Detecting Parallelism in C Programs with Recursive Data Structures*

Rakesh Ghiya, Laurie J. Hendren and Yingchun Zhu
School of Computer Science, McGill University
Montreal, CANADA H3A 2A7
{ghiya,hendren,ying}@cs.mcgill.ca

Abstract. In this paper we present techniques to detect three common patterns of parallelism in C programs that use recursive data structures. These patterns include, function calls that access disjoint sub-pieces of tree-like data structures, pointer-chasing loops that traverse list-like data structures, and array-based loops which operate on an array of pointers pointing to disjoint data structures. We design dependence tests using a family of three existing pointer analyses, namely points-to, connection and shape analyses, with special emphasis on shape analysis. To identify loop parallelism, we introduce special tests for detecting loop-carried dependences in the context of recursive data structures. We have implemented the tests in the framework of our McCAT C compiler, and we present some preliminary experimental results.

1 Introduction and Motivation

This paper focuses on detecting three common patterns for parallel computations that use recursive data structures: (1) function-call parallelism including parallel recursive calls on tree-like structures; (2) *forall* parallelism for loops traversing arrays of list/tree-like structures; and (3) *foreach* parallelism for loops traversing list/tree-like structures, which is similar to *doacross* parallelism.

In order to safely detect these patterns of parallelism in C programs, one must deal with dependences due to scalars, dependences due to pointers to stack-allocated objects (*stack-directed pointers*), and dependences due to pointers to heap-allocated objects (*heap-directed pointers*). Thus, our approach uses the results of the family of pointer analyses that have been implemented in the McCAT optimizing/parallelizing C compiler: *points-to analysis*[1], *connection analysis*[2] and *shape analysis*[3]. Points-to analysis is used to detect dependences due to scalars and stack-directed pointers, while connection and shape analysis are used to detect dependences due to heap-directed pointers.

The main focus of this paper is not the pointer analyses themselves, but rather how we can use the results of the analyses to detect parallelism. The remainder of the paper is structured as follows. In Section 2 we introduce the three parallelism patterns in more detail. In Section 3 we describe the overall setting of our approach, and present the rules to detect function call parallelism.

* This research supported in part by NSERC and FCAR.

We provide rules for safely identifying loop parallelism in Section 4. Section 5 gives some preliminary empirical results indicating how often we can successfully identify the patterns. Section 6 discusses related work, and Section 7 gives conclusions and future work.

2 Parallel Patterns

The focus of our approach is on detecting coarse-grain parallelism in the context of function calls and loops, that perform computation on heap-based recursive data structures. The three patterns we want to identify are illustrated in Figure 1. These patterns typically arise in programs using recursive data structures. Below, we discuss the parallelism opportunities they offer.

```
void treeAdd(tree *t)
{
  if (t == NULL)
   return;
  Q: tl = t->left;
  L: treeAdd(tl);
  M: tr = t->right;
  N: treeAdd(tr);
     t->i = tl->i +
            tr->i;
}
```

(a) function-call

```
for (i = 0; i < N; i++)
{
  t = list_arr[i];
  compute(t, x, y);
}
```

(b) `forall`

```
while (lp != NULL)
{
  S: lp->x = lp->y * 5;
  T: lp->y = lp->x * 6;
  U: lp = lp->next;
}
```

(c) `foreach`

Fig. 1. Parallelism Patterns

Function-call parallelism:

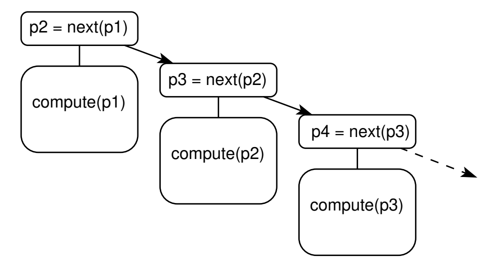
In Figure 1(a), the two calls to the function `treeAdd`, respectively perform the addition for the left and right sub-trees of the tree pointed to by the pointer `t`. If the sub-trees are disjoint, the two function calls access disjoint regions of the heap, and can be executed in parallel.

forall **parallelism:**

Figure 1(b) shows an array-based loop. However, the array `list_arr` is an array of pointers, with each pointer pointing to a heap data structure (a list). If each pointer points to a disjoint heap data structure, then each call to `compute` accesses a disjoint heap region, and the loop can be fully parallelized, with all iterations executed in parallel.

foreach **parallelism:**

Figure 1(c), shows a loop traversing a linked list. The loop body consists of two parts: one that does the *computation* on the list elements, and the second that performs the *navigation* through the list. The computation part is formed by the statements `S` and `T` in the loop, while the navigation part includes the statement `U: lp = lp->next`. The pointer used to navigate through the list (`lp`), is termed as *navigator*. The parallelism in this loop arises from the fact



(a) Staggered Execution

```
while (lp != NULL)
{ for(i=0; lp != NULL &&
        i<max_proc; lp=lp->next)
  { ptr[i] = lp; i = i + 1;
  }
  forall(j = 0; j < i; j++)
  { lp = ptr_arr[j];
    lp->x = lp->x * 5;
    lp->y = lp->y * 6;
  }
}
```

(b) *foreach* to *forall*

Fig. 2. Parallel Execution of a *foreach* Loop

that each loop iteration visits a disjoint node in the list. However, this loop cannot be considered a *forall* loop, because its iterations cannot be executed in parallel. The loop contains an intrinsic *loop-carried* dependence due to the navigator. The navigator for the next iteration is obtained via the navigator for the current iteration. We call these loops as *foreach* loops.

The parallelism in a *foreach* loop can be extracted by executing it in a *staggered* fashion as shown in Figure 2(a). Here, first the navigator for the next iteration is obtained. Subsequently, the next iteration can start before the first iteration completes, and the computation phases of the two iterations can overlap. Alternatively, if the navigation overhead of the loop is minimal compared to the computation performed, the navigators could first be stored in an array of pointers of size `max_proc`, where `max_proc` is the number of processors being used, via a separate pointer-collecting loop. The original loop can then be executed as a *forall* loop. This technique is illustrated in Figure 2(b).

3 Dependence Testing Framework for Function Call Parallelism

To identify if two function calls can be executed in parallel, we need to detect if there is a dependence between the statements containing them. To this end, we have developed a general dependence test that checks if a dependence exists between any two given statements in a function. The overall algorithm for the test *depTest* is outlined in Figure 3. It has been implemented at the high level SIMPLE intermediate representation of the McCAT C compiler [14].

The test *depTest* proceeds in a hierarchical fashion. Given two statements, *stmtS* and *stmtT*, and the type of dependence to be detected (flow, anti or output), it first applies the *stackTest* to disambiguate direct/indirect references to the stack. This test uses the results of points-to analysis [1], which estimates targets of stack-directed pointers as points-to triples of the form *(ptr, target, possible/definite)*. If the dependence cannot be disproved, the test then checks if the dependence is only due to heap accesses. In this case, heap analysis information is used. First, the test *connectionTest* is applied. This test uses connection-based heap read/write sets to identify if the two statements access heap locations belonging to disjoint heap data structures [2, 4]. If the test succeeds, statements are reported independent. Otherwise the test *shapeTest* is invoked to further identify if the statements access disjoint sub-pieces of a data structure. This is the focus of this paper. Detailed description of the first two tests can be found in [4, 5].

```
fun depTest(stmtS, stmtT, depType) =
 if (stackTest(stmtS, stmtT, depType) == NoDep)
  return NoDep; /* no dependence */
 /* use heap analyses if dependence only due to heap */
 else if (stackTest(stmtS, stmtT, depType) == OnlyHeap)
  if (connectionTest(stmtS, stmtT, depType) == NoDep)
   return NoDep; /* access disjoint heap data structures */
  if (shapeTest(stmtS, stmtT, depType) == NoDep)
   return NoDep; /* access disjoint pieces of a data structure */
  return Dep;/* heap dependence cannot be disproved */
 else/* stack dependence is not only due to the symbolic heap location */
  return Dep; /* dependence cannot be broken */
```

Fig. 3. Checking if two Statements are Dependent

The test *shapeTest* uses shape analysis information [3]. Shape analysis estimates the shape of the data structure accessible from a given heap-directed pointer: is it tree-like, DAG-like or a general graph containing cycles? Knowledge about the shape of the data structure accessible from a heap-directed pointer, provides crucial information for disambiguating heap accesses originating from it. For a pointer `p`, if `p.shape` is *Tree*, then any two accesses of the form `p->f` and `p->g` will always lead to disjoint subpieces of the tree (assuming `f` and `g` are distinct fields). If `p.shape` is *DAG*, then two distinct field accesses `p->f->f` and `p->g` can lead to a common heap object. However, if a dag-like structure is traversed using a sequence of links, every subsequence visits a distinct node. This information can be used to disambiguate heap accesses in different iterations of a loop, or different recursive calls, traversing such a data structure, as discussed in the following sections.

3.1 Shape Dependence Test

The shape dependence test relies on the shape of the data structure being traversed, and the *access paths* used to reach particular node(s) in the data structure. The access paths are computed with respect to a given node in the data structure, pointed to by the *anchor* pointer. An *anchor* pointer is a pointer that points to a fixed node in the data structure over the program region starting from the statement that defines it, and ending at the statement `T`, where a dependence is being checked from some statement `S` to statement `T`. Once the access paths are computed with respect to the *anchor*, dependence is resolved by checking if starting from the *anchored* node, the two paths can lead to the same node, in view of the shape of the data structure.

The shape dependence test is outlined in Figure 4. It first collects the sets of pointers *pSetS* and *pSetT* that access the heap, respectively for *stmtS* and *stmtT*. These sets are computed using the points-to based read/write set information. The shape dependence test is performed on each pair of pointers (*ptrS*, *ptrT*) from the two sets, and no dependence is reported only if the test succeeds on each pair. For a given pair, if the shape attribute of either pointer is found to be cyclic, a dependence is reported and the test terminates. If the shape attributes are acyclic, anchor-based access paths are constructed, as explained below.

```
fun shapeTest(stmtS, stmtT, depType) =
 [pSetS, pSetT] = heapAccessPtrs(stmtS, stmtT, depType);
 foreach pair (ptrS, ptrT) ∈ pSetS×pSetT
   if (isCycle(ptrS.shape) or isCycle(ptrT.shape))
     return DEP; /* Cyclic data structures */
   [defS, defT] = getDefChains(ptrS, stmtS, ptrT, stmtT);
   anchorStmt = findAnchor(defS, defT);
   if (anchorStmt != NULL)
     anchor = getRef(anchorStmt, lhs);
   if (!anchor or isCycle(anchor.shape))
     return DEP; /* cannot find anchor or its shape is cyclic */
   [pathS, pathT] = getPathExprs(anchor, defS, defT);
   if (fieldsUpdatedBetween(pathS, stmtS, pathT, stmtT, anchorStmt))
    return DEP; /* structural modification involved */
   if (checkPathExprs(pathS, stmtS, pathT, stmtT, anchor) == DEP)
     return DEP; /* path exprs indicate a possible conflict */
 return NODEP; /* no dependence detected */
```

Fig. 4. Shape Dependence Test

Constructing Access Paths using Definition Chains

The first step in the construction of access paths is computation of definition chains *defS* and *defT* for the pointers *ptrS* and *ptrT*. Definition chains are constructed by recursively traversing the reaching definitions of the given pointers, as illustrated by the following example. The complete algorithm is presented in [5].

Consider the construction of the definition chain for the use of pointer `tr` at the function call statement `N: treeAdd(tr)` in Figure 1(a). The definition that reaches this use is from the statement `M: tr = t->right`. So this statement is put in the definition chain. Next, the traversal looks for definitions that reach the use of pointer `t` at the statement `M`. In this case the definition comes from the function header, which is appended to the definition chain. Since the traversal cannot proceed any further up, it stops. Similarly, the definition chain for the use of pointer `tl` at statement `L` would consist of the statement `Q` and the function header.

In general, the definition chain traversal stops when either it has reached the function header, or it cannot find a *unique* definition that *definitely* reaches the given use [5]. This ensures that we construct only one *definite* access path for a given pointer, and not a set of *possible* paths. This is done for efficiency reasons, as comparing a set of paths can be expensive and further is less likely to disprove a dependence.

Finding a Common Anchor

Once the definition chains are constructed, the next step is to find an *anchor* pointer, with respect to which both the pointers under consideration can be defined. In our example above, the pointer `t` can be considered as an anchor pointer, as both pointers `tl` and `tr` can be defined in terms of `t`. This is infered from the fact that both the pointers have a common definition of the pointer `t` in their definition chains. Now, using the function header definition of `t` as the anchor, the definition chains of pointers `tl` and `tr` are traversed to construct their respective access paths with respect to the anchor `t`, giving the paths `t->left` and `t->right`. The detailed algorithms for finding the anchor and constructing the access paths can be found in [5].

Comparing Access Paths for Dependence Detection

The access paths are given to the function *checkPathExprs*, which detects if they definitely lead to disjoint parts of the data structure. Note that *shapeTest* reaches this function only if the shape of the data structure being traversed is Tree or DAG, and the traversal fields are not being modified. The input to the function *checkPathExprs* consists of two statements, *stmtS* and *stmtT*, and the respective access paths, *pathS* and *pathT*, expressed with respect to the anchor pointer. The function uses three operations to compare the access paths:

equivPaths(pathS, pathT): This function checks if the two access paths are equivalent, i.e., consist of the same sequence of field accesses. For example the access path `t->left->right` is equivalent to the access path `t->left->right`, but *not equivalent* to the access paths `t->left` or `t->left->left`. Two equivalent access paths always lead to the same node.

subPath(pathS, pathT): This function checks if the access path *pathS* is a proper sub-path of the access path *pathT*, i.e., *pathS* contains k field accesses less than *pathT*, and is equivalent to the access path obtained by removing the last k field accesses from *pathT*, where $k \geq 1$. For example, the access path `t->left` is a proper sub-path of the access path `t->left->right`, but not of the access paths `t->left` or `t->right`. For acyclic data structures (anchor.shape is Tree or DAG), if an access path is a proper sub-path of another path, the two paths lead to disjoint nodes. Further, the node reached via the former path *cannot* be accessed from the node reached via the latter path.

disjointPaths(pathS, pathT): This function checks three conditions: (i) the access path *pathS* is not equivalent to the access path *pathT*, (ii) *pathS* is not a proper sub-path of *pathT*, and (iii) *pathT* is also not a proper sub-path of path *pathS*. For example, `t->left` and `t->right` are disjoint paths, while `t->left` and `t->left->right` are not. For tree-like data structures (anchor.shape is Tree), disjoint paths not only lead to disjoint nodes, but also one node cannot be accessed from the other. Thus disjoint paths lead to disjoint sub-pieces of tree-like data structures.

If the data structure is DAG-like (anchor.shape is DAG), disjoint paths can lead to a common node. For example, if `left` and `right` links of the node pointed to by the anchor pointer `t`, point to the same node (giving a DAG), the disjoint paths `t->left` and `t->right` will lead to the same node.

```
fun checkPathExprs(pathS, stmtS, pathT, stmtT, anchor) =
 case type(stmtS) of
   < CallStmt > =>
     case type(stmtT) of
       < CallStmt > =>
         if (isDag(anchor.shape))
           return DEP; /* DAG shape is not useful with two call stmts */
         else /* shape is Tree */
           return(disjointPaths(pathS, pathT));
       < SimpleStmt > =>
         if(subPath(pathT, pathS))
           return NODEP; /* sufficient condition */
         if (isTree(anchor.shape))
           return(disjointPaths(pathS, pathT));
   < SimpleStmt > =>
     case type(stmtT) of
       < CallStmt > =>
         if(subPath(pathS, pathT))
           return NODEP;
         if (isTree(anchor.shape))
           return(disjointPaths(pathS, pathT)); /* need not be a subpath */
       < SimpleStmt > =>
         if (isTree(anchor.shape))
           return(!equivPaths(pathT, pathS));
         else /* anchor shape is DAG : one is subpath of another */
           return(subPath(pathT, pathS) or subPath(pathS, pathT));
```

Fig. 5. Comparing Access Paths for Dependence Detection

Different Cases for Dependence Detection

We now discuss the different cases and the associated disambiguation rules given in the function *checkPathExprs* in Figure 5. We have four cases depending on if the two statements, *stmtS* and *stmtT*, are call statements (contain a function call) or simple statements (do not contain a function call).

Case 1: Both statements are call statements

As call statements can access whole sub-pieces of a data structure, two call statements can be independent only if the access paths lead to disjoint sub-pieces of the data structure. This requires that the two access paths are disjoint and the shape attribute of anchor is Tree, as shown in the function *checkPathExprs* (Figure 5). This is the case for our example based on the `treeAdd` function shown in Figure 1(a). The two access paths are `t->left` and `t->right`, which are disjoint, and the shape attribute of the anchor pointer `t` is Tree. Thus the two calls to `treeAdd` are independent and can be executed in parallel.

Case 2: stmtS is a call statement and stmtT is a simple statement

Here *stmtS* can access a sub-piece of the data structure, while *stmtT* can access only fields in a specific node of the data structure. This is because in our SIMPLE representation, a statement can only have one level of indirection. Thus for the two statements to be independent, *pathS* should lead to a node, which is disjoint from the node corresponding to *pathT*, and also cannot reach the latter node. This can be guaranteed if *pathT* is a proper sub-path of *pathS*. The shape attribute of the anchor can be either Tree or DAG in this case. For example, if *pathS* is `t->left` and *pathT* is simply `t`, the proper sub-path condition is satisfied, and *stmtS* cannot access the node accessed by *stmtT*. If shape attribute of the anchor is Tree, the statements will be independent also for the case when *pathS* and *pathT* are disjoint. Note that **Case 3** is analogous to **Case 2**, with `stmtS` as simple statement, and `stmtT` as call statement.

Case 4: Both statements are simple statements

In this case both statements can respectively access only some specific node of the data structure. We simply need to check that *pathS* does not lead to the same node as *pathT*. This condition is satisfied, if *pathS* is not equivalent to *pathT*, when shape attribute for the anchor is Tree. With DAG attribute, we need to check for the stronger condition that one of the access paths is a proper sub-path of another.

For example, let *pathS* be `t->right->left` and *pathT* be `t->left->right`. The access paths are not equivalent. With shape attribute as Tree, the two access paths cannot lead to the same node. With DAG attribute they can lead to the same node. However, the access paths `t->left` and `t->left->right` can be proven to lead to disjoint nodes even with DAG attribute. This is because the common sub-path in the paths leads them to the same node, and then the additional field access in the latter path, leads it to a distinct node as the data structure is acyclic.

Above, we have described our overall strategy for detecting dependence between *two given statements* in a function, and discussed in detail how to use shape information for dependence testing. We use the test (*depTest*) during the DDG (data dependence graph) construction phase of the EARTH-McCAT compiler [15]. The DDG is then used to identify statements/function calls that can be executed in parallel, and to partition the program into threads.

4 Loop Parallelism

In this section, we present techniques to identify loop-level parallelism, in the form of *forall* and *foreach* loops traversing recursive heap data structures. For finding loop parallelism, we need to detect the presence of *loop-carried* dependences (henceforth refered to as LCDs). Two statements in a loop have an LCD, if a memory location accessed by one statement in a given iteration, is accessed by the other statement in a future iteration, with one of the accesses being a write access.

The presence of LCDs in a loop indicates that its iterations are not independent, and hence cannot be executed in parallel. Our particular focus is on finding heap-based *forall* and *foreach* loops (described in section 2). A *forall* loop should not have any LCD, while a *foreach* loop can only involve an LCD with respect to the navigator. Considering these constraints, we first describe the method for detecting *foreach* loops, and then explain how it can be adapted to detect *forall* loops. Given a loop, we identify if it is a heap-based *foreach* loop, using the steps explained in the following subsections.

4.1 Good Loop Detection

This is a pre-processing step, which detects potential heap-based *foreach* loops in the program. It is required so that we do not incur the overhead of detecting LCDs for each loop in the program. The criteria used to label a loop as a good loops are as follows: (i) the loop body involves read/write accesses to heap locations, (ii) the loop body is free from irregular control flow constructs such as `break`, `continue`, or `return` statements, or system calls such as `exit` and `abort`, and thus control can exit the loop only from the *loop condition test*; and (iii) a navigator can be identified for the loop. The second condition is required to ensure the correctness of the parallelizing transformations illustrated in Figure 2, as they assume that the loop does not terminate prematurely. The third condition is required to detect that the loop actually *navigates* a recursive heap data structure in a regular fashion.

Identifying the Navigator

The overall algorithm for identifying the navigator is outlined in in Figure 6. The process is closely related with the variables used in the *loop condition test*. For a given variable in the loop test, say `testVar`, the function *findNavigator* proceeds as follows. First, a definition chain is constructed for the use of `testVar` in the loop test. The function *getLoopDefChain* is used for this purpose. This function is similar to the function *getDefChain* defined in Figure 4. However, it only considers the definitions that arise from a statement within the given loop (loop-resident definitions). It terminates when either it cannot find a *unique* loop-resident definition that *definitely* reaches the given use, or it encounters a loop-resident definition for the second time [5]. In the latter case a recurrence is reported. If the traversal terminates without finding a recurrence, it returns NULL, indicating that a navigator cannot be detected.

If the function *getLoopDefChain* reports a recurrence, it indicates the presence of a variable in the loop whose value for the next iteration is defined in terms of its current value. Such a variable is a potential candidate for being a navigator. In this case, the definition chain is used to construct an access path for the loop test variable `testVar`. This access path is called the *test expression* and the base variable in the path is called the *navigator*. The test expression indicates how the `testVar` for the next loop test is obtained from the navigator for the current iteration. The component of the access path, contributed by

```
/* find the navigator for the given loop if one exists */
fun identifyNavigator(loopStmt) =
 cond = loopStmt.cond; /* loop condition test */
 navigator = findNavigator(cond.lhsvar, loopStmt);
 if (navigator != NULL) /* lhs var succeeds */
  return navigator;
 else /* try rhs var in the loop test */
  return (findNavigator(cond.rhsvar, loopStmt));

/* find the navigator for the given loop with respect to the var */
fun findNavigator(var, loopStmt) =
 defN = getLoopDefChain(var, loopStmt, loopStmt);
 if (defN == NULL || recFlag(defN) != RECUR)
   return NULL; /* no recurrence exists in the definition chain */
 pathT = getPathExpr(loopStmt.cond.var, defN); /* test expression */
 pathN = getNavExpr(pathT); /* navigator expression */
 varN = getBaseVar(pathN); /* navigator */
 if (fieldsUpdated(pathN, loopStmt, navigator))
   return NULL; /* structural modification involved */
 loopStmt.navigatorExpr = pathN;
 loopStmt.navigator = varN;
 return (varN);
```

Fig. 6. Identifying the Navigator

the definition(s) involved in the recurrence, is called the *navigator expression*. It indicates how the navigator for the next iteration is obtained in terms of the current navigator. We illustrate these concepts via an example below.

For example, consider the loop in Figure 1(c). The loop test variable in this case is the pointer `lp`. Its loop-resident definition comes from the statement `S:lp = lp->next`. So it is added to the definition chain. Next, the loop-resident definition for the use of `lp` at `S` is sought. It again happens to be the statement `S`. Here the definition chain construction terminates and a recurrence is reported. The definition chain gives the access path `lp->next`, which is the test expression for the loop, and the base pointer for the expression, `lp`, is the navigator. Here, since the navigator is identical to the loop test variable, the navigator expression is same as the test expression. More detailed examples for navigator identification can be found in [5].

4.2 Verifying the Navigator

Once a navigator is identified, and a navigator expression is constructed, the next step is to verify if the navigator visits a distinct node in the data structure in each iteration. Thus, the function *findNavigator* checks that none of the fields in the navigator expression (*navigator fields*) are updated inside the loop. For this purpose connection-based heap read/write sets are used [5]. This check ensures that the navigator advances in a regular fashion from one iteration to the next iteration, i.e., the fields along which the data structure is navigated remain static through the loop execution. Further, note that the navigator is *definitely* advanced in each iteration using the navigator expression, and *not conditionally*. From this information, we can make the following important observations.

Observation 1: If the shape attribute of the navigator is Tree or DAG, the navigator expression will lead the navigator to a distinct node in the data structure, in each iteration. In this case the data structure is acyclic, and since the navigator is advanced using the same expression every iteration (`lp = lp->next`), it cannot *revisit* a node.

Observation 2: If the shape attribute of the navigator is Cycle, the above claim still holds in an important case. This case represents loops, where the loop test involves testing a heap-directed pointer, say `ptr`, against a constant `(ptr != NULL)` or another pointer that is loop-invariant `(ptr != b)`, where `b` could be the navigator for the outer loop. Such loops typically arise in C programs using recursive data structures.

For such loops, if the navigator (`ptr`) visits a given node a second time, the loop will execute infinitely. This is because the navigator fields are not updated inside the loop body. Consider again the loop in Figure 1(c). Suppose after three iterations, its navigator `lp` visits the node it visited at the beginning of the loop. Since the navigator field `next` is not updated, `lp` will visit this node every three iterations, and the condition `(lp == NULL)` will never be satisfied, giving an infinite loop. Note that during good loop detection, we have already ensured that the only exit point for the loop is the loop condition test.

Thus, with the assumption that a loop does not run infinitely, we can infer that its navigator visits a distinct node in each iteration, for an important class of loops. These loops typically traverse parts of a cyclic data structure in an acyclic fashion: for example a loop that traverses a doubly linked list only using the `next` link. We term such loops as *acyclic loops*.

4.3 Detecting Heap-based Loop-carried Dependences

Once a valid navigator is found for a loop, we check if the loop involves any LCDs with respect to heap accesses. The function *heapLCD* in Figure 7, outlines this dependence test. It takes as input any two statements belonging to the loop, and determines if an LCD of depType (flow, anti or output) exists between them with respect to heap accesses. Given two statements from the loop, *stmtS* and *stmtT*, and the pointers they use to access the heap (*ptrS* and *ptrT*), the test first constructs the access paths for the two pointers with respect to the navigator. This is similar to constructing the access paths with respect to the anchor, for the *shapeTest*. Next, it compares the two access paths to detect if they can introduce an LCD. These access paths are termed as *navigator access paths* (NAPs). In case, the NAPs cannot be constructed, dependence is reported.

While traversing a heap data structure, an LCD can be introduced when fields of *neighbor* nodes, i.e., nodes other than the one being currently visited by the navigator, are accessed. To access the neighbor nodes via the navigator, *pointer fields* must be traversed. Thus a NAP can lead to a neighbor node only

```
fun heapLCD(stmtS, stmtT, loopStmt, depType) =
 [pSetS, pSetT] = heapAccessPtrs(stmtS, stmtT, depType);
 navigator = loopStmt.navigator;
 foreach pair (ptrS, ptrT) ∈ pSetS×pSetT
   [defsS, defsT] = getLoopDefChains(ptrS, stmtS, ptrT, stmtT, loopStmt);
   [pathS, pathT] = getPathExprs(navigator, defsS, defsT);
   if (!pathS or !pathT)
     return Dep; /* reference cannot be expressed wrt navigator */
   if (isTree(navigator.shape)) /* traversing a tree−like structure */
     if (fieldsUpdated(pathS, loopStmt, navigator) ||
         fieldsUpdated(pathT, loopStmt, navigator))
           return Dep; /* structural modification involved */
     if (navigatorFieldsUsed(pathS, pathT, loopStmt.navigatorExpr))
           return Dep; /* can access nodes from other iterations */
   else if (isAcyclic(loopStmt)) /* loop test is (ptr != someConstant) */
     if (ptrFieldsUsed(pathS, stmtS, pathT, stmtT))
           return Dep; /* can access nodes from other iterations */
   else return Dep; /* cannot check this depenence: assume dependence */
 return NoDep; /* no loop−carried dependence detected */
```

Fig. 7. Test for Loop-Carried Heap Dependences

if it involves one or more pointer fields. With this observation, we can infer that two statements can induce an LCD, only if one of the NAPs involves pointer fields. Otherwise, the NAPs lead to fields in the node currently being visited by the navigator, without introducing any LCD.

The function *heapLCD* essentially makes the above check. Additionally, it makes a weaker check if the shape attribute of the anchor is Tree. In this case, the NAPs can use pointer fields other than the navigator fields. For example, for a loop traversing a list using the pointer `lp`, if the shape attribute of `lp` is Tree, the statement `lp->hdr->num++` will not induce an LCD. This is because the header node cannot be common for any two nodes in the list, else it will violate the Tree shape attribute. However, the assignment statement `lp->next->i = lp->i` will still induce the dependence as the access path `lp->next->i` uses the pointer field `next`, which is a navigator field.

If no heap-based LCDs are detected, we flag this loop as a *foreach* loop with respect to the heap accesses. To identify it as a real *foreach* loop, we use existing tests implemented in the McCAT C compiler to check against LCDs induced by accesses to scalar variables, array references [11, 12], and stack-based indirect references [4]. If the only LCD detected is with respect to the navigator, the loop is flagged as a real *foreach* loop. Otherwise, it is flagged as a non-*foreach* loop.

4.4 Identifying *forall* Loops

To identify heap-based *forall* loops of the type shown in Figure 1(b), we use a similar strategy as for detecting *foreach* loops. The key difference is that the navigator for the *forall* loops is an integer, and the navigator expression is an integer expression. The tests for checking heap-based LCDs are modified to compare access paths which consist of array expressions as opposed to pointer references. To this end, subscript tests developed for array dependence testing are used. For example, in Figure 1(b), the navigator access path for the heap pointer `t` is computed as `list_arr[i]`. From the information that `i` is a navigator, and the shape attribute of `list_arr` is Tree, the test infers that pointer `t` accesses a disjoint list in each iteration and cannot induce an LCD. Further, if it needs to compare access paths of the form `list_arr[i + j]` and `list_arr[i + k]`, it uses subscript tests from array dependence testing [11, 12], to identify LCDs.

5 Experimental Results

We have implemented the dependence tests described in sections 3 and 4 in the framework of our McCAT C compiler. We have done a preliminary study of their effectiveness on a set of four recursive data structure based C benchmark programs. The results are summarized in Table 1.

Program	Description	Data Structures	Par Calls	*foreach* Loops	*forall* Loops	Total Loops	Total Calls
treeadd	Tree Addition	Binary Tree	4	0	0	0	4
power	Power System Opt.	k-ary Tree	0	0	4	10	25
circuit	Sparse Matrix Solver	Doubly-linked Lists	0	14	0	24	35
pug	Grid Triangulation	Interconnected Lists	0	5	0	15	34

Table 1. Benchmark Results

For the *treeadd* benchmark, the test *depTest* finds two pairs of parallel calls respectively to the functions `buildTree` and `treeAdd`. The *forall* loops in the *power* benchmark are detected using the *heapLCD* test. These loops iterate on arrays of pointers to tree data structures, and form the most compute-intensive part of the code. The benchmarks *pug* and *circuit* use cyclic data structures, but perform majority of their computation inside *acyclic* list-traversing loops, which are detected as *foreach* loops by the *heapLCD* test. Finally, the hand-written Earth-C [10] versions of the benchmarks *treeadd* and *power*, that *only* use the parallelism detected by our dependence tests, respectively obtain speed-up by factors of 16 and 12 on the EARTH-MANNA multithreaded machine [10] using 16 processors. We are presently working on analyzing and collecting runtime performance improvement statistics for a larger set of benchmarks.

6 Related Work

A considerable amount of work has been done on the problem of pointer analysis itself, and a detailed discussion can be found in [5]. More directly related to this paper are methods that use the results of heap pointer analysis in the context of dependence analysis and parallelization. The approaches include: techniques using path expressions to name locations [13], using syntax trees to name locations

[6], extending k-limited graphs with location names[9]; and dependence analysis based on shape information and path expressions [8]. The focus of these techniques is on identifying function-call parallelism for recursive data structures, and the heap analyses used are substantially more complex than our connection and shape analyses. Further, they do not consider the detection of loop parallelism, and also do not consider the presence of stack-directed pointers.

In contrast to the above techniques which are based on automatic heap analysis, Hummel et al. [7] use a language-based approach. They rely on the programmer to provide the information about the shape of the data structure via aliasing axioms. To compute dependence between two statements, they collect access paths with respect to an anchor. A theorem prover is used to identify if, given the aliasing axioms, the access paths can lead to the same node. This approach is quite powerful, as the aliasing axioms can accurately express the shape of even complex cyclic data structures, and the theorem prover can compare complex access paths. Our (shape) dependence test also uses the concept of collecting access paths with respect to a common anchor. However, it relies on connection and shape information that is automatically computed, is focused on identifying specific parallelism patterns, and thus uses only two types of access path comparisons (equivPaths and subPath[2]) which can be done efficiently. Further, it also considers identification of loop-carried dependencies, which is crucial for detecting loop parallelism.

7 Conclusions and Future Work

This paper has focused on how to use pointer analysis for detecting parallelism. We used a family of pointer analyses that run in a hierarchical fashion, for dependence testing. We particularly focused on using shape information, and presented special dependence tests that build access paths with respect to a common anchor, and then compare the two paths in view of the shape information available. Further, we introduced a separate test for identifying loop-carried dependences, which is crucial for detecting loop-parallelism. We also proposed the *foreach* loop construct for pointer-chasing loops, along with techniques to extract the parallelism available. Finally, we presented preliminary experimental results for a set of C benchmark programs.

Our future work will be in three major directions. Firstly, we plan to evaluate the effectiveness of our dependence tests on a larger set of benchmarks. Secondly, we plan to do a detailed study of the run time performance improvement achieved, due to the parallelism detected. Finally, we plan to continue to develop new analyses and transformations to detect parallelism in pointer-intensive programs, particularly those that use complex cyclic data structures.

[2] The disjointPath comparison is done via a combination of equivPaths and subPath comparisons.

References

1. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 1994.
2. Rakesh Ghiya and Laurie J. Hendren. Connection Analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6), pages 547–578, 1996.
3. 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 the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg, Florida, January 1996.
4. Rakesh Ghiya and Laurie J. Hendren. Putting Pointer Analysis to Work. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.
5. Rakesh Ghiya. Putting Pointer Analysis to Work. PhD Thesis, School of Computer Science, McGill University, Montreal, Canada. February 1998. Expected.
6. Vincent A. Guarna, Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume II, pages 212–220, August 1988.
7. Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 218–229, Orlando, Florida, June 1994.
8. 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.
9. Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, Portland, Oregon, June 1989.
10. Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 12–23, Boston, Massachusetts, October 1996.
11. Justiani. An array dependence testing framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, December 1994.
12. Christopher Lapkowski. A practical symbolic array dependence analysis framework for C. Master's thesis, McGill University, Montréal, Québec, April 1997.
13. 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, Atlanta, Georgia, June 1988.
14. Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.
15. Xinan Tang, Rakesh Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques*, San Franc., Calif., Nov. 1997.

This article was processed using the LaTeX macro package with LLNCS style