Parallel Implementation of Tree Skeletons

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

March 1995
External Technical Report
ISSN-0836-0227-
95-380

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared March 6, 1995

Abstract

Trees are a useful data type, but they are not routinely included in parallel programming systems because their irregular structure makes them seem hard to compute with efficiently. We present a method for constructing implementations of skeletons, high-level homomorphic operations on trees, that execute in parallel. In particular, we consider the case where the size of the tree is much larger than the the number of processors available, so that tree data must be partitioned. The approach uses the theory of categorical data types to derive implementation templates based on tree contraction. Many useful tree operations can be computed in time logarithmic in the size of their argument, on a wide range of parallel systems.

1 Contribution

One common approach to general-purpose parallel computation is based on packaging complex operations as templates, or skeletons [3, 12]. Skeletons encapsulate the control and data flow necessary to compute useful operations. This permits software to be written in a way that is independent of particular architectures, and indeed of underlying parallelism at all, while freeing implementations to be target-specific. Thus software is simpler to write, while at the same time portable.

Building parallel implementations for complex skeletons is difficult. In this paper we present a technique for implementing skeleton operations on binary trees. It is based on the categorical data type construction for the type of binary trees, which itself defines a useful set of tree-based skeleton operations. The construction also helps with building implementations. Implementations are SPMD, and so can be mapped to a large range of target architectures in a straightforward way. This amounts to automatic code generation.

A powerful side-effect of these implementations is that it is possible to derive costs for programs or program pieces. Furthermore, these costs are realistic in the sense that they account for communication costs. Information about program costs can be used to derive and optimise programs.

The technique is interesting as an example of a generalised kind of data parallelism applied to an irregular data structure. While the advantages of data parallelism are widely appreciated for regular data structures such as lists and arrays, there has been some doubt about whether it is as useful for less regular structures. This paper shows that irregular data structures do not necessarily rule out the benefits of high-level regularities of the sort captured by skeletons.

Trees are an important data structure. In particular many of the skeletons described here are directly applicable for parallel computation on structured text, which is often so large that parallelism is necessary to use it effectively [13]. Practical parallel implementations of such operations are described.

2 Construction of Trees

We will construct two types of trees. The first is the type of full binary trees. Although we will usually be interested in trees where the values at each node are taken from the same underlying type, it is technically useful to define trees in which leaves and internal nodes may have different types. Let A and B be arbitrary types.

Definition 1 A full binary tree is that type, $T(A, B)$, whose constructors are:

$$\begin{aligned} Leaf &: A \rightarrow T(A, B) \\ Join &: T(A, B) \times B \times T(A, B) \rightarrow T(A, B) \end{aligned}$$

Thus a tree is either a single leaf node of type A, or is formed by joining two trees together, and placing a value of type B at the join. Such trees are called full because each node has either two descendants or none.

We introduce a more general type of tree, the rose tree, which we will need for technical reasons in what follows. Binary trees are not powerful enough to model some common types of tree-structured data, for example structured text in the style of SGML, and so rose trees also have considerable intrinsic interest.

Definition 2 A rose tree is that type, $RT(A, B)$, whose constructors are:

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

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

Such trees are either single nodes of type A, or a list of rose trees, connected together by an internal node of type B. The name rose tree was coined by Lambert Meertens, and translates rhododendron, whose branches suggest this structure.

The justification for defining a type based only on its constructors comes from the theory of categorical data types [12], a particular style of initiality. This shows that giving suitably well-behaved constructors suffices to define a data type, a common structure for homomorphisms on the type, and a large set of equations relating functions on the type.

For full binary trees, homomorphisms are arrows between the free tree algebra, with carrier $T(A, B)$ and operations $Leaf$ and $Join$, and other algebras with carrier, say P, and operations

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

In fact, there is a one-to-one correspondence between homomorphisms and such codomain algebras, so we can write $Hom(p_1, p_2)$ for such a homomorphism.

```
eval_catamorphism(p1, p2, t)
case t of
      Leaf a : return p1( a )
      Join (t1, b, t2) : return p2 ( eval_catamorphism(p1, p2, t1 ),
                                     b,
                                     eval_catamorphism(p1, p2, t2 ))
end
```

Figure 1: Schema for All Tree Homomorphisms

Rose tree homomorphisms are similarly defined by carrier P and pairs of functions

$$\begin{aligned} p_1 &: A \to P \\ p_2 &: B \times P* \to P \end{aligned}$$

Because of the one-to-one correspondence between homomorphisms and their codomains, all homomorphisms on binary trees are fundamentally the same, and can be evaluated by a recursive schema parameterised by the structure of the codomain algebra. This is shown in Figure 1. This schema gives a direct way to evaluate homomorphisms on trees in sequential time linear in the size of tree, and parallel time proportional to the height of the tree, provided that p_1 and p_2 are constant time. In particular, the schema shows that a single, simple evaluator can be used to compute all homomorphisms, either sequentially or in parallel. However, there are pragmatic reasons for exploring more sophisticated implementation techniques: some homomorphisms do not require the complete structure implied by the recursive schema, so better-performing implementations can be substituted; and it is rare for the number of processors to match the number of nodes of the argument tree, so the effect of allocating multiple tree nodes to a single processor must also be considered.

The first special case of homomorphisms are maps, in which a pair of functions are applied to all of the nodes of a tree.

Definition 3 *(Map)* Given a pair of functions $f_1 : A \to C$ and $f_2 : B \to D$, the function

$$Map(f_1, f_2) : T(A, B) \to T(C, D)$$

that applies f_1 to each node of type A and f_2 to each node of type B is called a tree map.

Note that it is a homomorphism in which the component functions are

$$\begin{aligned} p_1 &= Leaf \cdot f_1 \\ p_2 &= Join \cdot id \times f_2 \times id \end{aligned}$$

The second special case are reductions, which condense or reduce the structure of a tree.

Definition 4 *(Reduction)* Given a function $g : A \times B \times A \to A$, the function

$$Reduce(g) : T(A, B) \to A$$

is the homomorphism with component functions

$$\begin{aligned} p_1 &= id \\ p_2 &= g \end{aligned}$$

The third special case are those operations in which, intuitively speaking, data must flow up the tree towards the root in order for each node to compute its local result. We further restrict these operations so that the result at a node can be computed incrementally from the results of its children.

Definition 5 *(Upwards Accumulation)* Given an arbitrary tree homomorphism with component functions p_1 and p_2 whose codomains are of type X, the upwards accumulation $\Uparrow (p_1, p_2)$ is the function

$$\Uparrow (p_1, p_2) : T(A, B) \to T(X, X)$$

given by

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

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

An upwards accumulation is show in Figure 2.

The fourth special case are those operations in which data must flow down the tree in order for each node to compute its result (that is, the result at a node depends on the data and path between it and the root). To define these operations we must first make a small technical construction. Let *Paths* be a type of non-empty concatenation lists with two mutually-associative concatenation operations which we will call *left turn* and *right turn*. The constructors

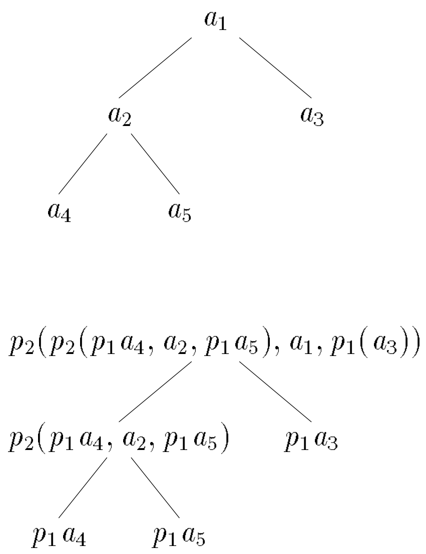

Figure 2: An Upwards Accumulation

are

$$\begin{aligned} \mathit{Singleton} &: A \to \mathit{Paths}(A) \\ \mathit{Leftturn} &: \mathit{Paths}(A) \times \mathit{Paths}(A) \to \mathit{Paths}(A) \\ \mathit{Rightturn} &: \mathit{Paths}(A) \times \mathit{Paths}(A) \to \mathit{Paths}(A) \end{aligned}$$

This defines a type that we will use to represent the paths between nodes and the root in trees, with *Leftturn* denoting the path to a left descendant and *Rightturn* denoting the path to a right descendant.

Path homomorphisms are arrows from the free path algebra to algebras with carrier P and component functions

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

with p_2 and p_3 mutually associative. They are written $\mathit{PathHom}(p_1, p_2, p_3)$.

Definition 6 *(Downwards Accumulation)* Given an arbitrary path homomorphism with component functions p_1, p_2, and p_3 whose codomains are of type X,

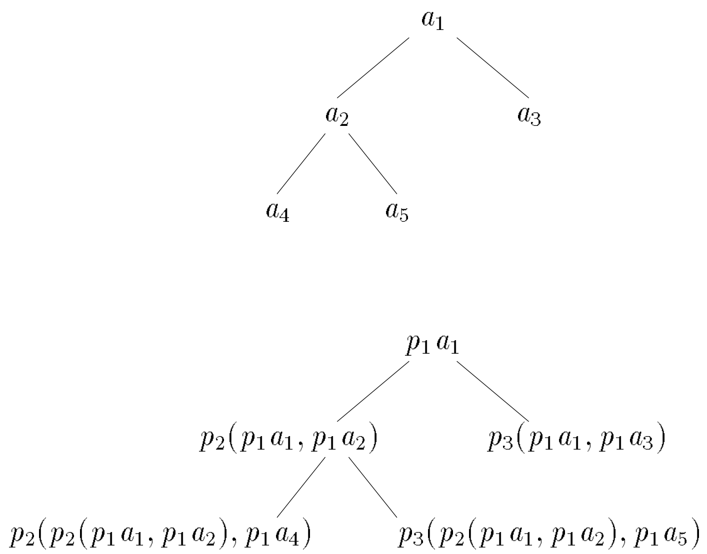

Figure 3: A Downwards Accumulation

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

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

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

A downwards accumulation is shown in Figure 3.

Rose trees also have special-case homomorphisms for maps, reductions, and upwards and downwards accumulations. The generalisations are obvious. For example, a rose tree map $RTMap(p_1, p_2)$ is the rose tree homomorphism with component functions

$$\begin{aligned} p_1 &= RTLeaf \cdot f_1 \\ p_2 &= RTJoin \cdot f_2 \times id* \end{aligned}$$

A rose tree reduction, $RTReduce(g)$, is a rose tree homomorphism with component functions

$$\begin{aligned} p_1 &= id \\ p_2 &= g : B \times A* \rightarrow A \end{aligned}$$

In the next section we examine how each of these restricted forms of homomorphisms can be implemented, beginning with the easiest case where the number of processors available for computation is the same as the number of nodes in the tree argument.

3 The Case $p = n$ for Binary Trees

For each of the special-case tree homomorphisms described in the previous section we will construct an implementation or implementations and compute their complexities. We assume that each homomorphism is applied to a tree of n nodes that has been mapped to an n-processor parallel computer with a single tree node placed on each processor. The time complexity of some implementations is sensitive to the communication topology of the parallel computer, so we discuss both together.

A map operation is implemented by applying the base functions to each node of the tree in place. In other words, given the operation $Map(f_1, f_2)$ each processor decides whether it holds an internal or leaf node, and applies either f_1 or f_2 to that node. No communication is required. The parallel time complexity of this implementation is

$$t_n(Map(f_1, f_2)) = max(t_1(f_1), t_1(f_2))$$

where $t_i(f)$ is the time complexity required to compute f on i processors.

A reduction operation, $Reduce(g)$, where $g : A \times B \times A \rightarrow A$, can be implemented in a number of ways depending on the properties of g. The most direct implementation is to use the recursive schema. If we assume that g is constant space, that is the size of the result is the same size as the size of the arguments, and that the implementing architecture has a rich enough communication topology for the tree to be embedded without dilation, then the parallel time complexity of $Reduce(g)$ is given by

$$t_n(Reduce(g)) = ht \times t_1(g)$$

where ht is the height of the tree argument. Note that the height of the tree is linear in the worst case.

When the operation g is not constant space, the parallel time complexity must increase because of the necessity of moving greater amounts of data around the tree. Even for an embedding of the tree without dilation, moving m data items between neighbours must take time proportional to m. Consider an operation g that takes arguments of size m and produces a result of size $2m$. The worst case scenario is a skewed tree, in which the two lowest leaves must transmit data of size m, their parent must transmit data of size $2m$, and so on. It therefore takes time at least

$$m + 2m + 3m + \ldots + ((n-1)/2)m$$

for the data to reach the root, giving an overall time complexity quadratic in the size of the tree.

A similar computation can be carried out for a g that takes arguments of size m to a result of size $m+1$. It would be attractive to generalise this to take into account any effect of g on result size, but this has so far proven too complicated for practical use.

Under certain fairly mild conditions on g, it is possible to replace the recursive implementation schema by an implementation based on *tree contraction* [1, 5, 10]. The basic idea of tree contraction 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 only one descendant is a leaf. 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.

The tree contraction algorithm is standard, so we only sketch its operation here. It depends on two symmetric operations, *contractl* and *contractr*, that replace a node, its leaf descendant, and its other descendant, by a single new node. Associate a value and a function with each node initially – in our setting the value is the content of that tree node, and the function is *id* for leaves and g for internal nodes.

Consider the situation shown in Figure 4. If the function associated with the new node can be partially evaluated, so that actual computation occurs in constructing this new function, then useful progress is made towards the final result during each local contraction, and many of them can be carried out in parallel.

The overall structure of the algorithm is as follows:

1. Number the leaves from left to right, beginning from 0, so that disjoint locations can be chosen to apply the local contraction operations. Such a numbering can be done using the Euler Tour technique in $O(\log n)$ time using $O(n/\log n)$ processors [4].

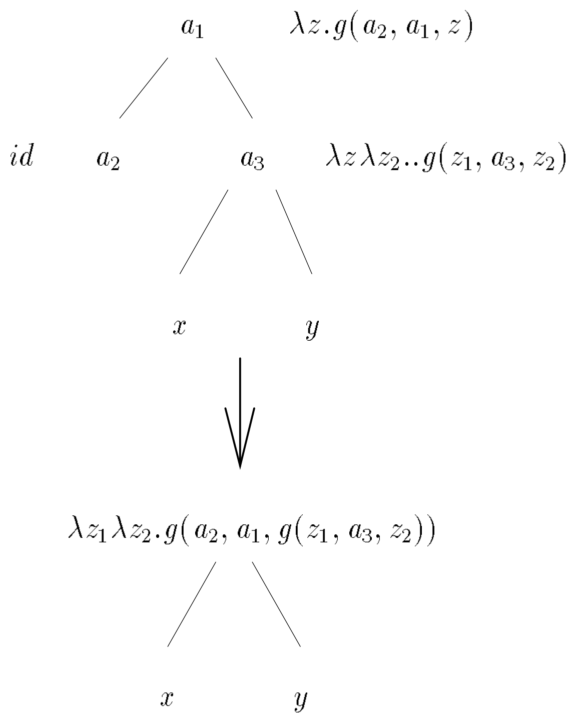

Figure 4: A *contractl* Operation

2. For every node whose left descendant has an even number, carry out *contractl*. If the right descendant is also a leaf, the result of *contractl* is a leaf which keeps the number of the original left descendant.

3. For every node that was not newly produced on the previous step, and whose right descendant has an even number, carry out *contractr*.

4. Divide the number attached to each leaf by two, and take the integer part.

Steps 2-4 are repeated until only a single node remains. Since about half the nodes are leaves at any step, and each local contraction replaces three tree nodes by one, it is straightforward to see that the algorithm runs in a number of steps logarithmic in the size of the tree.

To justify the overall logarithmic time complexity of the algorithm, each local contraction must be constant time and space. Conditions on g and the functions derived as its compositions and partial evaluations that ensure this are given by Abrahamson *et al.* [1] as:

1. For all nodes, the sectioned function $f = \lambda x_1 \lambda x_3 . g(x_1, b, x_3)$ of type $A \times A \rightarrow A$ is drawn from an indexed set of functions, F, that contains the identity function.

2. All functions in F can be applied in constant time.

3. If f_i and f_j are in F, the functions $f_i(a_2, a_1, f_j(x_1, a_3, x_3))$ and $f_i(f_j(x_1, a_2, x_3), a_1, a_3)$ are in F, and their indices can be computed from the indices of f_i and f_j in constant time.

The original version of tree contraction was described for the EREW PRAM, but the same performance can be achieved on the hypercube [9], and the technique can presumably be extended to the cube-connected-cycles topology. Note that we no longer need the assumption of an undilated embedding of the tree into the communication topology of the target architecture. Thus this implementation is both faster and more general, when the component functions have the necessary properties.

If the time complexity of internal functions, their partial evaluations, and compositions has some other complexity, the tree contraction algorithm still works correctly. For example, if all of the required functions have logarithmic time complexity, then the overall parallel time complexity of tree contraction will be $log^2 n$, and so on.

There are a number of ways of implementing upwards accumulations. The first is to use the implementation implied by the definition, that is to apply the *subtrees* function, and then map a tree homomorphism over it. However, this approach is clearly computationally expensive, since the *subtrees* function, applied to a tree of size n, produces a result of size n^2, to which further homomorphisms must still be applied.

The second method of implementation is to use the recursive schema directly, since the upwards accumulation is a tree homomorphism. This is again computationally unattractive, not least because it requires an unnecessary decomposition step. In fact, upwards accumulations were carefully defined to capture computations in which data flowed upwards in the tree and, given one processor per node, this is the sensible way to implement them. So the third method of implementation is to pass results upwards, computing partial results on the way.

The parallel time complexity of an upwards accumulation implemented in this way is given by

$$t_n = t_1(p_1) + ht \times t_1(p_2)$$

under the assumptions that there is an embedding of the tree in the parallel computer's topology without dilation, and that p_2 is constant space.

Once again, the linear time complexity is caused by the data dependencies along the path between the root and the furthest leaf. And once again, under certain conditions on p_2, a fourth implementation, based on tree contraction, can be built. The algorithm is complex, and is fully described in [6]. The essential idea is to carry out a tree contraction, but leaving the contracted nodes connected to the new node that replaced them. A subsequent phase takes some partial results of the contraction and sends them to previously contracted nodes, which may then compute their final values.

Under the same conditions on component functions as above, upwards accumulations can be computed in parallel time logarithmic in the size of the tree, regardless of how skewed it is. The result holds for the EREW PRAM and the hypercube.

Downwards accumulations can be implemented in the same four ways, including a fast parallel algorithm based on tree contraction. This algorithm becomes even more complex and requires further modification of the tree contraction algorithm [6], but has the same parallel time complexity.

This completes the discussion of implementations for special homomorphisms on binary trees for the simple case when a single processor is available for each tree node. This is clearly an unrealistic assumption, but before we move on to considering implementations using fewer than one processor per node, we turn to considering implementations of homomorphisms on rose trees, as these are required later on.

4 The Case $p = n$ for Rose Trees

Rose trees are much more common in applications than binary trees, so homomorphism implementations for them are interesting in their own right. We will, however, use these implementations to implement binary tree homomorphisms in the next section. As before, we assume one processor per tree node.

Rose tree maps can be implemented in exactly the same way as binary tree maps, applying the operation locally in each processor. As expected,

$$t_n(Map_n(f_1, f_2)) = max(t_1(f_1), t_2(f_2))$$

Rose tree reductions can be implemented recursively, giving a parallel time complexity of

$$t_n(Reduce_n(g)) = ht \times t_1(g)$$

Notice that here the type of g is

$$g : B \times A* \rightarrow A$$

so that the sequential time complexity of g might be expected to be linear in the number of descendants of each node. The overall parallel time complexity is no worse than linear in the size of the tree regardless of its shape, since a single edge of the tree can potentially be handled by each processor on each step.

For certain special cases of g, namely those that can be expressed as

$$g = g'.id \times \oplus/$$

where

$$\begin{aligned} g' &: B \times A \rightarrow A \\ \oplus &: A \times A \rightarrow A \text{ (associative)} \end{aligned}$$

the $\oplus/$ part of each application of g can be parallelized, and no additional processors are required. This reduces the time complexity of a single application of g to logarithmic in the number of descendants of the node at which it is being applied.

The tree contraction implementation of reductions can also be generalised to rose trees. The idea is to have each processor convert the node it holds into a small section of binary tree and then apply the binary tree algorithms to this enhanced tree. The rearrangement is shown in Figure 5. This rearrangement requires the processor holding each node to be able to determine the processor that holds its right sibling in constant time. If we assume that each node holds

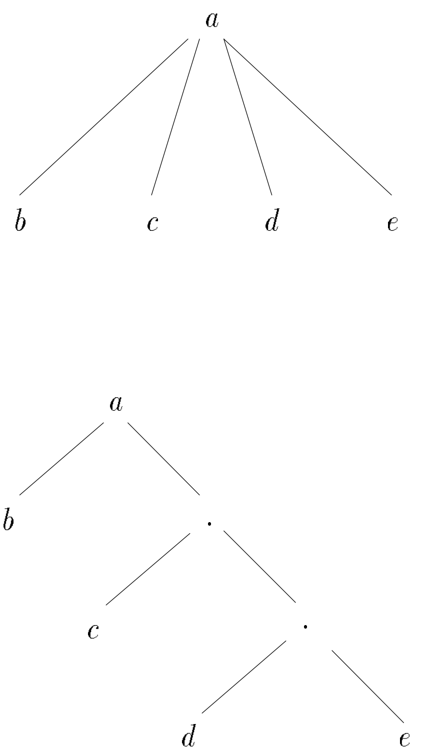

Figure 5: Local Rearrangement of a Rose Tree into a Binary Tree

the processors in which its children are located as part of the tree structure itself, then each child processor can determine the sibling address by reading it from the parent (and each read is from a disjoint location). This assumption does not seem particularly strong given likely ways in which the tree storage might actually be implemented.

Tree contraction in the rose tree is now replaced by tree contraction in the binary tree. The rose tree reduction operation g must be expressible as

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

as before, and both g' and $\oplus$ must satisfy the requirements on component functions. The functions associated with each node initially are:

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

The tree contraction algorithm relies on numbering the leaves of the binary tree in time logarithmic in its size. The Euler Tour technique can be extended to this case, provided each child node can find out the location of its right sibling in constant time.

Under these conditions, a rose tree contraction takes parallel time

$$t_n = \log 2n \times t_1(\oplus)$$

if g' is constant time.

Upwards and downwards accumulations have similar implementations. Both can be implemented using data flow with parallel time complexities proportional to the height of the tree. Once a rose tree has been reconfigured into a binary tree, as was done for the rose tree contraction above, the binary tree versions of upwards and downwards accumulations can be used to implement rose tree accumulations with parallel time complexity logarithmic in the size of the tree.

5 Using $p < n$ Processors

When there are fewer processors than nodes of the argument tree, which is much more realistic than the one-processor-per-node assumption we have used so far, implementations must partition the tree and map pieces of it to each processor. This introduces new complexities.

The first issue is the level at which the tree structure is to be partitioned. In general the nodes of a tree may themselves be elements of types with internal structure, for example a tree of lists of integers. Should a tree be divided so that there are an equal number of tree nodes allocated to each processor, or should it be divided so that there are an equal number of basic data objects allocated to each processor? Taking the first approach may lead to serious load imbalances – for instance a tree of lists may contain lists of very different lengths. Taking the second approach may generate extra communication if a single tree node gets split across more than one processor. We adopt the first approach because it is simpler, it reflects the importance of maps in homomorphisms, and it allows the implementations to be polymorphic, since the type of the nodes is not used in decisions about partitioning. Others have chosen to work at the finest grain, for example NESL [2].

The second issue is that the types of the functions applied to pieces of the argument are not the same as the type of the original program. A method for systematically generating the actual code for each processor from the original program is required. Here, the structure of homomorphisms is very helpful, enabling us to develop a formal code generation system.

6 Partitioning into Subtrees

For some types, the issue of how to partition objects is easily resolved. When a type is *separable*, that is constructors either insert elements of some base type into the constructed type, or recursively build elements of the constructed type into larger ones, but not both at once, there is always a *flatten* operation, a reduction using those constructors that manipulate structure. For example, the type of join or concatenation lists is separable, because the *make singleton* constructor inserts elements of the base type while *concatenate* ($+\!\!+$) joins lists but adds no new elements. The *flatten* operation is $+\!\!+/$ (reduce with concatenation) which flattens a list of list into a single list.

Neither homogeneous binary nor homogeneous rose trees are separable, so there is no automatic flatten function. Instead we choose one with useful properties, justifying it on pragmatic grounds.

We define a pair of functions, $dist_p$ to break a tree into p subtrees, and *flatten* to flatten p subtrees back into a single tree. They are related by this equation

$$flatten \cdot dist_p \;=\; id \qquad (1)$$

whose importance was first pointed out by Roe [11]. There are several choices for these functions. The obvious technique of breaking a tree into a tree of trees is very limited because of the requirement that the top-level structure

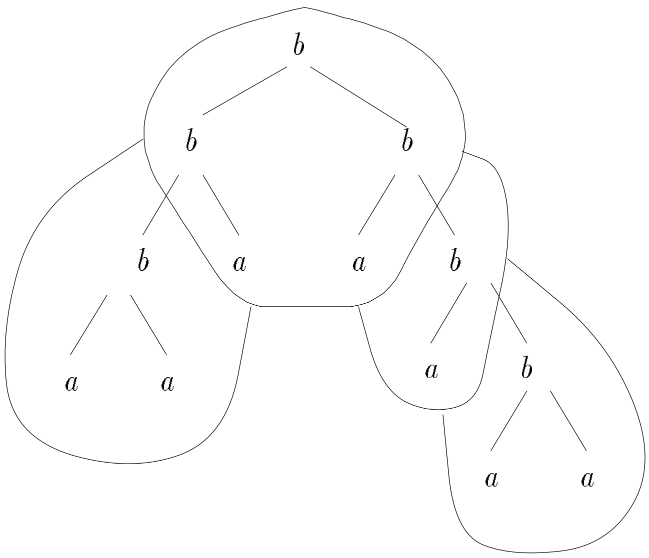

Figure 6: The Effect of Partitioning

remain a binary tree. This can be made to work, but it is very hard to divide trees into equal-sized subtrees.

It is more useful to take suitable type signatures to be

$$dist_p : T(A, B) \to RT(\,T(A, B), T(A+?, B))$$

and

$$\mathit{flatten} : RT(\,T(A, B), T(A+?, B)) \to T(A, B)$$

where ? acts as a placeholder for detached subtrees, and so is any one-element set. This definition seems to best permit the kind of tree partitioning that is needed to get partitions of approximately equal size from arbitrary binary trees. The effect of $dist_p$ can be seen in Figure 6. The function $dist_p$ can be any function with this type signature, although there are clearly some pragmatic requirements on it. It is quite difficult to build such a function and have it perform reasonably. Fortunately its exact construction does not affect anything else discussed here.

The function *flatten* is the rose tree homomorphism whose component functions are

$$\begin{aligned} p_1 &: T(A, B) \to T(A, B) \;=\; id \\ p_2 &: T(A+?, B) \times T(A, B)* \to T(A, B) \;=\; \mathit{flat} \cdot \mathit{zipup} \end{aligned}$$

where *zipup* is a polymorphic (in Z) function

$$zipup : T(A+?, B) \times Z* \rightarrow T(A + Z, B)$$

which replaces each leaf of its left argument tree of type ? by the next element of the list of Zs, reusing list elements if necessary. The function *flat* is of type

$$flat : T(A + T(A, B), B) \rightarrow T(A, B)$$

and observing that $A + T(A, B) \cong T(A, B)$ we see that *flat* is $Reduce(Join)$.

Now consider a composition of functions

$$S3 \cdot S2 \cdot S1$$

for which we want to build an implementation. This program is equivalent to the following by Equation 1:

$$(flatten \cdot dist_p) \cdot S3 \cdot (flatten \cdot dist_p) \cdot S2 \cdot (flatten \cdot dist_p) \cdot S1 \cdot (flatten \cdot dist_p)$$

or, rebracketing,

$$flatten \cdot (dist_p \cdot S3 \cdot flatten) \cdot (dist_p \cdot S2 \cdot flatten) \cdot (dist_p \cdot S1 \cdot flatten) \cdot dist_p$$

We see that the implementation of each of the Si is the function $dist_p \cdot Si \cdot flatten$. The implementation expects its argument to be a rose tree of trees, gathers them together into a single tree, applies the original function, and then breaks them up into a rose tree of trees again. Fortunately, it will often be possible to replace this expensive process by a version of the function that works "in place", that is which can be mapped over the rose tree, defining the functions to be applied sequentially to each component tree. The whole program begins with an initial breaking up of the argument tree into subtrees, and ends with a result-gathering step, which assembles the partial trees on each processor into a single tree. With present architectures we can ignore the cost of these initial and final steps since they are part of the overhead of all programs.

Observe that, if the function $dist_p$ is determinate, we also have the identity

$$dist_p \cdot flatten \ = \ id$$

when applied to a rose tree of trees whose shape was produced by an application of $dist_p$. For, in that case, the flattened tree will be broken up into its original pieces by the subsequent application of $dist_p$. We make use of this in what follows.

There is a general result for initial data types, known as the promotion theorem [8], which allows operations such as *flatten* to be permuted with other operations in the following sense

Theorem 7 *(Promotion through flatten)* For an arbitrary tree homomorphism, $Hom(h_1, h_2)$

$$\begin{aligned} Hom(h_1, h_2) \cdot flatten &= RTReduce(Reduce(h_2) \cdot zipup) \cdot \\ &\qquad RTMap(Hom(h_1, h_2), Map(h_1 + id, id)) \end{aligned}$$

This result shows that the application of any tree homomorphism after a *flatten* is equivalent to applying a rose tree map over the partitioned tree (that is applying $Hom(h_1, h_2)$ or $Map(h_1 + id, id)$ to the subtree within each processor), and then executing a globally-parallel rose tree reduction to glue the partial answers together. We use this theorem to rearrange terms of the form

$$dist_p \cdot Hom(h_1, h_2) \cdot flatten$$

to

$$dist_p \cdot RTReduce \cdot RTMap$$

For a Map operation with component functions $f_1 : A \rightarrow C$ and $f_2 : B \rightarrow D$ so that

$$Map(f_1, f_2) : T(A, B) \rightarrow T(C, D)$$

we get

$$\begin{aligned} & dist_p \cdot Map(f_1, f_2) \cdot flatten \\ &= dist_p \cdot RTReduce(Hom(id, Join \cdot id \times f_2 \times id) \cdot zipup) \cdot \\ &\qquad RTMap(Map(f_1, f_2), Map(f_1 + id, id)) \end{aligned}$$

However, we observe that

$$flatten = RTReduce(Hom(id, Join) \cdot zipup)$$

which suggests a small rearrangement to give

$$\begin{aligned} & dist_p \cdot Map(f_1, f_2) \cdot flatten \\ &= dist_p \cdot flatten \cdot RTMap(Map(f_1, f_2), Map(f_1 + id, f_2)) \\ &= RTMap(Map(f_1, f_2), Map(f_1 + id, f_2)) \end{aligned}$$

Thus the implementation of a tree map becomes a rose tree map, executed in parallel, with the mapped operation being a sequential tree map over the subtree held by each processor. This result is an example of an implementation that acts in place, and hence no redistribution step is necessary.

Given a reduction $Reduce(g)$, we get

$$\begin{aligned}
& dist_p \cdot Reduce(g) \cdot flatten \\
& \quad = \; dist_p \cdot RTReduce(Hom(id, g) \cdot zipup) \cdot \\
& \qquad\quad RTMap(Hom(id, g), Map(id + id, id)) \\
& \quad = \; dist_p \cdot RTReduce(Reduce(g) \cdot zipup) \cdot RTMap(Reduce(g), id)
\end{aligned}$$

The implementation is slightly complex. Recall that the argument is a rose tree of trees. The Map step applies the $Reduce(g)$ to the trees that are leaves of the rose tree, doing nothing to the internal node subtrees. The second step applies a rose tree reduction to the outer rose tree, at each step zipping the results of previous reductions into the structure and then reducing the trees at each node using $Reduce(g)$. The process is shown in Figure 7.

Here the effect of the operation is to collect the result at a single processor. If there are further parallel operations to be applied to the result of this operation, a redistribution across processors is necessary, and the cost of doing this must be included in the cost of the whole program.

The general strategy then for building implementations is to insert the expression $flatten \cdot dist_p$ between every pair of functions in the program and then use the promotion strategy to move $flatten$ operations leftwards (later) in the program, cancelling them wherever possible with $dist_p$s. The result is a architecture-independent template for implementing the program, which can then be used to generate machine-specific code.

7 The Case $p < n$ for Binary Trees

We are now in a position to compute the costs of executing common operations where the argument tree has been partitioned across processors.

The implementation of tree map involves a parallel rose tree map, each component of which is a tree map, either $Map(f_1, f_2)$ or $Map(f_1 + id, f_2)$, applied to the subtree held by each processor. An ideal implementation of $dist_p$ would ensure that the subtree held by each processor is of size n/p. However, most implementations will not be able to achieve this for all trees. Let us assume that the maximum number of nodes in a subtree is x and clearly $x \geq n/p$.

The complexity of tree map is given by

$$t_p(Map_n(f_1, f_2)) \; = \; x \times max(t_1(f_1), t_1(f_2))$$

In other words, the time taken for a parallel implementation of tree map is

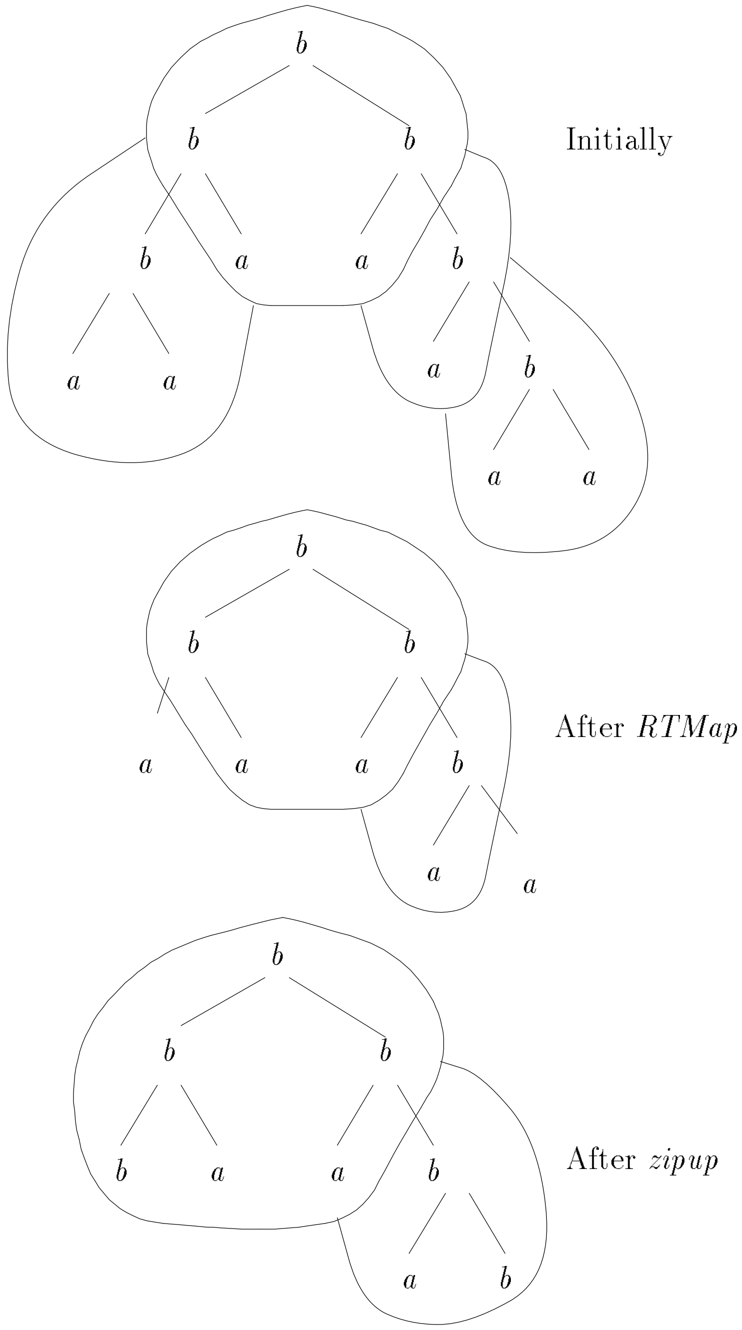

Figure 7: The Implementation of a Reduction

the time it takes for a sequential implementation of tree map on the largest subtree.

The case of tree reduction is more complex. The time complexity of $RTMap(Reduce(g), id)$ is

$$t_p(RTMap(Reduce(g), id)) = x$$

because each part of the $RTMap$ is a sequential reduction on a subtree of size at most x. The time complexity of $RTReduce(Reduce(g) \cdot zipup)$ is

$$\begin{aligned} t_p(RTReduce(Reduce(g) \cdot zipup) &= ht \times t_1(Reduce(g) \cdot zipup) \\ &= ht \times x \end{aligned}$$

since the cost of $zipup$ is negligible if we assume that pointers are maintained between subtrees (and it is hard to imagine an implementation where this wasn't true). In the worst case this gives a parallel time complexity for tree reductions of at least n^2/p. If $dist_p$ partitions the tree so that there are few partitions along any branch, that is the partitioned tree is not very deep, this can be reduced by a further factor of p. However, this complexity is disappointing, so we turn to an implementation based on tree contraction.

There are two problems with the existing implementation. The first is that the $RTMap$ only applies $Reduce(g)$ at subtrees that are leaves of the top-level tree. This leaves a lot of work to be done during the subsequent $RTReduce$. The second problem is that the $RTReduce$ uses the flow of data through the tree, and its performance is therefore limited by the height of the top-level tree. Both of the these problems can be avoided for suitable gs by using contraction at both levels of the tree.

First we replace $RTMap(Reduce(g), id)$ by

$$RTMap(Reduce(g), PReduce(g)) \tag{2}$$

where $PReduce(g)$ is a partial tree reduction mapping a tree of type $T(A+?, B)$ to a function $\hat{g} : ?* \rightarrow A$. The operation *contractl* is replaced by an operation that composes the associated functions of the two internal nodes, but does not evaluate the arguments from leaves, even if they are known. The nodes corresponding to unevaluated leaf arguments are not removed but become 'shadow' nodes, and take no further part in the tree contraction.

Provided that all of the partial compositions of these functions can be computed in constant time and space, this partial tree reduction can be completed in linear sequential time. If we again assume that $dist_p$ arranges for pointers to be kept from parent subtrees to their descendant subtrees, the result of the operation in Equation 2 is a rose tree in which each internal node has an associated function of type $A* \rightarrow A$, and each leaf node has an associated value of type A. Under suitable conditions on the associated functions, a rose tree implementation of tree contraction can be used to compute the final result.

The operation given by Equation 2 has time complexity

$$t_1(RTMap(Reduce(g), PReduce(g))) = x$$

while the rose tree reduction has time complexity

$$t_p(RTReduce) = \log p$$

giving an overall time complexity bounded below by

$$t_p(Implementation\ of\ Reduce(g)) = \log p + n/p$$

The conditions on g are that compositions and partial evaluations of partial functions $\hat{g}$, based on g, are constant time and space. Functions g that are based on reductions $\oplus/$ can certainly be composed in constant time. It might appear that constant space is a problem because of the necessity to remember which positions in the list argument are 'missing'. However, this is taken care of by our assumption of pointers to subtrees. Therefore, as we might expect, reductions over binary trees can be implemented by partitioned reductions with the expected time complexity $\log p + n/p$.

This implementation can be adapted for upwards and downwards accumulation, as before, to give the same time complexity for those operations.

8 Conclusions

We have presented a technique for implementing skeleton operations on trees. The strengths of the technique are that:

- implementations can be derived from programs automatically using equations that arise from the theory used to build the tree data type and its homomorphic skeletons;
- all scales of parallelism can be used, from the modest amounts available today up to massively parallel systems;
- the cost of a program can be computed from its implementation (and this cost is a real cost since it takes into account communication latency);
- the implementations remain architecture-independent, so that compilers can be based on common code until target-specific code generation. In particular, optimisation can be largely architecture-independent.

Such implementations are immediately useful because of a range of applications in which trees are important data structures. One such is structured text, where text archives are so large that parallelism is the only way to make sophisticated search (for example, path expressions [7]) possible.

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] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

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

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

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

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

[8] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.

[9] 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.

[10] 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.

[11] P. Roe. Derivation of efficient data parallel programs. Technical report, Queensland University of Technology, December 1993.

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

[13] D.B. Skillicorn. Structured parallel computation in structured documents. Technical Report 95-379, Queen's University, Department of Computing and Information Science, March 1995.