Green-Marl: A DSL for Easy and Efficient Graph Analysis

Sungpack Hong, Hassan Chafi, Eric Sedlar, Kunle Olukotun

Presented By
Albert Kim
Goal

- Tough to write parallel graph algorithms
- Current graph processing frameworks force user to rewrite their program

Want a Domain Specific Language (DSL)

- Easy to express graph algorithms
- Expose data-level parallelism
- Can compile to various backends
Sample Code

1 Procedure Compute_BC(
2 G: Graph, BC: Node.Prop<Float>(G)) {
3 G.BC = 0; // initialize BC
4 ForEach(s: G.Nodes) {
5 // define temporary properties
6 Node.Prop<Float>(G) Sigma;
7 Node.Prop<Float>(G) Delta;
8 s.Sigma = 1; // Initialize Sigma for root
9 // Traverse graph in BFS-order from s
10 InBFS(v: G.Nodes From s)(v!=s) {
11 // sum over BFS-parents
12 v.Sigma = Sum(w: v.UpNbrs) {w.Sigma};
13 }
14 // Traverse graph in reverse BFS-order
15 InRBFS(v!=s) {
16 // sum over BFS-children
17 v.Delta = Sum (w:v.DownNbrs) {
18 v.Sigma / w.Sigma * (1+ w.Delta)
19 });
20 v.BC += v.Delta @s; //accumulate BC
21 } } }

Figure 1. Betweenness Centrality algorithm described in Green-Marl
Scope of Language

• Graph is ordered pair of nodes and edges
  – $G = (N, E)$
• Each node/edge has some properties
• Given graph and set of properties $(G, \Pi)$, compute:
  – A scalar
  – A new set of properties for each node/edge
  – A subgraph of original graph
Data Structures

- Five primitives
  - Bool, Int, Long, Float, and Double
- Collection types
  - Set (unique and unordered)
  - Order (unique and ordered)
  - Sequence (not unique and ordered)
- Special semantics when dealing with collections in sequential/parallel context
Data Structures (Sample)

```plaintext
34 Procedure foo(G1, G2:Graph, n:Node(G1)) {
35    Node(G2) n2; // a node of graph G2
36    n2 = n; // type-error (bound to different graphs)
37    Node_Prop<Int>(G1) A; // integer node property for G1
38    n.A = 0;
39    Node_Set(G1) S; // a node set of G1
40    S.Add(n);
41 }
```
# Operations on Collections

<table>
<thead>
<tr>
<th>Group</th>
<th>Op-Name</th>
<th>sequential</th>
<th>parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$S$</td>
<td>$O$</td>
</tr>
<tr>
<td>Grow</td>
<td>Add</td>
<td>$v$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Push(Front/Back)</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>Shrink</td>
<td>Remove</td>
<td>$v$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pop(Front/Back)</td>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>Clear</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>Lookup</td>
<td>Has</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>Front(Back)</td>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>Copy</td>
<td>$=$</td>
<td>$v$</td>
<td>$v$</td>
</tr>
<tr>
<td>Iteration</td>
<td>Items</td>
<td>$v$</td>
<td>$v$</td>
</tr>
</tbody>
</table>

Modification under iteration $\rightarrow$ Shrink, Grow, or Copy: $X$

Conflicts under parallel execution $\rightarrow$

Procedure Compute_BC(
  G: Graph, BC: Node Prop<Float>(G)) {
  G.BC = 0;  // initialize BC
  Foreach(s: G.Nodes) {
    // define temporary properties
    Node Prop<Float>(G) Sigma;
    Node Prop<Float>(G) Delta;
    s.Sigma = 1;  // Initialize Sigma for root
    // Traverse graph in BFS-order from s
    InBFS(v: G.Nodes From s)(v!=s) {
      // sum over BFS-parents
      v.Sigma = Sum(w: v.UpNbrs) {w.Sigma};
    }
    // Traverse graph in reverse BFS-order
    InRBFS(v!=s) {
      // sum over BFS-children
      v.Delta = Sum (w:v.DownNbrs) {
        v.Sigma / w.Sigma * (1+ w.Delta)
      };
      v.BC += v.Delta @s;  //accumulate BC
    }
  }
}

Figure 1. Betweenness Centrality algorithm described in Green-Marl
### BFS Traversal Figure

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Range</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>D/UGraph</td>
<td>Nodes</td>
<td>Linear</td>
</tr>
<tr>
<td>Node(D/UGraph)</td>
<td>Nbrs</td>
<td>Random</td>
</tr>
<tr>
<td>Node(DGraph)</td>
<td>OutNbrs</td>
<td>Random</td>
</tr>
<tr>
<td>Node(DGraph)</td>
<td>InNbrs</td>
<td>Random</td>
</tr>
<tr>
<td>Node(D/UGraph)</td>
<td>UpNbrs</td>
<td>Random/-1</td>
</tr>
<tr>
<td>Node(D/UGraph)</td>
<td>DownNbrs</td>
<td>Random/+1</td>
</tr>
<tr>
<td>Node_Set</td>
<td>Items</td>
<td>Linear</td>
</tr>
<tr>
<td>Node_Order</td>
<td>Items</td>
<td>Linear</td>
</tr>
<tr>
<td>Node_Seq</td>
<td>Items</td>
<td>Random</td>
</tr>
</tbody>
</table>
Parallelism in Green-Marl

• Inspired by OpenMP
• Follows OpenMP’s memory consistency model
  – Writing to same shared variable in concurrently may cause conflicts

30   Foreach (s: G.Nodes)
31       Foreach (t: s.OutNbrs)
32           t.A =              // write-write conflict
33               t.A + s.B;   // read-write conflict
Reductions

58  `Int x, y;
59  x = `Sum(t:G.Nodes){t.A}; // equivalent to next 3 lines.
60  y = 0;
61  `Foreach(t:G.Nodes)
62       y += t.A;

<table>
<thead>
<tr>
<th>In-place</th>
<th>Assignment</th>
<th>In-place</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>&amp;&amp;=</td>
<td>Sum</td>
<td>+=</td>
</tr>
<tr>
<td>Any</td>
<td></td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>Min</td>
<td>min=</td>
<td>Count</td>
<td>++</td>
</tr>
<tr>
<td>Max</td>
<td>max=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 3. Overview of Green-Marl DSL-compiler Usage
Compiler Optimizations: Loop Fusion

103 \textbf{Foreach} (s: \texttt{G.Nodes}) (f(s))
104 \hspace{5mm} s.A = X(s.B);
105 \textbf{Foreach} (t: \texttt{G.Nodes}) (g(t))
106 \hspace{5mm} t.B = Y(t.A)

becomes

107 \textbf{Foreach} (s: \texttt{G.Nodes}) (\texttt{
108 \hspace{5mm} if (f(s)) s.A = X(s.B);
109 \hspace{5mm} if (g(s)) s.B = Y(s.A);
110 })
Compiler Optimizations: Hoisting Definitions

111  For(s:G.Nodes) { //sequential loop
112     Node_Prop<Int>(G) A;
113
114     ...
115
116  }

becomes

115  Node_Prop<Int>(G) A;
116  For(s:G.Nodes) {
117      ...
118  }
Compiler Optimizations: 
Set-Graph Loop Fusion

139  Node_Set S(G); // ...
140  Foreach (s: S.Items)
141      s.A = x(s.B);
142  Foreach (t: G.Nodes) (g(t))
143      t.B = y(t.A)

becomes

144  Foreach (s: G.Nodes) (
145      if (S.Has(s)) s.A = x(s.B);
146      if (g(s)) s.B = y(s.A);
147  )
## Evaluation (LOC)

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC Original</th>
<th>LOC Green-Marl</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>350</td>
<td>24</td>
<td>[9] (C OpenMp)</td>
</tr>
<tr>
<td>Conductance</td>
<td>42</td>
<td>10</td>
<td>[9] (C OpenMp)</td>
</tr>
<tr>
<td>Vetex Cover</td>
<td>71</td>
<td>25</td>
<td>[9] (C OpenMp)</td>
</tr>
<tr>
<td>PageRank</td>
<td>58</td>
<td>15</td>
<td>[2] (C++, sequential)</td>
</tr>
<tr>
<td>SCC(Kosaraju)</td>
<td>80</td>
<td>15</td>
<td>[3] (Java, sequential)</td>
</tr>
</tbody>
</table>
Evaluation (BC)

(a) RMAT

(b) Uniform
Evaluation (Vertex Cover)

(a) RMAT

(b) Uniform
My Impressions

- Syntax is galling

```cpp
// Computing PageRank

Procedure PageRank(G: Graph, e,d: Double, max_iter: Int, PR: Node_PROP<Double>(G)) {
    Double diff = 0;
    // Initialization
    Int cnt = 0;
    Double N = G.NumNodes();
    G.PR = 1 / N;

    Do {
        diff = 0.0;
        // Main Iteration.
        Foreach (t: G.Nodes) {
            Double val = (1-d) / N +
            d* Sum(w: t.InNbrs) (w.OutDegree() > 0) (w.PR / w.OutDegree());
            t.PR := val @ t;
            diff += | val - t.PR |;
        }
        cnt++;
        // ++ is a syntactic sugar.
    } While ((diff > e) && (cnt < max_iter));
    // Iterate for max num steps or difference is smaller than given threshold.
```
# My Impressions

- Have to deal with semantics of collections

<table>
<thead>
<tr>
<th>Group</th>
<th>Op-Name</th>
<th>sequential</th>
<th></th>
<th></th>
<th>parallel</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$S$</td>
<td>$O$</td>
<td>$Q$</td>
<td>$S$</td>
<td>$O$</td>
<td>$Q$</td>
</tr>
<tr>
<td>Grow</td>
<td>Add</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Push(Front/Back)</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td>Shrink</td>
<td>Remove</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pop(Front/Back)</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Clear</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Lookup</td>
<td>Has</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>Front(Back)</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Size</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>Copy</td>
<td>=</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Iteration</td>
<td>Items</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

Modification under iteration → Shrink, Grow, or Copy: X

Conflicts under parallel execution →

- Grow-Shrink: X
- Lookup-Shrink: ?
- Lookup-Grow: ?
My Impressions

• Have to deal with semantics of iterations
  – Uses OpenMP’s weak memory consistency model
  – Should make it impossible to share variables

• Have to deal with parallel workflow of iterations
  – BFS: each level is parallel
  – DFS: sequential
  – Not data-level parallelism
Overall Impressions

- **Like:**
  - Idea of DSL
  - Easy way to process graph (BFS traversal)
  - Expose data-level parallelism (necessary)
  - Compiler optimizations
  - Portable backend
  - Global view: easy to work on global variables

- **Dislike:** Green-Marl’s DSL

- **Want:** Higher-Level DSL