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

- "What would be the avg. hop-distance between any two (Australian) actors?"

- "Is he a central figure in the movie network? How much?"

- "Do these actors work together more frequently than others?"
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, ...
New Issue: Implementation Overhead

- 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)

Intuitive Description of a graph algorithm

(1) Inherent data-parallelism
(2) Good impl. templates
(3) High-level optimization

Efficient (parallel) Implementation of the given algorithm

DSL

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

.....

Edgeset

Foreach

BFS

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

Target Language (e.g. C++)

Source-to-Source Translation
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.

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

[Image source: Wikipedia]
Betweenness Centrality

Brandes 2001

Example: Betweenness Centrality

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

Looks complex

Parallel Iteration

Parallel Assignment

Parallel BFS

Reduction

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

Looks complex

Parallel Iteration

Parallel Assignment

Parallel BFS

Reduction
DSL Approach: Benefits

- Three benefits
  - Productivity
  - Portability
  - Performance
Productivity Benefits

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

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

≈ 400 line of codes (with OpenMP)

Vs. Green-Marl* LOC: 24

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

- It is more than LOC
  - Focusing on the algorithm, not its implementation
  - More intuitive, less error-prone
  - Rapidly explore many different algorithms

<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>
Portability Benefits

- 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

Compiler

Parsing & Checking

Arch. Independent Opt

Arch. Dependent Opt

Code Generation

Target Code (e.g., C++)

Back-end specific optimization

Optimized data structure & Code template

Target Arch. (SMP? GPU? Distributed?)

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

Use High-level Semantic Information
Arch-Indep-Opt: Loop Fusion

Optimization enabled by high-level (semantic) information

Syntactic sugars may create a lot of independent loops

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

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

G.A = G.C + 1; // Group Assignment
G.B = G.A + G.C; // (vector-like operation)
```

```cpp
Foreach (t: G.Nodes) {
    t.A = t.C + 1;
    t.B = t.A + t.C;
}
```
Arch-Indep-Opt: Flipping Edges

Graph-Specific Optimization

- Counting number of incoming neighbors whose B value is positive
- Adding 1 to for all outgoing neighbors, if my 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

```
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 = (t.C < val) ? t.C : val;
}
```
Prepare data structure for reverse BFS traversal during forward traversal, only if required.

**InBFS** (*t: G.Nodes From s*)

...  

**InRBFS**

Foreach (*s: t.DownNbrs*)

...  

// Preperation of BFS...

// Forward BFS (generated)

...  

// k is an out-edge of s

for (*k ... *)

if (!edge_bfs_child[*k*]) continue;

...  

// Reverse BFS (generated)

...  

// k is an out-edge of s

for (*k ... *)

if (!edge_bfs_child[*k*]) continue;

...
Code-Gen: Reduction

- Reduction to Scalar ➔ Privatization

```c++
// reduction by minimum
Foreach(t: G.Nodes)
x min= t.A;
```

```c++
// C++ OpenMP Implementation
#pragma omp parallel
{
  // Privatization
  int x_prv = x;
  #pragma omp for
  for(t=G.begin();...
    x_prv = min(x_prv, A[t]);

  // Test and Test - set
  if(x_prv < x) {
    bool success = false;
    while(!success) {
      if(x >= x_prv) break;
      success = CAS(x, x_prv);
    }
  }
}
```
Code-Gen: Code Templates

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

Generated code also benefits from optimized libraries

- Hong et al. PACT 2011 (for CPU and GPU)
- Better implementations coming; can be adapted transparently

- DFS
  - Inherently sequential
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) \(\sim x2.3\) 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

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

Effects of other optimizations:
• Flipping Edges
• Saving BFS children

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

(a) RMAT

Speed up

Num threads

SNAP
GreenMarl
NoFlipBe
NoSaveCh, NoFlipBe

Baseline: SNAP (single thread)

Shows speed up over Baseline: SNAP (single thread)

Better single thread performance:
(1) Efficient BFS code
(2) No unnecessary locks
Other Results

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

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

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

• Test and Test-set
• Privitization
Other Results

PageRank

Strongly Connected Component

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

Parallelism is still limited by Amdahl's Law

Compare against Seq. Impl
Usage Model

“Do you expect me to re-write my whole application with your DSL?”

- No. Our src-to-src translation does not demand it.
- Okay, maybe a little glue code
“Can I still use my custom library inside DSL?”

- Yes, via **foreign syntax**
  - Similar to _asm_ mechanism in gcc
  - Statements inside []
    - Compiler simply keeps the text as-is in the generated code
  - Just tell the compiler what are being read/mutated.

```plaintext
Procedure foo(x: Int, U: $User_Type) {
    ......    // Read-Set: x and U
    ......    // Write-Set: x
    [C_function($x, $U.get_val()) ]::[x];
    ......}
```

Any foreign (e.g. C++) statement inside []
Hand-tuned Codes

“I, as an expert, can create faster code by hand-tuning.”

- Yes, I’m sure you can
  - DSL will be more helpful to non-experts. (Productivity)
- DSL enables rapid exploration of different algorithms
- You can manually enhance compiler-generated code
  - Compiler output is fairly human-readable C++ code
- DSL also provides portability
What about debugging?

- Yes, another good question.
- Currently, we’re now relying on debugging at generated C++ code level.
  - I.e. you can use gdb.
  - This is no harder than you’re using a graph library (in theory).
  - Generated output is human readable.
- The compiler does not make mistakes.
  - The compiler can dump out the intermediate results (in Green-Marl syntax).
- We also plan to implement an ‘interpreter’ environment.
  - Will look like a MATLAB for graph.
Tracing the Compiler’s Work

Verbose = on
Stop after Stage 2.

Sums are expanded into loops

Loops are merged
Portability – Different Backends

- Different back-ends of Green-Marl
  - Cache-coherent shared memory: current
  - Pregel (Distributed Environment): on-going
  - Cray XMT: early investment
  - GPU: early investment
  - GraphLab (a different run-time): idea brainstorming
  - Custom hardware: idea brainstorming
  - RamCloud: idea brainstorming
Capacity Issue in Graph Analysis

- Large graph + Associated data
  - ≥ Main Memory

- Disk-based system (i.e. virtual memory) ?
  - A lot of *random* accesses  ➔ disk latency kills you

- Stand-alone distributed program?
  - Large development overhead

- Map-Reduce (Hadoop)?
  - Unable to keep *state across iterations*  ➔ performance loss

- ➔ Pregel (or its replicates)
Pregel (from Google)

- Map-Reduce like framework with enhancement
- Iterative, Sensitive, Vertex-centric
- A vertex can maintain its associated data
- Single `compute()` function
  - Called for every vertex by the system
  - At each time step
- Framework provides APIs for neighborhood communication
  - Messages are delivered at the next time step.

```java
Int x;
Int y;
compute()
```
Implementation Issue

New Issue: Your algorithm has to be converted for Pregel API

// Count number of teen followers
// for each node(person) in a SN
Foreach(n: G.Nodes) {
    n.teenCount = Count(t:n.InNbrs)
        (t.age>=10 && t.age<20);
} // Compute average number of
// teen-followers of people of
// certain age
Float avgAgeTeenFollowers =
    Avg(n:G.Nodes)(n.age>K)
    {n.teenCnt};

Your algorithm

class foo extends … {
    …
    public void compute(...){
        if (step == 1) {
            if (this.age >= 10 &&
                this.age <= 20)
                sendNeighbors (new IntMessage(1));
        }
        else if (step == 2) {
            this.teenCount = 0;
            for(r: getReceived())
                this.teenCount +=
                    r.IntValue();
        }
        else if (step == 3) {
            if (this.age > K)
                ….
        }
    }
}

Some global-scoped sequential computation

Based on random reading

Need context management

Need boilerplate code

Message Sending

Message Receiving

Message is always pushed, not pulled

need some tricks for global computation

Pregel Implementation

Automatic Translation?
Issues to be solved

- Sequential computation
- Globally scoped variables
- Management of Execution Context
- Communication (message sending/receiving)
- Enforcing Push-based messaging

......
Our framework

- Pregel (from Google) is not open to public.
- GPS: an implementation of Pregel from Stanford, with Semih Salihoglu
- With enhancements
  - Optimized for performance
    - x5~10 faster than Giraph (a popular Pregel implementation from Yahoo/Apache)
  - Elegant API for *global* objects and sequential computation
Handling Sequential Portion

- Your algorithm may include sequential portion
  - E.g. termination based on global sum of difference in page rank algorithm
- GPS provides a nice API for this:
  - master class, master.compute()

Sequential (global) computation

Parallel (vertex-wise) computation

Alternating execution
Globally shared variables

Another useful API: Global object map

Master puts an value object to the map

Map is cleared at the end of each computation step

The object is broadcast to every vertices at following vertex-compute()
Compiler Translation: Global Object Management

Procedure foo(age, teenCnt:N_P<Int>, K: Int) {
    ...
    Int S=0; // globally scoped
    Foreach (n:G.Nodes)
        If (n.age>K)
            S += n.teenCnt;
    ...
    class master {
        int S;
        int K;
        void compute() { ...
            S = 0; ...
            global.put("K", new IntVal(K));
            ...
            S+= global.get("S").intval();
            ...
        }
    }

    class vertex {
        int age;
        int teenCnt;
        void compute() {
            ...
            int K=
                global.get("K").intval();
            if (this.age > K){
                global.put("S", new IntSumVal(this.teenCnt);
            }
            ...
        }
    }

    Compiler knows when the variable is used
    master copy of global variables
    Reduction is implemented via special API

    Node property
Compiler Translation: Execution Context & Sequential Portion

Foreach (n: G.Nodes) {
    n.teenCnt = ...  
}

Int S=0;
Foreach (n:G.Nodes) {
    If (n.age>K)
        S += n.teenCnt;
}

class master {
    int _state;
    void compute() {
        switch(_state) {
            case 1: do_state_1();
            ...
        }
    }
    void do_state_3() {
        int K= ...
        if (this.age > K)
            ...
    }
}

class vertex {
    ...
    void compute(..) {
        int _state =
            global.get("_state")
            .intVal();
        switch(_state) {
            case 1: ..
            ...
        }
    }
}

Compiler can figure out phases of algorithm

Compiler generates state-machine at master

Current state is broadcast to vertices
Nested loop implies communication

Communication is split into two consecutive states: sending + receiving

Outer-loop becomes sending side

Inner-loop becomes receiving side
Enforcing Push-based algorithm

\[
\text{Foreach (n: G.Nodes)} \\
\text{Foreach (t: n.Nbrs)} \\
\quad t.X += f(t.Y, n.Z);
\]

\[
\text{Foreach (n: G.Nodes)} \\
\text{Foreach (t: n.Nbrs)} \\
\quad n.X += g(t.Y, n.Z);
\]

\[
\text{Foreach (t: G.Nodes)} \\
\text{Foreach (n: t.InNbrs)} \\
\quad n.X += g(t.Y, n.Z);
\]

This nested loop is a push.

For every n, push n.Y to out-neighbor t to update t.X

This nested loop is a pull. (cannot be implemented with API)

For every n, pull t.Y from out-neighbor t to update n.X

Compiler transforms it into push by flipping edges

For every t, push t.Y to in-neighbor n to update n.X
Node Prop<Int> _Stmp;
Foreach (n: G.Nodes)
    n._Stmp = 0;
Foreach (n: G.Nodes)
    n.teenCnt = n._Stmp;

Compiler Transformation:
Applying edge-flipping

Foreach (n: G.Nodes)
    n.teenCnt = Sum(t:n.InNbrs)(...) {1};

Compiler changes
Sum into Foreach

Foreach (n: G.Nodes)
    Int _S = 0;
    Foreach (t: n.InNbrs)(...)
        _S += 1;
    n.teenCnt = _S;

Replace scalar S
with temporary node
property Stmp

Node Prop<Int> _Stmp;
Foreach (n: G.Nodes)
    n._Stmp = 0;
Foreach (t: G.Nodes)(...)
    Foreach (n: n.Nbrs)
        n._Stmp += 1;
Foreach (n: G.Nodes)
    n.teenCnt = n._Stmp;

Split Loops

Node Prop<Int> _Stmp;
Foreach (n: G.Nodes)
    n._Stmp = 0;
Foreach (t: G.Nodes)(...)
    Foreach (n: n.Nbrs)
        n._Stmp += 1;
Foreach (n: G.Nodes)
    n.teenCnt = n._Stmp;
There are still other details …

- Defining message class
- Merging states together
- Optimizing temporary node properties
- Merging congruent message classes
- ......

➡ Current State:

➡ Can transform many algorithms into Pregel
➡ Compiler-generated code exhibits little overhead compared to hand-written code
➡ Still improving.
Conclusion

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

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

Questions?

“Programs must be written for people to read, and only incidentally for machines to execute.”

-- Abelson & Sussman
Language Features

- For graph analysis
  - Built-in data types
  - Node and edge property
  - Collections
  - Graph iteration and traversal

- For parallel and distributed execution
  - Implicit parallelism
  - Consistency Model
  - Reduction

- For extensibility
  - Embedded foreign syntax
Types and Properties

- Green-Marl is statically-typed languages
  - Primitive types
  - Graphs (directed, undirected),
  - Node/Edge, Node/Edge properties
  - Collections
  - Foreign types (later)

```plaintext
Procedure foo(G: Graph, // Graph
    s: Node(G), // Node of G
    A,B: Node_Typ<Int>(G), // Node Property of G
    C: Edge_Typ<Float>(G))
{
    // Property definition inside a scope
    Node_Typ<Int>(G) T;
    ...
}
```
Types and Properties

- **Node/Edge**
  - **Node**(graph)
  - Bound to a graph instance

- **Node/Edge Property**
  - **Node Prop< prim_type >**(graph)

- **Collection Types**
  - **Node_Set**(graph)
  - **Node_Order**(graph)
  - **Node_Seq**(graph)
  - **Node_Multiset**(graph)

---

```cpp
Graph G1, G2;
Node(G1) n;
Node(G2) m;
n = m; // type error
```

<table>
<thead>
<tr>
<th></th>
<th>Uniqueness</th>
<th>Ordered-ness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Order</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Sequence</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Multiset</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Graph Iteration and Traversal

- **Graph Iteration**

  ```
  Foreach(n : G.Nodes) (n.A > 0)
  ...
  ```

  For ➔ Sequential consistency
  Foreach ➔ Parallel consistency

- **Graph Traversal**

  ```
  InBFS(n : G.Nodes From r) (n.A > 0) [n.color == 0]
  {...}
  ```

  InDFS ➔ Depth-First Search Order
  InBFS ➔ Breadth-First Search Order
  InRDFS, InRBFS ➔ Reverse order traversal

  Filter; do not execute body if false

  Navigator; do not go further if false

- **Iterator and Range**

  Graph. Nodes/Edges
  Node. Nbrs/InNbrs/OutNbrs (UpNbrs/DownNbrs) ...
  Set. Items
Implicit Parallelism

- Parallel assignment
- Reduction expression

```
Graph G;
Node_Prop<Int>(G) x, y;

// parallel assignment
G.x = G.y + 1;
// Reduction (expression form)
Int z = Sum (t: G.Nodes) {t.x};

Foreach (n: G.Nodes)
  n.x = n.y + 1;

Int z = 0;
Foreach (t: G.Nodes)
  z += t.x; // Reduction (assignment form)
```

They are Syntax sugars
Consistency Model

- Sequential Consistency (For)
- Parallel Consistency (Foreach)
  - Things happen in parallel ...
  - No ordering is guaranteed between concurrent loops
  - No visibility is guaranteed between concurrent loops
  - Use reductions!

```java
Foreach (s: G.Nodes) {
    Foreach (t: s.Nbrs) {
        // Error (Warning)
        // (w-w conflict) multiple s can write to the same t.A
        // (r-w conflict) t.A can be read and written by different s.
        t.A = t.A + s.B*2;
    }
}
```

```java
Foreach (s: G.Nodes) {
    Foreach (t: s.Nbrs) {
        // But compiler understands reduction
        t.A += s.B*2;
    }
}
```
Reductions

- **Assignment Form**

```plaintext
Int z = 0;
Foreach (n : G.Nodes)
    z += n.X;
```

- **Expression Form**

```plaintext
Int z = \textbf{Sum}(n:G.Nodes)\{n.X\};
```

- **Argmax/Argmin**

```plaintext
Int x, z;
Node (G) m;
Foreach (n : G.Nodes)
    z <x, m> \textbf{max}= f(n.A) + n.B <f(n.A), n> ;
```

z: Max
x, m: Argmax
Bulk Synchronous Consistency

- Deferred assignment

```java
Foreach (s: G.Nodes) {
    // Reading t.A gives ‘old’ value
    s.A <= Sum (t: s.Nbrs) {t.A} @ s;
}
// modification to property A becomes visible at the end of s-loop
```

Loop bound indicator: tells to which loop this assignment is bound.
(e.g. nested loop)
A note on parallel/sequential consistency and parallel execution

- The compiler (runtime) may execute a `foreach` loop sequentially.
- The compiler (runtime) may execute a `for` loop in parallel, as long as it can guarantee sequential consistency.
  - E.g. transactional memory or locks
Data Access Analysis

Procedure `foo` (\(G:\text{Graph}, \ A, B: \text{N}_P\text{Int}(G); \ Z:\text{INT}\))
{
    \(\text{Int} \ Y = 0;\)
    \(\text{Foreach}(x: \ G.\text{Nodes})\)
    {
        \(\text{If}(x.B > 3)\)
        \(Y += x.A;\)
    }
    \(Z = Y;\)
}
I’m not a graph guy. Do you suggest that I create my own DSL?

- Yes, I encourage you.
- Green-Marl is a stand-alone DSL, created from the scratch
  - This paper is written with 3 of my managers.
  - Current compiler was implemented in less than 6 months.
  - It is a doable job: (1) Type checker is simple. (2) Code generation is also not very complicated as you emit C++ code
  - Designing a good language is challenging, though.

- There are easier ways, though.
  - Innovations in Embedded DSL
  - Delite [H. Chafi et al., PPoPP’11] ➔ a framework for DSL creation
  - Green-Marl on Delite is also being developed.
Can every graph algorithm be written in Green-Marl?

- Good question. We hope so, don’t have proof.
  - We think we have all the necessary basic blocks
  - Basic node/edge iteration; graph traversal
  - Four collections (set/seq/order/bag)
  - Reductions
- Foreign syntax / Foreign type may help you
- Still, we are improving our language specification
  - We’re hearing from users, including professionals
  - Your opinion is valuable to us