PARALLELIZING PROGRAMS WITH RECURSIVE DATA STRUCTURES

Laurie J. Hendren

Alexandru Nicolau

Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No.89-33

Parallelizing Programs with Recursive Data Structures *

Laurie J. Hendren
Dept. of Computer Science
Upson Hall
Cornell University
Ithaca, NY
14853

(607) 255-5033
hendren@cornell.edu

Alexandru Nicolau
Dept. of Information and
Computer Science
University of California - Irvine
Irvine, CA
92717

(714) 856-4079
nicolau@uci.edu

Abstract

In this paper, we present a novel method for parallelizing imperative programs in the presence of dynamic recursive data-structures. At the heart of parallelizing compilers are the dependence-analysis and disambiguation mechanisms. We present a three-pronged approach: (1) we augment an imperative language with easily parallelizable recursive data-structures, (2) we develop tools for disambugation and interference analysis for such structures, and (3) we present three methods for using information from the analysis to parallelize programs. We illustrate these techniques with a concrete example that has been processed by our system.

KEYWORDS: parallelizing, interference analysis, recursive data-structures

*This work was supported in part by NSF grant CCR 87-04367 and ONR grant N00014-86-K-0215

1 Introduction

Parallelizing and vectorizing compilers have made significant progress in the extraction of parallelism from ordinary sequential programs. The primary tool at the core of such parallelizing compilers is the dependence-analysis and disambiguation (interference analysis) mechanism. The ability to accurately disambiguate indirect memory references is critical in determining precise data-dependencies between operations, and thus to the correctness and applicability of any parallelizing transformations. While this sort of analysis is absolutely essential for imperative languages, it is also necessary for the efficient implementation of functional languages on existing machines, where the avoidance of excessive copying of large data-structures is crucial.

In the context of relatively simple languages with scalar and array data-structures, compile-time disambiguation techniques [Ban79a,Wol82,Nic84] have proved to be very successful in eliminating spurious aliases, thereby increasing the potential parallelism exposed by the compiler. Disambiguation techniques are currently used by all parallelizing and vectorizing compilers. However, current disambiguation techniques cannot, by themselves, deal well with complex dereferencing patterns. Such patterns are not only encountered in situations where complex data-structures and pointers are used, but also in many numerical applications involving recursive (and possibly overlapping) partitioning of matrices. Furthermore, many of these programs modify the data-structures in non-trivial ways, resulting in wide variations in the potential for parallelism at different points in the program execution. In such situations, the information derivable automatically by the techniques now in use is too weak to allow satisfactory parallelization of the code, particularly when attempting to exploit the full potential of large scale parallel machines.

The above situation is only partially due to weaknesses of the disambiguation mechanisms of parallelizing compilers. In restructuring programs for parallel execution, or in designing new parallel algorithms from scratch, the programmer has at her disposal large amounts of information about the original algorithm and the problem it attempts to solve, as well as knowledge about the structure of the data. Even when the user is coding in a parallel programming language, much of the meta-information that could facilitate parallelization of the program is lost in the coding process. Once lost, this information is often irretrievable from the actual code, so that a compiler, no matter how sophisticated, can not recover it.

We propose an approach designed to facilitate the expression and detection of parallelism in programs with complex data-structures. At the language level, we have developed new data types which allow the natural expression of parallelism explicit in recursive data-structures. For example, we provide a natural, recursive definition of trees, as well as various recursive partitionings of matrices and arrays. The key advantage of providing such data types is that they communicate to the compiler important properties of the structures that are relevant in detecting parallelism. For example, the knowledge that a structure *is* a tree implies that the children of a node are the roots of *distinct* subtrees.[1]

While a necessary starting point for effective and efficient disambiguation, providing such recursive data types is not sufficient. The program may modify a data-structure quite extensively; either explicitly in imperative languages, or as a desirable compiler optimization in functional languages—to avoid copying. For example, a tree may be changed temporarily into a DAG, as an intermediate step in swapping some nodes. Thus, to detect potential parallelism, an analysis tool should also be capable of checking the preservation (or violation) of critical properties of the data-structure across various parts of the program.

Another way the analysis tools presented in this paper may be used is in debugging parallel programs. By checking explicit parallel and synchronization constructs against data-structure specifications and manipulation, the system could detect inconsistencies and non-deterministic behavior that may be otherwise very hard to detect for a human user.

We have designed and implemented both recursive data types and associated analysis tools within the framework of a small imperative language, SIL. We have also developed a parallelization tool that uses the information derived by the analysis to determine (at compile-time) what parts of the code could be safely executed in parallel.

In this paper we describe the three components of our prototype system: (1) the definition of recursive data types, (2) tools for interference and structural verification analysis, and (3) methods for using the results of the analysis to parallelize programs. In the section on parallelization, we present concrete examples illustrating the effectiveness of our approach. While this paper examines only the case of trees and DAGs, similar techniques apply to recursively partitioned matrices and arrays.

[1]Note that this information would be irretrievably lost in a program written in languages such as C, Pascal, Lisp, where no means exist for differentiating a tree from a linked-list, DAG, or cyclic graph.

2 Related Work

Various approaches have been suggested for interference analysis in the presence of dynamic data structures and pointers.

Lucassen and Gifford have proposed an approach whereby a programming language is defined that encorporates an effect system as well as a type system [Luc87,LG88]. The *effect* of a computation is a summary of the observable side-effects of the computation. Although the effect system effectively differentiates between totally disjoint linked structures, their method does not provide a way of distinguishing between different parts of a large data structure. For example, even though the left and right sub-trees of a binary tree do not share any storage, the effect system forces both sub-trees to be associated with the same region. This lack of fine-grain information based on the recursive structure of the object results in an overly conservative interference analysis.

Another approach to static analysis for dynamic structures has been proposed by Neirynck [Nei88]. This method uses abstract interpretation techniques to provide information about aliasing and side effects in a higher-order expression language. Within this framework, dynamic data structures are handled by estimating each linked structure in an abstract store. Each call to a recursive function which creates a linked data structure is approximated by one entry in the abstract store. As with the previous method, this method fails to give fine-grain analysis for recursive data-structures.

Jones and Muchnick [JM81,JM82] have proposed a general purpose framework for data flow analysis on programs with recursive data structures. This method uses tokens to designate the points in a program where recursive data structures are created or modified. A retrieval function is then defined to finitely represent the relationships among tokens and data-values. By varying the choice of token sets and approximation lattices, a wide range of analysis can be expressed in this framework. Although flexible, the method is mostly of theoretical interest and is potentially expensive.

The most recent approach has been suggested by Larus and Hilfinger [LH88]. Their approach is designed to handle objects composed of structures. A *structure* is defined as a memory-resident object composed of a collection of named fields where each field may contain either a pointer to a structure or a non-pointer value. The analysis uses *alias graphs* to estimate the relation between variables, structures and pointers. In order to

handle general purpose structures, the alias graphs are complex and operations on alias graphs are potentially expensive.

Rather than develop a general method that encompasses all dynamic linked structures, our approach is to focus on methods for regular, recursive data structures that are widely used in practice. By restricting our method, we can exploit the regularities of the data structure in order to obtain an efficient solution which yields more useful and accurate results than would be possible otherwise. Our overall strategy is to focus on recursive data structures with regular access properties, to develop analysis methods and parallelization tools for such data structures, to implement the method with a prototype language, and to test the method on a set of representative programs. In developing and implementing the analysis and parallelization methods, we placed particular emphasis on reducing the run-time complexity by exploiting the regularities of the data structure and on chosing an abstract representation of the data structure which provides useful information for parallelization.

3 Trees and SIL Programs

In this section we will introduce our definition of the *TREE* and *DAG* data types and present our prototype language, SIL. SIL should be considered a subset of a more complete programming language. Currently, we are using SIL as a test-bed for experiments with various analysis tools. By limiting the language we can focus on the relevant constructs.

3.1 Trees and Dags

The basic building blocks of binary trees are nodes. Each *node* consists of one or more scalar values, a left pointer to a node, and a right pointer to a node. In general, objects built by linking such nodes together are *directed graphs*. We classify two special types of directed graphs: (1) a *TREE* is a directed graph in which each node has at most one parent, and (2) a *DAG* is a directed graph in which some node has more than one parent and the graph does not contain a directed cycle.

The potential for parallelism in programs that use binary trees arises from the following observation. If a program builds linked structures that are of type *TREE*, then the left and right sub-trees, T_{left} and T_{right}, of tree T are guaranteed to share no common storage.

Thus, a computation on T_{left} or any sub-tree of T_{left} will not interfere with a computation on T_{right} or any sub-tree of T_{right}. In addition, for both *TREE* and *DAG* data-structures, we can make the following observation: if node a is above node b, then node a can never be accessed starting at b.

Using these observations, we conclude that a useful interference analysis tool would: (1) check that the linked structures created by a program are guaranteed to be of type *TREE* or *DAG* and (2) recognize the relative position of named nodes in the structure.

3.2 SIL

A SIL program consists of a parameterless procedure *main* and a set of auxiliary procedures and functions. The auxiliary procedures and functions may be recursive. The language is statically scoped and has call-by-value semantics. That is, a procedure invocation passes integer and handle values to the called procedure. Note that this does *not* imply that the entire structure is copied, only the handle value in copied.

Two types are supported: integers and handles. A handle can be thought of as a name of a binary tree node. The recursive type for handles can be expressed as follows:

type handle = **Nil** | (*value*: **int**; *left*: **handle**; *right*: **handle**) .

SIL provides a built-in function *new* - each invocation of *new* allocates a new node in the store. The return value from an invocation of *new* must be assigned to a variable with type handle. A skeleton of the abstract syntax of SIL is given in figure 1 and an example program is presented in figure 7.

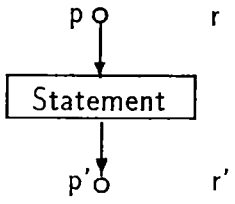
The statements of particular interest for interference analysis are those that access or modify the data-structures through the use of handles. Given that a and b are handle variables and x is an integer variable, the *basic handle statements* are of the form: $a :=$ nil, $a :=$ new(), $a := b$, $a := b.left$, $a := b.right$, $a.left := b$, $a.right := b$, $x := a.value$, and $a.value := x$. Note that more complex statements such as $a.left.right := b.right$ are easily translated into a sequence of basic handle statements ($t1 := a.left$; $t2 := b.right$; $t1.right := t2$).

```
<Program> ::= program <program_id> <ProcedureFunctionList>
<Procedure> ::= procedure <proc_id> ( <ParamList> )
                    <LocalList>
                    <Block>
<Function> ::= function <func_id> (<ParamList>) <return_type>
                    <LocalList>
                    <Block> => return ( <return_id> )
<Block> ::= begin <StmtList> end
<Arg> ::= <IntegerExpr> | <HandleName>
<Stmt> ::=  <ScalarAssignment>
          | <BasicHandleStatement>
          | if <Expr> then <Stmt> [else <Stmt>]
          | while <Expr> do <Stmt>
          | <Block>
          | <ProcedureName> ( <ArgList> )
          | <id> := <FunctionName> ( <ArgList> )
```

Figure 1: Abstract Syntax of SIL

4 Interference Analysis Tools - Computing Path Matrices

The critical information needed for interference analysis of programs containing basic handle statements is the relative position of handles at each point in the program. [2] A *point* refers to a position between two statements in the program. A handle h is *live* at a point p if there is some execution path starting at p that uses h. The structure of the analysis is illustrated as follows.



Given r, an estimate of the relationships among all handles live at point p, we wish to compute r', an estimate of the relationships among all handles live at point p'.

The estimate of relationships among handles captures the relative position of handles within a tree (or forest). Relative information can be used to detect if a statement creates a

[2] For a more complete discussion of these tools see [Hen88].

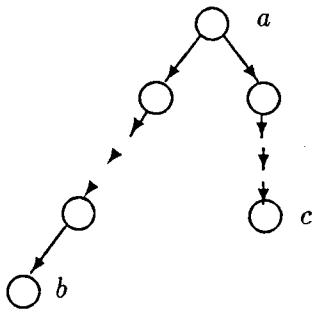
data structure that is possibly not a *TREE* or a *DAG*. For example, if node a is a descendent of node b, then the statement $a.left := b$ will create a cycle and the structure can no longer be considered a *TREE* or a *DAG*. Relative information may also be used to determine if two handles refer to disjoint sub-trees. If node a is not a descendent of node b and node b is not a descendent of node a, then a and b refer to disjoint sub-trees and a computation on a cannot interfere with a computation on b.

The *relationship* between two handles a and b, denoted by $r[a, b]$, is specified by a set of paths. A *path* is denoted either by S (meaning that two handles refer to the same node) or by a path expression which describes the directed path between two nodes. A *path expression* is non-empty sequence of links. A *link* is one of: L^i - i left edges, L^+ - one or more left edges, R^i - i right edges, R^+ - one or more right edges, D^i - i down edges, or D^+ - one or more down edges. Each path is classified as *definite*, the path is guaranteed to exist, or *possible*, the path may or may not exist. The relationships among a set of handles are described by a *path matrix*. Each entry in the matrix describes the relationship between two handles.

The structure of the analysis can now be more precisely stated. For each kind of statement, an analysis function is defined that takes as input an instance of a statement s and a path matrix p and produces as output a new path matrix p'. The *basic analysis functions* are those for handle assignments and handle updates.

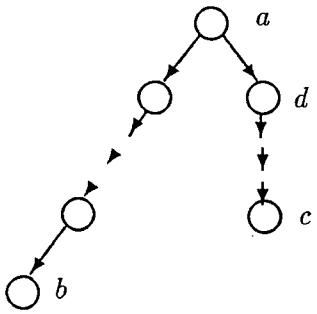
Figure 2 illustrates the application of the simple analysis function for statements of the form $a := b.f$. Figure 2(a) illustrates an initial path matrix representing the relationships between the handles a, b, and c. Note that two sorts of approximation are encoded in the path matrix: length approximation and direction approximation. The path L^2L^+ between handles a and b illustrates a path with an exact direction (left), but an approximate length (3 or more links). The path R^1D^+ between handles a and c has approximate direction (D links can be either right or left) and an approximate length (1 link right followed by 1 or more links down). Figure 2(b) shows the path matrix that would result from the statement $d := a.right$ and 2(c) illustrates the resulting path matrix after the statement $e := d.left$. Note that although the path matrices in 2(a) and 2(b) have only *definite* paths, the path matrix in 2(c) contains some *possible* paths (denoted by ?). Since the exact length of the path between handles d and a is not known, the path between of handles e and c is either $S?$ (e and c may be handles to the same node) or $D^+?$ (c is one or more edges below e).

(a) Initial path matrix



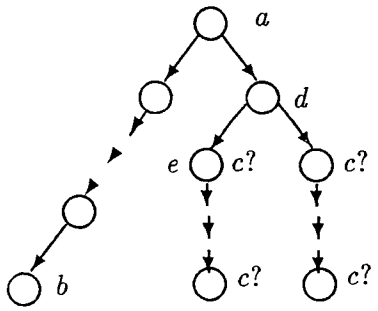
	a	b	c
a	S	$L^1L^+L^1$	R^1D^+
b		S	
c			S

(b) After statement : $d := a.right$



	a	b	c	d
a	S	$L^1L^+L^1$	R^1D^+	R^1
b		S		
c			S	
d			D^+	S

(c) After statement : $e := d.left$



	a	b	c	d	e
a	S	$L^1L^+L^1$	R^1D^+	R^1	R^1L^1
b		S			
c			S		$S?$
d			D^+	S	L^1
e			$S?, D^+?$		S

Figure 2: An example of handle assignments

$l := h;$
while $l.left \neq$ **nil do**
  $l := l.left$

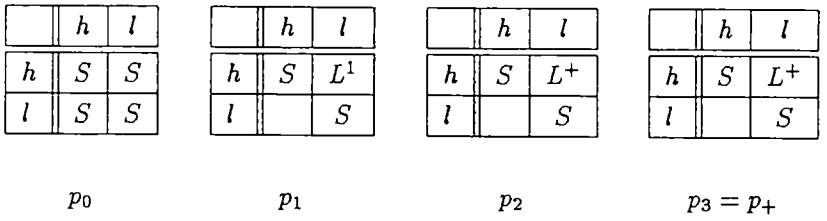


Figure 3: Iterative approximation for a simple while loop

Analysis statements are also defined for conditional statements, while statements, procedure calls and function calls. The analysis for while loops and recursive function and procedures makes use of an iterative approximation scheme. Figure 3 illustrates the approximation for a simple while loop, where p_0 represents the path matrix resulting from zero iterations of the loop, and p_+ represents the approximation for one or more iterations of the loop. The iterative approximation scheme makes use of efficient operations for merging and equality testing of path matrices.

5 Interference Analysis and Parallelization

In the previous section we presented a method for computing path matrices for each point in a program. In this section, we use path matrices in developing methods for interference analysis and parallelization.

5.1 Interference between Basic Statements

The first interference analysis method is used to determine if n basic handle statements interfere. As illustrated by figure 4, we can use such an interference analysis method to determine if several sequential statements can be transformed into a single parallel statement.

We will first consider interference between two statements. More precisely, given two

handle statements s_i and s_j, and a path matrix p, we want to determine if one statement writes to a location that the other statement reads or writes. We will then extend this method to handle n statements.

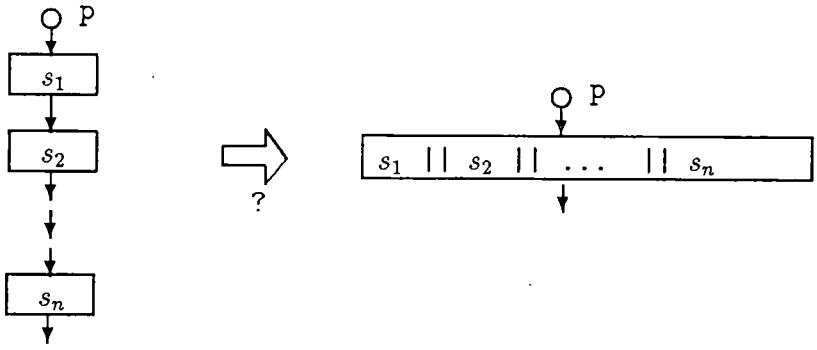


Figure 4: Transforming sequential statements to a parallel statement

For this analysis, we define the following abstraction for a location. A *location* is denoted by a pair *(name, kind)*, where *name* is the name of a variable and *kind* is one of: *var* - a variable, *left* - the left field of a node, *right* - the right field of a node, or *value* - the value field of a node.

We also define an alias function, $\mathcal{A}(a, f, p)$. Given a name a, a field kind f, and a path matrix p, the alias function returns the set of locations that may be aliased to location (a, f). Location (x, f) is an element of $\mathcal{A}(a, f, p)$ iff the path matrix entry $p[a, x]$ contains the regular expression $S?$ or S. Note that the regular expression S indicates that locations (x, f) and (a, f) are *definite aliases*, while the regular expression $S?$ indicates that locations (x, f) and (a, f) are *possible aliases*.

For each kind of handle statement, we have defined functions $\mathcal{R}(s, p)$ and $\mathcal{W}(s, p)$. Given a statement s and a path matrix p, $\mathcal{R}(s, p)$ defines a set of locations possibly read by s. Similarly, $\mathcal{W}(s, p)$ defines a set of locations possibly written by s. These functions are presented in figure 5.

The *interference set*, $\mathcal{I}(s_i, s_j, p)$, is defined as the set of locations through which statements s_i and s_j may interfere when executed at a program point with path matrix p. If $\mathcal{I}(s_i, s_j, p) = \{\}$, then there is no interference between s_i and s_j and it is safe to execute s_i

Statement - s	Read Set - $\mathcal{R}(s,p)$	Write Set - $\mathcal{W}(s,p)$
$a := nil$	$\{\}$	$\{(a, var)\}$
$a := new()$	$\{\}$	$\{(a, var)\}$
$a := b$	$\{(b, var)\}$	$\{(a, var)\}$
$a := b.f$	$\{(b, var)\} \cup \mathcal{A}(b, f, p)$	$\{(a, var)\}$
$a.f := b$	$\{(a, var), (b, var)\}$	$\mathcal{A}(a, f, p)$

Figure 5: Functions for read and write sets of statement s relative to path matrix p.

and s_j in parallel.

$$\mathcal{I}(s_i, s_j, p) = \Big[\mathcal{W}(s_i,p) \cap \big(\mathcal{R}(s_j,p) \cup \mathcal{W}(s_j,p) \big)\Big] \cup \Big[\mathcal{W}(s_j,p) \cap \big(\mathcal{R}(s_i,p) \cup \mathcal{W}(s_i,p) \big)\Big]$$

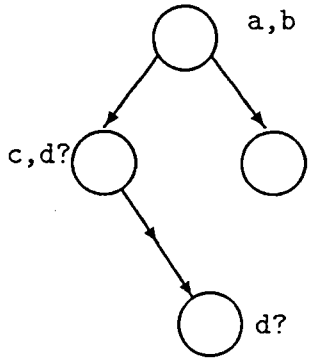
Three examples of interfering statements are given in figure 6. The first example illustrates variable interference; the statement $x := a.left$ writes variable x, while the statement $y := x$ reads variable x. The second example illustrates two statements that interfere by accessing the $left$ field of the same node. Since a and b are handles to the same node, the statement $x := a.left$ reads the same location that statement $b.left := nil$ writes. The third example illustrates the conservative nature of the interference analysis. Note that handles c and d may be handles to the same node, or handle d may be some number of $right$ links below handle c. In the first case, the statements $n := d.value$ and $c.value := 0$ would interfere on the $value$ field. However, in the second case c and d refer to different nodes and the statements would not interfere. Thus, uncertainty in the path matrix results in a conservative approximation of interference.

We can generalize this method to determine if n basic statements $[s_1, \ldots, s_n]$ interfere. $\mathcal{R}_n$, $\mathcal{W}_n$, and $\mathcal{I}_n$ are defined as follows.

$$\mathcal{R}_n([s_1, \ldots, s_n], p) = \bigcup_{i=1,n} \mathcal{R}(s_i, p)$$

$$\mathcal{W}_n([s_1, \ldots, s_n], p) = \bigcup_{i=1,n} \mathcal{W}(s_i, p)$$

A tree and corresponding path matrix



	a	b	c	d
a	S	S	L^1	D^+
b	S	S	L^1	D^+
c			S	$S?, R^+?$
d			$S?$	S

Example 1

	Statement	$\mathcal{R}(s_i, p)$	$\mathcal{W}(s_i, p)$	$\mathcal{I}(s_1, s_2, p)$
s_1	x := a.left	{(a,var),(a,left),(b,left)}	{(x,var)}	{(x,var)}
s_2	y := x	{(x,var)}	{(y,var)}	

Example 2

	Statement	$\mathcal{R}(s_i, p)$	$\mathcal{W}(s_i, p)$	$\mathcal{I}(s_1, s_2, p)$
s_1	x := a.left	{(a,var),(a,left),(b,left)}	{(x,var)}	{(a,left),(b,left)}
s_2	b.left := nil	{(b,var)}	{(b,left),(a,left)}	

Example 3

	Statement	$\mathcal{R}(s_i, p)$	$\mathcal{W}(s_i, p)$	$\mathcal{I}(s_1, s_2, p)$
s_1	n := d.value	{(d,var),(d,value),(c,value)}	{(n,var)}	{(c,value),
s_2	c.value := 0	{(c,var)}	{(c,value),(d,value)}	(d,value)}

Figure 6: Examples of interfering statements

$$\mathcal{I}_n([s_1,\ldots,s_n],s_j,p) = \begin{array}{l}\Big[\mathcal{W}_n([s_i,\ldots,s_n],p) \bigcap \Big(\ \mathcal{R}(s_j,p) \bigcup \mathcal{W}(s_j,p)\ \Big)\Big] \bigcup \\ \Big[\mathcal{W}(s_j,p) \bigcap \Big(\ \mathcal{R}_n([s_1,\ldots,s_n],p) \bigcup \mathcal{W}_n([s_1,\ldots,s_n],p)\ \Big)\Big]\end{array}$$

Statements $[s_1,\ldots,s_n]$ do not interfere at a program point with path matrix p if

$$\left(\bigcup_{i=1,n-1} \mathcal{I}_n([s_1,\ldots,s_i],s_{i+1},p)\right) = \{\}.$$

Note that we can incrementally build the set of statements that can be executed in parallel by first computing the interference set for s_1 and s_2. If this set is empty, then we can schedule s_1 and s_2 in parallel. We can continue to add statements to the parallel schedule until we reach a statement that results in a non-empty interference set.

5.2 Interference between Procedure Calls

In the previous section we outlined a fine-grain analysis that is used to detect interference among single statements. In this section we outline a coarse-grain approach to interference analysis between procedure calls. Given two procedure calls $f(x_1,x_2,\ldots,x_m)$ and $g(y_1,y_2,\ldots,y_n)$ at a program point with path matrix p, we wish to determine if the calls to f and g interfere.

A first approximation of the analysis can be obtained by examining the relationship between the handle arguments of f and the handle arguments of g at the program point just before the procedure call. The only nodes that a procedure can access are those accessed via a path from a handle argument. Recall that data structures of type $TREE$ have the property that if two handles h_i and h_j are unrelated, then all of the nodes accessed from h_i are unrelated to those accessed from h_j. Thus, if all handle arguments x_i of the call to f are unrelated to all of the handle arguments y_j of the call to g, we can conclude that the two procedure calls do *not* interfere. Since the path matrix at the point before the procedure calls is guaranteed to contain all possible relationships among handles, we can conclude that handles x_i and y_j are unrelated if $p[x_i,y_j] = p[y_j,x_i] = \{\}$.

As an example of using this method, consider the program presented in figure 7. This program adds 1 to the left sub-tree, adds -1 to the right sub-tree and then reverses the

```
program add_and_reverse
procedure main()
  root, l_side, r_side: handle; i: int
begin
  { ... build a tree at root ... }
  l_side := root.left;
  r_side := root.right;
  { ⇐ PROGRAM POINT A − path matrix p_A }
  add_n(l_side,1);
  add_n(r_side,−1);
  reverse(root)
end;
procedure add_n(h:handle; n: int)
  l,r: handle
begin
  if h ≠ nil then
    begin
      h.value := h.value + n;
      l := h.left;
      r := h.right;
      { ⇐ PROGRAM POINT B − path matrix p_B }
      add_n(l,n);
      add_n(r,n)
    end
end;
procedure reverse(h:handle)
  l, r: handle
begin
  if h ≠ nil then
    begin
      l := h.left;
      r := h.right;
      { ⇐ PROGRAM POINT C }
      reverse(l);
      reverse(r);
      h.left := r;
      h.right := l
    end
end;
```

	root	*l_side*	*r_side*
root	S	L^1	R^1
l_side		S	
r_side			S

p_A

	$h*2$	$h**2$	h	l	r
$h*2$	S	D^+	D^+	D^+L^1	D^+R^1
$h**2$		S	S	L^1	R^1
h		S	S	L^1	R^1
l				S	
r					S

p_B

Figure 7: Example Program

whole tree. Note that the procedure *add_n* updates the nodes of a tree and *reverse* actually changes the structure of the tree. The three critical program points for parallelization of procedure calls occur at program points A, B, and C. First, consider the path matrix p_A at program point A. By examining p_A we can determine that handles *l_side* and *r_side* are not related ($p_A[l\_side, r\_side] = p_A[r\_side, l\_side] = \{\}$). Therefore, the procedure calls $add\_n(l\_side, 1)$ and $add\_n(r\_side, -1)$ may be executed in parallel.

A more interesting sort of parallelism is exhibited at program points B and C. Consider the path matrix p_B at program point B. The handles in p_B can be divided into three groups: (1) h*2 is used as a symbolic name for the calling procedure's argument handle (in this case *root* from procedure *main*), (2)h**2 is used as a symbolic name that collects information on all possible handle arguments from stacked recursive invocations of *add_n*, and (3) h, l, and r contain information about the handles local to the current invocation of *add_n*. The path matrix p_B summarizes all possible relationships between handles for the recursive calls of *add_n*. Since handles l and r are not related in p_B, it is always safe to execute the recursive calls $add\_n(l, n)$ and $add\_n(r, n)$ in parallel. A similar result is obtained for the recursive calls to *reverse* at program point C.

A more accurate analysis of procedure call interference can be performed by utilizing further information about how handle arguments to a procedure are used in the body of the procedure. With the previous method, we assumed that nodes accessed through a handle argument may be updated. This is an overly conservative assumption, since some procedures may only read nodes. By adding further analysis it can be determined that a handle argument is either a *read-only* argument or an *update* argument. Handle argument x_i is *read-only* if all nodes accessed through x_i are read and not written, otherwise x_i is an *update* argument. The interference analysis can now be restricted to checking for interference due to update arguments. Let f_{update} be the set of update arguments of the call $f(x_1, \ldots, x_m)$ and g_{update} be the set of update arguments of the call $g(y_1, \ldots, y_n)$. The calls to f and g will not interfere if all handles in f_{update} are unrelated to all arguments of g and all handles in g_{update} are unrelated to all arguments of f.

Using the technique for detecting basic statement interference presented in the previous section and the technique for detecting procedure call interference presented in this section, the example program of figure 7 can be transformed into the parallel program shown in figure 8.

```
program add_and_reverse
procedure main()
  root, l_side, r_side: handle; i: int
begin
  { ... build a tree at root ... }
  l_side := root.left || r_side := root.right;
  add_n(l_side,1) || add_n(r_side,-1);
  reverse(root)
end;
procedure add_n(h:handle; n: int)
  l,r: handle
begin
  if h ≠ nil then
    begin
      h.value := h.value + n || l := h.left || r := h.right;
      add_n(l,n) || add_n(r,n)
    end
end;
procedure reverse(h:handle)
  l, r: handle
begin
  if h ≠ nil then
    begin
      l := h.left || r := h.right;
      reverse(l) || reverse(r);
      h.left := r || h.right := l
    end
end;
```

Figure 8: Parallel Version of Example Program

5.3 Interference between Statement Sequences

In this section we examine the problem of determining if two statement sequences interfere. As illustrated in figure 9, we want to determine if it is safe to execute statements sequences U and V in parallel (given the same initial program point). More precisely, given statement sequences $U = [u_1, \ldots, u_m]$ and $V = [v_1, \ldots, v_n]$ at an initial program point with path matrix p, we want to determine if one statement sequence writes to a location that the other statement sequence reads. This sort of analysis is useful both in checking that the parallel specification of $U \parallel V$ is safe and in determining that the statement sequence $U; V$ can be transformed into the parallel statement $U \parallel V$.

For this analysis we require a new notion of location. Let $\mathcal{L}$ be the set of handles that are used before they are defined in either U or V. All nodes that can be accessed in both U and V must be accessed along some path from a handle in $\mathcal{L}$ [3]. Therefore, we will

[3]For ease of presentation, we will assume that the handles in $\mathcal{L}$ are not redefined in U or V. This

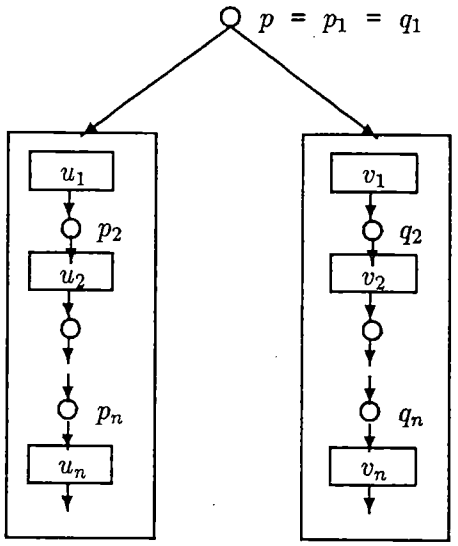


Figure 9: Initial path matrix p with two parallel statement sequences U and V

refer to locations by their access path from handles in $\mathcal{L}$. A *relative location* is a triple (*name*,*field_type*,*access_path*) where *name* is the name of a handle in $\mathcal{L}$, *field_type* is one of *var*, *left*, *right*, or *value*, and *access_path* is a set of path expressions describing the path from *name* to the node which is being read or updated.

Statement - s	$\mathcal{R}^r(s,p)$	$\mathcal{W}^r(s,p)$
$a := nil$	$\{\}$	$\{(a, var, S)\}$
$a := new()$	$\{\}$	$\{(a, var, S)\}$
$a := b$	$\{(b, var, S)\}$	$\{(a, var, S\}$
$a := b.f$	$\{(b, var, S)\} \cup \mathcal{A}^r(b, f, \mathcal{L}, p)$	$\{(a, var, S)\}$
$a.f := b$	$\{(a, var, S), (b, var, S)\}$	$\mathcal{A}^r(a, f, \mathcal{L}, p)$

Figure 10: Relative read and write sets given path matrix p and live handles $\mathcal{L}$.

The relative read and write functions $\mathcal{W}^r(s,p,\mathcal{L})$ and $\mathcal{R}^r(s,p,\mathcal{L})$ are defined as in figure 10. In these rules we use the relative alias function $\mathcal{A}^r$. Given a handle h, a field name f, a

restriction can be lifted by automatic renaming of variables.

set of live handles $\mathcal{L}$, and a path matrix p, $\mathcal{A}^r(h, f, \mathcal{L}, p)$ returns the set of relative locations which are possibly aliased to the location referred to by $h.f$. $\mathcal{A}^r(h, f, \mathcal{L}, p)$ contains the relative location (x, f, r) iff l is one of the handles in $\mathcal{L}$ and the path matrix entry $p[l, h]$ contains the path expression r.

We define $\mathcal{R}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L})$ to be the *relative read set* for statement sequence $[s_1, \ldots, s_n]$ and $\mathcal{W}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n, \mathcal{L}])$ to be the *relative write set* for statement sequence $[s_1, \ldots, s_n]$.

$$\mathcal{R}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L}) = \bigcup_{i=1,n} \mathcal{R}^r(s_i, p_i, \mathcal{L})$$

$$\mathcal{W}^r_n([s_1, \ldots, s_n], [p_1, \ldots, p_n], \mathcal{L}) = \bigcup_{i=1,n} \mathcal{W}^r(s_i, p_i, \mathcal{L})$$

Let $P = [p_1, \ldots, p_m]$ be the sequence of path matrices associated with the statements u_i of U, and $Q = [q_1, \ldots, q_n]$ be the sequence of path matrices associated with the statements v_j of V (as illustrated in figure 9. The *relative interference set*, $\mathcal{I}^r(U, P, V, Q, \mathcal{L})$, is defined as follows.

$$\mathcal{I}^r(U, P, V, Q, \mathcal{L}) = \begin{bmatrix} \mathcal{W}^r(U, P, \mathcal{L}) \cap \left(\mathcal{R}^r(V, Q, \mathcal{L}) \cup \mathcal{W}^r(V, Q, \mathcal{L}) \right) \end{bmatrix} \cup \begin{bmatrix} \mathcal{W}^r(V, Q, \mathcal{L}) \cap \left(\mathcal{R}^r(U, P, \mathcal{L}) \cup \mathcal{W}^r(U, P, \mathcal{L}) \right) \end{bmatrix}$$

If the data structure is a *TREE* at the program point just before $U \parallel V$, then U and V do not interfere if $\mathcal{I}^r(U, P, V, Q, \mathcal{L}) = \{\}$. The proof of this observation is an induction on the height of the tree. This result does not hold if the data-structure is a *DAG* or an arbitrary graph. For these structures a more complex interference analysis method must be used.

6 Conclusions and Further Work

In this paper, we have presented a new approach to parallelizing programs with dynamic recursive data-structures. By focusing our approach on data-structures with regular (recursive) properties, we were able to develop efficient and effective tools for interference analysis

and parallelization. We illustrated our approach will a small example. A more demanding example, the adapative bitonic sort in [BN86], has also been analyzed resulting in significant parallelism detection.

We plan to extend the parallelization techniques to detect fine-grain parallelism such as is achieved by partial overlapping and pipelining of procedure calls. In addition, we plan to extend the use of our analysis to include program transformations that expose parallelism. Finally, we plan use the analysis in developing debugging tools for more sophisticated parallel and synchronization constructs.

References

[Ban79a] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October 1979. Dept. of Computer Science Rpt. 79-989.

[Ban79b] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *6th ACM Symposium on Principles of Programming Languages*, 1979.

[Bar78] J. Barth. A practical interprocedural data flow analysis algorithm. *CACM*, 21:724–736, 1978.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *ACM Sigplan Notices, Vol 21,7*, 1986.

[BN86] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical Report TR86-769, Cornell University, 1986.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *Proceedings of 1986 ACM Conference on Lisp and Functional Programming*, 1986.

[Hen88] Laurie J. Hendren. Recusive data structures and parallelism detection. Technical Report TR 88-924, Cornell University, June 1988.

[Hud86] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986.

[JM81] N. D. Jones and S. S. Muchnick. *Flow Analysis and Optimization of LISP-like Structures*, chapter 4, pages 102 – 131. Prentice-Hall, 1981.

[JM82] N. D. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings 15th POPL*, pages 47–57, 1988.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 - Conference on Programming Language Design and Implementation*, pages 21–34, 1988.

[Luc87] J. M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT, 1987.

[Nei88] A. Neirynck. *Static Analysis of Aliasing and Side Effects in Higher-Order Languages.* PhD thesis, Cornell University, January 1988.

[Nic84] A. Nicolau. *Parallelism, Memory Anti-Aliasing, and Correctness for Trace Scheduling Compilers.* PhD thesis, Yale University, 1984.

[NPD87] A. Neirynck, P. Panangaden, and A.J. Demers. Computation of aliases and support sets. In *14th ACM Symposium of Principles of Programming languages*, 1987.

[PA72] J.R. Phillips and H.C. Adams. Dynamic partitioning for array languages. *CACM*, 15:1023–1032, 1972.

[Rey78] J. C. Reynolds. Syntactic control of interference. In *5th ACM Symposium on Principles of Programming Languages*, 1978.

[RM88] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings 15th POPL*, pages 285–293, 1988.

[Ten83] R. D. Tennent. Semantics of interference control. *Theoretical Computer Science*, 27:297–310, 1983.

[Wei80] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *7th ACM Symposium on Principles of Programming Languages*, 1980.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.