Structured Parallel Computation in Structured Documents

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

March 1995

External Technical Report
ISSN-0836-0227-
95-379

Department of Computing and Information Science

Queen's University

Kingston, Ontario K7L 3N6

Document prepared March 6, 1995

Abstract

Document archives contain large amounts of data to which sophisticated queries are applied. The size of archives and the complexity of evaluating queries makes the use of parallelism attractive. The use of semantically-based markup such as

SGML makes it possible to represent documents and document archives as data types.

We present a theory of trees and tree homomorphisms, modelling structured text archives and operations on them, from which it can be seen that:

- many apparently-unrelated tree operations are homomorphisms;
- homomorphisms can be described in a simple parameterised way that gives standard sequential and parallel implementations for them;
- special classes of homomorphisms have parallel implementations of practical interest. In particular, we develop an implementation for path expression search, a novel powerful query facility for structured text, that takes time logarithmic in the text size.

Keywords: structured text, categorical data type, software development methodology, parallel algorithms, query evaluation.

1 Structured Parallel Computation

Computations on structured documents are a natural application domain for parallel processing. This is partly because of the sheer size of the data involved. Document archives contain terabytes of data, and analysis of this data often requires examining large parts of it. Also, advances in parallel computer design have made it possible to build systems in which each processor has sizeable storage associated with it, and which are therefore ideal hosts for document archives. Such machines will become common in the next decade.

The use of parallelism has been suggested for document applications for some time. Some of the drawbacks have been pointed out by Stone [24] and Salton and Buckley [17] – these centre around the need of most parallel applications to examine the entire text database where sequential algorithms examine only a small portion, and the consequent performance degradation in accessing secondary storage. While this point is important, it has been to some extent overtaken by developments in parallel computer architecture, particularly the storage of data in disk arrays, with some disk storage local to each processor. As we shall show, the use of parallelism allows such an increase in the power of query operations that it will be useful even if performance is not significantly increased.

Parallel computation is difficult in any application domain for the following reasons:

- There are many degrees of freedom in the design space of the software, because partitioning computations among processors strongly influences communication and synchronisation patterns, which in turn have a strong effect on performance. Hence finding good algorithms requires extensive searching in the absence of other information;
- Parallelism in an algorithm is only useful if it can be harnessed by some available parallel architectures, and harnessed in an efficient way;
- Expressing algorithms in a way that is abstract enough to survive the replacement of underlying parallel hardware every few years is difficult;
- It is hard to predict the performance of software on parallel machines without actually developing it and trying it out.

Making parallelism the workhorse of structured document processing requires finding solutions to these difficulties.

The extensive use of semantically-based markup, and particularly the use of SGML [13], means that most documents have a *de facto* tree structure. This makes it possible to model them by a data type with enough formality that useful theory can be applied. We will use the theory of categorical data types [19], a particular approach to initiality, emphasising its ability to hide those aspects of a computation that are most difficult in a parallel setting.

Categorical data types generalise abstract data types by encapsulating not only representation of the type, but also the implementation of homomorphisms on it. In object-oriented terms, the only methods available on constructed types are homomorphisms. As a parallel programming model this is ideal, since the partitioning of the data objects across processors, and the communication patterns required to evaluate homomorphisms can remain invisible to programmers. Programs can be written as compositions of homomorphisms without any necessary awareness that the implementations of these homomorphisms might contain substantial parallelism.

Recall that a homomorphism is a function that respects the structure of its arguments. If an argument object is a member of a data type with constructor $\bowtie$, then h is a homomorphism if there exists an operation $\circledast$ such that

$$h(a \bowtie b) = h(a) \circledast h(b)$$

This equation describes two different ways of computing the result of applying h to the argument $a \bowtie b$. The left hand side computes it by building the argument object completely and then just applying h to this object. The right hand side, however, applies h to the component objects from which the argument was built, and then applies the operation $\circledast$ to the results of these two applications.

There are two things to note about the computation strategy implied by the right hand side. The first is that it is recursive. If b is itself an object built up from smaller objects, say $b = c \bowtie d$, then

$$h(b) = h(c) \circledast h(d)$$

and so

$$h(a \bowtie b) = h(a) \circledast (h(c) \circledast h(d))$$

The structure of the computation follows the structure of the argument. Second, the evaluations of h on the right hand sides are independent and can therefore be evaluated in parallel if the architecture permits it. These simple ideas lead to a rich approach to parallel computation.

Homomorphisms include many of the interesting functions on constructed data types. In particular, all injective functions are homomorphisms. Furthermore, all functions can be expressed as almost-homomorphisms [2], the composition of a homomorphism with a projection, and this is often of practical importance.

In the next section we introduce the construction of a type for trees to represent structured text. We show that the construction reduces algorithm design for homomorphisms to the simpler problem of finding component functions, and illustrate a recursive parallel schema for computing homomorphisms on trees. In Section 3, we distinguish four special kinds of homomorphisms that capture common patterns for information flow in tree algorithms, and for which it is worth optimizing the standard schema implementation. These are: tree *maps*, tree *reductions*, *upwards* and *downwards accumulations*. In the subsequent five sections we illustrate tree homomorphisms of increasing sophistication, beginning with the computation of global document properties, then search problems (that is, query evaluation), and finally problems that involve communicating information throughout documents.

2 Parallel Operations on Trees

We build the type of homogeneous binary trees, that is trees in which internal nodes and leaves are all of the same type. Binary trees are too simple to represent the full structure of tagged texts, since any tagged entity may contain many subordinate entities, but it simplifies the exposition without affecting any of the complexity results.

In SGML an entity is delimited by start and end tags. The region between the start and end tags is either 'raw' text or is itself tagged. The structure is

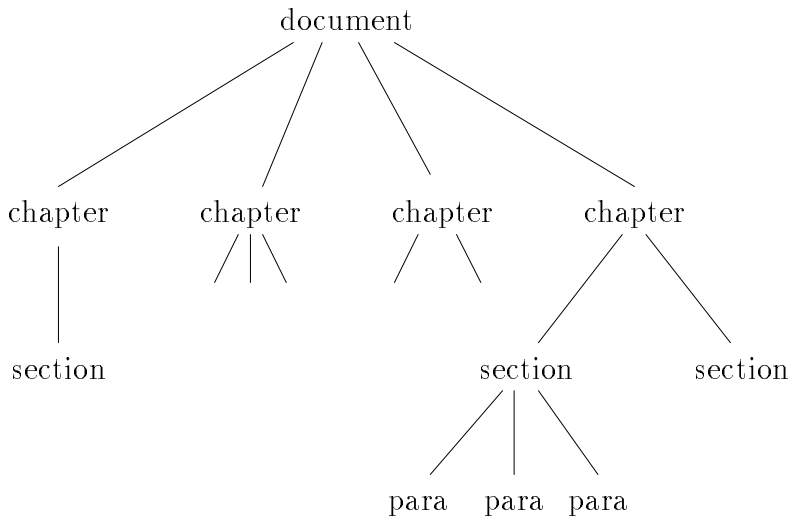


Figure 1: Modelling a Document as Tree

hierarchical, and can be naturally represented by a tree whose internal nodes represent tags, and whose leaves represent 'raw' data. All nodes may contain values for attributes. In particular, tags will often have associated text – for example, *chapter* tags contain the text of the chapter heading as an attribute, *figure* tags contain figure captions, and so on. Thus a typical document can be represented as a tree of the kind shown in Figure 1. Document archives can be modelled by trees as well, in which nodes near the root represent document classifications, as shown in Figure 2. We build trees over some base type A that can model entities and their attributes. For our purposes this is a tuple consisting of an entity name and a set of attributes. In our particular examples it will suffice to have one attribute, the string of text associated with each node of the document.

Trees have two constructors:

$$\begin{aligned} \mathit{Leaf} &: A \to \mathit{Tree}(A) \\ \mathit{Join} &: \mathit{Tree}(A) \times A \times \mathit{Tree}(A) \to \mathit{Tree}(A) \end{aligned}$$

The first constructor, *Leaf*, takes a value of type A and makes it into a tree consisting of a single leaf. The second constructor, *Join*, takes two trees and a value of type A and makes them into a new tree by joining the subtrees and

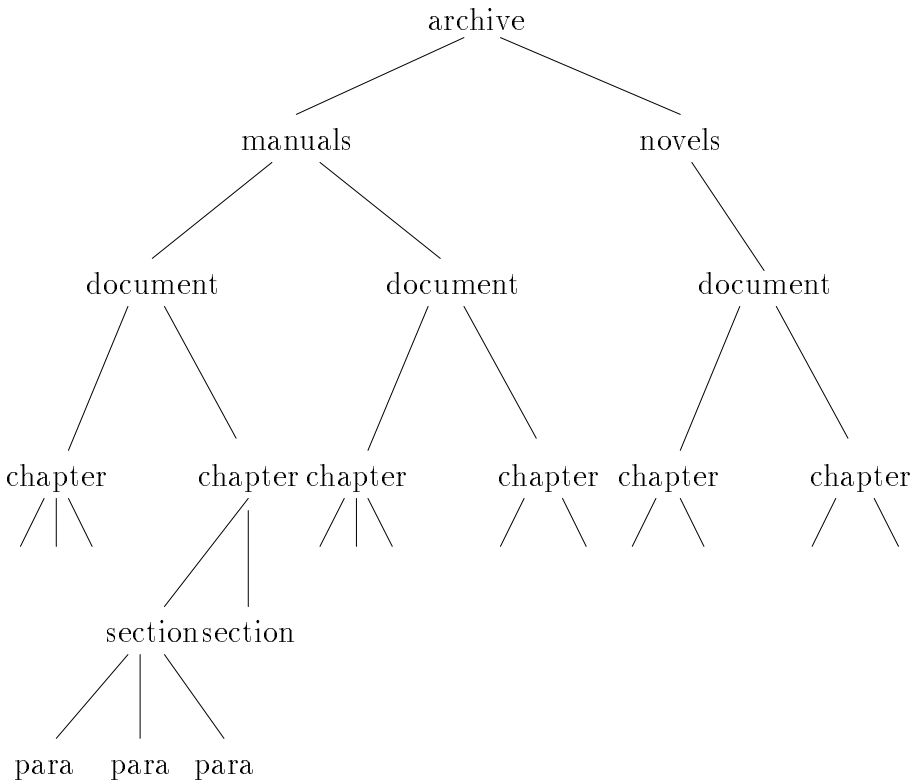
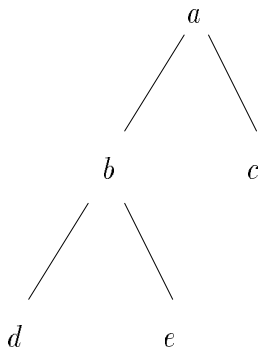


Figure 2: Modelling a Document Archive



$$Join(Join(Leaf(d), b, Leaf(e)), a, Leaf(c))$$

Figure 3: A Constructor Expression and the Tree it Describes

putting the value of type A at the internal node generated. Thus a tree is either a single leaf, or a tree obtained by joining two subtrees together with a new value at the join. A constructor expression describing a tree and the tree itself are shown in Figure 3.

Definition 1 A homomorphism, h, on trees is a function that respects tree structure, that is there must exist functions f_1 and f_2 such that

$$h(Leaf(a)) = Leaf(f_1(a))$$

and

$$h(Join(t_1, a, t_2)) = f_2(h(t_1), a, h(t_2))$$

Call f_1 and f_2 component functions of the homomorphism h.

Here f_2 is the glueing function that relates the effect of h on pieces of the argument to its effect on the whole argument. Notice that if h has type

$$h : Tree(A) \to P$$

then the types of f_1 and f_2 are

$$f_1 : A \to P$$

$$f_2 : P \times A \times P \rightarrow P$$

Thus finding a homomorphism amounts to finding such a pair of functions. However, finding h means being aware of the structure of $Tree(A)$, which is usually complex, whereas f_1 and f_2 are operations on (usually) simpler types. This is a considerable practical advantage.

Result 2 [19] Every homomorphism on trees is actually a function between an algebra (that is, a set with some operations defined on it) $(Tree(A), Leaf, Join)$ and a related algebra (P, f_1, f_2) for some type P and a pair of functions

$$\begin{aligned} f_1 &: A \rightarrow P \\ f_2 &: P \times A \times P \rightarrow P \end{aligned}$$

Note that the type signatures of f_1 and f_2 are obtained from those of $Leaf$ and $Join$ by replacing occurrences of $Tree(A)$ by P.

Furthermore, the homomorphism to each such algebra is unique, and there is therefore a one-to-one correspondence between algebras and homomorphisms. Each homomorphism is completely determined by the functions f_1 and f_2.

This correspondence justifies the notation

$$Hom(f_1, f_2)$$

for the (necessarily unique) tree homomorphism from $(Tree(A), Leaf, Join)$ to the algebra (P, f_1, f_2).

Using the fact that h is a homomorphism and that it corresponds to a pair of component functions, we can now make formal the recursive computation of h that we saw earlier. The recursive schema for evaluating all tree homomorphisms is shown in pseudocode in Figure 4. It is clear that the two recursive calls to `evaluate_tree_homomorphism` can be executed in parallel. Using this simple approach to parallelism, all tree homomorphisms on trees can be evaluated in parallel time proportional to the height of the tree (assuming that the evaluation of f_1 and f_2 take only constant time).

Here are some simple examples of tree homomorphisms:

```
evaluate_tree_homomorphism( f1, f2, t )
case t of
  Leaf(a) :
    return f1( a ) ;
  Join(Tree(t1), a, Tree(t2)) :
    return f2(
      evaluate_tree_homomorphism( f1, f2, t1 ),
      a,
      evaluate_tree_homomorphism( f1, f2, t2 ) )
end case
end
```

Figure 4: Tree Homomorphism Evaluation Schema

Example 3 *The tree homomorphism that computes the number of leaves in a tree replaces the value at each leaf by the constant 1, and then sums these values over the tree. It is given by*

$$\text{number-of-leaves} \;=\; \mathit{Hom}(K_1, f_2)$$

where $K_1 : A \to \mathbb{N}$ *is the constant 1 function and* $f_2 : \mathbb{N} \times A \times \mathbb{N} \to \mathbb{N}$ *is* $f_2(t_1, a, t_2) = t_1 + t_2$*. The recursive schema specialises to:*

```
evaluate_tree_homomorphism( t )
case t of
  Leaf(a) :
    return K_1 ( a ) ; { = 1 }
  Join(Tree(t1), a, Tree(t2)) :
    return
      evaluate_tree_homomorphism( t1 )
      +
      evaluate_tree_homomorphism( t2 )
end case
end
```

Example 4 The tree homomorphism to compute the number of internal nodes in a tree replaces each leaf by the value 0 and then adds 1 to the sum for each

internal node encountered. It is

$$\text{number-of-internal-nodes} \;=\; Hom(K_0, f_2)$$

where $K_0 : A \to \mathbb{N}$ is the constant 0 function, and $f_2 : \mathbb{N} \times A \times \mathbb{N} \to \mathbb{N}$ is $f_2(t_1, a, t_2) = t_1 + t_2 + 1$.

Example 5 *The tree homomorphism that finds the maximum value in a tree does nothing at the leaves and at internal nodes selects the maximum of the three values from its two subtrees and the node itself. It is*

$$\text{treemax} \;=\; Hom(id, f_2)$$

where $f_2(t_1, a, t_2) = \uparrow(t_1, a, t_2)$ and $\uparrow$ is ternary maximum.

The fact that all tree homomorphisms are the same in some sense is useful in two ways. First, finding tree homomorphisms reduces to finding component functions. Since component functions describe local actions at the nodes of trees, they are usually simpler conceptually than homomorphisms. There is a separation between common global tree structure and detailed node computations. Second, there is a common mechanism for computing any homomorphism, so it makes sense to spend time optimising it, rather than developing a variety of *ad hoc* techniques.

Nevertheless, there are some classes of homomorphisms for which better implementations are possible. These classes have many members that are of practical interest.

3 Four Classes of Tree Homomorphisms

There are four special classes of homomorphisms that are paradigms for common tree computations. They are: tree maps, tree reductions, and two forms of tree accumulations, upwards and downwards.

In our discussion of implementations, we will assume either the EREW PRAM or a distributed-memory hypercube architecture as the target com-

puter. The EREW PRAM does not charge for communication, and our implementations can all be arranged on the hypercube so that communication is always with nearest neighbours, except for the tree contraction algorithm used in several places. This enables us to include communication costs in our complexity measures. We use a result of Mayr and Werchner [16] to justify the complexity of tree contraction on the hypercube. We will also assume that a tree of n nodes is processed by an n-processor system, so that there is a processor per tree node. We return to this very unrealistic assumption later.

A *tree map* is a tree homomorphism that applies a function to all the nodes of a tree, but leaves its structure unchanged. If f is some function $f : A \to B$, then $TreeMap(f)$ is the function

$$TreeMap(f) \;=\; Hom(Leaf \cdot f, Join \cdot id \times f \times id) \;:\; Tree(A) \to Tree(B)$$

The recursive schema becomes

```
evaluate_tree_homomorphism( t )
case t of
  Leaf(a) :
    return Leaf( f( a )) ;
  Join(Tree(t1), a, Tree(t2)) :
    return Join(
      evaluate_tree_homomorphism( t1 ),
      f( a ),
      evaluate_tree_homomorphism( t2 ) )
end case
end
```

The recursion "unpacks" the argument tree into a set of leaves and internal nodes. These are then joined back together in exactly the same structure, except that the function f is applied to each value before it is placed back in the tree. The resulting tree has exactly the same shape as the argument tree; only the values (and types) at the leaves and internal nodes have changed.

Using the recursive schema as an implementation is therefore unnecessarily cumbersome. A tree map can be computed directly by skipping the disassembly and subsequent reassembly and just applying f to each leaf and internal

node. In a parallel implementation *tree map* can be computed without any communication.

Let $t_i(f)$ denote the time complexity of f in i processors. The sequential complexity of a tree map is given by

$$t_1(TreeMap(f)) = n \times t_1(f)$$

and the parallel complexity by

$$t_n(TreeMap(f)) = t_1(f)$$

A *tree reduction* is a tree homomorphism that replaces structure without explicitly manipulating values. Applied to a tree, a tree reduction most often produces a single value. Formally, a tree reduction is a tree homomorphism

$$TreeReduce(g) = Hom(id, g) \ : \ Tree(A) \rightarrow A$$

where the type of g is

$$g : A \times A \times A \rightarrow A$$

The recursive schema becomes

```
evaluate_tree_homomorphism( t )
case t of
  Leaf(a) :
    return  a ;
  Join(Tree(t1), a, Tree(t2)) :
    return g(
      evaluate_tree_homomorphism( t1 ),
      a,
      evaluate_tree_homomorphism( t2 ) )
end case
end
```

Figure 5 shows a tree and the result of applying a reduction to it.

A tree reduction can be computed in parallel in time proportional to the height of the tree, assuming that each application of the function g takes

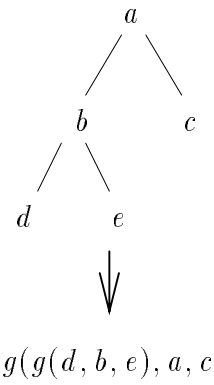


Figure 5: A Tree Contraction

constant time. Knowing the result of the reduction on subtrees, the result of the reduction on the tree formed by joining them requires a further application of g, so that critical path is the application of gs along the longest branch.

The sequential complexity of tree reduction is

$$t_1(\mathit{TreeReduce}(g)) = n \times t_1(g)$$

and the parallel complexity is

$$t_n(\mathit{TreeReduce}(g)) = ht \times t_1(g)$$

where ht is the height of the tree.

Somewhat surprisingly, tree reduction can be computed much faster for many kinds of functions g. For theoretical models such as the EREW PRAM and more practical architectures such as the hypercube, tree reduction can be computed in time logarithmic in the number of nodes of the tree, subject to some mild conditions on the function g [1, 16], described in Appendix A. This is a big improvement over the method suggested above, since a completely left or right branching tree with n nodes requires time proportional to n to reduce directly, whereas the faster algorithm takes time $\log n$.

The key to fast tree reduction is making some useful progress towards the eventual result at many nodes of the tree, whether or not the reductions for

the subtrees of which they are the root have been completed. Naive reduction applies g only at those nodes both of whose children are leaves. For an unbalanced tree, this creates a chain of reductions whose length is proportional to the height of the tree. However, for well-behaved reduction operations it is also possible to carry out partial reductions for nodes only one of whose descendants is a leaf. This brings the time complexity of the tree reduction down to logarithmic in the size of the tree, no matter how unbalanced it is. The details are given in Appendix A.

Thus we have a parallel time complexity for tree reduction of

$$t_n(\mathit{TreeReduce}(g)[\mathit{tree\ contraction}]) = \log n$$

since $t_1(g)$ must be $O(1)$.

The third useful family of tree homomorphisms are the *upwards accumulations*. Upwards accumulations are operations in which data can be regarded as flowing upwards in the tree, and where the computations that take place at each node depend on the results of computations at lower nodes. These are like tree reductions that leave all their partial results behind. The final result is a tree of the same shape as the argument tree, in which each node is the result of a tree reduction rooted at that node. This is illustrated in Figure 6.

Upwards accumulations is characterized as follows. Let *subtrees* be the function that replaces each node of a tree by the subtree rooted at that node. Hence it takes a tree and produces a tree of trees. Although *subtrees* is a messy and expensive function, it is still a homomorphism.

Upwards accumulations are those functions that can be expressed as the composition of *subtrees* with a tree homomorphism mapped over the nodes of the intermediate tree.

$$\text{upwards accumulation} \ = \ \mathit{TreeMap}(\mathit{Hom}(f_1, f_2)) \cdot \text{ subtrees}$$

If $\mathit{Hom}(f_1, f_2) : \mathit{Tree}(A) \to X$ then the upwards accumulation has type signature $\mathit{Tree}(A) \to \mathit{Tree}(X)$.

Computing an upwards accumulation in the way implied by this definition is expensive. The point of defining upwards accumulations like this is that the result at any node shares large common expressions with the results at its

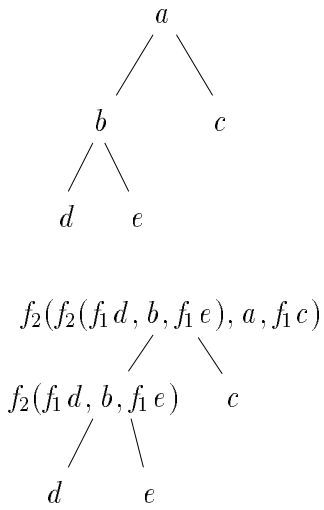


Figure 6: An Upwards Accumulation

immediate descendants. Therefore, the number of partial results that must be computed in an upwards accumulation is linear in the number of tree nodes. Thus it is possible that there is a fast parallel algorithm, provided the dependencies introduced by the communication involved are not too constraining; and indeed there is. The algorithm is an extension of tree contraction; when a node u is removed, it is stacked by its remaining child. When this child receives its final value, it unstacks u and computes its final value. Details may be found in [7].

The sequential time complexity of an upwards accumulation is

$$t_1(UpAccum(f_1, f_2)) = n(t_1(f_1) + t_1(f_2))$$

its parallel time complexity is

$$t_n(UpAccum(f_1, f_2)) = t_1(f_1) + ht \times t_1(f_2))$$

and its parallel time complexity using extended tree contraction is

$$t_n(UpAccum(f_1, f_2)[tree\ contraction]) = \log n$$

because f_1 and f_2 must be $O(1)$.

The fourth useful family of tree homomorphisms are the *downwards accumulations*. Downwards accumulations replace each node of a tree by some function of the nodes on the path between it and the root. This models functions in which the flow of information is broadcast down through the tree.

We first define the type of *non-empty paths*, which are like lists except that they have two concatenation constructors, *left turn*, $\smile$, and *right turn*, $\frown$. This type models the paths that occur between the root of a tree and any other node, where it is important to remember whether the path turns towards a left or right descendant at each step. The two constructors are mutually associative, that is

$$(a?b)??c = a?(b??c)$$

where ? and ?? are either of the constructors. Homomorphisms on *paths* are the unique arrows to algebraic structures

$$(P, f : A \to P, \oplus, \otimes : P \times P \to P)$$

where $\oplus$ and $\otimes$ satisfy the same mutual associativity property. Define *paths* to be the function that replaces each node of a tree by the path between the root and that node. A downwards accumulation is a tree homomorphism that can be expressed as the composition of *paths* with a path homomorphism mapped over the nodes of the intermediate tree.

$$\text{downwards accumulation} \;=\; \mathit{TreeMap}(\mathit{PathHom}(\oplus, \otimes)) \cdot \;\text{paths}$$

If $\mathit{PathHom}(\oplus, \otimes) : \mathit{Path}(A) \to X$ then the downwards accumulation has type signature $\mathit{Tree}(A) \to \mathit{Tree}(X)$.

A downwards accumulation is shown in Figure 7. The value that results at each node of the tree depends on the values of the nodes that appear in the path between the node and the root, together with their structural arrangement, that is, the combination of left and right turns that appear along the path.

Clearly it is expensive to compute a downwards accumulation by computing all of the paths and then mapping a path reduction over them. However, there are again large common expressions between each node and its parent, so that

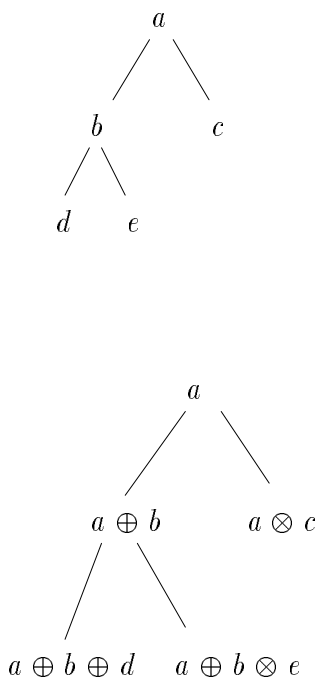


Figure 7: A Downwards Accumulation

the total number of expressions to be computed is linear in the size of the tree. As a result, there is an algorithm that runs in parallel time proportional to the logarithm of the number of the nodes in the tree [7].

The sequential time complexity of a downwards accumulation is

$$t_1(UpAccum(\oplus, \otimes) = n(t_1(\oplus) + t_1(\otimes))$$

its parallel time complexity is

$$t_n(UpAccum(\oplus, \otimes) = t_1(\oplus) + ht \times t_1(\otimes))$$

and its parallel time complexity using extended tree contraction is

$$t_n(UpAccum(\oplus, \otimes)[tree\ contraction]) = \log n$$

because $\oplus$ and $\otimes$ must be $O(1)$.

These complexity results are based on the assumption of one processor per document node. It is possible to implement all of these homomorphisms using fewer processors by allocating a region of each tree to a processor which then applies each homomorphism to that region. The actual partitioning is complex, because a partition of a tree into regions is not normally itself a tree. However, with some care it is possible to get implementations of the fast parallel operations above in time complexity

$$n/p + \log p$$

for trees of size n using p processors [20]. For small p, this gives almost linear speed-up over sequential implementations, while for large p it gives almost logarithmic execution times.

Binary trees are easily extended to trees in which each internal node has a list of subtrees (so-called Rose trees [6]). Rose trees much more naturally model SGML tagged text. The complexity of the algorithms we will present only changes by a constant factor, since any Rose tree can be replaced by a binary tree without changing the order of magnitude of the the number of nodes.

4 Global Properties of Documents

We now have a set of homomorphic tree operations that can be evaluated in parallel effectively. We now begin to show how these operations can be applied to structured text.

We begin with operations that are tree maps or tree reductions. A broad class of such operations are those that count the number of occurrences of some text or entity in a document. In such tree homomorphisms, the pair of functions used are of the general form

$$\begin{aligned} f_1(a) &= \text{if } (a = \text{entity name}) \text{ then 1 else 0} \\ f_2(t_1, a, t_2) &= t_1 + t_2 + (\text{if } (a = \text{entity name}) \text{ then 1 else 0}) \end{aligned}$$

with types

$$\begin{aligned} f_1 &: A \to \mathbb{N} \\ f_2 &: \mathbb{N} \times A \times \mathbb{N} \to \mathbb{N} \end{aligned}$$

Some tree operations can be factored into the composition of a tree map and a tree reduction. This is often an easy way to understand and compute the complexity of a homomorphism. Notice that f_2 above can be written as

$$f_2 = \oplus \cdot id \times f_1 \times id$$

where $\oplus$ is ternary addition. We can express the 'count entities' homomorphism above as a composition like this:

$$\text{count entities} = \mathit{TreeReduce}(\oplus) \cdot \mathit{TreeMap}(f_1)$$

The tree map can be computed in a single parallel step, taking time $t_1(f_1)$. The tree reduction step can be computed using tree contraction in a logarithmic number of steps, each involving an application of functions related to f_2, taking time $\log n$. Thus the total parallel time complexity of evaluating this tree homomorphism is

$$t_n(\text{total time}) = \log n + t_1(f_1) \tag{1}$$

The following document properties of frequency can be determined in the parallel time given by Equation 1:

- number of occurrences of a word (that is, a delimited terminal string),
- number of occurrences of a structure or entity (section, subsection, paragraph, figure),
- number of reference points (labels).

An extension of these counting tree homomorphisms produce simple data types as results. For example, to produce a list of the different entity names used in a document, we use a tree homomorphism with component functions

$$\begin{aligned} f_1(a) &= \{\text{entity name } a\} \\ f_2(t_1, a, t_2) &= \cup(t_1, t_2, \{\text{entity name } a\}) \end{aligned}$$

that produces a set containing the entity names that are present. Each leaf is replaced by the name of the entity it represents. Internal nodes merge the sets of entity names of their descendants with the name of the entity they represent. Using sets means that we record each entity name only once in the final set. To determine the number of different entities present in a document, we need only to compute the size of the set produced by this tree homomorphism.

Changing the set used to a list, we define a tree homomorphism to produce a table of contents. It is

$$\begin{aligned} f_1(a) &= [a.string] \\ f_2(t_1, a, t_2) &= a.string +\!\!+ t_1 +\!\!+ t_2 \end{aligned}$$

where $+\!\!+$ is list concatenation, and $a.string$ extracts the string attribute associated with node a. This homomorphisms produces a list of all of the tag instances in the document in level order. The parallel time complexity of these tree homomorphisms is not straightforward to compute for two reasons: the operation of concatenation is not necessarily constant time, so the computation of f_2 will not be; and the size of lists grows with the distance from the leaves, creating a communication cost that must be accounted for on real computers [21].

Another useful class of tree homomorphisms computing global properties are those that compute properties of extent. The most obvious example computes the length of a document in characters. It is

$$\begin{aligned} f_1(a) &= \text{length}(a.string) \\ f_2(t_1, a, t_2) &= t_1 + t_2 + \text{length}(a.string) \end{aligned}$$

Similarly, the tree homomorphism that computes the deepest nesting depth of structures in a document is

$$\begin{aligned} f_1(a) &= 0 \\ f_2(t_1, a, t_2) &= \uparrow(t_1, t_2) + 1 \end{aligned}$$

Again their parallel time complexity is given by Equation 1.

Software is an example of structured text. Some tree homomorphisms that apply to software are: computing the number of statements, and building a simple (that is, unscoped) symbol table.

5 Search Problems

Another important class of operations on structured text involve searching them for data matching some key. Four levels of search can be distinguished:

1. **Search on index terms**. This requires preprocessing of the document to allocate search terms (with the attendant problems of choice varying between individuals). It is usually implemented using indexing. Parallelism can be used by partitioning the index.

2. **Search on full text**. This is implemented using a signature file technique [4, 22, 23] or special purpose hardware [10, 11]. Parallelism can be used by partitioning the signature file.

3. **Search on non-hierarchical tagged regions**. This is a more expressive variant of full text search in which non-hierarchical tags are present in the text. Searches may include references to tags as well as to content. This approach is used in the PAT system [8] for searching the Oxford

English Dictionary. The descriptive markup of historical documents is typically too *ad hoc* to be captured by SGML-style tags, but is nevertheless an important part of the organisation of the document. The PAT index contains the strings beginning at each new word or tag position in the document, stored in a Patricia trie.

Fulcrum use a similar idea with zone tags, associating a set of zones with each range of the text. Zones are added to the document index, allowing searches to be based on both content and zone. This allows content references to be modified by limited context information (e.g. "dog" within a section heading) [12].

4. **Search on full text and structure**. This allows search to include information about both contents and tags, and uses regular expressions, so that content references can be modified by context information (e.g. "dog" within the third section heading), and truncated term expansion is trivially available. No existing system has this capability, but the *path expressions* query language [14] allows such queries to be expressed, and we show in subsequent sections how such searches may be implemented efficiently.

Fortunately, this fourth level of search is no more expensive to implement in parallel than the previous three. Thus even if parallelism does not provide absolute performance improvements because of limits on disk speeds, it can provide improvements in functionality. We explore this style of search further in the next two sections.

6 Parallel Search of Flat Text

There is a well-known parallel algorithm for recognizing whether a given string is a member of a regular language in time logarithmic in the length of the string [5, 9]. This algorithm is naturally parallel and readily adapted to document search, even on quite modest parallel computers. Although it was described for the Connection Machine [9] it never seems to have been used on any real parallel system. It is related to the technique used by Hollaar [10, 11], but is more expressive, since the use of special-purpose hardware limits the flexibility of Hollaar's searches. We describe the parallel algorithm and show how it

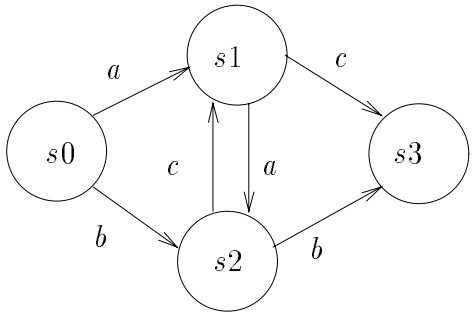


Figure 8: A Simple Finite State Automaton

allows queries that are regular expressions, and hence can search for patterns involving both content and tags.

Suppose that we want to determine if a given string is a member of a regular language over some alphabet $\{a, b, c\}$. The regular language can be defined by a finite state automaton, whose transitions are labelled by elements of the alphabet. This automaton is preprocessed to give a set of tables, each one labelled with a symbol of the alphabet, and consisting of pairs of states such that the labelling symbol labels a transition from the first pair in each state to the second. For example, given the automaton in Figure 8, the tables are:

a	b	c
$(s0, s1)$	$(s0, s2)$	$(s2, s1)$
$(s1, s2)$	$(s2, s3)$	$(s1, s3)$

The tree homomorphism (on lists) consists of two functions:

$$\begin{aligned} f_1(a) &= \mathit{Table}(a) = \{(s_i, s_j) \mid \exists \,\text{transition} s_i \xrightarrow{a} s_j\} \\ f_2(t_1, t_2) &= \{(s_i, s_k) \mid (s_i, s_j) \in t_1, (s_j, s_k) \in t_2\} \end{aligned}$$

It will be convenient to write f_2 as an infix binary operation, $\circledast$, on tables. The function f_1 replaces each symbol in the input string with the table defining how that symbol maps states to states. The function f_2 then composes tables to reflect the state-to-state mapping of longer and longer strings. Consider $\circledast$

applied to the tables for a and b.

$$\begin{array}{cccc} a & & b & & ab \\ \hline (s0, s1) & \circledast & (s0, s2) & = & (s1, s3) \\ (s1, s2) & & (s2, s3) & & \end{array}$$

At the end of the reduction, the single resulting table expresses the effect of the entire input string on states. If it contains a pair whose first element is the initial state and whose second element is a final state, the string is in the regular language.

All of the tables are of finite size (no larger than the number of transitions in the automaton). Each table composition takes no longer than linear in the number of states of the automaton. The reduction itself takes time logarithmic in the size of the string being searched on a variety of parallel architectures [18].

The regular language recognition problem is easily adapted for query processing. Suppose we wish to determine if some regular expression, RE, is present in an input string. This regular expression defines a language, $L(RE)$, that is then extended to allow for the existence of other symbols and for the string described by the regular expression to appear in a longer string. Call this extended language $L(RE)'$. Then the search problem becomes: is the text s in the language $L(RE)'$? There is a finite state automaton corresponding to $L(RE)'$. It is based on the finite state automaton for $L(RE)$ with the addition of extra transitions labelled with symbols from the extended alphabet. It can be as large as exponential in the size of the regular expression, but is independent of the size of the string being searched. An example of the automaton corresponding to the search for the word "cat" is shown in Figure 9 and the resulting algorithm on an input string "bcdabcat" in Figure 10.

Queries that are boolean expressions of simpler regular language queries could be evaluated by evaluating the simple queries independently and then carrying out the required boolean operations on the results. However, the closure of the class of regular languages under union, intersection, complementation, and concatenation means that such complex queries can be evaluated in a single pass through the data by constructing the appropriate deterministic finite state automaton. For example, a query of the form "are x and y in the

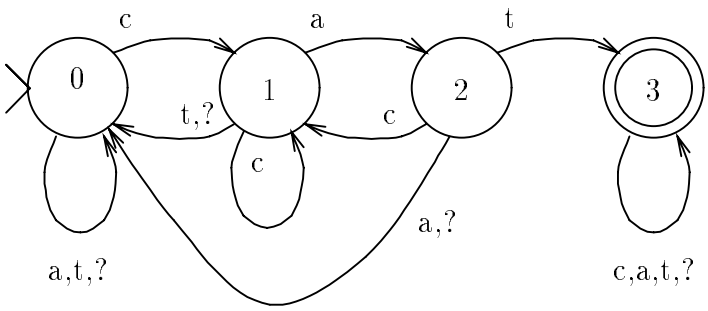


Figure 9: Finite State Automaton for "cat"

b	*c*	*d*	*a*	*b*	*c*	*a*	*t*
$0,0$	$0,1$	$0,0$	$0,0$	$0,0$	$0,1$	$0,0$	$0,0$
$1,0$	$1,1$	$1,0$	$1,2$	$1,0$	$1,1$	$1,2$	$1,0$
$2,0$	$2,1$	$2,0$	$2,0$	$2,0$	$2,1$	$2,0$	$2,3$
$3,3$	$3,3$	$3,3$	$3,3$	$3,3$	$3,3$	$3,3$	$3,3$

	$0,1$		$0,0$		$0,1$		$0,0$
	$1,1$		$1,0$		$1,1$		$1,3$
	$2,1$		$2,0$		$2,1$		$2,0$
	$3,3$		$3,3$		$3,3$		$3,3$

		$0,0$				$0,3$	
		$1,0$				$1,3$	
		$2,0$				$2,3$	
		$3,3$				$3,3$	

				$0,3$			
				$1,3$			
				$2,3$			
				$3,3$			

Figure 10: Algorithm Progress for the Search

text" amounts to asking if the text is a string of the augmented language of the intersection of the regular languages of the strings x and y.

Regular language recognition for strings is easily generalized to trees. The first step applies a tree map that replaces each node of the tree by a table, mapping states to states, that is the tree homomorphism:

$$Hom(f_1(a) = Leaf \cdot \ Table(a),\ f_2 = Join \cdot id \times f_1 \times id)$$

The second step is a tree reduction, in which the tables of a node and its two subtrees are composed using a ternary generalisation of the composition operator used in the string algorithm above:

$$Hom(id, f_2(t_1, t_2, t_3) = \circledast(t_2, t_1, t_3))$$

where table composition has been generalized to a ternary operation

$$\circledast(t_2, t_1, t_3) = \{(s_i, s_l) \mid (s_i, s_j) \in t_2, (s_j, s_k) \in t_1, (s_k, s_l) \in t_3\}$$

Notice that the composition of tables composes the table belonging to the internal node first, followed by the tables corresponding to the subtrees in order. This represents the case where the text in the internal node represents some kind of heading. The order may or may not be significant in an application, but care needs to be taken to get it right so that strings that cross entity boundaries are properly detected. The extension of the search in a linear string to a tree is shown in Figure 11.

This algorithm takes parallel time logarithmic in the number of nodes of the tree, using the tree contraction algorithm discussed in the previous section. The automaton can be extended as before to solve query problems, giving a fast parallel query evaluator for a useful class of queries.

Query evaluation problems that are of this kind include: word and phrase search, and boolean expressions involving phrase search. Such queries are of about the complexity of those permitted by the PAT system.

Because the tree structure of structured text encodes useful information, we now turn to extending this search algorithm to a new kind of search in which not only the presence of data but also its relationships can be expressed in the query. This increases the power of the query language substantially. It

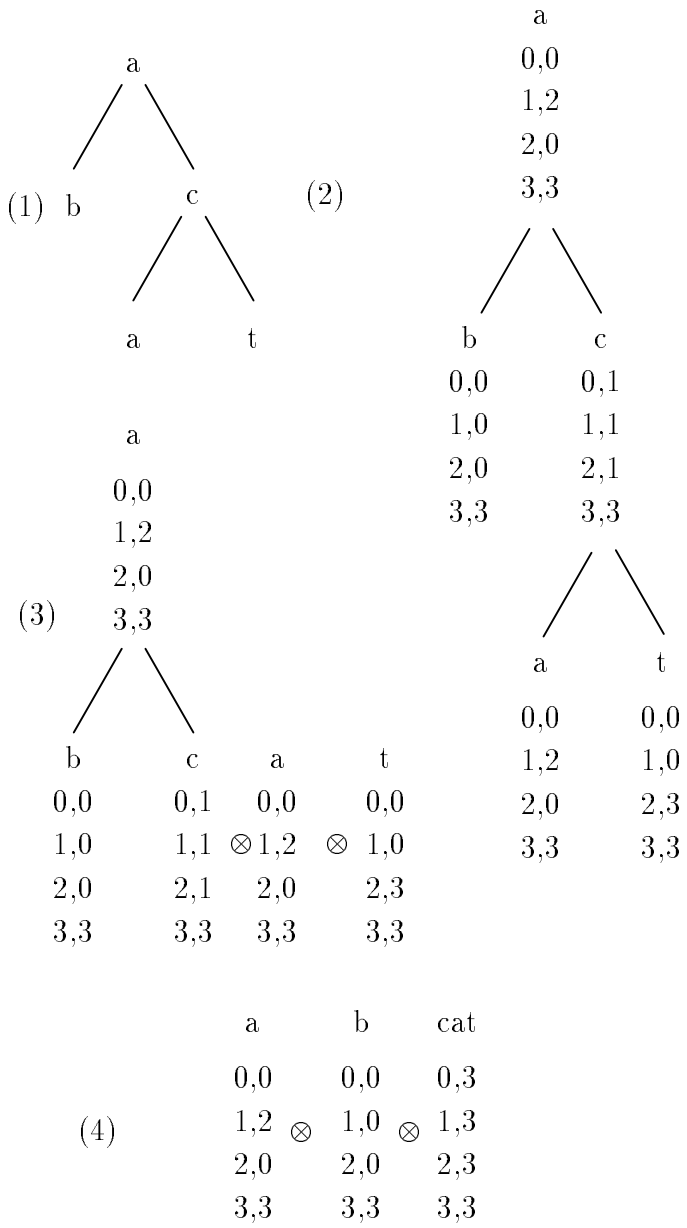


Figure 11: Algorithm Progress for Tree Search

turns out that such structured queries can still be computed in parallel within logarithmic time bounds, making structured queries of practical importance.

7 Accumulations and Information Transfer

Accumulations allow nodes to find out information about all their neighbours in a particular direction. Upwards accumulation allow each node of a tree to accumulate information about the nodes below it. Downwards accumulations allow each node to accumulate information about those nodes that lie between it and the root. Powerful combination operations are formed when an upwards accumulation is followed by a downwards accumulation, because this makes arbitrary information flow between nodes of the tree possible. As we have seen, both kinds of accumulations can be computed fast in parallel if their component functions are well-behaved.

Some examples of upwards accumulations are: computing the length in characters of each object in the document; and computing the offset from each segment to the next similar segment (for example the offset from each section heading to the next section heading).

Some examples of downwards accumulations are: structured search problems, that is searching for a part of a document based on its content and its structural properties; and finding all references to a single label. Structured search is so important that we investigate its implementation further in the next section.

Gibbons gives a number of detailed examples of the usefulness of upwards followed by downwards accumulations in [6]. Some examples that are important for structured text are: evaluating attributes in attribute grammars (which can represent almost any property of a document); generating a symbol table for a program written in a language with scopes; rendering trees, which is important for navigating document archives; determining which page each object would fall on if the document were produced on some device; determining which node the ith word in a document is in, and determining the font of each object (in systems where font is relative not absolute, such as LaTeX).

Operations such as resolving all cross references and determining all the

references to a particular point can also be cast as upwards followed by downwards accumulations, but the volume of data that might be moved on some steps makes this expensive.

8 Parallel Search of Structured Text

Queries on structured text involve finding nodes in the tree based on information about the content of the node (its text and other attributes) and on its context in the tree, particularly its relative context such as "the third section". Here are some examples, based on [14]:

Example 6 **document where** *('database'* **in document***)*

This returns those documents that contain the word 'database'.

Example 7 **document where** *('Smith'* **in** *Author)*

This returns those documents where the word 'Smith' occurs as part of the Author structure within each document. This query depends partly on structural information (that an Author substructure exists) as well as text. Notice that the object returned depends on a condition on the structure below it in the tree.

Example 8 *Section* **of document where** *('database'* **in document***)*

This returns all sections of documents that contain the word 'database'. The object returned depends on a condition of the structure above it in the tree. Notice that there is a natural way to regard this as a two-step operation: first select the documents, then select the sections.

Example 9 *Section* **of document where** *('database'* **in** *Section-Heading)*

This returns those sections whose headings contain the word 'database'. Notice that the object returned depends on a condition of a structure that is neither above or below it in the tree, but is nevertheless related to it.

All of these queries describe patterns in the tree; patterns may include "don't care"s in both the nodes and the branch structure. Queries are relative to a particular node in the tree (usually the root) and return a bag of nodes corresponding to the roots of trees containing the pattern (bags are sets in which repetitions count; we want to know all the places where a pattern is present, so the result must allow for more than one solution). Allowing queries to take a bag of nodes as their inputs, so that searches begin from all of these nodes, allows queries to be composed. Note that a node is precisely identified by the path between itself and the root, so we might as well think of node identifiers as paths.

There are two different kinds of bag operations in the query examples above. The first are *filters* that take a bag of nodes and return those elements of the bag that satisfy some condition. The second are *moves* that take a bag of nodes and return a bag in which each node has been replaced either by a node related to it (its ancestor, its descendant, its sibling to the right) or by one of its attributes. A simple query is usually a filter followed by the value of an attribute at all the nodes that have passed the filter. More complex queries such as the last example above require more complex moves.

This insight is the critical one in the design of *path expressions* [15], a general query language for structured text applications. The crucial property of path expressions that we require is that filters can be broken up into searches for patterns that are single paths, and are therefore expressible as regular expressions over paths.

Such filters can be computed by replacing each node of the structured text tree by the path from it to the root, and then applying the regular string recognition algorithm, extended to include left and right turns, to each of these paths. Those nodes for which the string recognition algorithm returns True are the nodes selected by the filter.

For example, a query about the presence of a chapter whose first section contains the word "About" is equivalent to asking if there is a path in the tree that contains the following:

$$\text{(entity = chapter) left turn (entity = section)} \wedge \text{ (“About” } \in \text{ section.string)}$$

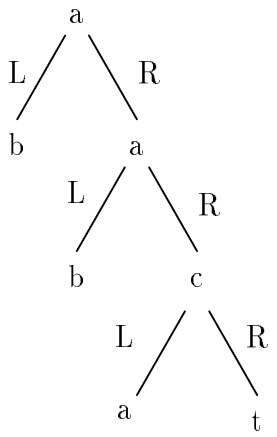


Figure 12: A Tree Annotated with Turn Information

A tree annotated with left and right turns is shown in Figure 12, and the result of applying the *paths* function to it is shown in Figure 13.

The regular language recognition algorithm for strings can be extended to paths straightforwardly. The query string may contain references to right and left turns, and so may the extended regular language. Otherwise the operations of table construction and binary table composition ($\circledast$) behave just as before.

Filters can be expressed as

$$\text{filter} \;=\; \mathit{TreeMap}(\mathit{PathHom}(\mathit{Table}, \circledast)) \cdot \mathit{paths}$$

The right hand side replaces the text tree with the tree in which each node contains the path between itself and the root. The *TreeMap* then maps the regular language recognition algorithm for paths over each of these nodes. Those that find an instance of the string are the roots of subtrees that correspond to the query pattern.

But we have already seen that operations that can be expressed as maps over paths are downward accumulations, and hence can be computed efficiently in parallel when the component functions are well-behaved. We have also seen that table composition is a constant-time, associative function and so satisfies

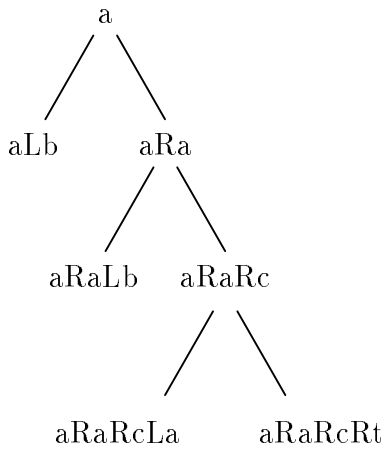


Figure 13: The Result of Applying the paths Function

the requirements for tree contraction. Thus there is a logarithmic time parallel algorithm for computing such filters

$$t_n(\mathit{filter}) \;=\; \log n$$

Path expression queries may therefore be included in structured text systems without additional cost, dramatically improving the sophistication of the way in which such resources can be queried.

9 Conclusions

The SGML approach of structural tagging of entities makes it possible to model structured text as a data type. This in turn makes it possible to use the machinery of categorical data types to examine trees and tree homomorphisms. This reveals the underlying similarities between apparently different operations on structured text, suggests sequential and parallel implementations for them, and shows how their construction can be reduced to the construction of component functions.

In particular we have shown how tree contraction and its extensions can be used to build fast parallel implementations of standard search techniques.

A major contribution of the paper is the discovery of a fast parallel implementation for searches based on path expressions. These allow much more sophisticated searching than existing query languages with the same performance.

A The Tree Reduction Algorithm

The tree reduction algorithm is well-known in the parallel algorithms literature. We follow the presentation in [7].

We begin by defining a contraction operation that applies to any node and its two descendants, if at least one descendant is a leaf. Suppose that each internal node u contains a pointer $u.p$ to its parent, a boolean flag $u.left$ that is true if it is a left descendant, a boolean flag $u.internal$ that is true if the node is an internal node, a variable $u.g$ describing an auxiliary function of type $A \rightarrow A$, and, for internal nodes, two pointers, $u.l$ and $u.r$ pointing to their left and right descendants respectively.

We describe an operation that replaces u, $u.l$, and $u.r$, where $u.l$ is a leaf, by a single node $u.r$ as shown in Figure 14. A symmetric operation contracts in the other direction when $u.r$ is a leaf. The operations required are

$$
\begin{array}{rcl}
\text{u.r.g} & \leftarrow & \lambda\text{x. u.g}(f_2(\text{u.l.g(u.l.a)}, \text{u.a}, \text{u.r.g(x)})) \\
\text{u.r.p} & \leftarrow & \text{u.p} \\
\text{if u.left then} & \text{u.p.l} & \leftarrow \text{u.r else u.p.r} \leftarrow \text{u.r} \\
\text{u.r.left} & \leftarrow & \text{u.left} \\
\text{if u = root} & \text{then} & \text{root} \leftarrow \text{u.r}
\end{array}
$$

The first step is the most important one. It 'folds' in an application of f_2 so that the function computed at node u.r after this contraction operation is the one that would have been computed at u before the operation. The contraction algorithm will only be efficient if this step is done quickly and the resulting expression does not grow long. We will return to this point below.

The contraction operations must be applied to about half of the leaves on each step if the entire contraction is to be completed in about a logarithmic

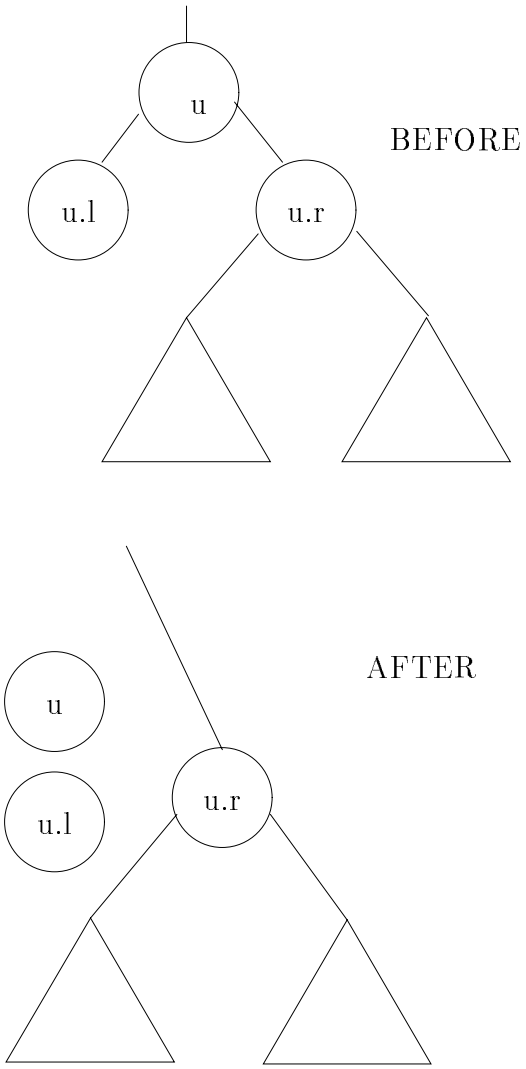


Figure 14: A Single Tree Contraction Step: Nodes u and u.l are deleted from the tree, and u.r is updated

number of steps. The algorithm for deciding where to apply the contraction operations is the following:

1. Number the leaves left to right beginning at 0 – this can be done in $O(\log n)$ time using $O(n/\log n)$ processors [3].
2. For every u such that $u.l$ is an even numbered leaf, perform the contraction operation.
3. For every u that was not involved in the previous step, and for which $u.r$ is an even numbered leaf, perform the contraction operation.
4. Renumber the leaves by dividing their positions by two and taking the integer part.

Sufficient conditions for preventing the lambda expressions in the $u.g$s from growing large are the following [1]

1. For all nodes u, the sectioned function $f_2(\_, a, \_)$ of type $A \times A \rightarrow A$ is drawn from an indexed set of functions F, and the function $u.g$ is drawn from an indexed set of functions G. Both F and G contain the identity function.
2. All functions in F and G can be applied in constant time.
3. For all f_i in F, g in G, and a in A, the functions $\lambda x.f_i(g(x), a)$ and $\lambda x.f_i(a, g(x))$ are in G and their indices can be computed from a and the indices of f_i and g in constant time.
4. For all g_i, g_j in G, the composition $g_i \cdot g_j$ is in G and its indices can be computed from i and j in constant time.

These conditions ensure three properties: that no function built at a node takes more than constant time to derive from the functions of the three nodes it is replacing, that no such function takes more than constant time to evaluate, and that no such function requires more than a constant amount of storage. Together these three properties guarantee that the tree contraction algorithm executes in logarithmic parallel time.

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] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In D. Trystram, editor, *Proceedings of Parco 93*. Elsevier Series in Advances in Parallel Computing, 1993.

[3] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Control*, 81:334–352, 1989.

[4] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2:267–288, April 1984.

[5] Charles N. Fischer. *On Parsing Context-Free Languages in Parallel Environments*. PhD thesis, Cornell University, 1975.

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

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

[8] G.H. Gonnet, R.A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W.B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 66–82. Prentice-Hall, 1992.

[9] W. Daniel Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29, No.12:1170–1183, December 1986.

[10] L. Hollaar. The Utah Text Search Engine: Implementation experiences and future plans. In *Database Machines: Fourth International Workshop*, pages 367–376. Springer-Verlag, 1985.

[11] L. Hollaar. Special-purpose hardware for text searching: Past experience, future potential. *Information Processing and Management*, 27:371–378, 1991.

[12] Fulcrum Technologies Inc. Ful/text reference manual. Fulcrum Technologies, Ottawa, Ontario, Canada, 1986.

[13] Information processing – text and office systems – standard generalized markup language (sgml), 1986.

[14] I.A. Macleod. A query language for retrieving information from hierarchical text structures. *The Computer Journal*, 34, No.3:254–264, 1991.

[15] I.A. Macleod. Path expressions as selectors for non-linear text. Preprint, 1993.

[16] E.W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. In *Proceedings of the Symposium on Parallel Architectures and Algorithms*, June 1993.

[17] G. Salton and C. Buckley. Parallel text search methods. *Communications of the ACM*, 31:203–215, 1988.

[18] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

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

[20] D.B. Skillicorn. Parallel implementation of tree skeletons. Technical Report 95-380, Queen's University, Department of Computing and Information Science, March 1995.

[21] D.B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, to appear. An early version appears as Department of Computer Science Technical Report 92-329.

[22] C. Stanfill and B. Kahle. Parallel free-text search on the Connection Machine. *Communications of the ACM*, 29:1229–1239, 1986.

[23] C. Stanfill and R. Thau. Information retrieval on the Connection Machine. *Information Processing and Management*, 27:285–310, 1991.

[24] H. Stone. Parallel querying of large databases. *IEEE Computer*, 20, No.10:11–21, October 1987.