A Parallel Tree Difference Algorithm

D.B. Skillicorn
skill@qucis.queensu.ca

March 1995

External Technical Report
ISSN-0836-0227-
95-381

Department of Computing and Information Science

Queen's University

Kingston, Ontario K7L 3N6

Document prepared March 6, 1995

Abstract

We present a tree difference algorithm with expected sequential execution time $O(n \log\log n)$ and expected parallel execution time of $O(\log n)$, for trees of size n. The algorithm assumes unique labels and permits operations only on leaves and frontier subtrees. Despite these limitations, it can be useful in the analysis of structured text.

1 Applications of Tree Difference

In this paper we describe an algorithm for determining the difference between two trees under the assumption that each node has a unique label chosen from an ordered set. The algorithm uses a novel form of hashing to quickly extract neighbourhood information for each node. In a second phase, this neighbourhood information is processed to determine what differences exist between the trees. We assume that trees may be arbitrarily branching, and that the following operations may have been applied to them:

1. a node was inserted to become a new leaf;
2. a leaf node was deleted;
3. a leaf node was moved to become a leaf node in another part of the tree;
4. a subtree was inserted below any node;
5. a subtree was deleted;
6. a subtree was moved to another part of the tree

In reporting differences, preference is given to subtree operations since they provide more compact and meaningful descriptions of changes, for example, "a section has moved" rather than "each paragraph of a section has moved". The algorithm could be enhanced at the processing stage to detect other possible tree operations, at the expense of execution time.

The tree difference algorithm for trees with n nodes has expected sequential time complexity $O(n \log \log n)$ and parallel time complexity $O(\log n)$ on the EREW PRAM model.

Other tree operations have been considered in the literature. For example, there is a sequence of algorithms based on insertions and deletions where an insertion below a node x allows the new node to acquire some of x's children as its children, and hence x's grandchildren. A symmetric deletion moves the children of a deleted node up to become the children of the parent of the deleted node [8].

The assumption that each tree node has a unique label is probably not supportable for today's techniques for storing such trees. However, attributed file systems such as DFR [6] require such identifiers for each node in the file system, so we expect this assumption to quickly become realistic. In any case, non-unique labels force difference algorithms to carry out some variation of dynamic programming, for which time bounds are necessarily much worse.

The application domain envisaged is structured text. Structured text contains tags, that indicate and delimit regions of the text of semantic interest. SGML is the standard for this kind of markup. SGML tagged text is naturally modelled by trees, in which internal nodes model entities with opening and closing tag pairs (and often attributes, such as text). Leaf nodes correspond to unstructured text, that is the basic entities from which a text is built up such as sentences. A given entity will usually have many entities contained within it, such as a section containing many paragraphs, so trees are arbitrarily branching.

There are many applications in which the ability to detect differences between two structured texts is important. Many documents get produced by collaborations in which one person acts as author for some period, and then passes the document to a second person, and so on. When the document returns to an earlier author, it may be difficult to detect what changes have been made. It is unworkably tedious to require each author to log all changes, and maintain this list with the document. The ability to directly compare the original with the returning version permits each author to recreate the effect of the changes. This is actually better than keeping a log, since it computes only the net effect of the changes and not the actual mechanism by which the changes were effected. Maintaining versions of software is a very similar problem.

Many hypertext systems now store links separately from the documents to which they point, because of the flexibility it gives to control access to documents. Doing so means, however, that alterations to documents must trigger checks of the link data to make sure that they remain consistent. This is expensive since it requires a pass through the entire data, and may need to be done after every edit of a document. The ability to detect changes after the fact makes it possible to update links periodically rather than after every change.

A practical difference algorithm such as the one described here is an effective tool in such environments. The size of texts, and especially software systems, is such that a parallel difference algorithm may be required for acceptable performance.

2 Background

The algorithm for tree difference described here uses two technical constructions from the literature. The first is *universal hashing* which enables hash tables for arbitrary key sets to be built in almost-constant parallel time on the EREW PRAM model. The second is results on fast implementation of certain tree homomorphisms, particularly upwards and downwards accumulations, developed for the categorical data type of trees. Both types of accumulations can be computed in $O(\log n)$ parallel time on the EREW PRAM.

A hash function maps an arbitrary set of n keys K, whose maximum length is bounded, to a set of bins $0, 1, \ldots, b-1$ described by a positive integer. A uniform hash function [10] produces random outputs uniformly distributed over $[0, b-1]$ for arbitrary key sets. It does this by ensuring that every bit of output depends on all of the bits of the input, and that the probability of a 0 or 1 in every position of the output is the same (that is, 1/2). Of course, all hash functions are limited in the sense that the set of inputs that hash to the same bin can themselves be considered a set of inputs on which the hash function performs badly. In practical terms, however, the expected value of the longest chain in a bin, when direct chaining is used, grows as , $^{-1}(b)$ which is $O(\log b / \log\log b)$ [5]. Using a binary tree instead of a chain reduces this to $O(\log\log b)$ since the uniform distribution guarantees that the tree will be balanced. Furthermore, the expected length of the longest chain or tree is independent of the occupation factor of the hash table, that is the ratio of n to b. Since each key is hashed independently a hash table for n keys can be created in parallel in time $O(\log\log b)$ with high probability.

We model structured text by trees with arbitrary branching factor whose nodes are chosen from an enumerated type of entities with attributes. Objects of this type are known as *rose trees* [3] defined as follows:

Definition 1 A rose tree is that type, $RT(A)$, whose constructors are:

$$\begin{aligned} RTLeaf &: A \rightarrow RT(A) \\ RTJoin &: A \times RT(A)* \rightarrow RT(A) \end{aligned}$$

where $RT(A)*$ denotes non-empty join lists of rose trees.

Such trees are either single nodes of type A, or a list of subtrees, joined together by an internal node of type A, where A is a suitable enumerated type of entities.

A function $h : RT(A) \rightarrow X$ is a homomorphism if and only if there are a pair of functions p_1 and p_2 of types

$$\begin{aligned} p_1 &: A \rightarrow X \\ p_2 &: A \times X* \rightarrow X \end{aligned}$$

such that

$$\begin{aligned} h(RTLeaf(a)) &= p_1(a) \\ h(RTJoin(a, [t_1, t_2, \ldots, t_n])) &= p_2(a, h*[t_1, t_2, \ldots, t_n]) \end{aligned}$$

where $h*$ is the mapping of h over a list of trees [9]. This one-to-one correspondence between rose tree homomorphisms and such function pairs justifies the notation $Hom(p_1, p_2)$ for h.

```
eval_homomorphism(p1, p2, t)
case t of
    $RTLeaf$ (a) : return p1 ( a )
    $RTJoin$ (a, [ti]) : return p2 ( a,
                                        eval_homomorphism(p1, p2, _)* [ti] )
end
```

Figure 1: Recursive Schema for Rose Tree Homomorphisms

It is straightforward to see from the definition of rose tree homomorphisms that all such homomorphisms can be evaluated recursively as shown in Figure 1. This schema can be used directly to evaluate rose tree homomorphisms in sequential time linear in the size of tree, and parallel time linear in the height of the tree, provided that p_1 and p_2 are constant time.

Certain homomorphisms have specialised implementations that enable them to be computed more efficiently. A *rose tree map* is a tree homomorphism that applies a function $p : A \rightarrow X$ to every node of a rose tree. Thus

$$RTMap(p) : RT(A) \rightarrow RT(X)$$

A rose tree map can implemented sequentially in linear time and in parallel in constant time.

Another special class of homomorphisms are the *upwards accumulations*. These are homomorphisms in which the data dependencies can be satisfied by data flowing up the tree from leaves to root, and in which the result at each node can be computed incrementally from the results of its children.

Definition 2 *(Upwards Accumulation)* Given an arbitrary rose tree homomorphism $Hom(p_1, p_2) : RT(A) \rightarrow X$, the upwards accumulation $\Uparrow (p_1, p_2)$ is the function

$$\Uparrow (p_1, p_2) : RT(A) \rightarrow RT(X)$$

given by

$$\Uparrow (p_1, p_2) = RTMap(Hom(p_1, p_2)) \cdot subtrees$$

where $subtrees$ is the function that replaces each node of a tree by the subtree rooted at that node. Both $RTMap(Hom(p_1, p_2))$ and $subtrees$ are homomorphisms, so upwards accumulations are also.

An upwards accumulation is show in Figure 2. Upwards accumulations on rose trees can be efficiently computed by:

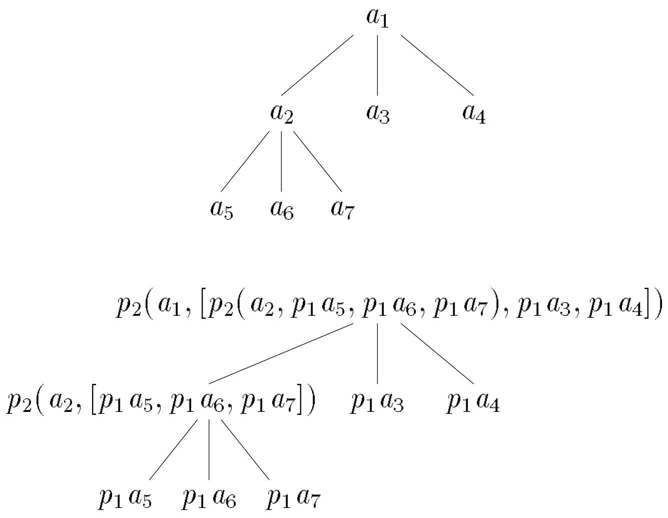
a_1
a_2
a_3
a_4
a_5
a_6
a_7
$p_2(a_1, [p_2(a_2, p_1 a_5, p_1 a_6, p_1 a_7), p_1 a_3, p_1 a_4])$
$p_2(a_2, [p_1 a_5, p_1 a_6, p_1 a_7])$
$p_1 a_3$
$p_1 a_4$
$p_1 a_5$
$p_1 a_6$
$p_1 a_7$

Figure 2: An Upwards Accumulation

1. transforming the rose tree into an equivalent binary tree, and
2. using an extension of *tree contraction* to compute the desired result on the binary tree.

A rose tree can be straightforwardly be converted to a binary tree as shown in Figure 3. In a parallel setting, this conversion requires the processor responsible for each node to be able to find the address of the the processor responsible for its right sibling in constant time. This information is stored in the processor responsible for the parent, and each child accesses a different part of it. It is also possible to number the leaves of the resulting binary tree from zero using an extension of the Euler tour technique in parallel logarithmic time under these same assumptions about storage arrangement.

Tree contraction [1, 2, 7] computes a reduction on a binary tree in parallel logarithmic time, regardless of how skewed the tree might be. The idea is that the sequential dependency along the longest path from root to leaves can be avoided by doing useful work, on every step, at nodes where some descendants are leaves. Since, at any stage, about half of the nodes are leaves, this creates the opportunity to reduce the total time of the reduction algorithm to logarithmic in the tree size, regardless of the tree's structure.

A function is associated with each node of the tree, and a pair of operations *contractl* and *contractr* are applied at nodes where one or more children are leaves. The contraction operations, one of which is shown in Figure 4, replace two internal nodes of the tree and one leaf node by a single internal node, forming partial compositions of the functions associated

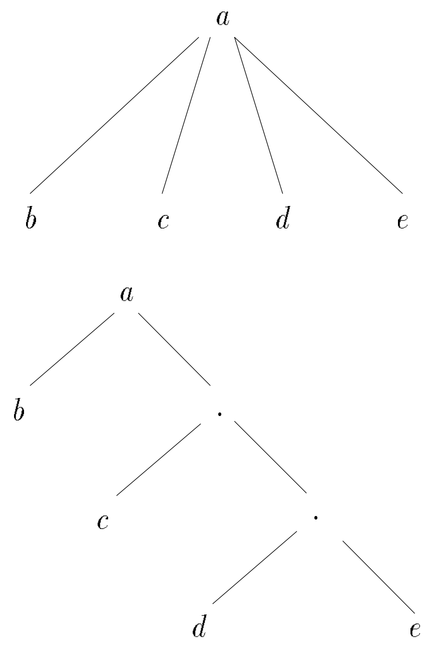

Figure 3: Local Rearrangement of a Rose Tree into a Binary Tree

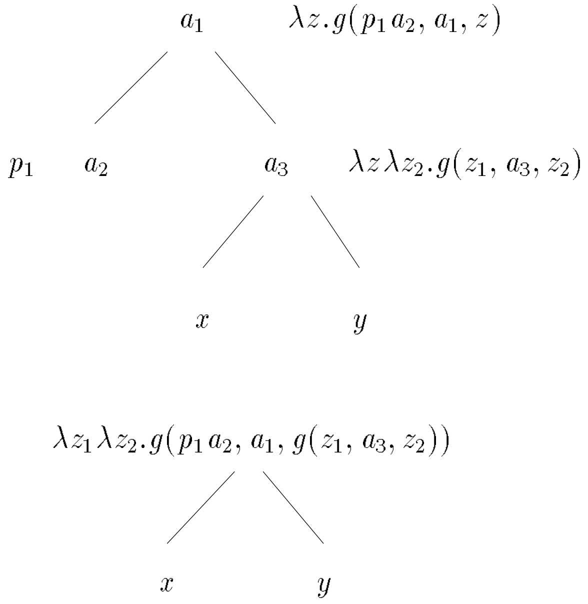
a_1
$\lambda z.g(p_1 a_2, a_1, z)$
p_1
a_2
a_3
$\lambda z \lambda z_2.g(z_1, a_3, z_2)$
x
y
$\lambda z_1 \lambda z_2.g(p_1 a_2, a_1, g(z_1, a_3, z_2))$
x
y

Figure 4: The *contractl* Operation

with the nodes. As long as these partial compositions can be partially evaluated, real progress is made. For the overall algorithm to complete in logarithmic parallel time, it must be possible to carry out the partial compositions and evaluations in constant time, and the results must be constant space. These restrictions are fairly mild for ordinary algebraic and boolean operations. Tree contraction can be extended to upwards accumulations under the same restrictions on the component functions [4].

So that the operations at internal nodes of the rose tree can be distributed over the nodes of the binary tree constructed from it, the component function p_2 must be expressible as

$$p_2 = g' \cdot id \times \oplus /$$

where $\oplus /$ is a list reduction with an associative operation $\oplus$. The functions associated with each node of the binary tree initially are then:

- $\oplus$ for the internal dummy nodes,
- p_1 for the leaf nodes, and
- $g' \cdot id \times \oplus$ for the other internal nodes (e.g. a).

Thus $\oplus$ and $g' \cdot id \times \oplus$ must satisfy the requirements for tree contraction, that is their compositions must be constant time and space.

Under these conditions, upwards accumulations with well-behaved component functions can be computed in parallel logarithmic time, and linear sequential time.

A third special class of homomorphisms are the *downwards accumulations*. These are operations in which data flows down the tree and each node computes a result that depends on the path between it and the root and the values along that path. We represent the paths between the root and other nodes by join lists with two kinds of join operations, one representing a left child and the other a right child. As before it suffices to consider the case of binary trees, since we will convert rose trees to binary trees before applying these homomorphisms. Call the type of these join lists *Paths*. Path homomorphisms have component functions

$$\begin{aligned} p_1 &: A \rightarrow P \\ p_2 &: P \times P \rightarrow P \\ p_3 &: P \times P \rightarrow P \end{aligned}$$

with p_2 and p_3 mutually associative. We write them $PathHom(p_1, p_2, p_3)$.

Definition 3 *(Downwards Accumulation)* For an arbitrary path homomorphism with component functions p_1, p_2, and p_3 with codomains of type X, a downwards accumulation $\Downarrow (p_1, p_2, p_3)$ is the function

$$\Downarrow (p_1, p_2, p_3) : T(A, B) \rightarrow T(X, X)$$

given by

$$\Downarrow (p_1, p_2, p_3) = TreeMap(PathHom(p_1, p_2, p_3)) \cdot paths$$

where $paths$ is the function that replaces each node of a tree by the path between the root and that node. Downwards accumulations are tree homomorphisms because their pieces are.

A downwards accumulation is shown in Figure 5.

Downwards accumulations can be computed sequentially in linear time and in parallel in time proportional to the height of the tree. For suitably restricted component functions, they can also be implemented in parallel by an extension of tree contraction [4] in time logarithmic in the size of the tree.

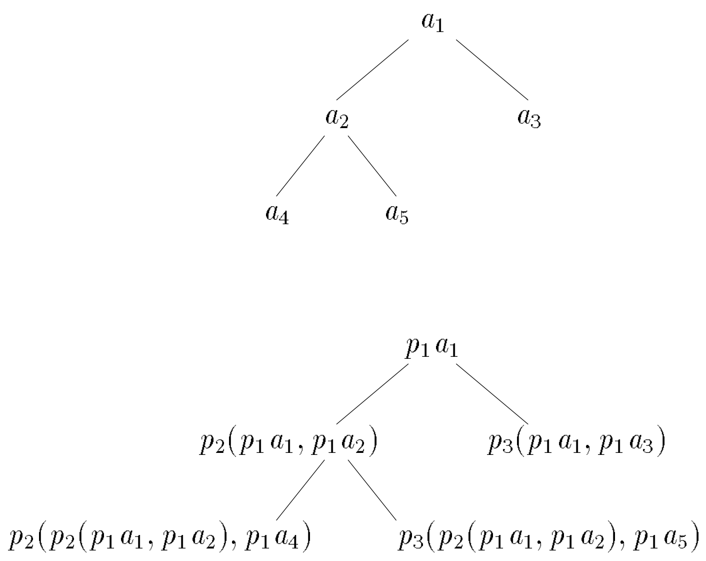

Figure 5: A Downwards Accumulation

3 The Difference Algorithm

The algorithm assumes the EREW PRAM model, although the upwards accumulation step can be carried out with the same complexity on more realistic models such as the hypercube and cube-connected-cycles. Let T_1 be the initial tree and T_2 the final tree. The algorithm consists of three steps:

1. **Creating hash tables of parents**. For each tree, a hash table is created recording, for each node, its parent in that tree. Each node in both of the trees examines its parents in the two tables and determines its status:

   - it has a parent in table 1 but not in table 2 – it has been deleted;
   - it has a parent in table 2 but not in table 1 – it has been inserted;
   - it has the same parent in both tables – it remains in the same local environment (but might have been moved as part of a subtree);
   - it has different parents in both tables, and so is the root of a subtree that has been moved.

   Each node of each tree is labelled to indicate whether it has been *inserted*, *deleted*, *moved*, or *unchanged*. At this point, almost all of the detail of how the trees differ is known. The remaining analysis minimises the expression of this difference and, if an edit sequence is required, arranges the differences in the most efficient order.

2. **Finding the largest subtrees involved in insertions, deletions, and moves.**

   This part of the algorithm merges local information about changes in the tree into larger units, for example, merging a subtree of inserted nodes into a single subtree insertion. This is done using an upwards accumulation to find those nodes that are the roots of subtrees all of which have the same change property. A subsequent downwards accumulation finds the first nodes along each path with the given change property – these are the roots of the largest changed subtree.

   The general strategy is an upwards accumulation using the following component functions:

$$\begin{aligned} p_1 &= \textit{if property then } T \textit{ else } F \\ p_2 &= (\textit{if property then } T \textit{ else } F) \wedge (\wedge/\ a_i) \end{aligned}$$

   In other words, each leaf with the property is labelled with T, and each internal node with the property is labelled with T if it has the property and so have all of its descendants. An example of this operation applied to a tree to find deletions is shown in Figure 6.

   Second, a downwards accumulation is applied to the resulting tree to find those paths whose labels are regular expressions of the form F^*T. These are the roots of the largest subtrees with the property. This is also shown in Figure 6.

   These two operations are applied first to Tree 1 to find the largest deleted subtrees. They are then applied to Tree 2 to find the largest inserted subtrees, and again to find the largest moved subtrees.

3. **Finding nodes that have moved but retained the same parent node.**

   The hash table data do not detect nodes that have moved but still retain the same parent, that is nodes that have moved at the same level. If we assume that the cost of moving some contiguous set of subtrees of a node to another place below that node is independent of the number of subtrees (for example, a block move in an editor), and that the unique labels for nodes can be ordered in sibling order, then the number of moves among the descendants of a node can be determined by examining the descendant nodes in T_2 and counting the number of locations where an identifier is smaller than the preceding one. This can be computed as a list reduction, and therefore takes (at each node) sequential time x for x the maximum tree branching factor, and parallel time $\log x$. Note that if there is one processor per node, these processors are available to carry out the reduction.

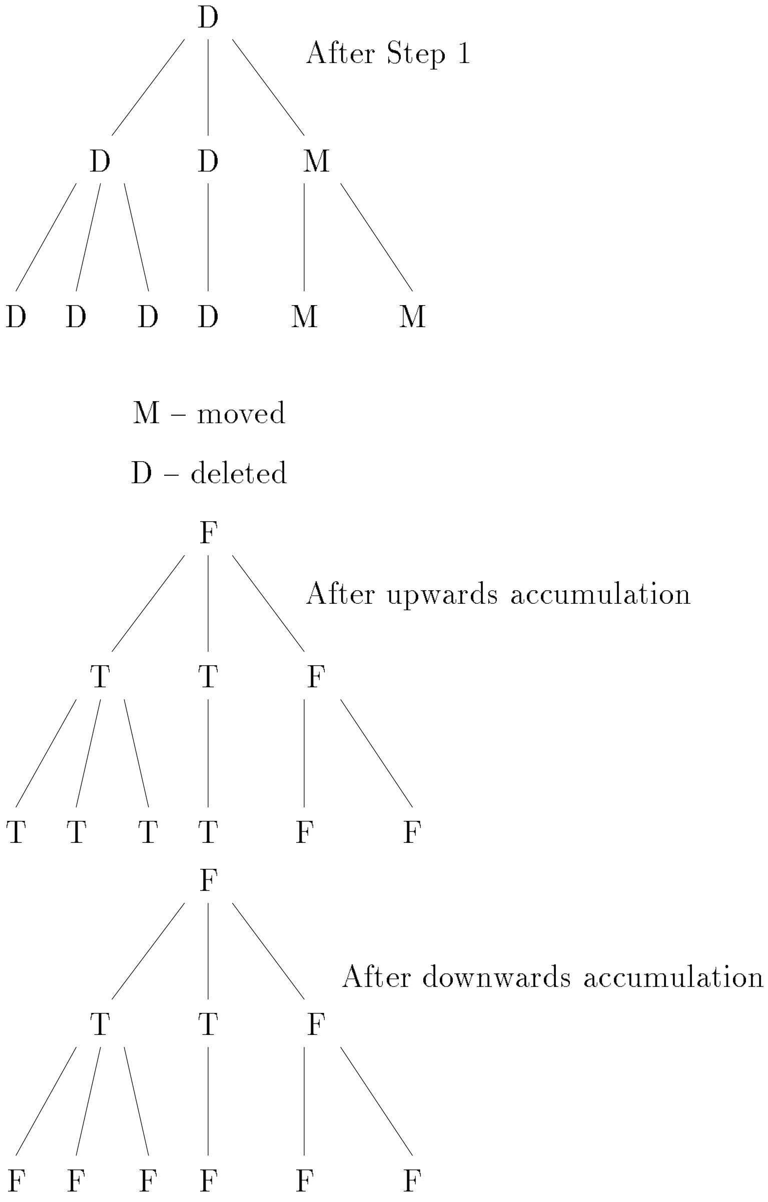

Figure 6: An Example of an Upwards Accumulation for Deletions

4 Complexity

The sequential complexity of the algorithm is

1. $O(n \log\log n)$ for the hash table creation and lookup;
2. $O(n)$ for the upwards and downwards accumulations;
3. $O(n)$ for same parent resolution, since each node of the tree is examined once;

The expected value, over all inputs, of the sequential execution time is $O(n \log\log n)$.

The parallel time complexity of the algorithm is

1. $O(\log\log n)$ for the hash table creation and lookup;
2. $O(\log n)$ for the upwards and downwards accumulations;
3. $O(\log x)$ for the same parent resolution, where x is the maximal branching factor;

The expected value, over all inputs, of the parallel execution time is $O(\log n)$.

5 Conclusions

We have described a tree difference algorithm that can be effectively parallelised. It uses two new results from the literature: universal hashing to quickly determine neighbourhoods of tree nodes, and upwards accumulations to effectively compute maximal subtrees that have been altered. Although this algorithm makes some strong assumptions, namely that nodes have unique labels, and allows only a limited set of edit operations, its performance is much better than previous tree difference algorithms. In practical settings such as analysis of structured text, its weaknesses are outweighed by its performance.

References

[1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. In *Proceedings of the Twenty-Fifth Allerton Conference on Communication, Control and Computing*, pages 624–633, September 1987.

[2] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[3] J. Gibbons. *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, University of Oxford, 1991.

[4] J. Gibbons, W. Cai, and D.B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23:1–14, 1994.

[5] G. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, 28(2):289–, 1981.

[6] International Organisation for Standardisation. *Information Processing Systems - Document Filing and Retrieval*, 1987. ISO/TC 97/SC 18/WG 4.

[7] G.L. Miller and J. Reif. Parallel tree contraction and its application. In *26th IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[8] D. Shasha and K. Zhang. Fast algorithms for the unit cost editing. *Journal of Algorithms*, 11:581–621, 1990.

[9] D.B. Skillicorn. *Foundations of Parallel Computing*. Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[10] R.C. Uzgalis. General hash functions. Technical Report TR-92-01, The University of Hong Kong, 1993.