Elixir

A System for Synthesizing Concurrent Graph Programs
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
\[
\text{initDist} = \left[ \text{nodes}(\text{node a, dist d}) \right] \rightarrow \\
\left[ d = \text{if } (a == \text{source}) \ 0 \ \text{else} \ \infty \right]
\]

\[
\text{relaxEdge} = \left[ \text{nodes}(\text{node a, dist ad}) \\
\text{nodes}(\text{node b, dist bd}) \\
\text{edges}(\text{src a, dst b, wt w}) \\
ad + w < bd \right] \rightarrow \\
\left[ bd = ad + w \right]
\]

\[
\text{init} = \text{foreach} \ \text{initDist} \\
\text{sssp} = \text{iterate} \ \text{relaxEdge} \gg \text{sched} \\
\text{main} = \text{init} ; \ \text{sssp}
\]
## Algorithms in Elixir

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Schedule specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>sched = <code>metric</code> ad (\Rightarrow) <code>group</code> b</td>
</tr>
<tr>
<td></td>
<td>NUM_NODES : <code>unsigned int</code></td>
</tr>
<tr>
<td></td>
<td>// override sssp</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>sssp = <code>for</code> i = 1..(NUM_NODES - 1) step</td>
</tr>
<tr>
<td></td>
<td>step = <code>foreach</code> relaxEdge</td>
</tr>
</tbody>
</table>
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
Matching subgraphs 1/2

\[
\text{assume } (ad + w < bd) \\
\text{new\_bd} = ad + w \\
\text{assert } !((ad + w < \text{new\_bd})
\]
Matching subgraphs 2/2

\[
\begin{align*}
&\text{dist}=ad \\
&\text{wt}=w \\
&\text{dist}=bd \\
&\text{wt}=w2 \\
&\text{dist}=dd
\end{align*}
\]

\begin{align*}
\text{assume} \ (ad + w < bd) \\
\text{assume} \ !(bd + w2 < dd) \\
\text{new\_bd} = ad + w \\
\text{assert} \ !(\text{new\_bd} + w2 < dd)
\end{align*}
Evaluation 1/2
Evaluation 2/2
A Graph G...

Definition 3.1 (Graph). A graph \( G = (V^G, E^G, \text{Att}^G) \) where \( V^G \subseteq \text{Nodes} \) are the graph nodes, \( E^G \subseteq V^G \times V^G \) are the graph edges, and \( \text{Att}^G : ((\text{Attrs} \times V^G) \rightarrow \text{Vals}) \cup ((\text{Attrs} \times V^G \times V^G) \rightarrow \text{Vals}) \) associates values with nodes and edges. We denote the set of all graphs by Graph.

\[
\begin{align*}
V^D & \overset{\text{def}}{=} \{ \mu(x) \mid x \in V^R \} \\
E^D & \overset{\text{def}}{=} \{ (\mu(x), \mu(y)) \mid (x, y) \in E^R \} \\
\text{Att}^D & \overset{\text{def}}{=} \{ (a, \text{Att}^G(a, u)), (b, \text{Att}^G(b, v, w)) \mid a, b \in \text{Attrs}, u \in V^D, (v, w) \in E^D \}
\end{align*}
\]

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

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

Definition 3.2 (Pattern). A pattern \( P = (V^P, E^P, \text{Att}^P) \) is a connected graph over variables. Specifically, \( V^P \subseteq \text{Vars} \) are the pattern nodes, \( E^P \subseteq V^P \times V^P \) are the pattern edges, and \( \text{Att}^P : (\text{Attrs} \times V^P) \rightarrow \text{Vars} \cup (\text{Attrs} \times V^P \times V^P) \rightarrow \text{Vars} \) 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.
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