Actor-based Parallel Dataflow Analysis

Jonathan Rodriguez and Ondřej Lhoták

University of Waterloo,
Waterloo, Ontario, Canada
{j2rodrig,olhotak}@uwaterloo.ca

Abstract. Defining algorithms in a way which allows parallel execution is becoming increasingly important as multicore computers become ubiquitous. We present IFDS-A, a parallel algorithm for solving context-sensitive interprocedural finite distributive subset (IFDS) dataflow problems. IFDS-A defines these problems in terms of Actors, and dataflow dependencies as messages passed between these Actors. We implement the algorithm in Scala, and evaluate its performance against a comparable sequential algorithm. With eight cores, IFDS-A is 6.12 times as fast as with one core, and 3.35 times as fast as a baseline sequential algorithm. We also found that Scala's default Actors implementation is not optimal for this algorithm, and that a custom-built implementation outperforms it by a significant margin. We conclude that Actors are an effective way to parallelize this type of algorithm.

Keywords: Actors, compilers, concurrency, dataflow analysis, IFDS, Scala

1 Introduction

Multi-core CPU architectures are becoming increasingly common in all types of computer hardware, even now in low-end consumer devices. Learning to use these additional cores is a necessary step in developing more capable software. If compilers and program analysis tools could benefit from the additional computation power available in multi-core computers, then an increase in the precision of these tools could be accomplished without compromising speed.

In this paper, we present an algorithm for solving context-sensitive IFDS (Interprocedural Finite Distributive Subset) dataflow analysis problems [14] in a way which takes advantage of any additional CPU cores which may be present. Constructing this type of algorithm using traditional thread-and-lock expressions can be a difficult exercise because it requires reasoning about shared data consistency in the presence of non-deterministic thread interleavings, reasoning which is extraordinarily difficult for human minds [9, 16]. We approach the task by expressing the algorithm using the Actor model [1, 6]. The Actor model has no notion of shared variables. Instead, each actor maintains a local state only, and communicates by passing messages to other actors. As far as we are aware, this is the first implementation of IFDS which uses a message-passing model to communicate changes in state.

Like IFDS, many other dataflow analysis algorithms use a worklist to iterate to a fixed-point solution, and therefore have the same general structures as IFDS. Although

we do not explore other dataflow analysis algorithms here, we expect the actor-based approach to parallelization to work well for many of them.

This paper is based on previous thesis work with IFDS and Scala's Actor library [15], and we extend it here to include an alternative implementation of the runtime Actor scheduler which supports priority ordering of messages passed.

Section 3 summarizes the nature of IFDS problems and their solution. The single-threaded E-IFDS algorithm, which contains several practical extensions to the original IFDS algorithm, is presented here. Section 4 summarizes the Actor model and its semantics. Section 5 introduces the IFDS-Actors, or IFDS-A, algorithm. Section 6 discusses implementation details, including an Actor scheduler implementation which supports priority ordering of messages. Section 7 contains an empirical evaluation of the performance of IFDS-A and compares it to E-IFDS.

2 Related Work

The IFDS algorithm was originally presented by Reps, Horwitz, and Sagiv as a precise and efficient solution to a wide class of context-sensitive, interprocedural dataflow analysis problems [14]. The Extended IFDS algorithm formalized a set of extensions to the IFDS algorithm which increased its utility for a wider set of analysis problems [11]. The E-IFDS algorithm we present in this paper is essentially Extended IFDS with some minor differences.

Adapting analysis algorithms to operate using multiple CPU cores may lead to significantly improved performance of these algorithms. The Galois system approaches this problem by providing new syntactic constructs which enable thread-safe parallel iteration over unordered sets [7, 8]. Méndez-Lojo, Matthew, and Pingali [10] used the Galois system to implement a multi-core version of a points-to analysis algorithm by Hardekopf and Lin [5]. Using an eight-core computer, they were able to show performance improvements over Hardekopf and Lin for analyses which took longer than 0.5 seconds to run.

A second approach to creating multi-core analysis algorithms is to express them in terms of the Actor Model, which describes computations as sets of logical processes which communicate via message-passing [1, 6]. The Erlang lanugage [19] has built-in support for the Actor model, and Scala [12] includes an implementation of the Actor model in its standard library [4]. Panwar, Kim, and Agha studied the application of the Actor model to graph-based algorithms, and in particular tested varying strategies of work distribution among CPU cores [13].

3 Baseline Sequential Algorithm: E-IFDS

The E-IFDS algorithm is a sequential dataflow analysis algorithm which extends the IFDS algorithm of Reps, Horwitz, and Sagiv [14]. We briefly explain the IFDS dataflow analysis problem, followed by a presentation of E-IFDS.

Dataflow analysis seeks to determine, for each instruction in an input program, facts which must hold during execution of that instruction. The types of facts which the

analysis discovers depend on the type of analysis used. For example, an uninitialized-variables analysis discovers facts of the form "*x* is uninitialized at this instruction," and a variable-type analysis discovers facts of the form "the type of the object *x* points to is a subtype of *T*."

A *control-flow graph* (*CFG*) describes the structure of the input program. Each node in the CFG represents an instruction. A directed edge from instruction a to b indicates that b may execute immediately after a.

A *flow function* models the effect of each type of instruction in the input program. A flow function takes a set of facts as input, and it computes a new set of facts as output. Whenever the dataflow analysis discovers new facts holding after an instruction, it propagates the new facts along the edges in the CFG to the successors of the instruction.

```
      algorithm Solve(N*, s_main, successors, flow)
      begin
[1]     ResultSet := { ⟨s_main, 0⟩ }
[2]     WorkList := { ⟨s_main, 0⟩ }
[3]     while WorkList ≠ ∅ do
[4]       Remove any element ⟨n, d⟩ from WorkList
[5]       for each d' ∈ flow(n, d) and n' ∈ successors(n) do
[6]         Propagate(⟨n', d'⟩)
[7]       od
[8]     od
[9]     return ResultSet
      end

      procedure Propagate(item)
      begin
[10]    if item ∉ ResultSet then Insert item into ResultSet; Insert item into WorkList fi
      end
```

Algorithm 1: A Naive Algorithm for Solving IFDS Problems

IFDS, or Interprocedural Finite Distributive Subset, problems are dataflow analysis problems with the following properties.

- The analysis is *interprocedural* in that it takes the effects of called procedures into account.
- Each instruction is associated with a *finite* set of facts, and each such set is a *subset* of a larger finite fact set D.
- At control flow merge points, the sets of facts coming from different control flow predecessors are combined using set union.
- The flow functions are *distributive*, i.e. for any two fact sets D_1 and D_2, and any flow function f, $f(D_1 \cup D_2) = f(D_1) \cup f(D_2)$. The distributive property enables flow functions to be compactly represented and efficiently composed.

The distributivity of the flow function makes it possible for an analysis to evaluate each transfer function f one fact at a time. For example, consider the input set of facts $D_I = \{a, b, c\}$. This set can be written as the union $\{\} \cup \{a\} \cup \{b\} \cup \{c\}$. Therefore, the result of the transfer function $f(D_I)$ can be computed as $f(\{\}) \cup f(\{a\}) \cup f(\{b\}) \cup f(\{c\})$. In general, the transfer function can be computed for any input sets by taking

the union of the results of applying the transfer function to the empty set and to singleton sets.

Algorithm 1 is a simple algorithm that finds the merge over all paths solution of an IFDS problem. Its inputs are N^*, the set of nodes in the control-flow graph; s_{main}, the entry point of the main procedure; successors, which maps a node in N^* to its control-flow successors in N^*; and flow, the flow function. The flow function takes two parameters n, a node in N^*, and d, a fact in the set $D \cup \{\mathbf{0}\}$. A value of $d \in D$ represents the singleton set $\{d\}$, and $d = \mathbf{0}$ represents the empty set. The flow function evaluates the transfer function for the given node n on the singleton or empty set, and returns a set of facts to be propagated to successor nodes in the CFG.

The ResultSet collects all facts along with the nodes at which they were discovered. The first element put into the ResultSet is $\langle s_{main}, \mathbf{0} \rangle$, indicating that the empty set of facts reaches the beginning of the program. Every time the algorithm discovers a new fact reaching a node, it accumulates it in ResultSet and adds it into the WorkList. Elements from the WorkList may be removed and processed in any order. Whenever an element is removed, the flow function is evaluated on it and any newly generated facts are added to the ResultSet and the WorkList. When the WorkList is empty, no additional facts can be derived, so the algorithm terminates.

The actual IFDS algorithm [14] is more precise in that it computes the merge over all valid paths solution rather than the merge over all paths. A valid path is a path through the interprocedural control flow graph in which calls and returns are matched: the control flow edge taken to return from a procedure must lead to the point of the most recent call of that procedure. Here, we present and parallelize E-IFDS, a variation of the Extended IFDS algorithm [11], which in turn is an extension of the original IFDS algorithm [14]. The full E-IFDS algorithm is given in Algorithm 2.

The differences between E-IFDS and Extended IFDS are:

- The Extended IFDS algorithm maintains a SummaryEdge set, whereas E-IFDS does not.
- The Extended IFDS algorithm explicitly supports the Static Single Assignment form, or SSA, without loss of precision. E-IFDS does not make any explicit provisions for SSA.[1]
- E-IFDS explicitly allows multiple called procedures at a single call-site, whereas the Extended IFDS algorithm does not.

The key idea that enables the IFDS class of algorithms to compute a solution over only valid paths is that they accumulate *path-edges* of the form $d_1 \rightarrow \langle n, d_2 \rangle$ rather than just facts of the form $\langle n, d \rangle$. Instead of representing a fact, a path-edge represents a function: the path-edge $d_1 \rightarrow \langle n, d_2 \rangle$ means that if the fact d_1 is true at the beginning of the procedure containing n, then the fact d_2 is true at n. Given a node n, the set of path-edges terminating at n defines the function:

$$f(D_{in}) = \{d_2 : d_1 \in D_{in} \cup \{\mathbf{0}\} \text{ and } d_1 \rightarrow \langle n, d_2 \rangle \in \text{path-edges}\}$$

[1] Adding SSA support is largely just a matter of propagating predecessor nodes along with the path-edges so that the Phi nodes know which branch a given fact came from.

```
     algorithm Solve(N*, s_main, successors, flow_i, flow_call, flow_ret, flow_thru)
     begin
[1]    PathEdge := { 0 → ⟨s_main, 0⟩ }
[2]    WorkList := { 0 → ⟨s_main, 0⟩ }
[3]    CallEdgeInverse := ∅
[4]    ForwardTabulate()
[5]    return all distinct ⟨n, d2⟩ where some d1 → ⟨n, d2⟩ ∈ PathEdge
     end

     procedure Propagate(item)
     begin
[6]    if item ∉ PathEdge then Insert item into PathEdge; Insert item into WorkList fi
     end

     procedure ForwardTabulate()
     begin
[7]    while WorkList ≠ ∅ do
[8]      Remove any element d1 → ⟨n, d2⟩ from WorkList
[9]      switch n

[10]       case n is a Call Site :
[11]         for each d3 ∈ flow_call(n, d2, p) where p ∈ calledProcs(n) do
[12]           Propagate(d3 → ⟨s_p, d3⟩)
[13]           Insert ⟨p, d3 → ⟨n, d2⟩⟩ into CallEdgeInverse
[14]           for each d4 such that d3 → ⟨e_p, d4⟩ ∈ PathEdge do
[15]             for each d5 ∈ flow_ret(e_p, d4, n, d2) do
[16]               Propagate(d1 → ⟨returnSite(n), d5⟩)
[17]             od
[18]           od
[19]         od
[20]         for each d3 ∈ flow_thru(n, d2) do
[21]           Propagate(d1 → ⟨returnSite(n), d3⟩)
[22]         od
[23]       end case

[24]       case n is the Exit node e_p :
[25]         for each ⟨c, d4⟩ such that ⟨p, d1 → ⟨c, d4⟩⟩ ∈ CallEdgeInverse do
[26]           for each d5 ∈ flow_ret(e_p, d2, c, d4) do
[27]             for each d3 such that d3 → ⟨c, d4⟩ ∈ PathEdge do
[28]               Propagate(d3 → ⟨returnSite(c), d5⟩)
[29]             od
[30]           od
[31]         od
[32]       end case

[33]       case n is not a Call Site or Exit node :
[34]         for each d3 ∈ flow_i(n, d2) and n' ∈ successors(n) do
[35]           Propagate(d1 → ⟨n', d3⟩)
[36]         od
[37]       end case

[38]     end switch
[39]   od
     end
```

Algorithm 2: The E-IFDS Algorithm

Thus the path-edges accumulated at the return of a procedure define a function that computes the facts holding after the procedure from a given set of facts holding before that specific call of the procedure.

Intra-procedurally, the algorithm generates new path-edges by composing existing path-edges with the flow function. If a path-edge $d_1 \rightarrow \langle n, d_2 \rangle$ exists and n' is a control flow successor of n, then for each $d_3 \in \text{flow}(n, d_2)$, a new path-edge $d_1 \rightarrow \langle n', d_3 \rangle$ is created (lines 33–37). Thus the computed path-edges represent the transitive closure of the flow function within each procedure.

The flow function for E-IFDS is separated into four functions for convenience:

- $\text{flow}_i(n, d)$: Returns the facts derivable from d at instruction node n.
- $\text{flow}_{call}(n, d, p)$: Computes call-flow edges from the call-site n to the start of a called procedure p.
- $\text{flow}_{ret}(n, d, c, d_c)$: Computes return-flow edges from the exit node n to the return-site. The fact d_c at the call-site c is the *caller context* required by some analyses; $\langle n, d \rangle$ is reachable from $\langle c, d_c \rangle$.
- $\text{flow}_{thru}(n, d)$: A convenience function which allows transmission of facts from call-site to return-site without having to propagate through called procedures.

When a call is encountered, the algorithm creates *summary edges* which summarize the effect of a procedure from the caller's point of view. Each summary edge is the composition of a call-flow edge, a path-edge of the called procedure, and a return-flow edge. The summary edges are then composed with existing path-edges that lead to the call site to create new path-edges that lead to the return site. Lines 14–18 and lines 25–31 compute summary edges and propagate the corresponding new path-edges. Figure 1 illustrates the relationships between the CFG (left), dataflow edges determined by flow functions (right, solid lines), and dataflow edges computed by the algorithm (right, dashed lines). Whenever a summary edge is generated, E-IFDS uses it to generate new path-edges, as if the summary edge were an ordinary dataflow edge. Unlike the original IFDS algorithm, E-IFDS does not need to keep a set of all summary edges; it discards each summary edge immediately after using it to extend a path-edge.

4 The Actor Model

The basic notion behind the Actor model is that any computation can be defined as a set of entities, or *actors*, which communicate by passing messages. Each actor processes the messages it receives in some sequential order. The actor buffers received messages until it can process them, as shown in Figure 2. The theory behind this model was orginally developed by Hewitt [6] and the semantics of actors were refined by a number of others, notably Agha [1].

The main difference between actor-based programming and object-oriented programming is that actor-based message-passing is *asynchronous* and *unordered*[2], whereas in many object-oriented systems method calls are by default synchronous and ordered. The sending actor does not wait for the receiving actor to process the message, and the time between the send and the receive may be arbitrarily long. [1, 6]

[2] As we will show later, however, applying a message prioritization policy can result in performance improvements.

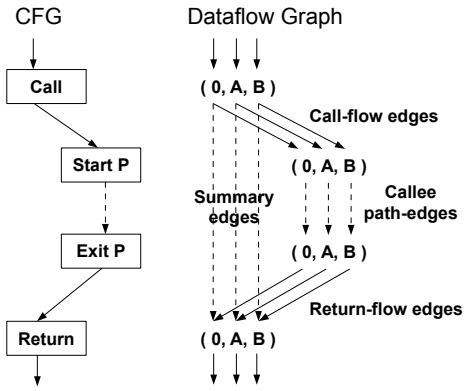

Fig. 1. Generating Call-Site Summary Edges for E-IFDS

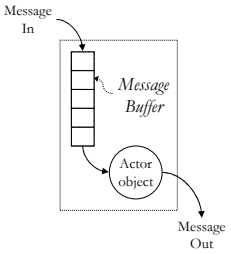

Fig. 2. The Actor Abstraction

The notation used to define actors is shown in Figure 3. An actor definition is called an *actor class* to distinguish it from an ordinary class. This notation contains the following components: a name, a list of arguments, a statement block $stmts_{init}$ which executes upon actor construction, and a set of cases which are matched against incoming messages. A match succeeds if the message type and the number of message parameters is the same as the pattern. For example, the message AddEdge⟨"A","B"⟩ matches the pattern AddEdge$\langle d_1, d_2 \rangle$. When the match succeeds, the values of d_1 and d_2 are "A" and "B", respectively. Scala's pattern matching semantics select the first pattern that matches; subsequent patterns are not tested. Actor classes may also contain a **finally** clause, which is executed immediately after the execution of any case statement.

```
def ActorName(arguments)
  stmts_init
begin (message) switch
  case message matches pattern_1 : stmts_1
  ...
  case message matches pattern_n : stmts_n
  finally : stmts_final
end
```

Fig. 3. Actor Class Definition

All argument variables and all local variables created by $stmts_{init}$ persist for the lifetime of the actor and are visible to all statements $stmts_1$ through $stmts_n$ and $stmts_{final}$. These variables are analogous to member variables in object-oriented languages; they persist until the actor is garbage-collected. Any variables created by $stmts_1$ through $stmts_n$ or $stmts_{final}$ are only live and visible inside their respective statement blocks. These variables are created in response to a received message and so do not persist after the message is processed.

When the actor receives a message, it selects at most one case statement for processing. Local variables created by the statements following **case** are stored in a temporary frame that is discarded when those statements finish executing.

5 Actor-based Parallel Algorithm: IFDS-A

The IFDS-Actors algorithm, or IFDS-A, takes the same parameters and produces the same results as E-IFDS, but takes advantage of additional CPU cores. Algorithm 3 shows the main IFDS-A algorithm. Algorithm 4 defines the actors that respond to path-edge propagation.

IFDS-A is based on a simple conceptual mapping. IFDS-A constructs one actor for each node in the CFG. For each propagated path-edge, IFDS-A sends a message. The algorithm does not need a centralized WorkList because the actor library implicitly buffers all messages until they are processed. Instead of a centralized PathEdge set, each actor records a local set of all the path-edges leading into it. Where E-IFDS stores

```
     algorithm Solve(N*, s_main, successors, flow_i, flow_call, flow_ret, flow_thru)
     begin
[1]    for each n ∈ N* do switch
[2]      case n is a Call Site : N^A[n] := new CallSiteActor(n) end case
[3]      case n is the Exit node e_p : N^A[n] := new ExitActor(p) end case
[4]      case n is not a Call Site or Exit node : N^A[n] := new IntraActor(n) end case
[5]    end switch od
[6]    Tracker := new TrackerActor(currentThread)
[7]    Propagate(s_main, AddPathEdge⟨0, 0⟩)
[8]    Wait for Done⟨⟩
[9]    return all distinct ⟨n, d2⟩ where some d1 → d2 ∈ N^A[n].PathEdge
     end

     pure function Propagate(n, message)
     begin
[10]   Send synchronous Inc⟨⟩ to Tracker
[11]   Send message to N^A[n]
     end

     def TrackerActor(receiver)
[12]   local count := 0
     begin (message) switch
[13]   case message matches Inc⟨⟩ :
[14]     count := count + 1
[15]   case message matches Dec⟨⟩ :
[16]     count := count - 1
[17]     if count = 0 then Send Done⟨⟩ to receiver fi
     end
```

Algorithm 3: The Top-Level IFDS-A Algorithm

a path-edge $d_1 \rightarrow \langle n, d_2 \rangle$, the IFDS-A node-actor corresponding to CFG node n, or $N^A[n]$, simply stores $d_1 \rightarrow d_2$.

IFDS-A defines three types of node-actors, corresponding to call-site nodes, exit nodes, and other instruction nodes. Lines 1–5 in Algorithm 3 create the node-actors, and Algorithm 4 defines the node-actor classes.

In addition to a PathEdge set, the CallSiteActor contains a CallEdge set. The CallEdge set retains the known call-flow edges (Algorithm 4 line 9) because summary-edge generation requires the inverse of flow_{call} (i.e. given some fact d_2, finding all d_1 such that $d_2 \in \text{flow}_{call}(n, d_1, p)$) (line 22). CallEdge stores elements of the form $\langle p, d_1 \rightarrow d_2 \rangle$ because it must remember both the call-flow edge $d_1 \rightarrow d_2$ and the procedure p the edge goes to. The function of CallEdge in IFDS-A is the same as CallEdgeInverse in E-IFDS.

Node-actor operation is identical to the corresponding operations in E-IFDS, with one exception: IFDS-A's exit node-actor forwards its edges to the call-site node-actor instead of generating summary edges itself. The reason for this is that the summary edges generated in response to facts at the exit node require access to the PathEdge and CallEdge sets present in the CallSiteActor. Summary edge generation must read both of these sets in one atomic operation to avoid missing updates to one of them. Therefore, the ExitActor sends every path-edge it receives to all of its call-site actors via the AddCalleePathEdge message.

The CallSiteActor retains the path-edges it receives in the CalleePathEdge set, for use by the normal AddPathEdge code (Algorithm 4 lines 10–14). E-IFDS call-site code

```
     def CallSiteActor(n)
[1]    local PathEdge := ∅
[2]    local CallEdge := ∅
[3]    local CalleePathEdge := ∅
     begin (message) switch
[4]    case message matches AddPathEdge⟨d1, d2⟩ :
[5]      if d1 → d2 ∉ PathEdge then
[6]        Insert d1 → d2 into PathEdge
[7]        for each p ∈ calledProcs(n) and d3 ∈ flow_call(n, d2, p) do
[8]          Propagate(sp, AddPathEdge⟨d3, d3⟩)
[9]          Insert ⟨p, d2 → d3⟩ into CallEdge
[10]         for each d4 such that ⟨p, d3 → d4⟩ ∈ CalleePathEdge do
[11]           for each d5 ∈ flow_ret(ep, d4, n, d2) do
[12]             Propagate(returnSite(n), AddPathEdge⟨d1, d5⟩)
[13]           od
[14]         od
[15]       od
[16]       for each d3 ∈ flow_thru(n, d2) do
[17]         Propagate(returnSite(n), AddPathEdge⟨d1, d3⟩)
[18]       od
[19]     fi
[20]   case message matches AddCalleePathEdge⟨p, d1, d2⟩ :
[21]     Insert ⟨p, d1 → d2⟩ into CalleePathEdge
[22]     for each d4 such that ⟨p, d4 → d1⟩ ∈ CallEdge do
[23]       for each d5 ∈ flow_ret(ep, d2, n, d4) do
[24]         for each d3 such that d3 → d4 ∈ PathEdge do
[25]           Propagate(returnSite(n), AddPathEdge⟨d3, d5⟩)
[26]         od
[27]       od
[28]     od
[29]   finally : Send Dec⟨⟩ to Tracker
     end

     def ExitActor(p)
[30]   local PathEdge := ∅
     begin (message) switch
[31]   case message matches AddPathEdge⟨d1, d2⟩ :
[32]     if d1 → d2 ∉ PathEdge then
[33]       Insert d1 → d2 into PathEdge
[34]       for each c ∈callers(p) do
[35]         Propagate(c, AddCalleePathEdge⟨p, d1, d2⟩)
[36]       od
[37]     fi
[38]   finally : Send Dec⟨⟩ to Tracker
     end

     def IntraActor(n)
[39]   local PathEdge := ∅
     begin (message) switch
[40]   case message matches AddPathEdge⟨d1, d2⟩ :
[41]     if d1 → d2 ∉ PathEdge then
[42]       Insert d1 → d2 into PathEdge
[43]       for each d3 ∈ flow_i(n, d2) and n′ ∈ successors(n) do
[44]         Propagate(n′, AddPathEdge⟨d1, d3⟩)
[45]       od
[46]     fi
[47]   finally : Send Dec⟨⟩ to Tracker
     end
```

Algorithm 4: IFDS-A Node-Actor Classes

accesses callee path-edges directly (Algorithm 2 lines 14–18), but the concurrent environment of IFDS-A requires each CallSiteActor to retain its own copy.

IFDS-A does not have a centralized WorkList, so it cannot use the "empty-worklist" condition to determine analysis completion. Instead, it uses a separate Tracker object to detect completion. The Tracker, created on line 6 and defined on lines 12–17 of Algorithm 3, keeps a count of unprocessed node-actor messages. Every time an actor calls Propagate to issue a new unit of work, it issues an Inc$\langle\rangle$ message to increment the Tracker's count. Whenever an actor finishes processing a message, it sends a Dec$\langle\rangle$ message to the Tracker to decrement the count. When the count reaches zero, the Tracker sends Done$\langle\rangle$ (line 17, Algorithm 3) to wake up the main thread (line 8). Propagate sends Inc$\langle\rangle$ synchronously because the Tracker must process the Inc$\langle\rangle$ before its corresponding Dec$\langle\rangle$; otherwise the count could reach zero before all units of work complete. Communicating with the Tracker may appear to incur excessive message-passing overhead, but in practice we implement the Tracker with hardware-supported atomic-integer operations.

6 Implementation

We implement E-IFDS and IFDS-A in the Scala language, version 2.8.0 Beta 1. We use Soot [18], a Java bytecode optimization framework, to convert the input programs into CFGs suitable for our analysis.

6.1 The Variable Type Analysis

The analysis used for evaluation is the flow-sensitive variant of Variable Type Analysis (VTA) [17] defined by Naeem, et al. [11]. This analysis defines the fact-set D to be the set of all pairs $\langle v, T\rangle$ where v is a variable and T is a class type in the source program. The presence of a fact $\langle v, T\rangle$ in the result set means that the variable v may point to an object of type T. Stated differently, the presence of $\langle v, T\rangle$ means that the analysis is unable to prove that v will *not* point to an object of type T.

Redundant Fact Removal The original IFDS algorithm does not assume any relationships among facts, yet VTA and some other analyses do provide structured relationships. For VTA, if $\langle v, T\rangle$ and $\langle v, superclass(T)\rangle$ are in the same fact-set, then $\langle v, T\rangle$ is redundant. In our implementation of VTA, we check for redundant facts whenever we consider inserting a new element in the PathEdge set. A path-edge $d_1 \rightarrow \langle n, d_2\rangle$ where d_2 is redundant is not inserted. If d_2 is not redundant, then any other path-edges which become redundant are removed from the PathEdge set. This prevents redundant facts from being considered in any future processing.

Priority Ordering When using redundant fact removal, it may be advantageous to execute worklist items in a prioritized order. We implement an ordering for VTA facts which gives a higher priority to path-edges where the type T in d_2 has a smaller distance from the root type (i.e. Object). Without this ordering, the algorithms could do the work of constructing large fact sets only to discover later that much of the work was unnecessary.

6.2 Scheduling Actor Executions

Executing an actor-based algorithm on current mainstream hardware requires a *scheduler* to distribute work items among available processors. Normally, the scheduler creates some number of *worker threads* which the operating system dynamically assigns to available processors. Each unit of work is passed to the scheduler as soon as it is generated, and the scheduler eventually[3] assigns it to a worker thread for execution.

Scala's Actor Library In our first implementation, we used Scala's standard Actor library. Scala's Actor implementation creates a Mailbox, or queue of unprocessed messages, for each actor created. The sending actor calls the `Actor.send` function on the receiving actor to insert a message into the receiving actor's Mailbox. Whenever the receiving actor completes processing a message, it checks its Mailbox. If the Mailbox is non-empty, it removes a message which it passes to the underlying scheduler (by default, the Java `ForkJoinPool`) for execution. If the Mailbox is empty when the receiving actor finishes processing a message (or if no messages have yet been sent to the it), it becomes idle. If the receiving actor is idle, then calls to `Actor.send` bypass the Mailbox and submit the message directly to the scheduler.

There are two weaknesses of this implementation. The first is a performance weakness. In our preliminary experiments, we found that the overhead of handling a message was approximately 2 to 5 μs, an overhead which in some cases approached 50% of total execution time. Overhead from the `synchronized` keyword is incurred on every call to `Actor.send` and every call to the scheduler, which may be one source of inefficiency. In addition, many messages must be queued and dequeued twice: once in a Mailbox, and again in the scheduler. Furthermore, the Scala Actor library seems to suffer from some scalability problems. For some inputs, the implementation runs more slowly with 8 threads than 4 threads. The second weakness of this implementation is that it does not support priority ordering of messages.

The Task-Throwing Scheduler We created the Task-Throwing Scheduler to remedy these weaknesses. Instead of implementing an actor abstraction on top of a generic scheduler, we created a scheduler which is built specifically for actor-based programs. Algorithm 5 shows the Task-Throwing Scheduler.

There are two basic concerns when implementing an actor scheduler. The first concern is efficient scheduling of executable tasks, a concern which is common to all schedulers. In two respects we follow the lead of Arora et al. [2]. First, that work-stealing schedulers tend to make efficient use of processor resources, and second, that calling a `Yield()` statement after failing to acquire a lock is necessary for optimal performance on systems where we have no direct control over how the operating system schedules threads. Each of `T` threads in the scheduler runs Algorithm 5 until stopped. Each thread executes tasks from its own queue (specified by qid) until a failure occurs, at which time it yields control to any other threads ready to run (line 3) and then executes a task from

[3] By "eventually" we mean that every unit of work must be executed, not that the delay between time of submission to the scheduler and execution time is necessarily long.

```
     declare Q: static array of queues [0 .. T - 1] of pairs (Actor &, Message)
     declare L: static array of volatile booleans [0 .. T - 1]

     abstract class Actor:
       lock: volatile boolean
       qid: Integer
       virtual function execute (m: Message)
     end

     algorithm WorkerThread (qid: Integer)
     begin
[1]    while true do
[2]      if ExecuteNext (qid) == false then
[3]        Yield ()
[4]        while ExecuteNext (rand () % T) == false do Yield () od
[5]      fi
[6]    od
     end

     static function ExecuteNext (qid: Integer)
     begin
[7]    if TryAcquire (L[qid]) then
[8]      if Q[qid] is not empty then
[9]        Remove next (a, m) from Q[qid]
[10]       Release (L[qid])
[11]       if TryAcquire (a.lock) then
[12]         a.qid := qid
[13]         a.execute (m)
[14]         Release (a.lock)
[15]         return true
[16]       else
[17]         Send (a, m)
[18]       fi
[19]     else
[20]       Release (L[qid])
[21]     fi
[22]   fi
[23]   return false
     end

     global function Send (a: Actor &, m: Message)
     begin
[24]   Acquire (L[a.qid])
[25]   Insert (a, m) into Q[a.qid]
[26]   Release (L[a.qid])
     end

     function TryAcquire (lock: volatile boolean &)
     begin
[27]   return AtomicExchange (&lock, true) == false
     end

     function Acquire (lock: volatile boolean &)
     begin
[28]   while AtomicExchange (&lock, true) == true do Yield () od
     end

     function Release (lock: volatile boolean &)
     begin
[29]   lock := false
     end
```

Algorithm 5: The Task-Throwing Scheduler

a random queue (line 4) before checking its own queue again. Each queue is protected by a lock (from array `L`).

The second basic concern of an actor scheduler is maintaining mutually exclusive execution of tasks destined for the same actor. We accomplish this by including a lock for each actor which is acquired prior to running the user-defined `Actor.execute` function (lines 11–14). Each actor also includes qid, the queue number the currently executing task came from (line 12). If another thread attempts to execute a second task on the actor while it is busy, then lock acquisition fails and the task is re-inserted into the queue that the first task came from (line 17). This "throwing" action not only maintains mutual exclusion between tasks executing on the same actor, but also tends to group these tasks into the same work queue.

The Task-Throwing Scheduler uses boolean variables as locks, and requires the underlying hardware to support an atomic-exchange operation. Lock variables are `true` if held by some thread, `false` if not. The `AtomicExchange` function atomically writes a given value to a lock variable and returns the value previously held. The `TryAcquire` function (line 37) makes a single attempt to acquire a lock, returning `true` if successful. `ExecuteNext` uses `TryAcquire` so that it doesn't block execution while there is potentially other useful work the thread could be doing. This non-blocking behaviour is particularly important when acquiring a lock on an actor, which could otherwise block for an arbitrarily long period of time. The `Send` function, which may be called from any thread or actor, uses `Acquire` (line 38), which blocks until the queue lock becomes free. Unlike actor locks, queue locks are only held for very short periods of time. Releasing a lock is as simple as setting the lock variable to `false` (line 39).

The Task-Throwing Scheduler is deadlock-free. An informal proof of this is as follows: After a queue lock is acquired by a thread, it is always released before the thread acquires any other locks (lines 7–10,20 and 24–26). While holding an actor lock (lines 11–14), the user-defined `Actor.execute` function may subsequently call `Send`, which acquires a queue lock. A thread holding an actor lock always releases it before attempting to acquire any other actor lock. Since any thread can hold at most one actor lock and one queue lock, and the actor lock is always acquired first, it follows that the execution of the scheduler code cannot introduce any lock acquisition cycles, and therefore cannot cause a deadlock.

7 Evaluation

We ran our performance tests on an eight-core AMD Opteron machine running the Oracle JRockit JVM version 3.1.2. We used the Oracle JVM because we were having some problems with the Sun JVM crashing.

Our test inputs are the programs luindex, antlr, and jython from the DaCapo Benchmark Suite version 2006-10-MR2 [3]. We analyse the source of the benchmarks only, but not the standard libraries. We make conservative worst-case assumptions about the types of objects returned from standard library methods.

7.1 Available Parallelism

Before evaluating actual performance, we wanted to estimate the maximum amount of parallelism available in the IFDS-A algorithm. To perform this estimation, we execute IFDS-A with a single-threaded worklist. Execution is performed using a doubly-nested loop, where the outer loop finds a maximal set of work units from the work-list that could be executed in parallel, and the inner loop executes this set of work units. Two work units are deemed to be executable in parallel if and only if they operate on different actors. The number of iterations the inner loop performs for a single iteration of the outer loop is the *available parallelism* (or *parallel width*), and the number of outer loop iterations is the length of a chain of sequentially dependent operations (or *parallel depth*). If all work units required the same amount of time to execute, the parallel depth is the optimal amount of time in which the algorithm could execute to completion, and the maximum parallel width is the number of processors that would be necessary to achieve this optimal time.

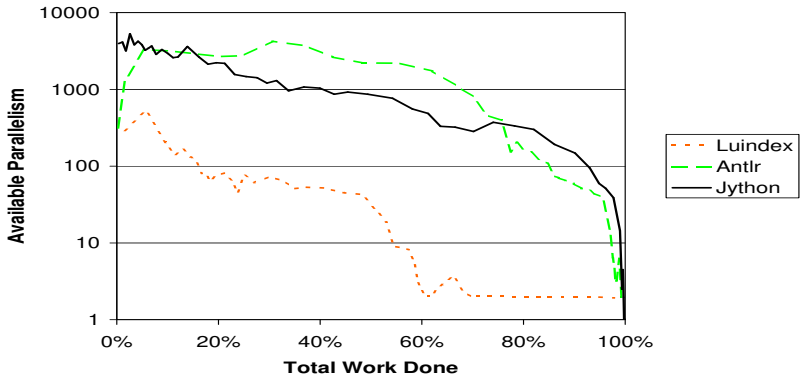

Fig. 4. Available Parallelism

Figure 4 shows the available parallelism for each input as a function of the percentage of work units completed. These charts provide an indication of how many processors the algorithm can keep busy for the given input, and for how long. For example, antlr provides sufficent parallelism to keep 100 processors busy for the first 85% of work units processed. The last 15% of work units will take longer than 15% of the total execution time because of insufficient parallelism to keep 100 processors busy. Jython has a similarly large available parallelism. In contrast, luindex has considerably smaller available parallelism, where a substantial fraction of the total work done exhibits a parallel width of only 2 units. This lack of available parallelism may limit the performance scalability of luindex.

7.2 Performance

We measured the performance of IFDS-A with the Scala Actor library and our Task Throwing Scheduler, and compared the results with the E-IFDS reference implementation. We tested the effects of priority queuing and different thread counts on the implementation's performance.

			LUINDEX				ANTLR				JYTHON			
Policy	Scheduler	Th	Time	Acc	Rej	Fwd	Time	Acc	Rej	Fwd	Time	Acc	Rej	Fwd
	None (E-IFDS)	1	38	0.9	-	-	220	5.9	-	-	363	10	-	-
	ScalaActor	1	92	1.1	1.3	0.06	432	6.2	3.1	0.4	881	14	39	19
FIFO	ScalaActor	**8**	**18**	**1.2**	**1.4**	**0.06**	**89**	**6.2**	**3.1**	**0.4**	**242**	**14**	**40**	**19**
	TaskThrow	1	81	1.2	1.2	0.05	371	6.2	3.2	0.4	533	13	37	18
	TaskThrow	**8**	**12**	**0.9**	**0.9**	**0.06**	**60**	**6.0**	**3.1**	**0.4**	**176**	**16**	**48**	**24**
	None (E-IFDS)	1	14	0.25	-	-	235	5.2	-	-	198	5.6	-	-
Priority	TaskThrow	1	31	0.37	0.30	0.02	366	5.2	2.4	0.4	349	5.6	8.2	7.2
	TaskThrow	**8**	**5**	**0.36**	**0.29**	**0.02**	**62**	**5.3**	**2.5**	**0.4**	**63**	**5.3**	**7.7**	**6.9**

Table 1. Comparing Scheduling Methods

Table 1 summarizes the effects of the scheduler policy on performance. The columns of this table are:

- Policy: Indicates FIFO or priority processing of worklist elements.
- Scheduler: "None" is the single-threaded E-IFDS algorithm, "ScalaActor" is the default scheduler in the Scala Actors library, and "TaskThrow" is the new task-throwing scheduler.
- Th: Number of worker threads. Rows with a thread count of 8 are shown in bold. Each worker thread has its own worklist.
- Time: Average (geometric mean) time in seconds taken to perform one solve.
- Acc: Average number of worklist items accepted for processing, in millions.
- Rej: Average number of worklist items rejected after removal from a worklist because they had already been processed earlier, in millions.
- Fwd: Average number of worklist items forwarded from end-nodes to call-site nodes, in millions.

The scheduling policy is only enforced within a single worker thread, not across worker threads. Different worker threads can move through their worklists at different speeds depending on how long each item takes to process. Task-throwing, work-stealing, lock acquisition, and kernel scheduling activities also affect which worker processes which message. It is interesting to note that using multiple worker threads can sometimes result in execution orders that are more efficient than a strict adherence to the scheduling policy.

The task-throwing scheduler is significantly faster than Scala's default scheduler on all the benchmarks tested. Furthermore, the task-throwing scheduler supports priority ordering of messages, whereas the default Actors implementation does not.

There are two major sources of overhead in the parallel IFDS-A algorithm compared to the sequential E-IFDS algorithm. The first source of overhead is passing messages for redundant path-edges. Unlike E-IFDS, which detects redundant path-edges before insertion into the worklist, IFDS-A checks for redundancy only after a message is received. The second source is forwarding end-node path-edges to call-site nodes. While E-IFDS can handle these edges immediately, IFDS-A must re-send each path-edge at an end-node to all associated call-site nodes.

For luindex, rejected and forwarded messages account for slightly more than half of the total messages sent by IFDS-A, more than doubling the total number of worklist items processed by E-IFDS. For antlr, rejected and forwarded messages account for a somewhat smaller proportion of the total messages, and for jython, they account for a signficanly larger proportion.

		LUINDEX		ANTLR		JYTHON	
Scheduler	Th	Time	Sp/Sc	Time	Sp/Sc	Time	Sp/Sc
E-IFDS	1	13.6	1.0/ -	235.0	1.0/ -	198.4	1.0/ -
TaskThrow	1	30.9	0.4/1.0	366.1	0.6/1.0	349.1	0.6/1.0
	2	14.6	0.9/2.1	188.9	1.2/1.9	177.2	1.1/2.0
	4	8.0	1.7/3.9	101.0	2.3/3.6	99.0	2.0/3.5
	8	5.2	2.6/6.0	61.5	3.8/6.0	62.6	3.2/5.6
	16	4.7	2.9/6.6	61.4	3.8/6.0	60.0	3.3/5.8
	32	5.2	2.6/6.0	61.8	3.8/5.9	69.6	2.9/5.0
	64	4.8	2.8/6.4	61.6	3.8/5.9	68.3	2.9/5.1

Table 2. Performance with Different Thread Counts (Priority Scheduling Policy)

Table 2 shows the performance of IFDS-A using the priority scheduling policy. Two performance figures for each benchmark/thread count combination. The first, "Sp," is the speedup obtained relative to the sequential E-IFDS. The second, "Sc," is the scalability of performance relative to IFDS-A with one thread. Figures 5, 6, and 7 show the speedup graphically. The vertical error bars are the 95% confidence intervals.

When only one thread is available, the additional overhead of IFDS-A puts it at a substantial performance disadvantage. With two threads, however, IFDS-A is largely able to match the performance of E-IFDS, and with more threads it is able to exceed the performance of E-IFDS by a substantial margin. Luindex exhibited the worst speedup of the three, and antlr exhibited the best. Luindex may have a performance disadvantage because the parallelism available in the problem is limited. Jython is hindered by message-passing overhead because of the relatively large number of rejected and call-site forwarding messages it generates.

Although our test machine has only eight cores, we also test with 16, 32, and 64 threads to see how our scheduler behaves with thread counts larger than the number of cores. As it turns out, 16 threads provides a small performance improvement over eight threads, indicating that our scheduling algorithm is not making full use of all processing resources available when only given eight threads. At 32 and 64 threads, however, we

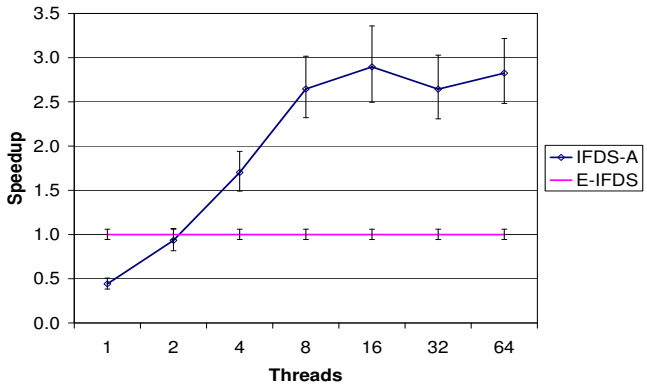

Fig. 5. Performance Chart for LUINDEX Analysis

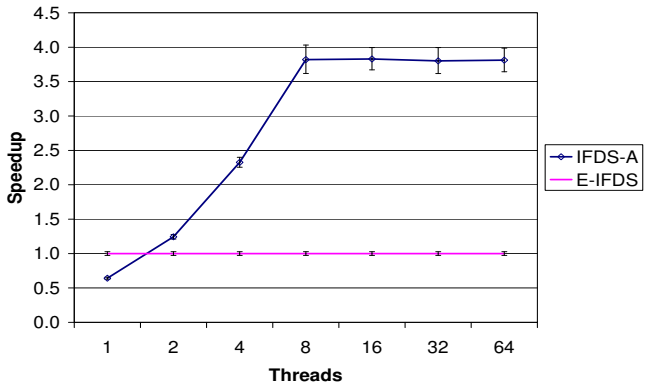

Fig. 6. Performance Chart for ANTLR Analysis

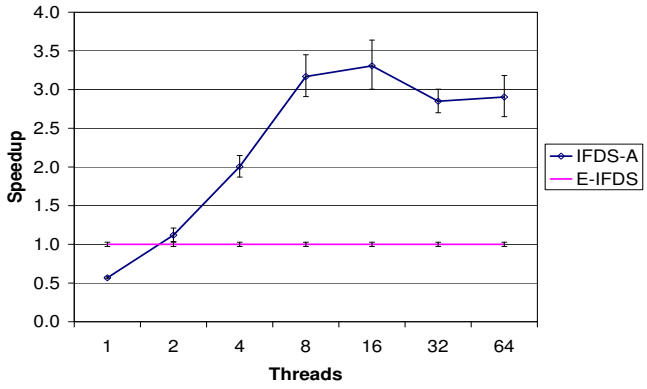

Fig. 7. Performance Chart for JYTHON Analysis

see performance worsen slightly due to the extra processing overhead incurred by larger numbers of threads.

At peak performance on our eight-core machine, we see IFDS-A reach 2.90x, 3.83x, and 3.31x speedup relative to E-IFDS for luindex, antlr, and jython, respectively, for an average speedup of 3.35x. Maximum scalability of these benchmarks is 6.56x, 5.97x, and 5.82x for an average scalability of 6.12x.

Since the slopes of these performance curves only level off after reaching eight threads on our eight-core machine, we can reasonably expect additional performance improvements on machines with larger numbers of cores.

8 Conclusions

This work began as an effort to see if a computationally expensive context-sensitive dataflow analysis algorithm could be expressed using message-passing, in the hope that some performance gain on multi-core computers would be seen. The resulting algorithm, IFDS-A, yielded performance gains which were significantly better than we initially expected. To the best of our knowledge, IFDS-A is the first implementation of IFDS that uses message-passing to communicate changes in state.

The implementation of IFDS-A performs substantially worse on a single core than the equivalent E-IFDS implementation. With two cores, there is still little reason to use IFDS-A because its performance is not much better E-IFDS (and in some cases is worse). With four or more cores, however, IFDS-A outperforms E-IFDS by a significant margin. On an eight-core computer, IFDS-A is on average 6.12 times as fast as it is with a single core, and 3.35 times as fast as the baseline implementation.

Priority ordering of worklist items was possible with the right scheduling mechanism, but it required implementation of a new scheduler.

There are several directions for possible future work, which include:

- verifying performance against a larger number of benchmarks,
- applying actor-based techniques to other types of analyses,
- reducing overhead by only creating one actor per function or one actor per basic block, and
- experimenting with different scheduling mechanisms to improve performance.

Acknowledgements This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada.

References

1. Agha, G.: Actors: A model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
2. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Systems 34(2), 115–144 (2001)

3. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA (2006)
4. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)
5. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 290–299. ACM, New York, NY, USA (2007)
6. Hewitt, C., Baker, H.: Laws for communicating parallel processes. In: IFIP (1977)
7. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Casçaval, C.: How much parallelism is there in irregular applications? In: PPoPP (2009)
8. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. In: PLDI (2007)
9. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
10. Méndez-Lojo, M., Mathew, A., Pingali, K.: Parallel inclusion-based points-to analysis. In: OOPSLA (2010)
11. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical extensions to the IFDS algorithm. In: CC, LNCS, vol. 6011, pp. 124–144. Springer-Verlag, Berlin (2010)
12. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide. Artima Incorporation, USA (2008)
13. Panwar, R., Kim, W., Agha, G.: Parallel implementations of irregular problems using high-level actor language. In: IPPS (1996)
14. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL (1995)
15. Rodriguez, J.D.: A Concurrent IFDS Dataflow Analysis Algorithm Using Actors. Master's thesis, University of Waterloo, Canada (2010)
16. Stein, L.A.: Challenging the computational metaphor: Implications for how we think. Cybernetics and Systems 30(6), 473–507 (1999)
17. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. In: OOPSLA (2000)
18. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: CASCON (1999)
19. Virding, R., Wikström, C., Williams, M.: Concurrent programming in ERLANG (2nd ed.). Prentice Hall International (UK) Ltd., Hertfordshire, UK (1996)