Green-Marl

A DSL for Easy and Efficient Graph Analysis

Motivation

Issues with large-scale graph analysis

- Performance
- Implementation
- Capacity

Performance Issues

- RAM latency dominates running time for large graphs
- Solved by exploiting data parallelism

Implementation Issues

Writing concurrent code is *hard*

- Race-conditions
- Deadlock
- Efficiency requires deep hardware knowledge
- Couples code to architecture

Alternative: a DSL

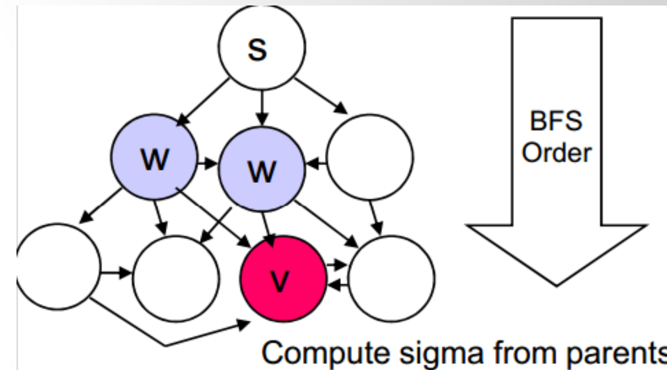
Green-Marl and its compiler

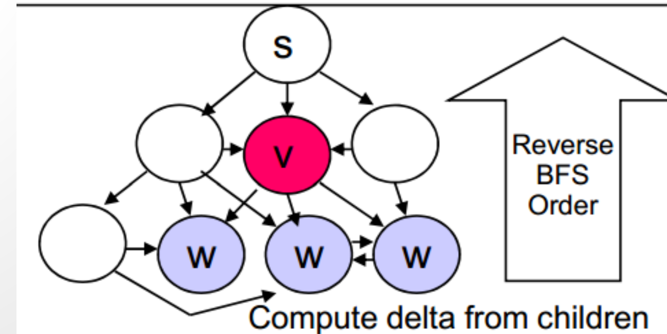
- High level graph analysis language
- Hides underlying complexity
- Exposes algorithmic concurrency
- Exploits high level domain information for optimisations

Example

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

```
Procedure Compute_BC(
    G: Graph, BC: Node_Prop<Float>(G)) {
  G.BC = 0; // initialize BC
  Foreach(s: G.Nodes) {
    Node_Prop<Float>(G) Sigma;
    Node_Prop<Float>(G) Delta;
    s.Sigma = 1; // Initialize Sigma for root
    InBFS(v: G.Nodes From s)(v!=s) {
      v.Sigma = Sum(w: v.UpNbrs) {w.Sigma};
    }
    InRBFS(v!=s) {
      v.Delta = Sum(w:v.DownNbrs) {
        v.Sigma / w.Sigma * (1+ w.Delta)
      };
      v.BC += v.Delta @s; //accumulate BC
} } }
```





Language Design

- Based on processing graph *properties*
- Mappings from a node/edge to a value
- e.g. the average number of phone calls between two people

Language Design

Green-Marl is designed to compute

- scalar values from a graph and its properties
- new properties for nodes/edges
- selecting subgraphs (instance of above)

Language Design

Support for parallelism (*fork-join* style)

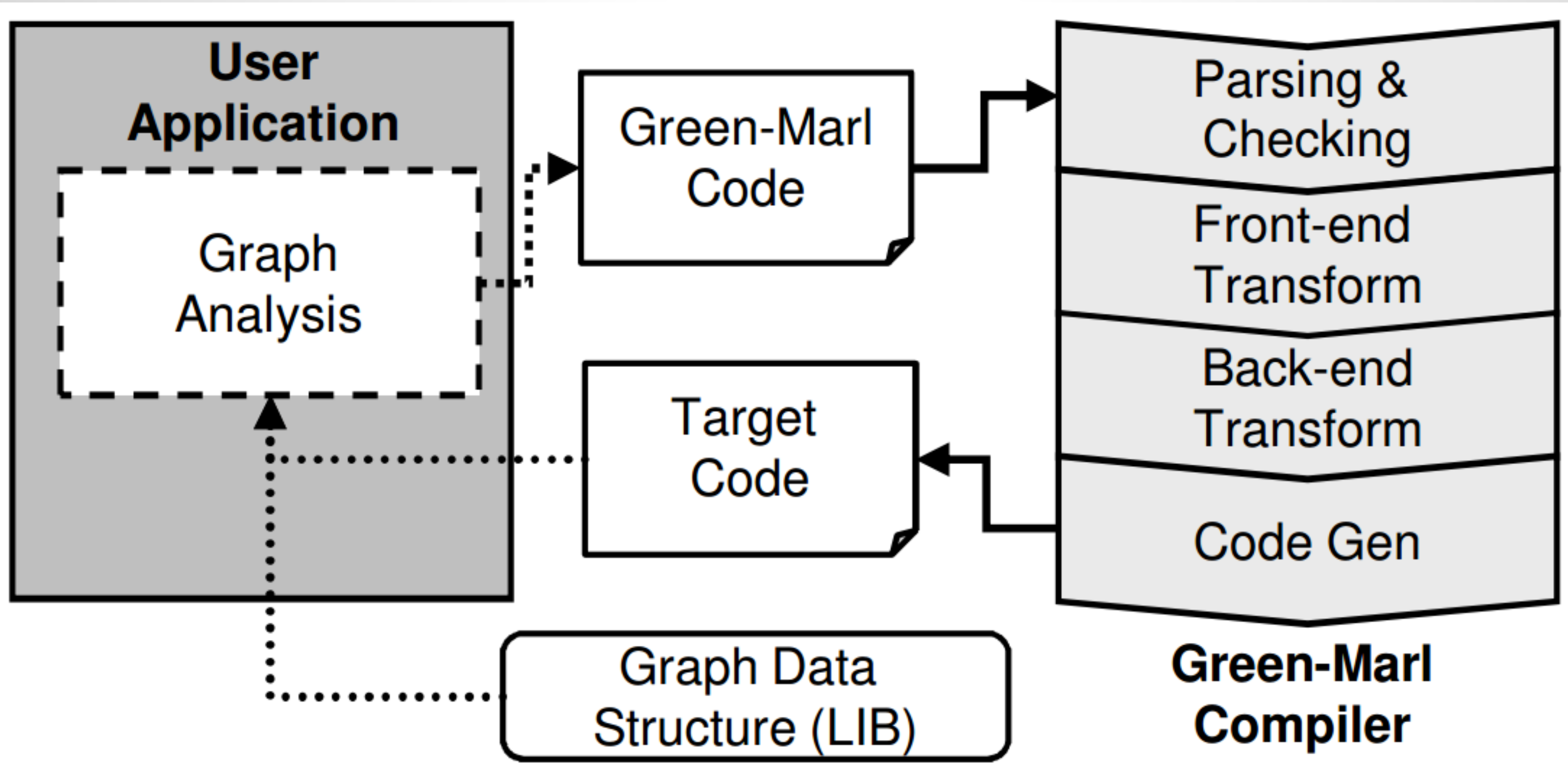
- Implicit
  - `G.BC = 0;`
- Explicit
  - `Foreach(s: G.Nodes) (s!=t)`
- Nested

Language Design

Other characteristics

- Relaxed memory model (but atomic)
- Reductions
- Built-in graph and collection types
- Built-in operations: BFS, DFS, etc.

The Compiler



The Compiler

- Currently compiles to C++
- Semantic analysis checks for conflicts in parallel sections of code
- Generic and graph-specific optimisations
  - 9 in total

The Compiler

Architecture Independent Optimisations

e.g. *Flipping Edges*

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

becomes

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

The Compiler

Architecture Dependent Optimisations

e.g. *Saving BFS Children*

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

becomes

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

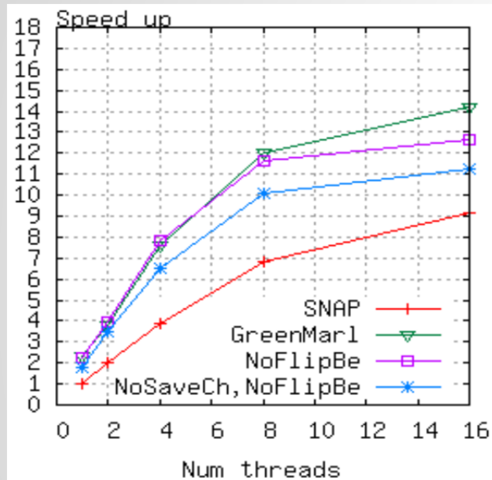
⇒

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

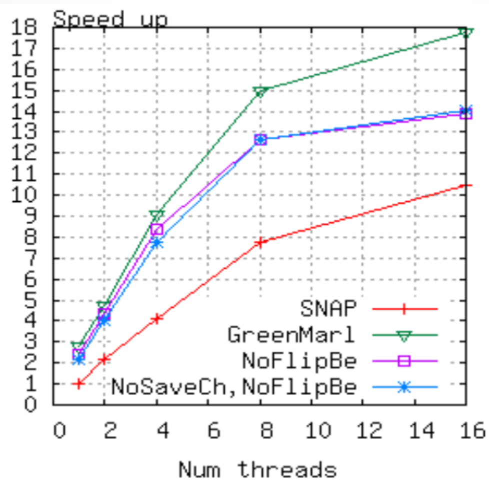
Evaluation

- 1 machine, 5 algorithms, 2 graph generators
- 32 million nodes, 256 million edges
- Compared with the SNAP graph analysis platform (only 3 algorithms)

Evaluation



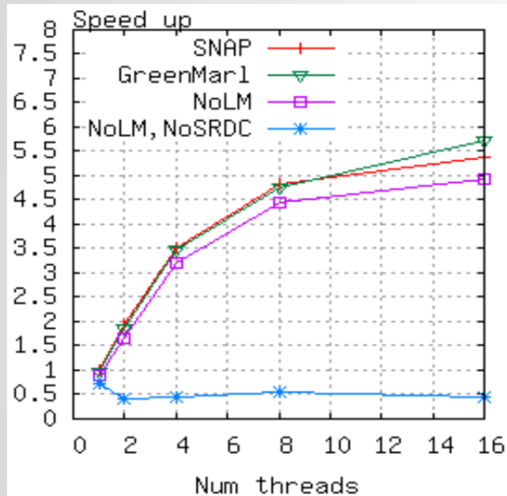
(a) RMAT



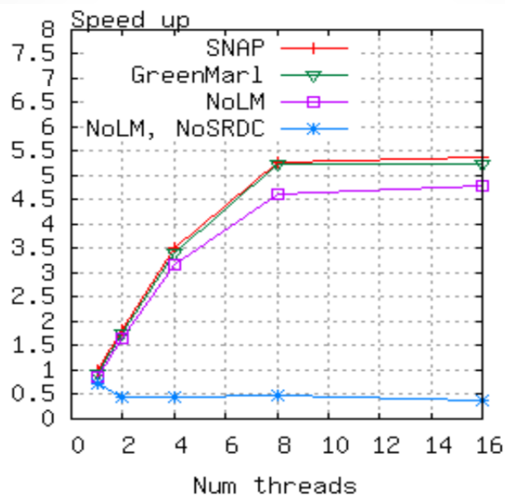
(b) Uniform

BC scaling across cores

Evaluation



(a) RMAT



(b) Uniform

Conductance

Evaluation

In a nutshell…

- At least as fast as SNAP
- Good speedup of up to ~16 threads
- Algorithms that are hard to parallelise do not scale so well (Amdahl's Law)

Evaluation

Usability

- Between 50% and 10% the lines of code of other implementations
- Does not require application rewriting
- Embedded foreign code
- Concise and intuitive descriptions of graph algorithms (in their opinion!)