Green-Marl: A Domain-Specific Language for Easy and Efficient Graph Analysis

Oct 25th 2023
Wenxuan Li

Context & Motivation

- Context: large amount of graph data to be analysed and mined
- Challenges for efficient graph analysis on large-scale graph data:
    - Capacity: limited memory space
    - **Performance**: when run on large scale data
    - **Implementation**: difficult to implement correctly and efficiently
- Main focus: tight coupling between high-level graph analysis algorithm design and underlying hardware architecture

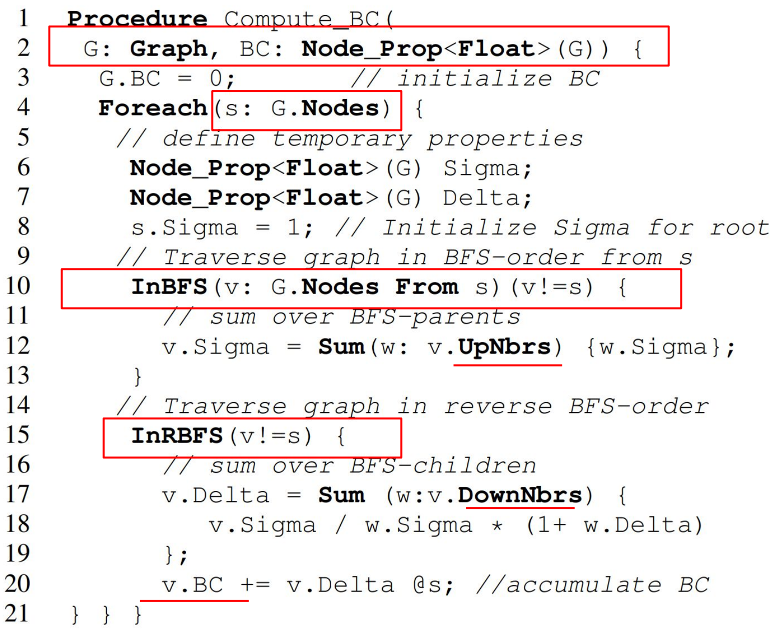
Overview

Green-Marl:

- A high-level domain specific **language**
- An associated **compiler** for producing optimized and parallelized low-level implementation

=> for both performance (optimization) and implementation (decoupling)

Language Design: Domain-specific Syntaxes

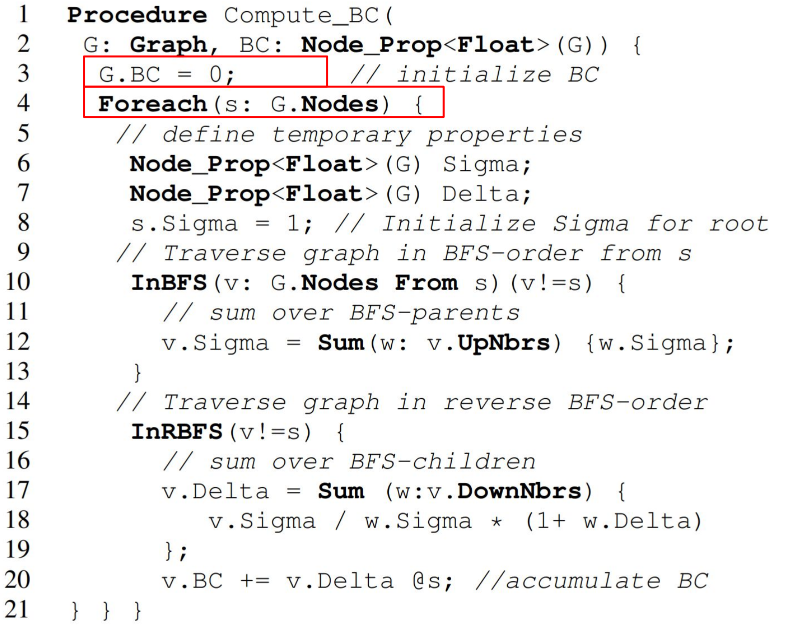


```
Procedure Compute_BC(
 G: Graph, BC: Node_Prop<Float>(G)) {
  G.BC = 0;        // initialize BC
  Foreach(s: G.Nodes) {
   // define temporary properties
    Node_Prop<Float>(G) Sigma;
    Node_Prop<Float>(G) Delta;
    s.Sigma = 1; // Initialize Sigma for root
   // Traverse graph in BFS-order from 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

- Data-Types:
  - Graph
  - Nodes
  - Node_Prop
- Traversal:
  - InBFS (BFS)
  - InRBFS (reverse-order BFS)
  - UpNbrs and DownNbrs

Language Design: Parallelism



```
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

- Implicit Parallelism
    - Group Assignment: `G.BC=0`
- (Explicit) Parallel Execution Region
    - `Foreach`
    - following fork-join style

Compiler

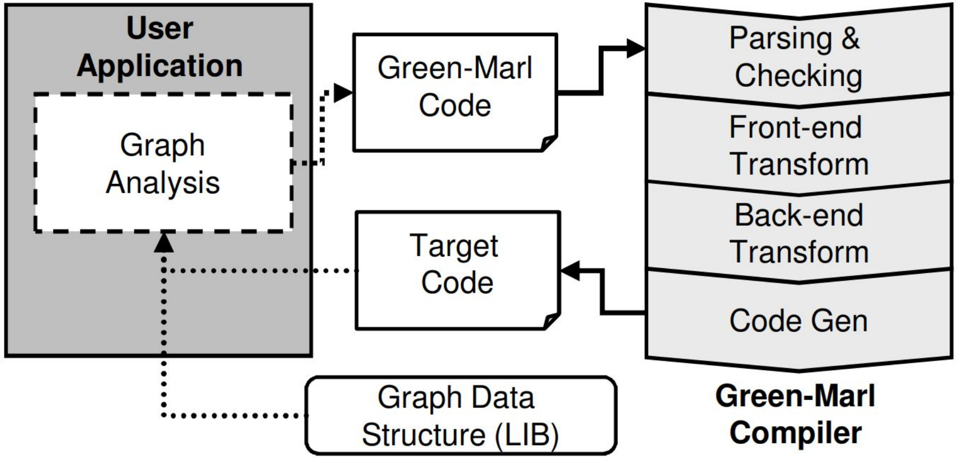
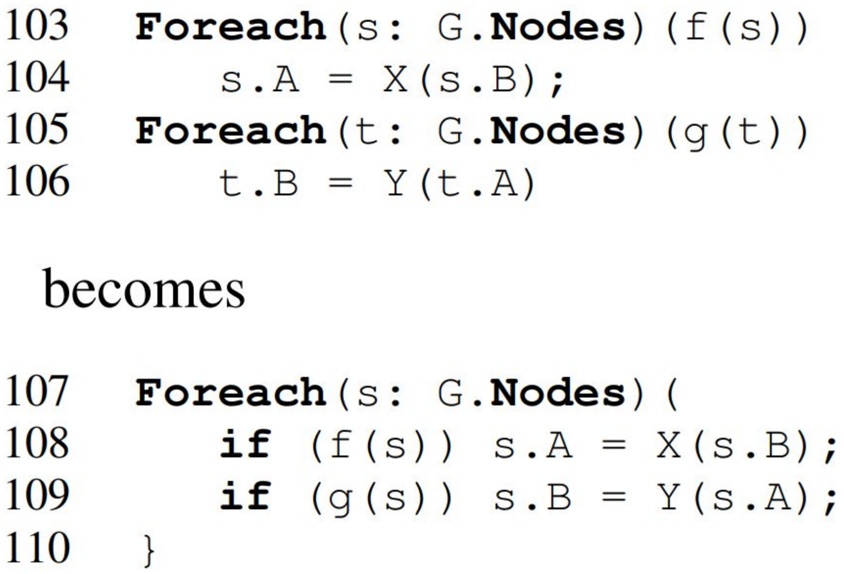


Figure 3. Overview of Green-Marl DSL-compiler Usage

Compiler: Loop Fusion

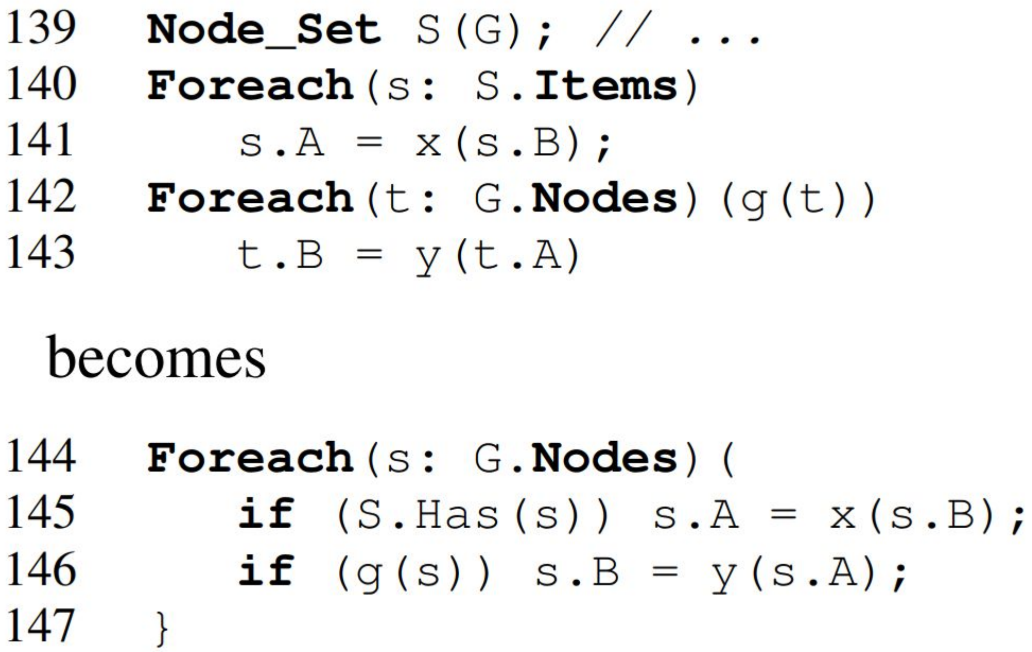


```
Foreach(s: G.Nodes)(f(s))
    s.A = X(s.B);
Foreach(t: G.Nodes)(g(t))
    t.B = Y(t.A)
```

becomes

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

Compiler: 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)
```

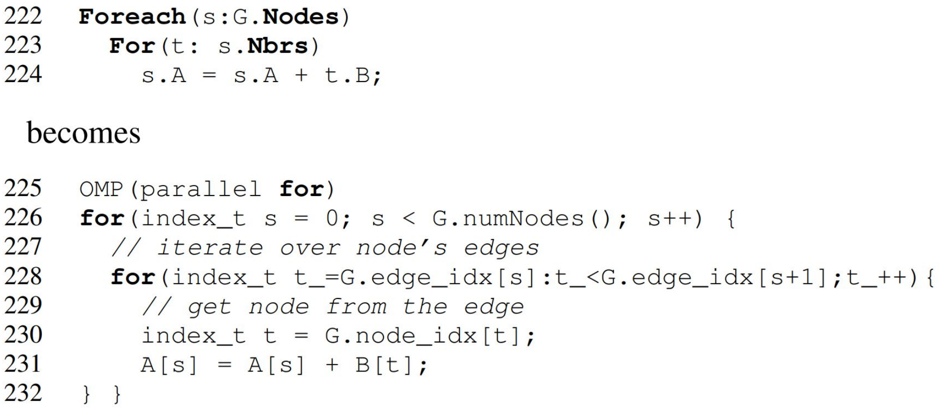
becomes

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

Compiler: Code Generation and Architecture Portability

Compiler emits out target code using code-generation templates

- Example: `Foreach` implementation with backend OpenMP



```
Foreach(s:G.Nodes)
  For(t: s.Nbrs)
    s.A = s.A + t.B;
```

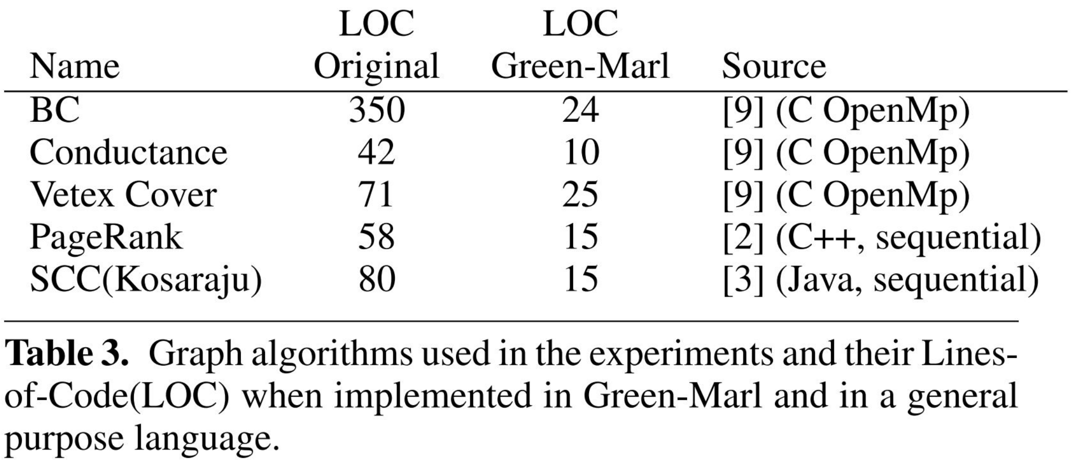
becomes

```
OMP(parallel for)
for(index_t s = 0; s < G.numNodes(); s++) {
  // iterate over node's edges
  for(index_t t_=G.edge_idx[s]:t_<G.edge_idx[s+1];t_++){
    // get node from the edge
    index_t t = G.node_idx[t];
    A[s] = A[s] + B[t];
} }
```

Allows replacement of backends for code-generations, e.g. CUDA

Evaluation: Productivity

- Measured by Line of Codes (LoC)
- Compared with implementations in existing graph analysis libraries

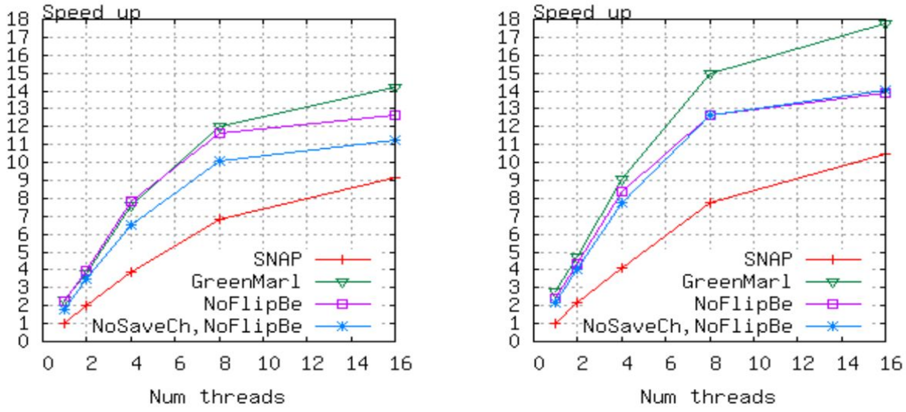


Name	LOC Original	LOC Green-Marl	Source
BC	350	24	[9] (C OpenMp)
Conductance	42	10	[9] (C OpenMp)
Vetex Cover	71	25	[9] (C OpenMp)
PageRank	58	15	[2] (C++, sequential)
SCC(Kosaraju)	80	15	[3] (Java, sequential)

Table 3. Graph algorithms used in the experiments and their Lines-of-Code(LOC) when implemented in Green-Marl and in a general purpose language.

Evaluation: Performance Gain

- Measured by Speed-up with number of threads growing
- Compared with implementations in SNAP library (Bader et al. 2008)
- Ablation study by disabling some optimizations of the compiler (e.g. FlippingEdge)



(a) RMAT (b) Uniform

Figure 4. 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) and *Saving BFS Children* (Section 3.5) optimizations respectively.

(Evaluated on randomly synthesized graphs with 32 million nodes and 256 million edges)

Limitations

- No backends supported for distributed environments when the paper was released
    - Later works introduced Pregel (Hong et al. 2014), CUDA (Shashidhar and Nasre. 2017) and MPI (Rajendran and Nandivada 2020)
- No baseline provided for evaluating speed-ups on PageRank and Kosaraju's algorithm
- Still extra cost for mastering this language

Reference

- Bader et al. 2008. SNAP, Small-world Network Analysis and Partitioning: An open-source parallel graph framework for the exploration of large-scale networks
- Hong et al. 2014. Simplifying Scalable Graph Processing with a Domain-Specific Language
- Shashidhar and Nasre 2017. LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs
- Rajendran and Nandivada 2020. DisGCo: A Compiler for Distributed Graph Analytics

Thank you