Identifying Parallelism in Programs with Cyclic Graphs

Yuan-Shin Hwang
Department of Computer Science & Information Engineering
National Chi Nan University
Puli, Nantou 545
Taiwan
shin@csie.ncnu.edu.tw

Joel Saltz
Department of Computer Science
University of Maryland
A.V. Williams Building
College Park, MD 20742
U.S.A.
saltz@cs.umd.edu

January 19, 2000

Abstract

Dependence analysis algorithms have been proposed to identify parallelism in programs with tree-like data structures. However, they can not analyze the dependence of statements if recursive data structures of programs are cyclic. This paper presents a technique to identify parallelism in programs with cyclic graphs. The technique consists of three steps: (1) Traversal patterns that loops or recursive procedures traverse graphs are identified, and the statements that construct the links of traversal patterns will be located by definition-use chains of recursive data structures; (2) Traversal-pattern-sensitive shape analysis is performed to estimate possible shapes of traversal patterns; (3) Dependence analysis is performed to identify parallelism using the result of shape analysis. This approach can identify parallelism in programs with cyclic data structures due to the facts that many programs follow acyclic structures (i.e. traversal patterns) to access all nodes on the cyclic data structures. Once the traversal patterns are isolated from the overall data structures, dependence analysis can be applied to identify parallelism.

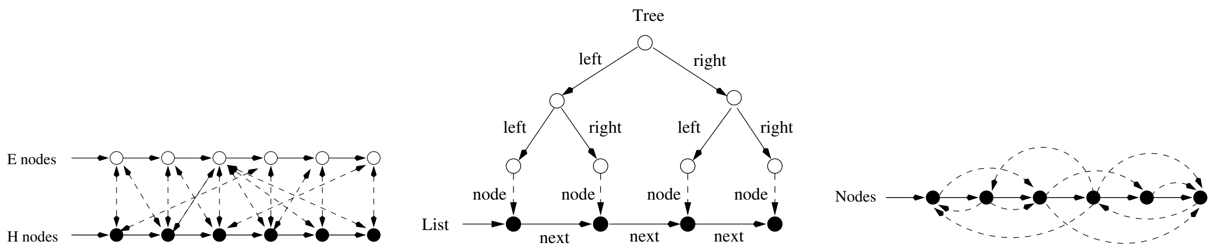
Corresponding Author:

Yuan-Shin Hwang
Department of Computer Science & Information Engineering
National Chi Nan University
Puli, Nantou 545
Taiwan
Email: shin@csie.ncnu.edu.tw

1 Introduction

Dependence analysis with the help of alias analysis or shape analysis is the key technique to identify parallelism in programs with pointers and dynamic recursive data structures. There have been several dependence analysis techniques proposed to analyze the data dependence of programs with pointers and dynamic recursive data structures [13, 16, 17, 18, 22]. These techniques successfully identify parallelism in certain classes of programs, but can not recognize parallelism in programs with cyclic recursive data structures even though the traversal patterns of these programs exhibit high degree of parallelism. This paper presents a dependence analysis algorithm that identifies the traversal patterns of programs with cyclic graphs, isolates the statements that traverse or define the links of traversal patterns, and analyzes the data dependence and shape information of these statements. Consequently, this approach can identify parallelism based on data dependence and shape analysis on programs even with cyclic graphs [19].

Although many programs create recursive data structures which appear to be cyclic overall, they usually follow acyclic structures to access all nodes on the data structures. For instance, graph algorithms frequently extract acyclic structures, such as spanning trees, from cyclic graphs and traverse the graphs following the links of the acyclic structures, while the rest of edges are merely used to reference values on neighboring nodes. Furthermore, recursive data structures can have unbounded numbers of nodes and are commonly traversed by either iterative or recursive program constructs, such as loops or recursive procedures. The edges along which loops or recursive procedures traverse a recursive data structure constitute traversal patterns of the program constructs, and can be viewed as the skeleton of the recursive data structure. On the other hand, the remaining edges of the recursive data structure are generally used to reference values on other nodes. Accordingly, it will be beneficial if compilers can differentiate links of recursive data structures based on the traversal patterns when pointer analysis is performed.



(a) Bipartite (EM3D)

(b) Leaf-Linked Tree (B-H)

(c) Cyclic List (Sparse Matrix)

Figure 1: Cyclic Graphs with Acyclic Traversing Patterns

One example of programs with cyclic structures but acyclic traversal patterns is the EM3D program, which constructs a bipartite graph shown in Figure 1(a) to model the propagation of electromagnetic waves [23]. It updates the values of E nodes (electric field) by a weighted sum of neighboring H nodes (magnetic field), and then H nodes are similarly updated using the E nodes. As a result, it traverses the list of E nodes through the solid (traversal) links and reads information from H nodes by dashed (reference) links, and then similarly traverses the H node list and references E nodes. Although the bipartite graph is cyclic, the traversal patterns (lists of E and H nodes) are not. Another example is the Barnes-Hut N-Body solver, which creates a leaf-connected tree to exploit spatial properties of the N bodies [3], as depicted in Figure 1(b). The structures traversed by the program are a list and a tree, respectively. Figure 1(c) shows the cyclic structures used to represent sparse matrixes, which are common in programs that simulate interactions among entities. These types of cyclic data structures are common in computational scientific simulation programs. Most of these simulation programs exhibit high degree of parallelism since their traversal patterns are either lists or trees, even though the overall data structures appear to be cyclic. These examples show that the knowledge of future traversal patterns can provide important hints for compilers to perform proper pointer analysis on programs even with cyclic recursive data structures.

The approach to identify parallelism from programs with cyclic graphs can be divided into the following steps:

- Traversal patterns that loops or recursive procedures traverse graphs are identified. In order to estimate the possible shapes of traversal patterns, the statements that construct the links of traversal patterns must be located. Definition-use chains of recursive data structures will be used to identify such statements from traversal patterns.
- Shape analysis will be performed on the identified statements. The results will be the possible shapes of traversal patterns.
- The shape information of traversal patterns will facilitate dependence analysis to identify parallelism in programs with cyclic graphs.

The rest of this paper will be organized following the above steps. The impact of traversal patterns on pointer analysis and background information will be outlined in Section 2. Section 3 will present an algorithm to compute the definition-use chains of recursive data structures. Section 4 will propose a shape analysis approach, called *traversal-pattern-sensitive* shape analysis, that estimates possible shapes of traversal patterns. Section 5 will briefly outline a dependence analysis algorithm that exploits the shape information of traversal patterns to identify parallelism in programs with cyclic graphs. Section 6 will present the experimental results. Related work will be presented in Section 7.

2 Background

2.1 Programming Model

The algorithms presented in this paper are designed to analyze programs with dynamic recursive data structures that are connected through pointers defined in the languages like Pascal and Fortran 90. Pointers are specified by declared pointer variables, and are simply references to nodes with a fixed number of fields, some of which are pointers. Memory allocations are done by the function new(). Pointer arithmetic in languages such as C is not allowed. Although multi-level pointers are not considered in this paper, they can be handled by converting them into levels of records, each of which contains only one field that carries the node location of the next level. Consequently, pointer dereferences of multi-level pointers can be treated as traversal of multi-level records [19].

Programs will be normalized such that each statement contains only simple *binary access paths*, each of which has the form $v.n$ where v is a pointer variable and n is a field name [5]. Therefore, excluding regular assignment statements, the three possible forms of pointer assignment statements are

1. $p = q$ (*aliasing statements*)
2. $p = q.n$ (*link traversing statements*)
3. $p.n = q$ (*link defining statements*)

Note that although the other possibility $p_i.m = q_j.n$ is also valid, it is represented by two consecutive statements, $t_k = q_j.n$ and $p_i.m = t_k$, for the reason of simplicity.

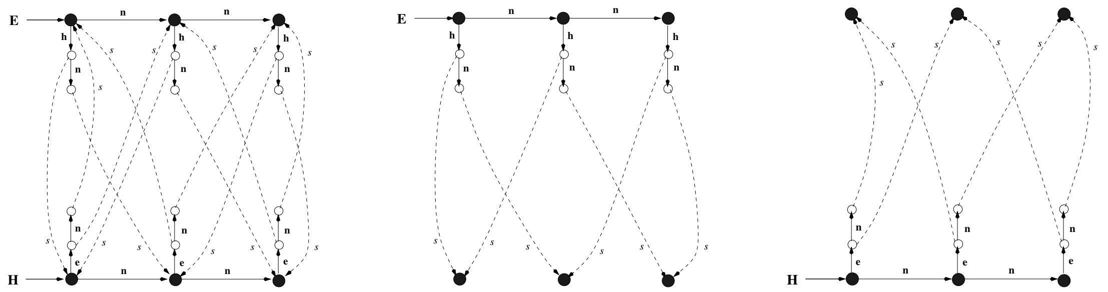
Since only single-level pointers are allowed, programs can be transformed into an SSA (Static Single Assignment) intermediate representation [8]. Although SSA form is designed specially for programs with fixed-location variables only, e.g. Fortran-77 programs, same transformation can be applied to pointer variables since contents (location addresses) of pointer variables can be treated as values in regular variables. Once normalized programs are transformed into SSA representation, a new form of pointer assignments $p_i = \phi\ (p_j,\ p_k)$ will be introduced and be placed at merging points of programs.

2.2 Impact of Traversal Patterns on Pointer Analysis

Although the example programs in previous section create recursive data structures which appear to be cyclic overall, they follow acyclic structures to access all nodes on the data structures. Previously proposed shape analysis algorithms will conclude the recursive data structures created by these example to be cyclic, while the existing alias analysis algorithms will report all the nodes of these structures are aliased. The reason is because these techniques are performed without the knowledge of traversal patterns by programs on the recursive data structures to be analyzed. Without such knowledge, for each statement $p.f = q$ compilers have to estimate all possible shapes or aliases of following connections (paths):

- From any pointers, say x, to p and then to q through edge $p.f$, i.e. $x \overset{\star}{\leadsto} p \overset{f}{\leadsto} q$, and
- From p to q through edge $p.f$ and then from q to all pointers, say y, i.e. $p \overset{f}{\leadsto} q \overset{\star}{\leadsto} y$.

In the first case, if the program does not traverse the path from x to p, the path p to q is irrelevant to x. Similarly, if there do not exist traversal patterns that travel the paths from q to y in the second case, it is not necessary to analyze the paths from p to y. In summary, both cases show that the traversal patterns starting from each pointer provide a useful piece of information to help compilers avoid analyzing irrelevant connections, which might in turn cause compilers to make over-conservative estimations.



(a) Bipartite Graph (b) Graphs Seen from E (c) Graphs Seen from H

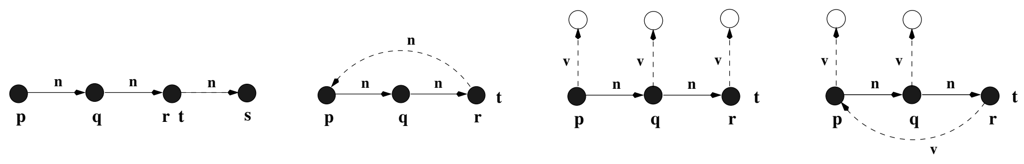
Figure 2: Traversal Patterns of EM3D on a Bipartite Graph

Consider the recursive data structure shown in Figure 2(a) that represents a bipartite graph created by EM3D. The traversal patterns of two loops that compute values of all E nodes and H nodes are $E(.n)^\star.h(.n)^\star.s$ and $H(.n)^\star.e(.n)^\star.s$, respectively. With this knowledge, compilers can perform shape analysis only on the subgraphs that are specified by these traversal patterns and hence can identify the fact that the graphs seen from E and H in EM3D are acyclic, even though the overall graph is cyclic, as shown in Figure 2(b) and Figure 2(c), respectively. Such approach can be called *traversal-pattern-sensitive* shape analysis. Otherwise, compilers must perform shape analysis on all possible paths, including $E(.n)^{\star}.h(.n)^{\star}.s.e(.n)^{\star}.s$ and $H(.n)^{\star}.e(.n)^{\star}.s.h(.n)^{\star}.s$, and will conclude the graphs starting from E and H are both cyclic.

2.3 Definition-Use Chains and Traversal-Pattern-Sensitive Shape Analysis

In order to estimate the possible shapes of graphs that are connected by the links specified by traversal patterns, the statements that construct these links must be identified since shapes of graphs are determined by such statements. In other words, for each graph link that is referenced by any link traversing statements, all the corresponding link defining statements that might define the link must be identified. Furthermore, traversal-pattern-sensitive shape analysis will performed on the identified link defining statements to estimate the possible shapes of traversal patterns.

Once traversal patterns are identified, the corresponding link defining statements can be located by the definition-use chains of recursive data structures, since traversal operations by link traversing statements are read references (i.e. *uses*) to links while link construction operations by link defining statements are write references (i.e. *definitions*). Consider the graphs in Figure 3(a)–(d) that are created by code fragments (a)–(d) in Figure 3(e). The graphs are traversed by the statements S6 and S7, and consequently S6 and S7 determine the traversal patterns of the graphs created by code fragments (a)–(d), .i.e. the solid lines in Figure 3(a)–(d). Furthermore, the definition-use chains of the code fragments (a)–(d) are the pairs (S1, S6) and (S2, S7). As a result, the possible shapes of traversal patterns of graphs in Figure 3(a)–(d) are determined by S1 and S2. After analyzing on S1 and S2, shape analysis will conclude the traversal patterns of all graphs are trees, whereas the overall shapes of graphs are either trees (i.e. Figure 3(a) and Figure 3(c)) or cyclic graphs (Figure 3(b) and Figure 3(d)).



(a) (b) (c) (d)

	(a)	(b)	(c)	(d)
S1:	p.n = q	p.n = q	p.n = q	p.n = q
S2:	q.n = r	q.n = r	q.n = r	q.n = r
S3:	r.n = s	r.n = p	p.v = new()	p.v = new()
S4:			q.v = new()	q.v = new()
S5:			q.v = new()	q.v = p
S6:	t = p.n			
S7:	t = t.n			

(e) Code Fragments that Create and Traverse Graphs (a)–(d)

Figure 3: Definition-Use Chains and Traversal-Pattern-Sensitive Shape Analysis

2.4 Approach Outline

The approach to identify parallelism in programs with cyclic graphs can be broken into three steps.

- *Gather Traversal Patterns and Compute Definition-Use Chains of Recursive Data Structures*
  Definition-use chains of recursive data structures will be computed and meanwhile traversal patterns of iterative or recursive program constructs, such as loops or recursive functions, will be gathered. For each statement that references a link of a graph, all corresponding statements that might define the link will be identified using the information of definition-use chains.

- *Perform Traversal-Pattern-Sensitive Shape Analysis*
  Once the statements that construct the graphs accessed by traversal patterns are identified, traversal-pattern-sensitive shape analysis will be performed to estimate possible shapes of the traversal patterns.

- *Perform Dependence Analysis*
  Dependence test is performed to determine if access conflicts occur between the sets of read and write references based on data reference patterns and the result of shape analysis on traversal patterns and overall data structures.

Algorithms that compute definition-use chains of recursive data structures, perform traversal-pattern-sensitive shape analysis, and analyze dependence will be presented in the following sections.

3 Definition-Use Chains of Recursive Data Structures

Traversal patterns are determined by link traversal operations on recursive data structures, i.e. they are *uses* of the referenced data structures. In order to identify the statements that construct the links accessed by traversal patterns, compilers must identify the *definitions* of these uses. Therefore, this section presents an algorithm that computes definition-use chains of recursive data structures.

3.1 Problem Specifications

Figure 4 depicts the effects of different types of pointer assignment statements on a recursive data structure $list$. The aliasing statement $t = list$ directs the pointer t to the node pointed to by the pointer $list$. This statement references the pointer $list$, but does not induce any references to the recursive data structure. The link traversing statement $p = r.next$ references a link of the recursive data structure, which is stored in the field $r.next$, and assigns the pointer p the destination node of the link $r.next$. As a result, the execution of this statement introduces a use on the structure $list$. On the other hand, the execution of the link defining statement $p.neigh = q$ creates an edge to connect the recursive structure $list$ and the node q. This statement changes the connections of the structure by adding one link, and consequently is a definition of the structure $list$.

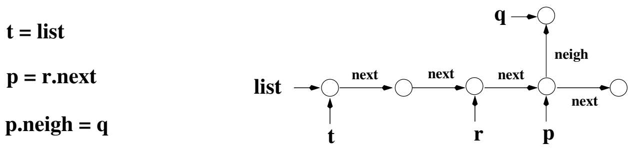


Figure 4: Effects of Pointer Assignment Statements on a Recursive Data Structure

Since link defining statements and link traversing statements correspond to definitions and uses of recursive data structures respectively, the definition-use chains of recursive data structures can be represented by the definition-use chains between linking defining statements and link traversing statements. Consequently, the problem to compute the definition-use chains of dynamic recursive data structures is formulated as

> For each statement S that defines links of recursive data structures, find the set of statements that access the links which are defined by S.

Specifically, when S_i is a definition of $p.f$ and S_j contains a use of $q.f$, as shown in Figure 5, then $q.f$ is a use of $p.f$ if

- There is at least one path from S_i to S_j such that $p.f$ can reach S_j, and
- p and q access the same locations.

Si: p.f = ...

Sj: ... = q.f

Figure 5: Definition-Use Association

3.2 Algorithm

The algorithm that computes the definition-use chains of recursive data structures models the backward data flow analysis that computes definition-use chains of fixed-location variables [1]. Instead of the sets of uses of fixed-location variables, the sets of uses of binary access paths that are reachable will be computed. Therefore, backward data flow analysis can be performed to compute the set of upward exposed uses of the binary access paths, each of which is a pair $(S,\, p.f)$ such that S is a statement that uses the binary access path $p.f$ and such that no prior definition of $p.f$ occurs. As each pair $(S,\, p.f)$ is propagated backward, the pointer p will be represented by an *access path expression*, which is a pointer (called the *entry* of the path) followed by a string of field names connected by the field component operator "." [6, 9]. The access path expressions of pointers in the use pairs will be transformed by statements, and the definition-use chains will be identified at link defining statements.

The set of upward exposed uses of every statement will be collected to compute the definition-use chains of dynamic recursive data structures. The set of uses that can be reached from statement S can be solved by the following data flow equations:

$$UPEXP_{out}[S] = \bigcup_{d \in succ(S)} UPEXP_{in}[d] \quad (1)$$

$$UPDEF_{out}[S] = \bigcup_{d \in succ(S)} UPDEF_{in}[d] \quad (2)$$

$$UPEXP[S] = USE[S] \cup F_S(UPEXP_{out}[S]) \quad (3)$$

$$UPDEF[S] = DEF[S] \cup F_S(UPDEF_{out}[S]) \quad (4)$$

$$UPDEF_{in}[S] = F_{UPDEF[S]}(UPDEF[S]) \quad (5)$$

$$UPEXP_{in}[S] = F_{UPDEF[S]}(UPEXP[S]) \quad (6)$$

algorithm ComputeDefUseChains (G)
input:
  G: CFG of the procedure
output:
  DefUse$[S]$: The set of statements that use the definition of S
begin
$\{$initialize sets$\}$
  for each CFG node S **do** $UPEXP_{in}[S]$ = $UPDEF_{in}[S]$ = $DefUse[S]$ = $\emptyset$
$\{$iteratively analyze a procedure$\}$
  while changed
    for each CFG node S in reverse topological order **do**
      $\{$Compute definition and use sets$\}$
      $UPEXP_{out}[S] = \bigcup_{d \in succ(S)} UPEXP_{in}[d]$
      $UPDEF_{out}[S] = \bigcup_{d \in succ(S)} UPDEF_{in}[d]$
      $UPEXP[S] = USE[S] \cup F_S\ (UPEXP_{out}[S])$
      $UPDEF[S] = DEF[S] \cup F_S\ (UPDEF_{out}[S])$
      $UPDEF_{in}[S] = F_{UPDEF[S]}(UPDEF[S])$
      $UPEXP_{in}[S] = F_{UPDEF[S]}(UPEXP[S])$
      $\{$Normalize access path expressions$\}$
      if S is the header node of a loop **then** $Factor(UPDEF_{in}[S])$ and $Factor(UPEXP_{in}[S])$
    end
  end
$\{$compute definition-use chains$\}$
  for each CFG node S **do**
    if $S : p_i.n = q_j$ and $\langle S', p_i, n\rangle \in UPEXP_{out}[S]$ **then** $DefUse[S] = DefUse[S] \cup \{S'\}$
    if $\langle S1, p, n\rangle \in UPDEF_{out}[S] \wedge \langle S2, q, n, r\rangle \in UPEXP_{out}[S] \wedge p \equiv q \wedge$ S1 partial-dominates S2
      then $DefUse[S1] = DefUse[S1] \cup \{S2\}$
  end
end

Figure 6: Algorithm for Computing Definition-Use Chains within a Procedure

where $UPEXP_{out}[S]$ and $UPEXP_{in}[S]$ are the set of upward exposed uses after and before S respectively, $USE[S]$ is the set of uses generated by the statement S, F_S is the *transfer function* of S, succ(S) is the set of successors of S, $UPDEF_{out}[S]$ and $UPDEF_{in}[S]$ are the set of upward exposed definitions after and before S respectively, $DEF[S]$ is the set of definitions generated by the statement S, and $F_{UPDEF[S]}$ is the transfer function of the set $UPDEF[S]$.

The iterative algorithm that computes the definition-use chains of dynamic recursive data structures by solving the data flow equations (1)–(6) is presented in Figure 6. The algorithm takes the CFG (control flow graph) of the procedure as input and computes the definition-use chains as output. After initializing the sets, it iteratively traverses all the nodes of the CFG in reverse topological order until no new definitions or uses are added to any sets. Once the iterative process terminates, the definition-use chains can then be computed.

algorithm Factor(p)
input:
  p: an access path expression
output:
  p': a normalized access path expression
begin
  Let $p \equiv p_i.f_1.f_2.\cdots.f_n$
  $p' = p_i$; $e = (f)^1$ where $f \equiv f_1$
  for each f_i $i = 2, n$ **do**
    if f_i is a recursively defined field and $type(e) \equiv type(f_i)$ **then**
      $e = (f)^+$ where $f \equiv f \mid f_i$
    else
      Append e to p'
      $e = (f)^1$ where $f \equiv f_i$
    end
  end
  return p'
end

Figure 7: Normalization Algorithm $Factor(p)$

3.3 Normalization of Access Path Expressions

Since the loops can have unbounded numbers of iterations, the access path expressions of definition and use tuples might have unbounded numbers of fields after being transformed by transfer functions of statements in loops. In order to ensure the iterative data flow algorithm will terminate, the length of access path expressions must be finite. Therefore, access path expressions will be normalized at header nodes of loops. Figure 7 shows the algorithm $Factor(p)$ that normalizes an access path expression p. The algorithm is a simplified algorithm which is adapted from the normalization algorithm for symbolic access paths proposed by Deutsch [9] and the normalization algorithm for path expressions proposed by Hendren [15].

The algorithm examines all the fields of the path string of an access path expression p. Initially the first field f_1 will be stored in e, and then fields will be examined from left to right. If f_i is a recursively defined type and its type is the same as the type of e, then f_i will be merged to e. Otherwise, e will be appended to the end of path string of p' and then e will be initialized as f_i. The result p' of the normalization $Factor(p)$ will have the form $p_i.e_1.f_i.e_2.f_j.\cdots.e_k$, where $e = (f|\cdots)^+$. For example, the access path expression $p_i.n.n$ will be normalized to $p_i.(n)^+$, while $h_i.left.right$ will be transformed to $h_i.(left|right)^+$.

3.4 Example

Figure 8 presents a loop that reverses a linked list and the iterations to compute the set of upward exposed uses of each statement. Each use pair $(S,\ p.f)$ is represented by a tuple $\langle S,\ p,\ f\rangle$. It identifies that the use of S8 is defined by statement S6. The use tuples at S1 mean that the uses of S5 and S9 are defined by the statements before the list reverse loop. This example shows the following features: (1) the algorithm can handle programs with destructive update operations [26]; (2) It correctly identifies the definition-use chain between S6 and S8, and does not introduce the spurious definition-use chain between S6 and S5 even though $list_2$ and $rlist_3$ will be deemed as aliases by existing

Program	Iteration 1	Iteration 2	Iteration 3
S1: $rlist_1$ = nil	$\langle S9, list_1, v\rangle$ $\langle S5, list_1, n\rangle$	$\langle S9, list_1, v\rangle$ $\langle S9, list_1.n, v\rangle$ $\langle S5, list_1, n\rangle$ $\langle S5, list_1.n, n\rangle$	$\langle S9, list_1, v\rangle$ $\langle S9, list_1.n^+, v\rangle$ $\langle S5, list_1, n\rangle$ $\langle S5, list_1.n^+, n\rangle$
S2: do while ($list_2$) $list_2 = \phi$ ($list_1$, $list_3$) $rlist_2 = \phi$ ($rlist_1$, $rlist_3$)	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S5, list_2, n\rangle$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S9, list_2.n, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n, n\rangle$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S9, list_2.n^+, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n^+, n\rangle$
S3: t = $rlist_2$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S5, list_2, n\rangle$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S9, list_2.n, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n, n\rangle$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S9, list_2.n^+, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n^+, n\rangle$
S4: $rlist_3$ = $list_2$	$\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S5, list_2, n\rangle$	$\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S9, list_2.n, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n, n\rangle$	$\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, list_2, v\rangle$ $\langle S9, list_2.n^+, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n^+, n\rangle$
S5: $list_3$ = $list_2$.n	$\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S5, list_2, n\rangle$	$\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, list_2.n, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n, n\rangle$	$\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, list_2.n^+, v\rangle$ $\langle S5, list_2, n\rangle$ $\langle S5, list_2.n^+, n\rangle$
S6: $rlist_3$.n = t	– $\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, rlist_3, v\rangle$	– $\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, list_3, v\rangle$ $\langle S5, list_3, n\rangle$	– $\langle S8, t, n\rangle$ $\langle S8, t.n^+, n\rangle$ $\langle S9, t, v\rangle$ $\langle S9, t.n^+, v\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, list_3, v\rangle$ $\langle S9, list_3.n, v\rangle$ $\langle S5, list_3, n\rangle$ $\langle S5, list_3.n, n\rangle$
S7: end do	$\langle S8, rlist_3, n\rangle$ $\langle S8, rlist_3.n^+, n\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, rlist_3.n^+, v\rangle$	$\langle S8, rlist_3, n\rangle$ $\langle S8, rlist_3.n^+, n\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, rlist_3.n^+, v\rangle$ $\langle S9, list_3, v\rangle$ $\langle S5, list_3, n\rangle$	$\langle S8, rlist_3, n\rangle$ $\langle S8, rlist_3.n^+, n\rangle$ $\langle S9, rlist_3, v\rangle$ $\langle S9, rlist_3.n^+, v\rangle$ $\langle S9, list_3, v\rangle$ $\langle S9, list_3.n, v\rangle$ $\langle S5, list_3, n\rangle$ $\langle S5, list_3.n, n\rangle$
S8:	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$	$\langle S8, rlist_2, n\rangle$ $\langle S8, rlist_2.n^+, n\rangle$ $\langle S9, rlist_2, v\rangle$ $\langle S9, rlist_2.n^+, v\rangle$

Figure 8: List Reverse Example and Its Upward Exposed Uses

aliases algorithms; (3) the same results will be produced regardless of the shape of the linked list $list_1$, which can be either an acyclic list or a cyclic list.

3.5 Interprocedural Analysis

The intraprocedural definition-use algorithm can be extended into a flow-sensitive interprocedural algorithm. The results of intraprocedural analysis of each procedure of a program will be stored as local information on an *interprocedural flow graph (IFG)*, which is in turn based on the interprocedural flow graph [14] and the program summary graph [4], and interprocedural definition-use chains will be identified by propagating the local information through the edges of the IFG [19].

The entry and the exit of each procedure are represented by an ENTRY node and an EXIT node in an IFG respectively, while each call site is represented by a pair of CALL and RETURN nodes. The edges between the nodes of an IFG represent the flow control of the program. Therefore, the steps to perform interprocedural analysis are:

- Construct IFG
- Perform intraprocedural analysis and annotate local information on IFG nodes
- Compute interprocedural definition-use chains

4 Traversal-Pattern-Sensitive Shape Analysis

This paper proposes an approach, called *traversal-pattern-sensitive* shape analysis, that can estimate possible shapes of traversal patterns. The idea is that once the traversal patterns of programs are recognized, the statements that construct the links that are accessed by traversal patterns can be identified by definition-use chains and shape analysis can be performed on these identified statements. The results of traversal-pattern-sensitive shape analysis will be possible shapes of traversal patterns of programs.

4.1 Primary vs. Secondary Traversal Links

When recursive data structures are traversed by either iterative or recursive program constructs, such as loops or recursive procedures, some of the destination nodes of links which are traversed at current iteration turn out to be the source nodes of link traversal operations of the next iteration. These links are the links that loops or recursive procedures traverse to access the nodes of the recursive data structures. Therefore, these links are called *primary traversal links* and the structures that are connected by primary traversal links are called *primary traversal structures*. The rest of links are called *secondary traversal links* or *reference links*. Their definitions are given as follows:

Definition 1 PRIMARY TRAVERSAL LINKS
A link of a recursive data structure is a primary traversal link of the structure if the link is traversed by a link traversing statement and if the destination node of the link traversal is the source node of link traversal by another link traversing statement or by the same link traversing statement at next iteration.

Definition 2 PRIMARY TRAVERSAL STRUCTURES
The primary traversal structure of a recursive data structure is the part of the structure that is connected by primary traversal links only.

For example, the solid edges in Figure 2 are primary traversal links while the dashed edges are reference links. Furthermore, the primary traversal structures in the figure can be represented by the access paths $E(.n)^\star.h(.n)^\star$ and $H(.n)^\star.e(.n)^\star$.

Since the link traversal operations are performed by link traversing statements only, the primary traversal links of recursive data structures can be identified by examining the source and destination nodes of link traversing statements. For example, the link that is traversed by a statement $S: p_i = q_j.f$ is a primary traversal link if there exists another statement that traverses a link starting from p_i, say $S': r_k = p_i.f$. Therefore, the statements that traverse the primary traversal links can be recognized by examining the statements in a program other than link defining statements, and these link traversing statements will be called *primary link traversing statements*.

Once the link traversing statements that traverse primary traversal links are identified, the link defining statements that construct the primary traversal links of recursive data structures can be located by the definition-use information. A link defining statement will define primary traversal links if any of its uses are primary link traversing statements, and such a link defining statement will be called a *primary link defining statement*.

4.2 Approach

Existing shape analysis can be easily modified to traversal-pattern-sensitive algorithms. This paper presents a traversal-pattern-sensitive shape analysis that is adapted from the shape analysis algorithm proposed by Ghiya and Hendren [11]. The original algorithm computes three storeless abstractions at each statement: a *direction matrix* D that abstracts the path existence relationship between pointers, an *interference matrix* I that abstracts the accessibility of a common object from two given pointers, and a *shape attribute* S that is the shape estimation. These abstractions are computed at each statement to estimate the possible shapes of overall recursive data structures that are accessible by pointers. In order to estimate the shapes of traversal patterns, a separate set of the three abstractions will be created at each statement. Specifically, in addition to the set of shape abstractions $\langle D, I, S\rangle$ for the overall recursive data structures, another set of shape abstractions $\langle D_p, I_p, S_p\rangle$ will be computed for traversal patterns. Therefore, the shape analysis at each statement can be formulated by the following forward data flow equations

$$\langle D, I, S\rangle[N]_{in} = \bigsqcup_{d \in pred(N)} \langle D, I, S\rangle[d]_{out} \tag{7}$$

$$\langle D_p, I_p, S_p\rangle[N]_{in} = \bigsqcup_{d \in pred(N)} \langle D_p, I_p, S_p\rangle[d]_{out} \tag{8}$$

$$\langle D, I, S\rangle[N]_{out} = F_N(\langle D, I, S\rangle[N]_{in}) \tag{9}$$

$$\langle D_p, I_p, S_p\rangle[N]_{out} = F_N^p(\langle D_p, I_p, S_p\rangle[N]_{in}) \tag{10}$$

where $\langle D, I, S\rangle[N]_{in}$ and $\langle D_p, I_p, S_p\rangle[N]_{in}$ are the sets of abstractions at the entry of statement N, $\langle D, I, S\rangle[N]_{out}$ and $\langle D_p, I_p, S_p\rangle[N]_{out}$ are the sets of abstractions at the exit of statement N, $\bigsqcup$ is the meet operation at merging points, and F_N and F_N^p are the transfer functions of statement N [11, 19].

4.3 Example

The traversal-pattern-sensitive shape analysis algorithm can be performed to estimate the possible shape of the recursive data structure that is created and traversed in the example shown in Figure 9. It first identifies the links that are traversed by S14 are the traversal patterns, which are defined by statement S4. The links that are traversed at S13 and defined at S5 are reference links. The algorithm then creates a set of shape abstractions $\langle D, I, S\rangle$ at each statement to estimate the shapes of overall structure, and the other set of abstractions $\langle D_p, I_p, S_p\rangle$ to store the shape information of the traversal patterns.

C	Build a doubly-linked list	C	Forward traverse the list
S1:	$list_1$ = new()	S11:	ptr_1 = $list_2$.next
S2:	do i = 1, N	S12:	do while (ptr_1.next)
	$list_2 = \phi$ ($list_1$, $list_3$)		$ptr_2 = \phi$ (ptr_1, ptr_3)
S3:	node = new()	S13:	prv = ptr_2.prev
S4:	node.value = $\cdots$	S14:	ptr_2.result += prv.value
S5:	node.next = $list_2$	S15:	ptr_3 = ptr_2.next
S6:	$list_2$.prev = node	S16:	end do
S7:	$list_3$ = node		
S8:	end do		

Figure 9: Build and Traverse a Doubly Linked List

Figure 10 shows the two sets of shape abstractions at end of first loop, i.e. S7. The first set of shape abstractions $\langle D, I, S\rangle$, as shown in Figure 10(a), shows the shape of overall recursive data structure that is created by the first loop of the example in Figure 9 is cyclic. On the other hand, the other set of shape abstractions $\langle D_p, I_p, S_p\rangle$, as shown in Figure 10(b), estimates the shape of the traversal pattern is a tree. In other words, the structure that is

D	list$_1$	list$_2$	list$_3$	node
list$_1$	1	1	1	1
list$_2$	1	1	1	1
list$_3$	1	1	1	1
node	1	1	1	1

I	list$_1$	list$_2$	list$_3$	node
list$_1$	1	1	1	1
list$_2$	1	1	1	1
list$_3$	1	1	1	1
node	1	1	1	1

S	
list$_1$	Cycle
list$_2$	Cycle
list$_3$	Cycle
node	Cycle

(a) $\langle D,\, I,\, S\rangle[S7]$

D_p	list$_1$	list$_2$	list$_3$	node
list$_1$	1	1	0	0
list$_2$	1	1	0	0
list$_3$	1	1	1	1
node	1	1	1	1

I_p	list$_1$	list$_2$	list$_3$	node
list$_1$	1	1	1	1
list$_2$	1	1	1	1
list$_3$	1	1	1	1
node	1	1	1	1

S_p	
list$_1$	Tree
list$_2$	Tree
list$_3$	Tree
node	Tree

(b) $\langle D_p,\, I_p,\, S_p\rangle[S7]$

Figure 10: The Shape Abstractions at Statement S7

specified by the traversal pattern of the second loop is in fact a singly linked list. This example demonstrates that if the traversal patterns of a program with cyclic recursive data structures are acyclic, traversal-pattern-sensitive shape analysis can differentiate the acyclic structures that are accessed by the traversal patterns from the overall cyclic recursive data structures. This information will be useful for dependence analysis and parallelization on programs with cyclic recursive data structures.

5 Dependence Analysis

Dependence analysis will be performed to identify if iterations of loops are independent. The iterations of a loop can be executed in parallel if there are no structure access conflicts and data dependence between iterations. Specifically, let S be the statements of the loop body and let $S^{(i)}$ specify the ith iteration and let $Read[S^{(i)}]$ and $Write[S^{(i)}]$ be the set of read and write references of $S^{(i)}$ respectively, then the condition for dependence test of a loop is

$$\bigcup_{i \neq j} \left[\begin{array}{l} Write[S^{(i)}] \,\cap\, (Read[S^{(j)}] \,\cup\, Write[S^{(j)}]) \,\cup \\ Write[S^{(j)}] \,\cap\, (Read[S^{(i)}] \,\cup\, Write[S^{(i)}]) \end{array} \right] = \emptyset \qquad (11)$$

The results of traversal-pattern-sensitive shape analysis will be used to facilitate dependence analysis on programs with cyclic graphs. If the traversal pattern of a loop is cyclic, then the loop will be assumed sequential. On the other hand, if the traversal pattern is a tree-like structure, then iterations of a loop will visit distinct locations. Similarly, if the traversal pattern is a DAG and the loop traverses the graph through one direction, iterations of a loop will visit distinct locations. Therefore, once it can be determined that iterations of the loop visit distinct locations, access conflicts can be identified by Equation 11.

6 Experimental Results

The algorithms that are presented in this paper have been implemented on the *ParaScope* parallel programming environment developed at Rice University [7]. The shape analysis implementation will estimate possible shapes of overall graphs and traversal patterns of the following programs: Barnes-Hut [3], EM3D [23], and Moldyn [24]. Table 1 presents the shapes of overall structures and traversal patterns of these programs in the first section, and shows the estimated shapes by the shape analysis algorithm in the last section.

	Exact Shapes		Estimated Shapes	
Program	Overall Graphs	Traversal Patterns	Overall Graphs	Traversal Patterns
Barnes-Hut	*DAG*	*Tree*	*Cycle*	*Tree*
EM3D	*Cycle*	*Tree*	*Cycle*	*DAG*
Moldyn	*Cycle*	*Tree*	*Cycle*	*DAG*

Table 1: Results of Traversal-Pattern-Sensitive Shape Analysis

The recursive data structures that are built by these programs are either DAGs or cyclic graphs, while the traversal patterns of these programs are all tree-like structures. The shape analysis algorithm correctly infers the shapes of overall data structures for 2 examples, except for Barnes-Hut. On the other hand, it makes right estimations for primary traversal structures of Barnes-Hut, and overestimates the connections of the other two programs, due to the reason that the algorithm can not make correct estimation if data structures are built by inserting new nodes between existing nodes. However, if the data structures are built by appending new nodes after existing nodes, the shape analysis algorithm will estimate correctly that primary traversal structures of EM3D and Moldyn are both trees.

Program	Independent Loops	Parallelism Coverage /Total
Barnes-Hut	3	96.0%
EM3D	2	97.7%
Moldyn	4	90.4%

Table 2: Results of Dependence Analysis

The dependence analysis algorithm uses the results of shape analysis to analyze access conflicts between the sets of read and write references of statements in each loop. Although the traversal-pattern-sensitive algorithm is not precise, the results of the shape analysis can still be used by dependence analysis to identify parallelism in the these examples. Table 2 presents the the number of independent loops that are identified by dependence analysis for each program. It shows that the dependence analysis implementation can identify independent loops in Barnes-Hut, EM3D, and Moldyn. The last column of Table 2 lists the percentages of computation times of these loops over total execution times in sequential code. It shows the dependence analysis detects parallelism in the computational intensive part of these examples, ranging from 90.4% to 97.7% of total execution times.

The independent loops that are identified in these examples will be parallelized following the transformation shown in Figure 11, while the rest sequential statements will be executed by a single processor. In order to exploit parallelism, list traversal operations in the loop will be moved into a critical region to achieve mutual exclusion. For example, a

(a) Original Loop	(b) Transformed Loop

	if (myproc().eq.0) p = list
	call gsync()
	call lock()
	plocal = p
	if (p) p = p.next
	call unlock()
	call unlock()

p = list
do while (p)
   call proc(p)
   p = p.next
end do
	do while (plocal)
	   call proc(plocal)
	   call lock()
	   plocal = p
	   if (p) p = p.next
	   call unlock()
	end do

Figure 11: Loop Transformation for Parallelization

sequential loop such as the loop in Figure 11(a) can be transformed into a parallel loop, as shown in Figure 11(b). The pointer p is a global pointer, which will be initialized by processor 0. Every processor has to pass the critical regions that are enclosed by $lock()$ and $unlock()$ functions before executing the procedure $proc$ on the node pointed by $plocal$.

The following tables are the timing results of these parallelized programs on SGI Power Challenge. The execution times in the following tables are the numbers of seconds that the programs have spent executing in user mode and in system mode. Table 3 shows the execution times of Barnes-Hut from 1 processor to 16 processors for 2 time units with time steps 0.025 units (i.e. total 81 iterations). The numbers of particles are 8K, and 16K, respectively. The speedup of 8K and 16K cases on 16 processors are 9.46 and 9.76, respectively.

Processors	1	2	4	8	16
8192	612.87	321.65	174.49	101.31	64.79
16384	1429.50	748.49	403.00	231.76	146.54

Table 3: Timing Results of Barnes-Hut

Table 4 lists of the execution times of EM3D with 1024 E nodes and 1024 H nodes for 500 iterations. The degree of the bipartite graph is 50, i.e. each E node has 50 neighboring H nodes and each H node connects to 50 neighboring E nodes. The second row shows the time to create the bipartite graph by a single processor, and the third row lists the execution time of computing electromagnetic forces for 500 iterations from 1 processors to 16 processors. The total execution times are presented in the last row. The speedup on 16 processors is 7.40. EM3D does not achieve the same speedup for two reasons: (1) graph construction is performed by a single processor while other processors are idle, and

(2) it is not computational intensive — each iteration of the inner loop involves only one floating-point multiplication.

Processors	1	2	4	8	16
Build Graph	2.66	2.75	2.75	2.76	2.76
Compute E & H	114.78	62.36	33.30	18.80	13.12
Total	117.44	65.11	36.05	21.56	15.88

Table 4: Timing Results of EM3D with 1024 E Nodes and 1024 H nodes

Table 5 shows the execution times of 2 cases of Moldyn on different numbers of processors for 150 time steps. The 729 particles in the system are initially uniformly distributed on a $9 \times 9 \times 9$ lattice in both cases, and the cutoff radius for force computation is 6. The first case updates the neighbor lists every 30 time steps, and hence the neighbor lists are updated 5 times during the execution. In order to reduce the error caused by the situation that particles outside cutoff radius move within the range before the neighbor lists are updated, all neighboring particles within $1.2 \times$ cutoff will be included in the neighbor list. The second case updates the neighbor lists every 50 iterations and neighboring particles within $1.3 \times$ cutoff are linked in the neighbor lists. The speedup for these two cases are 5.0 and 6.9, respectively. Moldyn does not scale well because of the periodic regeneration of neighbor lists, which is an order $\mathcal{O}(N^2)$ operation.

Processors	1	2	4	8	16
Update List every 30 iterations, $1.2 \times$ cutoff					
Total	250.44	144.34	88.15	61.51	50.01
Build Neighbor List	24.11	29.34	29.59	30.10	30.81
Update List every 50 iterations, $1.3 \times$ cutoff					
Total	252.30	132.57	75.39	48.53	36.72
Build Neighbor List	13.60	16.45	16.59	16.83	17.20

Table 5: Timing Results of Moldyn with $9 \times 9 \times 9$ Lattice

7 Related Work

Dependence analysis techniques have been developed as the core of optimizing/parallelizing compilers [2, 27, 28]. They use integer programming to determine if two array subscript expressions would result in referencing overlapping regions of arrays. In order to identify parallelism and opportunities of optimizations on programs with pointers and recursive data structures, dependence analysis techniques combined with pointer analysis have been proposed to handle pointers and dynamic recursive data structures [13, 16, 17, 18, 22].

Horwitz et al. develop an algorithm to determine dependence by detecting interferences in reaching stores [17], while Larus and Hilfinger propose to identify access conflicts on alias graphs [22]. In contrast to these approaches, Hendren and Nicolau use path matrixes to record connection information among pointers and present a technique to recognize interferences between computations for programs with acyclic structures [16]. The approach proposed by Guarna, Jr. determines dependence between patterns of traversal using syntax tree matching [13]. More recently, Ghiya and Hendren have proposed techniques to compare read/write sets of memory locations via pointer indirections and summarize the effect of pointer references over regions of programs [10, 12]. The techniques can facilitate compilers to perform optimizations and detect common patterns of parallelism in C programs with recursive data structures.

The above dependence analysis techniques can identify parallelism from programs with acyclic recursive data structures. If the overall structures are cyclic, they would assume the programs are sequential even though parallelism indeed exists. The dependence analysis technique proposed in this paper takes advantage of the information computed by traversal-pattern-sensitive shape analysis to first differentiate the acyclic structures of traversal patterns from the overall cyclic structures and then determine if parallelism exists. The dependence analysis method proposed by Hummel et al. can handle programs with cyclic graphs, but it relies on alias information provided by users [18].

An alternative dependence analysis approach that can simplify pointer analysis has been proposed by Hwang and Saltz [20]. Each reference pattern can be represented by a path expression under the assumption that each unique path leads to a distinct location, and hence conflict analysis can be done by comparing path expressions The outcome of conflict analysis will be forwarded to guide traversal-pattern-sensitive pointer analysis. This approach can simplify pointer analysis since if statements are deemed to be dependent by this approach, they are inherently sequential regardless of the shapes of the recursive data structures they traverse. Consequently, there is no need to perform pointer analysis on the statements that construct such data structures.

Side effect analysis is another approach to identify parallelism for programs with pointers, and algorithms combined with pointer analysis have been proposed by Choi et al. and Landi et al. to gather interprocedural side effects that are crucial for optimizations and dependence analysis [6, 21]. Commutativity analysis proposed by Rinard and Diniz is an approach that is different from dependence analysis to identify parallelism in programs with recursive data structures [25]. It is designed for objected-based programs and can discover when operations of objects commute. This approach identifies a different type of parallelism — it identifies the independent operations of different branches of tree-like data structures.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993. In Special Section on Languages and Compilers for Parallel Machines.

[3] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[4] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *SIGPLAN Notices*, 23(7):47–56, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[5] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[6] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[7] Keith D. Cooper, Mary W. Hall, Robert T. Hood, Ken Kennedy, Kathryn S. McKinley, John M. Mellor-Crummey, Linda Torczon, and Scott K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993. In Special Section on Languages and Compilers for Parallel Machines.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[10] Rakesh Ghiya and Laurie J. Handren. Putting pointer analysis to work. In *Conference Record of POPL '98: 25nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

[11] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

[12] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in c programs with recursive data structures. In *Proceedings of the 1998 International Conference on Compiler Construction*, March 1998.

[13] Vincent A. Guarna, Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of 1988 International Conference on Parallel Processing, Volume 2*, pages 212–220, August 1988.

[14] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[15] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell University, 1990.

[16] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[17] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. *SIGPLAN Notices*, 24(7):28–40, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[18] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. *SIGPLAN Notices*, 29(6):218–229, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[19] Yuan-Shin Hwang. *Interprocedural Definition-Use Chains of Dynamic Recursive Data Structure*. PhD thesis, University of Maryland, 1998.

[20] Yuan-Shin Hwang and Joel Saltz. Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In *Proceedings of 10th International Workshop on Languages and Compilers for Parallel Computing*, University of Minnesota, August 1997. Available via anonymous ftp directory hyena.cs.umd.edu:/pub/shin/papers/lcpc97.ps.gz.

[21] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[22] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[23] N.K. Madsen. Divergence preserving discrete surface integral methods for maxwel l's curl equations using non-orthogonal grids. Technical Report 92.04, RIACS, February 1992.

[24] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79. ACM Press, July 1995. ACM SIGPLAN Notices, Vol. 30, No. 8.

[25] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. *SIGPLAN Notices*, 31(5):54–67, May 1996. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[26] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, January 1996.

[27] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

[28] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. ACM Press, New York, NY, 1991.