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

Sungpack Hong*, Hassan Chafi**, Eric Sedlar+, and Kunle Olukotun*

*Pervasive Parallelism Lab, Stanford University
+Oracle Labs
Graph Analysis

- Classic graphs; New applications
  - Artificial Intelligence, Computational Biology, ...
  - SNS apps: Linkedin, Facebook, ...

- Example: Movie Database

  Graph Analysis: a process of drawing out further information from the given graph data-set

“Do these actors work together more frequently than others?”

“Is he a central figure in the movie network? How much?”

“What would be the avg. hop-distance between any two (Australian) actors?”
More formally …

- **Graph Data-Set**
  - *Graph* $G = (V,E)$: *Arbitrary* relationship $(E)$ between data entities $(V)$
  - *Property* $P$: any extra data associated with each vertex or edge of graph $G$ (*e.g. name of the person*)
  - Your Data-Set = $(G, \Pi) = (G, P_1, P_2, \ldots)$

- **Graph analysis on** $(G, \Pi)$
  - Compute a scalar value
    - e.g. Avg-distance, conductance, eigen-value, ...
  - Compute a (new) property
    - e.g. (Max) Flow, betweenness centrality, page-rank, ...
  - Identify a specific subset of $G$:
    - e.g. Minimum spanning tree, connected component, community structure detection, ...
The Performance Issue

- Traditional single-core machines showed limited performance for graph analysis problems
  - A lot of random memory accesses + data does not fit in cache
    - Performance is bound to memory latency
  - Conventional hardware (e.g. floating point units) does not help much

➡️ Use parallelism to accelerate graph analysis
- Plenty of data-parallelism in large graph instances
- Performance now depends on memory *bandwidth*, not *latency*.
- Exploit modern parallel computers: Multi-core CPU, GPU, Cray XMT, Cluster, ...
It is challenging to implement a graph algorithm
- correctly
- + and efficiently
- + while applying parallelism
- + differently for each execution environment

Are we really expecting a single (average-level) programmer to do all of the above?
Our approach: DSL

- We design a domain specific language (DSL) for graph analysis
- The user writes his/her algorithm concisely with our DSL
- The compiler translates it into the target language (e.g. parallel C++ or CUDA)

(1) Inherent data-parallelism

(2) Good impl. templates

(3) High-level optimization

Intuitive Description of a graph algorithm

Efficient (parallel) Implementation of the given algorithm

For(i=0;i<G.numNodes;i++) {
  __fetch_and_add(G.nodes[i], ...)
}

Foreach (t: G.Nodes)
  t.sigma += ...

Intuitive Description

For(i=0;i<G.numNodes;i++) {
  _fetch_and_add(G.nodes[i], ...)
}

DSL
Example: Betweenness Centrality

Betweenness Centrality (BC)

- A measure that tells how ‘central’ a node is in the graph
- Used in social network analysis
- Definition
  - How many shortest paths are there between any two nodes going through this node.

\[ C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \]
Example: Betweenness Centrality

[Brandes 2001]

Init BC for every node and begin outer-loop (s)

Looks complex

Queues, Lists, Stack… Is this parallelizable?

Compute delta from children

Accumulate delta into BC
Example: Betweenness Centrality

[Brandes 2001]

Procedure $\text{comp\_BC}(G: \text{Graph}, \text{BC}: \text{Node\_Property}\langle\text{Float}\rangle(G))$

1. $G.\text{BC} = \emptyset$; // Initialize
2. For each $s \in G.\text{Nodes}$ do
   1. // temporary values per Node
      1. $G.\text{Node\_Property}\langle\text{Float}\rangle(G).\text{sigma} = \emptyset$;
      1. $G.\text{Node\_Property}\langle\text{Float}\rangle(G).\text{delta} = \emptyset$;
   2. $G.\text{sigma} = \emptyset$; // Initialize
   3. $G.\text{delta} = \emptyset$;
   4. $s.\text{sigma} = 1$;
   5. // BFS order iteration from $s$
      1. $\text{InBFS}(v : G.\text{Nodes}\ From\ s)$
         1. $v.\text{sigma} = \emptyset$; // Summing over BFS parents
            1. $\text{Sum}(w : v.\text{UpNbrs}) \{w.\text{sigma}\}$;
      2. // Reverse-BFS order iteration to $s$
         1. $\text{InRBS}(v : G.\text{Nodes}\ To\ s) (v != s)$
            1. $v.\text{delta} = \emptyset$; // Summing over BFS children
               1. $\text{Sum}(w : v.\text{DownNbrs})$
                  1. $v.\text{sigma} / w.\text{sigma} * (1+ w.\text{delta})$;
      3. $\text{if } w / s \text{ then } C_{W}[w] = C_{W}[w] + |\delta[w]|$;
5. $v.\text{BC} += v.\text{delta} @ s$; // accumulate BC
Example: Betweenness Centrality

[Brandes 2001]

\[
C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}
\]

\[
\sigma_{st}(v) = \sigma_{st} - \sigma_{sv} - \sigma_{tv}
\]

\[
\sigma(v) = \sigma_{sv} + \sigma_{tv} + \sum_{u \in N(v)} \sigma(v/u)
\]

**Procedure** \(\text{comp}_\text{BC}(G, BC, \text{Node\_Property<Float>}(G))\)

```
G.BC = 0; // Initialize

foreach (s: G.Nodes) {
    // temporary values per Node
    Node\_Property<Float>(G) sigma;
    Node\_Property<Float>(G) delta;
    G.sigma = 0; // Initialize
    G.delta = 0;
    s.sigma = 1;

    // BFS order iteration from s
    InBFS(v: G.Nodes From s) {
        v.sigma = // Summing over BFS parents
        Sum (w:v.UpNbrs) \{ w.sigma; \}
    }

    // Reverse-BFS order iteration to s
    InRDFS(v: G.Nodes To s)(v!=s) {
        v.delta = // Summing over BFS children
        Sum (w:v.DownNbrs) {
            v.sigma / w.sigma * (1 + w.delta); 
        }
    }

    v.BC += v.delta @ s; // accumulate BC
}
```
DSL Approach: Benefits

Three benefits
- Productivity
- Portability
- Performance
Productivity Benefits

- A common limiting resource in software development is your brain power (i.e. how long can you focus?)

A C++ implementation of BC from SNAP (a parallel graph library from GT):

\[ \approx 400 \text{ line of codes (with OpenMP)} \]

Vs. Green-Marl* LOC: 24

*Green-Marl (그린 말) means Depicted Language in Korean
Productivity Benefits

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Manual LOC</th>
<th>Green-Marl LOC</th>
<th>Source</th>
<th>Misc</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>~ 400</td>
<td>24</td>
<td>SNAP</td>
<td>C++ openMP</td>
</tr>
<tr>
<td>Vertex Cover</td>
<td>71</td>
<td>21</td>
<td>SNAP</td>
<td>C++ openMP</td>
</tr>
<tr>
<td>Conductance</td>
<td>42</td>
<td>10</td>
<td>SNAP</td>
<td>C++ openMP</td>
</tr>
<tr>
<td>Page Rank</td>
<td>75</td>
<td>15</td>
<td>http:// ..</td>
<td>C++ single thread</td>
</tr>
<tr>
<td>SCC</td>
<td>65</td>
<td>15</td>
<td>http:// ..</td>
<td>Java single thread</td>
</tr>
</tbody>
</table>

- It is more than LOC
  - Focusing on the algorithm, not its implementation
  - More intuitive, less error-prone
  - Rapidly explore many different algorithms
Portability Benefits (On-going work)

- Multiple compiler targets
- SMP back-end
- Cluster back-end (*)
  - For large instances
  - We generate codes that work on Pregel API [Malewicz et al. SIGMOD 2010]
- GPU back-end (*)
  - For small instances
  - We know some tricks [Hong et al. PPOPP 2011]
Performance Benefits

Green-Marl Code

Back-end specific optimization

Target Arch. (SMP? GPU? Distributed?)

Optimized data structure & Code template

Threading Lib, (e.g. OpenMP) Graph Data Structure

Compiler

Parsing & Checking

Arch. Independent Opt

Arch. Dependent Opt

Code Generation

Use High-level Semantic Information

Target Code (e.g. C++)
Arch-Indep-Opt: Loop Fusion

Foreach (t: G.Nodes)
  t.A = t.C + 1;
Foreach (s: G.Nodes)
  s.B = s.A + s.C;

Optimization enabled by high-level (semantic) information

C++ compiler cannot merge loops (Independence not guaranteed)

Foreach (t: G.Nodes)
  t.A = t.C + 1;
  t.B = t.A + t.C;

“set” of nodes (elems are unique)
Graph-Specific Optimization

- Adding 1 to for all Outgoing Neighbors, if my B value is positive
- Counting number of Incoming Neighbors whose B value is positive

Optimization using domain-specific property

(Why?) Reverse edges may not be available or expensive to compute
Arch-Dep-Opt: Selective Parallelization

- Flattens nested parallelism with a heuristic

```java
Foreach (t: G.Nodes) {
    Foreach (s: G.Nodes) (s.X > t.Y) {
        Foreach (r: s.Nbrs) {
            s.A += r.B;
            t.C *= s.A;
        }
        val min = t.C
    }
}
```

Compiler chooses parallel region, heuristically

```java
For (t: G.Nodes) {
    Foreach (s: G.Nodes) (s.X > t.Y) {
        For (r: s.Nbrs) {
            s.A += r.B;
            t.C *= s.A;
        }
        val min = t.C
    }
}
```

Three levels of nested parallelism + reductions

```java
For (t: G.Nodes) {
    Foreach (s: G.Nodes) (s.X > t.Y) {
        For (r: s.Nbrs) {
            s.A = s.A + r.B;
            t.C *= s.A;
        }
        val = (t.C < val) ? t.C : val;
    }
}
```

[Why?]
- Graph is large
- # core is small
- There is overhead for parallelization

Optimization enabled by both architectural and domain knowledge

Reductions became normal read & write
**Code-Gen:**

Prepare data structure for reverse BFS traversal, *only if required*.

```
InBFS(t: G.Nodes From s) {
    ... 
}
InRBFS {
    Foreach (s: t.DownNbrs) 
        ... 
}
```

Generated code saves *edges to the down-nbrs* during reverse traversal.

- **Optimization enabled by code analysis (i.e. no BFS library could do this automatically)**

Compiler detects that down-nbrs are used in reverse traversal.

- **Generated code can iterate only *edges to down-nbrs* during reverse traversal**

```
// Preparation of BFS ...

// Forward BFS (generated)
{
    ... 
    // k is an out-edge of s
    for(k ... )
        node_t child = get_node(k);
        if (!is_not_visited(child)) {
            ...;  // normal BFS code here
            edge_bfs_child[k] = true;
        }
    }
    ...

// Reverse BFS (generated)
{
    ... 
    // k is an out-edge of s
    for(k ... ) {
        if (!edge_bfs_child[k]) continue;
        ...
    }
} 
```
Code-Gen: Code Templates

- **Data Structure**
  - Graph: similar to a conventional graph library
  - Collections: custom implementation

- **Code Generation Template**
  - BFS
    - Hong et al. PACT 2011 (for CPU and GPU)
    - Better implementations coming; can be adapted transparently
  - DFS
    - Inherently sequential

Compiler takes any benefits that a (template) library would give, as well
Experimental Results

- **Betweenness Centrality Implementation**
  (1) [Bader and Madduri ICPP 2006]
  (2) [Madduri et al. IPDPS 2009]
    - Apply some new optimizations
    - Performance improved over (1) ~ x2.3 on Cray XMT
  - Parallel implementation available in SNAP library based on (1) not (2) (for x86)

- **Our Experiment**
  - Start from DSL description (as shown previously)
  - Let the compiler apply the optimizations in (2), **automatically**.
Experimental Results

Effects of other optimizations
- Flipping Edges
- Saving BFS children

Nehalem (8 cores x 2HT), 32M nodes, 256M edges
(two different synthetic graphs)

Better single thread performance:
1. Efficient BFS code
2. No unnecessary locks

Parallel performance difference

Shows speed up over Baseline: SNAP (single thread)
Other Results

Compiler generated code performs as good as hand-tuned code through high-level optimizations.

Conductance
- Perf similar to manual impl.
  - Loop Fusion
  - Privitization

Vertex Cover
- Original code ➔ data race;
  - Naïve correction (omp_critical) ➔ serialization

- Test and Test-set
- Privitization
Automatic parallelization as much as exposed data parallelism (i.e. there is no black magic)

PageRank

Compare against Seq. Impl

Strongly Connected Component

DFS + BFS: Max Speed-up is 2 (Amdahl's Law)
Conclusion

- Green-Marl
  - A DSL designed for graph analysis
- Three benefits
  - Productivity
  - Performance
  - Portability (soon)

- Project page: ppl.stanford.edu/main/green_marl.html
- GitHub repository: github.com/stanford-ppl/Green-marl