University of Edinburgh

Department of Computer Science

Distributed Recursive Datatypes

M.Sc. Project Report

Tim Lewis

Abstract

Recursive data structures are a common feature of program design. Using them in a distributed system raises many problems. An existing solution uses a node-naming strategy to ensure that all the processors can refer to the nodes of the structure in an unambiguous way. Motivated by the problem of evaluating functions with commutative recursive function calls, I have extended the method to uniquely name the nodes that occur in the structures arising from these evaluations. The method also allows nested recursive structures, such as trees of lists to be described simply. The method has been proved correct, and implemented on the Cray T3D, along with a set of library functions that allow a user to manipulate recursive datatypes in a parallel environment.

Acknowledgements

I would like to thank the following people for their help:

My supervisor, Damal Arvind, for his guidance and encouragement throughout the period of the project, and especially in the last two weeks.

Prof. C. Lengauer, who acted as a catalyst for some of the ideas.

Ruth, for her support, understanding and food.

Contents

Chapter 1

Introduction

1.1 Recursive Functions

We begin by considering a problem that leads to a recursively defined data structure. We have a recursively defined integer function of the form:-

```
f(x) = if basecond(x) then return base(x)
 else return g( f(a(x)), f(b(x)), f(c(x)), x)
```

Then the evaluation of the function for a particular value of x will be a tree-like structure, each computation of $f(y)$ will require three sub-computations of $f(a(y))$, $f(b(y))$ and $f(c(y))$, which are combined to produce the result. This function could then be evaluated by using a three-way tree structure, the values of the function being stored and updated at each node of the tree, until the value at the root is computed. If we know more about the properties of the *descent functions* [1] a, b and c (properties known as *descent conditions*), then we could exploit those properties in the evaluation of the function. We will refer to this type of function as a function with commutative recursive calls.

For example, in [1], they consider the problem where all the functions *commute*, $ab(x) = ba(x)$. This can be exploited, because we can predict in advance many of the nodes in the data structure that share the same value. This can be used to avoid re-computing the values of f at these shared points. Another way of solving this problem of re-computation would be

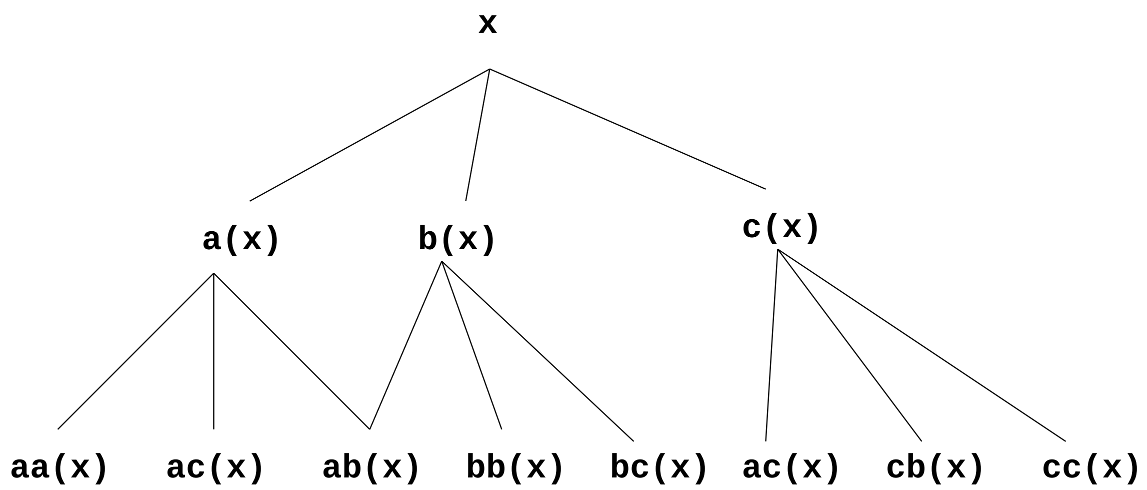

Figure 1.1: Function call graph of function with redundancy

by the use of a *memo-table*, whereby whenever f is computed, the value is stored and can simply be looked up if it is subsequently needed. If the evaluation of the function results in a large amount of computation, it is natural to look to ways of parallelising the work. In this situation, memo-tables will be inefficient, as the table must be a global to all the processors, and lookups will require a communications overhead, plus the synchronisation problems when the table needs to be updated. In [1] they use a system of *memo-lists*, which store, local to each part of the function call graph, all the re-usable values. Then during the computation, these lists are passed around, with the lookup and adding of the memo-table system reduced to quicker *add-head* and *remove-head* operations on these lists. Because the lists are no longer global, this method has the potential to be parallelised to a certain extent. The method is heavily reliant on the fact that all of the descent functions commute.

If we extend the generality to the situation where not all of the functions commute (say only a and b), then we arrive with a call graph as in fig. 1.1. Regarding method [1] it would be difficult to extend this method directly to exploit the remaining redundancy in the problem. The aim will be to use a more general method of manipulating tree-like structures in parallel, that also allows the redundancy in these problems to be expressed and exploited. So first we turn to the more general problem of manipulating recursive data structures in a parallel environment.

```
class Tree {
  int value;
  Tree *left;
  Tree *right;
}
```

Figure 1.2: Definition of a Tree structure

```
Map(Tree *t, int f(int)) {
  if(t == NULL) return;
  Tree *l, *r;
  t->value = f(t->value);
  l = t->left;
  r = t->right;
  Map(l);
  Map(r);
}
```

Figure 1.3: A Map function over a Tree

1.2 Distributed Recursive Datatypes

In [3] the author explores the problem of implementing distributed recursive datatypes. The programming model that he uses is Single Program, Multiple Data. The SPMD model simplifies the problem of programming multiple processors by distributing the main data structures and having each processor follow through the same basic program, but operating only on its own local data. This immediately introduces the idea of a processor *owning data*. We consider a simple integer tree data type, implemented in a sequential language such as C++ (fig. 1.2).

In a one processor environment, a tree would consist of a number of such objects dynamically created in the single address space, linked together using the pointers. If we were to implement this in the most naive way in a distributed system, we would firstly need to introduce the idea of a global pointer, that stores the processor name along with the address of the data. The pointers *left* and *right* would then become global pointers. We could then distribute these objects amongst the processors. If we then wanted to map a function f over the tree (doubling the integer value, say) we would use code as in fig. 1.3.

This introduces a problem. If the tree objects are stored on different processors then the

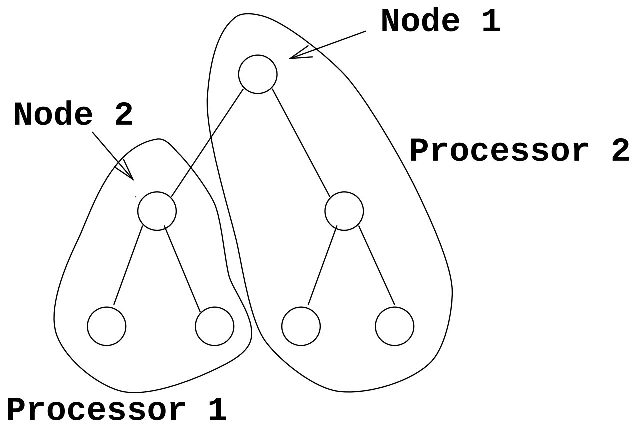

Figure 1.4: Traversing a tree

pointers that link them together will also be stored on different processors. Then in order to traverse through the structure each processor will have to communicate with other processors to find what the sub parts of the structure are. In fig. 1.4, suppose that processor 1 wants to traverse from node 1 to node 2, which it owns. It must first communicate with processor 2, which owns node 1, in order to find which node is the correct one to move to. Ideally, we would want processor 1 to know the whole structure of the tree, realise that node 2 was the one it wanted to traverse to, and move there directly without any communication.

This introduces the notion of *node naming*. If we can allocate to each possible node in the structure a unique name, and a scheme for mapping nodes to processors then we will not need the communication. In principle, a node naming system works as follows. If x is a node in the structure with sub node s, then given the name n of x, we need to be able to generate the name of s. As a concrete example, for a binary tree it is simple to name each node using an integer. The root node would be named 1, and if n is the name of a node, then its left sub-node would be named $2n$, and its right sub-node would be $2n+1$. In [3] there is a simple default strategy for trees of degree k, or alternatively the user can supply their own.

1.2.1 Aliases

If we are dealing with a structure where there is some sharing of nodes, a situation that is caused by a node having more than one parent. There will then be more than one possible name for the node. In fig. 1.5, if we assume that we are using the naming scheme outlined

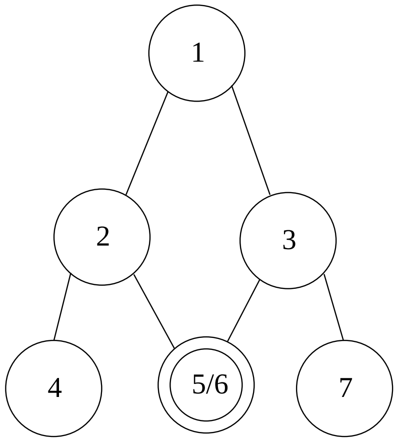

Figure 1.5: Multiple Node names

above, the ringed node can be assigned two different names, depending on whether it was created from node 2, or node 3.

The method used in [3] deals with this firstly by naming the nodes in the aliased structure according to the names for an unaliased structure. This generates multiple names for the nodes (*aliases*) which have to be created and manipulated together. So, for each node we need to maintain a unique name, plus a list of its aliases. If we then traverse through a node using an alias, we need to map that alias name to the unique name (*normalisation*). If a parent node of an aliased node is then deleted, this may require the deletion of the unique name, in which case a new one will have to be chosen from the list of aliases, and any sub-nodes will then have to be re-named.

If we can set out in advance which nodes are going to be shared in the structure, then we can design a node naming strategy that will assign to each node a unique name, that can be computed regardless of the node from which it was reached. In the above situation where we have the redundancy from the commutativity of some of the directions, we know in advance which nodes are going to be shared, and can exploit this to produce a unique naming scheme.

1.2.2 Mapping

We now need to map these names to processors to complete the scheme. Again [3] provides a default strategy for this. In [2] although they do not use a node naming scheme, they still require the user to explicitly map the data items in the structure to processors.

The mapping system we use will only take into account the structure of the data type, and the name of the node that is to be allocated. The drawback of this is that we do not take into account which nodes already exist. The result of this will be that for unbalanced structures the nodes will not be divided out as equally as possible, some processors may have many more nodes than others, and will therefore be required to do an unfair amount of computation. This is the disadvantage of a static allocation scheme, as opposed to a dynamic scheme, which would take account of the imbalance and re-distribute the nodes as they are allocated. A dynamic allocation scheme would be a valuable addition to the whole system, but is not a problem that will be considered here.

1.3 Aims

In order to extend the node-naming scheme to uniquely name the nodes in a function call graph, where we aim to exploit the redundancy in the structure, it will be necessary to work with a name that consists of the route or *path of directions* from the root node to the node in question. Providing a unique name for each node will be a major part of making the system work. Using full path names for the nodes will also lead us to be able to assign names automatically for a wide range of structures, without a user having to explicitly name the nodes themselves. It will be necessary to find a way to describe the structure of nested recursive data structures such as trees of lists, that can then be converted to a naming scheme. Finally, implementing the method and providing a library of routines that allow a user to manipulate structures in parallel in much the same way that a sequential program would be written will complete the project.

Chapter 2

Describing Structures

2.1 Introduction

In this chapter, we set aside the problem involving functions with commutative recursive calls, and consider the problem of implementing a set of library functions that allow a user to produce code that will manipulate a recursive structure on a parallel machine, such as the Cray T3D. We first turn to the problem of how to describe the structure of such data types.

In languages that allow pointer based data structures, such as C, Pascal or ML, there is a considerable amount of freedom over what structures can be defined and manipulated. Since the aim to provide a library of routines rather than a compiled language, there will have to be some restrictions on the kind of structures that can be created using my system. Also, the node naming method requires that all processors have complete knowledge about the layout of the whole structure from the outset. This negates the possibility of allowing nodes to be connected arbitrarily. So we will consider only the kind of structures where this complete knowledge applies. This will primarily consist of data types such as trees and lists, although it will be possible to use bi-directional links and nested structures. Finally, the idea of commutative directions will allow the structures arising from the evaluation of recursive functions to be handled, plus irregularly shaped 'diamond DAG' structures.

```
class RecStruct {

    Data    value;
    RecStruct *dir1;
    RecStruct *dir2;
}
```

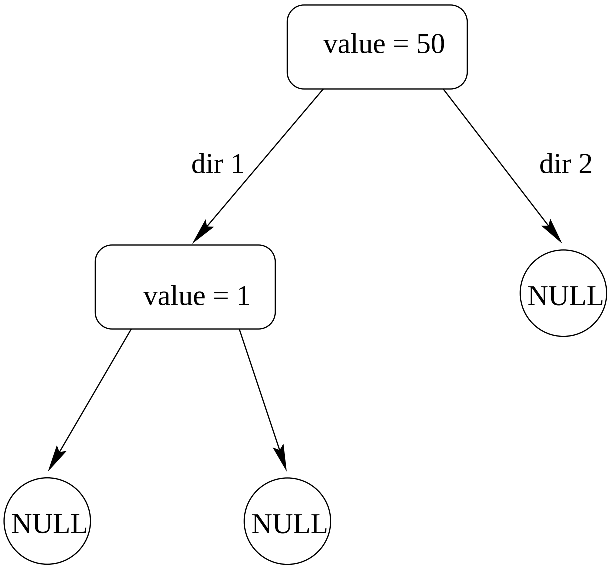

Figure 2.1: Definition and layout of a binary tree structure

2.2 Recursive Pointer Based Data Structures

Consider a recursive data structure defined in a sequential language, such as C++, see fig. 2.1. The structure stores a value of type *Data*, and has pointers to two other structures of the same type, giving the definition a recursive nature. A null pointer would indicate that there was no further sub-structures in that direction.

This particular definition would define a binary tree structure, providing that the sub-structures are distinct from each other, and distinct from the parent structure. Now consider a way to generalise this structure, by allowing the pointers to point to different types of structure.

```
class RecStruct[1] {

    Data[1] value;
    RecStruct[2] *dir[1];
    RecStruct[3] *dir[2];

    ...

    RecStruct[k] *dir[k-1]
}
```

Figure 2.2: A more general recursive structure

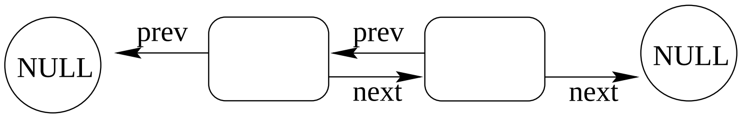

Figure 2.3: A doubly linked list

2.2.1 Nested recursive structures

In its most general form, the structure would be defined as in fig. 2.2. Each type of structure could potentially store a different type of data. These definitions would allow us to describe nested structures such as lists of lists, trees of lists and trees of trees.

2.2.2 Sharing nodes

So far, the definitions have assumed that all sub structures are distinct, and that arbitrarily traversing a structure can never bring us back to the same position. I shall extend the definition to allow for two situations where we can reach the same node from different routes. This does not extend to the most general case where any node can access any other node, such as in a graph, but it does allow us to have some additional structure.

Firstly, the idea of two directions being inverses of each other. The simplest instance of this would be a doubly linked list, where we have two possible directions *next* and *prev*. Another example would be a binary tree with an additional node *parent*.

Secondly, I will consider the situation where two directions can *commute*. Consider a simple mesh of nodes. If we describe the two directions as *north* and *east*, then we clearly reach the

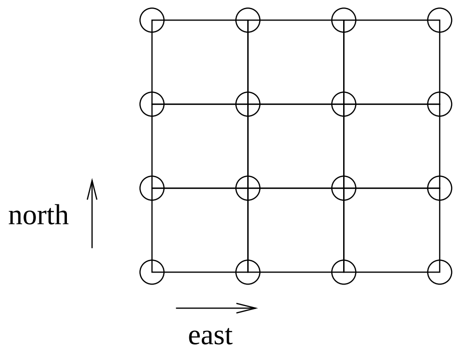

Figure 2.4: The mesh

same point by travelling *north*, *east* that we reach by moving *east* and *north*. So if we want to include these structures in our data types then commutative directions will be needed. Obviously we can extend this to allow commutativity between any pairs of direction that we choose, although this might not create particularly useful structures, apart from the ones for the evaluation of recursive functions mentioned earlier. The mesh or 'Diamond DAG' is, however another example, used in the solution of partial differential equations.

2.2.3 Structures considered

The kind of structures that I will be looking at will consist of the following elements.

- A set of nodes, all storing the same type of data
- A set of directions, grouped into levels, so that only directions in a lower level are accessible from a particular level.
- A collection of relations that hold between directions of a particular level. These include inverses and commutativity between directions.

It is important to emphasise that all nodes will contain the same type of data. This is not really a restriction, since the type of data will be user defined, and we could define an item of data that contains more than one type. For example, we could create a datatype that can store either an integer or real number.

2.3 Example Structures

Here are some examples of the kind of structures that my method will deal with.

Binary Tree

We have two main directions *Left* and *Right*, and we can also make the links bi-directional by having an additional direction *Parent* that is the inverse of both *Left* and *Right*. This makes every node accessible from any part of the structure.

Doubly-linked List of lists

Here we have two separate levels of the hierarchy. At the top is a list structure with direction *NEXT*, and its inverse *PREV*. The next level is conceptually the same with directions *next* and *prev*.

A tree of two dimenstional meshes

In this structure, at the top level we have a tree, with directions *Left* and *Right*. On the next level we have directions *North* and *East*. These higher dimensional structures are particularly suited to the naming scheme.

Call graph of a recursive function

This will consist of nodes corresponding to the values on which the evaluation depends. The directions will correspond to the descent functions, and the commutativity relations will be given by the commutativity conditions.

Chapter 3

Naming the Nodes

3.1 Introduction

In a recursive datatype, we can name each node by the path that was traversed from the root node in order to create it. The name produced will be far from unique if the some of the directions are commutative. Ideally, we want to be able to convert arbitrary paths into a normalised version, this way we can name each node uniquely. We show firstly that such a unique node name exists, and secondly, exhibit an algorithm for determining it and prove its correctness.

3.2 An example

Firstly, we consider an example in order to get a feel for the problem, and to motivate the formality that follows. We return to the example illustrated in fig. 1.1. Here we have 3 directions a, b, c, two of which commute a, b. We require to take any path and convert it to a unique normalised form. Intuitively, we would like to take the left-most path of the many routes that lead to a particular node. Consider wanting to normalise the path $abaacabba$. Clearly, the commutativity requires that we can swap around any two adjacent ab or ba pairs, but in these situations a decision has to be made which one will be considered normalised. To resolve this we put an order on the directions, which for this example reduces to the rule that, wherever possible, as should be placed before bs. In the above path, since neither a nor b commute with c, there are two separate sections, divided by a c. So we can order the as and bs independently within these sections. This leads us to the normalised path $aaabcaabb$.

In a more general case, normalising a whole path at once will not need to be done, as we will be only ever be creating paths incrementally, so we need only consider the problem of adding directions one at a time to an already normalised path. So now consider adding an a to the above normalised version of our example, $aaabcaabb$. Since it commutes with all of the directions past the c in the middle, we could put it anywhere in the second half. We want to place it where it minimises the path name, so we insert it before a symbol it is less than, in this case before the two bs at the end. This results in the path $aaabcaaabb$.

We now want to formalise the intuitive ideas described above.

3.3 The Problem

The directions available at each node are described by a set of symbols S. The commutativity is represented by a (commutative, not necessarily transitive) relation $\sim$ on S. A *path* P is a string of symbols from S, we use $P[k]$ to denote the kth symbol from P.

In the situation where we allow directions to be on l different levels, we essentially have a path made up of l different sub paths. We can then consider the problem of normalising these l paths completely independently of each other. So we need not consider the multi-level problem, as it reduces in each level to a one level problem. So for the rest of the chapter we assume all directions are on the same level.

The set S also has a total order $<$ described on it. We can extend this to a total order on paths by the *lexicographic* extension. This can be defined as in fig 3.1.

3.4 Normalised Paths

Given a path P we can manipulate it using the commutativity relation. Any consecutive symbols a and b such that $a \sim b$ can be swapped, to produce another path. The set of paths

$$P_1 = x_1, x_2, x_3, \ldots x_m \qquad P_2 = y_1, y_2, y_3, \ldots y_n$$

$$P_1 < P_2 \Leftrightarrow$$

$$\exists c \geq 0 \qquad \text{s.t.} \qquad x_i = y_i \qquad \forall i \leq c$$

$$\text{and} \qquad (c = m \qquad \text{or} \qquad x_{c+1} < y_{c+1})$$

Figure 3.1: Lexicographic ordering on paths

produced by all such manipulations of P is the set of *aliases for* P. Using the ordering on the paths in the alias set we can choose the unique smallest member. This is the *normalised* path for P. The remaining problem is then to find an efficient algorithm for finding the normalised version of any path.

3.4.1 The Algorithm

Since the paths are built up a symbol at a time we merely require a way of adding a new symbol to a normalised path, and manipulating to produce a normalised version.

Suppose we have a path

$$P_1 = x_1, x_2, x_3 \ldots x_m$$

and we want to add to it a new symbol b. The algorithm proceeds as in fig. 3.2

This can clearly be extended in the case where a path P has not been built up incrementally, we merely start with an empty path and add symbols from P to it using the above algorithm.

The algorithm is $\Omega(m)$, where m is the length of the path. This will represent an overhead in the final implementation whenever a node name is generated. This overhead will depend on the *depth* of the data structure, rather than the number of nodes in it.

$$c = m$$

$$\text{while}(c > 0 \quad \text{and} \quad b \sim x_c \quad \{c \to c - 1\}$$

$$c \to c + 1$$

$$\text{while}(c < m + 1 \quad \text{and} \quad b > x_c \quad \{c \to c + 1\}$$

$$\text{Insert } b \text{ before } x_c$$

Figure 3.2: Add Symbol Algorithm

3.5 Correctness Proof

It remains to prove that the above method produces a normalised path as its output, given that the path P_1 was normalised. The proof is basically a proof by contradiction; we show that if the output path of the algorithm is not normalised, then the input path must also have not been normalised.

Given that the input is

$$P_1 = x_1, x_2, x_3 \ldots x_m$$

The output of the algorithm is a path P_2, with the properties as shown in fig. 3.3

Any alias for P_2 is an alias for P_1 with b inserted at some point. So suppose we have a P_3, an alias for P_1

$$P_3 = y_1 \ldots y_m$$

We must have $P_3 \geq P_1$, as P_1 is normalised. If $P_3 = P_1$, then we clearly cannot insert b at a valid point in order to decrease P_3, so assume from now on that $P_3 > P_1$. This gives us:-

$$P_2 = x_1, \ldots x_p, \ldots x_q, b, x_{q+1}, \ldots x_m$$

$$b \sim x_i \quad (p < i \leq m)$$

$$b \not\sim x_p$$

$$b > x_i \qquad (p < i \leq q)$$

$$b < x_{q+1}$$

Figure 3.3: Properties of P_2

$$P_1 = x_1, x_2, x_3, \ldots x_m \quad P_3 = y_1, y_2, y_3, \ldots y_m$$

$$\exists c \geq 0 \qquad \text{s.t.} \qquad x_i = y_i \qquad \forall i \leq c$$

$$\text{and} \qquad (x_{c+1} < y_{c+1})$$

Now suppose P_3' is P_3 with b inserted at some point:

$$P_3' = y_1 \ldots y_k, b, y_{k+1} \ldots y_m$$

There are now several cases to consider depending on the relative sizes of (c, k, q). For each case we aim to show $P_3' > P_2$, and conclude that P_2 must be normalised.

$c < min(k, q)$

In this situation we have

$$P_3'[i] = P_2[i] \quad (i \leq c)$$

$$P_3'[c+1] = y_{c+1} > x_{c+1} = P_2[c+1]$$

So $P_3' > P_2$

$c \geq min(k, q)$ **and** $k < q$

Here the two paths look like:-

$$P_3' = y_1, \ldots y_k, b, y_{k+1} \ldots y_m$$

$$P_2 = x_1, \ldots x_p, \ldots x_q, b, x_{q+1}, \ldots x_m$$

What is important is where p is in relation to k. There are two sub cases.

$p \geq k + 1$

Since $b \not\succ x_p$, x_p must be somewhere in the set $\{y_1 \ldots y_k\}$. But this is equal to $\{x_1 \ldots x_k\}$, from the condition on c. From $p \geq k+1$, x_p cannot be in this set. So this case is not possible, the path P_3' does not correspond to an alias for P_2.

$p < k + 1$

Then, from the conditions on P_2 (see page 21), $b > x_{k+1}$. This gives us that $P_3' > P_2$.

$c \geq min(k,q)$ **and** $q < k$

Here the paths look like:

$$P_3' = y_1, \ldots y_q, y_{q+1} \ldots y_k, b, y_{k+1} \ldots y_m$$

$$P_2 = x_1, \ldots x_q, b, x_{q+1}, \ldots x_m$$

From the conditions on P_2, $b < x_{q+1}$. But $x_{q+1} = y_{q+1}$ from the condition on c, so $b < y_{q+1}$, giving that $P_3' > P_2$.

$q = k$

Here the two paths will agree up to the point c. The next elements from the lists P_3', P_2 will be y_{c+1} and x_{c+1}. The fact that $y_{c+1} > x_{c+1}$, gives that $P_3' > P_2$.

Chapter 4

Mapping Nodes to Processors

4.1 Introduction

We now turn to the problem of mapping the data nodes that we have named to the actual physical processors. Naturally this needs to be done in a manner that is transparent to the user, and allows the data structure to be manipulated in an efficient way. It is important to emphasise that in a situation where we have a dynamically growing and shrinking structure, the best solution is for a dynamic strategy that re-allocates nodes as the structure changes. I have not considered this problem, but have settled instead for a static allocation, which should provide a reasonable allocation for a balanced, static structure.

The issues that have motivated the allocation strategy are as follows.

- Locality: Data items that are near to each other in the structure need to be placed in nearby processors.
- Fair Division: As each data item corresponds to an amount of computation, to divide the work equally the data needs to be spread out amongst the processors as fairly as possible.
- Parallelism: The distinct sub parts of the structure need to be mapped to distinct sets of processors to encourage as much potential parallelism as possible.

Note that the idea here is to map the nodes to processors in a way that is completely known at the outset, and will not change as the structure changes dynamically. This means that we cannot use any information about the size of the structure, or how balanced it is. Ideally, we would also want to add a mechanism to re-distribute the nodes once certain conditions relating to how unbalanced the structure has become, but this problem has not been considered.

4.2 The Issues

4.2.1 Locality

This issue requires some knowledge of the target machine, as far as processor layout is concerned. In order to simplify the situation, we assume that all communications between processors take the same time, in other words that no two processors are nearer each other than any others. This needs to be justified. On the Cray T3D, as compared to a distributed system such as a network of workstations, communications are relatively fast, so the differences between different sets of processors are small. In fact, the Cray has hardware support for a single address space, and this is used to implement the message passing mechanism that I will be using for the implementation. This implies that messages from a processor to itself are certainly quicker than from distinct processors, but that the time cost for other pairs of processors will be constant. So, we can still exploit locality, as we can allocate nodes to the same processor (no communication required) or to different processors (one unit of communication).

4.2.2 Fair Division

What the user intends to do with the structure can influence what is actually meant by a fair division of computation. Since we cannot predict this we assume that the bulk of the operations will be traversals around the structure, with occasional adding and modification of data values. In the situation where we merely want to map a function over our data structure, dividing the data out equally between the processors is more important than the locality of the parts of the structure. It becomes more important when the function (or amount of work to be done) increases, and dominates the actual overhead of traversing the tree.

4.2.3 Parallelism

This issue is important as the user of the library of routines is writing a SPMD program and will primarily be using recursive functions to manipulate the data structures. This implies that all processors will have to recurse through all parts of the structure, whether they actually own data nodes in that part or not. Therefore we want to map the nodes to sets of processors in such a way that a test can be inserted into the code that can spot when a processor does not own any node in a sub structure. This test can be used to kill off unnecessary traversal of the data structure, and promote some degree of parallelisation.

4.3 The Mapping System

The mapping function uses the path name of the node. This is a string of directions that we can divide up into the hierarchical levels of the description. The method begins with the full set of processors then takes each direction from the path in turn, reducing the set of processors according to the relations that that direction has with the other directions in that level. If we are left with a single processor, the process terminates and the node is mapped to that processor. Otherwise, the process end when we run out of directions in the path, in which case the first processor in the remaining set is allocated the node.

As mentioned previously, in order to extract as much parallelism from the user functions as possible, we require a function that can check whether any of the nodes accessible from a particular node are allocated to a particular processor. This is similar to the allocation algorithm, but we always want to return a conservative estimate, since it is better to be inefficient than incorrect.

4.4 The allocation algorithm

This looks through the path, a direction at a time, and considers the relations the direction shares with the other directions in the relevant level. There are different situations to cater for, depending on the structure of that part of the data type, which can locally be considered tree-like, list-like or with commuting directions.

4.4.1 Tree-like

Consider a basic binary tree structure, where at each level there are at most two sub trees, and we have numbered the processors $[1 \ldots n]$. Then we can map the left sub tree to the set $[1 \ldots \frac{n}{2})$ and the right sub tree to the set $[\frac{n}{2} \ldots n]$. This idea extends naturally to the situation where we have a tree with at most k sub trees, and maps distinct sub trees to distinct sets of processors.

4.5 List-like

Here the data nodes are arranged in a linear list, with no distinct sub sections that we can exploit by arranging them on different processors. Instead, assuming p processors, we allocate the nth element of the list to processor $n \bmod p$. This does not unfortunately exploit any locality in the list. Ideally, we would want to know the length of the list and to divide it into p consecutive chunks, and distribute one to each processor. We cannot do this as the list is being created because there is no way of knowing the size of the list in advance.

4.6 With commuting directions

Here again, there is no obvious way to partition the data structure so that the above issues are all satisfied. Here we settle for distributing the nodes as it were a list in one of the commuting directions.

4.7 Example

We return to our call graph example to describe how the method will allocate the nodes. Consider fig. 4.1, where we allocate the call graph to three processors. The algorithm spots that the right hand sub tree, where all the paths begin with c, is completely independant and allocates it on the last third of the processors, in this case solely on processor 1. The remaining part has commutative directions so is allocated to the first two-thirds of the processors, namely processors 2 and 3. Within that, the allocation is done as if it were a list, paths with one a

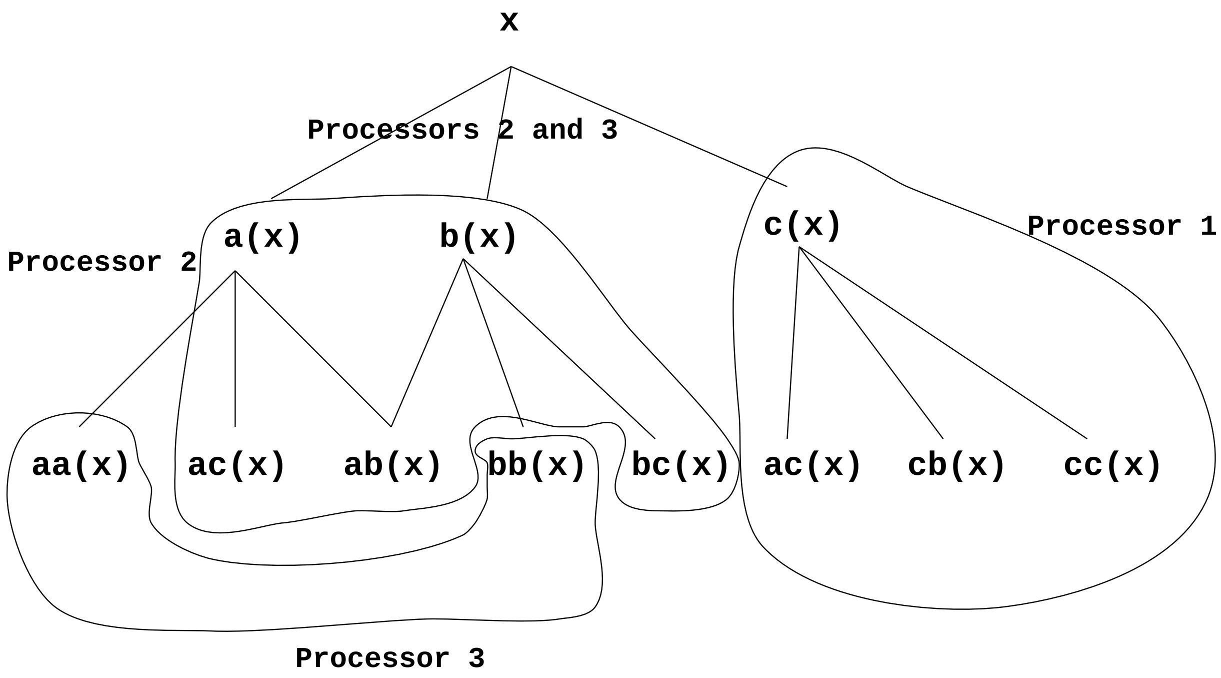

Figure 4.1: Allocation of nodes in the call graph

go onto processor 2, those with two go on processor 3. Similary for one and two bs. Clearly, from looking at the diagram, there should be a better mapping of the nodes to processors, as in fig. 4.2. This is the sort of distribution that we would want from a dynamic allocation scheme.

4.8 Pseudocode for mapping function

When we are mapping nodes to processors we work with *(* ranges) of processors at each step, denoted $[a, b]$. So we begin with processors $[1, n]$, where n is the total number of processors involved in the computation. We assume that we have a subdividing function *Subsection(r, p, q)* that divides r into q subsections and returns the pth. We also need another function *Lastsection(r, p, q)*, that divides r into q sub sections and returns the sections $p+1 \ldots q$ as one combined range. We assume that the directions in each level are numbered so that the directions that do not commute with any of the others are placed first. We take a path p and calculate which processor to allocate it to.

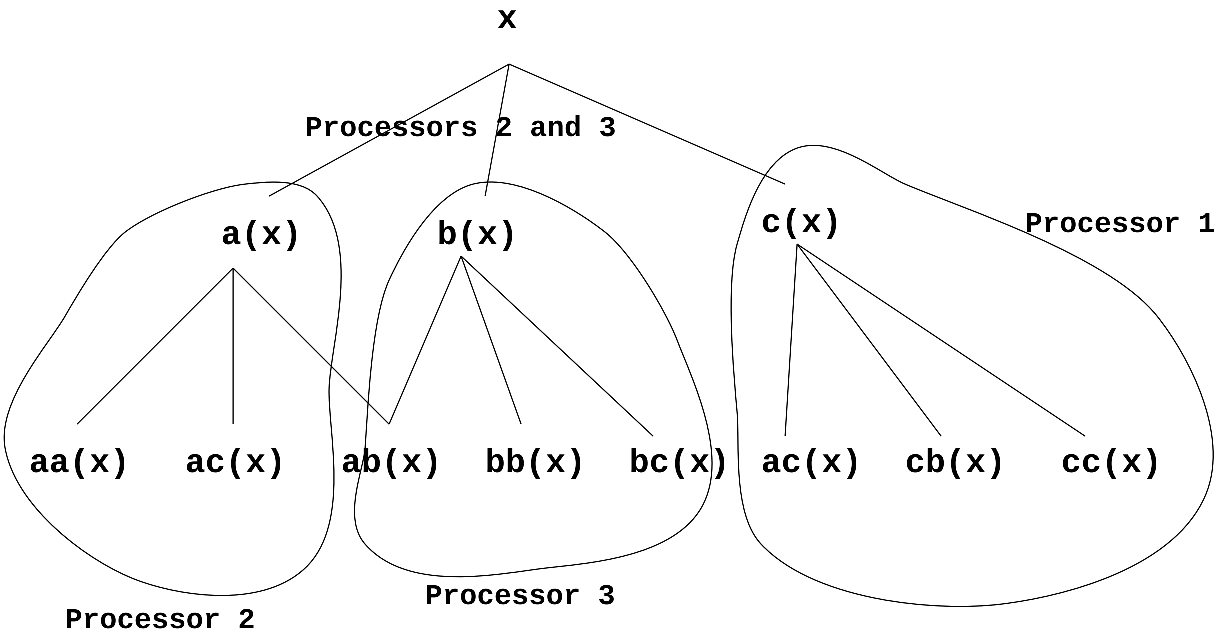

Figure 4.2: A better allocation of nodes for the call graph

```
Range r = [1, n]
i = first direction in p
while((i is a valid element) and (r contains more than one element))
{
 currdegree = number of directions in level
 if (currdegree = 1) then map list over r
 if (i commutes with another) {
   treat as a list with direction i,
   and set of processors Lastsection{r, num of commuting directions, currdegree}
 }
 else subdivide:-  r = Subsection(r, i, currdegree)
 i -> next direction in p
}
return minimum of r
```

Figure 4.3: Node mapping function

```
Range r = [1, n]
i = first direction in p
while((i is a valid element) and (r contains more than one element))
{
 currdegree = number of directions in level
 if (currdegree = 1) then return r
 if (i commutes with another) {
   return Lastsection{r, num of commuting directions, currdegree}
 }
 else subdivide:-   r = Subsection(r, i, currdegree)
 i -> next direction in p
}
return r
```

Figure 4.4: 'Estimate Range' function

4.9 Pseudocode for 'Estimate Range' function

This is the function that a processor can call with a path, and returns all the processors that own nodes below that node in the structure. This can then allow a processor to drop out of a part or of the computation if it owns none of the nodes in that sub section. It works much the same as the processor mapping algorithm.

Chapter 5

Implementation on the Cray T3D

5.1 Introduction

This chapter describes my implementation, and the user interface it presents to a programmer wanting to use the system to perform computations on a recursive data structure. The implementation is written in C++, with use of the Message Passing Interface (MPI) library of routines for communications between the processors.

At the topmost level the user will describe the structure of the datatype, write a sequential program using the datatype and compile with the library code to produce a program that will implement the code in parallel.

A datatype consists of a set of nodes, each containing an item of type *Data* and links, or branches to other nodes. The nodes are described using *Path*s. The structure of the datatype is described in an *Algebra* object. The main *Dtype* structure handles all the functions that the user can apply to the data structure.

5.2 The Implementation

The nodes in the data structure are named using *Path*s, a doubly-linked list of integers that represent the branches. Each processor stores a hash table linking the paths to the global pointers that describe to which processor a node belongs.

5.3 The user interface to the library

5.3.1 The *Data* class

The *Data* class is supplied by the user to hold the actual data that is to be stored in the nodes. Arithmetical operations on *Data* are expected to be supplied, as are two functions that perform sending and receiving of the *Data* in an MPI environment. Ideally, it should be possible to create the *Data* class without any knowledge of MPI. This would still require the user to supply the functions to pack and unpack the data that is to be sent as a message. For simplicity, this was not implemented, but to make to library easrier to use, it would be a valid addition.

5.3.2 Describing the structure

First the user creates an *Algebra* that contains information about the algebraic relations between the various directions. Directions are described as integers, starting from zero.

This is done by firstly adding the directions, or *branches* to a particular level of the description *Add(branch, level)*. If any directions have inverses, these are then described *AddInv(branch)*. If a pair of direction commute, this is described using the *AddComm(pair of branches)*. As an example a 'tree of 2D meshes' would be created using the code in fig. 5.1

5.3.3 The main *Dtype* Structure

This is created by passing the *Dtype* object to the *Algebra* that describes the structure. The user can then perform various operations on the structure. The *Path* objects are used to refer to actual nodes in the structure, and nodes are added, modified and described using *Path*s. The functions that are provided for the user are listed in fig. 5.3. We will now look at some of the functions in greater detail.

```
#define Left 0
#define Right 1
#define North 2
#define South 3
#define West 4
#define East 5

Algebra treeofmeshes;

treeofmeshes.Add(Left,0);
treeofmeshes.Add(Right,0);
treeofmeshes.Add(North,1);
treeofmeshes.Add(East,1);

treeofmeshes.AddInv(BranchPair(North,South));
treeofmeshes.AddInv(BranchPair(East,West));
treeofmeshes.AddComm(BranchPair(North, East));
```

Figure 5.1: User code for a tree of meshes structure

NotInvolved

As the programming model is SPMD, if we are traversing a large structure, we want to discourage processors from needlessly moving through parts of the structure where they own no nodes. This function can be inserted into code and used to implement this. As each processor can only affect nodes that they own, this cannot change the computation, and should increase the parallelism. As example of the user code that is required, consider fig. 5.2, a function to map a function over a binary tree. The upper code fragment is the sequential version, the lower the additions required to make it execute using the library functions. There is quite a close correspondance between the two. The test for a null pointer in the sequential is replaced by a call to check if the node exists in the parallel version, and the traversal of the sub-trees is handled in a similar way. The *NotInvolved* function is inserted in the body of the function, and causes the processor to return, and not spawn any more recursive calls.

The *Read* function

As the implementation is SPMD, calling this function will return to every processor in the computation, the value at the required node. This requires a global syncronisation and then

```
void
Map(Tree *t, Data fn(Data)) {

  if (!t) {return;}

  t->value = fn(t->value);

  Tree *left, *right;
  left = t->left;
  right = t-> right;

  Map(left, fn);
  Map(right, fn);
}
```

```
void
Map(Dtype &dt, Path *p, Data fn(Data) ) {

  if (dt.NotInvolved(p)) {return;}
  if (!dt.Exists(p)) {return;}

  dt.Update(p,fn);

  Path *l, *r;
  l = dt.Subnode(p,Left);
  r = dt.Subnode(p,Right);

  Map(dt,l,fn);
  Map(dt,r,fn);
}
```

Figure 5.2: User code for mapping over a binary tree

```
Dtype(Algebra algebra);  //Constructor, creates a Dtype with the Algebra
Bool  Me(ProcId);        //Am I this processor?
Bool NotInvolved(Path *p); //Do I not own nodes in this substructure?
Bool  Exists(Path *p);    //Does this node exist?
void  Root(Data d);    //Set the root data value
Path*  NewPath();      //Create a new path
Path*  CopyPath(Path *p); //Make a copy of any existing path
void  Add(Path *p, int dir, Data d); //Add a node to the structure
void  Traverse(Path *p, int dir); //Move through the structure
Path*  Subnode(Path *p, int direction); //Create a path as a sub node
void  Update(Path *p, Data fn(Data));   //Apply function to a node
void  Write(Path *p, Data value);   //Write a value to a node
void  CopyValue(Path *from, Path *to); //Copy a value from node to node
Data  Read(Path *p); //Gives all processors the value in a node
void  Combine(Path *p1, Path *p2,
              Data binop(Data,Data),
              Path *result, Data comb(Data,Data) );
              //Writes comb(result, binop(p1,p2)) into node result
Bool  ImWriter(); //Am I processor 1?
void  PrintTable(ProcId proc); //Display node distribution statistics
void  Display(Path *p); //Outputs node p
```

Figure 5.3: Dtype functions available to user

a broadcast of the data value from the processor that owns it to all the other processors. This will naturally impair the parallel performance of the system. Avoiding the use of the *Read* function is then important in any program that aims to maximise performance. However, many algorithms require this sort of global information between the various phases of computation.

The *Combine* function

The main aim of the *Combine* function is to allow computations with node values to take place without having to perform expensive *Read*s and *Write*s. Only the processors that own the data at nodes $p1$, $p2$ and *result* will take part in the computation and be involved in the communication required to compute the result.

```
Dist a(dt,p), b(dt,q), c(dt,r);
b = p;
c = r;
a = b + c;
```

Figure 5.4: Use of dist object

5.4 The *Dist* class

This class is used to create single, temporary, distributed items of *Data*. These have been created as solution to the following problem. If the user wants to form a computation using the nodes in the structure, it will be necessary to *Read* the data values, perform the computation, and then *Write* the values back into the structure. As the program is run in a SPMD fashion, this will result in *all* the processors reading, and computing with the data. This is obviously wasteful, and the read function also requires a global broadcast of data, which all processors much participate in. One solution would be to have combined operations that read values, compute, then write updated values back. Only the processors directly involved with the relevant parts of the structure are then required to take part. As has been shown the function *Combine* is provided, which partly gets around the problem, and can be used when efficiency is important. The disadvantage of this is that these combined functions are cumbersome to use, so we introduce the idea of a distributed temporary data item, which allows computations to be expressed in a much cleaner way.

Each *Dist* object is a single item of data, located on one of the processors. When the user creates the object, a node in the data structure is specified, and the data will be placed on the processor that owns that particular node. This enables the user to write code as in fig. 5.4.

Here p, q, r are all nodes of data structure dt. We first initialise b and c with values from dt, and then form a sum. The assignment and addition operators are overloaded in such a way that only the processors that store the arguments, actually carry out the computation. This allows parallelism in the code to be extracted, whilst making the user code clearer.

Chapter 6

Analysis of Method

6.1 Introduction

In this chapter, it is firstly demonstrated that the evaluation of functions with commutative recursive calls is a problem with a sufficient amount of redundancy to make its exploitation using my method worthwhile. Secondly, for a given problem and number of processors, we can compute how much of the redundancy is worth exploiting to achieve the best performance.

6.2 Redundancy in a Data type

It was mentioned previously that a data structure with commuting sub directions will exhibit some redundancy, in that we can predict in advance which nodes are to be shared, and avoid repeating computation. Ideally, we want to predict how much computation can be saved by this system. Consider a tree of degree 3, where two of the directions commute. If we consider the first two levels of the graphs (fig. 6.1), we have one shared node on the second level. If we then extend the structure below this level, the whole of that sub tree will be shared, so a least $\frac{1}{9}$ of the whole structure will be shared. A similar argument shows that for 4 directions, two of which commute, at least $\frac{1}{16}$ of the structure will be shared. These figures seem rather small, but in fact the amount of redundancy greatly increases as we consider deeper trees.

The implementation was used to calculate the number of nodes. It was found that for an

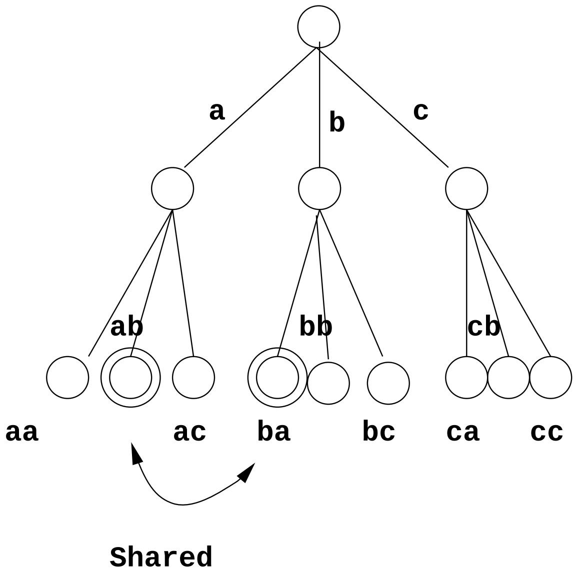

Figure 6.1: Node sharing

unbalanced tree of depth 8, with the above commutation properties, that only 1701 nodes out of a possible 2069 are distinct. This suggests that for this example, which is not balanced, that the node sharing figure is closer to one fifth. We should therefore be able to calculate a better estimate, if only for a *balanced* structure with the required commutation properties.

In general, consider a situation where we have a set of directions D, with commutation conditions on some of them, and an ordering on them all. Then each direction d will have a set P_d of directions that can legitimately precede it in a normalised path name. Then if $n_{l,d}$ is the number of paths of length l that conclude with direction d, then we can express $n_{l,d}$ as $\sum\limits_{x \in P_d} n_{l,x}$.

This will lead to a set of recurrence relations for the number of directions in each level of the graph. For our example outlined above, this gives us a total, for level l of:-

$$\frac{(2+\sqrt{3})^{l+1}}{2\sqrt{3}} - \frac{(2-\sqrt{3})^{l+1}}{2\sqrt{3}}$$

If the call graph is balanced, so that it is the same depth n at all points, we can compute the approximate number of nodes in it, simply by ignoring the small $(2-\sqrt{3})$ term and summing from 1 to depth n. This gives an approximate value of:-

$$\frac{1}{2\sqrt{3}} \frac{(2+\sqrt{3})^{n+2}}{(1+\sqrt{3})}$$

We can compare this with the number of nodes in a graph with no commutativity, a figure of $\frac{4^{n+1}}{3}$. Thus the ratio of nodes in the non commuting structure, to the nodes in the commuting structure is:-

$$\frac{\text{Non commuting}}{\text{Commuting}} = c\left(\frac{4}{2+\sqrt{3}}\right)^{n+1}$$

Where $c = \frac{2}{\sqrt{3}}\frac{1+\sqrt{3}}{2+\sqrt{3}}$, or approximately 0.8453. The above estimate, shows that as n increases the proportion of computation we can save by exploiting some of the commutativity, increases also. In fact this proportion is unbounded as we increase the depth of the tree. This will be the case in other examples. So exploiting as much commutativity as possible is worthwhile, especially for deep function call graphs.

6.3 Predicting the amount of parallelism in the evaluation

For the recursive function evaluation system, it can be predicted in advance how much useful parallelism can be obtained from a given problem. By 'useful parallelism' I mean the point at which adding more processors and splitting the problem up amongst them using my method, does not result in any performance improvement. First we look at an example to explain the reasoning. Consider the example where we have 6 sub functions, 2 of which commute, see fig. 6.2. The directions that commute create a part of the structure that, if split up amongst many processors, require much communication to evaluate, and dominate the time taken for the whole computation. Also, since the processors involved in this part of the tree will both have to traverse the same part of the structure, very little of the computation is actually being shared between them. So, if the parallelisation requires splitting up this sub section we will not have any benefit from using the extra processors. As we increase the number of processors to which we map the sub trees from 3 to 6, there is be a point at which the two commuting sub trees are split.

Consider a recursive function that is defined in terms of k sub functions. This leads to a call graph of degree k. Now suppose that r of the functions commute with at least one other function, so that $k - r$ of the sub trees are distinct. These distinct sub trees can be exploited to provide the parallelism. So if we are mapping to p processors, if $p > \frac{k}{r}$, then the commuting directions will be split and we cannot expect any improvement in the performance. Thus,

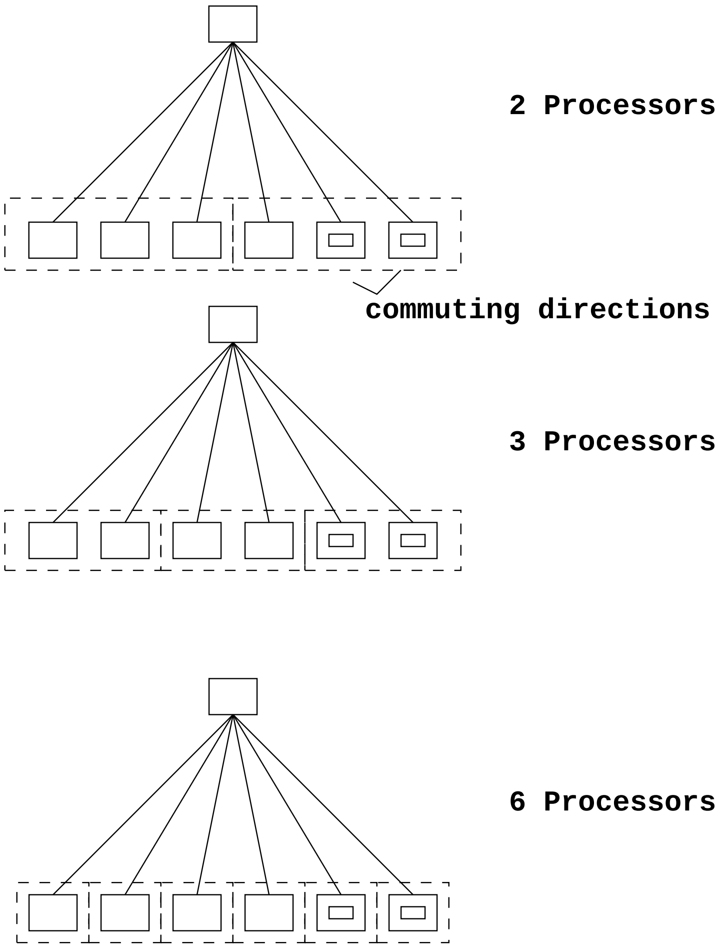

Figure 6.2: Dividing out the sub trees

with a function for which k and r are defined as above, we will recieve no more speedup if we run the evaluation on more than $\frac{k}{r}$ processors.

The method is therefore more suited for functions where there are many sub functions, few of which commute. However, specifying functions that do commute as not commuting will not change the correctness of the evaluation, and will only decrease the amount of redundancy that can be exploited. We can artificially make the number of directions that are reported as commuting as small as we choose, and thus increase the potential for parallelism. This can be exploited in the situation where we have a fixed number of processors p available to us, and an evaluation call graph for which we know the values of k and r. We then find the minimum r such that $\frac{k}{r}$ is not greater than p. We then use that level of commutativity to evaluate the tree. This will provide the optimal performance.

Chapter 7

Experimental Results

7.1 The Example

Since the method is geared towards the evaluation of recursive functions that have some redundancy, the example that I have chosen is of this form. It is a function f with four sub functions, a, b, c, d, two of which commute (a, b), see fig. 7.1. We can then compare the run-times when exploiting the commutativity with the results obtained when we do not.

```
f(x) = if (x=0 or x=1) then 1
       else f(a(x)) + f(b(x)) + f(c(x)) + f(d(x))

a(x) = x - 2

b(x) = x - 1

c(x) = if (x is even) then x - 2
       else x - 1

d(x) = if (x is even) then x - 1
       else x - 2
```

Figure 7.1: The example function $f(x)$

7.1.1 Running the implementation

The implementation of this function was run on 1, 2 and 4 processors, the times for the computation of $f(8)$, a tree of depth at most 8, containing 1701 nodes. Fig. 7.2 shows the time taken for the evaluation to run, the error bars representing the range in time taken for the various processors involved with the computation. Naturally, the run time of a set of parallel processes is taken to be the time for the longest process to complete, given that they all begin at the same time. The two processor version therefore runs in 60% of the time for the one processor computation, a significant speedup. When we run using four processors, however, the total time increases drastically. If we apply the analysis developed earlier to this particular example, we have $k = 4$ and $r = 2$, so we can expect no more that two-way parallelism. So, as predicted, we can achieve no more than two way parallelism with this particular example.

7.1.2 Running without exploiting commutativity

The above example was run, but without specifying the functions a, b as commuting. This does not affect the value of the computation of $f(8)$, but it does involve more computation, a 2069 node call graph this time. This is potentially 22% more computation than before.

The advantage now is that there is now more parallelism in the evaluation, and therefore, if we run on more processors we can achieve a better speed up, and eventually will improve on the run time. I have also plotted the run-times for the one and two processor versions that exploit the redundancy, see fig. 7.3. This gives a cut off point at around 6 processors, assuming that we could run it on that number. This gives the result that by exploiting redundancy we can run the evaluation faster on 2 processors than on 4 processors without. Unfortunately, we cannot improve on this as we are limited by the parallelism in the function.

We return briefly to the analysis included earlier to see if the predictions for the size of these graph is borne out in practice. Recall the formula on page 39. For $n = 8$, this gives a value of 1.58. If we compare this with the experimental result for this example of 1.23. The reason for the slight discrepancy is the fact that the example tree is not balanced, indeed at some points it is of depth 4, and at others depth 8. The formula implies that the deeper parts of

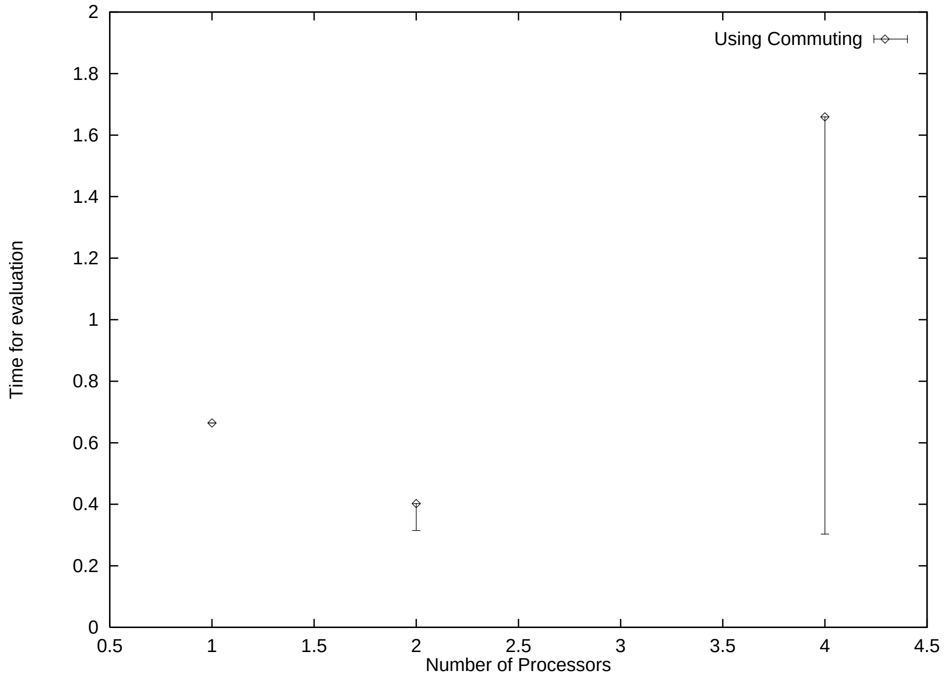

Figure 7.2: Exploiting commutativity with different numbers of processors

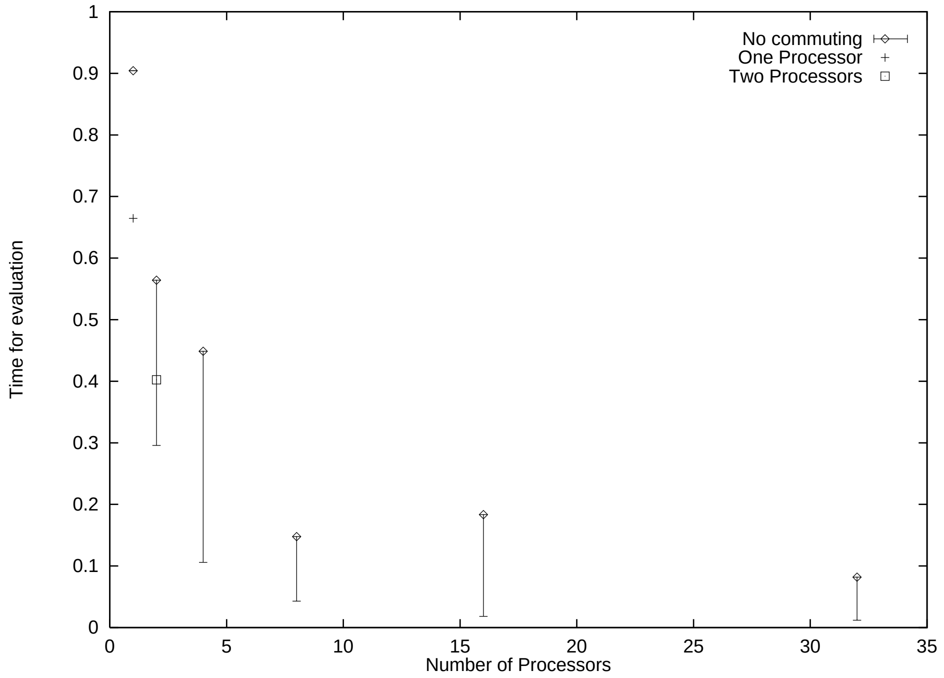

Figure 7.3: Comparison of exploiting / not exploiting commutativity

the tree will have a greater proportion of redundancy, so the actual value is lower.

Chapter 8

Conclusions

I have extended the node-naming strategy to allow relations between the directions to be included, so that the redundancy in certain recursive functions could be exploited. This required a full path name to be stored for each node, and an algorithm for normalising such a path name was found and proved correct. In order to implement such a scheme, a mapping function from node names to processors was required. A simple static mapping system was used, which can be thought of as first pass allocation system, which would ideally be improved at run-time with a dynamic allocation scheme. The system was analysed so that for a given recursive function and number of processors, a specific amount of the redundancy could be exploited to provide an efficient implementation in parallel. This was backed up by experiments performed on the Cray T3D.

The extension of the mapping scheme also allowed for a class of recursively defined data types to be automatically named, from a simple description supplied by the user. This meant that the node naming scheme could be used without the user explicitly supplying the relevant naming functions. This was used to create a set of library functions that allow a user to manipulate nested recursive structures such as trees and trees of lists in parallel, without explicitly using parallel constructs.

8.1 Further work

The weak link in my system was the static mapping function. It could not use information about which nodes existed in the structure to divide the nodes up more equally when the structures were not balanced. Also, a better mapping function could have lead to a system where more parallelism could have been extracted for the specific problem of the recursive function evaluation.

Bibliography

[1] *Mechanical Elimination of Commutative Redundancy* H. Khoshnevisan and M. Afshar, First International Static Analysis Symposium, 1994, p 449.

[2] *Supporting SPMD Execution for Dynamic Data Structures* A.Rogers, J.Reppy, L.Hendren, Languages and Compilers for Parallel Computing, LNCS 757

[3] *SPMD Execution of Programs with Pointer-based Data Structures on Distributed Memory Machines* R.Gupta, Journal of Parallel and Distributed Computing, Vol 16, 1992, pp. 92-107

[4] *Early Experiences with Olden* C.Carlisle, A.Rogers, J.Reppy, L.Hendren, Languages and Compilers for Parallel Computing, 6th International Workshop, 1993, p 1.

[5] *Analysis of Pointers and Structures* D. Chase, M. Wegman, F. Zadeck SIGPLAN Notices Vol. 26 No. 6 p 296

[6] *Parallelising Programs with Recursive Data Structures* L.Hendren, A.Nicolau University of California Tech Report No. 89-33

Appendix A

Code Listings