Elixir

A System for Synthesizing Concurrent Graph Programs

UNIVERSITY OF CAMBRIDGE

Motivation

- Performance of standard graph algorithms dependent on:
  - Graph topology (diameter)
  - Scheduling
  - Architecture (SIMD/MIMD)
  - …
- Elixir: high-level tool to generate optimal implementations

Specification

- Typed graph definition
- Operators with shape and value constraints define updates on subgraphs
- Statements: foreach, for i..range, iterate
- Scheduling operators restrict order

$$
\begin{aligned}
&\text{initDist} = [\ \text{nodes}(\textbf{node}\ a,\ \textbf{dist}\ d)\ ] \rightarrow \\
&\qquad [\ d = \text{if}\ (a == \text{source})\ 0\ \text{else}\ \infty]
\end{aligned}
$$

$$
\begin{aligned}
&\text{relaxEdge} = [\ \text{nodes}(\textbf{node}\ a,\ \textbf{dist}\ ad) \\
&\qquad \text{nodes}(\textbf{node}\ b,\ \textbf{dist}\ bd) \\
&\qquad \text{edges}(\textbf{src}\ a,\ \textbf{dst}\ b,\ \textbf{wt}\ w) \\
&\qquad ad + w < bd\ ] \rightarrow \\
&\qquad [\ bd = ad + w\ ]
\end{aligned}
$$

$$
\begin{aligned}
&\text{init} = \textbf{foreach}\ \text{initDist} \\
&\text{sssp} = \textbf{iterate}\ \text{relaxEdge} \gg \text{sched} \\
&\textbf{main} = \text{init}\ ;\ \text{sssp}
\end{aligned}
$$

UNIVERSITY OF CAMBRIDGE

Algorithms in Elixir

Algorithm	Schedule specification
Dijkstra	sched = **metric** ad $\gg$ **group** b

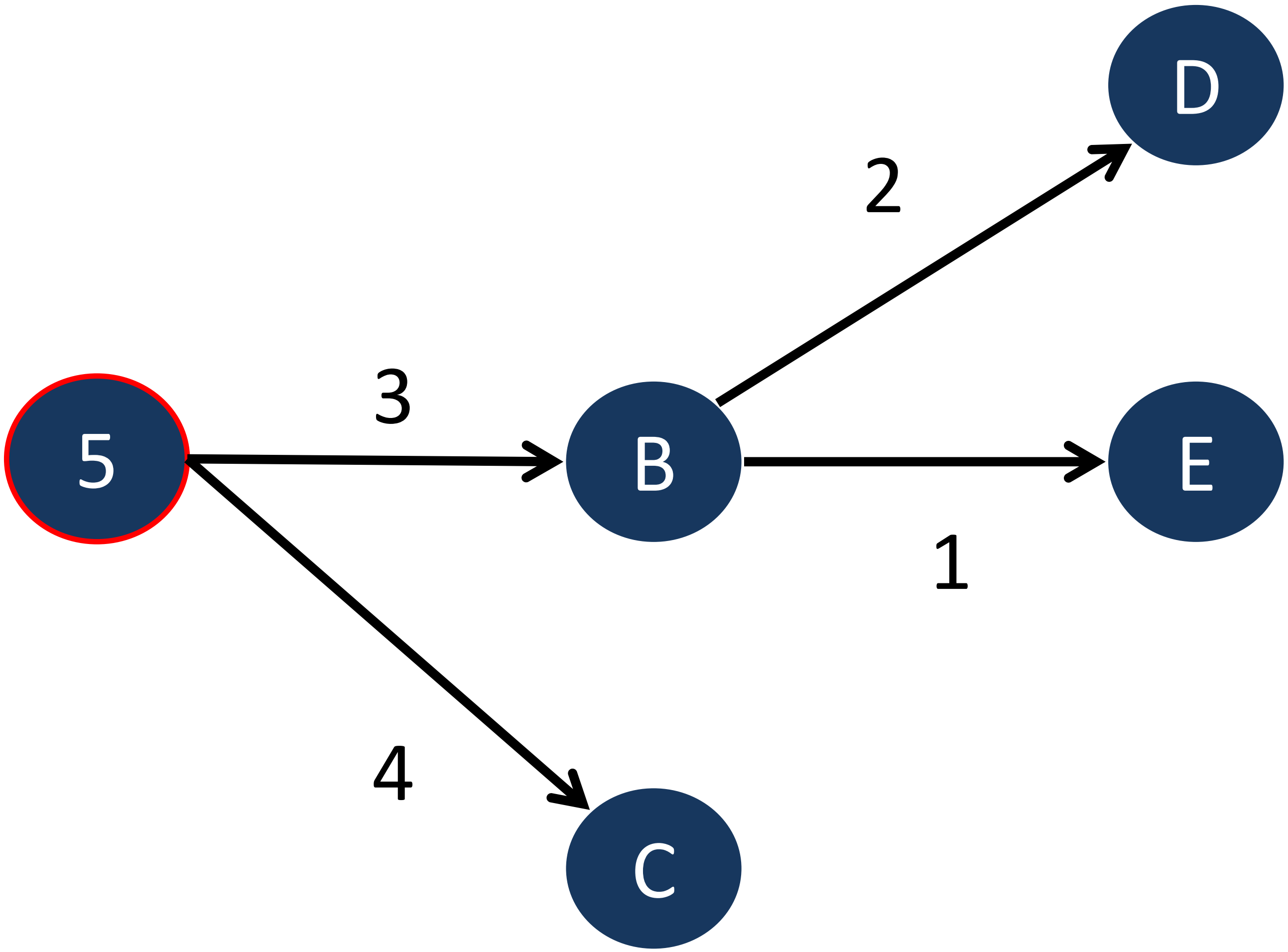
Bellman-Ford	NUM_NODES : **unsigned int**
	// override sssp
	sssp = **for** i =1..(NUM_NODES − 1)
	step
	step = **foreach** relaxEdge

UNIVERSITY OF CAMBRIDGE

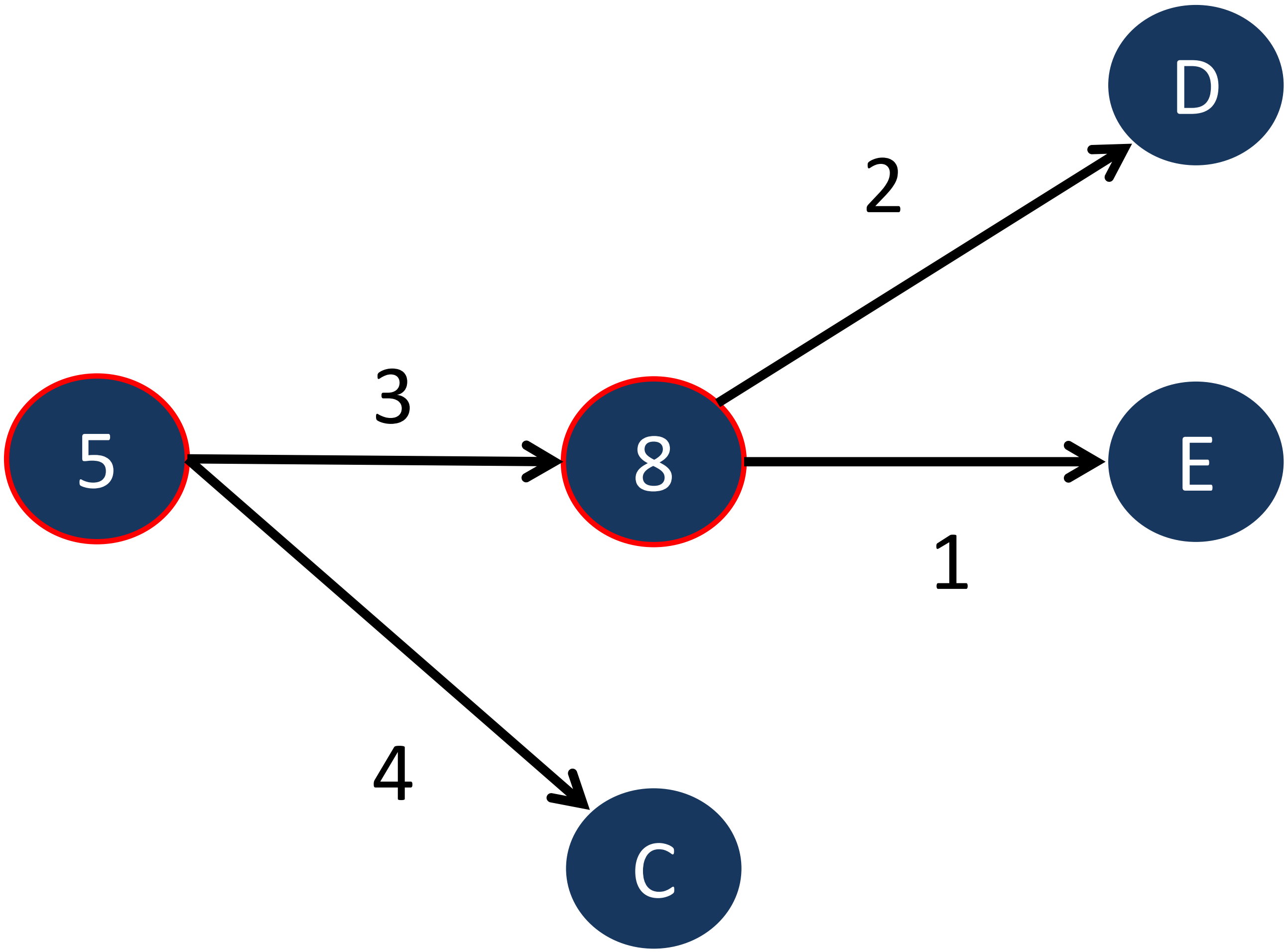
Scheduling operators

- Metric
  - Define online priorities
- Group
  - Co-schedule edges from same source
- Fuse
- Unroll
- Ordered/unordered

Example: unroll and group



Example: unroll and group

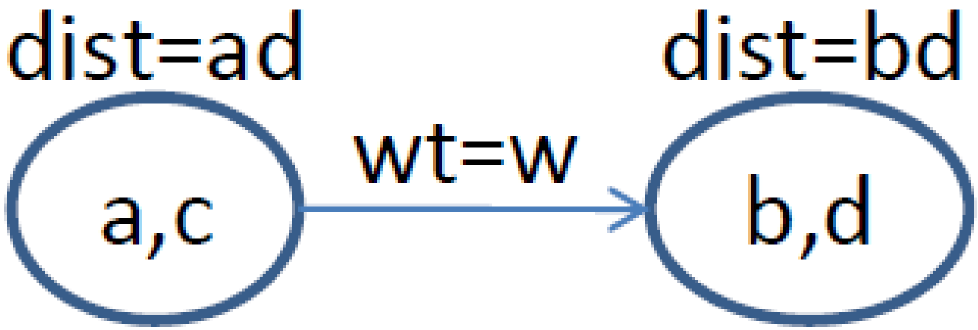


Synthesis

- Translated to C++, predefined graph types
- Worklists hold potential matching subgraphs
- Dynamic scheduling in OpenMP
- Enumerative exploration approach
  - Tests a predefined set of combinations of unroll, grouping

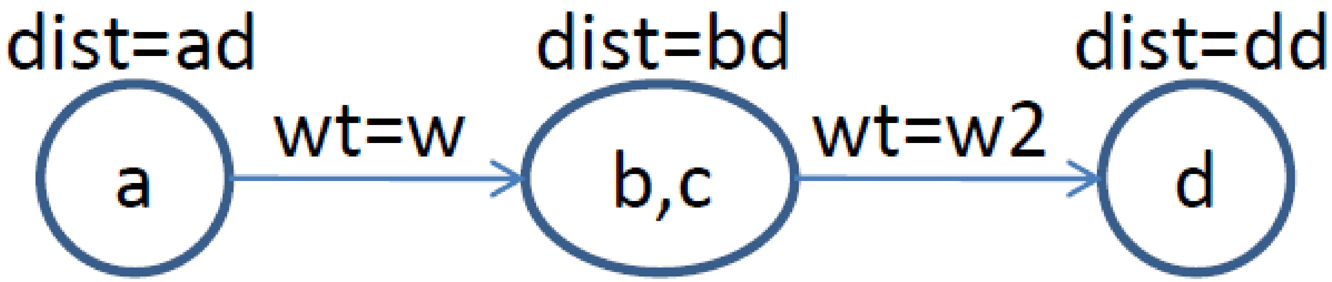
UNIVERSITY OF CAMBRIDGE

Matching subgraphs 1/2



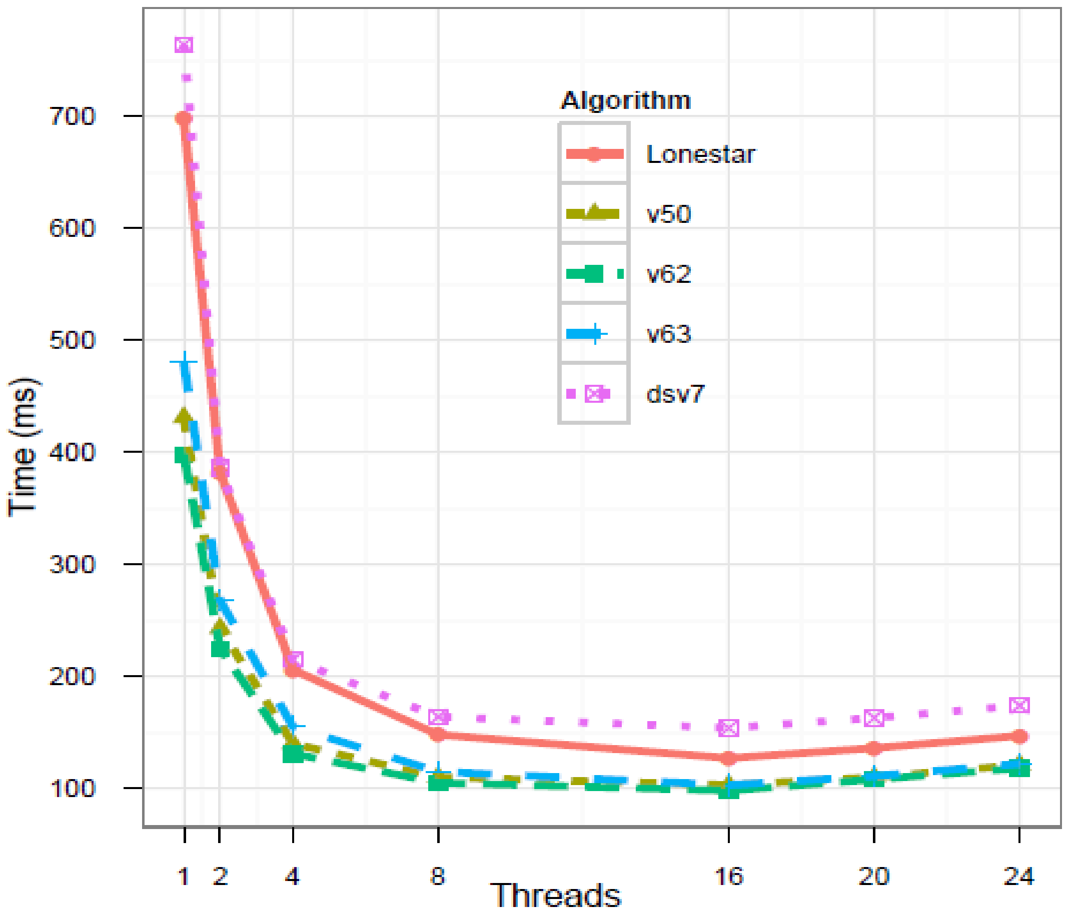
assume (ad + w < bd)
new_bd = ad + w
assert !(ad + w < new_bd)

Matching subgraphs 2/2



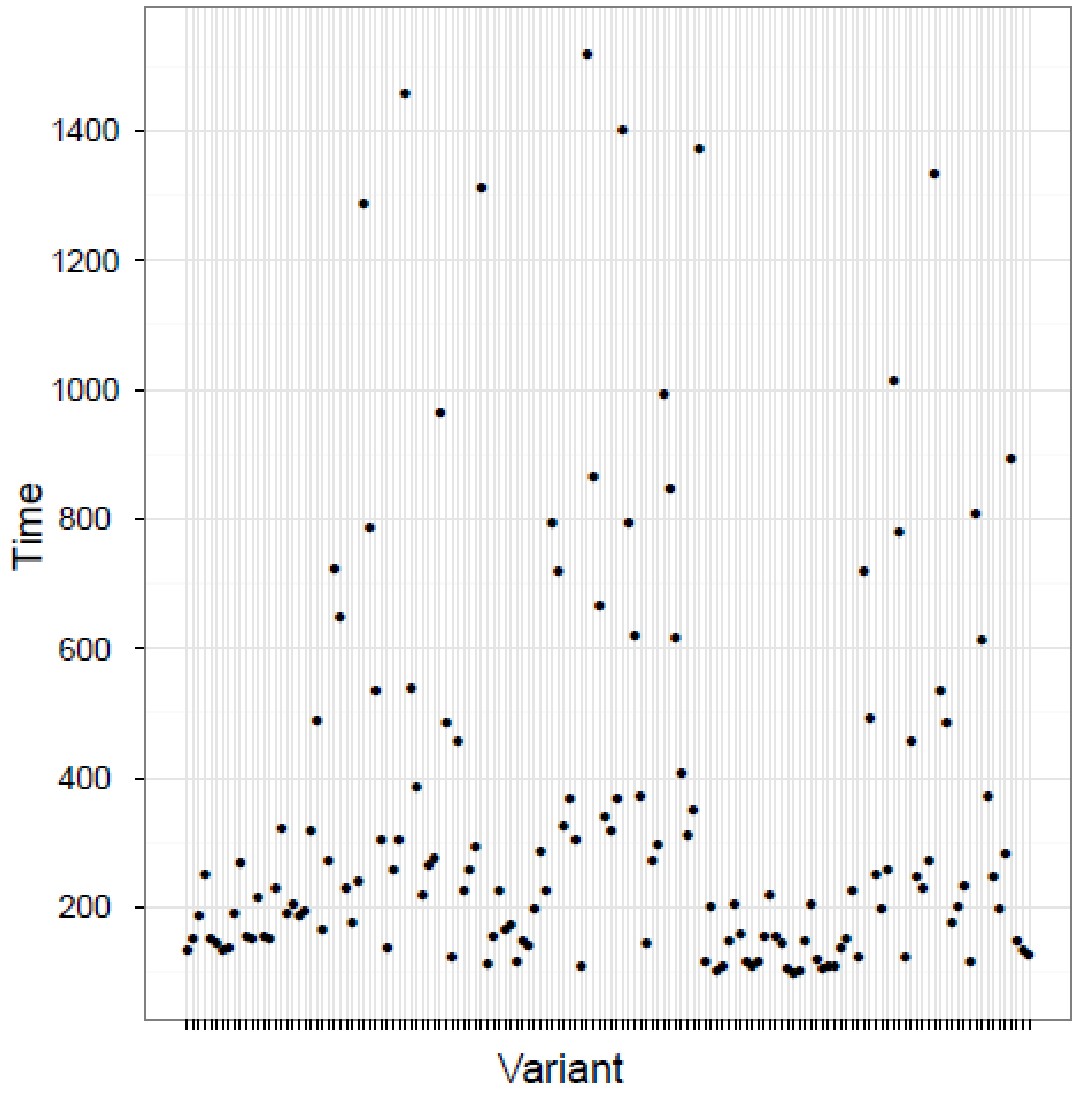

```
assume  (ad + w < bd)
assume !(bd + w2 < dd)
new_bd = ad + w
assert !(new_bd + w2 < dd)
```

Evaluation 1/2



UNIVERSITY OF CAMBRIDGE

Evaluation 2/2



UNIVERSITY OF CAMBRIDGE

A Graph G…

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.*

$$
\begin{aligned}
V^D &\stackrel{\text{def}}{=} \{\mu(x) \mid x \in V^R\} \\
E^D &\stackrel{\text{def}}{=} \{(\mu(x), \mu(y)) \mid (x, y) \in E^R\} \\
Att^D &\stackrel{\text{def}}{=} \{(a, Att^G(a, u)), (b, Att^G(b, v, w)) \mid \\
&\qquad a, b \in Attrs,\ u \in V^D, (v, w) \in E^D\}
\end{aligned}
$$

$$
Att'(a, v) = \begin{cases} \mu(Upd^{op}(y)), & v \in V^D, v = \mu(x_v) \\ & \text{and } Att^R(a, x_v) = y; \\ Att(a, v) & \text{else.} \end{cases}
$$

$$
Att'(a, u, v) = \begin{cases} \mu(Upd^{op}(y)), & (u, v) \in E^D, \\ & u = \mu(x_u), v = \mu(x_v) \\ & \text{and } Att^R(a, x_u, x_v) = y; \\ Att(a, u, v) & \text{else.} \end{cases}
$$

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.*

UNIVERSITY OF CAMBRIDGE

Conclusion

- “Does not rely on expert knowledge”
- High up-front effort to learn specification
- Bloated formalism
- Can beat hand-written implementations through intricate load-balancing
- No dynamic graphs supported