A Parallel Buffer Tree

Nodari Sitchinava
Institite for Theoretical Informatics
Karlsruhe Institute of Technology
nodari@ira.uka.de

Norbert Zeh[*]
Faculty of Computer Science
Dalhousie University
nzeh@cs.dal.ca

ABSTRACT

We present the *parallel buffer tree*, a parallel external memory (PEM) data structure for batched search problems. This data structure is a non-trivial extension of Arge's sequential buffer tree to a private-cache multiprocessor environment and reduces the number of I/O operations by the number of available processor cores compared to its sequential counterpart, thereby taking full advantage of multicore parallelism.

The parallel buffer tree is a search tree data structure that supports the batched parallel processing of a sequence of N insertions, deletions, membership queries, and range queries in the optimal $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ parallel I/O complexity, where K is the size of the output reported in the process and $\mathrm{sort}_P(N)$ is the parallel I/O complexity of sorting N elements using P processors.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and Concurrency*

General Terms

Algorithms, Theory

Keywords

Parallel external memory model, PEM model, buffer tree, batched data structures, parallel data structures, parallel buffer tree, parallel buffered range tree

1. INTRODUCTION

Parallel (multicore) processors have become the standard even for desktop systems. This spawned a renewed focus on parallel algorithms, with a particular focus on multicore systems. The main difference between modern multicore processors and previous models for parallel computing is the combination of a shared memory, much like in the PRAM model, and private caches of the cores that cannot be accessed by any other core. The cache architectures of current multicore chips vary. Some provide a flat model where each processor has its private cache and all processors have access to a shared memory. Others provide a hierarchical cache structure where intermediate cache levels are added and subsets of processors share access to caches at these levels.

Several models have been proposed to capture the hierarchical memory design of modern parallel architectures to a varying degree [6, 10, 11, 14, 22]. Private-cache models are the simplest among them and assume a 2-level memory hierarchy: a fast *private cache* for each processor and a slow *shared memory.* While these models may not accurately represent the full memory hierarchy of most multicores, every multicore architecture has private caches at some level of the memory hierarchy, as well as RAM shared by all processors, which is typically much slower. Thus, private-cache models focus on the common denominator of multicore processors and are useful for the development of techniques to utilize the private caches on such processors.

In this paper, we work in one such private-cache model: the *parallel external memory (PEM) model* [6,20]. The PEM model is a natural extension of the widely accepted *I/O model* [1] to multiple processors/processor cores. In this model, the computer has P processors, each with its own private *cache* of size M. Each processor can access only its own cache. At the same time, all processors have access to a *shared memory* of conceptually unlimited size. All computation steps carried out by a processor have to happen on data present in its cache. Data is transferred between shared memory and the processors' caches using (parallel) *I/O operations* (I/Os). In one I/O operation, each processor can transfer one block of B consecutive data items between its cache and shared memory. The complexity of an algorithm in the PEM model is the number of such parallel I/O operations it performs. Throughout this paper, we assume *block-level concurrent read, exclusive write* access, that is, multiple processors may read the same block of data simultaneously, but a block written by one processor during an I/O operation may not be accessed (read or written) by another processor during the same I/O operation, even if the two processors access different addresses within the block.

[*]Supported by NSERC and the Canada Research Chairs programme.

1.1 Our Results

We present the *parallel buffer tree*, a batched data structure that is able to process a sequence of N INSERT, DELETE, FIND, and RANGEQUERY operations in the optimal number of $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ parallel I/Os, where K is the total output size produced by all queries. This structure is an extension of the sequential buffer tree [5] to the PEM model. The sequential buffer tree has been used as a basis for a wide range of sequential I/O-efficient algorithms (see, for example, Arge's survey [4]). By extending this data structure to the PEM model, we provide an important first step towards enabling analogous solutions to these problems in the PEM model.

1.2 Previous Work

The first problems studied in the PEM model were fundamental problems such as sorting, scanning, and computing prefix sums. Sorting in the PEM model takes $\mathrm{sort}_P(N) = \mathrm{O}\left(\frac{N}{PB}\log_{M/B}\frac{N}{B}\right)$ I/Os, provided $P \le N/B^2$ and $M = B^{\mathrm{O}(1)}$ [6, 20]. Scanning a sequence of N elements and computing prefix sums take $\mathrm{O}(\log P + N/PB)$ I/Os, which equals $\mathrm{O}(N/PB)$ whenever $N \ge P\log P$. Subsequent papers in this model studied graph algorithms [7] and some geometric problems [2, 3]. Other related papers explored the connection between cache-oblivious algorithms and private-cache architectures and proposed general techniques for translating cache-oblivious algorithms into algorithms for private- and mixed-cache architectures [10, 11, 13, 14]. None of these results immediately leads to a buffer tree for private cache models because no cache-oblivious equivalent of the buffer tree is known.

Sequential search structures are ubiquitous in computer science. The B-tree [8] is widely used in database systems for fast searching of data stored on disk. The proliferation of parallelism particularly through multicore processors has led to a recognition in the research community that strictly sequential data structures quickly become a bottleneck in applications that can otherwise benefit greatly from multicore parallelism, including database systems. One approach to alleviate this bottleneck is the support of *concurrency*, where multiple processors are allowed to access and manipulate the data structure simultaneously. The support of concurrency in B-trees has been investigated extensively in the database community (for example, see [21] and references therein). More recently, cache-oblivious concurrent B-trees were proposed in [9].

While certainly related and motivated by the same trend in modern computer architectures, these results do not address the problem studied in this paper: how to process a *sequence* of operations, as opposed to individual operations, as quickly as possible. Just as the sequential buffer tree [5], the parallel buffer tree may delay the answering of queries but ensures that queries produce the same answer as if they had been answered immediately. This is crucial for achieving the I/O bound stated in Section 1.1 because processing each query immediately requires at least one I/O per query, even if it produces very little output. The structures discussed in the previous paragraph and even structures designed to efficiently support batched updates [19] answer queries immediately and, thus, cannot achieve an optimal I/O complexity for processing a sequence of operations.

Concurrency also provides much looser support for parallel processing than explicitly *parallel* data structures, such as the parallel buffer tree presented in this paper. Concurrency only allows multiple processors to operate simultaneously *and independently* on the data structure. In a parallel data structure, on the other hand, processors *cooperate* to speed up the processing of updates and queries. In the PRAM model several search tree data structures have been proposed [12, 17, 18]. However, these structures are not I/O-efficient, as they makes use of random access to the shared memory provided by the PRAM model.

2. PRIMITIVES

Two primitives we use extensively in our algorithms are the merging of two sorted sequences and the distribution of a sorted sequence of elements into buckets. The next two lemmas state the complexities of these operations.

LEMMA 1. *Two sorted sequences S_1 and S_2 can be merged using $\mathrm{O}\left(\left\lceil\frac{|S_1|+|S_2|}{PB}\right\rceil + \log P\right)$ I/Os.*

PROOF. The algorithm is shown in Figure 1. Choosing the subsequences S_1' and S_2' in Step 1 takes $\mathrm{O}(1)$ I/Os, and merging these subsequences to obtain S' takes $\mathrm{O}(\log P)$ I/Os. Each subsequence $S_{(a,b)}$ to be produced by a processor in Step 2 is contained in the union of two subsequences of S_1 and S_2 bounded by consecutive elements in S_1' and S_2'. Thus, a single processor can produce $S_{(a,b)}$ using $\mathrm{O}\left(\left\lceil\frac{|S_1|+|S_2|}{PB}\right\rceil\right)$ I/Os, and each processor has to produce only two such sequences. This leaves the issue of concatenating these sequences to obtain the final merged sequence S. Using a prefix sum computation on the lengths of these sequences $S_{(a,b)}$, we can compute the position of each subsequence $S_{(a,b)}$ in S. This takes $\mathrm{O}(\log P)$ I/Os. Given these positions, each processor can write $S_{(a,b)}$ to the correct blocks in S. The only issue to be resolved is when multiple processors try to write to the same block. in this case, at most P processors try to write to the same block, and the portions of S to be written to this block by different processors can be merged using $\mathrm{O}(\log P)$ I/Os. $\square$

LEMMA 2. *A sorted sequence of N elements can be distributed to k buckets delimited by a sorted sequence R of $k-1$ pivot elements using $\mathrm{O}\left(\left\lceil\frac{N+k}{PB}\right\rceil + \lceil k/P\rceil + \log P\right)$ I/Os.*

PROOF. The algorithm is shown in Figure 2. By Lemma 1, the I/O complexity of the first step is $\mathrm{O}\left(\left\lceil\frac{N+k}{PB}\right\rceil + \log P\right)$. Step 2 can be accomplished by using a prefix sum computation to annotate each element of S with the index of the last pivot that precedes it. The same prefix sum computation can be used to count the number of elements of S between every pair of consecutive pivot elements in the merged list (Step 3). Using a similar technique, we can separate the elements of S into two lists, one containing full blocks of B elements to be sent to the same bucket, and the second one containing one block per bucket, which stores the remaining less than B elements to be sent to this bucket. Step 4 distributes the blocks of the first list to the buckets. Step 5 distributes the elements of the non-full blocks to the buckets. In each step, we allocate the same number of blocks from the respective lists to each processor. Since there are at most k non-full blocks, the last three steps take $\mathrm{O}((N/PB) + \lceil k/P\rceil + \log P)$ I/Os. The lemma follows. $\square$

```
MERGE($S_1, S_2$):
  1. Choose $P$ evenly spaced elements from each of $S_1$ and $S_2$ and merge the resulting two subsequences $S'_1$ and $S'_2$ to obtain
     a sequence $S'$.
  2. Assign two pairs of consecutive elements in $S'$ to each processor. Each such pair $(a, b)$ delimits a subsequence $S_{(a,b)}$ of
     the merged sequence $S = S_1 \cup S_2$ that contains all elements $x \in S$ with $a \leq x < b$. The processor that is assigned the
     pair $(a, b)$ produces the subsequence $S_{(a,b)}$ of the output sequence $S$.
```

Figure 1: Merging two sorted sequences S_1 and S_2

```
DISTRIBUTE($S, R$):
  1. Merge the sets $S$ and $\{r_0\} \cup R$ using Lemma 1, where $r_0 = -\infty$.
  2. Annotate each element in $S$ with the bucket $i$ where it should be sent.
  3. Compute the number of items to be sent to each bucket.
  4. Distribute full blocks of $B$ items.
  5. Distribute the non-full blocks (at most one per bucket).
```

Figure 2: Distributing a sorted sequence S of N items to a set of k buckets defined by pivots $R = \{r_1, \cdots, r_{k-1}\}$

3. THE PARALLEL BUFFER TREE

In this section, we describe the basic structure of the parallel buffer tree and show how to support INSERT, DELETE, and FIND operations. In Section 4, we show how to extend the structure to support RANGEQUERY operations. The input is a sequence of N INSERT, DELETE, and FIND operations. An INSERT(x) operation inserts element x into the buffer tree, a DELETE(x) operation removes it, and a FIND(x) operation needs to decide whether x is in the buffer tree at the time of the query.[1] All these operations may be delayed—in particular, the answers to FIND queries may be produced much later than the queries are asked—but the results of all queries must be the same as if all operations were carried out immediately. More precisely, a FIND(x) query answers yes if and only if the last operation affecting x and preceding this FIND(x) query in the input sequence is an INSERT(x) operation.

We show how to process any sequence of N such operations using $O(\text{sort}_P(N))$ parallel I/Os. Note that we could achieve this simply by sorting the operations in the sequence primarily by the elements they affect and secondarily by their positions in the sequence of operations (time stamps) and then applying a parallel prefix sum computation to the resulting sequence of operations. This is indeed what we do at each node of the buffer tree. However, we use the buffer tree to answer range queries in Section 4, a problem that cannot be solved in this naïve manner.

The parallel buffer tree is an (a, b)-tree with parameters $a = f/4$ and $b = f$, for some branching parameter $f \geq PB$, that is, all leaves are at the same distance from the root, every non-root internal node has between $f/4$ and f children, and the root r has between 2 and f children. Each leaf of the tree stores $\Theta(B)$ elements, so the height of the tree is $O(\log_f(N/B))$. Each non-leaf node v has an associated buffer $\mathcal{B}_v$ of size $\Theta(g)$, where $g = fB$, which is stored in shared memory. We call an internal (i.e., non-leaf) node a *fringe node* if its children are leaves of the tree. As in a standard (a, b)-tree, every internal node v with children $w_1, w_2, \ldots, w_k$ stores $k - 1$ *routing elements* $r_1, r_2, \ldots, r_{k-1}$, which separate the children of v: each routing element r_i is no less than the elements stored in w_i's descendant leaves and no greater than the elements in w_{i+1}'s descendant leaves.

We assume the sequence of operations is provided to the buffer tree in batches of PB operations.[2] Each operation is annotated with a time stamp indicating its position in the sequence of operations. We do not require the operations in each batch to be sorted by their time stamps, which potentially makes the parallel generation of these batches of operations easier, but we do require that any operation in a batch has a greater time stamp than any operation in a preceding batch. For each batch $\mathcal{O}$ of PB operations, we assign one processor to each block of B operations in $\mathcal{O}$, and the processor inserts this block into the root buffer $\mathcal{B}_r$. If $\mathcal{B}_r$ contains more than g operations, we apply the buffer emptying procedure NONFRINGEEMPTY (Figure 3) to r, to distribute the operations in $\mathcal{B}_r$ to the buffers of r's children. For each child v whose buffer overflows as a result of receiving additional operations, we invoke NONFRINGEEMPTY(v) recursively unless v is a fringe node. Every fringe node whose buffer overflows is added to a list $\mathcal{F}$ of full fringe nodes. Once the recursive application of the buffer emptying procedure for non-fringe nodes finishes, we apply the buffer emptying procedure FRINGEEMPTY (Figure 4) to the nodes in $\mathcal{F}$, one at a time. Part of the work done by this procedure is rebalancing the tree to reflect the creation/destruction of leaves as a result of the insertion/deletion of data items.

This general approach of propagating operations down the tree in a batched fashion is identical to the approach used in the sequential buffer tree [5]. The difference lies in the implementation of the steps of the two buffer emptying procedures, which need to distribute the work across processors. Next we discuss these two procedures in more detail.

3.1 Emptying Non-Fringe Buffers

The buffer emptying procedure for non-fringe internal nodes is shown in Figure 3. The goal of Step 1 is to sort $\mathcal{B}_v$. This can be done as described in the algorithm because all but the first at most g elements in $\mathcal{B}_v$ were inserted into $\mathcal{B}_v$

[1]Alternatively, we could require a FIND(x) operation to report information associated with x in the tree. Either of these variants of the FIND operation are equally easy/hard to support.

[2]We cannot control how the sequence of operations is produced by an application using the buffer tree. If the entire sequence of operations is given up front, it is easy to divide this sequence into batches of size PB.

```
NonFringeEmpty($v$):
 1. Divide $\mathcal{B}_v$ into two parts $\mathcal{B}'_v$ and $\mathcal{B}''_v$. $\mathcal{B}'_v$ contains the first $g$ elements in $\mathcal{B}_v$, $\mathcal{B}''_v$ the remaining elements. Sort the
    elements in $\mathcal{B}'_v$ primarily by their keys and secondarily by their time stamps and merge the resulting list with $\mathcal{B}''_v$ by
    running Merge($\mathcal{B}'_v, \mathcal{B}''_v$) (Figure 1).
 2. Answer and eliminate Find queries with matching Insert or Delete operations in $\mathcal{B}_v$ and eliminate matching Insert
    and Delete operations in $\mathcal{B}_v$.
 3. Distribute the remaining elements in $\mathcal{B}_v$ to the buffers of $v$'s children by running Distribute($\mathcal{B}_v, R_v$) (Figure 2), where
    $R_v$ is the set of routing elements at node $v$.
 4. For every child $w$ of $v$ whose buffer $\mathcal{B}_w$ now contains more than $g$ operations, invoke NonFringeEmpty($w$) recursively
    unless $w$ is a fringe node. If $\mathcal{B}_w$ contains more than $g$ elements and $w$ is a fringe node, append $w$ to the list $\mathcal{F}$ of full
    fringe nodes.
```

Figure 3: The buffer emptying procedure for non-fringe internal nodes

```
FringeEmpty($v$):
 1. Sort the operations in $\mathcal{B}_v$.
 2. Merge the operations in $\mathcal{B}_v$ with the elements stored in $v$'s children, treating each such element as an Insert operation
    with time stamp $-\infty$. Then remove matching Insert and Delete operations and answer Find queries. Finally, replace
    every Insert operation in the list without a matching Delete operation with the element it inserts. The resulting list
    is the list of elements $E_v$ to be stored in $v$'s children.
 3. Populate $v$'s children:
    3.1. If $g/4 \le |E_v| \le g$, store the elements of $E_v$ in $f/4 \le \lceil E_v/B \rceil \le f$ leaves and make them the new children of $v$.
    3.2. If $|E_v| \le g/4$, store the elements of $E_v$ in $\lceil E_v/B \rceil$ leaves, make these leaves the new children of $v$, and add $v$ to a
         list $\mathcal{U}$ of underfull fringe nodes.
    3.3. If $|E_v| > g$, partition $E_v$ into groups of $g/2$ elements. If the last group has less than $g/4$ elements, merge it with
         the previous group to obtain a group with between $g/2$ and $3g/4$ elements. The $f/2$ blocks in the first group
         become the new children of $v$. For each subsequent group, one at a time, create a new fringe node $w$, make the
         blocks in the group $w$'s children, make $w$ a sibling of $v$, and rebalance the tree using node splits as necessary.
```

Figure 4: The buffer emptying procedure for fringe nodes

by a single buffer emptying operation applied to v's parent and, hence, are already sorted. The elimination of matching Insert and Delete operations in Step 2 takes a single prefix sum computation on the sorted sequence of elements in $\mathcal{B}_v$. We can answer a Find(x) operation in $\mathcal{B}_v$ without propagating it further down the tree if it is preceded by an Insert(x) or Delete(x) operation in $\mathcal{B}_v$. In the former case, x is in the tree at the time of the Find(x) operation; in the latter, it is not. This condition can be checked and the Find(x) operation eliminated in the same prefix sum computation used to eliminate matching Insert and Delete operations. If there is no Insert(x) or Delete(x) operation in $\mathcal{B}_v$ preceding the Find(x) operation, the Find(x) operation needs to be propagated to the appropriate child.

Lemma 3. *Emptying the buffer $\mathcal{B}_v$ of a non-fringe internal node and placing these operations into the buffers of v's children takes $\mathrm{O}(\lceil X/g \rceil \cdot \mathrm{sort}_P(g))$ I/Os, where X is the number of operations in $\mathcal{B}_v$.*

Proof. It takes $\mathrm{O}(\mathrm{sort}_P(g))$ I/Os to sort the first g elements in $\mathcal{B}_v$. By Lemmas 1 and 2 and because $f = g/B \le X/B$, the remainder of the procedure takes $\mathrm{O}(X/PB + \log P) = \mathrm{O}(\lceil X/g \rceil \cdot \mathrm{sort}_P(g))$ I/Os. $\square$

3.2 Emptying Fringe Buffers

After recursively emptying the buffers of non-fringe internal nodes as discussed in Section 3.1, we now process the full fringe nodes in $\mathcal{F}$ one at a time. For each such node v, it is guaranteed that the buffers of its ancestors in the tree are empty, a fact we use when performing node splits or fusions to rebalance the tree.

To empty the buffer of a fringe node v, we invoke the buffer emptying procedure in Figure 4. As in the sequential buffer tree, we first compute the set of elements to be stored in v's children. If this requires fewer or more children than the node currently has, we add or remove children. When adding leaves makes it necessary to rebalance the tree, we do so immediately. When deleting leaves decreases the degree of a fringe node v to less than $f/4$, on the other hand, we add v to a list $\mathcal{U}$ of underful fringe nodes to be rebalanced after the buffers of all fringe nodes in $\mathcal{F}$ have been emptied. The main difference to the sequential buffer tree is that we cannot perform these leaf additions/deletions one leaf at a time.

The sorting of $\mathcal{B}_v$ in Step 1 and the merging of $\mathcal{B}_v$ with the elements in v's children in Step 2 can be implemented as when emptying the buffer of a non-fringe node. In Step 3, making the blocks in a partition of E_v the children of v or of a new fringe node w requires the construction of a list of pointers to these nodes to be stored with v or w. This can be done using a prefix sum computation on the elements in each such group of blocks and thus takes $\mathrm{O}(g/(PB) + \log P) = \mathrm{O}(\mathrm{sort}_P(g))$ I/Os. Every node split necessary to rebalance the tree can be implemented using $\mathrm{O}(\mathrm{sort}_P(g))$ I/Os in a similar fashion.

Once we have applied procedure FringeEmpty to all nodes in $\mathcal{F}$, we process the underfull fringe nodes in $\mathcal{U}$, i.e., nodes that have fewer than $f/4$ children. For each such node v, we traverse the path from v to the root and perform node fusions until we reach a node that is no longer underfull. For a node w to be fused with a sibling, we first empty the buffers of its two immediate siblings. If this triggers recursive buffer emptying steps and adds new overfull fringe nodes to $\mathcal{F}$, we first process these nodes before continuing the processing of w. Then we choose an immediate sibling w' of w and replace w and w' with a single node that has the children of w and w' as its children. If this new node now has more than f children, we split it into two nodes again. Similarly to node splits, a node fusion takes $\mathrm{O}(\mathrm{sort}_P(g))$ I/Os, as it involves emptying two buffers with less than g elements in them and concatenating the $\mathrm{O}(f)$ routing elements and child pointers of two nodes. (Note that if emptying the siblings' buffers triggers recursive buffer emptying operations, this is the result of these buffers overflowing and thus can be charged to the elements in these buffers that are pushed one level down the tree.)

THEOREM 1. *The parallel buffer tree can process a sequence of N* INSERT*,* DELETE*, and* FIND *operations using $\mathrm{O}(\mathrm{sort}_P(N))$ parallel I/Os.*

PROOF. By Lemma 3, emptying each non-fringe buffer containing $X \geq g$ elements takes $\mathrm{O}((X/g) \cdot \mathrm{sort}_P(g))$ I/Os. A similar analysis proves the same bound for emptying a fringe buffer. Since each element is involved in one such buffer emptying procedure per level and the height of the tree is $\mathrm{O}(\log_f(N/B))$, the total cost of emptying buffers is $\mathrm{O}((N/g) \cdot \mathrm{sort}_P(g) \cdot \log_f(N/B)) = \mathrm{O}(\mathrm{sort}_P(N))$ I/Os.

Since every new fringe node we create has at least $g/2$ and at most $3g/4$ elements in its children, and we split/fuse such a node only when its children store more than g or less than $g/4$ elements, the total number of fringe nodes we create/destroy during a sequence of N updates is $\mathrm{O}(N/g)$. Using the analysis of (a, b)-trees from [15], this implies that the total number of node splits/fusions is $\mathrm{O}(N/g)$. Since each such rebalancing operation costs $\mathrm{O}(\mathrm{sort}_P(g))$ I/Os, the total rebalancing cost is also $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os.

Finally, after inserting the last batch of operations into the root buffer, some operations may remain in buffers of internal nodes without being pushed down the tree because the buffers containing them have not overflowed. We force a buffer emptying procedure on each node of the tree in a top-down fashion. This causes at most one buffer emptying operation per node that cannot be charged to an overflowing buffer and hence to the elements in the buffer. Since there are $\mathrm{O}(N/g)$ internal nodes in the tree, the cost of these forced buffer emptying operations is also $\mathrm{O}(\mathrm{sort}_P(N))$. $\square$

4. SUPPORTING RANGE QUERIES

In this section, we show how to support range queries on the parallel buffer tree. Each such RANGEQUERY(x_1, x_2) operation is represented by its query range $[x_1, x_2]$ and is to report all elements that have been inserted before the time of the query, fall within the query range $[x_1, x_2]$, and have not been deleted before the time of the query. As in [5], we assume the sequence of operations to be applied to the buffer tree is *well-formed* in the sense that every INSERT operation inserts an element that *is not* in the buffer tree at the time of the operation, and every DELETE operation deletes an element that *is* in the buffer tree at the time of the operation.

To answer range queries, we again follow the framework of the sequential buffer tree [5], modifying the buffer emptying processes for internal nodes to utilize all available processors. As in [5], the output of each query may be reported in parts, during different buffer emptying operations. In other words, the result of processing a sequence of INSERT, DELETE, and RANGEQUERY operations is an unordered sequence of query-element pairs (q, x), where x is part of the output of query q as defined at the beginning of this section.

We associate a range $\mathcal{R}_v$ with every node v in the buffer tree. For the root r, we define $\mathcal{R}_r := (-\infty, +\infty)$. For each child w_i of a node v with range $\mathcal{R}_v = [x_1, x_2]$, with children $w_1, w_2, \ldots, w_k$, and with routing elements $r_1, r_2, \ldots, r_{k-1}$, we define $\mathcal{R}_{w_i} := [r_{i-1}, r_i]$, where $r_0 := x_1$ and $r_k := x_2$. When emptying the buffer $\mathcal{B}_v$ of a node v with children $w_1, w_2, \ldots, w_k$, we consider all range queries $[x_1, x_2]$ in $\mathcal{B}_v$ and all children w_i of v such that $\mathcal{R}_{w_i} \subseteq [x_1, x_2]$. We output the elements in each such node w_i's subtree that are alive at the time of the query and then forward the query to the buffers of the two children w_h and w_j such that $x_1 \in \mathcal{R}_{w_h}$ and $x_2 \in \mathcal{R}_{w_j}$. One of the endpoints of the query may be outside the range $\mathcal{R}_v$, in which case only one of the nodes w_h and w_j exists.

4.1 Time Order Representation

To support range queries, the sequential buffer tree introduced a *time order representation* of a sequence S' of INSERT, DELETE, and RANGEQUERY operations. We employ the same representation here. In this representation, all DELETE operations are "older" (have an earlier time stamp) than all RANGEQUERY operations, which in turn are older than all INSERT operations. The elements in each of these three groups are ordered by their keys (by their left endpoints in the case of RANGEQUERY operations). The main property of this representation is that the RANGEQUERY operations in S' cannot report any elements affected by INSERT or DELETE operations in S', which follows from the well-formedness of the sequence of operations given to the buffer tree. In general, of course, the input sequence S of the buffer tree is not in time order representation—otherwise range queries would never produce any output. The central operation needed to support range queries on the buffer tree is to transform various subsequences of S into time order representation and report matching query-element pairs in the process. A single sorting step suffices to bring the elements in such a subsequence S' into time order representation. To report the necessary query-element pairs, the sequential buffer tree implements this sorting step using pairwise element swaps, employing the following rules to report matching query-element pairs and eliminate matching pairs of INSERT and DELETE operations. Let o_1 and o_2 be two adjacent operations in S' to be swapped, with o_1 preceding o_2.

- If o_1 is an INSERT operation and o_2 is a DELETE operation affecting the same element, all range queries precede the insertion or succeed the deletion. Thus, o_1 and o_2 can be discarded.

- If o_1 is an INSERT(x) operation and o_2 is a RANGEQUERY(x_1, x_2) operation with $x \in [x_1, x_2]$, then the deletion of x succeeds the range query. Thus, we report x as part of the query's output and then swap the two operations.

- If o_1 is a RANGEQUERY(x_1, x_2) operation and o_2 is a DELETE(x) operation with $x \in [x_1, x_2]$, then the insertion of x precedes the range query. Thus, we report x as part of the query's output and then swap the two operations.

- Any other pair of operations o_1 and o_2 is swapped without any special action being taken.

This construction of a time order representation is inherently sequential. Our first step towards supporting range queries on a parallel buffer tree is to show how to construct a time order representation efficiently in parallel.

LEMMA 4. *Provided $g \geq PB^2 \log^2 P$, a sequence of g operations can be brought into time order representation using $\mathrm{O}(\mathrm{sort}_P(g) + K'/PB)$ I/Os and $\mathrm{O}(g + K')$ space, while also reporting all K' elements that need to be reported according to the swapping rules just discussed.*

PROOF. As already discussed, once all pairs of matching INSERT and DELETE operations have been eliminated, a single sorting step suffices to transform the sequence into time order representation. This takes $\mathrm{O}(\mathrm{sort}_P(g))$ I/Os. Thus, it suffices to discuss how to eliminate these matching pairs of INSERT and DELETE operations and how to report matching query-element pairs before sorting the sequence.

To do the latter, we use the connection between range queries and orthogonal line segment intersection. We map every range query q with query interval $[x_1, x_2]$ and time stamp t to the horizontal segment with endpoints (x_1, t) and (x_2, t). Similarly, we map every element x inserted at time t_1 and deleted at time t_2 to the vertical segment with endpoints (x, t_1) and (x, t_2). Element x is to be reported by query q if and only if the two segments intersect. Thus, to report all matching query-element pairs in the current sequence, we construct this set of horizontal and vertical segments and apply the PEM orthogonal line segment intersection algorithm of [3] to report their intersections using $\mathrm{O}(\mathrm{sort}_P(g) + K'/PB)$ I/Os and $\mathrm{O}(g + K')$ space, where K' is the number of intersections found. Constructing the horizontal segments corresponding to range queries is trivial. To construct the vertical segments corresponding to input elements, we sort the operations so that range queries succeed insertions and deletions and so that insertions and deletions are sorted primarily by the elements they affect and secondarily by their time stamps. Let S be the resulting sequence. Then we turn every matching pair of consecutive INSERT and DELETE operations into a vertical segment. Every DELETE(x) operation at time t not preceded by an INSERT(x) operation is represented as a segment with top endpoint (x, t) and bottom endpoint $(x, -\infty)$. Every INSERT(x) operation at time t not succeeded by a DELETE(x) operation is represented as a segment with bottom endpoint (x, t) and top endpoint $(x, +\infty)$. We apply a parallel prefix sum computation to S to eliminate matching INSERT and DELETE operations immediately prior to sorting the remaining operations in time order. □

LEMMA 5. *Let S_1 and S_2 be two operation sequences in time order representation, with all elements in S_2 older than all elements in S_1. The time order representation of $S_1 \cup S_2$ can be constructed using $\mathrm{O}\Big(\frac{|S_1|+|S_2|}{g} \cdot \mathrm{sort}_P(g) + K/PB\Big)$ I/Os and $\mathrm{O}(g+K)$ space, where K is the number of elements reported in the process.*

PROOF. We "merge" S_1 and S_2 using the algorithm in Figure 5. The correctness proof of this procedure is straightforward and therefore omitted. By Lemma 1, Steps 1 and 4 take $\mathrm{O}\left(\frac{|S_1|+|S_2|}{PB} + \log P\right) = \mathrm{O}\left(\frac{|S_1|+|S_2|}{g} \cdot \mathrm{sort}_P(g)\right)$ I/Os. Step 3 is identical to Step 2. Thus, it suffices to analyze Step 2. The total cost of splitting list L in Step 2.2.1 is $\mathrm{O}((N + K)/PB)$ I/Os because every element in L either becomes part of L' or of L''. In the former case, it adds at least one element to the output. In the latter case, this is the last iteration it is part of L. Every invocation of the line segment intersection algorithm on $2g$ elements in Step 2.2.2 can be charged either to g queries in L'' or to the g deletions in D'. Hence, the total cost of Step 2.2.2 is $\mathrm{O}\left(\frac{|S_1|+|S_2|}{g} \cdot \mathrm{sort}_P(g) + K/PB\right)$ I/Os. In Step 2.2.3, we report at least one query-element pair per deletion in D'. Since D' contains $g \geq PB$ deletions unless this is the last batch, it is trivial to report the query-element pairs in this step using $\mathrm{O}(K/PB)$ I/Os. Every iteration of Step 2.2.4 except the last can be charged to g queries retrieved from Q_2. Hence, the total cost of this step is also $\mathrm{O}\left(\frac{|S_1|+|S_2|}{g} \cdot \mathrm{sort}_P(g) + K/PB\right)$ I/Os. □

LEMMA 6. *Let $\mathcal{T}$ be a buffered range tree with N/B leaves and each of whose buffers stores at most g elements. Emptying all buffers of $\mathcal{T}$ and collecting their operations in time order representation takes $\mathrm{O}\Big(\mathrm{sort}_P(g) \sum_{x \in \mathcal{T}} h_x + (N + K)/PB\Big)$ I/Os, where the sum is over all operations x in buffers in $\mathcal{T}$, h_x is the distance of the buffer storing operation x from the leaf level, and K is the number of elements reported in the process.*

PROOF. The algorithm is almost the same as for the sequential buffer tree, except that we use Lemmas 1, 4, and 5 to implement its basic steps efficiently in parallel.

First we collect the buffer contents of the nodes on each level in a single sequence. This takes $\mathrm{O}(N/PB)$ I/Os. Since each buffer contains at most g operations, we can divide the sequence representing each level into subsequences of $\Theta(g)$ elements such that each buffer content is contained in such a subsequence. By Lemma 4, we can transform all of these subsequences into time order representation using $\mathrm{O}((X/g)\mathrm{sort}_P(g) + K/PB)$ I/Os, where X is the total number of operations in $\mathcal{T}$'s buffers. Next we merge the deletion, range query, and insertion sequences of the time order representations at each level to obtain a single time order representation of all operations at this level. This takes $\mathrm{O}(X/(PB) + \log P) = \mathrm{O}((X/g)\mathrm{sort}_P(g))$ I/Os. Let $S_1, S_2, \ldots, S_k$ be the resulting time order sequences of all levels, sorted from the fringe nodes to the root. The operations in S_{i+1} are younger than the operations in S_i, for all $1 \leq i < k$. Thus, we can apply Lemma 5 to merge S_{i+1} into S_i, for $i = k-1, k-2, \ldots, 1$. Each such merge step has cost $\mathrm{O}\Big(\frac{|S_i|+|S_{i+1}|}{g} \cdot \mathrm{sort}_P(g) + K'/PB\Big)$ I/Os, where K' is the size of the output it produces. Since each operation in a sequence S_i participates in i such merge steps, the lemma follows. □

4.2 Emptying Buffers

The buffer emptying process is similar to the one in the sequential buffer tree. The main difference is that we use the results from Section 4.1 to construct time order representations as needed and that we need to ensure the reporting of output elements during each buffer emptying process is distributed evenly across processors. As in Section 3, the buffer emptying processes for non-fringe and fringe nodes differ. These procedures are shown in Figures 6 and 7.

```
TimeOrderMerge($S_1, S_2$):
1. Merge the subsequences $D_1$ and $I_2$ of deletions in $S_1$ and insertions in $S_2$ using Lemma 1, delete matching Insert and
   Delete operations, and arrange the remaining insertions and deletions in time order $D_1' I_2'$ using a parallel scan of the
   merged list to split it into two output streams $D_1'$ and $I_2'$.
2. Swap the sequences $D_1'$ and $Q_2$ and report all pairs $([x_1, x_2], x)$ such that $Q_2$ contains a range query with interval $[x_1, x_2]$
   and $D_1'$ contains a Delete($x$) operation with $x \in [x_1, x_2]$. The swapping of $D_1'$ and $Q_2$ is easily achieved using a parallel
   scan. We report the query answers as follows:
   2.1. Create a list $L$ of "long" queries. Initially, this list is empty.
   2.2. Divide $D_1'$ into batches of $g$ deletions. For each batch $D'$, do the following:
        2.2.1. Using a parallel scan of $L$, divide $L$ into two lists $L'$ and $L''$ such that all queries in $L'$ have their right
               endpoints to the right of the last element in $D'$ and the queries in $L''$ do not.
        2.2.2. Divide $L''$ into batches of size $g$ and, for each batch, report the queries in the batch containing each
               element in $D'$ using orthogonal line segment intersection as in Lemma 4. After processing all elements
               in $L''$ in this fashion, discard $L''$.
        2.2.3. Every query in $L'$ contains all elements in $D'$. Distribute the elements in $D'$ evenly across processors.
               Then let each processor report the query-element pairs defined by its assigned deletions and the queries
               in $L'$. After processing $L'$ in this fashion, set $L := L'$.
        2.2.4. While the next element in $Q_2$ has a left boundary no greater than the rightmost deletion in $D'$, read the
               next batch $Q'$ of queries from $Q_2$. Run the line segment intersection algorithm on $D'$ and $Q'$ to report
               all matches between queries in $Q'$ and deletions in $D'$. Then apply a parallel scan to $Q'$ to identify all
               queries whose right endpoints are to the right of the rightmost deletion in $D'$. Append these queries
               to $L$.
3. Swap the sequences $I_2'$ and $Q_2$ and report all pairs $([x_1, x_2], x)$ such that $Q_1$ contains a range query with interval $[x_1, x_2]$
   and $I_2'$ contains an Insert($x$) operation with $x \in [x_1, x_2]$. This is analogous to Step 2.
4. Merge $D_1'$ with $D_2$, $Q_1$ with $Q_2$, and $I_1$ with $I_2'$ using Lemma 1.
```

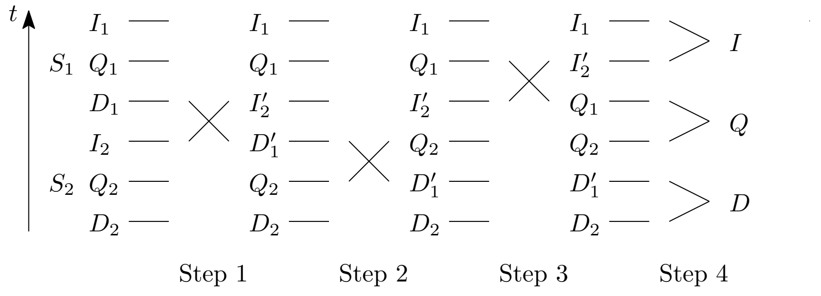

Figure 5: Merging two sequences $S_1 = D_1 Q_1 I_1$ and $S_2 = D_2 Q_2 I_2$ in time order representation

To empty the buffer $\mathcal{B}_v$ of a non-fringe node v, we first compute its time order representation (Step 1) and distribute the Delete operations in the resulting sequence to the children of v (Step 2). Then we inspect the range queries in $\mathcal{B}_v$ and identify all children w_i of v that are spanned by at least one range query in $\mathcal{B}_v$ (Step 3). We do this by merging the set of range queries in $\mathcal{B}_v$ with the set of children of v, each represented by its range $\mathcal{R}_{w_i}$. In the resulting list, the range queries and children of v are sorted by the left endpoints of their ranges. A prefix sum computation now suffices to label each child w_i of v with the rightmost right boundary of any range query preceding $\mathcal{R}_{w_i}$ in the merged list. $\mathcal{B}_v$ contains a range query spanning w_i if and only if this rightmost right boundary is to the right of the right boundary of $\mathcal{R}_{w_i}$. An analogous procedure allows us to mark every range query in $\mathcal{B}_v$ that spans at least one child of v, and a prefix sum over the list of range queries followed by a parallel scan suffices to extract the subset Q' of all marked queries.

In Step 4, we compute the lists of elements to be reported from the subtrees of all marked children of v and answer range queries pending in each such subtree $\mathcal{T}_{w_i}$. The correctness of this procedure is not difficult to see. The reason for first excluding the set of deletions just sent to such a child w_i in Step 4.1 and then, in Step 4.3, merging them with the time order representation constructed in Step 4.2 is that Lemma 6 requires that each buffer in $\mathcal{T}_{w_i}$ contains at most g operations. Similar to Section 3, we postpone the emptying of fringe buffers in $\mathcal{T}_{w_i}$ because emptying these buffers might trigger rebalancing operations that could interfere with buffer emptying procedures still pending at non-fringe nodes. A difference to Section 3 is that we schedule even non-full fringe buffers of $\mathcal{T}_{w_i}$ to be emptied once we are done emptying non-fringe buffers (Step 4.8) if these buffers contain more than $g/16$ operations or the leaves below the corresponding nodes store fewer than $g/8$ elements after Step 4.5. This is necessary to ensure that we can charge the cost of merging L_{w_i} with the contents of the leaves of $\mathcal{T}_{w_i}$ to elements deleted from $\mathcal{T}_{w_i}$ or reported by range queries in $\mathcal{B}_v$ that span $\mathcal{R}_{w_i}$.

In Step 5, we report all the elements in subtrees spanned by range queries in $\mathcal{B}_v$, for all queries spanning these subtrees. The key in this step is to balance the work of reporting output elements among the processors. We process the queries spanning the ranges of children of v in batches of g

```
RangeNonFringeEmpty($v$):
  1. Compute the time order representation $S = DQI$ of $\mathcal{B}_v$. The first $g$ operations can be brought into time order representation using Lemma 4. The remaining operations are already in time order representation because they were added to $\mathcal{B}_v$ during a single buffer emptying process at $v$'s parent. Thus, they can be merged with the time order representation of the first $g$ operations using Lemma 5.
  2. Distribute the Delete operations in $D$ to the buffers of the relevant children of $v$ using Lemma 2.
  3. Mark all children of $v$ that are spanned by at least one range query in $Q$ and compute the subset $Q' \subseteq Q$ of range queries that span at least one child of $v$.
  4. For each marked child $w_i$ of $v$, do the following, where $\mathcal{T}_{w_i}$ denotes the subtree with root $w_i$:
     4.1. Set aside the set $D_{w_i} \subseteq D$ of all Delete operations distributed to $\mathcal{B}_{w_i}$ in Step 2.
     4.2. Using Lemma 6, empty all buffers of internal nodes of $\mathcal{T}_{w_i}$ and compute a time order representation $L_{w_i}$ of the operations in these buffers.
     4.3. Merge the deletions in $D_{w_i}$ into $L_{w_i}$ using Lemma 5.
     4.4. Distribute copies of the operations in $L_{w_i}$ to the buffers of the fringe nodes of $\mathcal{T}_{w_i}$ according to the routing elements stored in $\mathcal{T}_{w_i}$. (Range queries are sent to the leaves containing their endpoints.) This can be done using Lemma 1 by merging $L_{w_i}$ with the sequence of routing elements of the fringe nodes after creating copies of the RangeQuery operations, sorting these copies by their right endpoints, and merging them into $L_{w_i}$.
     4.5. Mark all fringe nodes spanned by at least one range query in $L_{w_i}$ and construct the list $Q'_{w_i}$ of all range queries in $L_{w_i}$ that span at least one fringe node. This can be done similarly to Step 3. For each marked fringe node $u$, apply the deletions in its buffer $\mathcal{B}_u$ to the leaves below it and remove the deletions from $\mathcal{B}_u$. Now answer the range queries in $Q'_{w_i}$ in a manner similar to Step 5 below.
     4.6. Remove all RangeQuery operations from $L_{w_i}$. Merge the list of elements stored in the leaves of $\mathcal{T}_{w_i}$ into $L_{w_i}$ using Lemma 1 and eliminate matching Insert and Delete operations using a prefix sum computation.
     4.7. Store the value of $|L_{w_i}|$ with node $v$.
     4.8. Add each fringe node of $\mathcal{T}_{w_i}$ whose buffer contains at least $g/16$ operations or whose leaves store less than $g/8$ elements to the list $\mathcal{F}$ of fringe nodes to be emptied after we are done emptying non-fringe nodes.
  5. Divide $Q'$ into batches of $g$ RangeQuery operations. For each such batch $Q''$, do the following:
     5.1. For each range query $q = [x_1, x_2] \in Q''$, determine the range $w_h, w_{h+1}, \ldots, w_j$ of children of $v$ such that $\mathcal{R}_{w_i} \subseteq [x_1, x_2]$, for all $h \le i \le j$, and create copies $q_{w_h}, q_{w_{h+1}}, \ldots, q_{w_j}$ of $q$. Let $wt(q_{w_i}) = |L_{w_i}|$ be the weight of $q_{w_i}$, for all $h \le i \le j$.
     5.2. For each query $q_{w_i} \in Q''$, report the elements in $L_{w_i}$. Distribute the load of reporting the output of all queries evenly among the processors.
  6. Distribute all range queries in $Q$ to the children of $v$ so that each query $q = [x_1, x_2] \in Q$ is sent to children $w_h$ and $w_j$ satisfying $x_1 \in \mathcal{R}_{w_h}$ and $x_2 \in \mathcal{R}_{w_j}$, if these children exist. When sending query $[x_1, x_2]$ to a child $w_i$, replace the query range with $[x_1, x_2] \cap \mathcal{R}_{w_i}$.
  7. Distribute the insertions in $I$ to the buffers of the appropriate children of $v$. For each marked child $w_i$, distribute the operations sent to $w_i$ in Steps 6 and 7 to the fringe buffers of $\mathcal{T}_{w_i}$. This can be done in a way similar to Step 4.4.
  8. If the buffer of any child node $w_i$ is now full, and $w_i$ is not a fringe node, recursively empty $w_i$'s buffer. If $w_i$'s buffer is full and $w_i$ is a fringe node, add $w_i$ to the list $\mathcal{F}$ of fringe nodes to be emptied after we are done emptying non-fringe nodes.
```

Figure 6: Emptying the buffer of a non-fringe node of the parallel buffered range tree

queries. For each batch Q'' and each query $q \in Q''$, we first create a separate copy q_{w_i} of q for each subtree $\mathcal{T}_{w_i}$ such that q spans $\mathcal{R}_{w_i}$ (Step 5.1). We do this as follows: A single prefix sum computation similar to Step 3 suffices to label every query in Q'' with the leftmost child of v it spans. We sort the queries in Q'' by their right endpoints and repeat this procedure to label each query with the rightmost child of v it spans. Then we return the queries to their original order. Using a prefix sum computation, we can count the total number of copies of queries in Q'' to be created. Next we distribute the creation of these copies evenly among the processors: If C is the total number of copies to be created, we use a parallel scan to divide Q'' into two subsequences, those queries with no more than C/P copies to be created and those with more than C/P copies to be created. Using a prefix sum computation, we partition the first sequence of queries into P subsequences, one per processor, so that no processor has to create a total of more than $2C/P$ copies. Each processor then produces the copies of its assigned queries. Using a second prefix sum computation, we assign processors to each query in the second sequence so that each query q with $C_q > C/P$ copies to be created is assigned at least $\lceil PC_q/C \rceil$ processors and each processor is assigned to at most two queries. We divide the creation of copies of each such query evenly among its assigned processors, which ensures that no processor creates more than $2C/P$ copies. The load balancing in Step 5.2 is achieved analogously, based on the weights assigned to the copies of the queries. Steps 6–8 are analogous to the buffer emptying process for non-fringe nodes in Section 3.

After we have emptied the buffers of all full non-fringe nodes, we need to empty the buffers of all fringe nodes in $\mathcal{F}$. When emptying the buffer of a fringe node v (Figure 7), the merging of the buffer $\mathcal{B}_v$ with the elements in the leaves below v answers all range queries in $\mathcal{B}_v$. Thus, after the merge, we can remove the RangeQuery operations from $\mathcal{B}_v$ and construct the new leaf contents from the current leaf contents and the Insert and Delete operations remaining in the fringe buffer, followed by rebalancing the tree as in Section 3.

```
RangeFringeEmpty($v$):
  1. Construct the time order representation of $\mathcal{B}_v$ as in Step 1 of Figure 6.
  2. Merge the elements in the children of $v$ into $\mathcal{B}_v$ using Lemma 5.
  3. Remove all range queries from $\mathcal{B}_v$.
  4. Empty the buffer $\mathcal{B}_v$ (and rebalance the tree) using algorithm FringeEmpty in Figure 4.
```

Figure 7: Emptying the buffer of a fringe node of the parallel buffered range tree

Theorem 2. *The total cost of an arbitrary sequence of N Insert, Delete, and RangeQuery operations performed on an initially empty range tree is $O(\mathrm{sort}_P(N) + K/PB)$ I/Os, where K is the number of reported elements. The data structure uses $O(N + K)$ space.*

Proof. The correctness of the algorithm is easy to see because all range queries are answered correctly during the construction of the appropriate time order representations (Lemmas 4 and 5) and by reporting the elements in subtrees completely spanned by these queries (Steps 4 and 5 of Figure 6).

To prove the I/O bound, we assign $\Theta(\frac{1}{PB}\log_{M/B} N/B) = \Theta(\frac{1}{PB} \cdot \log_{g/B} N/B \cdot \log_{M/B} g/B)$ credits to each operation when inserting it into the root buffer of the buffer tree and maintain the invariant that each element in the buffer $\mathcal{B}_v$ of a node v that is the root of a subtree of height h has $\Theta(\frac{h}{PB} \cdot \log_{M/B} g/B)$ credits remaining. We show that we can use these credits plus $O(1/PB)$ credits per output element to pay for the cost of all buffer emptying operations, excluding the cost of rebalancing operations triggered by the emptying of fringe buffers. Since the rebalancing cost is the same as in the basic parallel buffer tree discussed in Section 3, and the proof of Theorem 1 bounds this cost by $O(\mathrm{sort}_P(N))$ I/Os, the I/O bound follows.

Consider the emptying of a non-fringe buffer $\mathcal{B}_v$. The cost of Steps 1, 2, 3, 6, and 7 is $O((|\mathcal{B}_v|/g)\mathrm{sort}_P(g) + K'/PB)$ I/Os, where K' is the number of output elements produced by these steps. For Steps 1, 2, 6, and 7, this follows from Lemmas 2, 4, and 5. For Step 3, this follows because this step involves only merging two sequences of total size $O(|\mathcal{B}_v|)$ (Lemma 1), followed by a prefix sum and parallel scan of the merged list. Since all operations in $\mathcal{B}_v$ are either eliminated or moved to v's children, the credits of these operations and the produced output elements can pay for the cost of these steps.

In Step 4, consider a single subtree $\mathcal{T}_{w_i}$ with N'/B leaves. By Lemma 6, Step 4.2 takes $O\Big(\mathrm{sort}_P(g)\sum_{x\in\mathcal{T}_{w_i}} h_x + (N' + K')/PB\Big)$ I/Os, where K' is the number of elements reported in this step. Steps 4.1 and 4.3 take $O((X/g)\mathrm{sort}_P(g) + K'/PB)$ I/Os, where X is the total number of operations in $\mathcal{T}_{w_i}$'s buffers and K' is again the number of elements reported by this step. Step 4.4 takes $O(N'/PB + \log P)$ I/Os, by Lemma 1. Step 4.5 takes $O\left(\frac{N'+K'}{PB} + \log P\right)$ I/Os, by the same analysis as for Steps 3 and 5 (below). Steps 4.6–4.8, finally, take $O\left(\frac{N'+X}{PB} + \log P\right)$ I/Os, by Lemma 1. The $O(K'/PB)$ terms can be paid for by the reported elements. The remaining cost is dominated by the cost of Step 4.2. The $O\Big(\mathrm{sort}_P(g)\sum_{x\in\mathcal{T}_{w_i}} h_x\Big)$ term can be paid for by the credits of the operations in $\mathcal{T}_{w_i}$'s buffers because these operations are moved to fringe buffers. The $O(N'/(PB))$ term can be paid for by the elements in the leaves of $\mathcal{T}_{w_i}$ because a constant fraction of these elements either get deleted or become part of the output of at least one range query in $\mathcal{B}_v$ that spans $\mathcal{R}_{w_i}$.

The cost of Step 5 is bounded by $O(f/P + K'/(PB)) = O\left(\frac{g+K'}{PB}\right)$ I/Os because we ensure that each processor creates roughly the same number of copies of queries and reports roughly the same number of output elements, and the number of copies of queries we create is $O(K')$ because we only copy queries that span the ranges of children of v.

The analysis of the cost of emptying fringe buffers is analogous once we observe that each such buffer emptying operation can be charged to $\Omega(g)$ operations: either the emptied buffer contains at least $g/16$ operations or the leaves below the corresponding node must have lost at least $3g/8$ elements since their creation, which can only happen as a result of Delete operations.

The space bound follows because all steps in our algorithms use linear space. The only exception is the orthogonal line segment intersection algorithm of [3], which requires $O(N + K)$ space in order to achieve the optimal I/O complexity. □

5. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] D. Ajwani, N. Sitchinava, and N. Zeh. Geometric algorithms for private-cache chip multiprocessors. In *ESA*, 2010.

[3] D. Ajwani, N. Sitchinava, and N. Zeh. I/O-optimal distribution sweeping on private-cache chip multiprocessors. In *IPDPS*, 2011.

[4] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of massive data sets*, pages 313–357. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[5] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[6] L. Arge, M. T. Goodrich, M. J. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, pages 197–206, 2008.

[7] L. Arge, M. T. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. In *IPDPS*, 2010.

[8] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[9] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and*

architectures, SPAA '05, pages 228–237, New York, NY, USA, 2005. ACM.

[10] G. Blelloch, P. Gibbons, and H. Simhadri. Low depth cache-oblivious algorithms. In *SPAA*, pages 189–199, 2010.

[11] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, pages 501–510, 2008.

[12] M. Carey and C. Thompson. An efficient implementation of search trees on [lg n + 1] processors. *Computers, IEEE Transactions on*, C-33(11):1038 –1041, nov. 1984.

[13] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming for multicores. In *SPAA*, pages 207–216, 2008.

[14] R. Cole and V. Ramachandran. Resource-oblivious sorting on multicores. In *ICALP*, pages 226–237, 2010.

[15] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

[16] Intel Corp. Futuristic Intel chip could reshape how computers are built, consumers interact with their PCs and personal devices. Press release: `http://www.intel.com/pressroom/archive/releases/2009/20091202comp_sm.htm`, Dec. 2009. Retrieved April 28, 2012.

[17] H. Park and K. Park. Parallel algorithms for red-black trees. *Theor. Comput. Sci.*, 262(1-2):415–435, July 2001.

[18] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *ICALP*, pages 597–609, 1983.

[19] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylonen. Concurrency control in B-trees with batch updates. *Knowledge and Data Engineering, IEEE Transactions on*, 8(6):975 –984, dec 1996.

[20] N. Sitchinava. *Parallel external memory model – a parallel model for multi-core architectures*. PhD thesis, University of California, Irvine, 2009.

[21] V. Srinivasan and M. J. Carey. Performance of B+ tree concurrency control algorithms. *The VLDB Journal*, 2:361–406, October 1993.

[22] L. G. Valiant. A bridging model for parallel computation. In *ESA*, 2008.