Parallelizing Graph Construction Operations in Programs with Cyclic Graphs [*]

Yuan-Shin Hwang
Department of Computer Science
National Taiwan Ocean University
Keelung 20224
Taiwan
shin@cs.ntou.edu.tw

ABSTRACT

Combining pointer analysis and dependence analysis can identify parallelism in programs with pointers or graphs. The compiler analysis techniques that have been proposed so far can only recognize parallelism in the traversal references of programs with cyclic graphs by analyzing the traversal patterns on graphs. Consequently, only the graph traversal operations will be parallelized while the graph construction operations will be left to be executed in sequential. Although this approach can usually parallelize the most time-consuming part of these programs (the graph traversal operations account for over 90% of total execution times in most programs), the rest of programs that accounts for less than 10% will dominate the execution on multiprocessor systems according to Amdahl's law. This paper presents a technique that can identify parallelism in the construction operations of programs even with cyclic graphs. Experiment results show this approach can parallelize the construction phase of such programs and achieve good speedup on multiprocessor systems.

KEY WORDS

Parallelizing Compilers, Pointer Analysis, Dependence Analysis, Cycilc Graphs, Multiprocessors

1. Introduction

Dependence analysis with the help of alias analysis [2, 4, 6, 7, 16, 21, 23, 24, 25] or shape analysis [3, 8, 20, 22] is the key technique to identify parallelism in programs with pointers or graphs, which are in turn represented by dynamic pointer-linked data structures. There have been several dependence analysis techniques proposed to analyze the data dependence of programs with pointers and dynamic recursive data structures [9, 11, 12, 13, 17]. These techniques can identify parallelism in certain classes of programs, but can not recognize parallelism in programs with cyclic graphs even though the traversal references of these programs exhibit high degree of parallelism.

A recent algorithm has been proposed to identify parallelism in such programs. It first identifies the traversal patterns of programs with cyclic graphs, isolates the statements that traverse that traverse the graphs (which determine the traversal patterns) and the statements that construct the traversed part of the graph (which decide the shapes of the traversed subgraphs), and then analyzes the data dependence based on the traversal patterns and shape information of these statements [15]. Consequently, this approach can identify parallelism based on data dependence and shape analysis on programs even with cyclic graphs. This approach has been applied to parallelize the traversal references of some well-known application programs with cyclic graphs and obtain over 90% parallelism coverage, which is defined as overall percentage of time spent in parallelized regions [10]. However, this approach can only analyze dependence of the traversal references of programs with cyclic graphs, but not the graph construction operations, which accounts for less than 10% of execution times. Consequently, only the graph traversal operations will be parallelized while the graph construction operations will be left to be executed in sequential.

Although the part of programs that accounts for over 90% of execution times can now be parallelized, according to Amdahl's law [1] these programs will not be scalable well since the rest 10% of sequential execution will dominate the execution on multiprocessor systems. For example, with only 10% of execution being sequential, the maximum speedup is 10 irrespective of the number of processors. Similarly, if 20% of the computation is executed sequentially, the maximum speedup will then be limited to 5. In order to achieve good scalability, the graph construction operations must be parallelized as well. This paper presents a technique to identify parallelism in the graph construction operations of programs even with cyclic graphs.

2. Background

2.1 Parallelism in Traversal References on Cyclic Graphs

Many scientific computational programs create cyclic graphs to represent adjacent matrices, which in turn describe the interactions among simulated objects. For ex-

---

[*] Supported in part by NSC grant NSC89-2213-E-019-016-T

ample, a bipartite graph will be created by EM3D [18] to represent the interactions between three E nodes (electric field) and three H nodes (magnetic field). Although the bipartite graph is cyclic overall, EM3D follows acyclic structures to access all the E and H nodes and their neighboring nodes. Specifically, the first of the two most time-consuming loops of EM3D visits every E node by traveling along the path $E(.n)^\star$ and computes values of each E node by referencing the values of its neighboring nodes on the path $E(.n)^\star.h(.n)^\star.s$. Similarly, the other loop travels to every H node along the path $H(.n)^\star$ and then computes values of each H node by traversing its neighboring nodes by the path $H(.n)^\star.e(.n)^\star.s$. These paths are the traversal patterns of the two most time-consuming loops of EM3D, and furthermore they represent the (acyclic) graphs that are viewed by the two loops.

Identifying parallelism by performing dependence analysis directly on the traversal references of a bipartite graph is difficult, since the graph is cyclic and shape analysis can not provide much help. On the other hand, if cyclic graphs can be reduced to acyclic graphs based on the information of traversal patterns, it would be much easier for compilers to identify parallelism in traversal references of programs with cyclic graphs. Furthermore, examining the traversal patterns of the two loops more closely will reveal that both loops in fact travel along paths on tree-like graphs to visit the E and H nodes and their neighbor lists, while they reference the values on the neighboring nodes through the rest edges of the acyclic traversal patterns. This observation will greatly simplify the dependence analysis even for programs with cyclic graphs. Since many scientific computational programs perform computations to simulate the interactions among sets of objects in similar fashions, compilers can use the knowledge of traversal patterns to parallelize the traversal references of such programs [15].

2.2 Parallelism in Construction Operations of Cyclic Graphs

The parallelism in the traversal references of EM3D shown above makes it possible for a compiler to parallelize the two most time-consuming loops. Every iteration of the first loop will visit an E node and then reference the neighboring H nodes of the E node by traversing its neighbor list. Furthermore, the iterations of the loop can be executed in any order since the iterations are independent. Once all the iterations of the first loop are executed, the iterations of the second loop will be executed in the same manner to visit H nodes.

Each loop in fact traverses a list of lists, which is comprised of two components: a list and the lists connected to the list. For the rest of the paper, the list will be called as the *main list*, while the lists will be called as the *tail lists*. Based on these terms, the tail lists of EM3D can be referenced in any order as the iterations of each loop are independent. Consequently, the tail lists can be built on the main list in any order and hence the construction operations of these tail lists can be parallelized. Specifically, after the lists of E nodes and H nodes are created, the neighbor lists (i.e. tail lists) can be constructed on the E node list and H node list in any order to construct the graphs.

```
p = E
do while (p)
   q = H
   do while (q)
      if (· · ·)
         x = new()
         x.s = q
         f = p.h
         x.n = f
         p.h = x
      end if
      q = q.n
   end do
   p = p.n
end do
```

(a) Original Loop

```
if (myproc().eq.0) p = E
call gsync()
call lock()
plocal = p
if (p) p = p.n
call unlock()
do while (plocal)
   q = H
   do while (q)
      if (· · ·)
         x = new()
         x.s = q
         f = p.h
         x.n = f
         p.h = x
      end if
      q = q.n
   end do
   call lock()
   plocal = p
   if (p) p = p.n
   call unlock()
end do
```

(b) Transformed Loop

Figure 1. Loop Transformation for Parallelization

Such parallelism exists in many scientific computational programs. Therefore, a compiler can exploit this property to parallelize the construction operations of cyclic graphs. For example, the program fragment shown in Figure 1(a) constructs the tail lists (i.e. neighbor lists) of the main list of E nodes, and the compiler can identify the parallelism and parallelize the construction operations by transforming the program fragment into parallel code, as shown in Figure 1(b).

3. Algorithm

The requirement that iterations of a loop which constructs the neighbor lists to build a list of lists described above can be parallelized is that all traversal references on the neighbor lists are independent as well. Therefore, the parallelism the paper intends to identify is the parallel construction operations of the tail lists. The approach can be broken into two steps.

- Identify parallelism in traversal references on cyclic graphs
- Identify parallelism in graph construction operations

3.1 Identifying Parallelism in Traversal References

This section recapitulates the technique to identify parallelism in traversal references by programs with cyclic graphs [15]. The algorithm can be broken into the following four steps.

3.1.1 Gathering Traversal Patterns

The traversal patterns of a program are determined by edge traversal operations on graphs by the statements with the form $p = q.n$, where p and q are pointer variables. Hence the first step of this technique is to gather the information of edges that are traversed by the statements with the form $p = q.n$. This step categorizes the traversed edges into two classes. The first class is called *primary traversal edges*, which are defined as follows, while the rest traversed edges are called *secondary traversal edges* or *reference edges*.

Definition 1 PRIMARY TRAVERSAL EDGES
A edge of a graph is a primary traversal edge of the graph if the edge is traversed, and if the destination node of the edge traversal operation is the source node of another edge traversal operation or the same edge traversal operation at next iteration. □

3.1.2 Computing Definition-Use Chains

Traversal patterns are determined by edge traversal operations on graphs, and hence they are *uses* of the referenced graphs. In order to estimate the possible shapes that are connected by the edges referenced by the traversal patterns, the statements that construct the referenced edges must be identified, i.e. the statements with the form $p.n = q$. Since these statements defines these accessed edges, they can be called as the *definitions* of these uses. Therefore, their relationships can be represented by definition-use chains of graphs, and consequently the second step of this technique is to compute the definition-use chains of graphs in the program.

3.1.3 Traversal-Pattern-Sensitive Shape Analysis

Instead of estimating the possible shapes the overall graphs, this technique identifies the possible shapes of the graphs that are connected by the primary traversal edges, which in turn are determined by the traversal patterns. Therefore, the third step of this technique can be called *traversal-pattern-sensitive shape analysis*. Existing shape analysis can be easily modified to traversal-pattern-sensitive algorithms, and the shape analysis algorithm proposed by Ghiya and Hendren [8] is adapted in this paper.

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 graphs 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 graphs, 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 data flow equations

$$\langle D, I, S\rangle[N]_{in} = \bigsqcup_{d\in pred(N)} \langle D, I, S\rangle[d]_{out} \quad (1)$$

$$\langle D_p, I_p, S_p\rangle[N]_{in} = \bigsqcup_{d\in pred(N)} \langle D_p, I_p, S_p\rangle[d]_{out} \quad (2)$$

$$\langle D, I, S\rangle[N]_{out} = F_N(\langle D, I, S\rangle[N]_{in}) \quad (3)$$

$$\langle D_p, I_p, S_p\rangle[N]_{out} = F_N^p(\langle D_p, I_p, S_p\rangle[N]_{in}) \quad (4)$$

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 [8, 14].

3.1.4 Dependence Analysis

The last step of identifying parallelism in traversal references is to perform dependence analysis to determine 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

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

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 the above equation.

Program	Exact Shapes		Original Algorithm		Modified Algorithm	
	Overall Graphs	Traversal Patterns	Overall Graphs	Traversal Patterns	Overall Graphs	Traversal Patterns
EM3D	*Cycle*	*Tree*	*Cycle*	*DAG*	*Cycle*	*Tree*
Moldyn	*Cycle*	*Tree*	*Cycle*	*DAG*	*Cycle*	*Tree*

Table 1. Results of Shape Analysis

3.2 Identify Parallelism in Construction Operations

As presented in Section 2.2, the parallelism that this approach intends to identify is parallel constructions of the tail lists on a list of lists. Therefore, one necessary condition that such construction operations can be parallelized is that all traversal references on the tail lists are independent. Once the dependence analysis on traversal references is performed, the following conditions must be examined to determine if construction operations are independent as well:

- Every graph that is connected by the primary traversal edges is a list of lists,
- The iterations of all loops that traverse the tail lists are independent once the graphs are constructed,
- The tail lists accessed by the traversal references are created by the construction operations, and
- The main lists are not modified between the traversal references and the construction operations.

The first two conditions can be examined during the dependence analysis on the traversal references as outlined in Section 3.1, while the rest two conditions can be analyzed by using the definition-use chains of graphs. If the conditions are met, the construction operations of graphs can be parallelized to enhance the scalability of programs.

3.3 A Modified Algorithm for Traversal-Pattern-Sensitive Shape Analysis

In order to parallelize the construction operations of tail lists, the graphs connected by the primary traversal edges must be lists of lists. Therefore, Section 3.1.3 outlines an algorithm to perform traversal-pattern-sensitive shape analysis. Unfortunately, the adapted algorithm is not precise enough due to the reason that the original algorithm can not make correct estimations if graphs are built by inserting new nodes between existing nodes [8]. For example, the second section of Table 1 lists the results of performing traversal-pattern-sensitive shape analysis adapted directly from Ghiya's algorithm on EM3D and Moldyn [19]. Compared with the exact shapes in the first section of the table, the algorithm makes over-conservative estimations.

This paper provides a remedy by modifying the representation of the direction matrix D. Instead of using $D[p,\, q] = 1$ to represent there is path from pointer p to