

Scaling Inter-procedural Dataflow Analysis on the Cloud

Static Analysis and Software Verification

Simone Colli

2025-12-10

Index

1 Introduction

2 Background

3 Methods

4 Results

5 Conclusion

Dataflow Analysis

Introduction

Definition: Dataflow analysis is a static technique used to gather information about the possible set of values calculated at various points in a program.

Beyond compiler optimizations, dataflow analysis is widely used in:

- Bug detection.
- Security analysis.

Scalability

Introduction

Applying inter-procedural dataflow analysis to large, real-world software presents major challenges:

- ① **Memory explosion:** Analysis tracks information at every program point. For precise analysis (e.g., context-sensitive alias analysis), the state size is huge.
- ② **Compute intensity:** Flow-sensitive analysis requires updating facts via transfer functions for every statement until convergence (Fixpoint).

Theoretical foundation

Background

The framework relies on the standard Monotone Dataflow Analysis Framework.

Mathematical model

- **Lattice L :** Represents the domain of abstract values.
- **Transfer function $f: L \rightarrow L$.** Models the effect of statements (e.g., *GEN/KILL* sets).
- **Merge operator \sqcup (Join) or \sqcap (Meet):** Combines information from predecessors.

The goal is to compute the **least fixpoint** (lfp) iteratively.

Worklist algorithm

Background

The standard sequential worklist algorithm iteratively updates dataflow facts:

- ① Initialize all IN sets to \perp .
- ② Add all CFG nodes to the worklist.
- ③ While the worklist is not empty:
 - Remove a node k from the worklist.
 - Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.
 - Compute $OUT_k = f_k(IN_k)$.
 - If OUT_k changed, add successors of k to the worklist.

This continues until no more changes occur (fixpoint reached).

Worklist animation: Step 1

Background



- Initialize all IN sets to \perp .

- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

Worklist: {A, B, C, D}

Worklist animation: Step 1

Background



- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

- Initialize all IN sets to \perp .
- Add all CFG nodes to the worklist.

Worklist: {A, B, C, D}

Worklist animation: Step 2

Background



- Remove A from worklist.

- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in pred(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

Worklist: {B, C, D}

Worklist animation: Step 2

Background



- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in pred(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

- Remove A from worklist.
- Compute IN_A (no predecessors).

Worklist: {B, C, D}

Worklist animation: Step 2

Background



- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in pred(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

Worklist: {B, C, D}

Worklist animation: Step 2

Background



- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in pred(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

- Remove A from worklist.
- Compute IN_A (no predecessors).
- Compute $OUT_A = f_A(IN_A)$.
- OUT_A changed, add B and C to worklist.

Worklist: {B, C, D}

Worklist animation: Step 3

Background

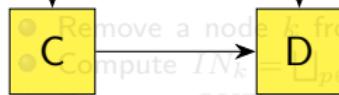


- Remove B from worklist.

1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node k from the worklist.

- Compute $IN_k = \bigcup_{p:preds(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

Worklist: {C, D}

Worklist animation: Step 3

Background

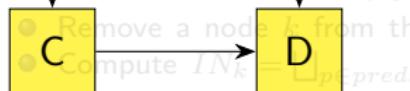


- Remove B from worklist.
- Compute $IN_B = OUT_A$.

1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node k from the worklist.

- Compute $IN_k = \bigcup_{p:preds(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

Worklist: {C, D}

Worklist animation: Step 3

Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node k from the worklist.

- Compute $IN_k = \bigcup_{p:preds(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

- Remove B from worklist.
- Compute $IN_B = OUT_A$.
- Compute $OUT_B = f_B(IN_B)$.

Worklist: {C, D}

Worklist animation: Step 3

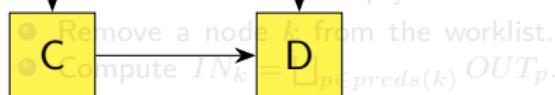
Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node l from the worklist.
- Compute $IN_k = \cup_{p:preds(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.
- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

- Remove B from worklist.
- Compute $IN_B = OUT_A$.
- Compute $OUT_B = f_B(IN_B)$.
- OUT_B changed, add D to worklist.

Worklist: {C, D}

Worklist animation: Step 4

Background

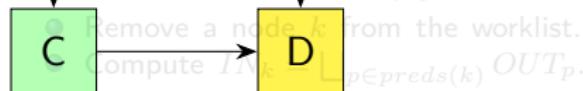


- Remove C from worklist.

1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node k from the worklist.

Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

Worklist: {D}

Worklist animation: Step 4

Background

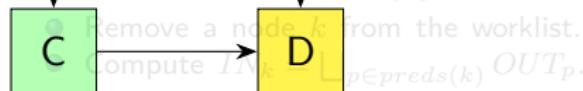


- Remove C from worklist.
- Compute $IN_C = OUT_A$.

1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

Worklist: {D}

Worklist animation: Step 4

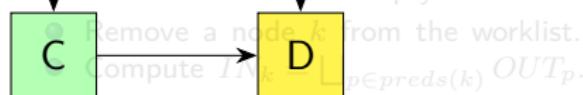
Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

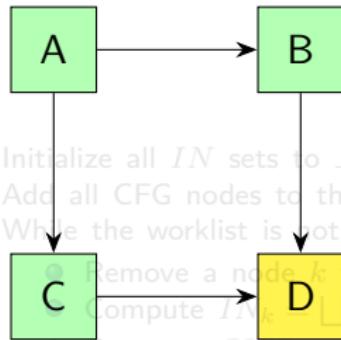
4: This continues until no more changes occur (fixpoint reached).

- Remove C from worklist.
- Compute $IN_C = OUT_A$.
- Compute $OUT_C = f_C(IN_C)$.

Worklist: {D}

Worklist animation: Step 4

Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:

- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.
- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

- Remove C from worklist.
- Compute $IN_C = OUT_A$.
- Compute $OUT_C = f_C(IN_C)$.
- OUT_C changed, add D to worklist (already there).

Worklist: {D}

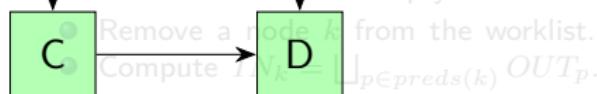
Worklist animation: Step 5

Background



- Remove D from worklist.

- 1: Initialize all IN sets to \perp .
- 2: Add all CFG nodes to the worklist.
- 3: While the worklist is not empty:



Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

- 4: This continues until no more changes occur (fixpoint reached).

Worklist: {}

Worklist animation: Step 5

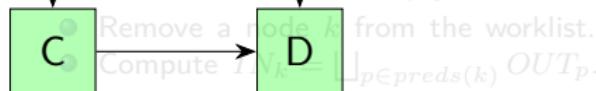
Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

- Remove D from worklist.
- Compute $IN_D = OUT_B \sqcup OUT_C$.

Worklist: {}

Worklist animation: Step 5

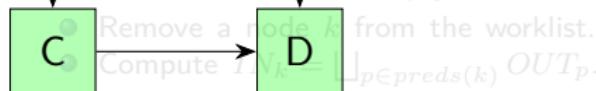
Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Remove a node k from the worklist.
- Compute $IN_k = \bigcup_{p \in \text{preds}(k)} OUT_p$.

- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

- Remove D from worklist.
- Compute $IN_D = OUT_B \sqcup OUT_C$.
- Compute $OUT_D = f_D(IN_D)$.

Worklist: {}

Worklist animation: Step 5

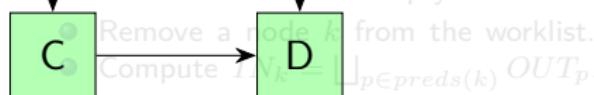
Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to the worklist.

4: This continues until no more changes occur (fixpoint reached).

- Remove D from worklist.
- Compute $IN_D = OUT_B \sqcup OUT_C$.
- Compute $OUT_D = f_D(IN_D)$.
- OUT_D changed, add successors of D to worklist (none).

Worklist: {}

Worklist animation: Step 5

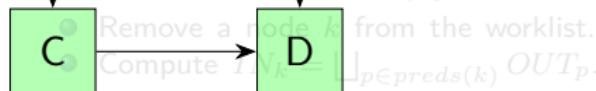
Background



1: Initialize all IN sets to \perp .

2: Add all CFG nodes to the worklist.

3: While the worklist is not empty:



- Compute $OUT_k = f_k(IN_k)$.

- If OUT_k changed, add successors of k to worklist (none).

4: This continues until no more changes occur (fixpoint reached).

- Remove D from worklist.
- Compute $IN_D = OUT_B \sqcup OUT_C$.
- Compute $OUT_D = f_D(IN_D)$.
- OUT_D changed, add successors of D to worklist (none).
- Worklist is empty, fixpoint reached.

Worklist: {}

Vertex-centric graph processing

Background

To distribute the analysis, the paper adopts the **vertex-centric model**.

The **GAS** model:

- **Gather**: A vertex collects data (messages) from neighbors (predecessors in CFG).
- **Apply**: The vertex updates its internal state (Dataflow fact) using the transfer function.
- **Scatter**: If the state changes, the vertex activates its neighbors (successors) and sends new messages.

Mapping:

- Graph vertices → CFG basic blocks/statements.
- Vertex value → Abstract state (*IN* and *OUT* sets).

Vertex-centric graph processing algorithm

Background

Algorithm 1 The vertex-centric GAS model algorithm.

```
1: Data:  $\mathcal{A}$ , the set of active vertices during processing
2: repeat
3:   for all each vertex  $k \in \mathcal{A}$  in parallel do                                ▷ done by system
4:     Remove  $k$  from  $\mathcal{A}$                                               ▷ done by system
5:     // Perform user-specified logic for each vertex
6:      $\mathcal{M}_k \leftarrow \text{Gather}(k)$           ▷ gather messages or information from neighbors
7:      $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$     ▷ update value of  $k$  based on gathered information
8:      $\langle \mathcal{M}, \mathcal{A}' \rangle \leftarrow \text{Scatter}(\mathcal{D}_k, k)$   ▷ activate new vertices and/or send out messages
9:   end for                                                               ▷ synchronize before next superstep
10:  SYNCHRONIZE()                                         ▷ done by system
11:   $\mathcal{A} \leftarrow \mathcal{A}'$                                          ▷ done by system
12: until  $\mathcal{A} = \emptyset$ 
```

GAS animation: superstep 1

Background

```
1: repeat
    for all each vertex  $k \in \mathcal{A}$  in parallel do
        Remove( $k$ ) from  $\mathcal{A}$ 
        // Perform user-specified logic for each vertex
         $\mathcal{M}_k \leftarrow \text{Gather}(k)$ 
         $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$ 
         $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$ 
    end for
    SYNCHRONIZE()
     $\mathcal{A} \leftarrow \mathcal{A}'$ 
11: until  $\mathcal{A} = \emptyset$ 
```

- Initial state: Vertex A is active.

▷ done by system
▷ done by system

▷ gather messages or information from neighbors
▷ update value of k based on gathered information
▷ activate new vertices and/or send out messages

▷ synchronize before next superstep
▷ done by system
▷ done by system

GAS animation: superstep 1

Background

```
1: repeat
    for all each vertex  $k \in \mathcal{A}$  in parallel do
        Remove( $k$ ) from  $\mathcal{A}$ 
        // Perform user-specified logic for each vertex
         $\mathcal{M}_k \leftarrow \text{Gather}(k)$ 
         $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$ 
         $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$ 
    end for
    SYNCHRONIZE()
     $\mathcal{A} \leftarrow \mathcal{A}'$ 
11: until  $\mathcal{A} = \emptyset$ 
```

- Initial state: Vertex A is active.

- A Gathers from neighbors
 - gather messages or information from neighbors

- update value (none). based on gathered information

- activate new vertices and/or send out messages

- synchronize before next superstep

- done by system

- done by system

GAS animation: superstep 1

Background

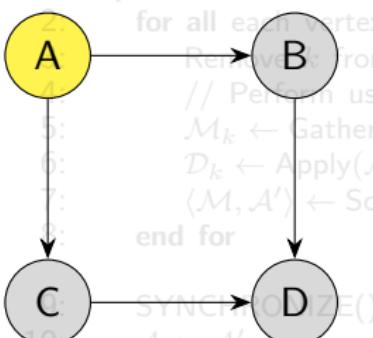
```
1: repeat
    for all each vertex  $k \in \mathcal{A}$  in parallel do
        Remove( $k$ ) from  $\mathcal{A}$ 
        // Perform user-specified logic for each vertex
         $\mathcal{M}_k \leftarrow \text{Gather}(k)$ 
         $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$ 
         $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$ 
    end for
    SYNCHRONIZE()
     $\mathcal{A} \leftarrow \mathcal{A}'$ 
11: until  $\mathcal{A} = \emptyset$ 
```

- Initial state: Vertex A is active.
- A Gathers from neighbors
 - gather messages or information from neighbors
 - update value (none) based on gathered information
- A Applies function, value changes.
 - Activate new vertices and/or send out messages
 - Synchronize before next superstep

GAS animation: superstep 1

Background

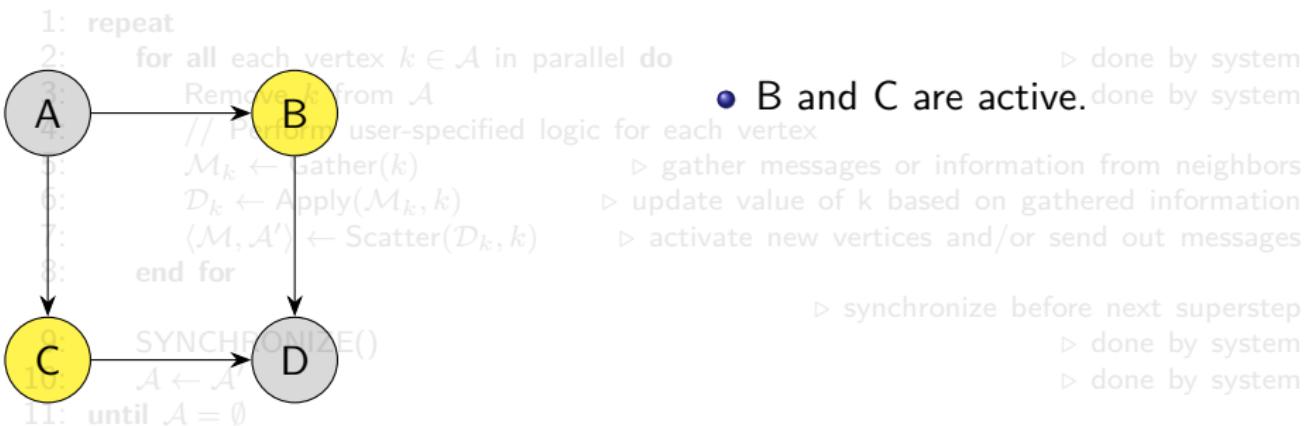
```
1: repeat
2:   for all each vertex  $k \in \mathcal{A}$  in parallel do
3:     Remove( $k$ ) from  $\mathcal{A}$ 
4:     // Perform user-specified logic for each vertex
5:      $\mathcal{M}_k \leftarrow \text{Gather}(k)$ 
6:      $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$ 
7:      $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$ 
8:   end for
9:   SYNCHRONIZE()
10:   $\mathcal{A} \leftarrow \mathcal{A}'$ 
11: until  $\mathcal{A} = \emptyset$ 
```



- Initial state: Vertex A is active.
- A Gathers from neighbors
 - ▷ gather messages or information from neighbors (none).
 - ▷ update value based on gathered information
- A Applies function, value changes.
 - ▷ synchronize before next superstep
- A Scatters to B and C, activating them.

GAS animation: superstep 2

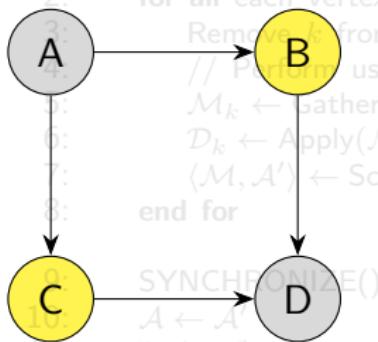
Background



GAS animation: superstep 2

Background

```
1: repeat
2:   for all each vertex  $k \in \mathcal{A}$  in parallel do
3:     Remove  $k$  from  $\mathcal{A}$ 
4:     // Perform user-specified logic for each vertex
5:      $\mathcal{M}_k \leftarrow \text{gather}(k)$ 
6:      $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$ 
7:      $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$ 
8:   end for
9:
10:   $\mathcal{A} \leftarrow \mathcal{A}'$ 
11: until  $\mathcal{A} = \emptyset$ 
```



- B and C are active.

▷ done by system

- B and C Gather from A.

▷ gather messages from neighbors

▷ update value of k based on gathered information

▷ activate new vertices and/or send out messages

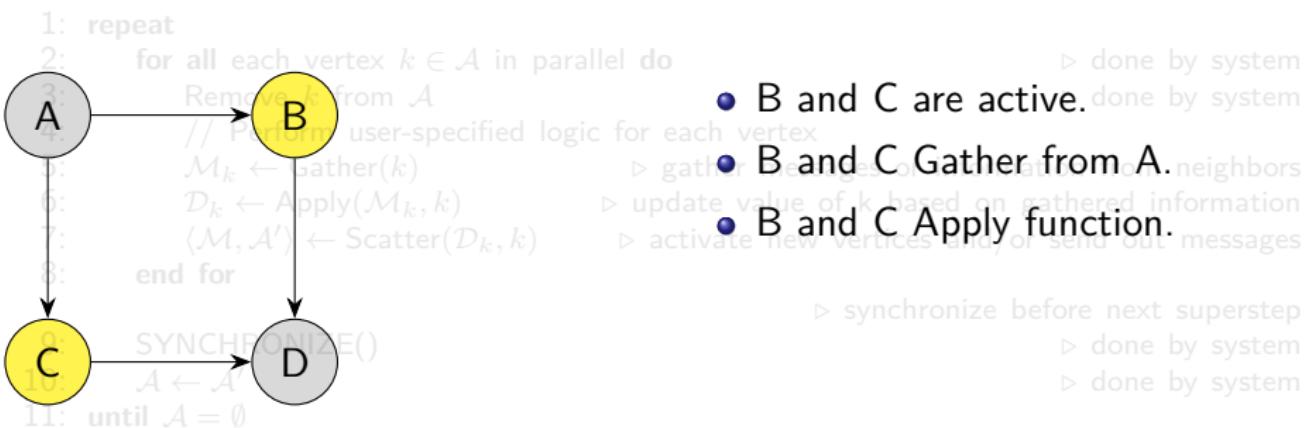
▷ synchronize before next superstep

▷ done by system

▷ done by system

GAS animation: superstep 2

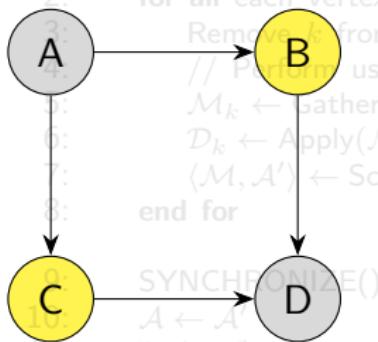
Background



GAS animation: superstep 2

Background

```
1: repeat
2:   for all each vertex  $k \in \mathcal{A}$  in parallel do
3:     Remove  $k$  from  $\mathcal{A}$ 
4:     // Perform user-specified logic for each vertex
5:      $\mathcal{M}_k \leftarrow \text{gather}(k)$ 
6:      $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$ 
7:      $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$ 
8:   end for
9:
10:  SYNCHRONIZE()
11: until  $\mathcal{A} = \emptyset$ 
```



- B and C are active. ▷ done by system
- B and C Gather from A. ▷ gather messages from neighbors
- B and C Apply function. ▷ update value of k based on gathered information
- B and C Scatter to D, activating it. ▷ synchronize before next superstep
- B and C Scatter to D, activating it. ▷ done by system
- B and C Scatter to D, activating it. ▷ done by system

GAS animation: superstep 3

Background

1: Data: \mathcal{A} , the set of active vertices during processing
2: repeat

3: for all each vertex $k \in \mathcal{A}$ in parallel do
4: Remove(k) from \mathcal{A}
5: // Perform user-specified logic for each vertex
6: $\mathcal{M}_k \leftarrow \text{Gather}(k)$
7: $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$
8: $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$
9: end for

10: SYNC()
11: $\mathcal{A} \leftarrow \mathcal{A}'$

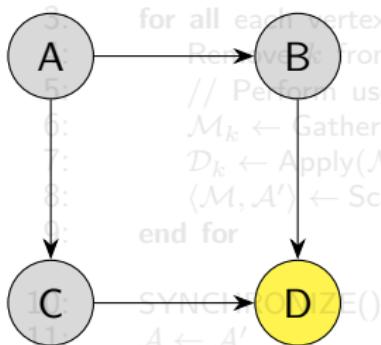
12: until $\mathcal{A} = \emptyset$

• D is active.

▷ done by system
▷ done by system

▷ gather messages or information from neighbors
▷ update value of k based on gathered information
▷ activate new vertices and/or send out messages

▷ synchronize before next superstep
▷ done by system
▷ done by system

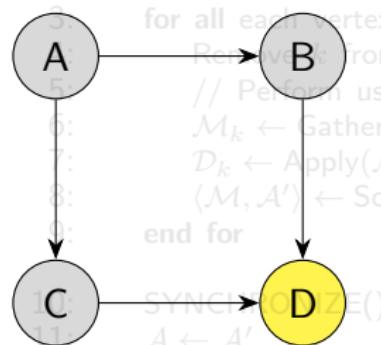


\mathcal{A}'

GAS animation: superstep 3

Background

1: Data: \mathcal{A} , the set of active vertices during processing
2: repeat



3: for all each vertex $k \in \mathcal{A}$ in parallel do
4: Remove(k) from \mathcal{A}
5: // Perform user-specified logic for each vertex
6: $\mathcal{M}_k \leftarrow \text{Gather}(k)$
7: $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$
8: $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$
9: end for

• D is active.

▷ done by system
▷ done by system

• D Gather from B and C.

▷ gather messages or information from neighbors
▷ update value of k based on gathered information
▷ activate new vertices and/or send out messages

▷ synchronize before next superstep
▷ done by system
▷ done by system

10: until $\mathcal{A} = \emptyset$

11:

12: SYNC(ODZE())

13: $\mathcal{A} \leftarrow \mathcal{A}'$

14:

15:

16:

17:

18:

19:

20:

21:

22:

23:

24:

25:

26:

27:

28:

29:

30:

31:

32:

33:

34:

35:

36:

37:

38:

39:

40:

41:

42:

43:

44:

45:

46:

47:

48:

49:

50:

51:

52:

53:

54:

55:

56:

57:

58:

59:

60:

61:

62:

63:

64:

65:

66:

67:

68:

69:

70:

71:

72:

73:

74:

75:

76:

77:

78:

79:

80:

81:

82:

83:

84:

85:

86:

87:

88:

89:

90:

91:

92:

93:

94:

95:

96:

97:

98:

99:

100:

101:

102:

103:

104:

105:

106:

107:

108:

109:

110:

111:

112:

113:

114:

115:

116:

117:

118:

119:

120:

121:

122:

123:

124:

125:

126:

127:

128:

129:

130:

131:

132:

133:

134:

135:

136:

137:

138:

139:

140:

141:

142:

143:

144:

145:

146:

147:

148:

149:

150:

151:

152:

153:

154:

155:

156:

157:

158:

159:

160:

161:

162:

163:

164:

165:

166:

167:

168:

169:

170:

171:

172:

173:

174:

175:

176:

177:

178:

179:

180:

181:

182:

183:

184:

185:

186:

187:

188:

189:

190:

191:

192:

193:

194:

195:

196:

197:

198:

199:

200:

201:

202:

203:

204:

205:

206:

207:

208:

209:

210:

211:

212:

213:

214:

215:

216:

217:

218:

219:

220:

221:

222:

223:

224:

225:

226:

227:

228:

229:

230:

231:

232:

233:

234:

235:

236:

237:

238:

239:

240:

241:

242:

243:

244:

245:

246:

247:

248:

249:

250:

251:

252:

253:

254:

255:

256:

257:

258:

259:

260:

261:

262:

263:

264:

265:

266:

267:

268:

269:

270:

271:

272:

273:

274:

275:

276:

277:

278:

279:

280:

281:

282:

283:

284:

285:

286:

287:

288:

289:

290:

291:

292:

293:

294:

295:

296:

297:

298:

299:

300:

301:

302:

303:

304:

305:

306:

307:

308:

309:

310:

311:

312:

313:

314:

315:

316:

317:

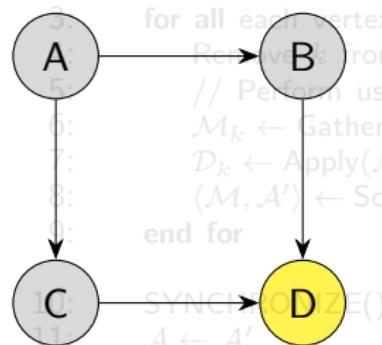
318:

319:

GAS animation: superstep 3

Background

1: Data: \mathcal{A} , the set of active vertices during processing
2: repeat



3: for all each vertex $k \in \mathcal{A}$ in parallel do
4: Remove k from \mathcal{A}
5: // Perform user-specified logic for each vertex
6: $\mathcal{M}_k \leftarrow \text{Gather}(k)$
7: $\mathcal{D}_k \leftarrow \text{Apply}(\mathcal{M}_k, k)$
8: $(\mathcal{M}, \mathcal{A}') \leftarrow \text{Scatter}(\mathcal{D}_k, k)$
9: end for

- D is active.

▷ done by system
▷ done by system

- D Gather from B and C.

▷ gather messages or information from neighbors
▷ update value of the base vertex with gathered information
▷ activate new vertices and/or send out messages

- D Apply function.

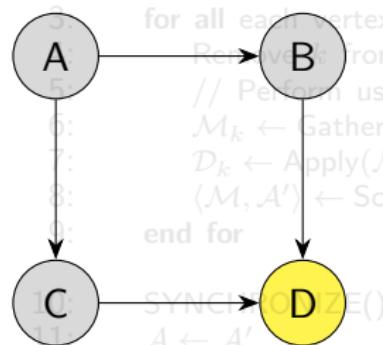
▷ synchronize before next superstep
▷ done by system
▷ done by system

10: until $\mathcal{A} = \emptyset$

GAS animation: superstep 3

Background

1: Data: \mathcal{A} , the set of active vertices during processing
2: repeat



- **D is active.**

▷ done by system
▷ done by system

- **D Gather from B and C.**

▷ gather messages or information from neighbors
▷ update value of the base vertex with gathered information

- **D Apply function.**

▷ activate new vertices and/or send out messages

- **D Scatter to nothing (no successors).**

▷ synchronize before next superstep
▷ done by system
▷ done by system

Proposed solutions

Methods

Inspired by the GAS model, the paper designs the distributed vertex-centric worklist algorithm.

The paper proposes 2 frameworks:

- BigDataflow.
- BigDataflow-incremental.

Each in a classic and an optimized version:

- BigDataflow-classic.
- BigDataflow.
- BigDataflow-incremental.
- BigDataflow-incremental optimized.

System implementation

Methods

BigDataflow is built on top of mature distributed systems to handle large-scale computation and state persistence.

Apache Giraph (Computation)

- **Role:** Executes the distributed worklist algorithms.
- Functionality:
 - **Partitions** the massive CFG across cluster nodes.
 - Manages **synchronization** (supersteps) and message passing between workers.

Redis (State storage)

- **Role:** Enabler for **incremental analysis**.
- Functionality:
 - Stores **whole-program analysis results**.
 - Allows low-latency **query** (for impact analysis) and **update** of facts.

BigDataflow-classic algorithm

Methods

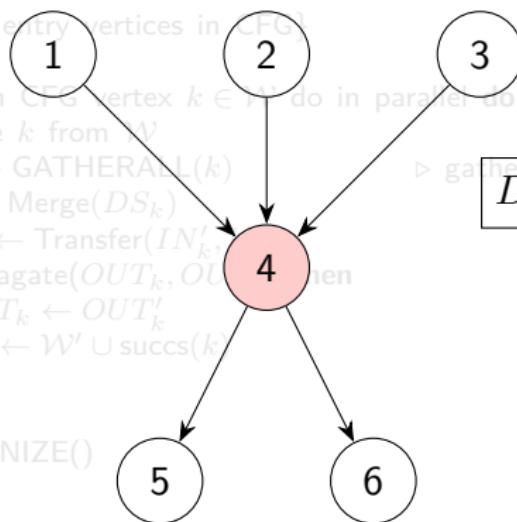
Algorithm 2 The BigDataflow-classic algorithm.

```
1: Data:  $\mathcal{W}$ , the list of all active vertices during analysis;  $DS_k : \{OUT_p | p \in preds(k)\}$  a set  
   containing all the dataflow facts of k's predecessors  
2:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$   
3: repeat  
4:   for all each CFG vertex  $k \in \mathcal{W}$  do in parallel do  
5:     Remove  $k$  from  $\mathcal{W}$   
6:      $DS_k \leftarrow \text{GATHERALL}(k)$                                  $\triangleright$  gather all the predecessors' dataflow facts  
7:      $IN'_k \leftarrow \text{Merge}(DS_k)$                                  $\triangleright$  merge  
8:      $OUT'_k \leftarrow \text{Transfer}(IN'_k, k)$                              $\triangleright$  transfer  
9:     if Propagate( $OUT_k, OUT'_k$ ) then                                 $\triangleright$  propagate  
10:     $OUT_k \leftarrow OUT'_k$   
11:     $\mathcal{W}' \leftarrow \mathcal{W}' \cup \text{succs}(k)$   
12:  end if  
13: end for  
14: SYNCHRONIZE()  
15:  $\mathcal{W} \leftarrow \mathcal{W}'$   
16: until  $\mathcal{W} = \emptyset$ 
```

BigDataflow-classic animation

Methods

```
1:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$ 
2: repeat
3:   for all each CFG vertex  $k \in \mathcal{W}$  do in parallel do
4:     Remove  $k$  from  $\mathcal{W}$ 
5:      $DS_k \leftarrow \text{GATHERALL}(k)$ 
6:      $IN'_k \leftarrow \text{Merge}(DS_k)$ 
7:      $OUT'_k \leftarrow \text{Transfer}(IN'_k, OUT_k)$ 
8:     if Propagate( $OUT_k, OUT'_k$ ) then
9:        $OUT_k \leftarrow OUT'_k$ 
10:       $\mathcal{W}' \leftarrow \mathcal{W}' \cup \text{succs}(k)$ 
11:    end if
12:  end for
13:  SYNCHRONIZE()
14:   $\mathcal{W} \leftarrow \mathcal{W}'$ 
15: until  $\mathcal{W} = \emptyset$ 
```



$DS_4 = \{OUT'_1, OUT'_2, OUT'_3\}$

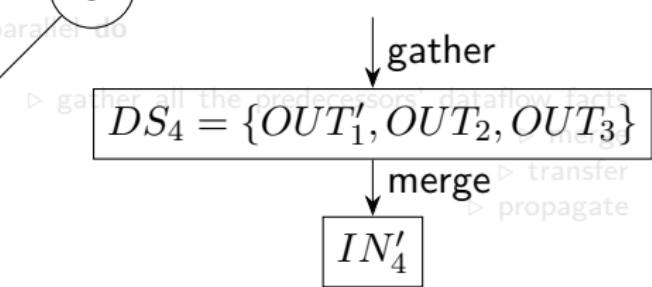
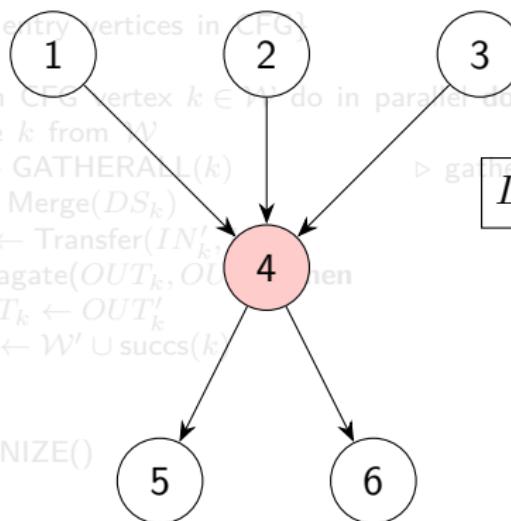
▷ gather all the predecessors dataflow facts
▷ merge

▷ transfer
▷ propagate

BigDataflow-classic animation

Methods

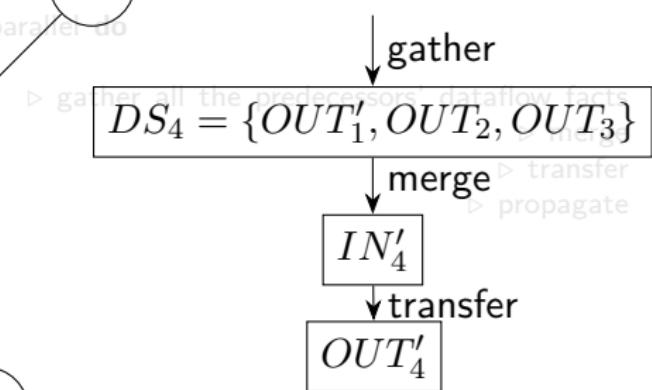
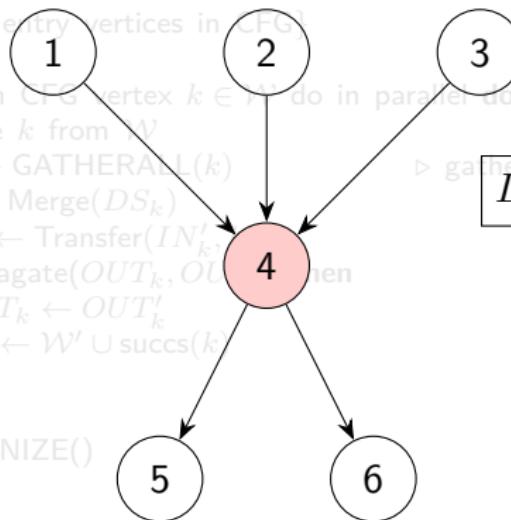
```
1:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$ 
2: repeat
3:   for all each CFG vertex  $k \in \mathcal{W}$  do in parallel do
4:     Remove  $k$  from  $\mathcal{W}$ 
5:      $DS_k \leftarrow \text{GATHERALL}(k)$                                 ▷ gather all the predecessors' dataflow facts
6:      $IN'_k \leftarrow \text{Merge}(DS_k)$ 
7:      $OUT'_k \leftarrow \text{Transfer}(IN'_k, OUT_k)$ 
8:     if Propagate( $OUT_k, OUT'_k$ ) then
9:        $OUT_k \leftarrow OUT'_k$ 
10:       $\mathcal{W}' \leftarrow \mathcal{W}' \cup \text{succs}(k)$ 
11:    end if
12:  end for
13:  SYNCHRONIZE()
14:   $\mathcal{W} \leftarrow \mathcal{W}'$ 
15: until  $\mathcal{W} = \emptyset$ 
```



BigDataflow-classic animation

Methods

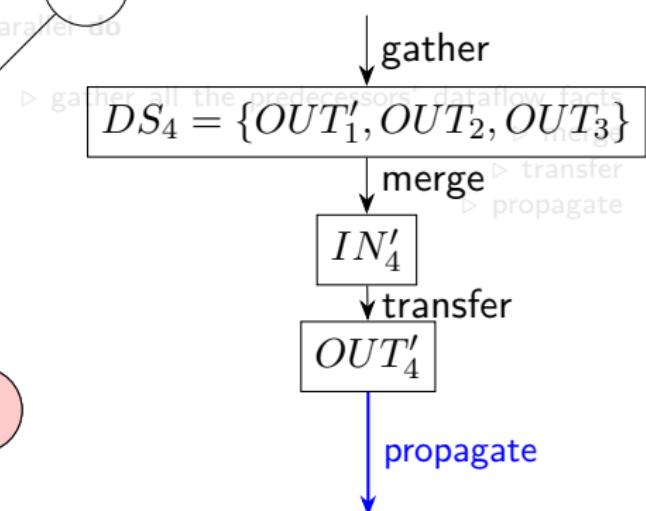
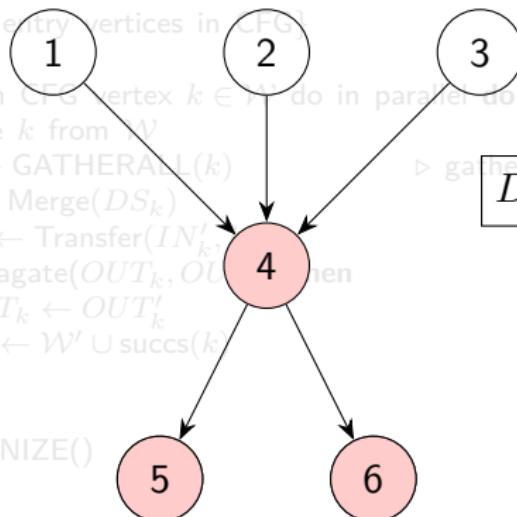
```
1:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$ 
2: repeat
3:   for all each CFG vertex  $k \in \mathcal{W}$  do in parallel do
4:     Remove  $k$  from  $\mathcal{W}$ 
5:      $DS_k \leftarrow \text{GATHERALL}(k)$                                  $\triangleright$  gather all the predecessors' dataflow facts
6:      $IN'_k \leftarrow \text{Merge}(DS_k)$ 
7:      $OUT'_k \leftarrow \text{Transfer}(IN'_k, OUT_k)$ 
8:     if Propagate( $OUT_k, OUT'_k$ ) then
9:        $OUT_k \leftarrow OUT'_k$ 
10:       $\mathcal{W}' \leftarrow \mathcal{W}' \cup \text{succs}(k)$ 
11:    end if
12:  end for
13:  SYNCHRONIZE()
14:   $\mathcal{W} \leftarrow \mathcal{W}'$ 
15: until  $\mathcal{W} = \emptyset$ 
```



BigDataflow-classic animation

Methods

```
1:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$ 
2: repeat
3:   for all each CFG vertex  $k \in \mathcal{W}$  do in parallel do
4:     Remove  $k$  from  $\mathcal{W}$ 
5:      $DS_k \leftarrow \text{GATHERALL}(k)$ 
6:      $IN'_k \leftarrow \text{Merge}(DS_k)$ 
7:      $OUT'_k \leftarrow \text{Transfer}(IN'_k)$ 
8:     if Propagate( $OUT_k, OUT'_k$ ) then
9:        $OUT_k \leftarrow OUT'_k$ 
10:       $\mathcal{W}' \leftarrow \mathcal{W}' \cup \text{succs}(k)$ 
11:    end if
12:  end for
13:  SYNCHRONIZE()
14:   $\mathcal{W} \leftarrow \mathcal{W}'$ 
15: until  $\mathcal{W} = \emptyset$ 
```



BigDataflow-classic limitations

Methods

```
1:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$ 
2: repeat
3:   ...
4:   Despite that the algorithm succeeds in leveraging large-
5:     scale distributed computing resources to accelerate dataflow facts
6:     analysis, it suffers from poor scalability.
7:   ...
8:   ...
9:   Each vertex gathers all predecessor facts in ev-
10:    ery superstep, leading to redundant data trans-
11:      mission and high memory consumption.
12:    ...
13:    Synchronize
14:     $\mathcal{W} \leftarrow \mathcal{W}'$ 
15: until  $\mathcal{W} = \emptyset$ 
```

dataflow facts
merge
transfer
propagate

BigDataflow optimized

Methods

The optimized algorithm gathers only the **updated** dataflow facts from predecessors in the previous superstep.

This approach is achieved via a push-based message passing mechanism

BigDataflow optimized algorithm

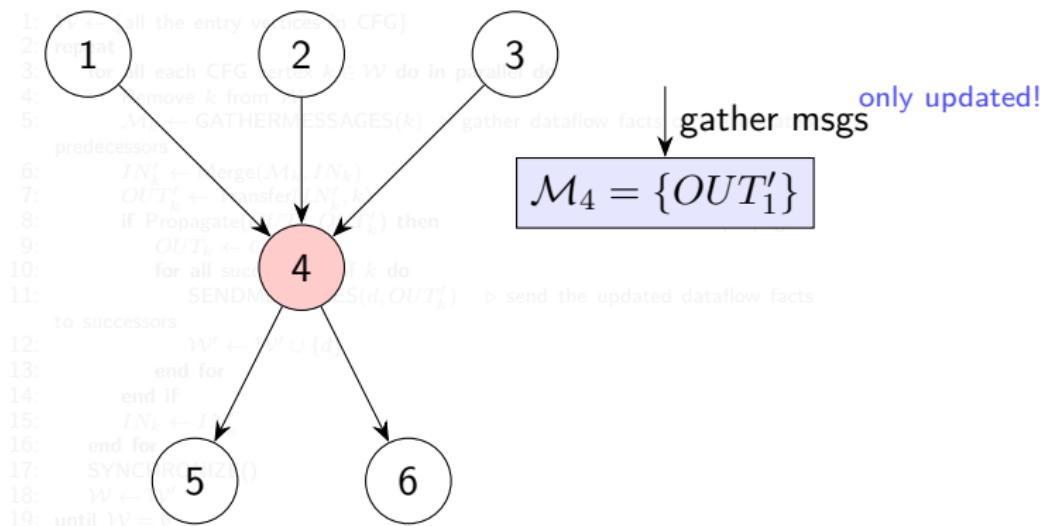
Methods

Algorithm 3 The BigDataflow optimized algorithm.

```
1:  $\mathcal{W} \leftarrow \{\text{all the entry vertices in CFG}\}$ 
2: repeat
3:   for all each CFG vertex  $k \in \mathcal{W}$  do in parallel do
4:     Remove  $k$  from  $\mathcal{W}$ 
5:      $\mathcal{M}_k \leftarrow \text{GATHERMESSAGES}(k)$                                  $\triangleright$  gather only updated dataflow facts
6:      $IN'_k \leftarrow \text{Merge}(\mathcal{M}_k, IN_k)$                              $\triangleright$  merge
7:      $OUT'_k \leftarrow \text{Transfer}(IN'_k, k)$                             $\triangleright$  transfer
8:     if Propagate( $OUT_k, OUT'_k$ ) then                                 $\triangleright$  propagate
9:        $OUT_k \leftarrow OUT'_k$ 
10:      for all successor  $d$  of  $k$  do
11:         $\text{SENDMESSAGES}(d, OUT'_k)$                                       $\triangleright$  send the updated dataflow facts
12:         $\mathcal{W}' \leftarrow \mathcal{W}' \cup \{d\}$ 
13:      end for
14:    end if
15:     $IN_k \leftarrow IN'_k$ 
16:  end for
17:   $\text{SYNCHRONIZE}()$ 
18:   $\mathcal{W} \leftarrow \mathcal{W}'$ 
19: until  $\mathcal{W} = \emptyset$ 
```

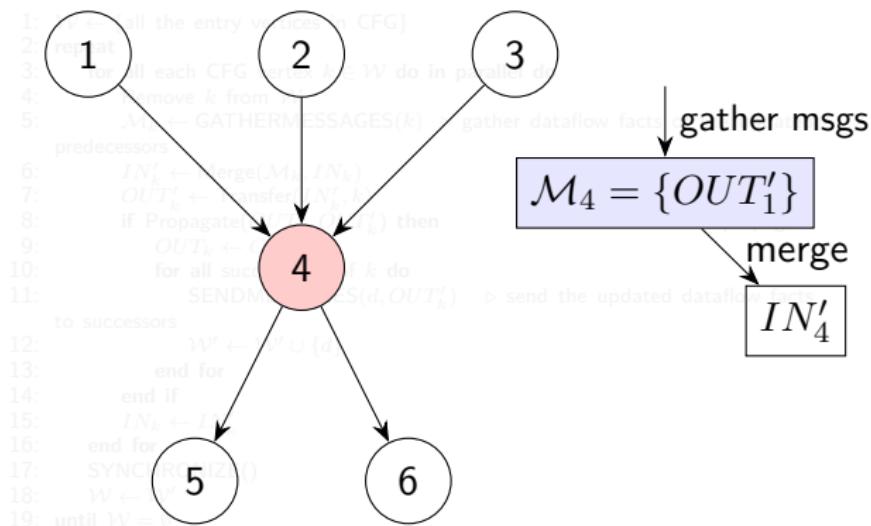
BigDataflow optimized animation

Methods



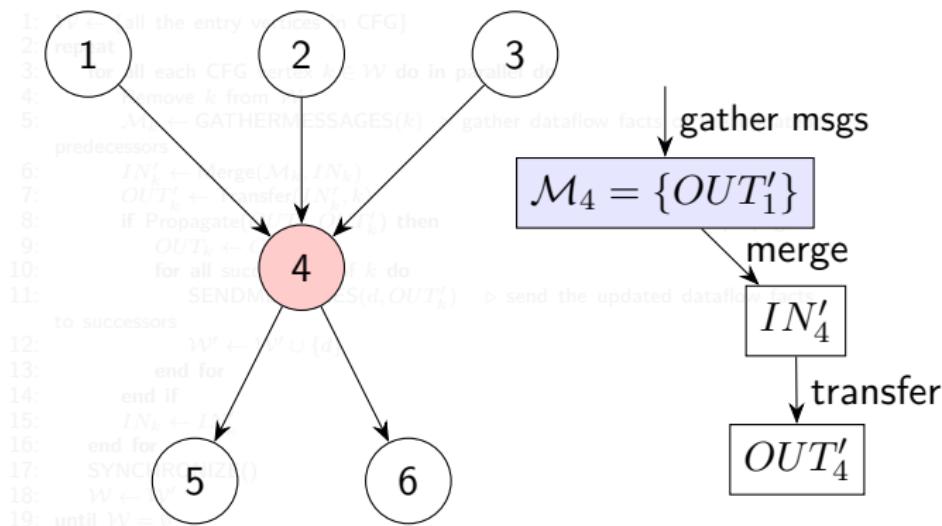
BigDataflow optimized animation

Methods



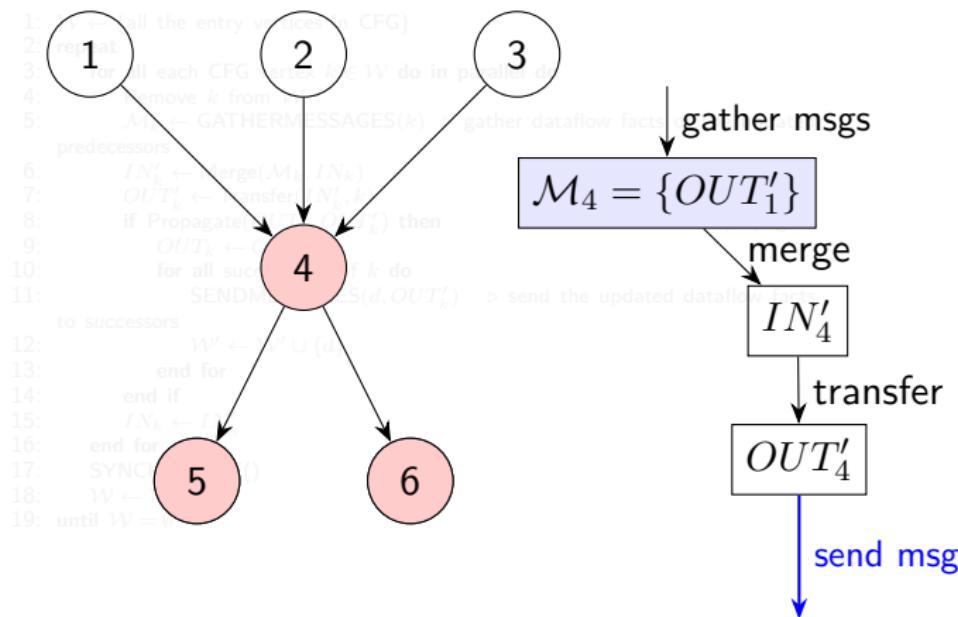
BigDataflow optimized animation

Methods



BigDataflow optimized animation

Methods



BigDataflow-incremental

Methods

Real-world software changes frequently, especially in open-source projects.
Re-analyzing from scratch is time-consuming and expensive.

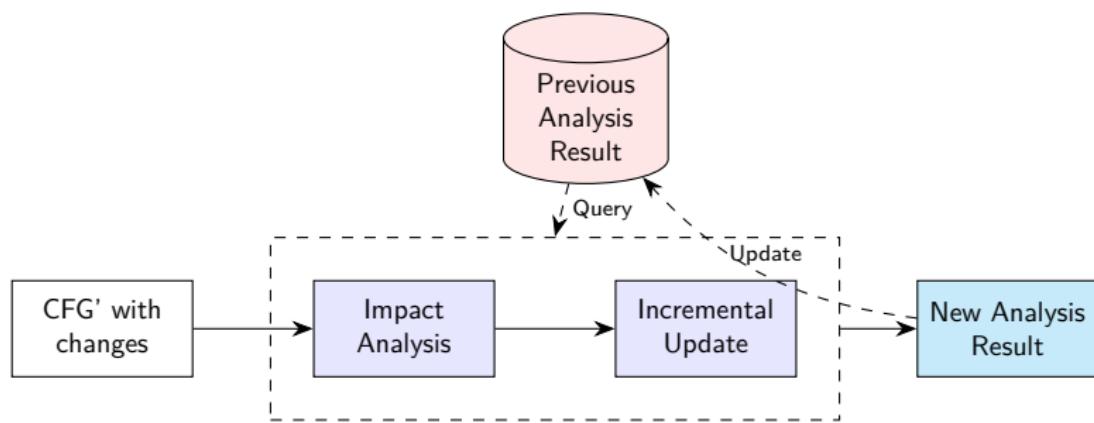


Figure: Workflow of Distributed Incremental Dataflow Analysis.

Workflow of BigDataflow-incremental Methods

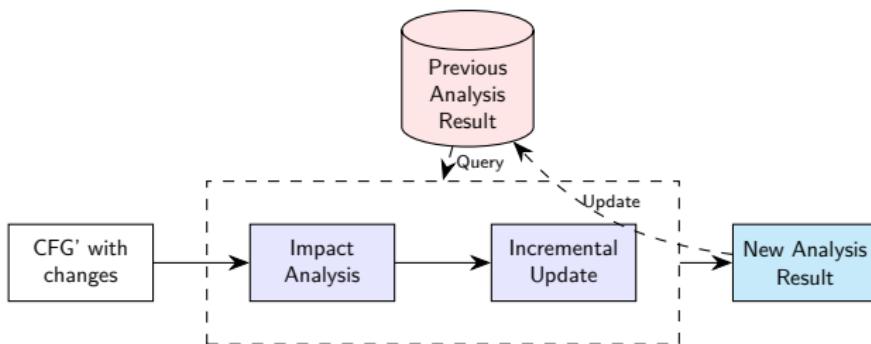


Figure: Workflow of Distributed Incremental Dataflow Analysis.

Given a **CFG** with a batch of updates, the framework:

- Performs **impact analysis** to identify affected nodes.
- Performs **incremental update**:
 - Identification of affected nodes.
 - Extraction of affected sub-CFGs.
 - Re-analyzing only the affected sub-CFGs.
- Produces the **new analysis result**.

BigDataflow-incremental

Methods

The distributed incremental analysis reuses the existing analysis result and only performs local analysis on the updated parts to achieve efficiency.

Atomic Changes: The framework handles three categories of changes on the CFG:

- **Addition:**

- Added edge between existing nodes.
- Added source node and edge.
- Added destination node.

- **Change:**

- Changed source node (updates transfer function).
- Changed destination node.

- **Deletion:**

- Deleted edge.
- Deleted source/destination node.

These atomic changes trigger the **Impact Analysis** to identify the initial set of affected nodes.

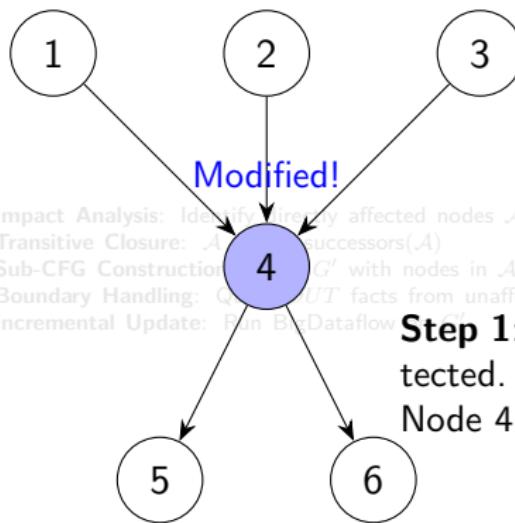
BigDataflow-incremental algorithm steps

Methods

- 1: **Impact Analysis:** Identify directly affected nodes \mathcal{A} based on atomic changes.
- 2: **Transitive Closure:** $\mathcal{A} \leftarrow \mathcal{A} \cup \text{successors}(\mathcal{A})$.
- 3: **Sub-CFG Construction:** Build G' with nodes in \mathcal{A} and connecting edges.
- 4: **Initialization:**
 - 5: For $k \in \mathcal{A}_{\text{add_only}}$: reuse previous results (Optimized).
 - 6: For others: reset to \perp or \top .
- 7: **Boundary Handling:** Query OUT facts from unaffected predecessors.
- 8: **Incremental Update:** Run BigDataflow on G' .

BigDataflow-incremental algorithm

Methods

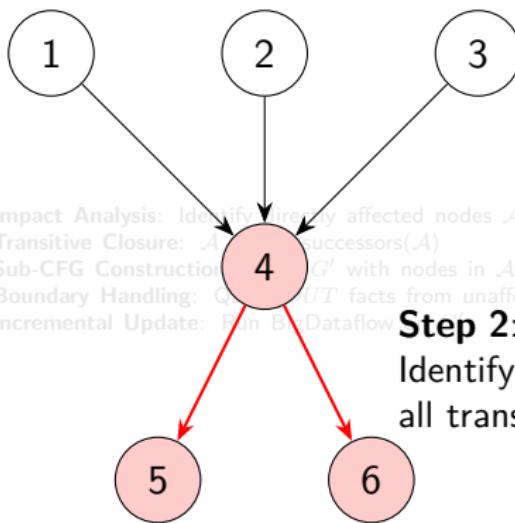


- 1: Impact Analysis: Identify directly affected nodes \mathcal{A}
- 2: Transitive Closure: $\mathcal{A} \leftarrow \text{successors}(\mathcal{A})$
- 3: Sub-CFG Construction: Build G' with nodes in \mathcal{A}
- 4: Boundary Handling: Query G' for UT facts from unaffected predecessors
- 5: Incremental Update: Run BigDataflow on G'

Step 1: Code change detected.
Node 4 is modified

BigDataflow-incremental algorithm

Methods

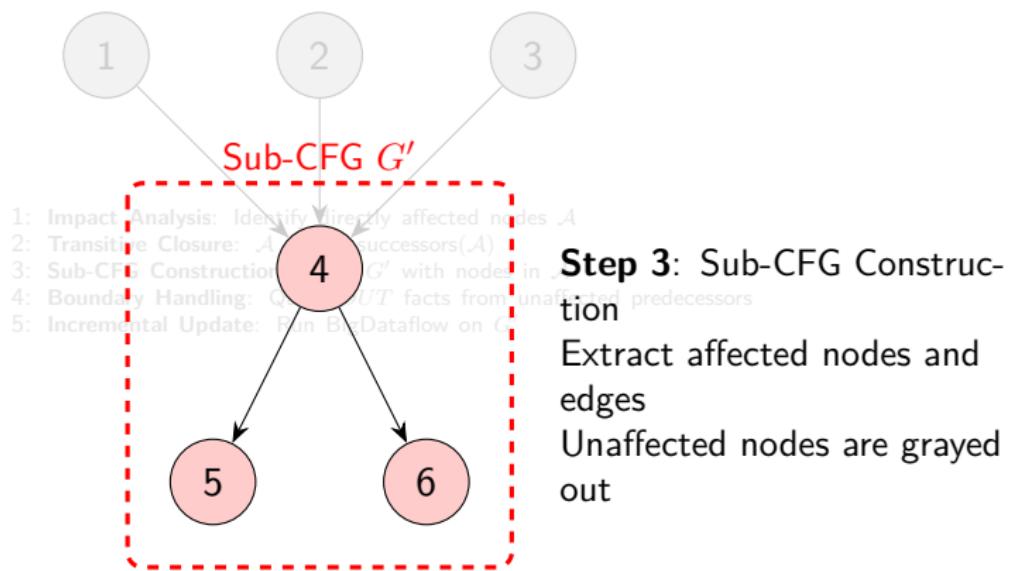


- 1: Impact Analysis: Identify directly affected nodes \mathcal{A}
- 2: Transitive Closure: $\mathcal{A} \leftarrow \text{transitive_closure}(\mathcal{A})$
- 3: Sub-CFG Construction: Build a Sub-CFG G' with nodes in \mathcal{A}
- 4: Boundary Handling: Query G' for UT facts from unaffected predecessors
- 5: Incremental Update: Run BigDataflow on G'

Step 2: Impact Analysis
Identify affected node (4) and all transitive successors (5, 6)

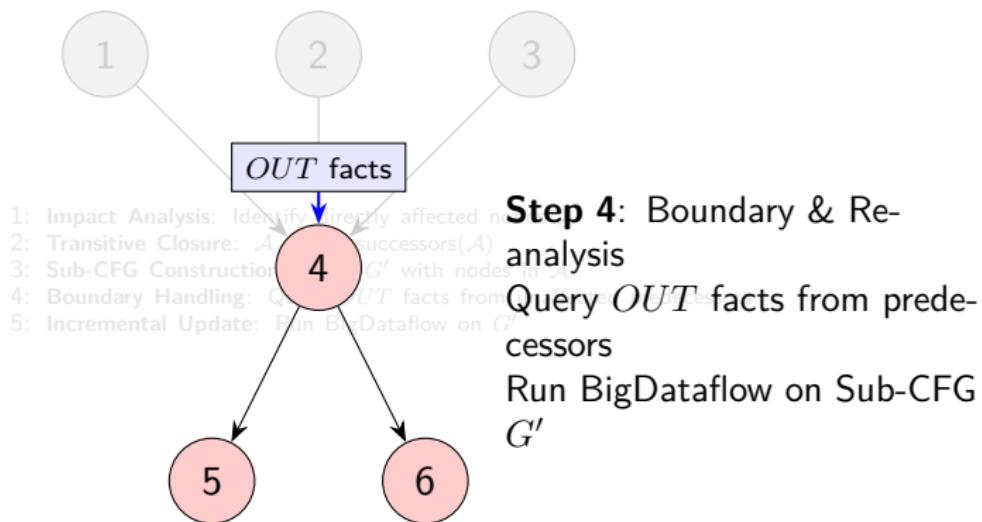
BigDataflow-incremental algorithm

Methods



BigDataflow-incremental algorithm

Methods



The evaluation aims to answer the following questions:

- Q1: What is the overall performance of BigDataflow given a rich set of distributed computing resources?
- Q2: How does BigDataflow perform compared with other competitive analysis systems/tools?
- Q3: What about the performance of BigDataflow given the varying numbers of cores and resources?
- Q4: How about the performance of BigDataflow in the mode of incremental analysis?

Evaluation subjects

Results

Table: Subjects for evaluation

Subject	Version	#LoC	#Functions	Description
Linux	5.2	17.5M	565K	operating system
Firefox	67.0	7.9M	770K	web browser
PostgreSQL	12.2	1.0M	30K	database system
OpenSSL	1.1.1	519K	12K	TLS protocol
Httpd	2.4.39	196K	6K	web server

Where #LoC is the number of lines of code and #Functions is the number of functions.

Reference tools and hardware

Results

Table: Tools and hardware

Tool	Hardware	Description
BigDataflow-classic (Baseline)	Cluster: 125 nodes (8 vCPUs, 64GB RAM)	Classic distributed algorithm with full predecessor gathering
BigDataflow (Proposed)	Cluster: 125 nodes (8 vCPUs, 64GB RAM)	Optimized with delta updates and message passing
Chianina (State-of-the-Art)	Single server (104 vCPUs, 768GB RAM, 1TB SSD)	Parallel single-machine system with out-of-core support

Evaluation analysis

Results

To test the performance and scalability of BigDataflow, two notoriously “heavy” dataflow analyses were chosen:

Context-sensitive alias analysis

Determines which pointers can point to the same memory location, considering the calling context of functions. It is computationally expensive because tracking every possible call path leads to an exponential explosion in the state space, resulting in massive memory consumption and long computation times.

Instruction cache analysis

Predicts whether a machine instruction is likely to be in the CPU's instruction cache. It is memory-intensive as it requires modeling the cache behavior for all possible execution paths. The number of states can be enormous for large programs with complex control flow.

Alias analysis

Results

Subject	Tool	#PAliases	#Workers/#Part	#PMem	Time	Cost (\$)
Linux	BigDataflow	12.5B	350	3.5T	16.7m	11.1
	BigDataflow-classic	-	350	x	x	x
	Chianina	-	4	453.4G	17.4h	110.4
Firefox	BigDataflow	11.5B	140	1.2T	16.5m	4.4
	BigDataflow-classic	-	140	x	x	x
	Chianina	-	4	131.6G	5.3h	33.6
PostgreSQL	BigDataflow	727.0M	50	329.7G	2.8m	0.3
	BigDataflow-classic	-	50	330.7G	4.9m	0.5
	Chianina	-	1	61.9G	50.4m	5.3
OpenSSL	BigDataflow	734.8M	30	285.3G	3.5m	0.2
	BigDataflow-classic	-	30	329.8G	6.8m	0.4
	Chianina	-	1	43.2G	35.4m	3.7
Httpd	BigDataflow	183.1M	10	119.9G	2.8m	0.1
	BigDataflow-classic	-	10	137.9G	4.0m	0.1
	Chianina	-	1	14.2G	11.2m	1.2

Where #PAliases is the number of pointer alias pairs discovered.

Instruction cache analysis

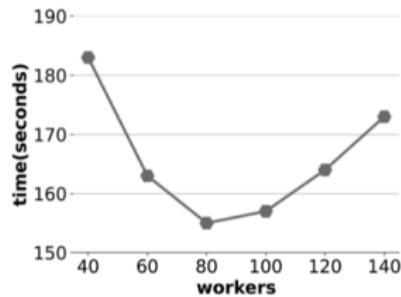
Results

Subject	Tool	#BCached	#Workers/#Part	#PMem	Time	Cost (\$)
Linux	BigDataflow	21.5B	500	5.6T	44.4m	42.0
	BigDataflow-classic	-	500	x	x	x
	Chianina	-	4	555.4G	9.4h	59.6
Firefox	BigDataflow	15.8B	400	4.4T	39.0m	29.5
	BigDataflow-classic	-	400	x	x	x
	Chianina	-	4	351.5G	7.2h	45.7
PostgreSQL	BigDataflow	1.4B	180	1.1T	3.2m	1.1
	BigDataflow-classic	-	180	1.1T	6.5m	2.2
	Chianina	-	1	115.3G	38.1m	4.0
OpenSSL	BigDataflow	2.8B	180	1.3T	6.9m	2.3
	BigDataflow-classic	-	180	1.5T	13.4m	4.6
	Chianina	-	1	227.6G	1.7h	10.8
Httpd	BigDataflow	782.0M	100	684.3G	3.0m	0.6
	BigDataflow-classic	-	100	781.3G	4.6m	0.9
	Chianina	-	1	58.3G	18.5m	2.0

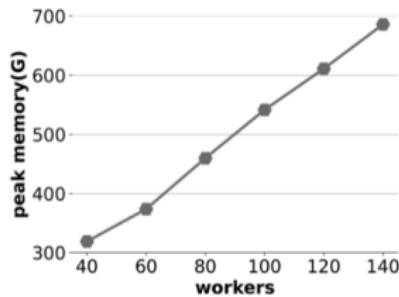
Where #BCached is the number of blocks that may be cached.

Scalability

Results



(a) time



(b) peak memory

Figure: Scalability of BigDataflow on OpenSSL subject for alias analysis.

Scalability

Results

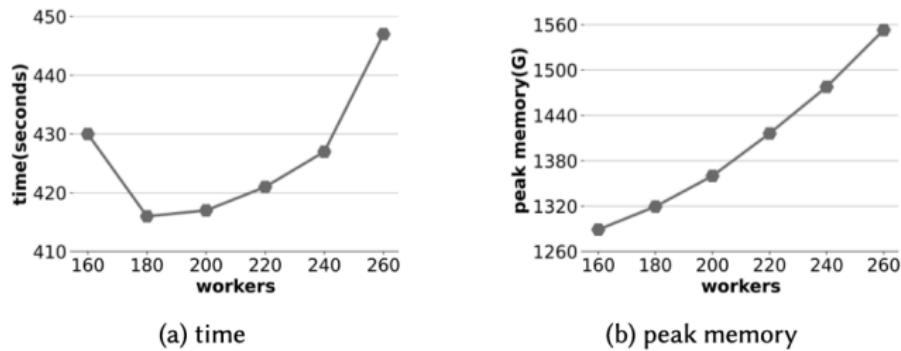


Figure: Scalability of BigDataflow on OpenSSL subject for cache analysis.

Incremental analysis: Httpd-3days

Results

Subject	BigDataflow-whole				BigDataflow-incremental			
	Cost	Time	#Workers	#PMem	Cost	Time	#Workers	#PMem
NU10	\$0.643	3.4mins	100	626.5G	\$0.038	1.0mins	20	90.6G
NU9	/	/	/	/	/	/	/	/
NU8	\$0.662	3.5mins	100	623.2G	\$0.129	1.7mins	40	206.2G
NU7	/	/	/	/	/	/	/	/
NU6	\$0.681	3.6mins	100	626.5G	\$0.030	0.8mins	20	75.6G
NU5	/	/	/	/	/	/	/	/
NU4	/	/	/	/	/	/	/	/
NU3	/	/	/	/	/	/	/	/
NU2	/	/	/	/	/	/	/	/
NU1	/	/	/	/	/	/	/	/

Where NU1 to NU10 are 10 consecutive updates to the Httpd codebase over 3 days.

Conclusion

BigDataflow addresses the scalability challenge of interprocedural dataflow analysis by leveraging cloud resources.

This work presents three key contributions:

- **Distributed Framework:** Adopts a vertex-centric graph model to distribute heavy computations across cluster nodes.
- **Algorithmic Innovation:**
 - Optimized worklist: Prunes redundant data transmission via delta updates.
 - Incremental analysis: Efficiently handles code updates, reducing cost and time by orders of magnitude.
- **Usability:** Provides high-level APIs, allowing users to implement complex client analyses (e.g., Alias, Cache) easily.

BigDataflow transforms “intractable” analysis tasks (e.g., analyzing 17.5M LoC Linux Kernel) from hours to minutes, making deep software verification more viable for modern CI/CD pipelines.

Thank you for your attention