate and it is returned. Otherwise, if $\text{apex}(x)$ is cheating, the old_father field of the rightmost inactive separator on the left of $\text{apex}(x)$ is returned. Such inactive separator is found by checking at most $k - 1$ edges on the left of the upgoing arc of $\text{apex}(x)$.*

Unfortunately, Find is strictly sequential in nature.

Theorem 1 *On the EREW-PRAM model, the procedure Find(x) is performed in $O(\log_k n + k)$ time and $O(p \log_k n + pk)$ work.*

This inefficiency is intrinsic to the pointer data structures.

4.2 Restore operation

The new Restore operation works on a set-apex, rather than on a cluster of edges. The procedure Restore(r_A, i_{\max}) receives r_A , a set-apex candidate, and the number i_{\max} of the last valid union.

To reprecinate the set-apex r_A , according to Invariant 1, if r_A is a dead apex(x) node (i.e., a node whose upgoing arc is dead), the Restore procedure removes the dead upgoing arc and possibly the other nodes at higher levels on the path of the k -PUF tree containing the set-name A . If r_A is a live apex storing an inactive set-name A , only the top of P_A is released by the Update procedure making A active again. Note that the corresponding node is not released as it is still in use. If r_A is a live apex containing an active set-name, r_A is already a set-apex and no work is required to update the upgoing arc.

Observe that, the set-apex of the set-name A is restored in constant parallel time accessing simultaneously all the entries of P_A and determining the entry which satisfies Invariant 1. This will be the new top of P_A .

The other goal of the Restore procedure is cleaning up the arcs incoming and outcoming from r_A preserving Invariants 0 and 2. For each downgoing arc, we check if it is live or not testing if $CH_{r_A}[j].\text{ordinal} \leq i_{\max}$.

When a live son is detected, the corresponding upgoing arc $\langle CH_{r_A}[j].\text{node}, r_A \rangle$ is forced back to r_A accessing $CH_{r_A}[j].\text{node}$ and the corresponding attributes are promptly reset. If $CH_{r_A}[j].\text{ordinal} > i_{\max}$, the corresponding entry in CH is made void. Let j be the first position satisfying this condition. On its right, every other position will be cleared in CH , and $j - 1$ will be the current live sons of r_A .

Finally, pointers head(r_A) and tail(r_A) of the doubly linked list are updated so that they point, respectively, to the first and the last sons which are live in CH . Eventually, the first son of r_A is detached from its old sibling. Since the array CH can be accessed in constant time, we conclude,

Theorem 2 *On the EREW-PRAM model, the procedure Restore(r_A, i_{\max}) is performed in $O(\frac{\log_k n + k}{p})$ time and $O(\log_k n + k)$ work, requiring $O(nk + n \log_k n)$ space. For $p = k = \frac{\log n}{\log \log n}$, the node r_A is restored in $O(1)$ time.*

In order to prove the correctness of the Restore procedure, observe that, the Find operation visits only live or cheating upgoing arcs, whereas the Restore operation removes dead upgoing arcs or it redirects the cheating upgoing arcs whose actual father is the set-apex returned by the Find operation. The remaining cheating upgoing arcs, and their inactive separators, are untouched. Therefore, after the Restore operation, Find(y) for any item $y \in S$, returns the same set-apex candidate as if executed before the Restore procedure. For more details, see [3].

4.3 Union operation

The Union operation for both k -PUF trees and k -BUF trees is exactly the same with the exception that the Restore procedure, when it is invoked, works in the first representation on a set-apex candidate and on an upgoing arc in the second one [3]. From the rules given in Section 2, using p processors, it holds:

Theorem 3 *On the EREW-PRAM model, two k -PUF trees are melded in $O(\frac{\log_k n + k}{p})$ time and $O(\log_k n + k)$ work using $O(nk + n \log_k n)$ space.*

However, due to the fact that the Find procedure is strictly sequential, the following performance is obtained for the overall Union operation, employing p processors:

Theorem 4 *On the EREW-PRAM model, the procedure Union is performed in $O(\log_k n + \frac{k}{p})$ time and $O(p \log_k n + k)$ work using $O(nk + n \log_k n)$ space.*

The union process is extended efficiently in the parallel computing environment by the two new operations: SetUnion and MultiUnion. The SetUnion operation is totally independent of the Find operation since it is working on prespecified set names, while the MultiUnion operation performs a batch of p union operations simultaneously, one for each processor available.

4.4 SetUnion procedure

The SetUnion procedure receives two set-names A and B . If either the subset-name A or B is not present on the partition when the SetUnion is called, the more recent partition Π of S where both A and B are present is forced back by a suitable Backtrack operation. In fact, if both A and B are inactive, let us consider the two last instants $t_A = P_A[\text{top}].\text{ordinal} - 1$ and $t_B = P_B[\text{top}].\text{ordinal} - 1$

when, respectively, A and B were active in the partition Π . Clearly, if A (resp., B) is active, $t_A = i_{\max}$ (resp., $t_B = i_{\max}$). W.l.o.g., let $t_A = \min\{t_A, t_B\}$. Then, t_A is fixed as the new current time stamp on the structure by a suitable backtrack operation. Therefore, the two set-apexes of A and B at the time t_A are located in the corresponding set-stacks by the *GetSetApex* procedure and the corresponding trees are melded. Since the *GetSetApex* procedure can be implemented in constant parallel time by assigning one processor to each entry of the set-stack to locate the entry satisfying Invariant 1, we have:

Theorem 5 *On the EREW-PRAM model, the procedure SetUnion is performed in $O(\frac{\log_k n+k}{p})$ time and $O(\log_k n + k)$ work using $O(nk + n \log_k n)$ space.*

4.5 MultiUnion procedure

The *MultiUnion* procedure performs a batch of p union operations preserving their relative order so that we can perform *Backtrack(i)*, for any $i > 0$, without paying attention if the unions to be destroyed were executed in batches or individually.

The *MultiUnion* operation receives p pairs (a_i, b_i) of sets, where $1 \leq i \leq p$. For each pair, the two set-apexes r_{a_i} and r_{b_i} returned, respectively, by *Find(a_i)* and *Find(b_i)* are stored into the i -th entry of an auxiliary array, *Batch*. The unions are then completed one after the other invoking *restore* and *combine* procedures. In order to preserve the relative order among the unions, after the i -th union, all the previous occurrences of either r_{a_i} or r_{b_i} on the rightmost $p - i$ entries in *Batch* are substituted by the set-apex created by the i -th union.

In this way, we complete $O(p)$ unions in $O(\log_k n + k)$ time using $O(p)$ processors. Unfortunately, here the CREW (concurrent-read and exclusive-write)-PRAM model is required as simultaneous accesses occur during the *MultiFind*. Nevertheless, the same performance can be obtained even on the weaker EREW-PRAM model using the pipeline technique and storing in a private array for each upgoing arc its moves from a node to another. (The details are omitted for space constraints [3]).

Theorem 6 *On the EREW-PRAM model, a batch of $O(p)$ Unions is performed in $O(p + \log_k n + k)$ time and $O(p(p + \log_k n + k))$ work using $O(nk + n \log_k n)$ space.*

5 Conclusions

In this paper, we have studied on the EREW-PRAM model the extended set union problem with unlimited backtracking which maintains a dynamic partition Π of a subset under the operations *Find*, *Restore*, *Backtrack*, *SetUnion*, *Union* and *MultiUnion*. Each disjoint set in Π is represented by a *k-Parallel Union Find* (in short, *k-PUF*) tree and a *Set-Stack*. Our data structure is a modification of the existing union-find trees [1] supporting the *Restore* operation in constant parallel time, and performing the new operations *SetUnion* (working on set names) and *MultiUnion* with optimal work using $O(\log n / \log \log n)$ processors.

References

[1] Apostolico, A., Italiano, G.F., Gambosi, G. and Talamo, M., "The set union problem with unlimited backtracking" *SIAM J. Comput.*, 23, (1994), pp. 50-70.

[2] Blum, N., "On the single operation worst-case time complexity of the disjoint set union problem" *SIAM J. Comput.*, 15, (1986), pp. 1021-1024.

[3] M.C. Pinotti, S.K. Das and V.A. Crupi, "A Parallel Solution to the Extended Set Union with Unlimited Backtracking", Tech Rep. CRPDC-95-8, Dept. of Computer Science, University of North Texas, Denton, TX 76203, September 1995.

[4] Duda, P. and Heart, R., *Pattern classification and scene analysis*, J. Wiley and Sons, New York, 1973.

[5] Gambosi, G., Italiano, G.F. and Talamo, M., "Worst-case analysis of the set union problem with extended backtracking" *Theoret. Comput. Sci.*, 68, (1989), pp. 57-70.

[6] Gambosi, G., Italiano, G.F. and Talamo, M., "The set union problem with dynamic weighted backtracking," *BIT*, 31, (1991), pp. 382-393.

[7] Hogger, C.J. *Introduction to Logic Programming*, Academic Press, New York, 1984.

[8] Mannila, H. and Ukkonen, E., "Time parameter and arbitrary deunions in the set union problem" *Proc. 1st Scandinavian Workshop on Algorithm Theory*, Halmstad, Sweden, 1988, pp. 34-42.