GREEN-MARL: A DSL FOR EASY AND EFFICIENT GRAPH ANALYSIS

Sungpack Hong, Hassan Chafi, Eric Sedlar, Kunle Olukotun

K.M.D.M Karunarathna
University Of Cambridge - 17th Nov 2015
Current Issues

Issues with large-scale graph analysis

- Performance
- Implementation
- Capacity
Performance Issues

- RAM latency dominates running time for large graphs

**Solution**: Solved by exploiting data parallelism
Implementation Issues

- Writing concurrent code is *hard*
- Race-conditions
- Deadlock
- Efficiency requires deep hardware knowledge
- Couples code to underlying architecture
Solution: A DSL Green-Marl and its compiler

- High level graph analysis language
- Hides underlying complexity
- Exposes algorithmic concurrency
- Exploits high level domain information for optimisations
Example

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

■ Scope of the Language
   Based on processing graph properties, mappings from a node/edge to a value
   - e.g. the average number of phone calls between two people

■ Green-Marl is designed to compute,
   • scalar values from a graph and its properties
   • new properties for nodes/edges
   • selecting subgraphs (instance of above)
Green-Marl Language Design

- Parallelism in Green-marl

  Support for parallelism (fork-join style)
  
  - Implicit
    
    \[ G_{BC} = 0; \]
  
  - Explicit
    
    \[ \text{Foreach}(s: G.Nodes) \ (s \neq t) \]
  
  - Nested
    
    \[ p\_sum \ *= t.B; \]
Language Constructs

- Data Types and Collections - **DATA**
  a) Five primitive types (**Bool**, **Int**, **Long**, **Float**, **Double**)
  b) Defines two graph types (**DGraph** and **UGraph**)
  c) Second, there is a node type and an edge type both of which are always bound to a graph instance
  d) Node properties and edge properties which are bound to a graph but have base-types as well
Language Constructs

- Data Types and Collections - **COLLECTION**
  - Set, Order, and Sequence.
    a) Elements in a Set are unique while a Set is unordered.
    b) Elements in an Order are unique while an Order is ordered.
    c) Elements in a Sequence are not unique while a Sequence is ordered
Language Constructs

- Iterations and Traversals
  
  ```
  Foreach (iterator:source(-).range)(filter)
  body_statement
  ```
Language Constructs

■ Deferred Assignment
  a) Supports bulk synchronous consistency via deferred assignments.
  b) Deferred assignments are denoted by $\leftarrow$ and followed by a binding symbol.
Language Constructs

Reductions

- an expression form (or in-place from)
- an assignment form

\[ y += t.A; \]
Compiler

- Compiler Overview

*Figure.* Overview of Green-Marl DSK-compiler Usage
Compiler

- Architecture Independent Optimizations
  - Group Assignment
  - In-place Reduction
  - Loop Fusion
  - Hoisting Definitions
  - Reduction Bound Relaxation
  - Flipping Edges

\[
\text{Foreach (t:G.Nodes) (f(t))}
\]
\[
\text{Foreach (s:t.InNbrs) (g(s))}
\]
\[
t.A += s.B;
\]

Becomes

\[
\text{Foreach (s:G.Nodes) (g(s))}
\]
\[
\text{Foreach (t:s.OutNbrs) (f(t))}
\]
\[
t.A += s.B;
\]
Compiler

- Architecture Dependent Optimizations
  - Set-Graph Loop Fusion
  - Selection of Parallel Regions
  - Deferred Assignment
  - Saving BFS Children

```c
_inBFS(v:G.Nodes; s) { ... //forward }
_inRBFS { // reverse-order traverse
  Foreach(t: v.DownNbrs) {
    DO_THING(t);
  }
}
```

Becomes

```c
_prepare_edge_marker(); // O(E) array
for (e = edges ... ) {
  index_t t = ...node(e);
  if (isNextLevel(t)) {
    edge_marker[e] = 1;
  }
}
```

```c
for (e = edges ..) {
  if (edge_marker[e] ==1) {
    index_t t = ...node(e);
    DO_THING(t);
  }
}
```
Compiler

- Code Generation
  - Graph and Neighborhood Iteration
  - Efficient DFS and BFS traversals
  - Small BFS Instance Optimization
  - Reduction on Properties
  - Reduction on Scalars
## Experiments

<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>Vertex 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>

*Table. Graph algorithms used in the experiments and their Lines-of-Code (LOC) when implemented in Green-Marl and in a general purpose language.*
Experiments

Figure. Speed-up of Betweenness Centrality. Speed-up is over the SNAP library [9] version running on a single-thread. NoFlipBE and NoSaveCh means disabling the Flipping Edges (Section 3.3 Architecture Independent Optimizations) and Saving BFS Children (Section 3.5 Code Generation) optimizations respectively.
Experiments

Figure. Speed-up of Conductance. Speed-up is over the SNAPlibrary [9] version running on a single-thread. NoLM and NoSRDC means disabling the Loop Fusion (Section 3.3 Architecture Independent Optimizations) and Reduction on Scalars (Section 3.5 Code Generation) optimizations, respectively.
Future Works

- Solutions for Capacity Issue
- Comments block to green Marl
- Combining with Graph Lab as back end (machine learning type)
- generate code for alternative architectures (Clusters, GPU).
- Green Marl as internal DSL.
Pros

• Easier to write graph algorithms
• Algorithms perform better
• Don’t need to rewrite entire application
• Code is portable across platforms
Critical Evaluation

• Assumes graph is immutable during the analysis