D4: Fast Concurrency Debugging with Parallel Differential Analysis

Bozhen Liu
Parasol Laboratory
Texas A&M University
USA
april1989@tamu.edu

Jeff Huang
Parasol Laboratory
Texas A&M University
USA
jeff@cse.tamu.edu

Abstract

We present D4, a fast concurrency analysis framework that detects concurrency bugs (e.g., data races and deadlocks) *interactively* in the programming phase. As developers add, modify, and remove statements, the code changes are sent to D4 to detect concurrency bugs in real time, which in turn provides immediate feedback to the developer of the new bugs. The cornerstone of D4 includes a novel system design and two novel parallel differential algorithms that embrace both change and parallelization for fundamental static analyses of concurrent programs. Both algorithms react to program changes by memoizing the analysis results and only recomputing the impact of a change in parallel. Our evaluation on an extensive collection of large real-world applications shows that D4 efficiently pinpoints concurrency bugs within 100ms on average after a code change, several orders of magnitude faster than both the exhaustive analysis and the state-of-the-art incremental techniques.

CCS Concepts • **Software and its engineering** ■ **Software testing and debugging**; **Integrated and visual development environments**;

Keywords Concurrency Debugging, Real Time, Data Races, Deadlocks, Parallel Differential Analysis, Differential Pointer Analysis, Static Happens-Before Analysis.

ACM Reference Format:
Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192390

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192390

1 Introduction

Writing correct parallel programs is notoriously challenging due to the complexity of concurrency. Concurrency bugs such as data races and deadlocks are easy to introduce but difficult to detect and fix, especially for real-world applications with large code bases. Most existing techniques [13, 16, 17, 22, 26, 28, 31–33, 39, 40] either miss many bugs or cannot scale. A common limitation is that they are mostly designed for *late phases* of software development such as testing or production. Consequently, it is hard to scale these techniques to large software because the whole code base has to be analyzed. Moreover, it may be too late to fix a detected bug, or too difficult to understand a reported warning because the developer may have forgotten the coding context to which the warning pertains.

One promising direction to address this problem is to detect concurrency bugs *incrementally* in the programming phase, as explored by our recent work ECHO [7]. Upon a change in the source code (insertion, deletion or modification), instead of exhaustively re-analyzing the whole program, one can analyze the change only and recompute the impact of the change for bug detection by memoizing the intermediate analysis results. This not only provides early feedback to developers (which reduces the cost of debugging), but also enables efficient bug detection by amortizing the analysis cost.

Despite the huge promise of this direction, a key challenge is how to scale to large real-world applications. Existing incremental techniques are still too slow to be practical. For instance, in our experiments with a collection of large applications from the DaCapo benchmarks [8], ECHO takes over half an hour to analyze a change in many cases. A main drawback is that existing incremental algorithms are either inefficient or inherently sequential. In addition, the existing tool runs entirely in the same process as the integrated development environment (IDE), which severely limits the performance due to limited CPU and memory resources.

In this paper, we propose D4, a fast concurrency analysis framework that detects concurrency bugs (e.g., data races and deadlocks) *interactively* in the programming phase. D4 advances IDE-based concurrency bug detection to a new level such that it can be deployed non-intrusively in the

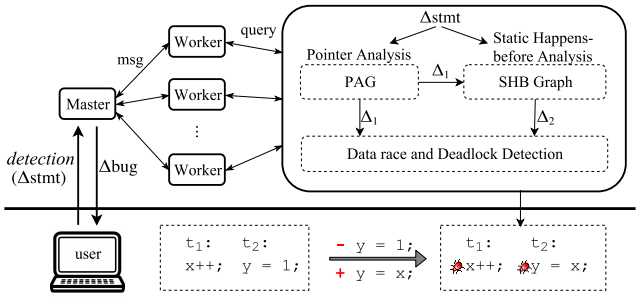


Figure 1. Architectural overview of D4.

development environment for read-world large complex applications. D4 is powered by two significant innovations: a novel system design and two novel incremental algorithms for concurrency analysis.

At the system design level, different from existing techniques which integrate completely into the IDE, D4 separates the analysis from the IDE via a client-server architecture, as illustrated in Figure 1. The IDE operates as a client which tracks the code changes on-the-fly and sends them to the server. The server, which may run on a high-performance computer or in the cloud with a cluster of machines, maintains a change-aware data structure and detects concurrency bugs incrementally upon receiving the code changes. The server then sends the detection results immediately back to the client, upon which the client warns the developer of the newly introduced bugs or invalidates existing warnings.

At the technical level, D4 is underpinned by two parallel incremental algorithms, which embrace both *change* and *parallelism* for pointer analysis and happens-before analysis, respectively. We show that these two fundamental analyses for concurrent programs, if designed well with respect to code changes, can be largely *parallelized* to run efficiently on parallel machines. As shown in Figure 1, D4 maintains two change-aware graphs: a pointer assignment graph (PAG) for pointer analysis and a static happens-before (SHB) graph for happens-before analysis. Upon a set of code changes, Δ_{stmt}, the PAG is first updated and its change Δ_1 is propagated further. Taking Δ_{stmt} and Δ_1 as input, the SHB graph is then updated incrementally and its change Δ_2 together with Δ_1 are propagated to the bug detection algorithms.

D4 can be extended to detect a wide range of concurrency bugs incrementally, since virtually all interesting static program analyses and concurrency analyses rely on pointer analysis and happens-before analysis. For example, the same race checking algorithm in ECHO can be directly implemented based on the SHB graph, and deadlock detection can be implemented by extending D4 with a lock-dependency graph, which simply tracks the lock/unlock nodes in the SHB graph. D4 can also be extended to analyze pull requests in the cloud. For example, in continuous integration of large software, D4 can speed up bug detection by analyzing the committed changes incrementally.

We have implemented both data race detection and deadlock detection in D4 and evaluated its performance extensively on a collection of real-world large applications from DaCapo. The experiments show dramatic efficiency and scalability improvements: by running the incremental analyses on a dual 12-core HPC server, D4 can pinpoint concurrency bugs within 100ms upon a statement change on average, 10X-2000X faster than ECHO and over 2000X faster than exhaustive analysis.

We note that exploiting change and parallelism simultaneously for concurrency analysis incurs significant technical challenges with respect to performance and correctness. Although previous research has exploited parallelism in pointer analyses [15, 24, 25, 29, 36], change and parallelism have never been exploited together. All existing parallel algorithms assume a static whole program and cannot handle dynamic program changes. D4 addresses these challenges by carefully decomposing the entire analysis into parallelizable graph traversal tasks while respecting task dependencies and avoiding task conflicts to ensure the analysis soundness.

In sum, this paper makes the following contributions:

- We present the design and implementation of a fast concurrency analysis framework, D4, that detects data races and deadlocks interactively in the IDE, *i.e.*, in a hundred milliseconds on average after a code change is introduced into the program.
- We present two novel parallel differential algorithms for efficiently analyzing concurrent programs by exploiting both the change-centric nature of programming and the algorithmic parallelization of fundamental static analyses.
- We present an extensive evaluation of D4 on real-world large applications, demonstrating significant performance improvements over the state-of-the-art.
- D4 is open source [5]. All source code, benchmarks and experimental results are publicly available at

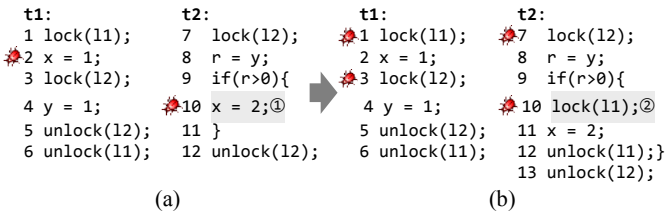
  https://github.com/parasol-aser/D4

2 Motivation and Challenges

In this section, we first use an example to illustrate the problem and the technical challenges. Then, we introduce existing algorithms and discuss their limitations.

2.1 Problem Motivation

Consider a developer, Amy, who is working on the Java program shown in Figure 2(a). The program consists of two threads t1 and t2, and two shared variables x and y. As soon as Amy inserts a write ① x=2 to t2 and saves the program in the IDE, D4, which runs in the background, will prompt a data race warning on lines 2 and 10, similar to syntax error checking.



(a) (b)

Figure 2. A motivating example. (a) a data race between lines (2,10) is detected by D4 when Change ① at line 10 is introduced; (b) a new deadlock between lines (1,3;7,10) is detected by D4 when Change ② at lines 10 and 12 is introduced which attempts to fix the race.

As Amy sees the warning, she can analyze and fix the bug immediately without waiting until it is found by a test or a code reviewer, or the bug happens in production. To eliminate the data race, Amy might want to introduce a lock l1 to protect the write to x at line 10. This fixes the data race, however, it introduces a deadlock between the lock pairs at lines (1,3;7,10) in Figure 2(b). Nevertheless, this deadlock is again instantly reported by D4 to guide Amy to fix the bug.

To realize D4 as above, there are three requirements:

1. We need to identify the two threads, the shared data x and y, and the two locks l1 and l2, *i.e.*, they refer to different locks.
2. We need to identify that the operations by the two threads can execute in parallel, *i.e.*, one does not always happen before the other, and the four lock operations may have circular dependencies.
3. To not interrupt Amy, D4 must be very fast, *i.e.*, it finishes within a sub-second time.

For the first two requirements, we need a pointer analysis and a happens-before analysis. For the third requirement, we must develop an efficient algorithm that can leverage these analyses to detect data races and deadlocks.

2.2 Existing Algorithms

Previous work [7] has proposed sequential incremental pointer analysis and happens-before algorithms for data race detection. Although these incremental algorithms are much more efficient than the exhaustive analysis, they are not efficient enough for large software.

2.2.1 Incremental Pointer Analysis

Many pointer analysis algorithms are based on the on-the-fly Andersen's algorithm [19, 20], which constructs a pointer assignment graph (PAG) and iteratively computes an inclusion-based graph closure until reaching a fixed point. Each program variable corresponds to a node in the PAG, and each variable assignment corresponds to one or more edges. There are two types of nodes in the PAG: *pointer node*s denoting pointer or reference variables, and *object node*s denoting

Figure 3. An example of pointer assignment graph (PAG).

memory locations or objects. Each pointer node is associated with a points-to set denoted by *pts*, which contains the set of object nodes that the pointer node may point to. Each edge represents a subset constraint between the points-to sets, *i.e.*, $p \rightarrow q$ means $pts(p) \subseteq pts(q)$. Figure 3 shows an example of PAG. x and y are pointer nodes, o_1 and o_2 are object nodes; $pts(x) = \{o_1\}$ and $pts(y) = \{o_1, o_2\}$.

Incremental pointer analysis must handle two types of changes: *inserting* a statement and *deleting* a statement[1]. Handling insertion is straightforward based on the on-the-fly Andersen's algorithm. However, deletion is much more complicated than insertion to handle. For deletion, one has to maintain provenance information on how facts are derived. When a statement is deleted, one has to delete all facts that are "no longer reachable" from existing statements through the provenance information. Consider three consecutive code changes: inserting a statement b=a, inserting another statement c=b, and deleting the first statement b=a. When b=a is inserted, $pts(b)$ is updated to $pts(b) \cup pts(a)$. When c=b is inserted, similarly, $pts(c)$ is updated to $pts(c) \cup pts(b)$. However, when b=a is deleted, not only the change in $pts(b)$ should be reversed, but also that the change in $pts(c)$ should be *recomputed*, because $pts(c)$ was previously updated based on $pts(b)$.

We refer the readers to [7] for the grammar and detailed rules for constructing the PAG for different types of program statements in Java, as they are orthogonal to our discussion of D4's novelty. Next, we focus on discussing the two existing incremental algorithms for handling deletion: *reset-recompute* [11, 23, 30, 37], and *reachability-based* [7, 34]. Both of them suffer from performance limitations.

Reset-recompute algorithm Upon a deletion, one can first remove from the PAG all edges related to the deleted statement and reset the points-to sets of their destination nodes as well as all nodes that they can reach (because the points-to sets of all those nodes may be affected). Then, for all the reset nodes, extract their associated points-to constraints and rerun the on-the-fly Andersen's algorithm.

Consider an example in Figure 4, in which an edge $x \rightarrow y$ is deleted from the PAG (e.g., due to the deletion of a statement $y = x$ in the program). The root variable of the change is y, since its points-to set may be changed immediately because of the edge deletion. The reset-recompute algorithm first resets $\mathrm{pts}(y)$ as well as $\mathrm{pts}(z)$ and $\mathrm{pts}(w)$ to empty (because z and w are reachable from y). Then it extracts the

[1] All code changes can be represented by insertion and deletion. E.g., modification can be treated as deletion of the old statement and insertion of the new statement. Large code chunks can be treated as a collection of small changes.

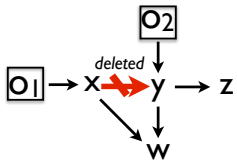


Figure 4. An example of edge deletion in the PAG.

points-to constraints $\text{pts}(y) = \text{pts}(y) \cup \{o_2\}$, $\text{pts}(z) = \text{pts}(z) \cup \text{pts}(y)$, $\text{pts}(w) = \text{pts}(w) \cup \text{pts}(y)$, and $\text{pts}(w) = \text{pts}(w) \cup \text{pts}(x)$, from the four edges connected to the three reset nodes, *i.e.*, $o_2 \rightarrow y$, $y \rightarrow z$, $y \rightarrow w$ and $x \rightarrow w$, and recomputes $\text{pts}(y)$, $\text{pts}(z)$ and $\text{pts}(w)$ until reaching a fixed point. The final values of the points-to sets are: $\text{pts}(x)=\{o_1\}$, $\text{pts}(y)=\{o_2\}$, $\text{pts}(z)=\{o_2\}$ and $\text{pts}(w)=\{o_1, o_2\}$.

The reset-recompute algorithm is inefficient because most computations on the points-to sets of the reset nodes could be redundant. For example, both before and after the deletion, $\text{pts}(w)$ remains the same and o_2 is included in the points-to sets of y, z and w.

Reachability-based algorithm Before removing an object node from a points-to set, one can first check the path reachability from the object node to the pointer node. In this way, the points-to sets of those nodes that are potentially affected by the deletion are not reset, but are updated lazily only if they are not reachable from the object nodes. This algorithm does not incur any redundant computation on the points-to set. However, it requires repeated whole-graph reachability checking, which is expensive for large PAGs.

Consider again the example in Figure 4. Upon the deletion of the edge $x \rightarrow y$, the algorithm first checks if x is still reachable to y (*i.e.*, via another path without $x \rightarrow y$). If yes, then the algorithm stops with no changes to any points-to set. Otherwise, it goes on to check if any object in $\text{pts}(x)$ should be removed from $\text{pts}(y)$, by checking if the corresponding object node can reach y in the PAG. In this case, $\text{pts}(x)$ contains only o_1 which cannot reach y, hence o_1 is removed from $\text{pts}(y)$. Because $\text{pts}(y)$ is changed, the algorithm then continues to propagate the change by checking the nodes connected to y (*i.e.*, z and w). Finally, because o_1 cannot reach z but can reach w (via the path $o_1 \rightarrow x \rightarrow w$), o_1 is removed from $\text{pts}(z)$ but $\text{pts}(w)$ remains unchanged.

The main scalability bottleneck of the reachability-based algorithm is that the worst case time complexity for checking path reachability is linear in the PAG size, which can be very large for real-world programs. For instance, in our experiments (§ 5) the PAG of the h2 database contains over 300M edges, even with some JDK libraries excluded. The performance can be improved by parallelizing the reachability check for different object nodes, however, the time complexity is still linear in the PAG size.

Table 1. Nodes in the SHB graph.

Statements	Nodes
❶* $x = y.f$	$write(x), \forall O_c \in pts(y) : read(O_c.f)$
❷* $x.f = y$	$read(y), \forall O_c \in pts(x) : write(O_c.f)$
❸ $synchronized(x)\{\ldots\}$	$\forall O_c \in pts(x) : lock(O_c), unlock(O_c)$
❹ $o.m(\ldots)$	$\forall O_c \in pts(o) : call(O_c.m)$
❺ $t.start()$	$\forall O_c \in pts(t) : start(O_c)$
❻ $t.join()$	$\forall O_c \in pts(t) : join(O_c)$

* ❶ and ❷ also represent array read ($x = y[i]$) and write ($x[i] = y$), resp.

Table 2. Edges in the SHB graph.

Statements	Edges
❹ $x = o.m(\ldots)$	$\forall O_c \in pts(o) : call(O_c.m) \rightarrow FirstNode(O_c.m)$ $LastNode(O_c.m) \rightarrow NextNode(call)$*
❺ $t.start()$	$\forall O_c \in pts(t) : start(O_c) \rightarrow FirstNode(O_c)$
❻ $t.join()$	$\forall O_c \in pts(t) : LastNode(O_c) \rightarrow join(O_c)$

* NextNode(call): the consecutive node of the method call statement.

2.2.2 Incremental Happens-Before Analysis

The existing technique [7] uses a static happens-before (SHB) graph to compute happens-before relation among abstract threads, memory accesses, and synchronizations. The SHB graph for Java programs is constructed incrementally following the rules in Table 1. Among them, statements ❹ (method *call*), ❺ (thread *start*) and ❻ (thread *join*) generate additional edges according to Table 2. The SHB graph is represented by sequential traces containing per-thread nodes in the SHB graph following the program order, connected by inter-thread happens-before edges. For race detection, the happens-before relation between nodes from different threads can then be computed by checking the graph reachability.

Large SHB graph A crucial limitation of this approach is that for large software it can produce a prohibitively large SHB graph. During the graph construction, when a method is invoked, it has to analyze the method and creates new nodes for statements inside the method. If a method is invoked multiple times (invoked repeatedly by a thread, occurs in a loop, or by multiple threads), multiple nodes representing the same statement will be created and inserted into the SHB graph.

Expensive graph update Updating the SHB graph with respect to code changes can be very expensive. Existing technique uses a map to record each method call and its corresponding location in the SHB graph. If there is a statement change in a method, all the matching nodes in the graph must be tracked and updated. For large software, this incurs significant repetitive computation because a changed method can be invoked many times.

3 D4: A Fast Framework

D4 is powered by three major contributions:

1. **A new incremental algorithm for pointer analysis** that leverages local neighboring properties of the

PAG for efficient incremental pointer analysis. Moreover, it can be parallelized to achieve orders of magnitude speedup over existing incremental algorithms.
2. **A new parallel incremental algorithm for happens-before analysis** that leverages a new representation of the SHB graph, which significantly reduces redundant computations caused by repeated identical method calls.
3. **A new system design and parallelization** that overcomes the scalability limitation of existing work. It may appear straightforward to extend ECHO with a client-server architecture, but realizing this idea requires careful design of the whole system.

In this section, we present the technical details of these algorithms and the system design.

3.1 Parallel Incremental Pointer Analysis

Our new algorithm is based on a fundamental *transitivity* property of Andersen's analysis. This enables us to prove two key properties of the PAG, which allow us to develop an efficient incremental algorithm without any redundant computation. We further prove a *change consistency* property of the PAG, which allows us to massively parallelize our algorithm.

Transitivity of PAG: For an object node o and a pointer node p in the PAG, $o \in pts(p)$ *iff* o can reach p. For two pointer nodes p and q, if p can reach q in the PAG, then $o \in pts(q)$ because of $pts(p) \subseteq pts(q)$.

Consider an *acyclic* PAG, *i.e.*, all strongly connected components (SCCs) are collapsed into a single node (SCCs can be handled by existing techniques [21]), and consider a pointer node q of which an object $o \in pts(q)$. We can prove the following property:

P1: Incoming neighbours property: If q has an incoming neighbour r (*i.e.*, there exists an edge $r \to q$) and $o \in pts(r)$, then o can reach r without going through q.

Proof. Consider an example Figure 5. First, because $o \in pts(r)$, due to transitivity, o can reach r. Second, there cannot exist a path $o \to \ldots \to q \to \ldots \to r \to q$ because the PAG is assumed to be acyclic.

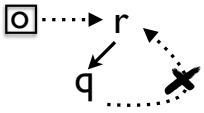


Figure 5. Illustration of the incoming neighbours property.

Suppose an edge $p \to q$ is deleted from an acyclic PAG and all the other edges remain unchanged, based on P1, we can prove the following theorem:

Theorem 1: *For any object $o \in pts(q)$, if there exists an incoming neighbour r of q such that $o \in pts(r)$, then o remains in $pts(q)$. Otherwise, if q does not have any incoming neighbour of which the points-to set contains o, then o should be removed from $pts(q)$.*

Proof. Due to P1, o can reach r without going through q. Hence, o can reach r without the edge $p \to q$. Because $r \to q$, o can hence reach q without the edge $p \to q$. Therefore, o remains in $pts(q)$ after deleting $p \to q$. Otherwise if no neighbour has a points-to set containing o, then o cannot reach q and hence should be removed from $pts(q)$.

With Theorem 1, to determine if a deleted edge introduces changes to the points-to information, we only need to check the incoming neighbours of the deleted edge's destination, which is much faster than traversing the whole PAG for checking the path reachability. Consider again the example in Figure 4. Upon deleting the edge $x \to y$, we only need to check o_2, which is the only incoming neighbour of y. Because the points-to set of o_2 does not contain o_1, o_1 should be removed from $pts(y)$.

Once the points-to set of a node is changed, the change must be propagated to all its outgoing neighbors. Again, based on transitivity, we can prove the following property:

P2: Outgoing neighbours property: If q has an outgoing neighbour w (*i.e.*, there exists an edge $q \to w$) and w has an incoming neighbour r (different from q) such that $o \in pts(r)$, then either o can reach w without going through q, or q can reach r.

Proof. Consider an example in Figure 6, which represents two scenarios that satisfy the predicate in P2. There must exist a path from o to w because $o \in pts(r)$ and $r \to w$, and the path must be $o \to \ldots \to r \to w$. The path may or may not contain q. However, if it contains q, then it must be $o \to \ldots \to q \to \ldots \to r \to w$, which means that q can reach r.

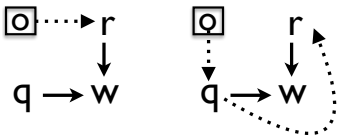


Figure 6. Illustration of the outgoing neighbours property.

Based on P2, if the path from o to w does not contain q, then o should remain in $pts(w)$, because $pts(r)$ cannot be affected by the change in q. On the other hand, if the path contains q, $pts(r)$ may change. Nevertheless, since q can reach r, the change will propagate to r and hence to w eventually. In either case, we only need to check the points-to sets of the incoming neighbours of w for change propagation. Therefore, we can prove the following theorem:

Theorem 2: *To propagate a change to a node, it is sufficient to check the other incoming neighbours of the node. If the points-to set of any incoming neighbour contains the change, the node can be skipped. Otherwise, the change should be applied to the node and propagated further to all its outgoing neighbours.*

With Theorem 2, to propagate a points-to set change, we only need to check the outgoing neighbours of the changed node and the points-to sets of their incoming neighbours, without traversing the whole PAG. Consider again the example in Figure 4. When o_1 is removed from $\mathrm{pts}(y)$, we only need to check z and w, which are the outgoing neighbours of y. For z, because it does not contain any other incoming neighbour, o_1 is hence removed from $\mathrm{pts}(z)$. However, for w, it has another incoming neighbour x (in addition to y) and $\mathrm{pts}(x)$ contains o_1, so $\mathrm{pts}(w)$ remains unchanged.

The two theorems above together guarantee that upon deleting a statement, it suffices to check the local neighbours of the change impacted nodes in the PAG to determine the points-to set changes and to perform change propagation. This significantly reduces the amount of computation on recomputing the points-to sets or traversing the whole graph.

To apply Theorems 1 and 2, we have made the assumption that the PAG is acyclic and we have considered only one edge deletion per time. The acyclic PAG can be satisfied by the SCC optimization, which is known in existing literature [21]. To support multiple edge deletions, we only need to slightly adapt the on-the-fly Andersen's algorithm. Specifically, we can change the algorithm such that within each iteration only a single edge deletion or addition is applied. This does not affect the performance of the original on-the-fly algorithm because the same amount of computation is required to reach the fixed point. Moreover, within each iteration, we can further prove the following property:

P3: Change consistency property: For an edge addition or deletion, if the change propagates to a node more than once (*i.e.*, from multiple incoming neighbours), then the effect of the change (*i.e*, the modification applied to the corresponding points-to set) must be the same.

Proof. Suppose two sets of object node changes Δ_1 and Δ_2 are propagated to the same node q along the two paths $path_1$: $p \rightarrow \ldots \rightarrow r_1 \rightarrow q$ and $path_2$: $p \rightarrow \ldots \rightarrow r_2 \rightarrow q$, respectively, where p is the root change node (the addition or deletion of an edge ending at p) and r_1 and r_2 are the two incoming neighbours of q. And suppose that there exits an object o such that $o \in \Delta_1$ and $o \notin \Delta_2$.

For deletion, $o \in \Delta_1$ means o has been deleted from $pts(p)$, vice versa. We can prove that there must exist a node w on $path_2$ such that o is reachable to w without going through p (otherwise, the deletion of o would have propagated to r_2, which contradicts with $o \notin \Delta_2$). Due to transitivity, we have $o \in pts(r_2)$. Because r_2 is an incoming neighbour of q, o will not be removed from $pts(q)$. In other words, any object $o \notin \Delta_1 \cap \Delta_2$ will be preserved in $pts(q)$. Therefore, the changes applied to $pts(q)$ are always the same.

For addition, $o \in \Delta_1$ means o has been added to $pts(p)$, and we can prove that o must be contained in $pts(q)$. The reason is that both Δ_1 and Δ_2 must be originated from the same root change Δ and o must be in Δ. If o is not in Δ_2, then there must exist a node w on $path_2$ such that $o \in pts(w)$, and again due to transitivity, $o \in pts(q)$. In other words, any object $o \notin \Delta_1 \cap \Delta_2$ should be already included in $pts(p)$. Therefore, the changes applied to $pts(q)$ are always the same.

Based on P3, in each iteration, we can parallelize the change propagation along different edges with no conflicts (if atomic updates are used). More specifically, we propagate the points-to set change of a node along all its outgoing edges in parallel without worrying about the order of propagation.

Algorithm 1: Parallel Incremental Pointer Analysis

```
Input  : Δ_P - a set of program changes.
         Deletions: D: -{d1,d2,...};
         Insertions: I: +{i1,i2,...}.
1 foreach s ∈ D do
2 |   e ← ExtractEdge(s)
3 |   DeleteEdge(e)
4 end
5 foreach s ∈ I do
6 |   e ← ExtractEdge(s)
7 |   AddEdge(e)
8 end
```

Algorithms 1-2 outline our parallel incremental algorithms for handling deletion. The input is a chunk of program changes containing two disjoint sets: D - a set of old statement deletions and I - a set of new statement insertions. For each statement, we first extract the corresponding edges in the PAG according to Andersen's analysis. For each identified edge, we then call Algorithm 2 if it is deleted and Algorithm *AddEdge* (omitted due to space reasons) if added, to compute the new points-to information and update the PAG.

We maintain a worklist, *WL*, initialized to the input edge. For both edge addition and deletion, in each iteration one edge from the worklist is processed, which involves two steps. First, we remove or add the edge from the PAG and handle the SCCs. We ensure that after deleting/adding the edge the PAG is acyclic and all SCCs are collapsed into a single node. For edge deletion, this step may break an existing SCC into multiple smaller SCCs or individual nodes. Second, we propagate the points-to set changes caused by the edge deletion or addition in parallel. This procedure takes two inputs: a set Δ of potential points-to set changes, and a node y that these changes are propagating to. For

Algorithm 2: DeleteEdge(e)

```
   Input  : e - a deleted edge
1  WL ← e // initialize worklist to e
2  while WL ≠ ∅ do
3  |  e ← RemoveOneEdgeFrom(WL)
4  |  DeleteEdgeAndDetectSCC(e)
   |  // let e be x → y
5  |  ParallelPropagateDeleteChange(pts(x), y)
6  end

7  ParallelPropagateDeleteChange(Δ, y):
   Input  : Δ - a set of points-to set changes
            y - a node that Δ propagates to
8  foreach z → y do
9  |  Δ = Δ \ (Δ ∩ pts(z))
10 |  if Δ = ∅ then
11 |  |  return
12 |  end
13 end
14 if !y.updated then
   |  // let updated be the flag of y
15 |  pts(y) ← (pts(y) \ Δ)
16 end
   // all outgoing edges in parallel
17 Parallel foreach y → w do
18 |  ParallelPropagateDeleteChange(Δ, w)
19 end
20 CheckNewEdges(Δ, y)

21 CheckNewEdges(Δ, y):
22 foreach o ∈ Δ do
   |  // process complex statements related to y.f
23 |  foreach node o.f generated from y.f do
   |  |  // add to WL all edges (e) from/to o.f
24 |  |  sync {WL} ← e
25 |  end
26 end
```

an edge $x \rightarrow y$, Δ is initialized to $pts(x)$. For deletion, we remove from Δ all the objects that overlap with the points-to sets of y's incoming neighbours. For the remaining objects in Δ, we then remove them from $pts(y)$ and propagate them further to all of y's outgoing neighbours. For addition, we simply check if the node's points-to set contains the change or not. If yes the change is skipped, otherwise the change is applied.

Because concurrent modifications to the same points-to set are always consistent, we only need to add a flag, *y.updated*, to indicate whether the change was successful, without the expensive synchronization among them. The only synchronization needed is on the worklist, because different parallel tasks may concurrently add different new edges to the worklist.

To handle dynamic edges that are deleted or added during the change propagation, we run the procedure *CheckNewEdges* once any change is applied to a node. This procedure takes a points-to set change Δ and a target node y as input, and returns a list of deleted or added PAG edges to the worklist. There are three types of statements that can introduce new edges: *load*, *store* and *call*, which we call *complex* statements. A complex statement can introduce multiple edges because its base variable may point to multiple objects. For example, if an object o is removed from $\text{pts}(y)$ and y is a base variable of a complex statement (e.g., x = y.f), we remove the edge $o.f \rightarrow x$. For a deleted method call, we simply remove the edges related to the method call, but keep the nodes corresponding to the method body (to improve performance if the method call is added back later).

3.2 Parallel Incremental Happens-Before Analysis

A key to our scalable happens-before analysis is a new representation of the SHB graph, which enables both compact graph storage and efficient graph updating. Instead of constructing per-thread sequential traces with repetitive nodes corresponding to the same statement, we construct a unique subgraph for each method/thread and connect the subgraphs with happens-before edges. The happens-before relation of nodes (e.g., in the multiply-visited methods) is then computed "on-the-fly" following the method-call edges and the inter-thread edges. When a change in a multiply-visited method happens, different node instances corresponding to the change can thus have different happens-before edges without sacrificing accuracy.

3.2.1 SHB Graph Construction

We maintain a map *exist* from the unique *id* of each method-/thread to its subgraph $subshb_{id}$. Each subgraph has two fields: *tids* which records the threads that have invoked/-forked the method/thread, and *trace* which stores the SHB nodes corresponding to the statements inside the method-/thread. Taking the main method (*target*), an empty subgraph ($subshb_{tar}$) and the executing thread id (*ctid*) as input, the algorithm returns the SHB graph (*shb*). Initially, we add the pair of $\langle tar, subshb_{tar} \rangle$ to the *exist* map and include *ctid* into the field *tids* of $subshb_{tar}$. Afterwards, we extract the statements in *target* and create SHB nodes according to Table 1 for each statement and insert it into $subshb_{id}.trace$.

Table 3. Edges in the new SHB graph.

Statements	Edges
❹ $x = o.m(\ldots)$	$\forall O_c \in pts(o) : call(O_c.m) \xrightarrow{t_{id}} subshb_{O_c.m}$
❺ $t.start()$	$\forall O_c \in pts(t) : start(O_c) \xrightarrow{t_{id}} subshb_{O_c}$
❻ $t.join()$	$\forall O_c \in pts(t) : subshb_{O_c} \xrightarrow{t_{id}} join(O_c)$

```
1 main() {              t1:              t2:
2  x = 0;               9   x = 1;       12  y = x;
3  y = 5;               10  m1();        13  m1();
4  t1 = new Thread();   11  m2();//add   14  m2();
5  t2 = new Thread();
6  t1.start();          15  void m1(){    18  void m2(){
7  t2.start();          16   x = 3;      19   x = 2;
8 }                     17   print(x);}  20   y = 0;//del}
```

Figure 7. An example for the SHB graph construction.

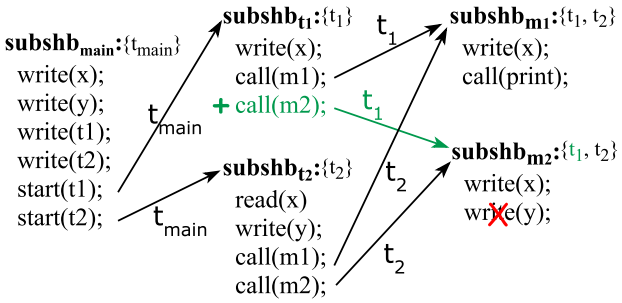


Figure 8. The SHB graph for the example in Figure 7.

The new happens-before edges are constructed according to Table 3. Each edge is labeled with the corresponding thread id. For method call ❹, we create a unique signature *sig* of each callee method $O_c.m$ and check the map *exist* if $subshb_{sig}$ has been created. If *sig* exists, it means $O_c.m$ has been visited before and its subgraph has been created, which avoids redundant statement traversal. We thus add the *ctid* into $subshb_{sig}.tids$ and add a new happens-before edge from the calling node to the existing subgraph with the label *ctid*. Otherwise, we create a new subgraph $subshb_{sig}$ for the newly discovered method. For thread start ❺, we create a new thread id (*tid*) for each object node in $pts(t)$, and follow the same procedure to construct $subshb_{tid}$ and add happens-before edges. For thread join ❻, we add an edge from the last node in $subshb_{tid}$ to the join node in $subshb_{tar}$, where *tid* is the thread id of the joined thread, corresponding to the object node in $pts(t)$. The procedure for creating different subgraphs can run in parallel, since different threads/methods are independent from each other.

Example We use an example in Figure 7 to illustrate our algorithm. Suppose the method call $m2()$ at lines 11 is not in the program initially. We first create $subshb_{main}$ and traverse the statements in main method. After inserting $write(x)$ and $write(y)$ into the *trace* field for the two writes at lines 2 and 3, we see the two thread start operations. We then create $subshb_{t_1}$ and $subshb_{t_2}$ for the two threads in parallel and add their corresponding happens-before edges. Consider the two method calls $m1()$ at lines 10 and 13, they introduce only one subgraph $subshb_{m1}$, which is created when $m1()$ is visited the first time. The final SHB graph is shown in Figure 8.

3.2.2 Incremental Graph Update

Thanks to our new SHB graph representation, incremental changes can be updated efficiently in parallel: 1) changes to statements in a method that is invoked multiple times need to be updated only once; and 2) multiple changes to different methods/threads can be updated in parallel (because they belong to different subgraphs).

For each added statement, we simply follow the same SHB graph construction procedure described in the previous subsection. For each deleted statement *s*, we first delete the node representing *s* from its belonging $subshb_{tar}$. In addition, for method call ❹, we locate the subgraph of the callee method and remove the corresponding SHB edges. For thread start ❺, we remove the corresponding SHB edges for each $subshb_{tid}$. Note that we do not remove the subgraph itself, such that the subgraph can be reused later if the method call or thread start is added back. For thread join ❻, we remove the SHB edge from $subshb_{tid}$ to $subshb_{tar}$.

Example Consider two changes in our example in Figure 7: (i) inserting a method call statement $m2()$ at line 11, and (ii) deleting the statement at line 20. For (i), we first create a method call node $call(m2)$ at the last position in $subshb_{t_1}$. Since $subshb_{m_2}$ already exists in the SHB graph, we skip traversing $m2()$. We add an edge $call(m2) \xrightarrow{t_1} subshb_{m_2}$ to the graph and add t_1 into $subshb_{m_2}.tids$. For (ii), we localize the $write(y)$ node corresponding to this statement and simply remove it from $subshb_{m_1}$.

3.2.3 Computing Happens-Before Relation

Our new SHB graph representation also makes computing the HB relation more efficient than existing approach [7]. For changes in a method invoked multiple times, instead of checking the path reachability between each individual pair of nodes, we can check for multiple node pairs altogether. For example, in Figure 8 although the method $m2()$ is invoked once by t_1 and once by t_2 which generates two write nodes, when computing the HB relation between the nodes in t_{main} and those from $m2()$, we can find that the nodes in t_{main} dominate all the nodes in $m2()$ in the SHB graph. Therefore, we can determine the happens-before relation for all these two write nodes by checking the path dominator once.

3.3 Distributed System Design

There are three main components in our design of distributing the analysis to a remote server, which is expected to have more computing power than the machine running the IDE. The first component is a change tracker that tracks the code changes in the IDE and sends them to the server with a compact data format. The second component is a real-time parallel analysis framework that implements our

incremental algorithms for pointer analysis and happens-before analysis. The third component is an incremental bug detector that leverages our framework to detect concurrency bugs and also sends the detection results to the IDE. We next focus on describing the second component, which is the core of our system.

Parallel Analysis Framework We implement a communication interface between the client and the server based on the open-source Akka framework [1], which supports efficient real-time computation on graphs via message passing and asynchronous communication. Akka is based on the actor model and distributes computations to actors in a hierarchical way. We hence can run the server on both a single multicore machine or multiple machines with a master-workers hierarchy. The master actor manages task generation and distribution, and the worker actor performs specific graph computations (e.g., adding/removing nodes/edges and updating the points-to sets). Tasks are assigned by the master and consumed by workers following a work stealing schedule until all tasks are processed.

Graph Storage Due to the distributed design, we can leverage distributed memory to store large graphs when the memory of a single computing node is limited. For the PAG, we partition the graph by following the edge cut strategy in Titan [2], in which nodes/edges created from the same method and those involved in the same points-to constraint are more likely to be stored together. For the SHB graph, we separate it into two parts: graph skeleton and subgraphs. The graph skeleton uses SHB edges to connect the *id*s of subgraphs and can be stored in a single memory region. The subgraphs can be stored in different memory regions and located efficiently by maintaining a map from each *id* to subgraph.

Message Format Akka provides protocol buffers and custom serializers to encode messages between client and server. We encode all graph nodes/edges and subgraph *id*s as integers or strings to facilitate message serialization. For example, deleting a statement "b=a" is encoded as "-id" where id is the unique id of the statement in the SSA form, and it is further encoded into "-(id1,id2)" on the server for graph computation, in which id1 and id2 represent integer identifiers of nodes a and b respectively, and id1 is the source and id2 the sink of the PAG edge.

3.4 Connection with Dynamic Graph Algorithms

D4 updates the two graphs (PAG and SHB) dynamically, which is related to dynamic algorithms on directed graphs. Existing dynamic graph algorithms have focused on shortest path [44], transitive closure [44, 45] and max/min flow [46]. For pointer analysis, our priority here is to efficiently update the points-to sets of a specific set of nodes in the PAG. For happens-before analysis, the problem is to effectively update the content of each node (*subshb*) as well as its affected nodes/edges based on the definition of the happens-before relation. Although existing algorithms cannot be directly applied to our cases, for certain tasks (e.g., SCC detection and checking reachability from a pointer node to an object node) we may utilize dynamic reachability algorithms [44, 45] to improve the performance.

4 D4: Concurrency Bug Detection

D4 can be used to develop many interesting incremental concurrency analyses, such as detecting data races, atomicity violations and deadlocks.

We have implemented both data race and deadlock detection in D4. Our race detection checks the happens-before relation and the lockset condition between every conflicting pair of *read* and *write* nodes on the same abstract heap from different threads. If the two nodes cannot reach each other in the SHB graph and there is no common lock protection, we will report them as a race. Our race detection algorithm is the same as that presented in [7], except that we use a different SHB graph representation to determine the happens-before relation.

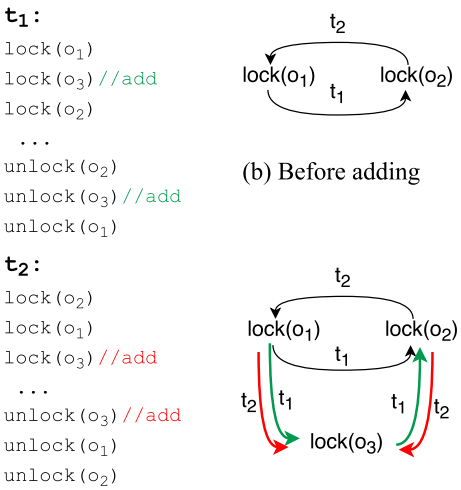
In this section, we focus on our novel incremental deadlock detection algorithm. Although exhaustive algorithms for deadlock detection exist, this is the first incremental deadlock detection algorithm, which is in fact highly non-trivial without D4. One has to develop new incremental data structures, update them correctly upon code changes, and integrate them efficiently with incremental race detection. Besides, the ability to detect deadlocks is particularly important for interactive race detection tools, because once a data race is detected, programmers often use locks to fix the race, which may introduce new deadlock bugs.

Next, we first introduce the *lock-dependency graph* which can be constructed from the SHB graph. Then, we present our incremental algorithm that uses the graph for deadlock detection.

Lock Dependency Graph The lock dependency (LD) graph contains nodes corresponding to lock operations, and edges corresponding to lock dependencies. For example, if a thread t is holding a lock l_1 and continues to acquire another lock l_2, an edge $lock(l_1) \xrightarrow{t} lock(l_2)$ is added to the LD graph.

The LD graph can be constructed from the SHB graph by traversing the lock/unlock nodes for each thread. For a lock statement on variable p, suppose $pts(p) = \{o_1,o_2\}$, it generates two lock nodes in the LD graph: $lock(o_1)$ and $lock(o_2)$. Figure 9 shows an example. The LD graph contains three nodes $lock(o_1)$, $lock(o_2)$ and $lock(o_3)$ connected by edges labeled with corresponding thread ids.

Incremental Deadlock Detection Our basic idea of incremental deadlock detection is to look for cycles in the LD graph with edge labels from multiple threads, which indicate circular dependencies of locks. We then check the happens-before relation between the involved nodes to find real deadlocks. For example, in Figure 9(b), $lock(o_1) \xrightarrow{t_1} lock(o_2)$ and



(a) Example code (b) Before adding (c) After adding

Figure 9. An example for the LD graph construction.

$lock(o_2) \xrightarrow{t_2} lock(o_1)$ form a circular dependency. To realize incremental deadlock detection, we develop an incremental algorithm for updating the LD graph and an incremental algorithm for deadlock checking.

Incremental LD Graph Update For an added synchronized statement in thread t, we first locate the method it belongs to and its corresponding subgraph $subshb_{tar}$, and create a pair of *lock*/*unlock* nodes and insert them into $subshb_{tar}$ according to the statement location. Starting from the changed *node*, we search the first *lock*/*unlock* node right before the added *lock* node (*pred*), and the consecutive *lock*/*unlock* node right after the added *lock* node (*succ*) along edges in the SHB graph. We call two *lock* nodes connected by an edge of LD graph a *lock pair*. If *pred* is a *lock* node, it means *pred* and *node* can form a lock pair with thread ids in $subshb_{tar}.tids$. Meanwhile, if *succ* is also a *lock* node, a lock pair between *node* and *succ* is added to the LD graph. Afterwards, we traverse the LD graph in the reverse order to discover the incoming *lock* nodes of *pred* with edges labeled t. For each such node *pred'*, we add a new lock pair between *pred'* and *node*. Then, we collect the outgoing *lock* nodes of *succ*, and create lock pairs for *node* and each of them. For a deleted synchronized statement, we simply remove its corresponding *lock*/*unlock* nodes from $subshb_{tar}$ as well as its lock pairs.

Consider Figure 9(a) in which $lock(o_3)/unlock(o_3)$ are added in both t_1 and t_2. We first localize the lock nodes before and after the added statement, and then add four edges: $lock(o_1) \xrightarrow{t1} lock(o_3)$, $lock(o_1) \xrightarrow{t2} lock(o_3)$, $lock(o_3) \xrightarrow{t1} lock(o_2)$ and $lock(o_2) \xrightarrow{t2} lock(o_3)$, as shown in Figure 9(c).

Incremental Deadlock Checking Algorithm 3 illustrates the incremental deadlock detection. The key idea is to check only the cycles containing the changed (added or deleted) *lock* nodes. We first collect all the circular dependencies that include the changed *lock* nodes. Then, we parallelize deadlock detection for all cycles by checking the happens-before relation between conflicting lock nodes from different threads in each cycle.

Algorithm 3: IncrementalDeadlockDetection

Global States : *shb* - updated SHB graph
ldg - updated LD graph
Input : Δ_{lock} - the changed lock nodes
Output : *deadlocks* - detected deadlocks

```
1  cycles ← DiscoverCircularDependency(ldg, Δ_lock)
2  foreach c ∈ cycles do
3  |  ParallelDeadlockDetection(c)
4  end

5  ParallelDeadlockDetection(c):
6  tids ← ExtractTidsInCycle(c)
7  foreach (t_i, t_j) ∈ tids do
   |  // for each pair of threads
8  |  lock(x), lock(y) ← FindConflictingLocks(t_i, t_j, c)
   |  // check happens-before condition
9  |  if (!CheckHBFor(lock(x)_{t_i}, lock(y)_{t_j}) &&
   |  !CheckHBFor(lock(x)_{t_j}, lock(y)_{t_i})) then
10 |  |  deadlocks ← c
11 |  end
12 end
```

5 Evaluation

We implemented D4 based on the WALA framework [3] and evaluated it on a collection of 14 real-world large Java applications from DaCapo-9.12, as shown in Table 4. We ran the D4 client on a MacBook Pro laptop with Intel i7 CPU and the server on a Mercury AH-GPU424 HPC server with Dual 12-core Intel©Xeon©CPU E5-2695 v2@2.40GHz (2 threads per core) processors. In this section, we report the results of our experiments.

Evaluation Methodology For each benchmark, we run three sets of experiments. (1) We first run the whole program exhaustive analysis on the local client machine to detect both data-races and deadlocks. Then, we initialize D4 with the graph data computed for the whole program in the first step and continue to conduct two experiments with incremental code changes. (2) For each statement in each method in the program, we delete the statement and run D4, which uses the parallel incremental algorithms for detecting concurrency bugs. (3) For the deleted statement in the previous step, we add it back and re-run D4.

We run D4 with two configurations: on the local client machine with a single thread (*D4-1*) to evaluate our incremental algorithms only, and on the server machine with

Table 4. Benchmarks and the PAG metrics.

App	#Class	#Method	#Pointer	#Object	#Edge
avrora	23K	238K	2M	33K	229M
batik	23K	60K	1.2M	31K	272M
eclipse	21K	36K	365K	7K	44M
fop	19K	68K	2M	42K	295M
h2	20K	69K	2M	32K	301M
jython	26K	79K	2M	53K	325M
luindex	20K	71K	1.8M	29K	299M
lusearch	20K	63K	1M	18K	185M
pmd	22K	42K	983K	25K	101M
sunflow	22K	73K	1.5M	32K	218M
tomcat	16K	36K	886K	23K	94M
tradebeans	14K	39K	674K	19K	99M
tradesoap	14K	38K	653K	20K	97M
xalan	21K	33K	576K	15K	138M

48 threads (*D4-48*) to evaluate our parallel incremental algorithms. We measure the time taken by each component in each step and compare the performance between the exhaustive analysis and D4. In addition, we repeat the same experiments for ECHO running on the client machine to compare the performance between D4 and ECHO.

Benchmarks The metrics of the benchmarks and their PAGs are reported in Table 4. Columns 2-6 report the numbers of classes, methods, pointer nodes, object nodes and edges in the PAGs, respectively. More than half of the benchmarks contain over 1M pointer nodes and over 200M edges in the PAG. The default pointer analysis is based on the ZeroOneContainerCFA in WALA, which creates an object node for every allocation site and has unlimited object-sensitivity for collection objects. For all benchmarks, certain JDK libraries such as *java.awt.** and *java.nio.** are excluded to ensure that the exhaustive analysis can finish within 6 hours. This exclusion makes a trade-off between soundness and computational cost, which is a common practice for both static and dynamic analysis tools to improve performance.

5.1 Performance of Incremental Pointer Analysis

Table 5 compares the performance between exhaustive pointer analysis and different sequential incremental algorithms. Overall, D4 achieves dramatic speedup over the other algorithms, especially for handling deletion. For most benchmarks, the exhaustive analysis takes several hours to compute (2.4h on average). For a deletion change, on average, the *reset-recompute* incremental algorithm in ECHO takes 26s, the *reachability*-based incremental algorithm in ECHO takes 39s, whereas *D4-1* takes only 73ms to analyze, which is at least 300X faster than the other incremental algorithms. The speedup is also significant for the worst case scenarios, where analyzing a certain deletion change takes the longest time among all changes in each benchmark. In the worst case, *reset-recompute* takes more than 17min, *reachability* takes more than 22min, while *D4-1* takes only 1.1min respectively for all benchmarks on average, which achieves 20X speedup.

Table 6. Performance of parallel incremental pointer analysis.

App	D4-48			
	insert		delete	
	avg.	worst	avg.	worst
avrora	0.82ms	0.1s	27ms	10.5s
batik	0.41ms	0.1s	42ms	6.1s
eclipse	0.62ms	0.07s	23ms	2.6s
fop	0.82ms	0.3s	29ms	5.9s
h2	0.16ms	1.1s	24ms	9.4s
jython	0.35ms	0.9s	18ms	22s
luindex	0.88ms	1.7s	31ms	7s
lusearch	0.42ms	0.2s	9ms	2.8s
pmd	0.53ms	0.1s	13ms	1.2s
sunflow	0.66ms	2.9s	36ms	8s
tomcat	0.19ms	0.05s	28ms	1.8s
tradebeans	0.37ms	0.1s	11ms	0.8s
tradesoap	0.43ms	0.2s	15ms	1s
xalan	0.5ms	0.13s	5ms	1.7s
Average	0.51ms	0.6s	24ms	5.5s

Performance of parallel incremental algorithms Table 6 reports the performance of our parallel incremental pointer analysis algorithms on the server running 48 threads. Compared to *D4-1*, it further improves performance by an order of magnitude. For a deletion change, *D4-48* takes only 24ms on average, more than three orders of magnitude faster than existing sequential incremental algorithms. In the worst case, *D4-48* takes only 5.5s on average per deletion change, achieving more than 200X speedup over existing algorithms.

For insertion changes, the average time for all the four incremental and parallel algorithms per change is within 0.1s, indicating that these algorithms are fast enough for practical use in the programming phase with respect to incremental code insertions (but not deletion). Nevertheless, for *reset-recompute* and *reachability* the worst case scenarios still take over 7s on average, which could be intrusive in the IDE. However, *D4-1* improves the performance to 4.1s, and *D4-48* further reduces the time to 0.6s, which is reasonably fast for practical use.

5.2 Performance of Concurrency Bug Detection

Table 7 reports the performance of concurrency bug detection for all the 13 multithreaded applications in DaCapo-9.12 (fop is excluded because it is single-threaded), including the time taken by exhaustive analysis, by ECHO (for race detection only), and by *D4-1* and *D4-48* (for both data race and deadlock detection). Note that the time for exhaustive analysis includes constructing both the PAG and the SHB graph for the whole code base and detecting both data races and deadlocks in the whole program. The time for ECHO and D4 includes that taken by incremental algorithms for

Table 5. Performance of the exhaustive and different sequential incremental pointer analysis algorithms.

App	Exhaustive	ECHO-Reset-Recompute				ECHO-Reachability				D4-1			
		insert		delete		insert		delete		insert		delete	
		avg.	worst	avg.	worst	avg.	worst	avg.	worst	avg.	worst	avg.	worst
avrora	4.8h	27ms	3s	52s	30+min	32ms	3s	76s	30+min	0.99ms	1s	89ms	228s
batik	4.1h	6ms	2s	48s	22min	6ms	2.2s	79s	30+min	0.86ms	0.8s	95ms	51s
eclipse	1h	5ms	1s	14s	12min	7ms	1.1s	20s	15min	0.74ms	0.4s	65ms	21s
fop	3.3h	12ms	7s	31s	16min	11ms	7.2s	38s	21min	1.33ms	5s	110ms	172s
h2	3.9h	11ms	21s	37s	25min	12ms	19s	82s	30+min	0.18ms	17s	78ms	120s
jython	3.2h	4.2ms	21s	43s	17min	4.5ms	20s	67s	30+min	0.49ms	12s	96ms	480s
luindex	2.9h	5.2ms	11s	22s	10min	5.3ms	12s	31s	12min	1.1ms	9s	143ms	162s
lusearch	2.5h	2.2ms	1.2s	17s	7min	2.4ms	1s	11s	8min	0.83ms	0.6s	15ms	44s
pmd	0.65h	1.8ms	0.7s	14s	30+min	1.8ms	0.7s	14s	30+min	0.61ms	0.2s	67ms	27s
sunflow	3.5h	2.8ms	15s	47s	11min	2.2ms	16s	61s	18min	0.87ms	7s	66ms	90s
tomcat	0.6h	9ms	8.7s	9.8s	30+min	8ms	9s	12s	30+min	0.32ms	0.3s	64ms	19s
tradebeans	0.75h	1ms	0.6s	3.5s	7min	0.9ms	0.6s	3s	9min	0.45ms	0.3s	24ms	14s
tradesoap	0.8h	1ms	0.7s	4s	10min	1ms	0.6s	5s	11min	0.62ms	0.3s	31ms	18s
xalan	0.47h	0.9ms	2s	38s	30+min	1.1ms	2.1s	14s	8min	0.9ms	0.4s	12ms	13s
Average	2.4h	6.8ms	7.2s	26s	17+min	7.2ms	7.5s	39s	22+min	0.72ms	4.1s	73ms	66s

Table 7. Performance of concurrency bug detection.

App	Exhaustive	Race Detection						Deadlock Detection			
		ECHO		D4-1		D4-48		D4-1		D4-48	
		avg.	worst	avg.	worst	avg.	worst	avg.	worst	avg.	worst
avrora	>6h	3min	1.8h	21s	15min	16ms	2min	231ms	2min	23ms	32s
batik	>6h	5.2min	2h	1.3s	13min	0.9s	57s	1ms	11ms	1ms	8ms
eclipse	1.2h	5s	10min	0.3s	5min	152ms	49s	110ms	2min	13ms	4s
h2	4h	1.2s	6min	33ms	39s	12ms	15s	1ms	18ms	1ms	10ms
jython	3.3h	1s	5min	19ms	20s	17ms	11s	0.4ms	242ms	1ms	53ms
luindex	3h	43ms	2min	4ms	7s	1.9ms	3.8ms	32ms	29s	25ms	17s
lusearch	2.6h	19ms	1.7min	7ms	5s	2.2ms	4.1ms	1ms	3ms	1ms	1.3s
pmd	0.8h	3.1s	7min	41ms	9s	6.8ms	1s	5.4ms	1s	0.2ms	53ms
sunflow	3.6h	1s	3min	0.15ms	23ms	0.1ms	12ms	0.3ms	8ms	0.1ms	2ms
tomcat	0.7h	1.7s	6min	6ms	4.3s	1.5ms	0.82s	0.1ms	0.9ms	0.1ms	0.4ms
tradebeans	0.8h	49ms	3min	1.1ms	1s	0.8ms	0.3s	0.1ms	1.3ms	0.1ms	0.4ms
tradesoap	0.9h	47ms	2.6min	0.9ms	1s	0.7ms	0.4s	0.1ms	1ms	0.1ms	0.3ms
xalan	0.5h	33ms	1.8min	0.2ms	42ms	0.1ms	15ms	1ms	2.7ms	0.1ms	1.1ms
Average	>2.6h	25s	21min	1.8s	2.9m	0.12s	20s	29ms	21s	5ms	4.2s

updating the graphs (*i.e.*, SHB and LD) and detecting bugs per change.

Overall, the exhaustive analysis requires a long time (>2.6h on average) to detect races and deadlocks in the whole program. The incremental detection algorithms are typically orders of magnitude faster than the exhaustive analysis, even in the worst case scenarios. Between D4 and ECHO, the incremental race detection algorithm implemented on top of D4 is much faster than ECHO, achieving 10X-2000X speedup for all cases on average, and 5X-50X speedup for the worst cases. ECHO takes 25s on average and 21min in the worst case to detect data races upon a change, while *D4-1* and *D4-48* take only 1.8s and 0.12s respectively on average, and 2.9min and 20s in the worst case. The incremental deadlock detection in D4 is also very efficient. It takes less than 29ms on average and 21s in the worst case for *D4-1*, and 5ms and 4.2s for *D4-48* per change. Compared to the exhaustive analysis, it is over 2000X faster.

Performance weakness of the new SHB analysis For small programs (e.g., <50 LOC), the new SHB analysis may require more time than the previous SHB analysis [7] to compute for incremental updates. There are two main reasons: (1) there are fewer repetitive method calls in small programs, hence the new SHB representation cannot be fully utilized; (2) the construction of the new SHB graph is more complex (e.g., maintenance of maps and subgraph fields), which leads to a trade-off between program size and performance.

5.3 Discussions

Network traffic time We also measured the network traffic time of the server mode in D4. In our lab environment with a standard wireless connection, the network traffic time is under 0.1ms per statement change, hence it is negligible.

Scalability We notice that the scalability of parallel incremental pointer analysis cannot catch up with that of parallel concurrency bug detection, due to two main reasons: (1) we

only process one edge in WL (lines 3-5 of Algorithm 2) per iteration in order to avoid conflict of edge updates; (2) the shape of the PAG determines the utilization of the parallel resources. For a deletion, if the chain of dependent variables of the change node is long and the in-degree of the variables on the chain is large, but the out-degree is small, our algorithm cannot scale well on this pattern, because most of the work has to be done sequentially, such as checking a large number of incoming neighbours and updating the affected edges.

Bug detection precision We note that although D4 focuses on improving scalability and efficiency through incremental analysis, it does not sacrifice precision compared to the exhaustive analysis. Being a static analysis (which is generally undecidable), D4 can report false positives, but it achieves the same precision as any whole-program static analyzers running the same bug detection algorithm.

We also studied the detection results reported by the whole program race detector in ECHO and D4, and confirmed that they report the same results. All the detection results are publicly available [5]. However, without significant knowledge in the application code it is difficult to verify the reported warnings (if they are true bugs or false alarms). On the other hand, the warnings reported by D4 are more manageable, because they are reported continuously driven by the current code changes, instead of providing the user with a long list of warnings by analyzing the whole program once.

D4 batch mode Although in our experiments D4 is evaluated for each single statement change (to avoid any biases caused by choosing a random set of changes), it is unnecessary to run D4 after every line of change, but D4 can be executed after a batch of changes. Currently, D4 runs whenever a file is saved by the user in the IDE, or the user can trigger D4 whenever an incremental check is necessary. The size of a batch varies in different applications but is typically small. For example, for good quality real-world projects such as h2 and eclipse, we observe that most of the commits contain only 1-50 lines of code changes. Besides, D4 can also run entirely on a single local machine to eliminate the cost of message passing over network.

Complex code changes As an IDE-based tool, we focus on source-level (i.e., Java bytecode) analysis. It is difficult for static analysis to handle link-time changes (such as dynamic libraries), because they only get into effect at integration time. We leave link-time changes for future research. Also, currently we do not handle package-level changes such as import. If a package is swapped out we simply re-build the PAG. We note that the analysis is only triggered after the program type checks. For changes that result in type errors, e.g., missing a class or method definition, they are handled by the type checker in the IDE. D4 is based on Andersen's algorithm, which does not deal with class-escape information. Hence, we analyze constraints from program changes without considering the modifiers.

Practicability Although we did not evaluate D4 in a production environment where even larger programs are running without an IDE, the fundamental and scalable techniques we provide can be utilized by other analysis tools since we aim at source code analysis. Besides, it is possible to make D4 independent of the IDE based on the Language Server Protocol [6], which we leave for future research.

6 Other Related Work

Do et al. [43] develop a Just-In-Time static analysis, Cheetah, which shares a similar goal as D4 to detect programming errors quickly. Differently, instead of incremental analysis, Cheetah uses a layered analysis which expands the analysis scope gradually from recent changes to code further away. In addition, Cheetah focuses on data-flow analyses such as taint analysis for Android apps instead of concurrency analysis.

RacerD [4] is a recent concurrency error detector developed by Facebook. Different from D4, RacerD relies on code annotation and performs race detection with aggressive ownership analysis, rather than using pointer analysis and happens-before analysis to analyze the impact of a code change.

There exist a few incremental and demand-driven pointer analysis algorithms based on CFL reachability [23, 35], logic programming [30], and data-flow analysis [11, 34]. However, none of these algorithms is change-aware. In particular, they cannot handle code deletion efficiently because pointer analysis is non-distributive.

A few parallel pointer analyses [24, 25, 27] have been proposed that leverage multicore CPUs or GPUs to improve performance. However, different from our parallel incremental algorithms, all these algorithms are designed for the exhaustive analysis. They require a pre-built call graph of the whole program and cannot handle dynamic code changes.

D4 is also related to regression testing approaches [38, 47] for concurrent programs. These approaches are effective for testing concurrent programs upon program changes because they can select test cases or interleavings by dynamically tracking change-relevant tests and shared memory accesses. Differently, they require dynamic tests and are slow.

There exist a wide range of techniques for detecting concurrency bugs in the whole program. For example, RacerX [16] and Chord [28] detected many real-world data races and deadlocks in C/C++ and Java programs with static analysis. HARD [42] utilizes hardware features to improve the race detection performance, Fonseca et al. [18] leverage linearizability testing to find concurrency bugs, and ConSeq [41] analyzes sequential memory errors to find concurrency bugs. Differently, all these techniques focus on whole program analyses which may be difficult to achieve efficiency.

7 Conclusion

We have presented a novel framework for detecting concurrency bugs efficiently in the programming phase. Powered by a distributed system design and new parallel incremental algorithms, D4 achieves dramatic performance improvements over the state-of-the-art. Our extensive evaluation on real-world large systems demonstrates excellent scalability and efficiency of D4, which is promising for practical use.

Acknowledgements

We thank our shepherd, Iulian Neamtiu, and the anonymous reviewers for their helpful feedback on earlier versions of this paper. We thank also Thomas Ball and Lawrence Rauchwerger for insightful discussions. This work was supported by NSF awards CCF-1552935 and CNS-1617985 and a Google Faculty Research Award to Jeff Huang.

References

[1] Akka Cluster Usage. http://http://doc.akka.io/docs/akka/current/java/cluster-usage.html.

[2] Titan Graph Partitioning. http://s3.thinkaurelius.com/docs/titan/0.5.0/graph-partitioning.html.

[3] T. J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[4] RacerD. http://fbinfer.com/docs/racerd.html.

[5] D4 website. https://github.com/parasol-aser/D4.

[6] The Language Server protocol. https://langserver.org/.

[7] Zhan, Sheng and Huang, Jeff ECHO: Instantaneous In Situ Race Detection in the IDE. In *Proceedings of the International Symposium on the Foundations of Software Engineering* (2016), pp. 775–786.

[8] Blackburn, S. M. and Garner, R. and Hoffman, C. and Khan, A. M. and McKinley, K. S. and Bentzur, R. and Diwan, A. and Feinberg, D. and Frampton, D. and Guyer, S. Z. and Hirzel, M. and Hosking, A. and Jump, M. and Lee, H. and Moss, J. E. B. and Phansalkar, A. and Stefanović, D. and VanDrunen, T. and von Dincklage, D. and Wiedermann, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (2006), pp. 169–190.

[9] Acar, U. A. *Self-adjusting Computation.* PhD thesis, 2005.

[10] Acar, U. A., Blelloch, G. E., Blume, M., and Tangwongsan, K. An experimental analysis of self-adjusting computation. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2006), pp. 96–107.

[11] Arzt, S., and Bodden, E. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM, pp. 288–298.

[12] Bhatotia, P., Fonseca, P., Acar, U. A., Brandenburg, B. B., and Rodrigues, R. ithreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), pp. 645–659.

[13] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, ACM, pp. 167–178.

[14] Chen, Y., Dunfield, J., and Acar, U. A. Type-directed automatic incrementalization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2012), pp. 299–310.

[15] Edvinsson, M., Lundberg, J., and Löwe, W. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, ACM, pp. 45–54.

[16] Engler, D., and Ashcraft, K. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM, pp. 237–252.

[17] Flanagan, C., and Freund, S. N. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM, pp. 121–133.

[18] Fonseca, P., Li, C., and Rodrigues, R. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, ACM, pp. 215–228.

[19] Grove, D., and Chambers, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6 (2001), 685–746.

[20] Sridharan, M., Chandra, S., Dolby, J., and Fink, S. J., and Yahav, E. Alias Analysis for Object-oriented Programs. *Aliasing in Object-Oriented Programming* (2013), pp. 196–232.

[21] Hardekopf, B., and Lin, C. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), pp. 290–299.

[22] Jin, G., Zhang, W., Deng, D., Liblit, B., and Lu, S. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, USENIX Association, pp. 221–236.

[23] Kastrinis, G., and Smaragdakis, Y. Efficient and effective handling of exceptions in java points-to analysis. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, Springer-Verlag, pp. 41–60.

[24] Mendez-Lojo, M., Burtscher, M., and Pingali, K. A gpu implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, ACM, pp. 107–116.

[25] Méndez-Lojo, M., Mathew, A., and Pingali, K. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, ACM, pp. 428–443.

[26] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, USENIX Association, pp. 267–280.

[27] Nagaraj, V., and Govindarajan, R. Parallel flow-sensitive pointer analysis by graph-rewriting. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, IEEE Press, pp. 19–28.

[28] Naik, M., Aiken, A., and Whaley, J. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, ACM, pp. 308–319.

[29] Putta, S., and Nasre, R. Parallel replication-based points-to analysis. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, Springer-Verlag, pp. 61–80.

[30] Saha, D., and Ramakrishnan, C. R. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, ACM, pp. 117–128.

[31] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson,

T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (Nov. 1997), 391–411.

[32] Sen, K. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, ACM, pp. 11–21.

[33] Serebryany, K., and Iskhodzhanov, T. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, ACM, pp. 62–71.

[34] Shang, L., Lu, Y., and Xue, J. Fast and precise points-to analysis with incremental cfl-reachability summarisation: Preliminary experience. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, ACM, pp. 270–273.

[35] Sridharan, M., Gopan, D., Shan, L., and Bodík, R. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, ACM, pp. 59–76.

[36] Su, Y., Ye, D., and Xue, J. Parallel pointer analysis with cfl-reachability. In *Proceedings of the 2014 Brazilian Conference on Intelligent Systems*, BRACIS '14, IEEE Computer Society, pp. 451–460.

[37] Szabó, T., Erdweg, S., and Voelter, M. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, ACM, pp. 320–331.

[38] Terragni, V., Cheung, S.-C., and Zhang, C. Recontest: Effective regression testing of concurrent programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (2015), pp. 246–256.

[39] Voung, J. W., Jhala, R., and Lerner, S. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, ACM, pp. 205–214.

[40] Yu, J., Narayanasamy, S., Pereira, C., and Pokam, G. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, ACM, pp. 485–502.

[41] Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., and Reps, T. Conseq: Detecting concurrency bugs through sequential errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, ACM, pp. 251–264.

[42] Zhou, P., Teodorescu, R., and Zhou, Y. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, IEEE Computer Society, pp. 121–132.

[43] Do, Lisa Nguyen Quang and Ali, Karim and Livshits, Benjamin and Bodden, Eric and Smith, Justin and Murphy-Hill, Emerson Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '17, ACM, pp. 307–317.

[44] Camil Demetrescu Fully Dynamic Algorithms for Path Problems on Directed Graphs. PhD thesis, 2001.

[45] Bender, Michael A. and Fineman, Jeremy T. and Gilbert, Seth and Tarjan, Robert E. A New Approach to Incremental Cycle Detection and Related Problems. In *ACM Trans. Algorithms* (2016), pp. 14:1–14:22.

[46] Italiano, Giuseppe F. and Nussbaum, Yahav and Sankowski, Piotr and Wulff-Nilsen, Christian Improved Algorithms for Min Cut and Max Flow in Undirected Planar Graphs. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing* (2011), pp. 313–322.

[47] Yu, T., Srisa-an, W., and Rothermel, G. SimRT: An automated framework to support regression testing for data races. In *Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 48–59.