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

```
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

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, Π), 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)

```
Procedure foo(G1, G2:Graph, n:Node(G1)) {
   Node(G2) n2; // a node of graph G2
   n2 = n;  // type-error (bound to different graphs)
   Node_Prop<Int>(G1) A; //integer node property for G1
   n.A = 0;
   Node_Set(G1) S; // a node set of G1
   S.Add(n);
}
```

Operations on Collections

Group	Op-Name	sequential S	sequential O	sequential Q	parallel S	parallel O	parallel Q
Grow	Add	v			v		
	Push(Front/Back)		v	v		v	v
Shrink	Remove	v			v		
	Pop(Front/Back)		v	v		?	?
	Clear	v	v	v	v	v	v
Lookup	Has	v	v	v	v	v	v
	Front(Back)		v	v		v	v
	Size	v	v	v	v	v	v
Copy	=	v	v	v	X	X	X
Iteration	Items	v	v	v	v	v	v

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

Conflicts under parallel execution →

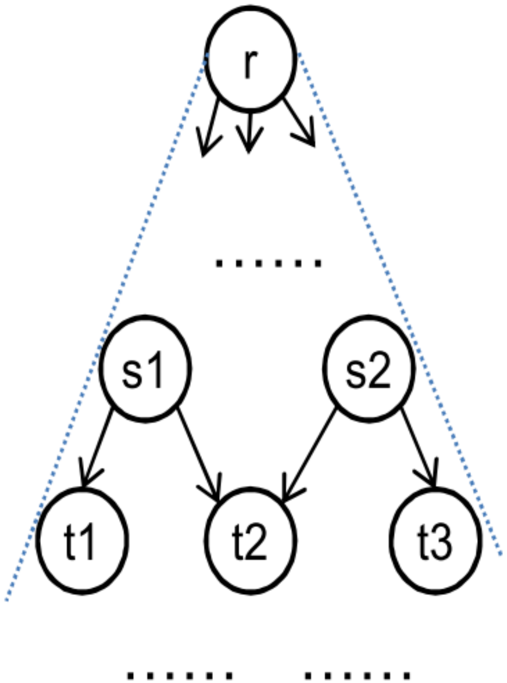
Grow-Shrink: X Lookup-Shrink: ? Lookup-Grow: ?

Iteration/Traversal

```
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



Source Type	Range	Access
D/UGraph	Nodes	Linear
Node(D/UGraph)	Nbrs	Random
Node(DGraph)	OutNbrs	Random
Node(DGraph)	InNbrs	Random
Node(D/UGraph)	UpNbrs	Random/-1
Node(D/UGraph)	DownNbrs	Random/+1
Node_Set	Items	Linear
Node_Order	Items	Linear
Node_Seq	Items	Random

Parallelism in Green-Marl

- Inspired by OpenMP
- Follows OpenMP's memory consistency model
  - Writing to same shared variable in concurrently may cause conflicts

```
Foreach(s:G.Nodes)
  Foreach(t:s.OutNbrs)
    t.A =              // write-write conflict
      t.A + s.B;       // read-write conflict
```

Reductions

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

In-place	Assignment	In-place	Assignment
All	&&=	Sum	+=
Any	||=	Product	*=
Min	min=	Count	++
Max	max=		

Overall Structure

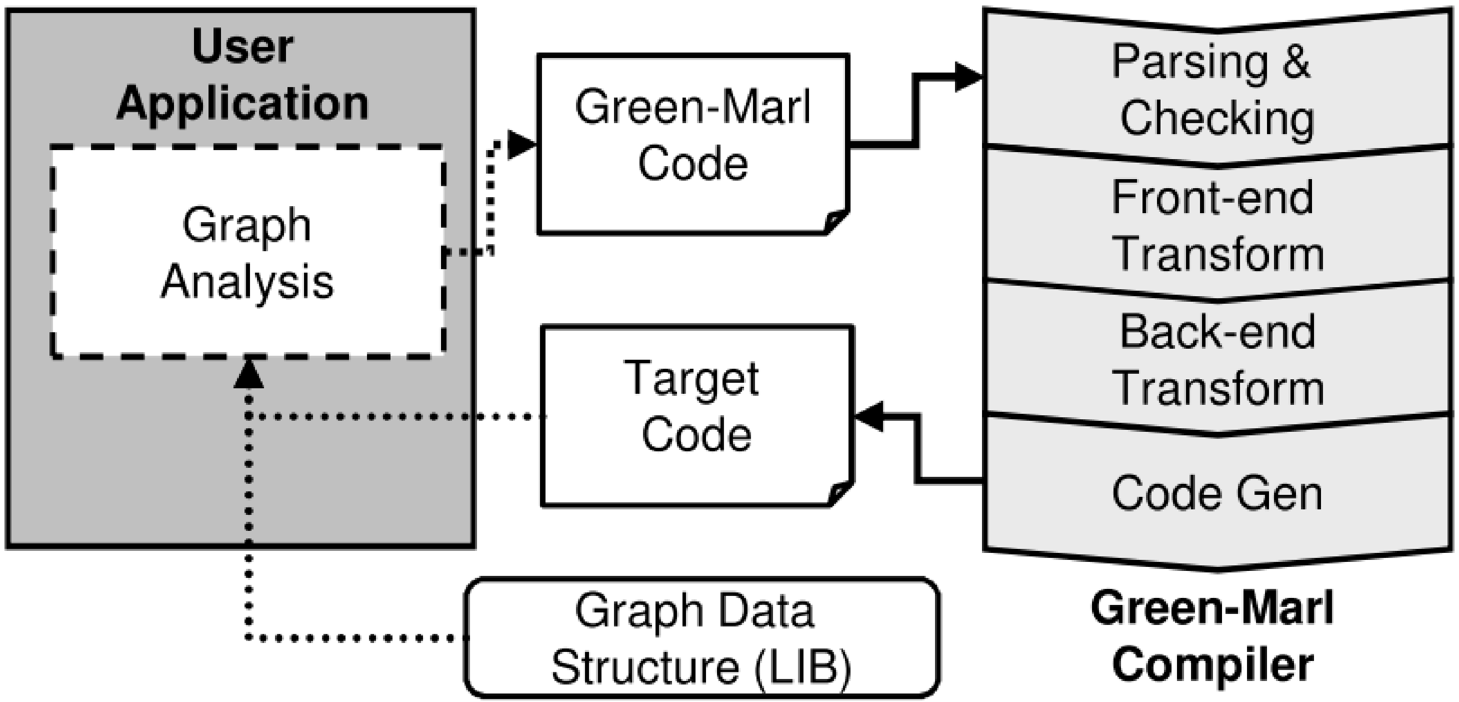


Figure 3. Overview of Green-Marl DSL-compiler Usage

Compiler Optimizations: Loop Fusion

```
Foreach(s: G.Nodes)(f(s))
    s.A = X(s.B);
Foreach(t: G.Nodes)(g(t))
    t.B = Y(t.A)
```

becomes

```
Foreach(s: G.Nodes)(
    if (f(s)) s.A = X(s.B);
    if (g(s)) s.B = Y(s.A);
}
```

Compiler Optimizations: Hoisting Definitions

```
For(s:G.Nodes) { //sequential loop
    Node_Prop<Int>(G) A;
    ...
}
```

becomes

```
Node_Prop<Int>(G) A;
For(s:G.Nodes) {
    ...
}
```

Compiler Optimizations: Set-Graph Loop Fusion

```
Node_Set S(G); // ...
Foreach(s: S.Items)
    s.A = x(s.B);
Foreach(t: G.Nodes)(g(t))
    t.B = y(t.A)
```

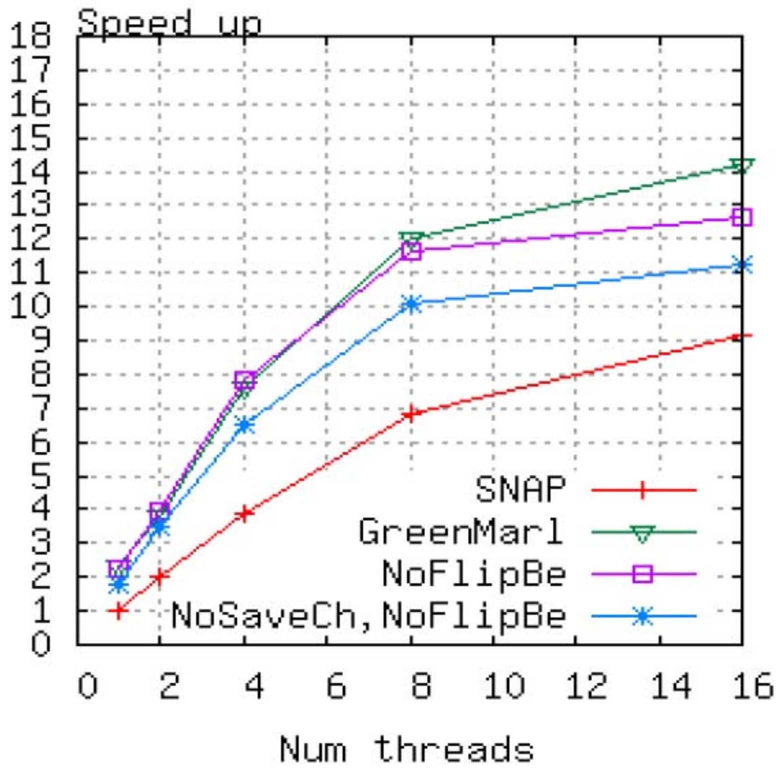
becomes

```
Foreach(s: G.Nodes)(
    if (S.Has(s)) s.A = x(s.B);
    if (g(s)) s.B = y(s.A);
}
```

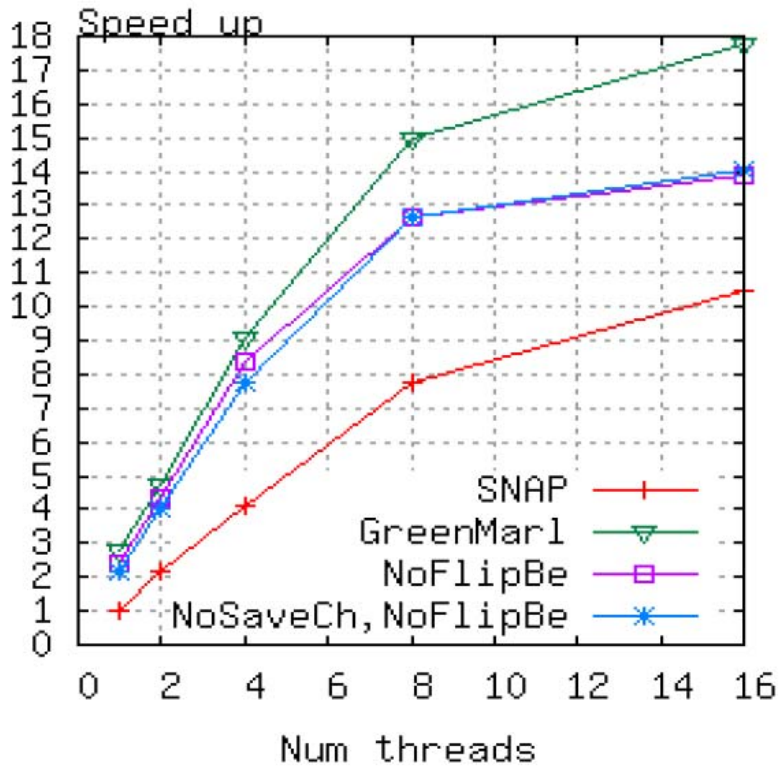
Evaluation (LOC)

Name	LOC Original	LOC Green-Marl	Source
BC	350	24	[9] (C OpenMp)
Conductance	42	10	[9] (C OpenMp)
Vetex Cover	71	25	[9] (C OpenMp)
PageRank	58	15	[2] (C++, sequential)
SCC(Kosaraju)	80	15	[3] (Java, sequential)

Evaluation (BC)

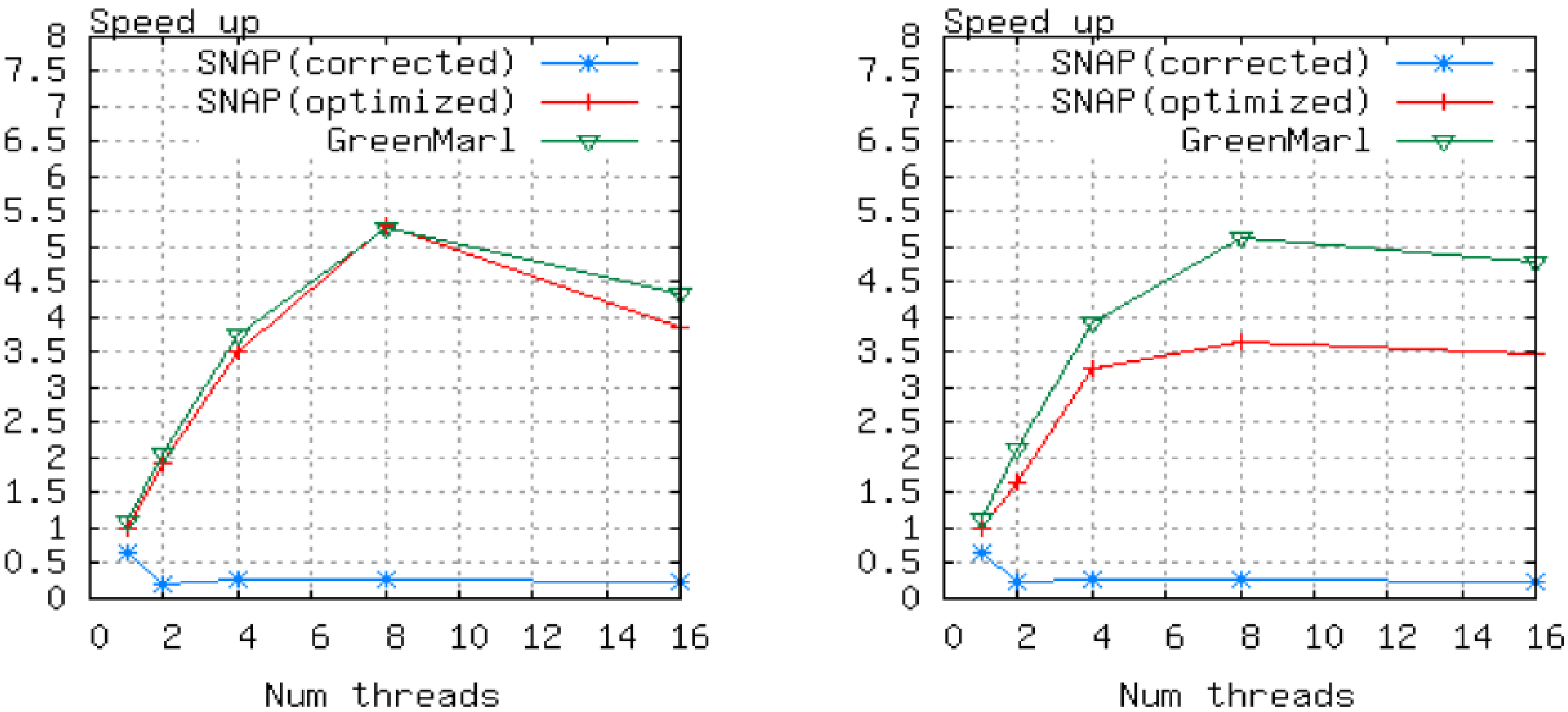


(a) RMAT



(b) Uniform

Evaluation (Vertex Cover)



(a) RMAT (b) Uniform

My Impressions

- Syntax is galling

```
//-----------------------------------------------------------------------------------------------------------
// 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 {                                                                       // Main Iteration.
    diff = 0.0;
    Foreach (t: G.Nodes) {                                                   // Compute PR from neighbor's current PR.
       Double val = (1-d) / N +
           d* Sum(w: t.InNbrs) (w.OutDegree()>0) {w.PR / w.OutDegree()};
       t.PR <= val @ t;                                                      // Modification of PR will be visible after t-loop.
       diff += | val - t.PR |;                                               // Accumulate difference (t.PR is still old value)
    }
    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

		sequential			parallel		
Group	Op-Name	S	O	Q	S	O	Q
Grow	Add	v			v		
	Push(Front/Back)		v	v		v	v
Shrink	Remove	v			v		
	Pop(Front/Back)		v	v		?	?
	Clear	v	v	v	v	v	v
Lookup	Has	v	v	v	v	v	v
	Front(Back)		v	v		v	v
	Size	v	v	v	v	v	v
Copy	=	v	v	v	X	X	X
Iteration	Items	v	v	v	v	v	v

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

Conflicts under parallel execution $\rightarrow$

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