Fast Set Operations Using Treaps

Guy E. Blelloch   Margaret Reid-Miller

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
{blelloch,mrmiller}@cs.cmu.edu

Abstract

We present parallel algorithms for union, intersection and difference on ordered sets using random balanced binary trees (treaps [26]). For two sets of size n and m ($m \leq n$) the algorithms run in expected $O(m \lg(n/m))$ work and $O(\lg n)$ depth (parallel time) on an EREW PRAM with scan operations (implying $O(\lg^2 n)$ depth on a plain EREW PRAM). As with the sequential algorithms on treaps for insertion and deletion, the main advantage of our algorithms are their simplicity. In fact, our algorithms for set operations seem simpler than previous sequential algorithms with the same work bounds, and might therefore also be useful in a sequential context. To analyze the effectiveness of the algorithms we implemented both sequential and parallel versions of the algorithms and ran several experiments on them. Our parallel implementation uses the Cilk [5] shared memory runtime system on a 16 processor SGI Power Challenge and a 6 processor Sun Ultra Enterprise 3000. It shows reasonable speedup: 6.3 to 6.8 speedup on 8 processors of the SGI, and 4.1 to 4.4 speedup on 5 processors of the Sun.

1 Introduction

Balanced trees provide a wide variety of low-cost operations on ordered sets and dynamic dictionaries. Of the many types of balanced trees that have been developed over the years, treaps [26] have the advantage of both being simple and general—in addition to insertion and deletion they easily support efficient finger searching, joining, and splitting. Furthermore, by using appropriate hash functions they require no balance information to be stored at the nodes. Treaps, however, have only been studied in the context of sequential algorithms, and there has been little study of their performance.

To appear in the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98), 28 June–2 July 1998, Puerto Vallarta, Mexico

In this paper we extend previous work on treaps by describing and analyzing parallel algorithms on treaps and by presenting experimental results that demonstrate their utility. We focus on the aggregate set operations *intersection*, *union*, and *difference* since, among other applications, these operations play an important role in databases queries and index searching [30, 19]. The techniques we describe can also be applied to searching, inserting and deleting multiple elements from a dictionary in parallel. Although we use two of the same building blocks used in the sequential algorithms (split and join), the design and analysis of our algorithms is quite different. The algorithms we present have the following properties, which together make them attractive from both a theoretical and practical point of view.

- For two sets of size n and m with $m \leq n$ our algorithms for union, intersection, and difference run in expected $O(m \lg(n/m))$ serial time or parallel work. This is optimal.
- Our algorithms are significantly simpler than previous parallel and serial algorithms with matching work bounds. We therefore expect that our algorithm could be useful even in a sequential setting. The paper includes the full C code for our algorithms.
- The parallelism in our algorithms comes purely from their divide-and-conquer nature. The algorithms are therefore easy to implement in parallel and well suited for asynchronous execution. We have implemented the parallel versions of the algorithms in Cilk [5] with only minor changes to the sequential C code.
- Our algorithms are fully persistent—they create the resulting set while leaving the input sets untouched.[1] Such persistence is important in database and indexing applications.
- Our algorithms are efficient when the results are from interleaving blocks of the ordered input sets. For example, a variant of our union algorithm requires expected serial time (or parallel work) that is only logarithmic in the input sizes when the inputs have only a constant number of blocks.

[1] Note that just copying the inputs does not work since this would violate the $O(m \lg(n/m))$ work bounds—it would require $O(n + m)$ work.

- The algorithms use the same representation of treaps as Seidel and Aragon [26] so that all their algorithms can be used on the same data without modification.

Treaps are randomized search trees in which each node in the tree has a key and an associated random priority. The nodes are organized so that the keys appear in in-order and the priorities appear in heap-order. Seidel and Aragon showed that insertion, deletion, searching, splitting, and joining can all be implemented on treaps of size n in $O(\lg n)$ expected time, and that given a finger in a treap (a pointer to one of the nodes), one can find the key that is d away in the sorted order in $O(\lg d)$ expected time (although this requires extra parent pointers). They also showed that one need not store the priorities at the nodes, but instead can generate them as needed using a hashing function. The simplicity of the algorithms led them to conjecture that the algorithms should be fast in practice. Although they did not present experimental results, we have run experiments that compared treaps to splay trees [27], red-black trees, and skip-lists [22] and the results show that treaps have around the same performance as these other fast data structures. These results are briefly presented in Section 3.1.

Our interest is in developing fast parallel algorithms for set operations. Set operations are used extensively for index searching—each term (word) can be represented as a set of the "documents" it appears in and searches on logical conjunctions of terms are implemented as set operations (intersection for **and**, union for **or**, and difference for **and-not**) [30, 19]. Most web search engines use this technique. The set union operation is also closely related to merging—it is simply a merge with the duplicates removed. It is well known that for two sets of size m and n ($m \leq n$) merging, union, intersection and difference can be implemented sequentially in $O(m \lg(n/m))$ time [6]. In applications in which the sets are of varying and different sizes, this bound is much better than using a regular linear time merge ($O(n+m)$ time) or inserting the smaller set into the larger one at a time using a balanced tree ($O(m \lg n)$ time). Previous sequential algorithms that achieve these bounds, however, are reasonably messy [6, 7]. The only parallel version we know of that claims to achieve these work bounds [15] is much more complicated.

In regards to treaps, Seidel and Aragon did not directly consider merging or set operations, but it is straightforward to use their finger search to implement these operations within the optimal time bounds, expected case. In particular if p_i is the position of element i of set A in set B, the expected time of inserting all of A into B by inserting one at a time using the previous element as a finger is $\sum_{i=1}^{|A|}(1+\lg(1+p_{i-1}-p_i))$. With $|A|=m$, $|B|=n$, and $m \leq n$ this sum is bounded by $O(m \lg(n/m))$. Although the algorithm is probably simpler than previous techniques, it requires parent pointers and does not parallelize.

In this paper we describe direct algorithms for union, intersection and difference on ordered sets using treaps. The algorithms run in optimal $O(m \lg(n/m))$ work and in $O(\lg n)$ depth (parallel time) on an EREW PRAM with unit-time scan operations (used for load balancing)—all expected case. This bound is based on automated pipelining [3]—without pipelining the algorithms run in $O(\lg^2 n)$ depth. As with the sequential algorithms on treaps, the expectation is over possible random priorities, such that the bounds do not depend on the key values. The algorithms are very simple (although the analysis of their bounds is not) and can be applied either sequentially or in parallel. All the algorithms have a similar divide-and-conquer structure and require no extensions to the basic treap data structure. We also show that for two ordered sets of size n and m and if k is the minimum number of blocks in a partitioning of the first set that contains no elements of the second, a variant of our union algorithm requires only $O(k \lg((n+m)/k)$ expected work. This is optimal with respect to the measure [8].

To analyze the effectiveness of the algorithms in practice we ran several experiments. We were interested in various properties including how well treaps compare sequentially with other balanced trees such as red-black trees, how well our algorithms perform for various overlaps in the keys, and how well the algorithms parallelize. The serial experiments show that treaps are competitive with splay trees, skip lists, and red-black trees. They also show that the algorithms perform well with small overlap in the keys. The parallel implementation was coded in Cilk [5], a parallel extension of C, and run on a Sun Ultra Enterprise 3000 and a SGI Power Challenge. The algorithms achieve speedups of between 4.1 and 4.4 on 5 processors of the Sun and between 6.3 and 6.8 on 8 processors of the SGI. We feel that this is reasonably good considering the high memory bandwidth and irregular access pattern required by the algorithm.

Related Work

Merging and the related set operations (union, intersection and difference) have been well studied. Merging two ordered sets N and M of size n and m requires at least $\lceil \lg \binom{n+m}{n} \rceil$ comparisons in the worse case since an algorithm needs to distinguish between the $\binom{n+m}{n}$ possible placements of the n keys of N in the result. Without loss of generality we will henceforth assume $m \leq n$, in which case $\lceil \lg \binom{n+m}{n} \rceil = \theta(m \lg(n/m))$. Hwang and Lin [14, 18] described an algorithm that matches this lower bound, but the algorithm assumes the input sets are in arrays and only returns cross pointers between the arrays. To rearrange the data into a single ordered output array requires an additional $O(n+m)$ steps. Brown and Tarjan gave the first $\theta(m \lg(n/m))$ time algorithm that outputs the result in the same format as the inputs [6]. Their algorithm was based on AVL-trees and they later showed a variant based on finger searching in 2-3 trees [7]. The same bounds can also be achieved with skip-lists [21].

The lower bound given above makes no assumptions about the input. When using a measure of the "easiness" of the input the bounds can be improved. In particular, suppose set A is divided into blocks of elements $A_1, A_2, \ldots, A_k$ and set B is divided into blocks of elements $B_1, B_2, \ldots, B_l$ such that the merged set is an alternation of these blocks from A and B ($k = l \pm 1$). Such inputs can be merged in $O(k \lg((n+m)/k))$ time [8, 21]. In fact any data structure that takes $O(\lg k)$ time for a split on the k^{th} element of an ordered set, or to append (join) k elements to an ordered set can be used to achieve these bounds. Pugh used this approach for skip lists [21] and although not discussed directly, the approach can also be applied to treaps when using the "fast" versions of splits and joins. However, as with finger searches, these "fast" versions require parent pointers.

In the parallel setting previous work has focused either on merging algorithms that take $O(n+m)$ work and are optimal when the two sets have nearly equal sizes or on multi-insertion algorithms that take $O(m \lg n)$ work and are optimal when the input values to be inserted are not presorted.

Anderson et al. [1], Dekel and Ozsvath [9], Hagerup and Rüb [12], and Varman et al. [29] provide $O(n/p + \lg n)$ time EREW PRAM algorithms for merging. Guan and Langston [11] give the first time-space optimal algorithm that takes $O(n/p + \lg n)$ time and O(1) extra space on an EREW PRAM. Katajainen et al. [16] gives a simpler algorithm with the same time and space bounds for the EREW and an optimal space-efficient $O(n/p + \lg \lg m)$ time and $O(1)$ space algorithm for the CREW PRAM.

Paul et al. provide EREW PRAM search, insertion, and deletion algorithms for 2-3 trees [20], and Highan and Schenk have extended these results to B-trees [13]. Ranade [23] gives algorithms for processing least-upper-bound queries and insertion on distributed memory networks. Bäumker and Dittrich [2] give algorithms for search, insertion, and deletion into BB*(a) trees for the BSP* model that are 1-optimal and 2-optimal. All these algorithms require $O(m \lg n)$ work and appear to have large constants.

In 1975 Gravil gave the first CREW PRAM merge algorithm that requires only $O(m \lg(n/m))$ comparisons. However, as with Hwang and Lin's serial algorithm, it requires an additional $O(n+m)$ operations to return the sorted merged results in an array. Katajainen describes EREW PRAM algorithms for union, intersection and difference that use the same input and output representations [15]. The algorithms are an extension of Paul *et al.'s* 2-3 tree algorithms and he claims they run in optimal $O(\lg n + \lg m)$ depth and $O(m \lg(n/m))$ work. The algorithms as described, however, do not actually meet these bounds since the analysis incorrectly assumes a 2-3 tree of depth $\lg m$ has $O(m)$ leaves. It may be possible to modify the algorithms to meet the bound and the general approach seems correct.

2 Treaps

Treaps use randomization to maintain balance in dynamically changing search trees. Each node in the tree has an associated *key* and a random *priority*. The data are stored in the internal nodes of the tree so that the tree is in in-order with respect to the keys and in heap-order with respect to the priorities. That is, for any node, x, all nodes in the left subtree of x have keys less than x's key and all nodes in the right subtree of x have keys greater than x's key (in-order), and all ancestors of x have priorities greater than x's priority and all descendents have priorities less than x's (heap-order). For example, Figure 1a shows a treap where each (letter, number) pair represents a node and the letter is the key value and the number is the priority value. When both the keys and the priorities are unique, there is a unique treap for them, regardless of the order the nodes are added to or deleted from the treap.

The code shown in this section and used in our experiments implements persistent versions of the operations. That is, rather than modifying the input trees, the code makes copies of nodes that need to be modified using the function `new_node`. This function fills a new node with the key and priority data given in its arguments. All code in this section along with nonpersistent and parallel versions are available at `http://www.cs.cmu.edu/~scandal/treaps.html`.

2.1 Sequential algorithms

Seidel and Aragon showed how to perform many operations on treaps [26]. We quickly review the operations split and join, which we use to manipulating pairs of treaps.

$(L,\ x,\ G) = \mathbf{split}(T,\ key)$ Split T into two trees, L with key values less than *key* and G with key values greater than *key*. If T has a node x with key value equal to *key* then x is also returned.

$T = \mathbf{join}(T1,\ T2)$ Join $T1$ and $T2$ into a single tree T, where the largest key value in $T1$ is less than the smallest key value in $T2$.

Below we give the recursive top-down C code for persistent `split` and `join`. The `split` and `join` we use in our experiments are the slightly more efficient iterative versions. In addition, there are bottom-up algorithms that use rotations for these operations [26].

Split To split a tree rooted at r by key value a `split` follows the in-order access path with respect to the key value a until either it reaches a node with key value a or a leaf node. When the root key is less than a the root becomes the root of the "less-than" tree. Recursively, `split` splits the right child of the root by a, and then makes the resulting tree with keys less than a the new right child of the root and makes the resulting tree with keys greater then a the "greater-than" tree. Similarly, if the root key is greater than a `split` recursively splits the left child of the root. If the root key is equal to a `split` returns the root and the left and right children as the "less-than" and "greater-than" trees, respectively. Figure 1 shows the result of a split on a treap. The expected time to split two treaps into treaps of size n and m is $O(\lg n + \lg m)$ [26]. The following code returns the less and greater results by side effecting the first two argument. It returns an equal key, if present, as the result.

```
node split(node *less, node *gtr, node r, key_t key)
{
  node root;
  if (r == NULL) {*less = *gtr = NULL; return NULL;}

  root = new_node(r->key, r->priority);
  if (r->key < key) {
     *less = root;
     return split(&(root->right), gtr, r->right, key);
  } else if (r->key > key) {
     *gtr = root;
     return split(less, &(root->left), r->left, key);
  } else {
     *less = r->left;
     *gtr = r->right;
     return root;
  }
}
```

Join To join two treaps $T1$ with keys less than a and $T2$ with keys greater than a `join` traverses the right spine of $T1$ and the left spine of $T2$. A left (right) spine is defined recursively as the root plus the left (right) spine of the left (right) subtree. To maintain the heap order `join` interleaves pieces of the spines so that the priorities descend all the way to a leaf. The expected time to join two treaps of size n and m is $O(\lg n + \lg m)$ [26].

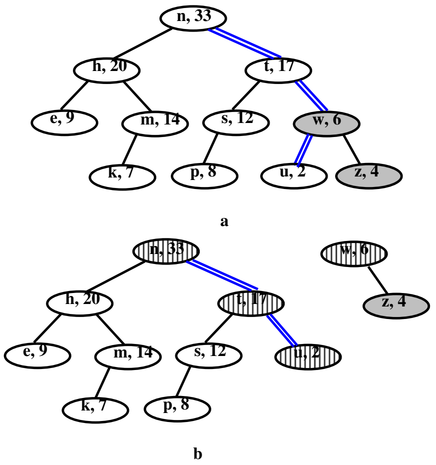

Figure 1: Figure **a** shows `split`'s input tree with each node's (key,priority) values. When splitting by the key v, the unshaded region becomes the less-than tree and the shaded region becomes the greater-than tree. The double links show the path `split` follows. Figure **b** shows the result trees; the striped nodes are the new nodes that the persistent version of `split` creates.

```
node join(node r1, node r2)
{
  node root;
  if (r1 == NULL) {return r2;}
  if (r2 == NULL) {return r1;}

  if (r1->priority < r2->priority) {
    root = new_node(r1->key, r1->priority);
    root->left = r1->left;
    root->right  = join(r1->right, r2);
  } else {
    root = new_node(r2->key, r2->priority);
    root->left  = join(r1, r2->left);
    root->right = r1->right;
  }
  return root;
}
```

2.2 Parallel Algorithms

In the parallel setting we view each treap as an ordered set of its keys and we consider the following operations:

T = **union**($T1$, $T2$) Find the union of treaps $T1$ and $T2$ to form a new treap T.

T = **intersect**($T1, T2$) Find the intersection of treaps $T1$ and $T2$ to form a new treap T.

T = **diff**($T1, T2$) Remove from $T1$ nodes that have the same key values as node in $T2$, returning its new root T.

All three algorithms have a similar divide-and-conquer structure in which we use the key from the larger priority root to split the tree with the smaller priority root, and then make two recursive calls (which can be parallel) on the values less and greater than the key. The algorithms differ in how they combine the results. The code we show below is for the sequential C versions. To make them parallel using Cilk one needs only put the `cilk` keyword before each function definition, the `spawn` keyword before each recursive call, and a `sync` after both of them. As discussed in 3.2 the versions used in the experiments also terminate the parallel calls at a given depth in the tree to reduce the overhead of spawning.

Union: To maintain the heap order, `union` makes r, the root with the largest priority, the root of the result treap. If the key of r is k then, to maintain the key in-order, `union` splits the other treap by k into a "less-than" tree with key values less than k and "greater-than" tree with key values greater than k, and possibly a duplicate node with a key equal to k. Then, recursively (and in parallel) it finds the union of the left child of r and the less-than tree and the union of the right child of r and the greater-than tree. The result of the two union operations become the left and right subtrees of r, respectively. The following C code implements the algorithm.

```
node union(node r1, node r2)
{
  node root, less, gtr, duplicate;

  if (r1 == NULL)  return r2;
  if (r2 == NULL)  return r1;

  if (r1->priority < r2->priority) swap(&r1, &r2);
  duplicate = split(&less, &gtr, r2, r1->key);

  root = new_node(r1->key, r1->priority);
  root->left  = union(r1->left, less);
  root->right = union(r1->right, gtr);
  return root;
}
```

Intersection: As with `union`, `intersection` starts by splitting the treap with the smaller priority root by k, the key of the root with the greater priority. It then finds the intersection of the two left subtrees, which have keys less than k, and the intersection of the two right subtrees, which have keys greater than k. If k appeared in both trees then these results become the left and right children of root used to split. Otherwise it returns the join of the two recursive call results.

```
node intersect(node r1, node r2)
{
  node root, less, gtr, left, right, duplicate;

  if ((r1 == NULL) || (r2 == NULL)) return NULL;

  if (r1->priority < r2->priority) swap(&r1, &r2);
  duplicate = split(&less, &gtr, r2, r1->key);

  left  = intersect(r1->left, less);
  right = intersect(r1->right, gtr);

  if (duplicate == NULL) {
    return destruct_join(left, right);
  } else {
    root = new_node(r1->key, r1->priority);
    root->left = left;
    root->right = right;
    return root;
  }
}
```

Notice that because the nodes returned by the intersection are all copies of input tree nodes, `intersect` can use a destructive version of join, one that modifies the nodes of the tree.

Difference: To find the difference of two treaps $T1$ and $T2$ `diff` splits the treap with the smaller priority root by k, the key of the root of the other treap. Then it finds the difference of the two left subtrees, which have keys less than k, and the difference of the two right subtrees, which have keys greater than k. Because difference is not symmetric `diff` considers two cases: when $T2$ is the subtrahend (the set specifying what should be removed) and when $T2$ is not, as specified by the boolean `r2_is_subtr`. If $T2$ is the subtrahend and it did not contain k, then it sets the left and right children of the root of $T1$ to the results of the recursive calls and returns this root. Otherwise it returns the join of the results of the recursive calls.

```
node diff(node r1, node r2, bool r2_is_subtr)
{
  node root, less, gtr, left, right, duplicate;

  if ((r1 == NULL) || (r2 == NULL))
    return r2_is_subtr ? r1 : r2;

  if (r1->priority < r2->priority) {
    r2_is_subtr = !r2_is_subtr;
    swap(&r1, &r2);
  }
  duplicate = split(&less, &gtr, r2, r1->key);

  left  = diff(r1->left, less, r2_is_subtr);
  right = diff(r1->right, gtr, r2_is_subtr);

  /* Keep r1 if no dupl. and subtracting r2 */
  if ((duplicate == NULL) && r2_is_subtr) {
    root = new_node(r1->key, r1->priority);
    root->left = left;
    root->right = right;
    return root;
  } else {                    /* Delete r1 */
    return join(left, right);
  }
}
```

2.3 Extensions

Using Fast Split and Join: The versions of split and join we use can split a treap into two treaps of size m and n or join two treaps of size m and n with $O(\lg \max(n, m))$ expected work. Seidel and Aragon also describe "fast" versions of split and join that use $O(\lg \min(n, m))$ expected work. These versions use parent pointers for quick access from the two ends of a set, and are similar to finger searching. The use of such fast versions of join and split does not effect our asymptotic work bounds assuming a general ordering of the input sets, but they do allow us to generate a parallel algorithm that is optimal with respect to the block measure. As described in the introduction if set A is divided into blocks of $A_1, A_2, \ldots, A_k$ and set B is divided into blocks of elements $B_1, B_2, \ldots, B_l$ such that the merged set is an alternation of these blocks from A and B, then A and B can be merged in $O(k \lg((n+m)/k))$ time. This bound also applies to union, although k is defined as the minimum number of blocks of A such that no value of B lies within a block. These times are optimal but previous algorithms are all sequential.

Section 2.5 shows that our parallel union algorithm achieves the same work bounds if it uses fast splits and joints. The modified algorithm also requires another change, at least in the version for which we prove bounds. In this change if `split(&less,&grt,r2,r1->key)` returns an empty tree in `less` then the algorithm executes the following instead of making the two recursive calls

```
km = minkey(r2);
dup = split(&ll,&rr,r1,km);
root = join(ll,union(rr,r2);
```

where `minkey(r2)` returns the minimum key in the set represented by `r2`. A symmetric case is used if `grt` is empty.

We note that although we use the fast splits and joins to prove the bounds, the algorithm with the slow versions still seem to work very well experimentally with respect to number blocks. Some experimental results are given in Section 3.

Nonpersistent versions: All the code we show and use in our experiments is fully persistent in that it does not modify the input treaps. Persistence is important in any application where the input sets or intermediate results might be reused in future set operations. Certainly in the application of index searching we do not want to destroy the sets specifying the documents when manipulating them. In other applications, such as updating a single dictionary kept as a treap, nonpersistent versions are adequate. In such applications we typically view one of the treaps as the treap to be modified and the other as the set of modifications to make. The only changes that need to be made to our code to make such nonpersistent versions is to have them modify the nodes rather than create new ones, and to explicitly delete nodes that are being removed.

The relative performance of the persistent and nonpersistent versions are discussed in Section 3.2. We note that our persistent code is not as space efficient as the more sophisticated method of Driscoll et al. [10], but their solution is much more complex and can only be applied to a tree of changes.

2.4 Analysis

In this section we first analyze the expected work to find the union of two treaps t_n and t_n of size n and m, respectively and $m \leq n$. The expected work to find intersection and difference of two treaps follows. Then we analyze the expected depth of these operations. The proof of some of the lemmas in this section and a more detailed discussion is given in [24].

Without loss of generality, assume that the key values for the two trees are $1, 2, 3, \ldots, n+m$. Let N and M be the sets of keys for t_n and t_m, respectively, such that $N \cup M = 1, 2, 3, \ldots, n+m$ and $N \cap M = \emptyset$. (If $N \cap M \neq \emptyset$ the expected work for union is less because the expected work for the split operation in union is less than when $N \cap M = \emptyset$.)

Since the priorities for the two treaps are chosen at random, arranging the keys so that their priorities are in decreasing order results in a random permutation $\sigma \in S_{n+m}$, where S_{n+m} is the set of all permutations on $n+m$ items. Along with N and M this permutation defines the result treap and the parallel work and depth required to find it. Therefore, we define $\mathbf{W}(N, M, \sigma)$ to be the work required to take the union of two *specific* treaps, which depends both on the interleaving of the key values in N and M and on the permutation σ defined by their priorities. We define $\mathbf{E}[\mathbf{W}(N,M)]$ to be the expected work to take the union of N and M averaged over all permutations (*i.e.*, $1/(n+m)!\sum_{\sigma\in S_{n+m}}\mathbf{W}(N,M,\sigma)$). Even when the sizes of N and M are fixed this expected work can depend significantly on the interleaving of N and M. We define $\mathbf{E}[\mathbf{W}(n,m)] = \max\{\mathbf{E}[\mathbf{W}(N,M)], |N| = n, |M| = m\}$. This is the worst case work over the interleavings and expected case over the permutations, and is what we are interested in.

The work for a permutation $\sigma = (a_1 = i, a_2, a_3, \ldots, a_{n+m})$ is the time to split one treap by i plus the work to take the union of the treaps with keys less than i and the union of the treaps with keys greater than i. We use the notation $N < i$ to indicate all keys in the set N which are less than i. Since it is equally likely that any i will have highest priority, we can write the following recurrence for the expected work for a given N and M averaged over all permutations

$$\begin{aligned}(n+m)\mathbf{E}[\mathbf{W}(N,M)] = \\ &\sum_{i=1}^{n+m}(\mathbf{E}[\mathbf{W}(N<i,M<i)] + \mathbf{E}[\mathbf{W}(N>i,M>i)]) + \\ &\sum_{i\in N}\mathbf{E}[\mathbf{T}_{split}(M,i)] + \sum_{i\in M}\mathbf{E}[\mathbf{T}_{split}(N,i)] + (n+m)d,\end{aligned}$$

where d is a constant. From [26] we know that $\mathbf{E}[\mathbf{T}_{split}(N,i)] = O(\lg n)$. These lead to the following lemma, proven in [24].

Lemma 2.1 *The expected work to take the union of two treaps of size n and m is bound by*

$$\begin{aligned}(n+m)\mathbf{E}[\mathbf{W}(n,m)] &\le (n+m)d + 2\mathbf{W}(0,0) \\ &+ \sum_{i=1}^{n+m-1}\max_{p_i}\{\mathbf{E}[\mathbf{W}(p_i, i-p_i)] + \\ &\qquad \mathbf{E}[\mathbf{W}(n-p_i, m-i+p_i)]\} \\ &+ nO(\lg m) + mO(\lg n), \qquad (1)\end{aligned}$$

where $0 \le p_i \le n$, $0 \le i - p_i \le m$, and $p_i \le p_{i+1}$.

Now we apply induction on the recurrence to show the bound on the expected work. If we assume $\mathbf{E}[\mathbf{W}(n,m)] = a\lg\binom{n+m}{n} - b\lg(n+m)$ and $\mathbf{W}(n,0) = \mathbf{W}(0,m) = c$, where a, b, and c are constants, and substitute in Equation 1 we get

$$\begin{aligned}(n+m)\mathbf{E}[\mathbf{W}(n,m)] &\le (n+m)d + 2c \\ &+ a\sum_{i=1}^{n+m-1}\max_{p_i}\left\{\lg\binom{i}{p_i} + \lg\binom{n+m-i}{n-p_i}\right\} \\ &- b\sum_{i=1}^{n+m-1}\{\lg i + \lg(n+m-i)\} \\ &+ nO(\lg m) + mO(\lg n). \qquad (2)\end{aligned}$$

Consider the expression containing the maximum. First we find the integral p_i values that maximizes the sum with the a constant preceding it. Then we will place an upper bound on this expression by using a nonintegral approximations to p_i and Sterling's approximation for factorials.

Lemma 2.2 *The expression $\lg\binom{i}{p_i} + \lg\binom{n+m-i}{n-p_i}$ is maximized when $p_i = \hat{p}_i$, where $\hat{p}_i = \left\lfloor\frac{(n+1)(i+1)}{n+m+2}\right\rfloor$.*

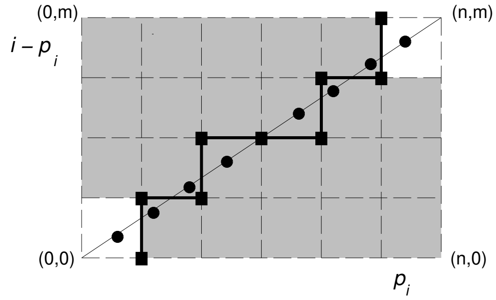

Figure 2: The shaded region is the region of possible values for $(p_i, i-p_i)$. The points (squares) on step function maximizes $\binom{i}{p_i} + \binom{n+m-i}{n-p_i}$. The points (circles) on the straight line is the continuous approximation to the points on the step function.

Lemma 2.3 *When p_i is an integer for all i and $0 \le p_i \le n$, $0 \le i - p_i \le m$, and $p_i \le p_{i+1}$*

$$\begin{aligned}a\sum_{i=1}^{n+m-1}\max_{p_i}\left\{\lg\binom{i}{p_i} + \lg\binom{n+m-i}{n-p_i}\right\} \le \\ a(n+m)\left\{\lg\binom{n+m}{n} + 2\lg n + 5\right\}. \qquad (3)\end{aligned}$$

Proof. (*Sketch*) Because floor values are hard to work with we use $\tilde{p}_i = ni/(n+m)$ as a continuous approximation to $\hat{p}_i$. Figure 2 graphically shows the relationship between the $\tilde{p}_i$, which lies on a straight line between $(0,0)$ and (n,m), and $\hat{p}_i$, which is on a step function that follows this line. In the following expressions we use $\binom{i}{p}$ to denote $i!/\Gamma(p+1)\Gamma(i-p+1)$, when p is not an integer. Then

$$\lg\left[\binom{i}{\hat{p}_i}\binom{n+m-i}{n-\hat{p}_i}\right] \le \lg\left[nm\binom{i}{\tilde{p}_i}\binom{n+m-i}{n-\tilde{p}_i}\right].$$

When we use $\gamma + (n+1/2)\lg n - n\lg e \le \lg\Gamma(n+1) \le \gamma + (n+1/2)\lg n - n\lg e + \lg e/12n$, where $\gamma = \lg\sqrt{2\pi}$, for n real (see [17] exercise 1.2.11.2-6) to substitute for the choose expressions we get Equation 3. ∎

Next we consider the second summation in Equation 2.

$$\begin{aligned}b\sum_{i=1}^{n+m-1}[\lg i + \lg(n+m-i)] \ge \\ 2b[\gamma + (n+m-1/2)\lg(n+m) - (n+m)\lg e] \qquad (4)\end{aligned}$$

Theorem 2.4 *The expected work to take the union of two treaps t_n and t_m of size n and m, respectively, and $m \le n$ is*

$$\mathbf{E}[\mathbf{W}(n,m)] = O(m\lg(n/m))$$

Proof. Substituting Equations 3 and 4 in Equation 2 gives the stated bounds. ∎

Corollary 2.5 *The expected work to take the intersection of two treaps t_n and t_m of size n and m, respectively, and $m \le n$ is:*

$$\mathbf{E}[\mathbf{W}(n,m)] = O(m\lg(n/m))$$

Proof. The only additional work that intersection does that union does not is a join when there is no duplicate key. But since the join must be on trees that are no larger than the result trees of the split prior to the recursive calls, the additional join only changes the constants in the work bound. ∎

Corollary 2.6 *The expected work to take the difference of two treaps t_n and t_m of size n and m, respectively, and $m \leq n$ is:*

$$\mathbf{E}\left[\mathbf{W}(n,m)\right] = O(m\lg(n/m))$$

Proof. (Sketch) As with intersection the only additional work difference does that union does not is a join. When a subtree of t_n was split this join takes no more work than the split preceding it. When a subtree of t_m was split the join may take work proportional to the lg of the size of the corresponding subtree of t_n. However, as this join only takes place when the key occurs in both t_n and t_m, the work over all permutations associated with the join is $mO(\lg n)$. Thus, the work bound for difference is same as for union. ∎

Theorem 2.7 *When $p = m/\lg m$ the expected depth to take the union, v intersection, or difference of two treaps t_n and t_m of size n and m, respectively, and $m \leq n$ is*

$$\mathbf{E}\left[\mathbf{D}(n,m)\right] = O(\lg m \lg n)$$

Proof. (*Sketch*) Let h_n and h_m is the height of t_m and t_n, respectively. Every time t_m is split it takes no more than $O(h_m)$ time. Although the heights of the resulting trees may not be smaller than the original tree, the height of the subtrees from t_n are reduced by one. Similarly, when t_n is split the heights of the subtrees from t_m are reduced by one. Thus, after $O(h_m h_n)$ steps the algorithms complete. Since the expected heights of t_m and t_n are $O(\lg m)$ and $O(\lg n)$, the expected depth of the operations are $O(\lg m \lg n)$. ∎

In [3] the authors show that the depths of the operations can be reduced to $O(\lg m + \lg n)$ using pipelining. Furthermore since the recursive calls in the algorithms are independent (never access the same parts of the trees) the algorithms run with exclusive reads and writes. Using Brent's scheduling principle these results together with the work bounds imply the algorithms will run in $O(\frac{m\lg(n/m)}{p} + T_c \lg n)$ time on an EREW PRAM, where T_c is the time for a compaction, which is needed for scheduling the tasks to the processors. Such a compaction can easily be implemented with a parallel prefix (scan) operation, giving $T_c = \lg p$ on a plain EREW PRAM or $T_c = 1$ in a PRAM with scan operations [4].

2.5 Analysis of Union with Fast Splits

Here we prove bounds on the work for Union using fast splits when using the block metric. We do not know, however, how to pipeline this version so the depth of the algorithm on two sets of size n and m is $O(\lg n \lg m)$.

Lemma 2.8 *Assuming the expected cost to cut a sequence into two sequences of nonzero length n and m is $1+\lg(\min(n,m))$, then any set of cuts that partitions a sequence of length n into k blocks will have a total expected cost $T_p \leq 2k(1 + \lg(n/k))$*

Proof. For a sequence of blocks N we will denote the lengths of the blocks as $\{n_1, n_2, \ldots, n_k\}$ and the sum of the lengths as n. We define $T_s(N) = \sum_1^k (1 + \lg n_i)$. Since the logarithm is concave downward, for a fixed k and n this sum is maximized when all the blocks are the same length, giving $T_s(N) \leq k(1 + \lg\lceil n/k\rceil)$. We can model a set of cuts that partitions a sequence into blocks as a tree with the blocks at the leaves and each internal node representing one of the cuts. We use $T_p(v)$ to denote the total expected cost of the cuts for a tree rooted at v. We use $T_s(v)$ to refer to $T_s(N)$ where N are the blocks at the leaves of the tree rooted at v. By our assumption of the cost of a cut we can write the recurrence

$$T_p(v) = \begin{cases} 0 & v \text{ a leaf} \\ T_p(l(v)) + T_p(r(v)) + 1 & \text{otherwise} \\ \quad + \lg(\min(|l(v)|, |r(v)|) & \end{cases} \tag{5}$$

where $l(v)$ and $r(v)$ are the left and right children of v, and $|v|$ is the sum of the sizes of the blocks in the tree rooted at v. The following is also true, by definition

$$T_s(v) = \begin{cases} 1 + \lg(|v|) & v \text{ a leaf} \\ T_s(l(v)) + T_s(r(v)) & \text{otherwise} \end{cases}.$$

Now we prove by induction on the tree that $T_p(v) \leq 2T_s(v) - \lg(|v|) - 2$. In the base case it is true for the leaves since $0 \leq 2(1 + \lg|v|) - \lg|v| - 2$. For the induction case we substitute T_s into the right hand side of Equation 5 giving

$$\begin{aligned} T_p(v) &\leq 2T_s(l(v)) - \lg(|l(v)|) - 2 \\ &\quad +2T_s(r(v)) - \lg(|r(v)|) - 2 \\ &\quad +1 + \lg(\min(|l(v)|, |r(v)|)) \\ &= 2T_s(v) - \lg(n) - 3 \\ &\leq 2T_s(v) - \lg(n+m) - 2 \\ &= 2T_s(v) - \lg(|v|) - 2, \end{aligned}$$

where $n = \max(|l(v)|, |r(v)|)$ and $m = \min(|l(v)|, |r(v)|)$. Since $T_s(N) \leq k(1 + \lg(\lceil n/k\rceil))$ we have for any set of cuts $T_p(N) \leq 2k(1 + \lg(\lceil n/k\rceil))$. ∎

Theorem 2.9 *The parallel union algorithm using fast splits and joins on two ordered sets A and B runs in $O(k\lg((n+m)/k))$ expected work where $|A| = n$, $|B| = m$, and k is the minimum number of blocks in a partitioning of A such that no value of B lies within a block.*

Proof. We count all the work of the algorithm against the cost of cutting A into k blocks, cutting B into $k \pm 1$ blocks, and joining the $2k \pm 1$ blocks.[2] Since the fast versions of split and join take work bounded by what is required by Lemma 2.8, the total expected work of cutting and joining is bound by $O(k\lg((n+m)/k))$.

We consider two cases. First, when the split in the union returns two nonempty sets. In this case we count the cost of the split and the constant overhead of the union against the partitioning of either A or B (whichever is being split). Note that on either input set there can be at most k cuts due to calls to the split function. Union can make additional cuts within a block when it removes the root of the tree (when it has the higher priority) and divides the tree into its left and right branches. But these trivial cuts will

[2] To be precise, we count against a constant multiple of the plain split and join times since we include some constant-work overheads in the union function in their times.

only reduce the cost of the split cuts and are charged against a split in the other tree. Second, consider when the split in the union returns an empty set. In this case, even though the split only takes constant work, we cannot count it or the union overhead against the k cuts of A or B. Assume that $(\emptyset, T_2) = \mathtt{split}(T_2, r_1)$ (*i.e.*, T_2 is being split by the root of T_1). Recall, that when the fast-split version of union gets an empty set, it then splits T_1 by the first value in T_2 giving T_{11} and T_{12} and executes $\mathtt{join}(T_{11}, \mathtt{union}(T_{12}, T_2))$. Finding the minimum value of T_2 takes constant time using a finger search. We count the cost of the split against the partitioning of the set corresponding to T_1 (unless T_{12} is empty, in which case we count the constant cost against the join). Since $\mathtt{minkey}(T_2) < \mathtt{minkey}(T_{12})$, the `join` is along one of the cuts between blocks of the result. We can therefore count the cost of the join and the constant overhead in the union against joining the $2k \pm 1$ result blocks. The constant work of any calls to union with an empty set (base of the recursion) are counted against their parent's split or join. ∎

3 Implementation

To evaluate the performance of these set-based operations, we implemented the serial treap algorithms in Gnu C and the parallel ones in Cilk 5.1 [5]. Cilk is a language for multithreaded parallel programs based on ANSI C, and is designed for computations with dynamic, highly asynchronous parallelism, such as divide-and-conquer algorithms. It includes a runtime system that schedules the multithreaded computation using work-stealing and provides dag-consistent distributed shared memory. We ran our experiments on an SGI Power Challenge with 16 195MHz R10000 processors and 4 Gbytes of memory running the IRIX 6.2 operating system, and on a Sun Ultra Enterprise 3000 with six 248MHz UltraSPARC II processors and 1.5Gbytes of memory running the SunOS 5.5.1 operating system.

3.1 Sequential experiments

Since speedup is a common measure of the performance of parallel algorithms, it is important that we compare the parallel performance with a good sequential algorithm. Our first step is to verify that sequential treaps compare reasonably well with other good sequential balanced tree algorithms. We implemented and compared the performance of red/black trees [25], splay trees [27] (Sleator's code), skip lists [21] (Pugh's code) and treaps on an SGI Power Challenge and Sun Ultra Enterprise 3000.

To evaluate the performance of the algorithms we performed a series of tests that create a tree of size n, and insert, delete and search for k keys in a tree of size n. For each test we also used four data distributions. One distribution inserts, searches, and deletes random keys. The remaining distributions insert, search, and delete consecutive keys in various orders. Figure 3 shows the time to create a tree from random keys. The union version for treaps creates the tree using recursive calls to union organized as in mergesort, instead of inserting one at a time. Our other experiments give similar results and in all cases all four data structure give running times that are within a factor of 2 of each other.

Next we show the results of union on one processor. Figure 4 shows the runtime for the union operation on treaps of various input sizes n and m, where the keys are random integers over the same range of values. Notice that the x-axis

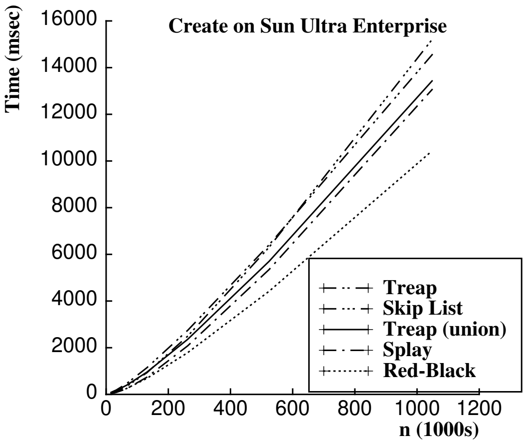

Figure 3: The running time to create various data structures from random keys on a single processor of a Sun Ultra Enterprise 3000.

specifies the sum of the treap sizes $n + m$ and each curve is for unions where one tree size stays fixed. As the graphs show the lines rise rapidly until $m = n$ and then rise more slowly. This change reflects the symmetry of the changing roles as the one tree switches from being the smaller to the larger. The envelope of the lines give the linear union times when $n = m$. Thus, one can see the sublinear times when the tree sizes are unequal.

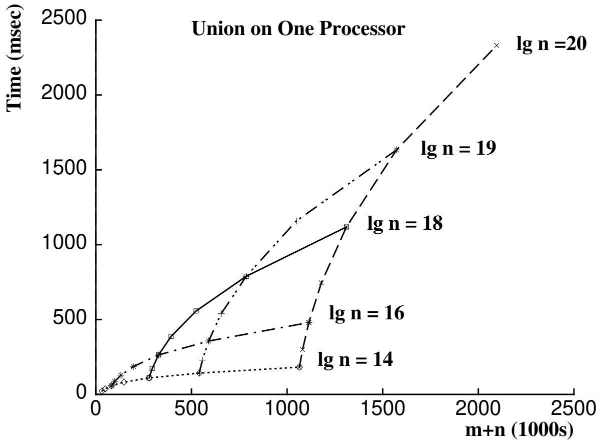

Figure 4: Time on one processor of a Sun Ultra Enterprise 3000 to find the union of two treaps of various sizes. Each line specifies the times when one treap is of the size indicated and the other treap size varies.

Another advantage to our treap algorithms is that they take advantage of the "easiness" of the data distributions. Union, intersection and difference take less time on sets that do not have finely interleaving keys than sets that do, even when they are not the versions using fast splits and joins. Figure 5 shows the runtimes on one processor to take the union (without fast splist and joins) of a tree with a million nodes and a tree with 16 thousand nodes for a varying number of blocks. Each tree has equal size blocks such that the k blocks from each tree interleave in the set union. The maximum number of blocks is the size of the smaller tree.

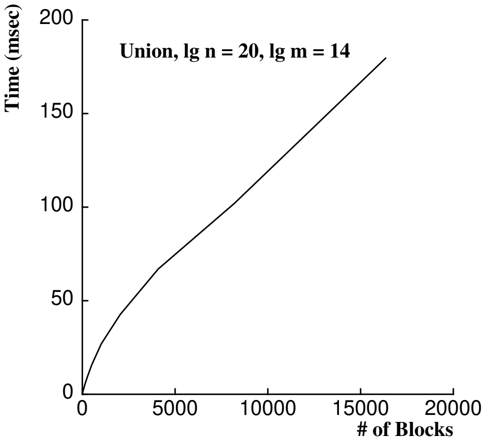

Figure 5: Time on one processor of a Sun Ultra Enterprise 3000 to find the union of a 1M node treap and 16K treap for a varying number of blocks.

Finally, Figure 6 shows the time on one processor for union, intersection and difference. The keys are from two uniform random distributions of the same range. Again we see the sublinear times similar to those for union.

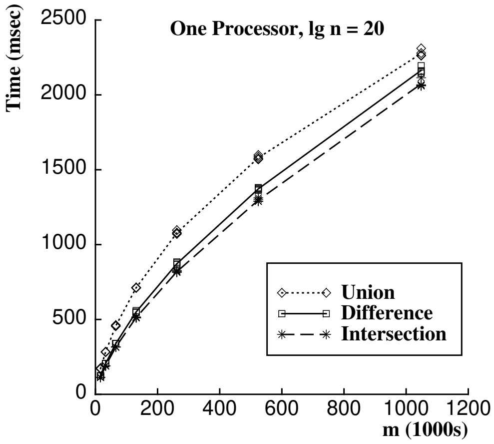

Figure 6: Time on one processor of a Sun Ultra Enterprise 3000 to take the union of a 1M node treap with a treap of various sizes and the intersection and difference of the result with the second treap in the union. Each treap has keys drawn randomly from the same range of values.

3.2 Parallel experiments

Our parallel versions of the treap operations are simple extensions to the C code in which we spawn threads for the two recursive calls. In Cilk this requires adding the `cilk` keyword before each function definition, the `spawn` keyword before each recursive call and the `sync` keyword after the pair of calls to wait for both calls to complete before continuing. As discussed below, we also use our own memory management and stop making parallel calls when reaching a given depth in the tree.

In our first effort to implement the parallel algorithms in Cilk, we used nonpersistent versions and relied on the Cilk memory management system to allocate and deallocate the nodes of the trees. The results were disappointing, especially for the difference operation where times were worse on more processors than on fewer processors. The slow down was due to two factors. One factor was that the granularity of memory allocation/deallocation is quite small, the size of a treap node. To ensure that the memory operations are atomic and independent, Cilk uses a memory lock. Because the granularity is small, there was high contention for the memory lock. The second factor was that the cache lines on the SGI are long, 128 bytes (32 words), so that 8 treap nodes share the same cache line. In the nonpersistent version of the code when two processors write to different nodes on the same cache line, called "false sharing", the processors need to exchange the cache line even though they are not sharing nodes. The treap operations result in a large amount of false sharing.

To solve the first problem, we wrote our own memory management system for the tree nodes on top of Cilk's. It allocates tree nodes from a large area of memory allocated by Cilk. It divides this memory into smaller blocks of consecutive memory and gives each processor one block from which to allocate tree nodes. Every time a processor runs out of nodes in its block it gets a new block. In this way, a processor only acquires a lock when it needs a new block. This assumes some form of copying garbage collection (either automatic or explicitly called by the user) since memory is allocated consecutively and never explicitly deallocated. We did not implement a garbage collector for the experiments but we did measure the time to copy a treap and it is very much less than the time to create it.

To solve the false sharing problem we converted our initial nonpersistent implementations into persistent versions. These versions write only to newly allocated nodes within a processor's own block of memory (they never modify an existing node). The cost of persistence varied on a single processor. For union, which allocates the $O(m \lg(n/m))$ new nodes, the persistent version was slower than the nonpersistent version by a modest 9%, when m is small compared to n, and 50%, when $m = n$. When the result of an intersection was small relative to the input sizes, the persistent version was 35% faster than the nonpersistent version. When the result was large the persistent version was 30% slower. For set difference the persistent version was 23%–12% faster than the nonpersistent version. For multiple processors the speedup was consistently better for the persistent version. For example, for intersection on 8 processors of the SGI Power Challenge the speedup is about 6.5 for the persistent version, and 5.0 for the nonpersistent version.

Another improvement we made was to revert to sequential code after spawning threads to a certain depth. That is, every time we applied another recursive call, we decremented a counter. Once the counter reached zero we made a recursive call to a non-Cilk procedure which only made calls to C procedures. The C procedure was exactly like the Cilk one except it did not use `Spawn` and `Sync`. In this way, in the C procedure we avoid the overhead of the Cilk function call, which is about three times the overhead of a C function call [28]. In Figure 7 we show the times for union when we vary the depth at which we revert to C. Notice that as the depth decreases the times improve and then get worse. If we revert to C too soon, the problem size on which the threads have to work can vary greatly and there are not enough threads to get sufficient load balancing. The run times for different data sets, therefore, vary quite widely. For larger depths, the execution incurs more Cilk overhead; the run times, however, are more consistent over different data sets.

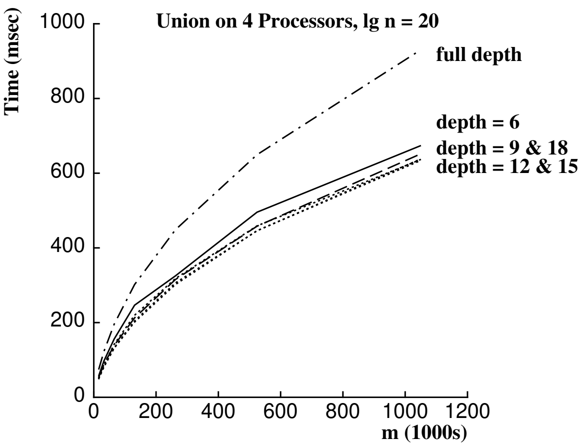

Figure 7: Times on four processors a Sun Ultra Enterprise 3000 to find the union of a million node treap with a treap of various sizes. Each line shows a different call depth at which the Cilk run reverted to sequential code.

Finally, Figure 8 shows speedups for up to 5 processors on the Sun Ultra Enterprise 3000 and up to 8 processors of the SGI Power Challenge. The times are for finding the union of two 1-million node treaps, and finding the intersection and difference of the result treap with one of the input treaps to union. The speedups are between a factor of 4.1 and 4.4 on 5 processors of the Sun and a factor of 6.3 and 6.8 on 8 processors of the SGI, which is quite reasonable considering the high bandwidth requirements of the operations.

4 Discussion

We considered parallelizing a variety of balanced trees (and skip lists) for operations on ordered sets. We selected treaps because they are simple, fully persistent (if implemented appropriately), and easy to parallelize. It is hard to make a definitive argument, however, that one data structure is simpler than another or that the "constant factors" in runtime are less since it can be very implementation and machine dependent. We therefore supply the code and experiments as data points. In terms of asymptotic bounds we believe we present the first parallel algorithms for set operations that run in $O(m \lg((n+m)/m))$ expected work and polylogarithmic depth, and similarly for the block metric with k blocks the first parallel algorithm that runs in $O(k \lg((n+m)/k))$ expected work.

We finish by briefly describing a parallel union algorithm for skip-lists that uses an approach similar to our algorithm on treaps. Recall that in a skip list every element is assigned a height $h \geq 1$ with probability 2^{-h}, and has h pointers $p_i, 1 \leq i \leq h$ which point to the next element in sorted order with height $\geq i$.[3] To merge two sets represented as skip lists, pick the set with the greater maximum height (or an arbitrary one if the heights are the same) and call this A and the other set B. Split A and B using the elements of height h in A, recurse in parallel on the split regions, and join the results. For two sets of size n and m this algorithm will have expected parallel depth $O(\lg n \lg m)$. With the appropriate extensions (including back pointers) splits and joins can be implemented to run with the same bounds as fast joins and splits in treaps. Using a similar argument as used in Theorem 2.9 we conjecture that the total expected work for this union with skip lists is $O(k \lg((n+m)/k))$ with k being the number of blocks. The main disadvantage with skip lists are that they are much more difficult to make persistent.

[3] More generally for a parameter $0 < p < 1$, the probability of being assigned height h is $(1-p)p^{h-1}$.

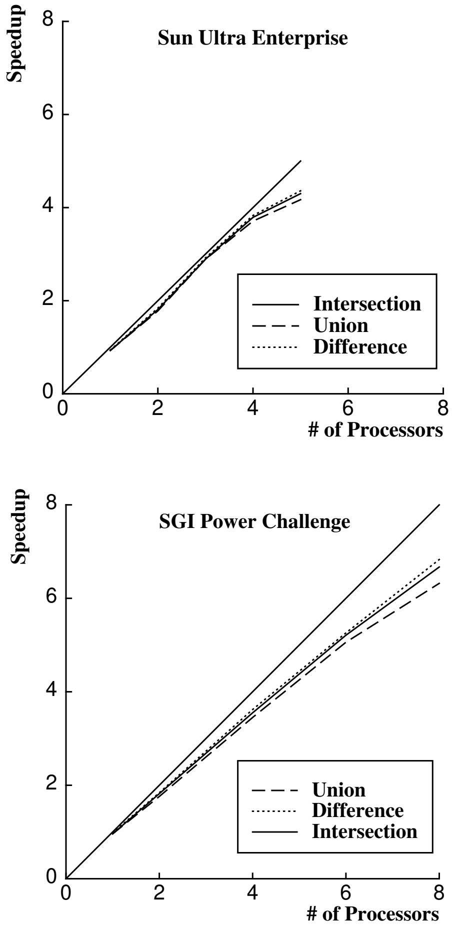

Figure 8: Speedup for the union of a 1M node treap with a 1M node treap, and the intersection and difference of the result with the same second 1M node treap.

Acknowledgements

This work was partially supported by DARPA Contract No. DABT63-96-C-0071 and by an NSF NYI award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government. Access to the SGI Power Challenge was provided by the the National Center for Supercomputing Applications (NCSA), whose staff was extremely helpful when dealing with all our requests.

References

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

[2] A. Bäumker and W. Dittrich. Fully dynamic search trees for an extension of the BSP model. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 233–242, 1996.

[3] G. Blelloch and M. Reid-Miller. Pipelining with futures. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 249–259, June 1997.

[4] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C–38(11):1526–1538, Nov. 1989.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, CA, July 1995.

[6] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the Association for Computing Machinery*, 26(2):211–226, Apr. 1979.

[7] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9(3):594–614, Aug. 1980.

[8] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural merge sort. In *Proceedings of the International Symposium on Algorithms SIGAL'90*, pages 251–260, Tokyo, Japan, Aug. 1990.

[9] E. Dekel and I. Azsvath. Parallel external merging. *Journal of Parallel and Distributed Computing*, 6:623–635, 1989.

[10] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, Feb. 1989.

[11] X. Guan and M. A. Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers*, 40:592–602, 1991.

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

[13] L. Highan and E. Schenk. Maintaining B-tree on an EREW PRAM. *Journal of Parallel and Distributed Computing*, 22:329–335, 1994.

[14] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1:31–39, Mar. 1972.

[15] J. Katajainen. Efficient parallel algorithms for manipulating sorted sets. *Proceedings of the 17th Annual Computer Science Conference, Australian Computer Science Communications*, 16(1):281–288, 1994.

[16] J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. In *Proceedings of the 4th International PARLE Conference (Parallel Architectures and Languages Europe)*, volume 605 of *Lecture Notes in Computer Science*, pages 37–49, 1992.

[17] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1968.

[18] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, MA, 1973.

[19] M. L. Mauldin. Lycos: Design choices in an Internet search service. *IEEE Expert*, 12(1), Jan. 1997. (www.computer.org/pubs/expert/1997/trends/x1008/mauldin.htm).

[20] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2–3 trees. In *Lecture Notes in Computer Science 143: Proceedings of the 10th Colloquium on Automata, Languages and Programming, Barcelona, Spain*, pages 597–609, Berlin/New York, July 1983. Springer-Verlag.

[21] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, University of Maryland, June 1990.

[22] W. Pugh. Skip lists: A probilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[23] A. Ranade. Maintaining dynamic ordered sets on processor networks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 127–137, San Diego, CA, June-July 1992.

[24] M. Reid-Miller. *Experiments with Parallel Pointer-Based Algoriths*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1998. To appear.

[25] R. Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, MA, 1990.

[26] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[27] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, 1985.

[28] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk-5.0 (Beta 1) Reference Manual*, Mar. 1997.

[29] P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: Algorithm and implementation results. *Parallel Computing*, 15:165–177, 1990.

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, 1994.