Systemized Program Analyses: A Big Data Perspective on Scaling Large-Scale Code Analyses

Harry Xu, Zhiqiang Zuo, Kai Wang, Aftab Hussain, and Khanh Nguyen

University of California, Irvine
{guoqingx, zzuo2, wangk7, aftabh, khanhtn1}@ics.uci.edu

Cover Page

Justification Statement: This paper presents our perspectives on turning program analyses into Big Data problems that are amenable to systems solutions. Traditional approaches to the scalability problem of a static analysis are to raise the abstraction level, attempting to reduce/hide/merge intermidate states of the analysis. Inspired by how large-scale data analytical systems were built to process datasets as large as the whole Internet, we argue that systems solutions are worth considering for program analysis workloads and they do exist if we can reduce the computation complexity of a program analysis algorithm and increase the size of data it operates on. We use the CFL-reachability-based pointer analysis as an example to show how to develop a scalable system that can analyze very large programs.

Attendance Statement: Harry Xu (faculty) and Zhiqiang Zuo (postdoc) will commit to attending the conference upon acceptance. Harry Xu will present the work.

Abstract

"Big Data" has become a central topic in modern computing. To deal with datasets that are too large to process with traditional approaches, applications (*e.g.*, information retrieval or machine learning) have built intimate relationships with systems, resulting in proliferation of large-scale, data-intensive systems tailored for various kinds of data analytics and learning applications. This paper shows that many sophisticated static analyses, which are known to be difficult to scale, can also be converted to Big Data problems that benefit from data analytical systems.

Our work was driven by a Big Data thinking – if we can turn a computationally difficult problem to an equivalent problem with simpler computation over large amounts of data, a systems solution may exist. We show that such a conversion is possible for many static analysis algorithms. Unlike traditional analysis techniques that trade off precision (*i.e.*, usefulness) for scalability, the Big Data treatment of a static analysis advocates to make intermediate states of the analysis explicit, enabling the development of parallel systems that utilize disk and cluster support to achieve efficiency and scalability.

1998 ACM Subject Classification F.3.2 Logics and Meaning of Programs: Semantics of Programming Languages; H.2.4 Database Management Systems: Parallel databases; H.3.4 Information Storage and Retrieval Systems and Software

Keywords and phrases Program analysis, Big Data thinking, disk-based systems, distributed systems

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2017.999

XYZ.
Editor: XX; pp. 1–11

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern computing has entered the Big Data era. Quickly obtaining useful information from extremely large amounts of data coming from all aspects of human lives requires processing capabilities that traditional data processing applications do not have — these applications are often hand-written and tuned by developers, and have assumptions on the execution environment (*e.g.*, memory requirement) that do not hold for real-world datasets. For example, a simple webgraph collected by Yahoo [1] has more than six billion edges and loading it entirely into memory needs at least 153GB of main memory [31], a requirement hard to fulfill even in industry.

To overcome these challenges, many Big Data software systems have been developed to meet different kinds of computational needs [10, 9, 18, 40, 39, 19, 12]. At their core is a "one-stone-two-birds" approach, in which the optimization for scalability is mainly achieved by the (distributed or disk-based) system itself, requiring the developers to only write simple programs using the interfaces provided by the system. Pioneered by MapReduce [9], data processing in each system relies on a computation model that partitions data and parallelizes the processing. Iterative algorithms are often used to resolve data dependencies across partitions. More recently, large-scale distributed systems – including DistBlief [8], Project Adam [7], or TensorFlow [3] – have been developed to tackle emerging machine learning problems, such as training and inference on deep neural networks.

The relationship between systems and applications has never been so intimate in the computing history. However, the PL community has not been in this relationship yet. This paper argues that many program analysis techniques, especially those designed for analyzing large codebases, are important workloads worthy of consideration of systems solutions. We believe that it is time for PL to (re)build intimacy with systems, which may lead to different angles of formulating and solving some of the most difficult problems in the PL community.

1.1 Big Data Thinking

Our research group at UC Irvine, while with a background in program analysis, has been working on Big Data systems for several years. We observe that the term Big Data has often been used only to refer to the fact that data is large and everywhere. However, there is an important aspect of Big Data that is often overlooked, which is – what we call – Big Data thinking. Big Data is not only a phenomenon; but also it defines a unified scheme for solving a class of computational problems with large inputs. We describe the scheme using the following formula:

$$\text{Big Data Solution} = \textit{Large}\ \text{Dataset} + \textit{Simple}\ \text{Computation Model} + \text{System Design} \quad (1)$$

The first two components (dataset and computation model) belong to the application, which can be thought of as parameters to the system design. On one hand, a large amount of data is the motivation; traditional approaches would suffice otherwise. On the other hand, having a simple computation model makes it possible for the computation to be "mechanized" by a system without much user intervention. Most of today's well-known Big Data problems (*e.g.*, graph processing, neural network training, or text analytics) are naturally data-driven — they were born with large data inputs; applying Big Data thinking to these problems seems straightforward. An interesting question is what other problems are amenable to this Big Data scheme? Would Big Data thinking also apply to the problems that do not operate on large datasets?

We observe that it is indeed possible to devise a Big Data solution for many traditional problems that do not appear data-intensive, if there exist means to turn *complex computation* into *large amounts of data*. For example, problems in computational logic and software analysis, such as model checking and constraint solving, often maintain an extremely large number of intermediate states. All the existing techniques attempt to *reduce* this intermediate information (*e.g.*, by employing abstractions or exploiting similarities to merge information), essentially decreasing the size of data at the cost of increased complexity in computation. The design of these techniques stems from the fear of state explosion and the resulting memory blowup, with an implicit assumption the execution environment is a single machine with a limited amount of memory and we do not know what to do if the need of the computation goes beyond that. However, handling large amounts of information is exactly the problem the Big Data community deals with everyday. The recent advancements in data-intensive systems (*e.g.*, the ability of processing graphs of trillions of edges) provide a strong indication that physical resources should no longer be the primary concern in the future algorithm design.

Freeing algorithm designers from worrying about lack of resources enables a critical mindset shift – making the state information *explicit* and letting a system deal with it, as opposed to reducing the information by developing algorithms to abstract or hide it. The large number of states maintained during the computation naturally form a big dataset, essentially converting the problem into a Big Data problem.

1.2 A Big Data Perspective on Program Analysis

Dynamic analysis generates a large amount of profiling data, lending itself naturally to a Big Data solution. In fact, there already exists work [43] that uses MapReduce to analyze logs generated by distributed systems. This paper focuses on static analysis, for which it is not immediately obvious that a Big Data solution would exist. We demonstrate how to apply Big Data thinking to a class of static analyses that can be formulated as the context-free-language (CFL) reachability problem. In particular, we show that, by making transitive edges explicit, a CFL static analysis can be converted to a Big Data problem that has a systems solution. We hope that this formulation can inspire further thoughts on applying Big Data thinking on other program analysis problems.

2 A "Big Data" Formulation of CFL Rechability

Pioneered by Reps et al. [21, 24], there is a large body of work on context free language reachability based program analyses [14, 37, 38, 20, 5, 42, 41, 30]. The program to be analyzed is first translated to a graph representation; the reachability computation is performed on the graph guided by a context-free grammar that encodes the *balanced parentheses* property of an analysis. At a high level, let us suppose each edge in the graph is labeled either an open parenthesis '(' or a close parenthesis ')'. A vertex is reachable from another vertex if and only if there exists a path between them, the string of labels on which has balanced '(' and ')'. The parentheses '(' and ')' have different semantics for different analyses. For example, for a C pointer analysis, '(' represents an address-of operation & and ')' represents a dereference *. A pointer variable can point to an object if there is an assignment path between them that has balanced & and *. For instance, a string "&&**" has balanced parentheses while "&**&" does not.

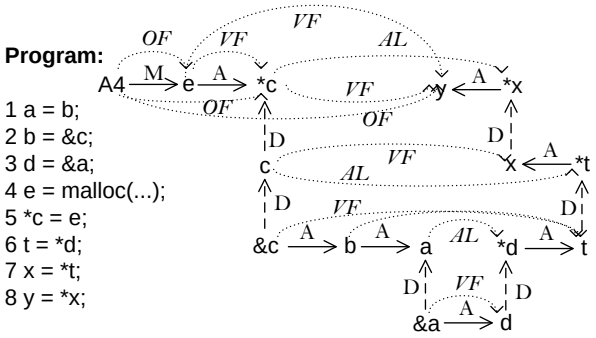

Figure 1 A program and its expression graph: solid, horizontal edges represent assignments (A- and M- edges); dashed, vertical edges represent dereferences (D-edge); dotted, horizontal edges represent transitive edges labeled non-terminals. *A4* indicates the allocation site at Line 4. *OF*, *VF*, and *AL* represent *objectFlow*, *valueFlow*, and *alias*, respectively.

2.1 Background: Pointer Analysis for C

Pointer analysis is a widely used example of CFL-reachability formulation. A pointer analysis computes, for each pointer variable, a set of heap objects (represented by allocation sites) that can flow to the variable. This set of objects is referred to as the variable's *points-to* set. Alias information can be derived from this analysis — if the points-to sets of two variables have a non-empty intersection, they may alias.

We use the formulation in [44] to illustrate the CFL-reachability formulation of a C pointer analysis. The analysis described in this section is *flow-insensitive* in the sense that we do not consider control flow in the program. A program consists of a set of pointer assignments. Assignments can execute in any order, any number of times. For simplicity of presentation, the discussion here focuses on four kinds of three-address statements (which are statements that have at most three operands):

$a = b$	Value assignment	$a = *b$	Load
$*b = a$	Store	$a = \&b$	Address-of

Complicated statements are often broken down into these three-address statements in the compilation process by introducing temporary variables. Our analysis does not distinguish fields in a struct. That is, an expression a-> f is handled in the same way as $*a$, with offset f being ignored. As reported in [44], ignoring offsets only has little influence on the analysis precision, because most fields are of primitive types.

For each function, an *expression graph* – whose vertices represent C expressions and edges represent value flow between expressions — is generated; graphs for different functions are eventually connected to form a whole-program expression graph. Each vertex on the graph represents an expression, and each edge is of three kinds:

- *Dereference edge (D)*: for each dereference $*a$, there is a D-edge from a to $*a$; there is also an edge from an address-of expression $\&a$ to a because a is a dereference of $\&a$.
- *Assignment edge (A)*: for each assignment $a = b$, there is an A-edge from b to a; a and b can be arbitrary expressions.
- *Alloc edge (M)*: for each assignment $a = malloc()$, there is an M-edge from a special Alloc vertex to a.

Figure 1 shows a simple program and its expression graph. Each edge has a label, indicating its type. Solid and dashed edges are original edges in the graph and they are

labeled M, A, or D. Dotted edges are transitive edges that will be discovered by the analysis as computation progresses.

Context-free Grammar The pointer information computation is guided by the following grammar:

Object flow:	*objectFlow*	::=	M *valueFlow*
Value flow:	*valueFlow*	::=	$(A$ *alias*?$)^*$
Expr alias:	*alias*	::=	$\overline{D}$ *valueFlow* D

This grammar has three non-terminals *objectFlow*, *valueFlow*, and *alias*. For a non-terminal T, a path in the graph is called a T-path if the sequence of the edge labels on the path is a string that can be reduced to T. In order for a variable v to point to an object o (*i.e.*, a malloc), there must exist an *objectFlow* path in the expression graph from o to v. The definition of *objectFlow* is straightforward: it must start with an alloc (M) edge, followed by a *valueFlow* path that propagates the object address to variables. A *valueFlow* path is either a sequence of simple assignment (A) edges or a mix of assignments edges and *alias* paths.

An *alias* path is represented by $\overline{D}$ *valueFlow* D. Each edge has an inverse edge with a "bar" label. For example, for each edge $a \xrightarrow{D} b$, the edge $b \xrightarrow{\overline{D}} a$ exists automatically. $\overline{D}$ represents the inverse of a dereference and is essentially equivalent to an address-of. $\overline{D}$ *valueFlow* D represents that if we take the address of a variable a, propagate the address through a *valueFlow* path to another variable b, and then do a dereference on b, the result is the same as the value in a.

Note that *valueFlow* and *alias* mutually refer each other. This definition captures the recursive nature of an *alias* and *valueFlow* path. In this grammar, $\overline{D}$ and D are the open and close parentheses that need to be balanced.

Example In Figure 1, e points to $A4$, since the M edge between them forms an *objectFlow* path. There is a *valueFlow* path from $\&a$ to d, which enables an *alias* path from a to $*d$. This alias path then induces two *valueFlow* paths from b to t and from $\&c$ to t, which, in turn, contribute to the forming of the *valueFlow* paths from c to x, making $*c$ and $*x$ alias. Hence, there exists a *valueFlow* path from e to y, which, together with the M edge at the beginning, forms an *objectFlow* path from $A4$ to y. This path indicates that y points to $A4$. The dotted edges in Figure 1 shows these paths.

2.2 Traditional PL Solution vs. Big Data Solution

Traditional Solution The traditional way to implement this analysis is to maintain a worklist, each element of which is a pair of a newly discovered vertex and a stack simulating a pushdown automaton. The implementation loops over the worklist, iteratively retrieving vertices and processing their edges. The traditional implementation does not add any physical edges into the graph (due to the fear of memory blowup), but instead, it tracks path information using pushdown automata. When a CFL-reachable vertex is detected, the vertex is pushed into the worklist together with the sequence of the labels on the path leading to the vertex. When the vertex is popped off of the list, the information regarding the reachability from the source to the vertex is discarded.

This traditional approach has at least two significant drawbacks. First, it does not scale well when the analysis becomes more sophisticated or the program to be analyzed becomes larger. For example, when the analysis is made *context-sensitive*, the grammar needs to be augmented with parentheses representing method entries/exists; the checking of the balanced property for these parentheses also needs to performed. Since the number

of distinct calling contexts can be very large for real-world programs, naïvely traversing all paths is guaranteed to be not scalable in practice. As a result, various abstractions and tradeoffs [29, 27, 13, 28] have been employed, attempting to improve scalability at the cost of precision as well as implementation straightforwardness. For example, in a widely-used Java pointer analysis [29], more than three quarters of the code is to perform approximations to make sure some results can be returned before a user-given time budget runs out. The base algorithm implementation takes a much smaller portion. This level of implementation and tuning complexity simply does not align with the "simplest-working-solution" [17] philosophy of systems builders, creating a practicality obstacle for static analysis to be used in industry.

Second, the worklist-based model is notoriously difficult to parallelize, making it hard to fully utilize modern computing resources. Even if multiple traversals can be launched simultaneously, since none of these traversals add transitive edges onto the program graph as they are being detected, every traversal performs path discovery completely independently, resulting in a great deal of wasted efforts.

Applying Big Data Thinking Recall that a Big Data solution may exist if the analysis can be formulated as a problem with a large dataset and simple computation. Following the above discussion of making intermediate states explicit (*cf.* §1), we advocate to add *physical* transitive edges into the program graph. In other words, a physical edge labeled E is added from a vertex A to vertex B if there exists a path from A to B whose edge label sequence matches a production in the context-free grammar with E being the non-terminal on the left hand side (LHS) of the production. In addition, inverse edges are also explicitly added into the graph before the analysis starts.

Adding physical edges as the analysis progresses makes it possible to devise a Big Data solution to this static analysis problem. First, representing transitive edges *explicitly* rather than *implicitly* leads to addition of a great number of edges (*e.g.*, even larger than the number of edges in the original graph). This gives us a large (evolving) dataset to process, satisfying the dataset component in Formula (1). Second, the computation only needs to match the labels of consecutive edges with the productions in the grammar and is thus simple enough to be "systemized". This satisfies the second component in Formula (1). Of course, dynamically adding many edges can make the computation quickly exhaust the main memory. However, this should not be a concern, since there are already many systems [18, 15, 34, 22, 11, 31] built to process very large graphs (*e.g.*, the webgraph for the whole Internet).

Existing Datalog and Database-backed Analyses: Treating Execution Engine as A Blackbox Recent work [36, 6] shows the effectiveness of expressing static analyses as Datalog programs. While leveraging Datalog makes analysis implementations easier, program analysis researchers often treat Datalog as a blackbox. Existing Datalog engines such as LogicBlox [2], Socialite [16], Myria [32], and BigDatalog [26] are designed for general-purpose relational algebra – rule evaluation is often implemented as table joining. While there exists a large body of work on efficient table joining in the database community, computing dynamic transitive closure on a program graph is a very special case where the input tables are exactly the same (representing existing edges) and the resulting table contains only a small addition of newly discovered transitive edges. This high degree of commonalities between input and output implies a large optimization space. Recently Weiss et al. [35] developed a "middle layer" that encodes dataflow analyses as graph problems tuned for database-backed interfaces. While we share the same goal of using a system to manage the resource usage of a static analysis, Weiss et al. relied completely on an existing Semantic Web database. This paper advocates to build execution engines tailored for the need of program analysis, instead of treating them as blackboxes as is done by most program analysis researchers.

2.3 Graspan: An Out-Of-Core System for Parallel Dynamic Transitive Closure Computation

This subsection discusses the design of an out-of-core system, Graspan, to fulfill the third component of Formula (1). Before we developed Graspan, the question we asked was whether any existing systems could be used to process program graphs. Unfortunately, we soon realized that a ground-up redesign (*i.e.*, from the programming model to the runtime engine) was needed to build a graph system for analyzing large programs. The main reason is that the graph workload for static analyses is significantly different from a regular graph algorithm (such as PageRank) that iteratively performs computations on vertex values on a static graph. A CFL-reachability-based analysis, on the contrary, focuses on computing reachability by repeatedly adding transitive edges, rather than on updating vertex values.

The dynamic transitive closure computation in the static analysis workload dictates two important abilities of the graph system. First, at each vertex, all its incoming and outgoing edges need to be visible to perform label matching and edge addition. For example, when vertex b is processed, both $a \xrightarrow{l_1} b$ and $b \xrightarrow{l_2} c$ need to be accessed to add the edge from a to c. This requirement immediately excludes edge-centric systems such as XStream [23] from our consideration, because these systems stream in edges in a random order and, thus, this pair of edges may not be simultaneously available. Second, the system needs to support a great number of edges added dynamically. In the presence of many dynamically added edges, it is critical that the system is able to (1) quickly check edge duplicates and (2) appropriately repartition the graph. Unfortunately, existing systems support neither of these features.

We have developed Graspan, a *single machine, disk-based* parallel graph processing system tailored for CFL-reachability-based static analyses. Since program analysis is intended to assist developers to find bugs in their daily development tasks, their machines are the environments in which we would like our system to run, so that developers can check their code on a regular basis without needing to access a cluster. Hence, a disk-based system became our choice.

Given a program graph and a grammar specification of an analysis, Graspan offers two major performance and scalability benefits: (1) the core computation of the analysis is automatically parallelized and (2) out-of-core support is exploited if the graph is too big to fit in memory. Graspan has three major phases: preprocessing, *edge-pair* (EP) centric computation, and post processing. Preprocessing partitions the graph into multiple partitions, each of which is a disk file containing a list of edges whose source vertices belong to an interval and that are sorted based on source vertex IDs. Each edge in the file carries a four-byte data field storing its label.

At the heart of Graspan is the parallel EP centric computation model that, in each iteration, loads two partitions of edges into memory and "joins" their edge lists to produce a new edge list. For example, if $a \xrightarrow{i} b$ is in the first partition and $b \xrightarrow{j} c$ is in the second partition and production $K := i \;\; j$ exists in the grammar, edge $a \xrightarrow{K} c$ is appended to the adjacency list of a in the first partition. The joining of these two partitions is done based on a min-heap algorithm [4] that merges, for each vertex a and each of its outgoing edges $a \xrightarrow{i} b$, all of b's out-neighbors. This step automatically performs label matching and filters out duplicate edges. Merging for multiple source vertices (*e.g.*, a) can be performed completely in parallel without any synchronization.

Graspan uses a novel scheduling algorithm to determine (1) which two partitions to load at each time and (2) whether the computation can be terminated. Partition loading favors partitions that are already in memory and those that have the best matching rates

(*i.e.*, how many edges in one partition whose target vertices are the source vertices of the edges in another partition). The computation is terminated when no new edge can be added between any two partitions. If too many edges are added in one single partition, Graspan repartitions the graph to achieve load balancing. After the computation is done, the postprocessing step provides the translation from graph vertices and edges back to variables and statements in the source code of the program. Edges labeled the start non-terminal represent the solution of the analysis. An additional benefit provided by Graspan is that at the end of the computation, the graph contains not only the final solution, but also all intermediate results of the analysis. For example, after Graspan's computation for the C pointer analysis described above, `objectFlow`, `valueFlow`, and `alias` solutions are all available in the graph. Edges labeled `objectFlow` represent the points-to relation while edges labeled `alias` represent the alias relation. Graspan does not need to perform any further computation to obtain one kind of information (*e.g.*, alias) from another kind (*e.g.*, points-to), as is done by the existing techniques.

Use Graspan We have implemented fully context-sensitive pointer/alias and dataflow analysis on Graspan. Context-sensitivity is achieved by making aggressive inlining [25]. That is, we clone the body of a function for every single context leading to the function. This approach is feasible only because the out-of-core support in Graspan frees us from worrying about additional memory usage incurred by inlining. We treat the functions in recursions *context insensitively* by merging functions in each strongly connected component on the call graph into one function without cloning function bodies. The context-free grammar for the pointer analysis is adopted from [44] and already discussed above. The grammar for the dataflow analysis is adopted from Reps' interprocedural, finite, distributive, subset (IFDS) formulation of dataflow problems [21].

Since Graspan performs edge-pair centric computation that inspects a pair of edges at a time, the user-defined grammar first needs to be normalized to an equivalent grammar in which the RHS of each production contains at most two terms (terminal or non-terminal), similar to the Chomsky normal form. At the center of Graspan's programming model is an API – `addConstraint`(`Label` *lhs*, `Label` *rhs1*, `Label` *rhs2*) – which can be used by the developer to register each production in the grammar. *lhs* represents the LHS non-terminal while *rhs1* and *rhs2* represent the two RHS terms. If the RHS has only one term, *rhs2* should be NULL. Hence, the only work for the analysis developer is to modify a compiler frondend to generate the graph and specify the grammar; no tuning is needed for scalability.

2.4 Current Status of Graspan

Graspan has both a Java and a C++ version: it was first implemented in Java and later adapted to C++ for performance. We ran Graspan (C++ version) on a low-end Dell desktop (with a quad-core 3.2GHZ Intel i5-4570 CPU, 8GB memory, and a 1TB SSD, running Linux 4.2.0) to process program graphs generated by the fully context-sensitive pointer analysis described above. Table 1 shows the programs we analyzed and a set of time comparisons among Graspan-based, the traditional worklist-based, and the Datalog-based implementations of the pointer analysis algorithm. For the worklist-base algorithm, we implemented the context-sensitive version of Zheng and Rugina's C pointer analysis [44] ourselves. We took the expression graph generated by our frontend and used a worklist-based algorithm to compute transitive closures. We used SociaLite [16] as our Datalog engine, which was developed at Stanford and shown to outperform LogicBlox [2] and other shared-memory Datalog engines.

The worklist-based algorithm either ran out of memory or took a very long time (longer than one day) on the same desktop where we ran Graspan. For example, when processing

Program	Ver	#LoC	#Inlines	GraphSize	GTime	ATime	STime
Linux kernel	4.4.0-rc5	16M	31.7M	B: (249.50M, 52.88M) A: (1.12B, 52.88M)	1.67 hrs	OOM	OOM
PostgreSQL	8.3.9	700K	290820	B: (24.97M, 5.20M) A: (842.18M, 5.20M)	5.96 hrs	> 1 day	OOM
Apache httpd	2.2.18	300K	58269	B: (8.19M, 1.72M) A: (904.34M, 1.72M)	8.43 hrs	> 1 day	OOM

Table 1 Programs analyzed, their versions, numbers of lines of code, numbers of function inlines, sizes of their program graphs (before (**B**) and after (**A**) computation), Graspan processing time (GTime), time used by a traditional worklist algorithm (ATime), and Datalog (SociaLite) processing time (STime).

Linux, it ran out of memory in 13 minutes. When we moved it onto a server with 32 2.60GHZ Xeon(R) processors and 32GB memory, it took this implementation 3.5 days to analyze Linux and it consumed 29GB out of the 32GB memory. On the contrary, Graspan finished processing Linux in 1.67 hours with less than 6GB memory on the desktop with a much less powerful CPU (due to exploited parallelism and disk support). Datalog engines such as SociaLite clearly could not scale to graphs that cannot fit into memory. For both the pointer/alias and the dataflow analysis, it ran out of memory for Linux and PostgreSQL. For httpd, although SociaLite processed the graphs successfully, SociaLite took much longer than Graspan.

The implementations of Graspan are publicly available at `https://github.com/Graspan`. The full technical description of Graspan can be found in our ASPLOS'17 paper [33].

3 Conclusions and Future Work

In this paper, we describe a Big Data perspective on scaling static program analysis to large codebases. We demonstrate how to apply Big Data thinking to formulate CFL-reachability based static analyses and the design of the Graspan system that utilizes out-of-core support to parallelize and scale static analysis workloads.

There are many computationally difficult problems that can benefit from a similar Big Data treatment. One example is SAT solving, which is fundamental to many software analysis and verification tasks. We are currently developing a Spark-based distributed SAT solver. By making intermediate resolvents explicit, SAT solving can also be converted to a Big Data solution. As another example, many sophisticated static analyses rely on constraint solving. Examples are path-sensitive type-state analysis, symbolic execution, and various verification/synthesis tasks. An immediate follow-up of Graspan is to extend its edge-pair-centric computation model to support analysis clients that depend on constraint solving. For instance, constraints can be encoded as edge values and two edges match as long as the conjunction of the constraints they carry has a satisfiable solution (as determined by a constraint solver).

References

1 Yahoo! webscope program. http://webscope.sandbox.yahoo.com/.

2 The LogicBlox Datalog engine. `http://www.logicblox.com/`, 2016.

3 Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

4 M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. Min-max heaps and generalized priority queues. *Commun. ACM*, 29(10):996–1000, 1986.

5 Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566, 2015.

6 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262, 2009.

7 Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

8 Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.

9 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

10 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

11 Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.

12 Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

13 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, pages 423–434, 2013.

14 John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, pages 207–218, 2004.

15 Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi : Large-scale graph computation on just a PC. In *OSDI*, pages 31–46.

16 Monica S. Lam, Stephen Guo, and Jiwon Seo. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289, 2013.

17 Butler W. Lampson. Hints for computer system design. In *SOSP*, pages 33–48, 1983.

18 Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

19 Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

20 J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

21 T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

22 Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP*, pages 410–424, 2015.

23 Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP*, pages 472–488, 2013.

24 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

25 M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

26 Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *SIGMOD*, pages 1135–1149, 2016.

27 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *POPL*, pages 17–30, 2011.

28 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *PLDI*, pages 485–495, 2014.

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

30 Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *POPL*, pages 83–95, 2015.

31 Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *USENIX ATC*, pages 507–522, 2016.

32 Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

33 Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*, 2017.

34 Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC*, pages 387–401, 2015.

35 Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis for scalable error propagation. In *ICSE*, pages 586–597, 2015.

36 John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

37 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.

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

39 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.

40 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. HotCloud, page 10, Berkeley, CA, USA, 2010.

41 Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *PLDI*, pages 435–446, 2013.

42 Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for C. In *OOPSLA*, pages 829–845, 2014.

43 Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. Lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI*, pages 629–644, 2014.

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