Efficient parallel algorithms for manipulating sorted sets

Jyrki Katajainen*
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East
Denmark

Abstract

The parallel complexity of various operations on sorted sets is studied. Let X and Y be two sorted sets of respective size n and m, $n \geq m$. Paul, Vishkin, and Wagener [*R.A.I.R.O. Informatique théorique* **17** (1983) 397–404] proved that the standard set operations $X \cap Y$, $X \cup Y$, and $X \setminus Y$ can be accomplished in $O(\log n)$ time with m processors on an EREW PRAM. In this paper we improve this result by showing that the same time-performance is obtainable with $\lceil (m \log((m+n)/m))/\log n \rceil$ processors. At the same time we reduce the space requirements from $O(n + m \log n)$ to $O(n)$. The work done by the new algorithms matches the lower bound that can be proved for any comparison-based sequential algorithm performing these operations. We provide also efficient parallel algorithms for other related set operations.

1 Introduction

Let X be a set of n items $x_1, x_2, \ldots, x_n$, each *item* x_i consisting of a *key*, denoted $key(x_i)$, from some linearly ordered set $(\mathcal{U}, \leq)$ and some *information* associated with that key. We assume that, for any key $k \in \mathcal{U}$, there is at most one item x in X for which $key(x) = k$. In a *sorted set* the items $x_1, x_2, \ldots, x_n$ are arranged in increasing order of keys, *i.e.*, $key(x_1) < key(x_2) < \ldots < key(x_n)$. We use $\langle x_1, x_2, \ldots, x_n \rangle$ to denote the sorted set corresponding to the set $\{x_1, x_2, \ldots, x_n\}$. The abstract data type "sorted set" (cf. [15, Chapter 4]) can be implemented by using search trees (*e.g.*, 2-3 trees as done in this paper) or arrays, depending on the operations that should be efficient in the application in mind.

Dynamic sets are often manipulated itemwise. For example, we might want to test membership in, insert items into, and delete items from a set, that is, we would like to implement a *dictionary*. In order to utilize the potential power of a parallel machine more efficiently, it is interesting to consider how a *batch* of access operations is executed in parallel. It is convenient to assume that the access operations in a batch are ordered according to the keys to be accessed. This way the batch can also be seen as a sorted set. Given two sorted sets X and Y, the operations $X \cap Y$, $X \cup Y$, and $X \setminus Y$ corresponds to the dictionary operations member, insert, and delete, respectively. In precise terms, the operations are defined as follows:

proc INTERSECTION(X, Y: sorted set **of** [item])
returns sorted set **of** [item]
Return the sorted set $X \cap Y$ containing those items x of X, for which there exists an item y in Y such that $key(x) = key(y)$.

proc UNION(X, Y: sorted set **of** [item])
returns sorted set **of** [item]
Return the sorted set $X \cup Y$ containing all items of X and Y, that is, the two sorted sets are merged together. If $X \cap Y$ is not empty, the items of $X \cap Y$ are removed from X. This operation destroys X and Y.

proc DIFFERENCE(X, Y: sorted set **of** [item])
returns sorted set **of** [item]
Return the sorted set $X \setminus Y$ containing those items x of X, for which there does not exist any item y in Y such that $key(x) = key(y)$. This operation destroys X.

To make these operations useful we need a constructor creating an item from a given key and information (possible dummy), and functions returning the key and information of a given item. These operations are application dependent, but we assume that the items can be manipulated in unit time by a single processor. We may also need a conversion routine that converts an array to a sorted set, and vice versa. In this paper, we are not interested in how these conversions (sometimes involving sorting) are done. One should, however, be aware that it might be more costly to

*Supported partially by the Academy of Finland under contract No. 1021129 (project "Parallel Models of Computation and Parallel Algorithms").

perform these conversions than the set operations mentioned above.

For two sets X and Y, we use the notation $X < Y$, if the largest key of the items in X is smaller than the smallest key of the items in Y. If $X < Y$ or $Y < X$, the sets are said to be *disjoint*. We define three other useful operations on sorted sets.

proc SEARCH(X, Y: sorted set **of** [item])
returns array$[1..|Y|]$ **of** [item]

Let $X = \langle x_1, x_2, \ldots, x_n\rangle$ and define x_0 and x_{n+1} such that $key(x_0) = -\infty$ and $key(x_{n+1}) = \infty$. Further, let $Y = \langle y_1, y_2, \ldots, y_m\rangle$. Return a sorted array $[x_{i_1}, x_{i_2}, \ldots, x_{i_m}]$ of items from $X \cup \langle x_{n+1}\rangle$ such that for each j, $j = 1, 2, \ldots, m$, $key(x_{i_j-1}) < key(y_j) \leq key(x_{i_j})$.

proc JOIN(Z:**array**$[1..m]$ **of** [sorted set **of** [item]])
returns sorted set **of** [item]

Return the sorted set formed by combining disjoint sets in $Z = [Z_1, Z_2, \ldots, Z_m]$. This operation assumes that $Z_1 < Z_2 < \ldots < Z_m$. It also destroys the sets Z_i.

proc SPLIT(X, Y: sorted set **of** [item])
returns array$[1..2|Y|+1]$ **of** [sorted set **of** [item]]

Return a partition $[X_1, X_2, \ldots, X_{2m+1}]$ of X determined by the items of $Y = \langle y_1, y_2, \ldots, y_m\rangle$ so that, for all i, $i = 1, 2, \ldots, m$, $X_{2i} = \langle y_i\rangle$; $X_1 < X_2 < \ldots < X_{2m+1}$; and $X = \bigcup_{i=1}^{2m+1} X_i$. This operation assumes that $Y \subseteq X$.

To express the complexity of the operations, we assume that $n = |X| = |\bigcup_{i=1}^{m} Z_i|$, $m = |Y|$, and $n \geq m$. Paul *et al.* [14] proved that each of the above mentioned operations can be executed in[1] $O(\log n)$ time by using m processors on an EREW PRAM. In this paper we improve their algorithms by showing that the time bound $O(\log n)$ can be achieved by using only $\lceil (m \log(n/m))/\log n\rceil$ processors. This result is significant since the processor-time product, *i.e.*, the *work* performed by our algorithms, is the same order of magnitude as the worst-case lower bound $\Omega(m \log(n/m))$ on all comparison-based sequential algorithms for most of these operations. Brown and Tarjan [4] proved the lower bound for the union operation. By using information-theoretic arguments similar to theirs, the same lower bound is seen to be valid for the intersection, difference, search, and split operations as well. Therefore, our algorithms are work-optimal for all values of m, whereas the previous result guarantees work-optimality only when m is significantly smaller than n. The new algorithms are presented in Section 3.

[1] We use $\log x$ to denote $\log_2(\max\{2, x\})$.

Storage allocation is often a problem in parallel algorithms that are based on data structures using pointers. When running p processors in parallel, we have to be able to allocate p new records (cf. the NEW operation in Pascal) or dispose of p old records simultaneously. Paul *et al.* [14] showed how one can solve this memory management problem when manipulating sorted sets. However, their algorithms still require $O(n + m \log n)$ space. When reducing the number of processors needed from m to $\lceil (m \log(n/m))/\log n\rceil$, we can also reduce the space requirements to optimal $O(n)$. The storage allocation of our algorithms is discussed in Section 4.

2 Preliminaries

The model of computation used throughout this paper is the EREW PRAM (for the definition, see, *e.g.*, [10]). We utilize the well-known self-simulation property of the EREW PRAM model. The result mentioned in the next lemma will be sufficient for our purposes.

Lemma 2.1 *Let C be a computation that requires t time with q processors on an EREW PRAM. The outcome of C can be produced by another EREW PRAM having only p, $p \leq q$, processors in $O(tq/p + \log p)$ time.*

Proof. The result is otherwise obvious but we are willing to pay $O(\log p)$ time in order to broadcast the number q (and possibly also the program, if not given for free) to all p processors. ■

As in many other parallel algorithms, prefix computations are frequently used in our algorithms. The next result, the proof of which can be found in [12], is part of computer folklore.

Lemma 2.2 *Given an array $[a_1, a_2, \ldots, a_m]$ of m elements, all prefix sums $[a_1, a_1 + a_2, \ldots, \sum_{i=1}^{m} a_i]$ can be computed in $O(\log m)$ time and $O(m)$ space by using $\lceil m/\log m\rceil$ processors on an EREW PRAM. This result holds for any associative operation $+$.*

We need Lemma 2.2 to solve the following problems within the stated resource bounds (for the reductions, see [6, Section 2.2]):

(1) *Ordered compaction*: Given an array of m elements where certain members are marked and the others are unmarked. Extract a subarray consisting only of the marked members.

(2) *Group copying*: Given an array of m elements which is divided into groups of consecutive elements. Distribute the information associated with each leader (the first member of a group) to other members.

Moreover, by combining Lemmas 2.1 and 2.2, we get the following load balancing result that we need later. The proof is left to the reader.

Lemma 2.3 *Assume that we have a collection of at most m independent tasks whose sizes are given in an array $[t_1, t_2, \ldots, t_m]$. Let $\sum_{i=1}^{m} t_i \leq m$. Further, assume that each task can be executed in $O(\log t_i)$ time with $\lceil t_i / \log t_i \rceil$ processors. By using $\lceil m/\log m \rceil$ processors, all tasks can be completed in $O(\log m)$ time.*

Following the guidelines of [14], we represent sorted sets as 2-3 trees. A *2-3 tree* is a height-balanced tree in which each *internal node* has two or three children, and every path from the root to an *external node* is of the same length. The items of a sorted set X are stored in the external nodes in increasing order of keys. An internal node v stores

(1) a pointer to its parent and pointers to its children;

(2) the height of the subtree rooted by v, *i.e.*, the distance to the external nodes;

(3) the size of the subtree rooted by v, *i.e.*, the number of the external nodes in the subtree; and

(4) the key values k_1, k_2, and k_3 such that for every item x_1 in the first subtree of v, $key(x_1) \leq k_1$; for every item x_2 in the second subtree of v, $k_1 < key(x_2) \leq k_2$; and for every item x_3 in the third subtree of v (if it exists), $k_2 < key(x_3) \leq k_3$.

It should be pointed out that the algorithms to be developed can be generalized to handle level-linked 2-3 trees as well. For the sake of simplicity, this is not done here.

Many efficient algorithms are known for the manipulation of arrays. In order to use these as subroutines we need algorithms for changing the representation of a sorted set from a 2-3 tree to an array, and vice versa. The following simple results, given here without proofs, turn out to be useful (cf. [10, Exercise 2.32]).

Lemma 2.4 *Given a 2-3 tree storing m items in its external nodes, a sorted array of size m containing the same set of items can be constructed in $O(\log m)$ time and $O(m)$ space by using $\lceil m/\log m \rceil$ processors on an EREW PRAM.*

Lemma 2.5 *Given a sorted array of m items, a 2-3 tree representing the same set of items can be constructed in $O(\log m)$ time and $O(m)$ space by using $\lceil m/\log m \rceil$ processors on an EREW PRAM.*

For the basic properties of the 2-3 trees, we refer to any textbook on data structures and algorithms, see, *e.g.*, [1, Section 4.9]. The following two lemmas summarize the previous results on 2-3 trees relevant to us.

Lemma 2.6 *Let X, Y, and Z_i, $i = 1, \ldots, m$, be sorted sets represented as 2-3 trees. Further, let $n = |X| = |\bigcup_{i=1}^{m} Z_i|$, $m = |Y|$, and assume that $n \geq m$. The operations* SEARCH(X,Y), JOIN$([Z_1, Z_2, \ldots, Z_m])$, SPLIT$(X,Y)$, INTERSECTION$(X,Y)$, UNION$(X,Y)$, *and* DIFFERENCE$(X,Y)$ *can be executed in $O(m \log(n/m))$ time and $O(n)$ space on a unit-cost RAM. Moreover, this is the best possible performance that can be achieved by any comparison-based algorithm accomplicing the operation* SEARCH(X,Y), SPLIT(X,Y), INTERSECTION(X,Y),UNION(X,Y), *or* DIFFERENCE(X,Y).

Remark 2.7 Brown and Tarjan [4] proved the above resource bounds for the search and union operations. Moffat *et al.* [13] considered the join and split operations even though their implementations are based on level-linked 2-3 trees. Other operations can be implemented in an analogous manner. The proof of the information-theoretic lower bound $\Omega(m \log(n/m))$ for the union operation can be found in [4]. The other lower bounds are easily proved by using similar arguments. We shall postpone the presentation of these lower bounds to the full version of the present paper.

Paul *et al.* [14] presented how the operations mentioned in the previous lemma can be implemented efficiently in parallel. Their algorithms use an interesting pipelining technique that is surveyed in the book by JáJá [10, Section 2.5]. The next lemma sums up the results obtained in [14].

Lemma 2.8 *Let X, Y, and Z_i, $i = 1, \ldots, m$, be sorted sets represented as 2-3 trees. Further, let $n = |X| = |\bigcup_{i=1}^{m} Z_i|$, $m = |Y|$, and assume that $n \geq m$. The operations* SEARCH(X,Y), JOIN$([Z_1, Z_2, \ldots, Z_m])$, SPLIT$(X,Y)$, INTERSECTION$(X,Y)$, UNION$(X,Y)$, *and* DIFFERENCE$(X,Y)$ *can be executed in $O(\log n)$ time and $O(n + m \log n)$ space with m processors on an EREW PRAM.*

Proof. The implementations given in [14] for the operations SEARCH(X,Y), UNION(X,Y), and DIFFERENCE(X,Y) assume that the sorted set Y is stored in an array. As a preprocessing step, we convert the set Y represented as a 2-3 tree to an array in $O(\log m)$ time by using $\lceil m/\log m \rceil$ processors (cf. Lemma 2.4). Therefore, this conversion can also be executed within the resource bounds stated (cf. Lemma 2.1). Observe that the operations JOIN$([Z_1, Z_2, \ldots, Z_m])$

and $\textsc{split}(X, Y)$ were not considered explicitely in [14] but as a part of the deletion algorithm (for the details, see [14, Section 5]). In principle, the intersection operation is nothing but a special case of the search operation. Therefore, its implementation is left as an exercise for the reader. ■

The proof of the next lemma can be found from several sources, see, *e.g.*, [2, 3, 5, 8, 9, 11].

Lemma 2.9 *Two sorted arrays of respective size n and m, $n \geq m$, can be merged stably in $O(\log n)$ time and $O(n)$ space with $\lceil n/\log n \rceil$ processors on an EREW PRAM.*

Gavril [7] observed that, when m is much smaller than n, more efficient parallel merging is possible, if the output need not be an array. However, Gavril's algorithm requires concurrent reads. Our union algorithm running on an EREW PRAM can be used to improve this early result on merging (cf. Remark 3.3).

3 Efficient manipulation of sorted sets

In this section we present work-efficient parallel algorithms for the set manipulation problems introduced in Section 1. Our purpose is to prove the following main result of this paper.

Theorem 3.1 *Let X, Y, and Z_i, $i = 1, \ldots, m$, be sorted sets represented as 2-3 trees. Further, let $n = |X| = |\bigcup_{i=1}^{m} Z_i|$, $m = |Y|$, and assume that $n \geq m$. The operations* $\textsc{search}(X, Y)$, $\textsc{join}([Z_1, Z_2, \ldots, Z_m])$, $\textsc{split}(X, Y)$, $\textsc{intersection}(X, Y)$, $\textsc{union}(X, Y)$, *and* $\textsc{difference}(X, Y)$ *can be executed in $O(\log n)$ time and $O(n)$ space with $\lceil (m \log(n/m))/\log n \rceil$ processors on an EREW PRAM.*

Remark 3.2 In the intersection, union, and difference operations the set X can also be smaller than the set Y. Therefore, for these operations the resource bounds stated are valid even if we denote $n = \max\{|X|, |Y|\}$ and $m = \min\{|X|, |Y|\}$.

The proof of Theorem 3.1 is given in different (sub)sections. The search, join, and split operations turn out to be fundamental. The algorithms performing these are presented in Subsections 3.1, 3.2, and 3.3, respectively. These fundamental operations are then used to implement the other standard set operations. The algorithms performing the intersection, union, and difference operations are given in Subsections 3.4, 3.5, and 3.6, respectively. The space requirement of the algorithms is analysed in Section 4.

3.1 Searches

Let $X = \langle x_1, x_2, \ldots, x_n \rangle$ and $Y = \langle y_1, y_2, \ldots, y_m \rangle$ be two sorted sets, $n \geq m$. Assume that X and Y are represented by 2-3 trees T_X and T_Y, respectively. Let us suppose that we want to compute, for each item y_i, the index j_i such that $key(x_{j_i})$ is minimal with $key(y_i) \leq key(x_{j_i})$. Next we show how this search can be accomplished efficiently.

The standard sequential search starts from the root and traces down the tree until an external node is encountered. Basically, the algorithm given by Paul *et al.* [14] is a parallellization of this technique. The drawback of this method is that it will require $\Theta(m \log n)$ work. However, as observed already by Brown and Tarjan [4], most search paths share many nodes near the top of the tree. By traversing the nodes on the top of the tree only once, a more efficient algorithm is obtained. Let us now consider how this idea can be implemented on an EREW PRAM.

We use $top(T_X, h)$ to denote the tree that is obtained from the tree T_X by cutting off all nodes whose depth is larger than h. Let T denote the tree $top(T_X, \lceil \log m \rceil)$. The external nodes of T are all nodes of T_X whose distance from the root is $\lceil \log m \rceil$. For an external node of T its key value is defined to be k_3 or k_2, if k_3 does not exist. Now we use Lemma 2.4 to move the external nodes of the trees T and T_Y to the arrays A and B, respectively. (Observe that the size of B is $O(m)$.) Then we merge the sorted arrays B and A (note the order) to an array C by using the algorithm of Lemma 2.9. The nodes of T partition the array B into at most m subarrays $B_1, B_2, \ldots, B_m$ (some possibly empty). Now the computation can be divided into m disjoint tasks that are executed in parallel. In the ith task we want to search for the proper positions of the items in B_i from the subtree rooted by the node following the block B_i in C. To execute these tasks we can use the algorithm of Lemma 2.8. The only precondition, that has to be fulfilled in order to use the algorithm, is that the starting position of B_i and the size of B_i should be known by each of the executing processors. It is easy to provide m copies of these by using the algorithm of Lemma 2.2. Then this information can be accessed without read conflicts. The parallel processes write their results into a separate array of size m, which is the output of the whole algorithm.

It is not difficult to see that all array manipulations in the above algorithm can be performed in $O(\log m)$ time with $\lceil m/\log m \rceil$ processors. The subtree traversals can be executed in $O(\log(n/m))$ time by using m processors. Accoding to Lemma 2.1, these computations can be slowed down so that with $\lceil (m \log(n/m))/\log n \rceil$ processors the running time is $O(\log n)$.

3.2 Joins

Let $Z_1, Z_2, \ldots, Z_m$ ($Z_1 < Z_2 < \ldots < Z_m$) be the collection of disjoint sorted sets that are to be joined. Further, let $X = \bigcup_{i=1}^{m} Z_i$ and $n = |X|$. We assume that the sorted sets Z_i are represented by 2-3 trees T_{Z_i}. The basic idea of our algorithm is similar to that used in searching. When creating a tree that represents X, we use the existing parallel algorithm for nodes that will end up deep in the tree, whereas the top of the tree is handled by using another method.

First, we should remove empty sets from the collection. This compaction operation is easily implemented by a prefix-sum algorithm (Lemma 2.2). Then we have to compute n. For this purpose we need a summing algorithm (cf. Lemma 2.2); the algorithm gets the sizes of the subsets from the roots. Let $h = \lceil \log(n/m) \rceil$. We say that a tree is *high* if its height is larger than h. Now the high trees are dashed into smaller pieces. If the height of a tree T is h', we remove from T at least $2^{h'-h}$ and at most $3^{h'-h}$ nodes. Our aim is to create subtrees whose height is precisely h. The tree dashing is done by using the algorithm of Lemma 2.4 and the load balancing is done as indicated in Lemma 2.3. After this the total number of trees is still $O(m)$. It is an easy matter to store the roots of the trees in an array in sorted order (utilize the original order and use Lemma 2.2 to get the new trees into their proper places). Call the resulting subsets $Z'_1, Z'_2, \ldots, Z'_{m'}$.

Assume that we have m' processors available. Now we can use these to join the trees of height at most h as done in the join algorithm of Lemma 2.8. In this algorithm (for details, see [14]) at time step $3j$, $j = 1, 2, \ldots, \lceil \log m' \rceil$, the processor P_k, $k \in \{2^{j-1}, 2^{j-1} + 2^j, \ldots\}$, starts to join the sets $\bigcup_{i=k-2^{j-1}-1}^{k} Z'_i$ and $\bigcup_{i=k+1}^{\min\{k+2^{j-1}, m'\}} Z'_i$. That is, each processor joins sequentially two sets together in a bottom-up manner. Therefore, each processor should locate the rightmost external node in the left subtree and the leftmost external node in the right subtree. This are located before starting the join process. Since no subset is empty, processor P_k finds the nodes required from the trees Z'_k and Z'_{k+1}, respectively. Each processor follows the join algorithm until it reaches the level h in its update. At this point the processor *stops* its computation. At the moment when the computation was stopped each processor P_k can hold at most one subtree. The root of this tree is stored in the kth location of some new array.

Now we have two collections of trees. The old collection can have some trees of height h that are not yet joined with any other tree. The new collection contains some new trees but the height of these is also h. By using Lemma 2.2 we can again remove the empty trees from these collections. The collections are then merged together by the algorithm of Lemma 2.9; k_1 of each root can be used as a key value in this merging. Now it is a simple matter to create a balanced 2-3 tree above the subtrees that are of equal height. Here we can use an algorithm similar to that given in Lemma 2.5.

Let us next analyse the performance of the algorithm. According to the lemmas given in Section 2 all array manipulations are done in $O(\log m)$ time with $\lceil m/\log m \rceil$ processors. The parallel join tasks require $O(\log(n/m))$ time with m processors. Again these computations can be slowed down from which the claimed performance follows.

3.3 Splits

Let X be a sorted set and let $Y = \langle y_1, y_2, \ldots, y_m \rangle$ be a subset of X. Assume that X and Y are represented by 2-3 tree T_X and T_Y, respectively. Now we want to compute a collection of maximal subsets $Z_0, Z_1, \ldots, Z_{m+1}$ of X such that $Z_0 < \langle y_1 \rangle < Z_1 < \langle y_2 \rangle < \ldots < \langle y_m \rangle < Z_{m+1}$. The algorithm given next is based on an idea similar to that used already in Subsections 3.1 and 3.2.

At first we create an array A containing the nodes of T_X whose distance from the root is $\lceil \log m \rceil$. We move also the items of T_Y to an array B. This is done as in Subsection 3.1 in connection with searches. Then we merge the two arrays. The items of A partition the array B into subarrays $B_1, B_2, \ldots, B_m$ (some possibly empty). The items of B_i are located in the subtree rooted by the node v_i lying just after the block B_i. Now the subtrees rooted by v_i are split by using the split operation of Lemma 2.8.

As a result of the computation so far each of the subtrees is split into a collection of at most $|B_i| + 1$ new trees. Total number of trees is still $O(m)$. By utilizing the original order, it is a simple matter to order the trees in increasing order determined by the first key value in the root nodes (use Lemma 2.2). This collection can still have a successive subtrees that have to be joined together. The height of the subtrees to be joined is however the same, h. Therefore, these can be joined as in Subsection 3.2. To perform this piecewise computation optimally we use Lemma 2.3.

The analysis of this algorithm is similar to those done already in Subsections 3.1 and 3.2. All array manipulation that is needed for handling the top part of the trees is done in $O(\log m)$ with $\lceil m/\log m \rceil$ processors. The parallel splittings performed in the lower levels of the tree takes $O(\log(n/m))$ time with m processors. The performance claimed in Theorem 3.1 follows from Lemma 2.1.

3.4 Intersections

Let X and Y be two sorted sets, whose intersection is to be computed. Further, let us use the following shorthand notations $n = \max\{|X|, |Y|\}$ and $m = \min\{|X|, |Y|\}$. One should observe that the intersection operation is not symmetric since, according to the definition of $X \cap Y$, those items of X are returned that have a key equal to that of some item in Y. We want that the algorithm runs equally well in the case that X is larger than Y as in the case that X is smaller than Y. Therefore, we divide the description of the algorithm into two parts depending on the sizes of the sets.

Case 1. $|X| \geq |Y|$. Let us first perform the operation SEARCH(X, Y). Call the resulting array A. Then we transform the sorted set Y to a sorted array B by using the algorithm of Lemma 2.4. The arrays A and B are then merged stably using Lemma 2.9. In the result of this merge the items with equal keys are consecutive; an X-item is followed by an Y-item. Therefore, we check for each Y-item y in this array whether its predecessor has a key equal to $key(y)$. If this is the case the Y-item is marked and the preceeding X-item is associated with it. Finally, the marked Y-items are compacted to the beginning of the array (use Lemma 2.2), the X-item are moved to another array, and then this array is transformed to a tree (use Lemma 2.5). The resulting set is returned as the answer of the whole algorithm.

Case 2. $|X| < |Y|$. Since X is smaller than Y, we perform the operation SEARCH(Y, X) to locate those items in Y that can have a counterpart in X. Call the resulting array B. Then the items of X are moved to an array A. Hereafter, the computation proceeds as in Case 1.

The most costly part of this algorithm is the search operation performed. All other parts of the algorithm can be executed in $O(\log m)$ time by using $\lceil m/\log m \rceil$ processors, or in $O(\log n)$ time by using $\lceil m/\log n \rceil$ processors. Therefore, the claimed resource bounds follow from the corresponding bounds proved for the search operation.

3.5 Unions

Let X and Y be two sorted sets, whose union is to be computed. Assume that 2-3 trees T_X and T_Y are used for representing X and Y, respectively. Let n denote the size of X and m the size of Y. Without loss of generality, we can assume that $X \cap Y$ is empty, since otherwise we can start the computation by performing the intersection operation $X \cap Y$. Moreover, we can assume that $n \geq m$. If this is not the case, one processor can read the sizes of the sets from the roots of T_X and T_Y and change the order of the sets if necessary.

Let us assume that $Y = \langle y_1, y_2, \ldots, y_m \rangle$. We perform first the operation SEARCH(X, Y). As a result we get an array that we call A. Then we remove from A items with duplicate keys by using Lemma 2.2. Call the array without duplicates A' and let Y' be the corresponding set obtained by applying Lemma 2.5. To compute $X \cup Y$, we partition first X into at most $2m + 1$ disjoint sets $Z_1, \ldots, Z_{2m+1}$ by SPLIT(X, Y'). Next we create singleton sets $\langle y_i \rangle$, for each $j = 1, 2, \ldots, m$. Then the tree arrays $[Z_1, \ldots, Z_{2m+1}]$ and $[\langle y_1 \rangle, \langle y_2 \rangle, \ldots, \langle y_m \rangle]$ are merged to get the trees in increasing order of the key values of the roots. Finally, the collection of at most $3m + 1$ trees is joined by using the algorithm described in Subsection 3.2.

The complexity of this algorithm is dominated by that of performing the intersection, split, and join operations. The array manipulations are easily done in $O(\log m)$ time by using $\lceil m/\log m \rceil$ processors, or if slowed down, in $O(\log n)$ time by using $\lceil m/\log n \rceil$ processors. Therefore, the claimed resource bounds for the union operation follow from the corresponding bounds proved for the operations described earlier.

Remark 3.3 It is not diffucult to see that the basic method given in this subsection can also be used to merge stably two sorted multisets within the same resource bounds.

3.6 Differences

Let X and Y be two sorted sets and assume that we want to compute the difference $X \setminus Y$. Further, let n denote the size of X and m the size of Y. Without loss of generality, we can assume that $n \geq m$. If this is not the case, we can first use the intersection routine described in Subsection 3.4 to compute $X \cap Y$ and then remove the items of $X \cap Y$ from X. Therefore, we can also assume that, for every $y \in Y$, there is an item $x \in X$ such that $key(x) = key(y)$.

The algorithm for computing $X \setminus Y$ is as follows. First, we partition the set X into at most $2m + 1$ sets by performing SPLIT(X, Y). Second, we remove every second set from the obtained collection since these sets contain items that have keys equal to those of the items in Y. Finally, we join the remaining (at most $m + 1$) sets and return the set obtained as the result of the whole algorithm.

The performance of this algorithm is determined by that of the intersection, split, and join operations. The other parts of the algorithm are easily seen to require $O(\log m)$ time with $\lceil m/\log m \rceil$ processors, or $O(\log n)$ time with $\lceil m/\log n \rceil$ processors. Hence, the claimed resource bounds follow from the corresponding bounds derived already for the other operations.

4 Memory management

The storage allocation in our algorithms can be performed in the same way as done by Paul *et al.* in their algorithms [14, Section 6]. We assume that the sorted sets manipulated are stored at the beginning of the shared memory. Let us call this area the *set storage* and call the area just after it the *free storage*. When executing the various operations on sorted sets, we need temporarily a constant number of arrays (of size m), for which space can be reserved by a single processor from the free storage. After a set operation, the allocated storage is again freed. Therefore, the manipulation of these arrays will not be a bottleneck in the algorithms.

In the manipulation of the sorted sets, implemented as 2-3 trees, new nodes are created and old nodes are released. A separate storage allocation scheme could be designed for the internal and external nodes of 2-3 trees representing the sets currently in the set storage. For the sake of simplicity, we however assume that the records storing both internal and external nodes are of equal size. Since the running time of the update algorithms is $O(\log n)$, each processor can create and release $O(\log n)$ nodes. Therefore, we reserve for each processor a private area of size $O(\log n)$ from the free storage at the beginning of the computation. This private bookkeeping area consists of two tables: the *insertion table* is used for storing new nodes and the *deletion table* is used for collecting information about released nodes. Since the number of processors used by our algorithms is only $\lceil (m \log(n/m))/\log n \rceil$, which is always less than $\lceil n/\log n \rceil$, these temporal tables will never require more than $O(n)$ space in total.

As a postprocessing step of the set manipulation algorithms the data in insertion tables is compacted and the storage space of the released nodes is filled with meaningful information. Here it is important that each node of a 2-3 tree is pointed only by a constant number of other nodes. Therefore, we can move a node from one place to another, if we also update the pointers in the nodes pointing to the moved node. The nodes in the insertions tables can be moved to the end of the set storage in $O(\log n)$ time with $\lceil (m \log(n/m))/\log n \rceil$ processors (cf. Lemma 2.2). The cursor pointing to the beginning of the free storage must be updated accordingly. In an analogous manner, some of the nodes at the end of the set storage can be moved to the holes created by the released nodes (for details, see [14, Section 6]).

Finally, we would like to point out that the theory we have developed generalizes naturally to sorted multisets that allow several items to have equal keys. Then, however, the storage allocation can be problematic. For example, consider the problem of computing $X \setminus Y$ for two multisets X and Y, that is, we want to remove those items of X that have a counterpart in Y. Now it might be necessary to remove more than $|Y|$ items from X, since X can contain many items with keys equal to an item in Y. Therefore, the time we need for filling the holes created could be larger than the actual time needed for the pointer manipulation. In the multiset case it would be a good idea to design a garbage collector that fills the empty holes created during the computation. The design of a parallel garbage collector is an interesting problem of its own.

Acknowledgements

The author would like to thank Alistair Moffat and Erkki Mäkinen for inspiring discussions.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[2] R.J. Anderson, E.W. Mayr, M.K. Warmuth: Parallel approximation algorithms for bin packing. *Information and Computation* **82** (1989) 262–277.

[3] G. Bilardi, A. Nicolau: Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing* **18** (1989) 216–228.

[4] M.R. Brown, R.E. Tarjan: A fast merging algorithm. *Journal of the ACM* **26** (1979) 211–226.

[5] E. Dekel, I. Ozsvath: Parallel external merging. *Journal of Parallel and Distributed Computing* **6** (1989) 623–635.

[6] D. Eppstein, Z. Galil: Parallel algorithmic techniques for combinatorial computation. *Annual Reviews in Computer Science* **3** (1988) 233–283.

[7] F. Gavril: Merging with parallel processors. *Communications of the ACM* **18** (1975) 588–591.

[8] X. Guan, M.A. Langston: Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers* **40** (1991) 596–602.

[9] T. Hagerup, C. Rüb: Optimal merging and sorting on the EREW PRAM. *Information Processing Letters* **33** (1989) 181–185.

[10] J. JÁJÁ: *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

[11] J. KATAJAINEN, C. LEVCOPOULOS, O. PETERSSON: Space-efficient parallel merging. *Informatique théorique et Applications* **27** (1993) 295–310.

[12] C.P. KRUSKAL, L. RUDOLPH, M. SNIR: The power of parallel prefix. *IEEE Transactions on Computers* **C-34** (1985) 965–968.

[13] A. MOFFAT, O. PETERSSON, N.C. WORMALD: Sorting and/by merging finger trees. In *Proc. of the 3rd International Symposium on Algorithms and Computation,* Lecture Notes in Computer Science **650**, Springer-Verlag, 1992, pp. 499–508.

[14] W. PAUL, U. VISHKIN, H. WAGENER: Parallel computation on 2-3 trees. *R.A.I.R.O. Informatique théorique* **17** (1983) 397–404.

[15] R.E. TARJAN: *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.