Elixir

A System for Synthesizing Concurrent Graph Programs

Prountzos D., Manevich R. & Pingali K.

Christopher Little

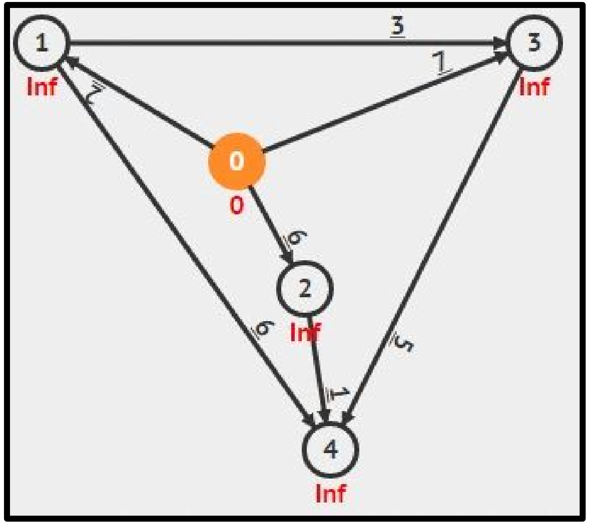
Motivation

Best solution to problems depends on:

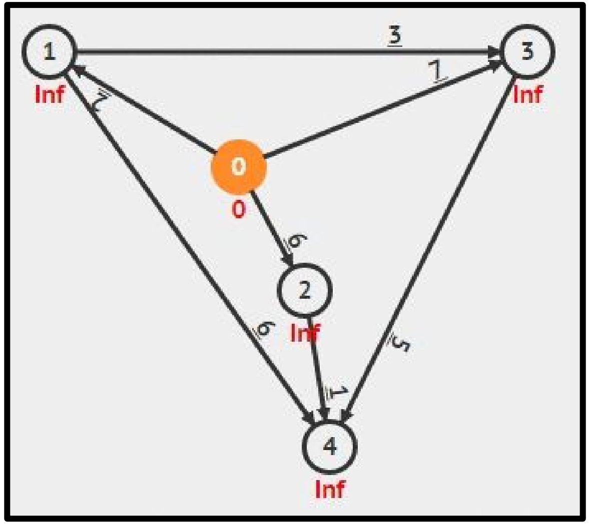
- Data
- Machine Architecture
- Intra-algorithm tuning
- …

Dream: let the compiler worry about it all

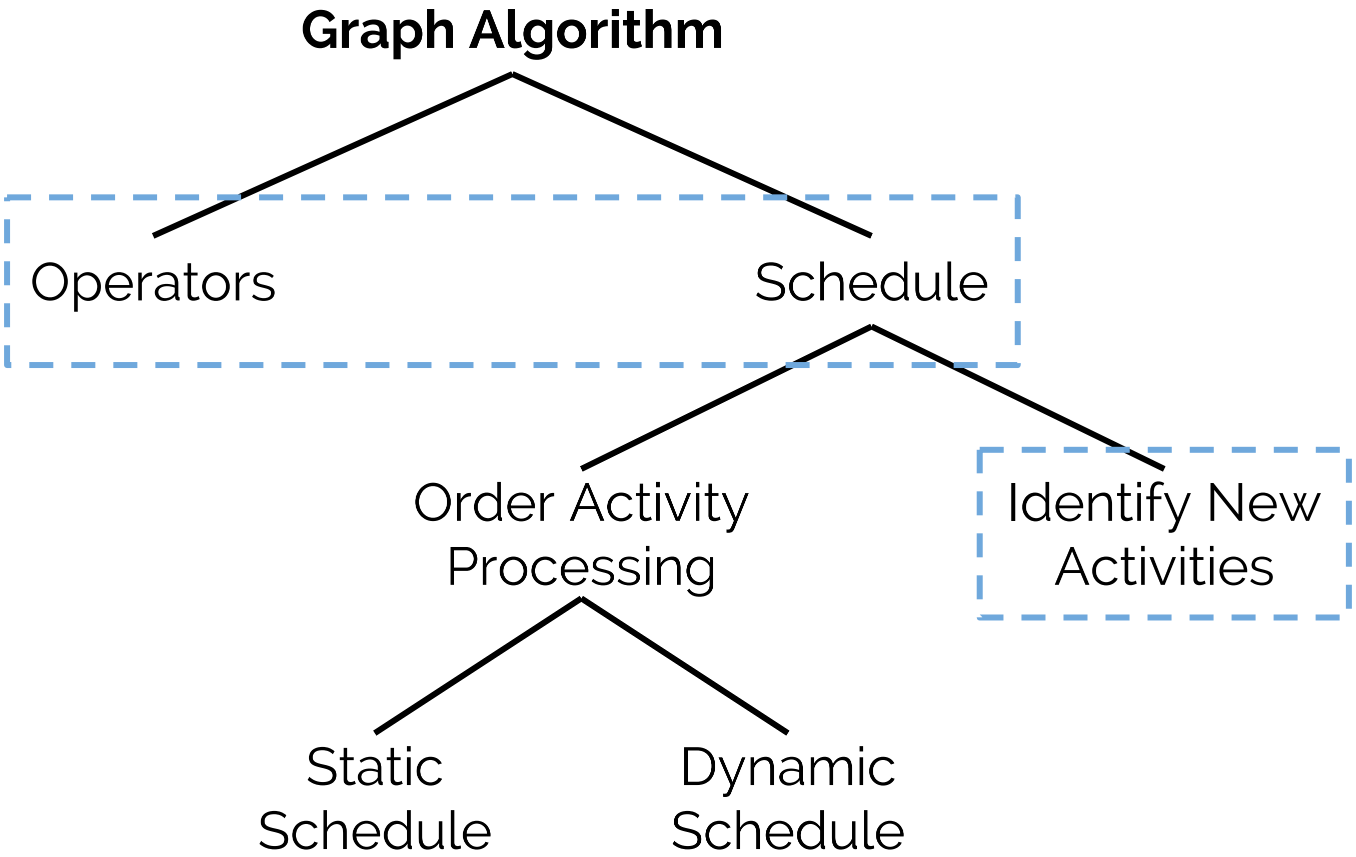
Running Example: SSSP
(Single-Source Shortest Path)



Dijkstra



Bellman-Ford


Graph Algorithm
Operators
Schedule
Order Activity Processing
Identify New Activities
Static Schedule
Dynamic Schedule

SSSP Elixir Specification

```
Graph [
     nodes(node: Node, dist: int)
     edges(src: Node, dest: Node, wt: int)
]

relax = [ nodes(node a, dist ad)
          nodes(node b, dist bd)
          edges(src a, dest b, wt w)
          bd > ad + w ] ->
          [bd = ad + w]

sssp = iterate relax >> schedule
```

Graph Type Definition

Operator Definition

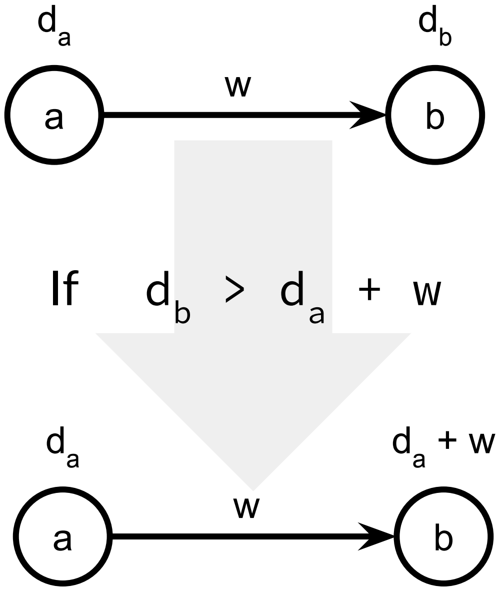
Fixpoint Statement

SSSP Elixir Specification

```
Graph [
     nodes(node: Node, dist: int)
     edges(src: Node, dest: Node, wt: int)
]

relax = [ nodes(node a, dist ad)
          nodes(node b, dist bd)               Redex Pattern
          edges(src a, dest b, wt w)
          bd > ad + w ] ->                     Guard
          [bd = ad + w]                        Update

sssp = iterate relax >> schedule
```

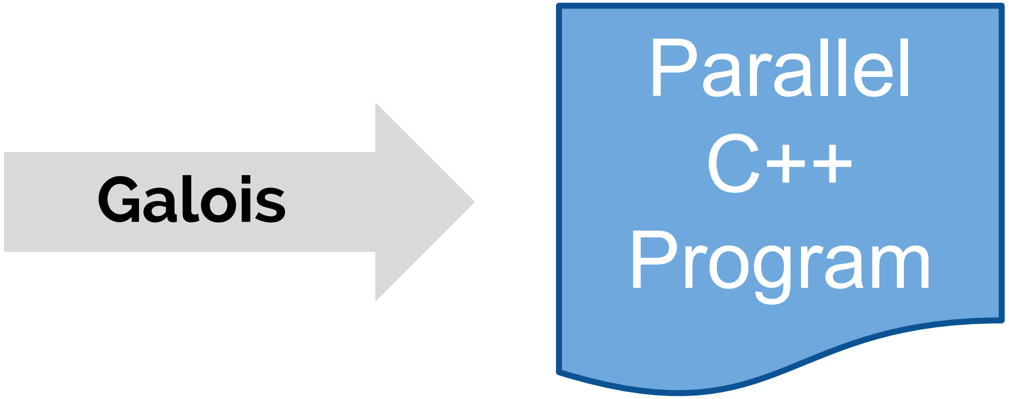
SSSP Elixir Specification

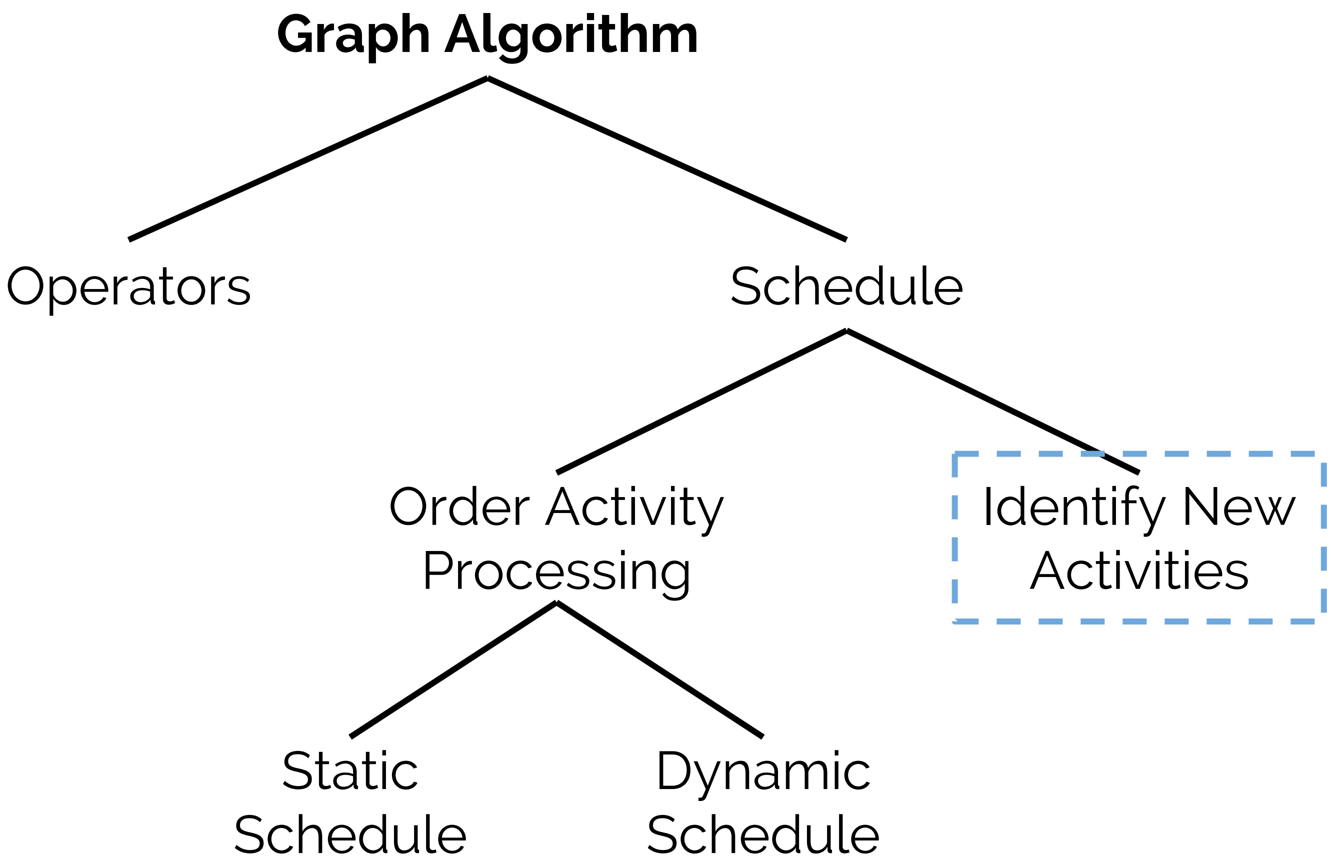
```
Graph [
     nodes(node: Node, dist: int)
     edges(src: Node, dest: Node, wt: int)
]

relax = [ nodes(node a, dist da)
          nodes(node b, dist db)          } Redex Pattern
          edges(src a, dest b, wt w)
          db > da + w ] ->                } Guard
          [db = da + w]                   } Update

sssp = iterate relax >> schedule
```

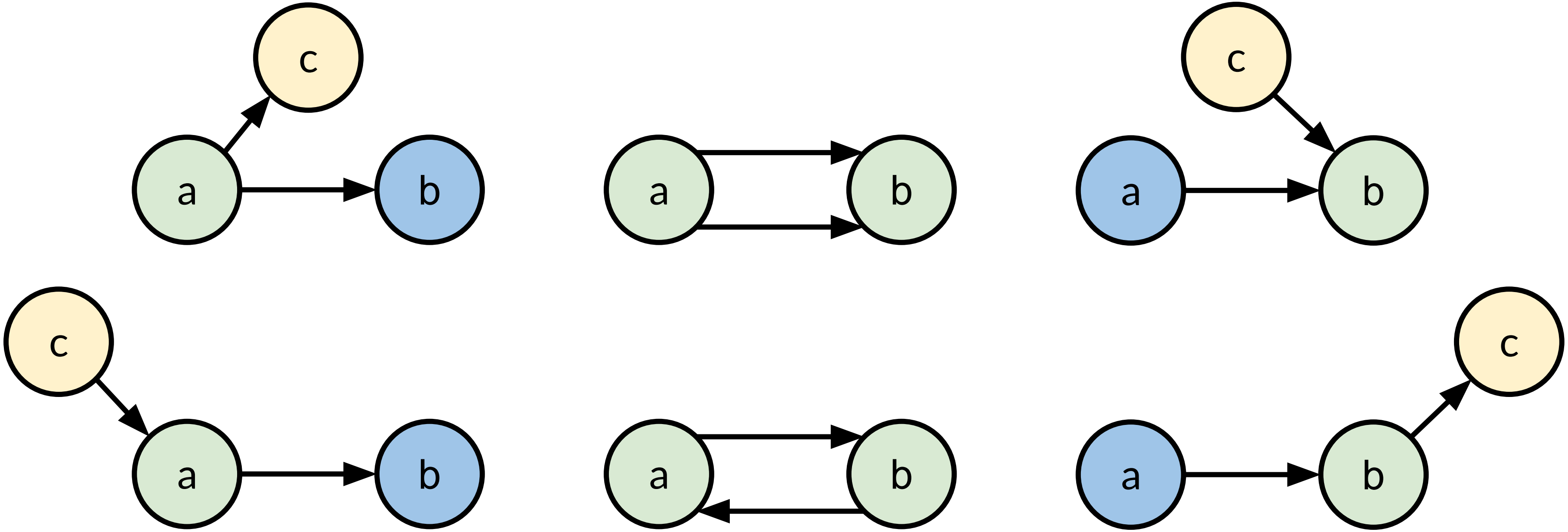


Scheduling

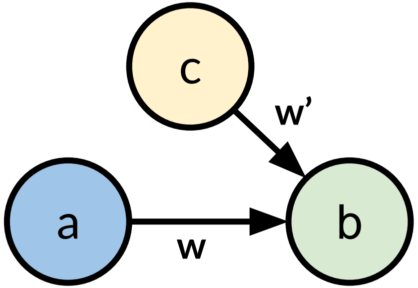
- Metric
- Group
- Fuse
- Unroll
- Ordered/unordered




Graph Algorithm
Operators
Schedule
Order Activity Processing
Identify New Activities
Static Schedule
Dynamic Schedule


c
a
b
a
b
c
a
b
c
a
b
a
b
c
a
b

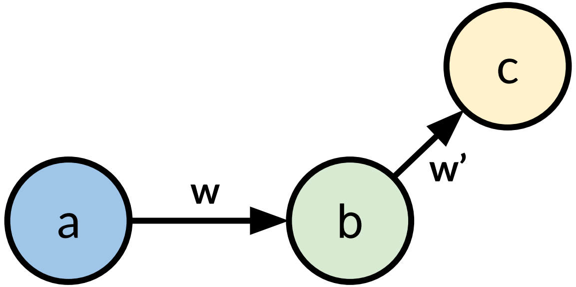
—




```
assume  ( da + w < db )
assume !( dc + w’ < db )
new_db = da + w
assert !( dc + w’ < new_db )
```

SMT Solver

✓




```
assume  ( da + w < db )
assume !( db + w' < dc )
new_db = da + w
assert !( new_db + w' < dc )
```

SMT Solver

✕

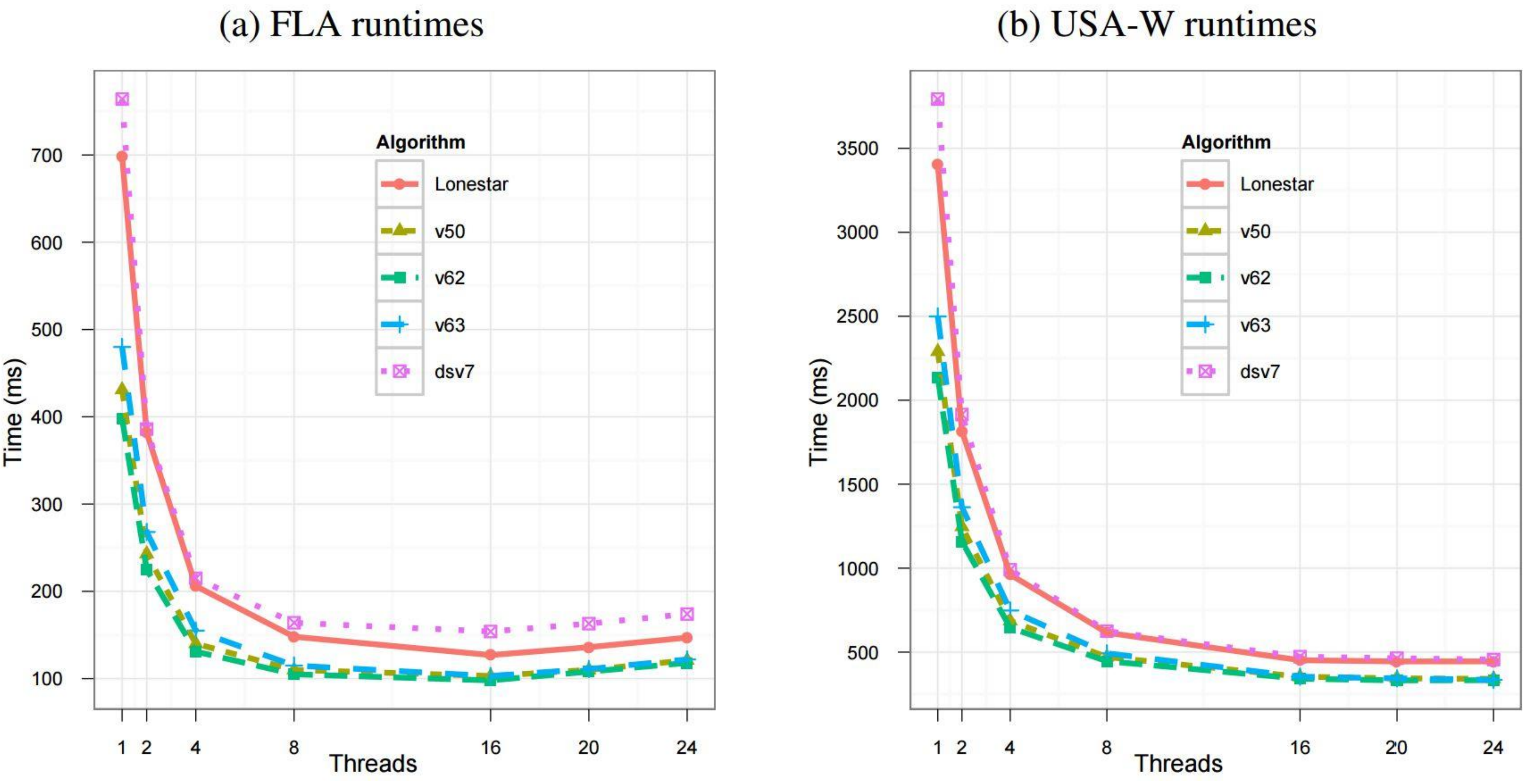
Evaluation

Experiments

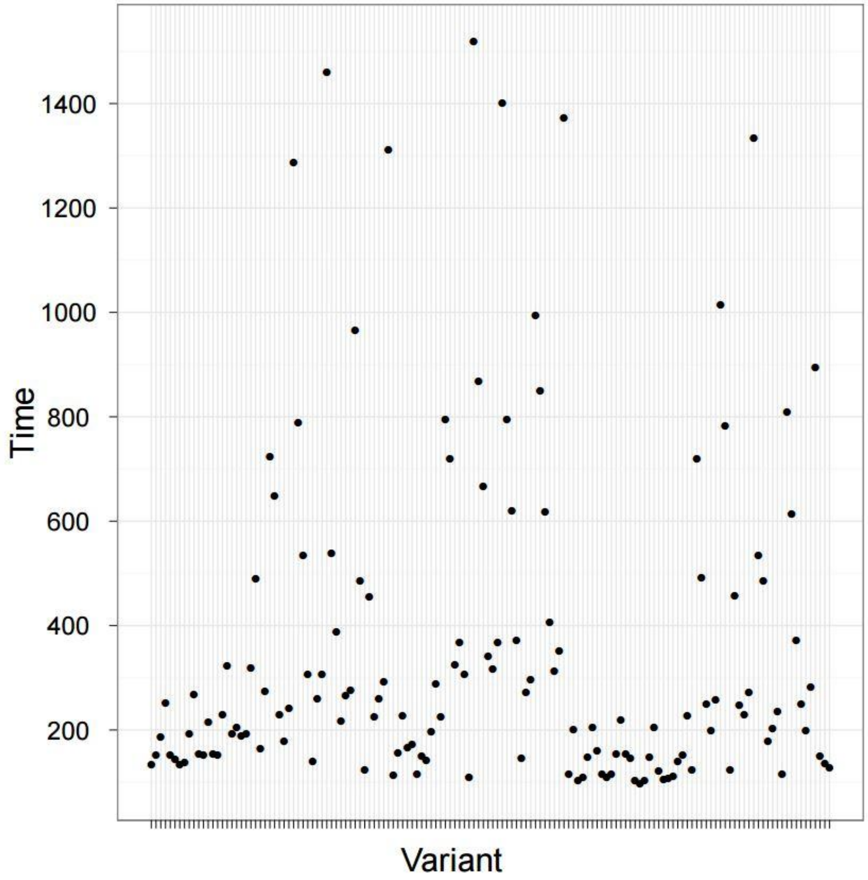
Explored Dimensions

`group`	Statically group multiple instances
`unroll k`	Statically unroll operator applications
`dynamic scheduler`	different worklist policy/implementation

...


(a) FLA runtimes
Algorithm
Lonestar
v50
v62
v63
dsv7
Time (ms)
700
600
500
400
300
200
100
Threads
1
2
4
8
16
20
24
(b) USA-W runtimes
Algorithm
Lonestar
v50
v62
v63
dsv7
Time (ms)
3500
3000
2500
2000
1500
1000
500
Threads
1
2
4
8
16
20
24

(c) FLA runtime distribution



Complexity

Definition 3.1 (Graph). [1] *A graph* $G = (V^G, E^G, Att^G)$ *where* $V^G \subset$ *Nodes are the graph nodes,* $E^G \subseteq V^G \times V^G$ *are the graph edges, and* $Att^G : ((Attrs \times V^G) \to Vals) \cup ((Attrs \times V^G \times V^G) \to Vals)$ *associates values with nodes and edges. We denote the set of all graphs by Graph.*

Definition 3.3 (Matching). *Let* G *be a graph and* P *be a pattern. We say that* $\mu : V^P \to V^G$ *is a* matching *(of* P *in* G*), written* $(G, \mu) \models P$*, if it is one-to-one, and for every edge* $(x, y) \in E^P$ *there exists an edge* $(\mu(x), \mu(y)) \in E^G$*. We denote the set of all matchings by* $Match : Vars \to Nodes$*.*

We extend a matching $\mu : V^P \to V^G$ to evaluate attribute variables $\mu : Vars \to Vals$ as follows. For every attribute a, pattern nodes $y, z \in V^P$, and attribute variable x, we define:

$$\begin{aligned} \mu(x) &= Att^G(a, \mu(y)) & \text{if} \quad & Att^P(a, y) = x \\ \mu(x) &= Att^G(a, \mu(y), \mu(z)) & \text{if} \quad & Att^P(a, y, z) = x \ . \end{aligned}$$

Definition 3.2 (Pattern). *A pattern* $P = (V^P, E^P, Att^P)$ *is a connected graph over variables. Specifically,* $V^P \subset Vars$ *are the pattern nodes,* $E^P \subseteq V^P \times V^P$ *are the pattern edges, and* $Att^P : (Attrs \times V^P) \to Vars \cup (Attrs \times V^P \times V^P) \to Vars$ *associates a distinct variable (not in* V^P*) with each node and edge. We call the latter set of variables* attribute variables. *We refer to* (V^P, E^P) *as the* shape *of the pattern.*

Let μ_R and $\mu_{R'}$ be two matchings corresponding to the operators above. We say that μ_R and $\mu_{R'}$ *overlap*, written $\mu_R \curlywedge \mu_{R'}$, if the matched subgraphs overlap: $\mu_R(V^R) \cap \mu_{R'}(V^{R'}) \neq \emptyset$. Then, the following equality holds:

$$\begin{aligned} &\textsc{Delta}[\![op, op']\!]\,(G, \mu_R) = \\ &\quad \textbf{let} \quad G' = [\![op]\!](G, \mu_R) \\ &\quad \textbf{in} \quad \{\mu_{R'} \mid \mu_{R'} \curlywedge \mu_R, \\ &\qquad\qquad (G, \mu_{R'}) \not\models R^{op}, Gd^{op}, \\ &\qquad\qquad (G', \mu_{R'}) \models R^{op}, Gd^{op}\} \ . \end{aligned}$$

Conclusion

- Elixir can beat hand-written implementations
- “High-level” specification could be simpler
- Not very accessible paper (unhelpful formalisms)
- Dynamic graphs unsupported
- Is auto-tuning integrated yet?