Parallel Pointer Analysis with CFL-Reachability

Yu Su Ding Ye Jingling Xue
School of Computer Science and Engineering
UNSW Australia
{ysu, dye, jingling}@cse.unsw.edu.au

***Abstract*—This paper presents the first parallel implementation of pointer analysis with Context-Free Language (CFL) reachability, an important foundation for supporting demand queries in compiler optimisation and software engineering. Formulated as a graph traversal problem (often with context- and field-sensitivity for desired precision) and driven by queries (issued often in batch mode), this analysis is non-trivial to parallelise.**

We introduce a parallel solution to the CFL-reachability-based pointer analysis, with context- and field-sensitivity. We exploit its inherent parallelism by avoiding redundant graph traversals with two novel techniques, *data sharing* and *query scheduling*. With data sharing, paths discovered in answering a query are recorded as shortcuts so that subsequent queries will take the shortcuts instead of re-traversing its associated paths. With query scheduling, queries are prioritised according to their statically estimated dependences so that more redundant traversals can be further avoided. Evaluated using a set of 20 Java programs, our parallel implementation of CFL-reachability-based pointer analysis achieves an average speedup of 16.2X over a state-of-the-art sequential implementation on 16 CPU cores.

I. Introduction

Pointer (or alias) analysis, which is an important component of an optimising or parallelising compiler, determines statically what a pointer may point to at runtime. It also plays an important role in debugging, verification and security analysis. Much progress has been made to improve the precision and efficiency of pointer analysis, across several dimensions, including field-sensitivity (to match field accesses), context-sensitivity (to distinguish calling contexts) and flow-sensitivity (to consider control-flow). In the case of object-oriented programs, e.g., Java programs, both context- and field-sensitivity are considered to be essential in achieving a good balance between analysis precision and efficiency [5].

Traditionally, Andersen's algorithm [2] has been frequently used to discover the points-to information in a program (e.g., in C and Java). Andersen's analysis is a whole-program analysis that computes the points-to information for all variables in the program. This algorithm has two steps: finding the constraints that represent the pointer-manipulating statements and propagating these constraints until a fixed point is reached.

For many clients, such as debugging [17], [18], [19] and alias disambiguation [21], however, the points-to information is needed *on-demand* only for some but not all variables in the program. As a result, there has been a recent focus on demand-driven pointer analysis, founded on Context-Free Language (CFL) reachability [15], which only performs the necessary work to compute the points-to information requested in a graph representation of the program [6], [16], [17], [18], [19], [26], [27]. By computing the points-to information of some (instead of all variables as in whole-program Andersen's analysis), demand-driven analysis can be performed both context- and field-sensitively to achieve better precision more scalably than Andersen's analysis, especially for Java programs.

However, precise CFL-reachability-based analysis can still be costly when applied to large programs. In the sequential setting, there are efforts for improving its efficiency for Java programs by resorting to refinement [18], [19], summarisation [17], [26], incrementalisation [6], [16], pre-analysis [25] and ad-hoc caching [18], [25]. However, multicore platforms, which are ubiquitous nowadays, have never been exploited for accelerating the CFL-reachability-based analysis.

There have been a few parallel implementations of Andersen's pointer analysis on multi-core CPUs [3], [8], [9], [14], GPUs [7], [10], and heterogeneous CPU-GPU systems [20], as compared in detail in Table II. As Andersen's analysis is a whole-program pointer analysis, these earlier solutions cannot be applied to parallelise demand-driven pointer analysis.

Despite the presence of redundant graph traversals both within and across the queries, parallelising CFL-reachability-based pointer analysis poses two challenges. First, parallelism-inhibiting dependences are introduced into its various analysis stages since processing a query involves matching calling contexts and handling heap accesses via aliased base variables (e.g., p and q in $x = p.f$ and $q.f = y$). Second, the new points-to/alias information discovered in answering some queries during graph traversals is not directly available to other queries on the (read-only) graph representation of the program.

In this paper, we describe the first parallel implementation of this important analysis for Java programs (although the proposed techniques apply equally well to C programs [27]). We expose its inherent parallelism by reducing redundant graph traversals using two novel techniques, *data sharing* and *query scheduling*. With data sharing, we recast the original graph traversal problem into a graph rewriting problem. By adding new edges to the graph representation of the program to shortcut the paths traversed in a query, we can avoid re-traversing (redundantly) the same paths by taking their shortcuts instead when handling subsequent queries. With query scheduling, we prioritise the queries to be issued (in batch mode) according to their statically estimated dependences so that more redundant graph traversals can be further reduced.

While this paper focuses on CFL-reachability-based pointer analysis, the proposed techniques for data sharing and scheduling are expected to be useful for parallelising other graph-reachability-based analyses (e.g., bug detection [22], [23]).

In summary, this paper makes the following contributions:

- The first parallel implementation of context- and field-sensitive pointer analysis with CFL-reachability;
- A data sharing scheme to reduce redundant graph traversals (in all query-processing threads) by graph rewriting;
- A query scheduling scheme to eliminate unnecessary traversals further, by prioritising queries according to their statically estimated dependences; and
- An evaluation with a set of 20 Java benchmarks, showing that our parallel solution achieves an average speedup of 16.2X over a state-of-the-art sequential implementation (with 16 threads) on 16 CPU cores.

The rest of this paper is organised as follows. Section II introduces the intermediate representation used for analysing Java programs and reviews CFL-reachability-based pointer analysis. Section III describes our parallel solution in terms of our data sharing and query scheduling schemes. Section IV evaluates and analyses our solution. Section V discusses the related work. Section VI concludes the paper.

II. Background

Section II-A describes the intermediate representation used for analysing Java programs. Section II-B reviews the standard formulation of pointer analysis in terms of CFL-reachability.

A. Program Representation

We focus on Java although this work applies equally well to C [27]. A Java program is represented as a *Pointer Assignment Graph (PAG)*, as defined in Fig. 1.

A node n represents a variable v or an object o in the program, where v can be local (l) or global (g). An edge e represents a statement in the program oriented in the direction of its value flow. An edge connects only to local variables (l_1 and/or l_2) unless it represents an assignment involving at least one global variable (assign^g). Let us look at the seven types of edges in detail. $l_1 \xleftarrow{\mathsf{new}} o$ captures the flow of object o to variable l_1, indicating that l_1 points directly to o. $l_1 \xleftarrow{\mathsf{assign}^l} l_2$ represents a local assignment ($l_1 = l_2$). A global assignment is similar except that one or both variables at its two sides are static variables in a class (i.e. g). $l_1 \xleftarrow{\mathsf{ld(f)}} l_2$ represents a load of the form $l_1 = l_2.f$ and $l_1 \xleftarrow{\mathsf{st(f)}} l_2$ represents a store of the form $l_1.f = l_2$. $l_1 \xleftarrow{\mathsf{param}_i} l_2$ models parameter passing, where l_2 is an actual parameter and l_1 is its corresponding formal parameter, at call site i. Similarly, $l_1 \xleftarrow{\mathsf{ret}_i} l_2$ indicates an assignment of the return value in l_2 to l_1 at call site i.

Fig. 2 gives an illustrating example and its PAG representation. Note that o_i denotes the object created at the allocation site in line i and v_m represents variable v declared in method m. Loads and stores to array elements are modeled by collapsing all elements into a special field, denoted `arr`.

B. CFL-Reachability-based Pointer Analysis

CFL-reachability [15] is an extension of traditional graph reachability. Let G be a directed graph with edges labelled

$$\begin{array}{rcll}
n & := & v \mid o & \text{Node} \\
v & := & l \mid g & \text{Variable} \\
e & := & l_1 \xleftarrow{\mathsf{new}} o & \text{Allocation} \\
 & \mid & l_1 \xleftarrow{\mathsf{assign}^l} l_2 & \text{Local Assignment} \\
 & \mid & g \xleftarrow{\mathsf{assign}^g} v \mid v \xleftarrow{\mathsf{assign}^g} g & \text{Global Assignment} \\
 & \mid & l_1 \xleftarrow{\mathsf{ld(f)}} l_2 & \text{Load} \\
 & \mid & l_1 \xleftarrow{\mathsf{st(f)}} l_2 & \text{Store} \\
 & \mid & l_1 \xleftarrow{\mathsf{param}_i} l_2 & \text{Parameter} \\
 & \mid & l_1 \xleftarrow{\mathsf{ret}_i} l_2 & \text{Return}
\end{array}$$

$$\begin{array}{lll}
l \in \mathit{Local} & g \in \mathit{Global} & o \in \mathit{Object} \\
i \in \mathit{CallSite} & f \in \mathit{Field} &
\end{array}$$

Fig. 1: Syntax of PAG (pointer assignment graph).

by letters over an alphabet Σ and L be a CFL over Σ. Each path p in G is composed of a string $s(p)$ in Σ^*, formed by concatenating in order the labels of edges along p. A path p is an *L-path* iff $s(p) \in L$. Node x is *L-reachable* to y iff a path p from x to y exists, such that $s(p) \in L$. For a single-source reachability analysis, the worst-case time complexity is $O(\Gamma^3 N^3)$, where Γ is the size of a normalised grammar for L and N is the number of nodes in G.

1) Field-sensitivity: Let us start with a field-sensitive formulation without context-sensitivity. When calling contexts are ignored, there is no need to distinguish the four types of assignments, assign^l, assign^g, param_i and ret_i; they are all identified as assignment edges of type assign.

For a PAG G with only new and assign edges, the CFL L_{FT} (FT for flows to) is a regular language, meaning that a variable v is L_{FT}-reachable from an object o in G iff o can flow to v. The rule for L_{FT} is defined as:

$$\mathit{flowsTo} \quad \rightarrow \quad \mathsf{new}\,(\mathsf{assign})^* \tag{1}$$

with *flowsTo* as the start symbol. If o *flowsTo* v, then v is L_{FT}-reachable from o. For example, in Fig. 2(b), since $\texttt{o}_{15} \xrightarrow{\mathsf{new}} \texttt{v1}_{\texttt{main}} \xrightarrow{\mathsf{param}_{15}} \texttt{this}_{\texttt{Vector}}$, $\texttt{o}_{15}$ flows to $\texttt{this}_{\texttt{Vector}}$.

When field accesses are also handled, the CFL L_{FS} (FS for field-sensitivity) is defined as follows:

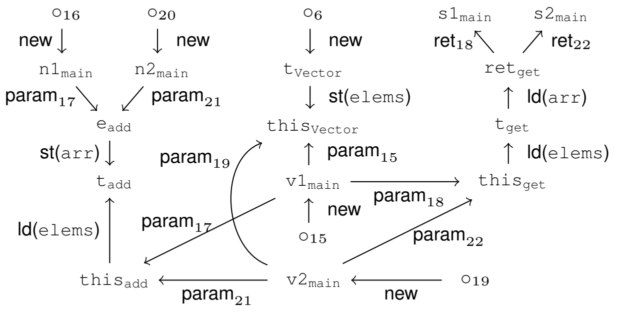
$$\begin{array}{rcl}
\mathit{flowsTo} & \rightarrow & \mathsf{new}\,(\,\mathsf{assign} \mid \mathsf{st(f)}\ \mathit{alias}\ \mathsf{ld(f)})^* \\
\mathit{alias} & \rightarrow & \overline{\mathit{flowsTo}}\ \mathit{flowsTo} \\
\overline{\mathit{flowsTo}} & \rightarrow & (\,\overline{\mathsf{assign}} \mid \overline{\mathsf{ld(f)}}\ \mathit{alias}\ \overline{\mathsf{st(f)}})^*\ \overline{\mathsf{new}}
\end{array} \tag{2}$$

The rule for *flowsTo* matches fields via $\mathsf{st(f)}$ *alias* $\mathsf{ld(f)}$, where $\mathsf{st(f)}$ and $\mathsf{ld(f)}$ are like a pair of parentheses [19]. For a pair of statements $q.f = y$ ($\mathsf{st}(f)$) and $x = p.f$ ($\mathsf{ld}(f)$), if p and q are aliases, then an object o that flows into y can flow first through this pair of parentheses (i.e. $\mathsf{st}(f)$ and $\mathsf{ld}(f)$) and then into x. For example, in Fig. 2(b), as $\texttt{o}_{15} \xrightarrow{\mathsf{new}} \texttt{v1}_{\texttt{main}} \xrightarrow{\mathsf{param}_{15}} \texttt{this}_{\texttt{Vector}}$ and $\texttt{o}_{15} \xrightarrow{\mathsf{new}} \texttt{v1}_{\texttt{main}} \xrightarrow{\mathsf{param}_{18}} \texttt{this}_{\texttt{get}}$, we have $\texttt{this}_{\texttt{Vector}}$ *alias* $\texttt{this}_{\texttt{get}}$. Thus $\texttt{o}_{6} \xrightarrow{\mathsf{new}} \texttt{t}_{\texttt{Vector}} \xrightarrow{\mathsf{st(elems)}} \texttt{this}_{\texttt{Vector}}$ *alias* $\texttt{this}_{\texttt{get}} \xrightarrow{\mathsf{ld(elems)}} \texttt{t}_{\texttt{get}}$, indicating that $\texttt{o}_6$ flows to $\texttt{t}_{\texttt{get}}$.

To allow the *alias* relation in the language, $\overline{\mathit{flowsTo}}$ is introduced as the inverse of the *flowsTo* relation. For a *flowsTo*-path p, its corresponding $\overline{\mathit{flowsTo}}$-path $\overline{p}$ can be obtained using

```
1  class Vector {
2    Object[] elems;
3    int count;
4    Vector(){
5      count = 0;
6      t = new Object[MAXSIZE];
7      this.elems = t;}
8    void add(Object e){
9      t = this.elems;
10     t[count++] = e;} // W t.arr
11   Object get(int i){
12     t = this.elems;
13     return t[i];} // R t.arr
14 static void main(String[] args){
15   Vector v1 = new Vector();
16   String n1 = new String("N1");
17   v1.add(n1);
18   Object s1 = v1.get(0);
19   Vector v2 = new Vector();
20   Integer n2 = new Integer(1);
21   v2.add(n2);
22   Object s2 = v2.get(0);}}
```

(a) Java code



(b) PAG

Fig. 2: A Java example and its PAG.

inverse edges, and vice versa. For each edge $x \overset{e}{\leftarrow} y$ in p, its inverse edge is $y \overset{\overline{e}}{\leftarrow} x$ in $\overline{p}$. Therefore, o *flowsTo* x iff x $\overline{\textit{flowsTo}}$ o, indicating that $\overline{\textit{flowsTo}}$ actually represents the points-to relation. To find the points-to set of a variable, we use the CFL given in (2) with $\overline{\textit{flowsTo}}$ as the start symbol.

2) Context-sensitivity: When context-sensitivity is considered, param_i and ret_i are treated as assign as before in L_{FS}. However, assign^l and assign^g are now distinguished.

A context-sensitive CFL requires param_i and ret_i to be matched, also in terms of balanced parentheses, by ruling out the unrealisable paths in a program [18]. The CFL R_{CS} (CS for context-sensitivity) for capturing realisable paths is:

$$\begin{aligned} c &\rightarrow \textit{entry}_i\ c\ \textit{exit}_i \mid c\ c \mid \epsilon \\ \textit{entry}_i &\rightarrow \mathsf{param}_i \mid \overline{\mathsf{ret}_i} \\ \textit{exit}_i &\rightarrow \mathsf{ret}_i \mid \overline{\mathsf{param}_i} \end{aligned} \tag{3}$$

When traversing along a *flowsTo* path, after entering a method via param_i from call site i, a context-sensitive analysis requires exiting from that method back to call site i, via either ret_i to continue its traversal along the same *flowsTo* path or $\overline{\mathsf{param}_i}$ to start a new search for a $\overline{\textit{flowsTo}}$ path. Traversing along a $\overline{\textit{flowsTo}}$ path is similar except that the direction of traversal is reversed. Consider Fig. 2(b). $\texttt{s1}_{\texttt{main}}$ is found to point to o_{16} as o_{16} reaches $\texttt{s1}_{\texttt{main}}$ along a realisable path by first matching param_{17} and $\overline{\mathsf{param}_{17}}$ and then param_{18} and ret_{18}. However, $\texttt{s1}_{\texttt{main}}$ does not point to o_{20} since o_{20} cannot reach $\texttt{s1}_{\texttt{main}}$ along a realisable path.

Let L_{PT} (PT for points-to) be the CFL for computing the points-to information of a variable field- and context-sensitively. Then L_{PT} is defined in terms of (2) and (3): $L_{\text{PT}} = L_{\text{FS}} \cap R_{\text{CS}}$, where $\overline{\textit{flowsTo}}$ is the start symbol.

3) Algorithm: With CFL-reachability, a query that requests for a variable's points-to information can be answered on-demand, according to Algorithm 1. This algorithm makes use of three variables: (1) E represents the edge set of the PAG of the program, (2) B is the (global) budget defined as the maximum number of steps that can be traversed by any query, with each node traversal being counted as one step [19], and (3) *steps* is query-local, representing the number of steps that has been traversed so far by a particular query.

Given a query (l, c), where l is a local variable and c is

Algorithm 1 CFL-reachability-based pointer analysis, where POINTSTO computes $\overline{\textit{flowsTo}}$ and FLOWSTO is analogous to its inverse POINTSTO and thus omitted.

```
Global $E$; Const $B$; QueryLocal $steps$; // initially 0
Procedure POINTSTO($l, c$)
begin
1    $pts \leftarrow \emptyset$;
2    $W \leftarrow \{<l, c>\}$;
3    while $W \neq \emptyset$ do
4        $<x, c> \leftarrow W.pop()$;
5        $steps \leftarrow steps + 1$;
6        if $steps > B$ then OUTOFBUDGET (0);
7        foreach $x \xleftarrow{\mathsf{new}} o \in E$ do $pts \leftarrow pts \cup \{<o, c>\}$;
8        foreach $x \xleftarrow{\mathsf{assign}^l} y \in E$ do $W.push(<y, c>)$;
9        foreach $x \xleftarrow{\mathsf{assign}^g} y \in E$ do $W.push(<o, \epsilon>)$ ;
10       foreach $<y, c'> \in$ REACHABLENODES($x, c$) do
11           $W.push(<y, c'>)$;
12       foreach $x \xleftarrow{\mathsf{param}_i} y \in E$ do
13           if $c = \emptyset$ or $c.top() = i$ then
14               $W.push(<y, c.pop()>)$; // $\epsilon.pop() \equiv \epsilon$
15       foreach $x \xleftarrow{\mathsf{ret}_i} y \in E$ do $W.push(<y, c.push(i)>)$;
16   return $pts$;
Procedure REACHABLENODES($x, c$)
begin
17   $rch \leftarrow \emptyset$;
18   foreach $x \xleftarrow{\mathsf{ld(f)}} p \in E$ do
19       foreach $q \xleftarrow{\mathsf{st(f)}} y \in E$ do
20           $alias \leftarrow \emptyset$;
21           foreach $<o, c'> \in$ POINTSTO($p, c$) do
22               $alias \leftarrow alias \cup$ FLOWSTO($o, c'$);
23           foreach $<q', c''> \in alias$ do
24               if $q' = q$ then $rch \leftarrow rch \cup \{<y, c''>\}$;
25   return $rch$;
Procedure OUTOFBUDGET($BDG$)
begin
26   exit();
```

a context, POINTSTO computes the points-to set of l under c. It traverses the PAG with a work list W maintained for variables to be explored. pts is initialised with an empty set and W with $<l,c>$ (lines 1 – 2). By default, $steps$ for this query is initialised as 0. Each variable x with its context c, i.e., $<x,c>$ obtained from W is processed as follows: $steps$ is updated, triggering a call to OUTOFBUDGET if the remaining budget is 0 (lines 5 – 6), and the incoming edges of x are traversed according to (2) and (3) (lines 7 – 15).

Field-sensitivity is handled by REACHABLENODES(x,c), which searches for the reachable variables y to x in context c, due to heap accesses by matching the load ($x = p.f$) with every store ($q.f = y$), where p and q are aliases (lines 17 – 25). Both POINTSTO and FLOWSTO are called (recursively) to ensure that p and q are aliased base variables.

To handle context-sensitivity, the analysis stays in the same context c for assign^l (line 8), clears c for assign^g as global variables are treated context-insensitively (line 9), matches the context ($c.top() = i$) for param_i but allows for partially balanced parentheses when $c = \emptyset$ since a realizable path may not start and end in the same method (lines 12 – 14), and pushes call site i into context c for ret_i (line 15).

III. METHODOLOGY

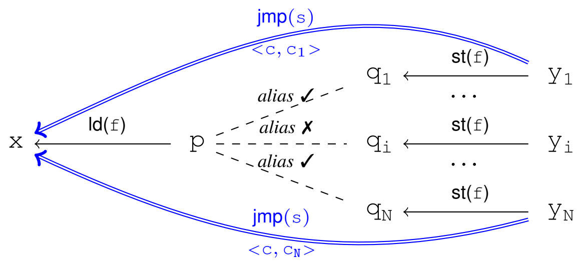
CFL-reachability-based pointer analysis is driven by queries issued by application clients. There are two main approaches to dividing work among threads, based on different levels of parallelism available: *intra-query* and *inter-query*.

To exploit intra-query parallelism, we need to partition and distribute the work performed in computing the points-to set of a single query among different threads. Such parallelism is irregular and hard to achieve with the right granularity. In addition, considerable synchronisation overhead that may be incurred would likely offset the performance benefit achieved.

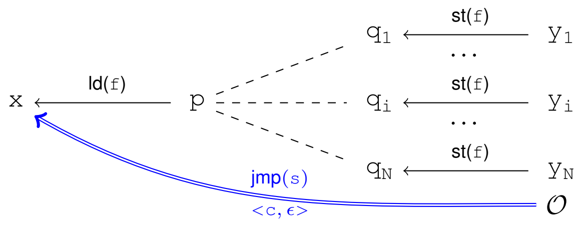
To exploit inter-query parallelism, we assign different queries to different threads, harnessing modern multicore processors. This makes it possible to obtain parallelism without incurring synchronisation overhead unduly. In addition, some clients may issue queries in batch mode for a program. For example, the points-to information may be requested for all variables in a method, a class, a package or even the entire program. This provides a further optimisation opportunity. The focus of this work is on exploiting inter-query parallelism.

A. A Naive Parallelisation Strategy

A naive approach to exploiting inter-query parallelism is to maintain a lock-protected shared work list for queries and let each thread fetch queries (to process) from the work list until the work list is empty. While achieving some good speedups (over the sequential setting), this naive strategy is inefficient due to a large number of redundant graph traversals made. We propose two schemes to reduce such redundancies. Section III-B describes our data sharing scheme, while Section III-C explains our query scheduling scheme.



(a) Within budget: all N stores analysed completely in s steps from (x,c)



(b) Out of budget: fewer than N stores analysed in s steps from (x,c)

Fig. 3: Adding jmp edges by graph rewriting, for a single iteration of the loop in line 18 of REACHABLENODES(x,c). In (a), $x \xLeftarrow[<c,\,c_k>]{\mathsf{jmp}(s)} y_k$ is introduced for each (y_k, c_k) added to rch in line 24 of REACHABLENODES(x,c) when p and q_k are aliases. In (b), a special $x \xLeftarrow[<c,\,\epsilon>]{\mathsf{jmp}(s)} \mathcal{O}$ edge is introduced.

B. Data Sharing

Given a program, we are motivated to add edges to its PAG to serve as shortcuts for some paths traversed in a query so that subsequent queries may take the shortcuts instead of re-traversing their associated paths (redundantly). The challenge here is to perform data sharing context- and field-sensitively. Section III-B1 formulates data sharing in terms of graph rewriting. Section III-B2 gives an algorithm for realising data sharing in the CFL-reachability framework.

1) Data Sharing by Graph Rewriting: We choose to share paths involving heap accesses, which tend to be long (time-consuming to traverse) and common (repeatedly traversed across the queries). As illustrated in Fig. 3, we do so by avoiding making redundant alias tests in REACHABLENODES(x,c). For its loop at line 18, each iteration starts with a load $x = p.f$ and then examines all the N matching stores $q_1.f = y_1, \ldots, q_N.f = y_N$ at line 19. For each $q_k.f = y_k$ accessed in context c_k such that q_k is an alias of p, (y_k, c_k) is inserted into rch, meaning that (x,c) is reachable from (y_k, c_k) (lines 20 - 24). Note that during this process, mutually recursive calls to POINTSTO(), FLOWSTO() and REACHABLENODES() for discovering other aliases are often made.

There are two cases due to the budget constraint. Fig. 3(a) illustrates the case when an iteration of line 18 is completely analysed in s steps starting from (x,c) within the pre-set budget. A jmp edge, $x \xLeftarrow[<c,\,c_k>]{\mathsf{jmp}(s)} y_k$, is added for each q_k that is an alias of p. Instead of rediscovering the path from

(x, c) to (y_k, c_k), a subsequent query will take this shortcut.

Fig. 3(b) explains the other case when an iteration of line 18 is only partially analysed since the analysis runs out of budget after s steps have elapsed from (x, c). A special jmp edge, $x \xleftarrow[<c,\,\epsilon>]{\text{jmp}(s)} \mathcal{O}$, is added to record this situation, where $\mathcal{O}$ is a special node added and ϵ is a "don't-care" context. A later query will benefit from this special shortcut by making an *early termination (ET)* if its remaining budget is smaller than s.

Therefore, we have formulated data sharing as a graph rewriting problem by adding jmp edges to the PAG of a program, in terms of the syntax given in Fig. 4.

$$\begin{array}{rcll} l & := & \ldots \mid \mathcal{O} & \text{Extended Local Variable} \\ e & := & \ldots & \\ & \mid & l_1 \xleftarrow[<c_1,\,c_2>]{\text{jmp}(s)} l_2 & \text{Jump (or Shortcut)} \\ & & \mathcal{O} \text{ is Unfinished} & c_1,\, c_2 \in \mathit{Context} \end{array}$$

Fig. 4: Syntax of extended PAG.

As described below, jmp edges are added on the fly during the analysis. Given a PAG extended with such jmp edges, the CFL given earlier in (2) is modified to:

$$\begin{array}{rcl} \mathit{flowsTo} & \rightarrow & \text{new} \,(\, \text{assign} \mid \text{jmp}(s) \mid \text{st(f)}\ \mathit{alias}\ \text{ld(f)})^* \\ \mathit{alias} & \rightarrow & \overline{\mathit{flowsTo}}\ \mathit{flowsTo} \\ \overline{\mathit{flowsTo}} & \rightarrow & (\,\overline{\text{assign}} \mid \overline{\text{jmp}(s)} \mid \overline{\text{ld(f)}}\ \mathit{alias}\ \overline{\text{st(f)}})^*\ \overline{\text{new}} \end{array} \quad (4)$$

By definition of jmp, this modified CFL generates the same language as the original CFL if all jmp edges of the type illustrated in Fig. 3(b) (for handling OUTOFBUDGET) in the PAG of a program are ignored, since the jmp edges of the other type illustrated in Fig. 3(a) serve as shortcuts only. Two types of jmp edges are exploited in our parallel implementation to accelerate its performance as described below.

2) Algorithm: With data sharing, REACHABLENODES(x, c) in Algorithm 1 is revised as shown in Algorithm 2. There are three cases, the original one plus the two shown in Fig. 3:

- In the **if** branch (line 2) for handling the scenario depicted in Fig. 3(b), the analysis makes an early termination by calling REACHABLENODES() if its remaining budget at (x, c), $B - \mathit{steps}$, is smaller than s. Otherwise, the analysis moves to execute the second **else**.
- In the first **else** branch for handling the scenario in Fig. 3(a), the analysis takes the shortcuts identified by the jmp(s) edges instead of re-traversing its associated paths. The same precision is maintained even if we do not check the remaining budget $B - \mathit{steps}$ against s, since the source node of a jmp edge is a variable (not an object). When this variable is explored later, the remaining budget will be checked in line 6 of Algorithm 1 or line 3 of Algorithm 2.
- In the second **else** branch, we proceed as in REACHABLENODES(x, c) given in Algorithm 1 except that we will need to add the jmp edge(s) as illustrated in either Fig. 3(a) (line 20) or Fig. 3(b) (line 24).

Algorithm 2 REACHABLENODES with data sharing.

```
Global $E$; Const $B$; QueryLocal $steps$, $S$;
Procedure REACHABLENODES$(x, c)$
begin
1       $rch \leftarrow \emptyset$;
2       if $\exists\, x \xleftarrow[<c,\,\epsilon>]{\text{jmp}(s)} \mathcal{O} \in E$ then
3           if $B - steps < s$ then OUTOFBUDGET$(s)$;
4       else if $\exists\, x \xleftarrow[<c,\,c'>]{\text{jmp}(s)} y \in E$ then
5           $steps \leftarrow steps + s$;
6           foreach $x \xleftarrow[<c,\,c'>]{\text{jmp}(s)} y \in E$ do
7               $rch \leftarrow rch \cup \{<y, c'>\}$;
8           return $rch$;
9       else
10          $s' = steps$;
11          $S \leftarrow S \cup \{<x, c, s'>\}$;
12          foreach $x \xleftarrow{\text{ld(f)}} p \in E$ do
13              foreach $q \xleftarrow{\text{st(f)}} y \in E$ do
14                  $alias \leftarrow \emptyset$;
15                  foreach $<o, c'> \in$ POINTSTO$(p, c)$ do
16                      $alias \leftarrow alias \cup$ FLOWSTO$(o, c')$;
17                  foreach $<q', c''> \in alias$ do
18                      if $q' = q$ then
19                          $rch \leftarrow rch \cup \{<y, c''>\}$;
20                          $E \leftarrow E \cup \{x \xleftarrow[<c,\,c''>]{\text{jmp}(steps - s')} y\}$;
21          $S \leftarrow S \setminus \{<x, c, s'>\}$;
22          return $rch$;

Procedure OUTOFBUDGET$(BDG)$
begin
23      foreach $<x, c, s> \in S$ do
24          $E \leftarrow E \cup \{x \xleftarrow[<c,\,\epsilon>]{\text{jmp}(\min(B, BDG + steps - s))} \mathcal{O}\}$;
25      exit();
```

OUTOFBUDGET(BDG) is called from line 6 (by passing 0) in Algorithm 1 or line 3 in Algorithm 2 (by passing s). In both cases, let n be the node visited before the call. With a remaining budget no larger than BDG on encountering n, the analysis will surely run out of budget eventually. For each $(x, c, s) \in S$, the analysis first reaches (x, c) and then n in $\mathit{steps} - s$ steps. Thus, $x \xleftarrow[<c,\,\epsilon>]{\text{jmp}(\min(B, BDG + steps - s))} \mathcal{O}$ is added.

C. Query Scheduling

The order in which queries are processed affects the number of early terminations made, due to $B - \mathit{steps} < s$ tested in line 3 of Algorithm 2, where s appears in a jmp(s) edge that was added in an earlier query and steps is the number of steps already consumed by the current query. In general, if we handle a variable y before those variables x such that x is reachable from y, then more early terminations may result.

To increase early terminations, we organise queries (available in batch mode) in groups and assign a group of queries rather than a single query to a thread at a time to reduce synchronisation overhead on the shared work list for queries. We discuss below how the queries in a group and the groups themselves are scheduled. Section III-C1 describes how to group queries. Section III-C2 discusses how to order queries. Section III-C3 gives an illustrating example.

1) Grouping Queries: A group contains all possible variables such that every member is connected with at least another member in the PAG of the program via the following relation:

$$\textit{direct} \quad \rightarrow \quad (\,\mathsf{assign}^{l} \mid \mathsf{assign}^{g} \mid \mathsf{param}_{i} \mid \mathsf{ret}_{i})^{*} \qquad (5)$$

Both $l_1 \xleftarrow{\mathsf{ld(f)}} l_2$ and $l_1 \xleftarrow{\mathsf{st(f)}} l_2$ edges are not included since there is no reachability between l_1 and l_2.

2) Ordering Queries: For the variables in the same group, we use their so-called *connection distances* (CDs) to determine their issuing order. The CD of a variable in a group is defined as the length of the longest path that contains the variable in the group (modulo recursion). For the variables in a group, the shorter their CDs are, the earlier they are processed.

For different groups, we use their so-called *dependence depths* (DDs) to determine their scheduling order. For example, computing $\textsc{PointsTo}(x, c)$ for x in Algorithm 1 depends on the points-to set of the base variable p in load $x = p.f$ (line 21). Preferably, p should be processed earlier than x.

To quantify the DD of a group, we estimate the dependences between variables based on their (static) types. We define the *level* of a type t (with respect to its containment hierarchy) as:

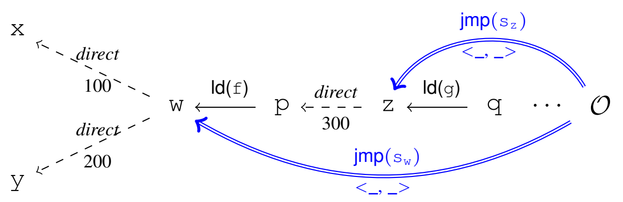
$$L(t) = \begin{cases} \max_{t_i \in F_T(t)} L(t_i) + 1 & \mathsf{isRef}(t) \\ 0 & \text{otherwise} \end{cases}$$

where $F_T(t)$ enumerates the types of all instance fields of t (modulo recursion) and $\mathsf{isRef}(t)$ is true if t is a reference type. The DD of a variable of type t is defined to be $1/L(t)$. Note that the DD of a static variable is also approximated heuristically this way. The DD of a group of variables is defined as the smallest of the DDs of all variables in the group.

During the analysis, groups are issued (sorted) in increasing values of their DDs. Let M be the average size of these groups. To ensure load balance, groups larger than M are split and groups smaller than M are merged with their adjacent groups, so that each resulting group has roughly M variables.

3) An Example: In Fig. 5, we focus on its three variables x, y, and z, which are assumed to all run out of budget B. According to (5), as shown in Fig. 5(a), x and y (together with w) are in one group and z (together with p) is in another group. The CDs of x, y and z are 100, 200 and 300 steps, respectively. As both x and y depend on z, the latter group will be scheduled before the former group. As a result, our query scheduling scheme will likely cause x, y and z to be processed sequentially according to O_3 (in some thread interleaving) among the three orders, O_1, O_2 and O_3, listed in Fig. 5(b).

For O_1, y is processed first, which takes B steps (i.e., the



(a) PAG with the *direct* relation

Order	Traversed #Steps			jmp(s)		$\#ETs$
	x	y	z	s_z	s_w	
$O_1 : y, x, z$	B	B	B	$B - 500$	$B - 200$	0
$O_2 : x, y, z$	B	200	B	$B - 400$	$B - 100$	1
$O_3 : z, x, y$	400	200	B	B	B	2

(b) Three scheduling orders

Fig. 5: An example of query scheduling, where x has a smaller CD than y and $\{x, y\}$ has a higher DD than $\{z\}$.

maximum budget allowed), with the two jmp edges added as shown, where $s_z = B - 500$ and $s_w = B - 200$. When x is processed next, neither shortcut will be taken, since x still has more budget remaining: $B - 400 > B - 500$ at z and $B - 100 > B - 200$ at w. Similarly, the two shortcuts do not benefit z either. Thus, no early termination occurs.

For O_2, x is issued first, resulting in also the same two jmp edges added, except that $s_z = B - 400$ and $s_w = B - 100$. So when y is handled next, an early termination is made at w, since its budget remaining at w is $B - 200$ ($< s_w = B - 100$).

According to O_3, the order that is mostly likely induced by our query scheduling scheme, z is processed first. Only the $\mathsf{jmp}(s_z)$ edge at z is added, where $s_z = B$. When x is analysed next, z is reached in 400 steps. Taking $\mathsf{jmp}(s_z)$ (since $B - 400 < s_z = B$), an early termination is made. Meanwhile, the $\mathsf{jmp}(s_w)$ edge at w is added, where $s_w = B$. Finally, y is issued, causing w to be visited in 200 steps. Taking $\mathsf{jmp}(s_w)$ (since $B - 200 < s_w = B$), another early termination is made. Of the three orders illustrated in Fig. 5(b), O_3 is likely to cause more early terminations, resulting in fewer traversal steps.

IV. Evaluation

We demonstrate that our parallel implementation of CFL-reachability-based pointer analysis achieves significant speedups than a state-of-the-art sequential implementation.

A. Implementations

The sequential one is coded in Java based on the publicly available source-code implementation of the CFL-reachability-based pointer analysis [18] in Soot 2.5.0 [24], with its non-refinement (general-purpose) configuration used. Note that the refinement-based configuration is not well-suited to certain clients such as null-pointer detection. Our parallel implementation given in Algorithms 1 and 2 are also coded in Java. In both cases, the per-query budget B is set as 75,000 steps, recursion cycles of the call graph are collapsed, and points-to cycles are eliminated as described as in [18].

Benchmark	#Classes	#Methods	#Nodes	#Edges	#Queries	T_{Seq} (secs)	#Jumps	$\#S$ $(\times 10^6)$	R_S	$\overline{S_g}$	$\#ETs$	R_{ET}
_200_check	5,758	54,514	225,797	429,551	1,101	2.88	428	4.14	25.76	16.7	0	1
_201_compress	5,761	54,549	225,765	429,808	1,328	3.72	1,210	4.21	8.42	4.6	5	1.00
_202_jess	5,901	55,200	232,242	440,890	7,573	121.11	4,755	193.77	42.68	16.1	617	1.15
_205_raytrace	5,774	54,681	227,514	432,110	3,240	9.39	2,325	62.02	92.84	7.2	8	0.88
_209_db	5,753	54,549	225,994	430,569	1,339	16.98	4,202	10.06	10.02	10.3	18	1.17
_213_javac	5,921	55,685	240,406	473,680	14,689	258.34	5,309	467.28	64.60	9.2	76	0.99
_222_mpegaudio	5,801	54,826	230,349	435,391	6,389	46.52	2,306	86.17	53.33	3.8	53	3.17
_227_mtrt	5,774	54,681	227,514	432,110	3,241	10.38	2,358	62.17	115.70	7.2	7	0.86
_228_jack	5,806	54,830	229,482	435,159	6,591	39.54	25,030	79.48	40.03	14.2	100	1.62
_999_checkit	5,757	54,548	226,292	431,435	1,473	12.61	2,180	10.14	7.94	16.9	23	0.78
avrora	3,521	29,542	108,210	189,081	24,455	51.16	32,046	47.46	6.18	9.4	24	2.83
batik	7,546	65,899	252,590	477,113	64,467	72.72	14,876	114.57	11.95	10.3	38	1.37
fop	8,965	79,776	266,514	636,776	71,542	118.22	25,418	169.92	19.03	18.6	76	1.20
h2	3,381	32,691	115,249	204,516	44,901	25.50	22,094	91.38	12.39	16.0	283	0.66
luindex	3,160	28,791	108,827	191,126	22,415	23.28	62,457	60.93	8.72	8.2	113	0.71
lusearch	3,120	28,223	109,439	193,012	17,520	57.78	77,153	66.26	7.90	9.3	75	1.52
pmd	3,786	33,432	110,388	195,834	56,833	61.05	77,313	69.10	7.93	9.2	84	1.06
sunflow	6,066	56,673	233,459	447,002	21,339	55.56	20,946	49.04	5.57	7.4	24	2.38
tomcat	8,458	83,092	265,015	574,236	185,810	202.89	24,601	243.90	23.14	13.1	574	1.33
xalan	3,716	33,248	109,317	192,441	56,229	54.11	33,459	60.35	7.90	9.4	82	1.43
Average	5,486	50,972	198,518	383,592	30,624	62.19	22,023	97.62	28.6	10.9	114.0	1.35

TABLE I: Benchmark information and statistics.

In our parallel implementation, we use a `ConcurrentHashMap` to manage jmp edges efficiently. We apply a simple optimisation to further reduce synchronisation incurred and thus achieve better speedups.

If we create jmp edges exhaustively for all the paths discovered in Algorithm 2, the overhead incurred by such operations as search, insertion and synchronisation on the map may outweigh the performance benefit obtained. As an optimisation, we will introduce the $\mathsf{jmp}(s)$ edges in Fig. 3(a) only when $s \geq \tau_F$ and the special $\mathsf{jmp}(s)$ edge in Fig. 3(b) only when $s \geq \tau_U$, where τ_F and τ_U are tunable parameters. In our experiments, we set $\tau_F = 100$ and $\tau_U = 10000$. Their performance impacts are evaluated in Section IV-D2.

For the case in Fig. 3(a), the set of all jmp edges is associated with the key (x, c) when inserted into the map. So no two threads reaching (x, c) simultaneously will insert this set of jmp edges twice into the map. For the case in Fig. 3(b), if one thread tries to insert $< (x, c), x \xLeftarrow[<c,\,\epsilon>]{\mathsf{jmp}(s_1)} \mathcal{O} >$ and another tries to insert $< (x, c), x \xLeftarrow[<c,\,\epsilon>]{\mathsf{jmp}(s_2)} \mathcal{O} >$ into the map, only one of the two will succeed. An attempt that selects the one with the large s value (to increase early terminations) can be cost-ineffective due to the extra work performed.

B. Experimental Setting

The multi-core system used in our experiments is equipped with two Intel Xeon E5-2650 CPUs with 62GB of RAM. Each CPU has 8 cores, which share a unified 20MB L3 cache. Each CPU core has a frequency of 2.00MHz, with its own L1 cache of 64KB and L2 cache of 256KB. The Java Virtual Machine used is the Java HotSpot 64-Bit Server VM (version 1.7.0_40), running on a 64-bit Ubuntu 12.04 operating system.

C. Methodology

We evaluate the performance advantages of our parallel implementation over the sequential one by comparing the query-processing times taken in both cases. SEQCFL denotes the sequential implementation. In order to assess the effectiveness of our parallel implementation, we consider a number of variations. PARCFL_m^t represents a particular parallel implementation, where t stands for the number of threads used. Here, m indicates one of the three parallelisation strategies used: (1) the naive solution described in Section III-A when $m = naive$, (2) our parallel solution with the data sharing scheme described in Section III-B enabled when $m = D$, and (3) the parallel solution (2) with the query scheduling scheme described in Section III-C also enabled when $m = DQ$.

Table I lists a set of 20 Java benchmarks used, consisting of all the 10 SPEC JVM98 benchmarks and 10 additional benchmarks from the DaCapo 2009 benchmark suite. For each benchmark, the analysed code includes both the application code and the related library code, with their class and method counts given in Columns 2 and 3, respectively. The node and edge counts in the original PAG of a benchmark are given in Columns 4 and 5, respectively. For each benchmark, the queries that request points-to information are issued for all the local variables in its application code, collected from Soot 2.5.0 as in prior work [17], [25]. Note that more queries are generated in some DaCapo benchmarks than some JVM98 benchmarks even though the DaCapo benchmarks have smaller PAGs. This is because the JVM98 benchmarks involve more library code. The remaining columns are explained below.

D. Performance Results

We examine the performance benefits of our parallel pointer analysis and the causes for the speedups obtained.

1) Speedups: Fig. 6 shows the speedups of our parallel implementation over SEQCFL (as the baseline), where the analysis times of SEQCFL for all the benchmarks are given in Column 7 of Table I. Note that SEQCFL is 49% faster than the open-source sequential implementation of [18] in Soot 2.5.0, since we have simplified some of its heuristics and em-

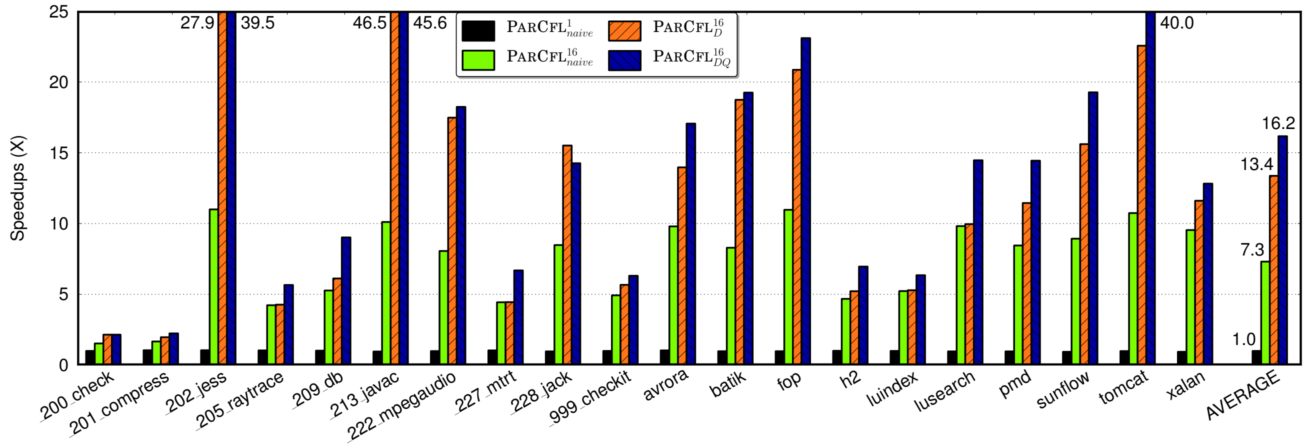


Fig. 6: Speedups of our parallel implementation (in various configurations) normalised with respect to SeqCfl.

ployed different data structures. When the naive parallelisation strategy is used, ParCfl$^1_{naive}$ (with one single thread) is as efficient as SeqCfl, since the locking overhead incurred for retrieving the queries from the shared work list is negligible. With 16 threads, ParCfl$^{16}_{naive}$ attains an average speedup of 7.3X. When our data sharing scheme is used, ParCfl$^{16}_{D}$ runs a lot faster, with the average speedup being pushed up further to 13.4X. When our query scheduling scheme is also enabled, ParCfl$^{16}_{DQ}$, which traverses significantly fewer steps than SeqCfl, has finally reached an average speedup of 16.2X. The superlinear speedups are achieved in some benchmarks due to the avoidance of redundant traversals (a form of caching) in all concurrent query-processing threads as analysed below.

2) *Effectiveness of Data Sharing:* Our data sharing scheme, which enables the traversal information obtained in a query to be shared by subsequent queries via graph rewriting, has succeeded in accelerating the analysis on top of the naive parallelisation strategy (ParCfl$^t_{naive}$) for all benchmarks.

To understand its effectiveness, some statistics are given in Columns 8 – 10 in Table I. For a benchmark, #Jumps denotes the number of jmp edges added to its PAG due to data sharing, $\#S$ represents the total number of steps traversed by SeqCfl (without data sharing) for all the queries issued from the benchmark, and R_S is the ratio of the number of steps saved by the jmp edges for the benchmark over the number of steps traversed across the original edges (when data sharing is enabled). For the 20 benchmarks used, 22,023 jmp edges have been added on average per benchmark. The number of steps saved by these jmp edges is much larger than that of the original ones, by a factor of 28.6X on average. This implies that a large number of redundant traversals ($\#S \times \frac{R_S}{R_S+1}$ for a benchmark) have been eliminated. Thus, ParCfl$^{16}_{D}$ exhibits substantial improvements over ParCfl$^{16}_{naive}$, with the superlinear speedups observed in `_202_jess`, `_213_javac`, `_222_mpegaudio`, `batik`, `fop` and `tomcat`.

The optimisation described in Section IV-A for adding jmp edges selectively to reduce synchronisation overhead is also useful for improving the performance. Fig. 7 reveals the

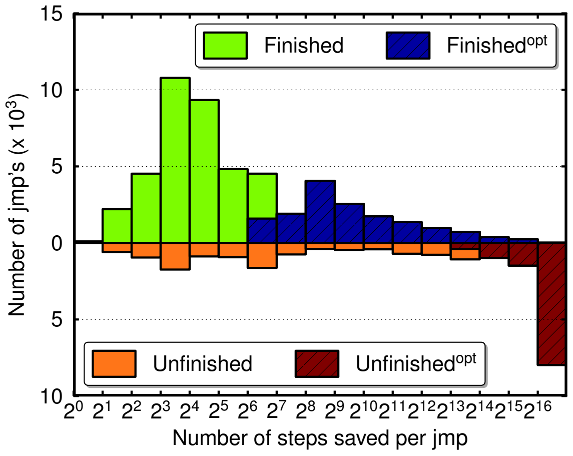


Fig. 7: Histograms of jmp edges (identified by the number of steps saved). Finished represents jmp edges in Fig. 3(a) and Unfinished jmp edges in Fig. 3(b). Finished$^{\text{opt}}$ (Unfinished$^{\text{opt}}$) is the version of Finished (Unfinished) with the selective optimisation described in Section IV-A being applied.

histograms of added jmp edges with and without this optimisation. In the absence of such optimisation, many jmp edges representing relatively short paths are also added, causing ParCfl$^{16}_{DQ}$ to drop from 16.2X to 12.4X on average.

3) *Effectiveness of Query Scheduling:* When query scheduling is also enabled, queries are grouped and reordered to increase early terminations made. ParCfl$^{16}_{DQ}$ achieves superlinear speedups in two more benchmarks than ParCfl$^{16}_{D}$: `avrora` and `sunflow`. ParCfl$^{16}_{DQ}$ is faster than ParCfl$^{16}_{D}$ as the average speedup goes up from 13.4X to 16.2X.

To understand its effectiveness, some statistics are given in Columns 11 – 13 in Table I. For a benchmark, $\overline{S_g}$ gives the average number of queries in a group, $\#ETs$ is the number of early terminations found without query scheduling, and R_{ET} is the ratio of $\#ETs$ obtained with query scheduling over $\#ETs$ obtained without query scheduling. On average, our

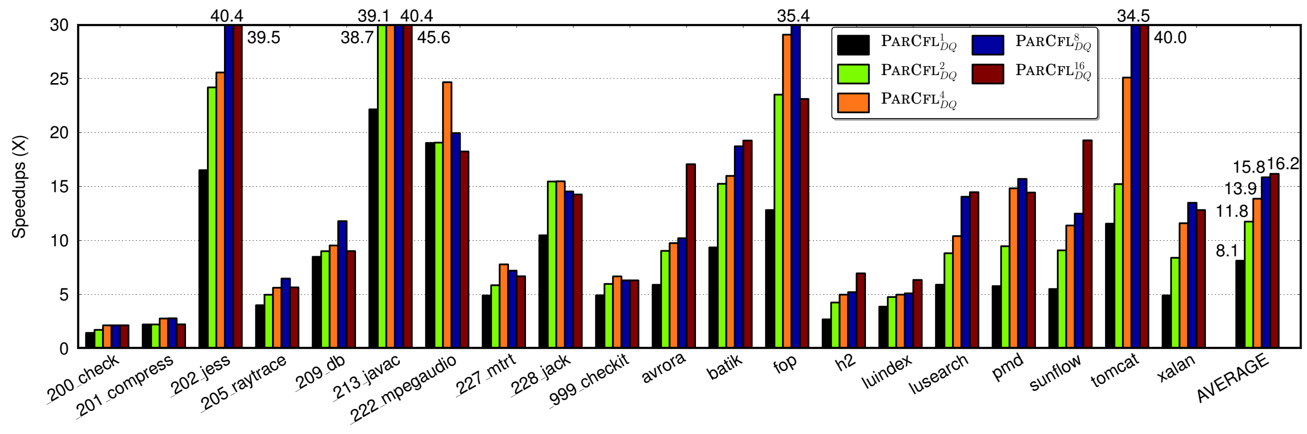


Fig. 8: Speedups of our parallel modes with different numbers of threads normalised with respect to SeqCfl.

query scheduling scheme leads to 35% more early terminations, resulting in more redundant traversals being eliminated.

4) Scalability: To see the scalability of our parallel implementation, Fig. 8 plots its speedups with a few thread counts over the baseline. ParCfl^1_{DQ} achieves an average speedup of 8.1X, due to data sharing and query scheduling. Our parallel solution scales well to 8 threads for most benchmarks. When moving from 8 to 16 threads, ParCfl^{16}_{DQ} suffers some performance drops over ParCfl^{8}_{DQ} in some benchmarks (with `_209_db` being the worst case at 31%). However ParCfl^{16}_{DQ} is still slightly faster than ParCfl^{8}_{DQ} on average.

5) Memory Usage: As garbage collection is enabled, it is difficult to monitor memory usage precisely. By avoiding redundant graph traversals, ParCfl^{16}_{DQ} reduces the memory usage by SeqCfl (the open-source sequential implementation [18]) by 35% (32%) in terms of the peak memory usage, despite the extra memory required for storing jmp edges. In the worst case attained at `tomcat` (`fop`), ParCfl^{16}_{DQ} consumes 103% (118%) of the memory consumed by SeqCfl ([18]).

V. Related Work

A. CFL-Reachability-Based Pointer Analysis

In the sequential setting, there is no shortage of optimisations on improving the performance of demand-driven CFL-reachability-based pointer analysis. To ensure quick response, queries are commonly processed under budget constraints [17], [18], [19], [26], [27]. In addition, refinement-based schemes [18], [19] can be effective for certain clients, e.g., type casting if field-sensitivity is gradually introduced. Summary-based schemes avoid redundant graph traversals by reusing the method-local points-to relations summarised statically [26] or on-demand [17], achieving up to 3X speedups. Must-not-alias information obtained during a pre-analysis can be exploited to yield an average speedup of 3X through reducing unnecessary alias-related computations [25]. Incremental techniques [6], [16], which are tailored for scenarios where code changes are small, take advantage of previously computed CFL-reachable paths to avoid unnecessary reanalysis.

Unlike these efforts on sequential CFL-reachability-based pointer analysis, this paper introduces the first parallel solution on multicore processors with significantly better speedups.

B. Parallel Pointer Analysis

In recent years, there have been a number of attempts on parallelising pointer analysis algorithms for analysing C or Java programs on multi-core CPUs and/or GPUs [3], [7], [8], [9], [10], [14], [20]. As compared in Table II, all these parallel solutions are different forms of Andersen's pointer analysis [2] with varying precision considerations in terms of context-, flow- and field-sensitivity. Our parallel solution is the only one that can be performed on-demand based on CFL-reachability.

Méndez-Lojo *et al.* [8] introduced the first parallel implementation of Andersen's pointer analysis for C programs. While being context- and flow-insensitive, their parallel analysis is field-sensitive, achieving a speedup of up to 3X on eight CPU cores. The same baseline sequential analysis was later parallelised on a Tesla C2070 GPU [7] (achieving an average speedup of 7X with 14 streaming multiprocessors) and a CPU-GPU system [20] (boosting the CPU-only parallel solution by 51% and the GPU-only parallel solution by 79% on average).

Edvinsson *et al.* [3] described a parallel implementation of Andersen's analysis for Java programs, achieving a maximum speedup of 4.4X on eight CPU cores. Their analysis is field-insensitive but flow-sensitive (only partially since it does not perform strong updates). Recently, both field- and flow-sensitivity (with strong updates for singleton objects) are handled for C programs on multi-core CPUs [9] and GPUs [10]. The speedups are up to 2.6X on eight CPU cores and 7.8X (with precision loss) on a Tesla C2070 GPU, respectively.

Putta and Nasre [14] use replications of points-to sets to enable more constraints (i.e., more pointer-manipulating statements) to be processed in parallel. Their context-sensitive implementation of Andersen-style analysis delivers an average speedup of 3.4X on eight CPU cores.

This paper presents the first parallel implementation of demand-driven CFL-reachability-based pointer analysis, achieving an average speedup of 16.2X for a set of 20 Java

Analysis	Algorithm	On-demand	Sensitivity (Precision)			Applications	Platform
			Context	Field	Flow		
[8]	Andersen's [2]	✗	✗	✔	✗	C	CPU
[3]	Andersen's [2]	✗	✗	✗	✓*	Java	CPU
[7]	Andersen's [2]	✗	✗	✔	✗	C	GPU
[14]	Andersen's [2]	✗	✔	✗	✗	C	CPU
[9]	Andersen's [2]	✗	✗	✔	✔	C	CPU
[10]	Andersen's [2]	✗	✗	✔	✔	C	GPU
[20]	Andersen's [2]	✗	✗	✔	✗	C	CPU-GPU
this paper	CFL-Reachability [15]	✔	✔	✔	✗	Java	CPU

$*$: Partial flow-sensitivity without performing strong updates

TABLE II: Comparing different parallel pointer analysis.

programs on 16 CPU cores. Based on a version of CFL-reachability-based pointer analysis for C [27], our parallel solution is expected to generalise to C programs as well.

C. Parallel Graph Algorithms

There are many parallel graph algorithms, including breadth-first search (BFS) [1], [4], [11], minimum-cost path [12] and flow analysis [13]. However, parallelising CFL-reachability-based pointer analysis poses different challenges. The presence of both context- and field-sensitivity that is enforced during graph traversals makes it hard to avoid redundant traversals efficiently, limiting the amount of parallelism exploited (especially within a single query). Exploiting inter-query parallelism, this work has demonstrated significant performance benefits that can be achieved on parallelising CFL-reachability-based pointer analysis on multi-core CPUs.

VI. Conclusion

This paper presents the first parallel implementation of CFL-reachability-based pointer analysis on multi-core CPUs. Despite the presence of redundant graph traversals, this demand-driven analysis is non-trivial to parallelise due to the dependences introduced by context- and field-sensitivity during graph traversals. We have succeeded in parallelising it by using (1) a data sharing scheme that enables the concurrent query-processing threads to avoid traversing earlier discovered paths via graph rewriting and (2) a query scheduling scheme that allows more redundancies to be eliminated based on the dependences statically estimated among the queries to be processed. For a set of 20 Java benchmarks evaluated, our parallel implementation significantly boosts the performance of a state-of-the-art sequential implementation with an average speedup of 16.2X on 16 CPU cores.

Acknowledgment

This work is supported by Australian Research Grants, DP110104628 and DP130101970.

References

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11, 2010.

[2] Lars Ole Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, University of Copenhagen*, 1994.

[3] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. Parallel points-to analysis for multi-core machines. In *HiPEAC*, pages 45–54, 2011.

[4] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, pages 78–88, 2011.

[5] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC*, pages 47–64, 2006.

[6] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with CFL-reachability. In *CC*, pages 61–81, 2013.

[7] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP*, pages 107–116, 2012.

[8] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443, 2010.

[9] Vaivaswatha Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *PACT*, pages 19–28, 2013.

[10] Rupesh Nasre. Time- and space-efficient flow-sensitive points-to analysis. *TACO*, 10(4):39, 2013.

[11] Robert Niewiadomski, José Nelson Amaral, and Robert C. Holte. A parallel external-memory frontier breadth-first traversal algorithm for clusters of workstations. In *ICPP*, pages 531–538, 2006.

[12] Robert Niewiadomski, José Nelson Amaral, and Robert C. Holte. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *AAAI*, pages 1039–1044, 2006.

[13] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary W. Hall. EigenCFA: Accelerating flow analysis with GPUs. In *POPL*, pages 511–522, 2011.

[14] Sandeep Putta and Rupesh Nasre. Parallel replication-based points-to analysis. In *CC*, pages 61–80, 2012.

[15] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, 1998.

[16] Lei Shang, Yi Lu, and Jingling Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: Preliminary experience. In *ASE*, pages 270–273, 2012.

[17] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *CGO*, pages 264–274, 2012.

[18] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[19] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.

[20] Yu Su, Ding Ye, and Jingling Xue. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In *HiPC*, pages 149–158, 2013.

[21] Yulei Sui, Yue Li, and Jingling Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO*, pages 1–11, 2013.

[22] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA*, pages 254–264, 2012.

[23] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107 – 122, 2014.

[24] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

[25] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[26] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA*, pages 155–165, 2011.

[27] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208. ACM, 2008.