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 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.

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
In the cloud. For example, in continuous integration of large
SHB graph. SHB graph, which simply tracks the lock/unlock nodes in the
be implemented by extending
mented based on the SHB graph, and deadlock detection can
race checking algorithm in ECHO can be directly imple-
analysis and happens-before analysis. For example, the same
gram analyses and concurrency analyses rely on pointer
bugs incrementally, since virtually all interesting static pro-
are propagated to the bug detection algorithms.

is propagated incrementally and its change
further. Taking
and
as input, the SHB graph is then
updated incrementally and its change
2
with
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 pro-
gram analyses and concurrency analyses rely on pointer
analysis and happens-before analysis. For example, the same
race checking algorithm in ECHO can be directly imple-
mented 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 dead-
lock detection in D4 and evaluated its performance exten-
sively on a collection of real-world large applications from
DaCapo. The experiments show dramatic efficiency and scal-
ability 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 simultane-
ously 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 al-
gorithms 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 ex-
ploting both the change-centric nature of program-
ing and the algorithmic parallelization of fundamen-
tal static analyses.
- We present an extensive evaluation of D4 on real-
world large applications, demonstrating signif-
ificant 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 @ 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.
Andersen’s algorithm [19, 20], which constructs a pointer as- 
vARIABLE ASSIGNMENT CORRESPONDS TO ONE OR MORE EDGES. There
pointer or reference variables, and

Many pointer analysis algorithms are based on the on-the-

2.2 Existing Algorithms
Previous work [7] has proposed sequential incremental pointer
analysis and happens-before algorithms for data race detec-
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) ∪ pts(a). When c=b is inserted, similarly, pts(c) is updated to pts(c) ∪ 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, 54]. 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 → 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 pts(y) as well as pts(z) and 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.
points-to constraints $pts(y) = pts(y) \cup \{o_2\}$, $pts(z) = pts(z) \cup pts(y)$, $pts(w) = pts(w) \cup pts(y)$, and $pts(w) = pts(w) \cup pts(x)$, 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 $pts(y)$, $pts(z)$ and $pts(w)$ until reaching a fixed point. The final values of the points-to sets are: $pts(x) = \{o_1\}$, $pts(y) = \{o_2\}$, $pts(z) = \{o_2\}$ and $pts(w) = \{o_1, o_2\}$.

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, $pts(w)$ 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 sets. 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 $pts(x)$ should be removed from $pts(y)$, by checking if the corresponding object node can reach $y$ in the PAG. In this case, $pts(x)$ contains only $o_1$ which cannot reach $y$, hence $o_1$ is removed from $pts(y)$. Because $pts(y)$ 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 $pts(z)$ but $pts(w)$ 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.

### 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 ❶ (method call), ❷ (thread start) and ❹ (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

<p>| Table 1. Nodes in the SHB graph. |</p>
<table>
<thead>
<tr>
<th>Statements</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>❶ $x = y.f$</td>
<td>write($x$), $O_c \in pts(y) : read(O_c, f)$</td>
</tr>
<tr>
<td>❷ $x.f = y$</td>
<td>read($y$), $O_c \in pts(x) : write(O_c, f)$</td>
</tr>
<tr>
<td>❸ synchronized(...)</td>
<td>$O_c \in pts(x) : lock($O_c$), unlock($O_c$)</td>
</tr>
<tr>
<td>❹ $o.m(...)$</td>
<td>$O_c \in pts(o) : call($O_c.m$)</td>
</tr>
<tr>
<td>❼ $t.start$</td>
<td>$O_c \in pts(t) : start($O_c$)</td>
</tr>
<tr>
<td>⓪ $t.join$</td>
<td>$O_c \in pts(t) : join($O_c$)</td>
</tr>
</tbody>
</table>

* ❶ and ⓪ also represent array read ($x = y[i]$) and write ($x[i] = y$), resp.

<p>| Table 2. Edges in the SHB graph. |</p>
<table>
<thead>
<tr>
<th>Statements</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>❹ $x = o.m(...)$</td>
<td>$O_c \in pts(o) : call($O_c.m$) $LastNode($O_c.m$) \rightarrow NextNode(call)$</td>
</tr>
<tr>
<td>❼ $t.start$</td>
<td>$O_c \in pts(t) : start($O_c$) \rightarrow FirstNode($O_c$)</td>
</tr>
<tr>
<td>⓪ $t.join$</td>
<td>$O_c \in pts(t) : LastNode($O_c$) \rightarrow join($O_c$)</td>
</tr>
</tbody>
</table>

NextNode(call): the consecutive node of the method call statement.
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, \( o \in pts(p) \) if and only if \( o \) can reach \( p \). For two pointer nodes \( p \) and \( q \), if \( p \) can reach \( q \) in the PAG, then \( o \in pts(p) \) 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 \( r \rightarrow q \)) 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 \( o \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow r \rightarrow q \) because the PAG is assumed to be acyclic.

![Figure 5](image1.png)

**Figure 5.** Illustration of the incoming neighbours property.

Suppose an edge \( p \rightarrow q \) 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 \( o \) 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 \( p \rightarrow q \). Because \( r \rightarrow q \), \( o \) can hence reach \( q \) without the edge \( p \rightarrow q \). Therefore, \( o \) remains in \( pts(q) \) after deleting \( p \rightarrow q \). 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 \( x \rightarrow y \), 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 \( q \rightarrow w \)) 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 \( r \rightarrow w \), and the path must be \( o \rightarrow \ldots \rightarrow r \rightarrow w \). The path may or may not contain \( q \). However, if it contains \( q \), then it must be \( o \rightarrow \ldots \rightarrow q \rightarrow \ldots \rightarrow r \rightarrow w \), which means that \( q \) can reach \( r \).

![Figure 6](image2.png)

**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 \( \text{pts}(y) \), 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 \( \text{pts}(z) \). However, for \( w \), it has another incoming neighbour \( x \) (in addition to \( y \)) and \( \text{pts}(x) \) contains \( o_1 \), so \( \text{pts}(w) \) 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:

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

\[ \text{Proof: Suppose two sets of object node changes } \Delta_1 \text{ and } \Delta_2 \text{ are propagated to the same node } q \text{ along the two paths } \text{path}_1: p \rightarrow \ldots \rightarrow r_1 \rightarrow q \text{ and } \text{path}_2: p \rightarrow \ldots \rightarrow r_2 \rightarrow q, \text{ respectively, where } p \text{ is the root change node (the addition or deletion of an edge ending at } p) \text{ and } r_1 \text{ and } r_2 \text{ are the two incoming neighbours of } q. \text{ And suppose that there exits an object } o \text{ such that } o \in \Delta_1 \text{ and } o \notin \Delta_2. \]

For deletion, \( o \in \Delta_1 \) means \( o \) has been deleted from \( \text{pts}(p) \), vice versa. We can prove that there must exist a node \( w \) on \( \text{path}_2 \) 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 \( o \in \text{pts}(r_2) \). Because \( r_2 \) is an incoming neighbour of \( q \), \( o \) will not be removed from \( \text{pts}(q) \). In other words, any object \( o \notin \Delta_1 \cap \Delta_2 \) will be preserved in \( \text{pts}(q) \). Therefore, the changes applied to \( \text{pts}(q) \) are always the same.

For addition, \( o \in \Delta_1 \) means \( o \) has been added to \( \text{pts}(p) \), and we can prove that \( o \) must be contained in \( \text{pts}(q) \). The reason is that both \( \Delta_1 \) and \( \Delta_2 \) must be originated from the same root change \( \Delta \) and \( o \) must be in \( \Delta \). If \( o \) is not in \( \Delta_2 \), then there must exist a node \( w \) on \( \text{path}_2 \) such that \( o \in \text{pts}(w) \), and again due to transitivity, \( o \in \text{pts}(q) \). In other words, any object \( o \notin \Delta_1 \cap \Delta_2 \) should be already included in \( \text{pts}(p) \). Therefore, the changes applied to \( \text{pts}(q) \) 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.

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 \( \Delta \) of potential points-to set changes, and a node \( y \) that these changes are propagating to. For

---

**Algorithm 1: Parallel Incremental Pointer Analysis**

**Input**
- \( \Delta_p \) - a set of program changes.
- Deletions: \( D = \{d_1,d_2,\ldots\} \);
- Insertions: \( I = \{i_1,i_2,\ldots\} \).

```plaintext
1 foreach \( s \in D \) do
2 \( e \leftarrow \text{ExtractEdge}(s) \)
3 DeleteEdge(e)
4 end
5 foreach \( s \in I \) do
6 \( e \leftarrow \text{ExtractEdge}(s) \)
7 AddEdge(e)
8 end
```

---
### Algorithm 2: DeleteEdge(e)

**Input**: e - a deleted edge

1. \(WL \leftarrow e\) // initialize worklist to e
2. **while** \(WL \neq \emptyset\) **do**
   3. \(e \leftarrow \text{RemoveOneEdgeFrom}(WL)\)
   4. DeleteEdgeAndDetectSCC(e)
   5. // let e be \(x \rightarrow y\)
   6. \(\text{ParallelPropagateDeleteChange}(\text{pts}(x), y)\)
3. **end**

7. \(\text{ParallelPropagateDeleteChange}(\Delta, y)\):
   **Input**: \(\Delta\) - a set of points-to set changes
   
   8. **foreach** \(z \rightarrow y\) **do**
      9. \(\Delta = \Delta \setminus (\Delta \cap \text{pts}(z))\)
      10. **if** \(\Delta = \emptyset\) **then**
         11. **return**
      **end**
   **end**
   // all outgoing edges in parallel

13. **if** \(y\_\text{updated}\) **then**
   14. // let \(updated\) be the flag of \(y\)
      15. \(\text{pts}(y) \leftarrow (\text{pts}(y) \setminus \Delta)\)
   **end**
   // all outgoing edges in parallel

17. **Parallel foreach** \(y \rightarrow w\) **do**
   18. \(\text{ParallelPropagateDeleteChange}(\Delta, w)\)
3. **end**
4. \(\text{CheckNewEdges}(\Delta, y)\)
5. \(\text{CheckNewEdges}(\Delta, y)\):
   6. **foreach** \(o \in \Delta\) **do**
      7. // process complex statements related to \(y.f\)
         8. **foreach** node \(o.f\) generated from \(y.f\) **do**
            9. // add to \(WL\) all edges \((e)\) from/to \(o.f\)
            10. **sync** \(\{WL\} \leftarrow e\)
       **end**
7. **end**

---

An edge \(x \rightarrow y\), \(\Delta\) is initialized to \(\text{pts}(x)\). For deletion, we remove from \(\Delta\) all the objects that overlap with the points-to sets of \(y\)'s incoming neighbours. For the remaining objects in \(\Delta\), we then remove them from \(\text{pts}(y)\) 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, \(y\_\text{updated}\), 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 \(\Delta\) and a target node \(y\) as input, and returns a list of deleted or added FAG 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 \(\text{exist}\) from the unique \(id\) of each method/thread to its subgraph \(\text{subshb}_{id}\). Each subgraph has two fields: \(\text{tids}\) which records the threads that have invoked/forked the method/thread, and \(\text{trace}\) which stores the SHB nodes corresponding to the statements inside the method/thread. Taking the main method \((\text{target})\), an empty subgraph \((\text{subshb}_{\text{target}})\) and the executing thread id \((\text{ctid})\) as input, the algorithm returns the SHB graph \((\text{shb})\). Initially, we add the pair of \((\text{tar}, \text{subshb}_{\text{tar}})\) to the \(\text{exist}\) map and include \(\text{ctid}\) into the field \(\text{tids}\) of \(\text{subshb}_{\text{tar}}\). Afterwards, we extract the statements in \(\text{target}\) and create SHB nodes according to Table 1 for each statement and insert it into \(\text{subshb}_{id}\_\text{trace}\).

### Table 3. Edges in the new SHB graph.

<table>
<thead>
<tr>
<th>Statements</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = o.m(...))</td>
<td>(V_{O_c} \in \text{pts}(o) : \text{call}(O_c.m) \xrightarrow{\text{tid}} \text{subshb}_{O_c,m})</td>
</tr>
<tr>
<td>(t_\text{start}())</td>
<td>(V_{O_c} \in \text{pts}(t) : \text{start}(O_c) \xrightarrow{\text{tid}} \text{subshb}_{O_c})</td>
</tr>
<tr>
<td>(t_\text{join}())</td>
<td>(V_{O_c} \in \text{pts}(t) : \text{subshb}_{O_c} \xrightarrow{\text{tid}} \text{join}(O_c))</td>
</tr>
</tbody>
</table>
The new happens-before edges are constructed according to Table 3. Each edge is labeled with the corresponding thread id. For method call θ, we create a unique signature sig of each calleee method $O_{.m}$ and check the map exist if $subshb_{sig}$ has been created. If sig exists, it means $O_{.m}$ 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 θ, 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 θ, 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 $m_2()$ 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 $m_1()$ at lines 10 and 13, they introduce only one subgraph $subshb_{m_1}$, which is created when $m_1()$ is visited the first time. The final SHB graph is shown in Figure 8.

**Figure 7.** An example for the SHB graph construction.

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

### 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 θ, we locate the subgraph of the calleee method and remove the corresponding SHB edges. For thread start θ, 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 θ, 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 $m_2()$ at line 11, and (ii) deleting the statement at line 20. For (i), we first create a method call node $call(m_2)$ at the last position in $subshb_{t_1}$. Since $subshb_{m_2}$ already exists in the SHB graph, we skip traversing $m_2()$. We add an edge $call(m_2)\rightarrow subshb_{m_2}$ 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 $m_2()$ 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 $m_2()$, we can find that the nodes in $t_{main}$ dominate all the nodes in $m_2()$ 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 ids 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 ids 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 l1 and continues to acquire another lock l2, 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
锁($o_2 \rightarrow t_2 \rightarrow o_1$)形成一个循环依赖。为了实现增量死锁检测，我们开发了一个增量算法来更新LD图和一个增量算法来检查死锁。

增量LD图更新  对于一个同步化语句在第$t$个线程中同步化，我们首先找到该方法以及其对应的子图$subshb_{tar}$，然后插入一个lock/unlock节点对，并将它们插入$subshb_{tar}$，根据语句位置。从变化的节点开始，我们搜索第一个lock/unlock节点对（pred），以及连续的lock/unlock节点对（succ）沿着SHB图中的边。我们称两个lock节点为一个锁对。如果pred是一个lock节点，那么pred和node可以形成一个锁对与thread ids在$subshb_{tar}$.tids。同时，如果succ也是一个lock节点，它与node和succ添加到LD图中。然后，我们使用LD图的逆序来发现与pred节点相关的锁节点。对于每个这样的节点pred’，我们添加一个新锁对 pred’和node。然后，我们收集所有的locked(unlock)节点，并为node和彼此创建一个新的lock 对。对于删除的同步化语句，我们简单地将其从对应的子图$subshb_{tar}$中删除。

增量死锁检测 Algorithm 3 说明了增量死锁检测。该算法的关键想法是检查每个语句行或被删除的语句行对应的锁节点的循环依赖。我们首先收集所有包含变化的锁节点的循环依赖。然后，我们并行地检查所有循环，以发现发生了死锁。最后，我们更新锁和解锁关系之间的循环依赖。

Algorithm 3: IncrementalDeadlockDetection

Global States: $shb$ - updated SHB graph
$ldg$ - updated LD graph

Input: $\Delta_{lock}$ - the changed lock nodes
Output: $deadlocks$ - detected deadlocks

1. cycles ← DiscoverCircularDependency($ldg, \Delta_{lock}$)
2. foreach $c \in$ cycles do
   3. ParallelDeadlockDetection($c$)
3. end

4. ParallelDeadlockDetection($c$):
5. $tids$ ← ExtractTidsInCycle($c$)
6. foreach $(t_i, t_j) \in tids$ do
   7. lock($x$), lock($y$) ← FindConflictingLocks($t_i, t_j, c$)
   8. // check happens-before condition
   9. if (!CheckHBFFor(lock($x$)$_{t_i}$, lock($y$)$_{t_j}$) \&\&
      !CheckHBFFor(lock($x$)$_{t_j}$, lock($y$)$_{t_i}$)) then
   10. deadlocks ← $c$
   11. end
12. end

lock($o_2 \rightarrow t_2 \rightarrow o_1$)形成一个循环依赖。为了实现增量死锁检测，我们开发了一个增量算法来更新LD图和一个增量算法来检查死锁。

增量LD图更新  对于一个同步化语句在第$t$个线程中同步化，我们首先找到该方法以及其对应的子图$subshb_{tar}$，然后插入一个lock/unlock节点对，并将它们插入$subshb_{tar}$，根据语句位置。从变化的节点开始，我们搜索第一个lock/unlock节点对（pred），以及连续的lock/unlock节点对（succ）沿着SHB图中的边。我们称两个lock节点为一个锁对。如果pred是一个lock节点，那么pred和node可以形成一个锁对与thread ids在$subshb_{tar}$.tids。同时，如果succ也是一个lock节点，它与node和succ添加到LD图中。然后，我们使用LD图的逆序来发现与pred节点相关的锁节点。对于每个这样的节点pred’，我们添加一个新锁对 pred’和node。然后，我们收集所有的locked(unlock)节点，并为node和彼此创建一个新的lock 对。对于删除的同步化语句，我们简单地将其从对应的子图$subshb_{tar}$中删除。

增量死锁检测 Algorithm 3 说明了增量死锁检测。该算法的关键想法是检查每个语句行或被删除的语句行对应的锁节点的循环依赖。我们首先收集所有包含变化的锁节点的循环依赖。然后，我们并行地检查所有循环，以发现发生了死锁。最后，我们更新锁和解锁关系之间的循环依赖。

Algorithm 3: IncrementalDeadlockDetection

Global States: $shb$ - updated SHB graph
$ldg$ - updated LD graph

Input: $\Delta_{lock}$ - the changed lock nodes
Output: $deadlocks$ - detected deadlocks

1. cycles ← DiscoverCircularDependency($ldg, \Delta_{lock}$)
2. foreach $c \in$ cycles do
   3. ParallelDeadlockDetection($c$)
3. end

4. ParallelDeadlockDetection($c$):
5. $tids$ ← ExtractTidsInCycle($c$)
6. foreach $(t_i, t_j) \in tids$ do
   7. lock($x$), lock($y$) ← FindConflictingLocks($t_i, t_j, c$)
   8. // check happens-before condition
   9. if (!CheckHBFFor(lock($x$)$_{t_i}$, lock($y$)$_{t_j}$) \&\&
      !CheckHBFFor(lock($x$)$_{t_j}$, lock($y$)$_{t_i}$)) then
   10. deadlocks ← $c$
   11. end
12. end

为了实现增量死锁检测，我们开发了一个增量算法来更新LD图和一个增量算法来检查死锁。

增量LD图更新  对于一个同步化语句在第$t$个线程中同步化，我们首先找到该方法以及其对应的子图$subshb_{tar}$，然后插入一个lock/unlock节点对，并将它们插入$subshb_{tar}$，根据语句位置。从变化的节点开始，我们搜索第一个lock/unlock节点对（pred），以及连续的lock/unlock节点对（succ）沿着SHB图中的边。我们称两个lock节点为一个锁对。如果pred是一个lock节点，那么pred和node可以形成一个锁对与thread ids在$subshb_{tar}$.tids。同时，如果succ也是一个lock节点，它与node和succ添加到LD图中。然后，我们使用LD图的逆序来发现与pred节点相关的锁节点。对于每个这样的节点pred’，我们添加一个新锁对 pred’和node。然后，我们收集所有的locked(unlock)节点，并为node和彼此创建一个新的lock 对。对于删除的同步化语句，我们简单地将其从对应的子图$subshb_{tar}$中删除。

增量死锁检测 Algorithm 3 说明了增量死锁检测。该算法的关键想法是检查每个语句行或被删除的语句行对应的锁节点的循环依赖。我们首先收集所有包含变化的锁节点的循环依赖。然后，我们并行地检查所有循环，以发现发生了死锁。最后，我们更新锁和解锁关系之间的循环依赖。

Algorithm 3: IncrementalDeadlockDetection

Global States: $shb$ - updated SHB graph
$ldg$ - updated LD graph

Input: $\Delta_{lock}$ - the changed lock nodes
Output: $deadlocks$ - detected deadlocks

1. cycles ← DiscoverCircularDependency($ldg, \Delta_{lock}$)
2. foreach $c \in$ cycles do
   3. ParallelDeadlockDetection($c$)
3. end

4. ParallelDeadlockDetection($c$):
5. $tids$ ← ExtractTidsInCycle($c$)
6. foreach $(t_i, t_j) \in tids$ do
   7. lock($x$), lock($y$) ← FindConflictingLocks($t_i, t_j, c$)
   8. // check happens-before condition
   9. if (!CheckHBFFor(lock($x$)$_{t_i}$, lock($y$)$_{t_j}$) \&\&
      !CheckHBFFor(lock($x$)$_{t_j}$, lock($y$)$_{t_i}$)) then
   10. deadlocks ← $c$
   11. end
12. end


评价方法 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.


评价方法 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.
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.

**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.

<table>
<thead>
<tr>
<th>App</th>
<th>Exhaustive</th>
<th>ECHO-Reset-Recompute</th>
<th>ECHO-Reachability</th>
<th>D4-1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>delete</td>
<td>insert</td>
<td>delete</td>
</tr>
<tr>
<td></td>
<td>avg. worst</td>
<td>avg. worst</td>
<td>avg. worst</td>
<td>avg. worst</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avrora</td>
<td>4.8h</td>
<td>27ms</td>
<td>3s</td>
<td>52s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30+min</td>
<td>76s</td>
<td>30+min</td>
</tr>
<tr>
<td>batik</td>
<td>4.1h</td>
<td>6ms</td>
<td>2s</td>
<td>48s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22min</td>
<td>79s</td>
<td>30+min</td>
</tr>
<tr>
<td>eclipse</td>
<td>1h</td>
<td>5ms</td>
<td>1s</td>
<td>14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12min</td>
<td>20s</td>
<td>15min</td>
</tr>
<tr>
<td>fop</td>
<td>3.3h</td>
<td>12ms</td>
<td>7s</td>
<td>31s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16min</td>
<td>72s</td>
<td>38s</td>
</tr>
<tr>
<td>h2</td>
<td>3.9h</td>
<td>11ms</td>
<td>21s</td>
<td>37s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25min</td>
<td>12s</td>
<td>82s</td>
</tr>
<tr>
<td>jython</td>
<td>3.2h</td>
<td>4.2ms</td>
<td>21s</td>
<td>43s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>75min</td>
<td>20s</td>
<td>67s</td>
</tr>
<tr>
<td>luindex</td>
<td>2.9h</td>
<td>5.2ms</td>
<td>11s</td>
<td>22s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10min</td>
<td>5.3ms</td>
<td>12s</td>
</tr>
<tr>
<td>lusearch</td>
<td>2.5h</td>
<td>2.2ms</td>
<td>1.2s</td>
<td>17s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7min</td>
<td>2.4ms</td>
<td>1s</td>
</tr>
<tr>
<td>pmid</td>
<td>0.65h</td>
<td>1.8ms</td>
<td>0.7s</td>
<td>14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30+min</td>
<td>1.8ms</td>
<td>7s</td>
</tr>
<tr>
<td>sunflow</td>
<td>3.5h</td>
<td>2.8ms</td>
<td>15s</td>
<td>47s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11min</td>
<td>2.2ms</td>
<td>16s</td>
</tr>
<tr>
<td>tomcat</td>
<td>0.6h</td>
<td>9ms</td>
<td>8.7s</td>
<td>9.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30+min</td>
<td>8ms</td>
<td>9s</td>
</tr>
<tr>
<td>tradebeans</td>
<td>0.75h</td>
<td>1ms</td>
<td>0.6s</td>
<td>3.5s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7min</td>
<td>0.9ms</td>
<td>6.6s</td>
</tr>
<tr>
<td>tradesoap</td>
<td>0.8h</td>
<td>1ms</td>
<td>0.7s</td>
<td>4s</td>
</tr>
<tr>
<td>xalan</td>
<td>0.47h</td>
<td>0.9ms</td>
<td>2s</td>
<td>38s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30+min</td>
<td>1.1ms</td>
<td>14s</td>
</tr>
<tr>
<td>Average</td>
<td>2.4h</td>
<td>6.8ms</td>
<td>7.2s</td>
<td>26s</td>
</tr>
</tbody>
</table>

Table 7. Performance of concurrency bug detection.

<table>
<thead>
<tr>
<th>App</th>
<th>Exhaustive</th>
<th>Race Detection</th>
<th>Deadlock Detection</th>
<th>Performance of concurrency bug detection.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>insert</td>
<td>D4-1</td>
<td>D4-48</td>
<td>D4-1</td>
</tr>
<tr>
<td></td>
<td>avg. worst</td>
<td>avg. worst</td>
<td>avg. worst</td>
<td>avg. worst</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avrora</td>
<td>&gt;6h</td>
<td>3min</td>
<td>21s</td>
<td>15min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.8h</td>
<td>21s</td>
<td>16min</td>
</tr>
<tr>
<td>batik</td>
<td>&gt;6h</td>
<td>5.2min</td>
<td>2h</td>
<td>13min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.3s</td>
<td>2h</td>
<td>0.9s</td>
</tr>
<tr>
<td>eclipse</td>
<td>1.2h</td>
<td>5s</td>
<td>10min</td>
<td>0.3s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5min</td>
<td>19min</td>
<td>0.55min</td>
</tr>
<tr>
<td>h2</td>
<td>4h</td>
<td>1.2s</td>
<td>6min</td>
<td>33ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12min</td>
<td>39s</td>
<td>12ms</td>
</tr>
<tr>
<td>jython</td>
<td>3.3h</td>
<td>1s</td>
<td>5min</td>
<td>19min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20s</td>
<td>17ms</td>
<td>11s</td>
</tr>
<tr>
<td>luindex</td>
<td>3h</td>
<td>43ms</td>
<td>2min</td>
<td>4min</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7s</td>
<td>1.9ms</td>
<td>3.8ms</td>
</tr>
<tr>
<td>lusearch</td>
<td>2.6h</td>
<td>19ms</td>
<td>1.7min</td>
<td>7ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5s</td>
<td>2.2ms</td>
<td>4.1ms</td>
</tr>
<tr>
<td>pmid</td>
<td>0.8h</td>
<td>1.3s</td>
<td>7min</td>
<td>41ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9s</td>
<td>6.8ms</td>
<td>1s</td>
</tr>
<tr>
<td>sunflow</td>
<td>3.6h</td>
<td>1s</td>
<td>3min</td>
<td>0.15ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23ms</td>
<td>0.1ms</td>
<td>12ms</td>
</tr>
<tr>
<td>tomcat</td>
<td>0.7h</td>
<td>1.7s</td>
<td>6min</td>
<td>6ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3s</td>
<td>1.5ms</td>
<td>0.82s</td>
</tr>
<tr>
<td>tradebeans</td>
<td>0.8h</td>
<td>49ms</td>
<td>3min</td>
<td>1.1ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1s</td>
<td>0.8ms</td>
<td>0.3s</td>
</tr>
<tr>
<td>tradesoap</td>
<td>0.9h</td>
<td>47ms</td>
<td>2.6min</td>
<td>0.9ms</td>
</tr>
<tr>
<td>xalan</td>
<td>0.5h</td>
<td>33ms</td>
<td>1.8min</td>
<td>0.2ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42ms</td>
<td>0.1ms</td>
<td>15ms</td>
</tr>
<tr>
<td>Average</td>
<td>&gt;2.6h</td>
<td>25s</td>
<td>21min</td>
<td>1.8s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.9ms</td>
<td>0.12s</td>
<td>20s</td>
</tr>
</tbody>
</table>

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

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.
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

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


