Green-Marl

A DSL for Easy and Efficient Graph Analysis

S. Hong, H. Chafi, E. Sedlar, K. Olukotun [1]

LSDPO (2017/2018) Paper Presentation
Tudor Tiplea (tpt26)

Problem

- Paper identifies three major challenges in large-scale graph analysis:
  1) **Capacity** — graph won’t fit in memory
  2) **Performance** — many graph algorithms fail to perform on large graphs
  3) **Implementation** — hard to write correct and efficient graph algorithms
- Tackle last two by only focusing on **graphs that fit in memory**
- In this case, a major impediment to performance is **memory latency** (working-set size exceeds cache size)

Towards a solution

- Can improve **performance** by exploiting **data parallelism** abundant in graphs
- However, **performance** and **implementation** are **not orthogonal**
- Parallelism makes implementation more difficult
- Need to think about race conditions, deadlock, etc.
- There needs to be a **balance**

Contribution

- **Green-Marl** — A Domain-Specific Language
  - Exposes **inherent parallelism**
  - Has constructs designed specifically for easing graph algorithm implementation
  - Expressive but **concise**
- A Green-Marl **compiler**
  - Automatically **optimises** and **parallelises** the program
  - Produces **C++ code** (for now)
  - **Extendable** to target other architectures
- An evaluation of a number of graph algorithms implemented in Green-Marl claiming an **increase** in **performance** and **productivity**

The language

Overview

- Operates over **graphs** (directed or undirected) and associated **properties** (one kind of data stored in each node/edge)
- Assumes graphs are **immutable** and **no aliases** between graph instances or properties
- Given a graph and a set of properties it can compute
    - A **scalar value** (e.g. conductance of graph)
    - A **new property**
    - A **subgraph selection**
- Has **typed** data**:** primitives, nodes/edges bound to a graph, collections

```
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);
}
```

Parallelism

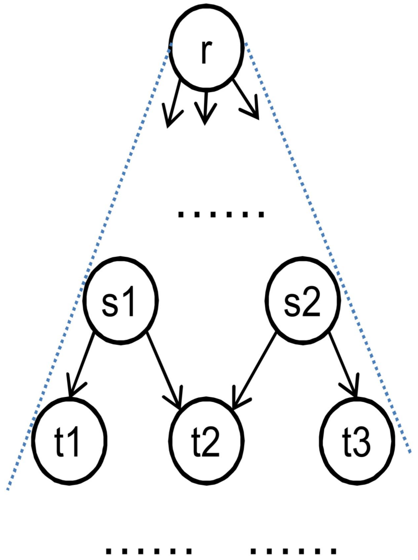
- **Group assignments** (implicit)
  - e.g. *graph_instance.property = 0*
- **Parallel regions** (explicit)
  - Uses **fork-join** parallelism
  - The compiler can detect some possible conflicts in here
- **Reductions**
  - Have **syntactic sugar** constructs
  - Can specify at which iteration scope reduction happens

```
Int sum=0;
Foreach(s: G.Nodes) {
  Int p_sum = u.A;
  Foreach(t: s.Nbrs)
     p_sum *= t.B;
  sum += p_sum;
}
Int y = sum / 2;
```

```
Int x,y;
x = Sum(t:G.Nodes){t.A};
y = 0;
Foreach(t:G.Nodes)
  y+= t.A;
```

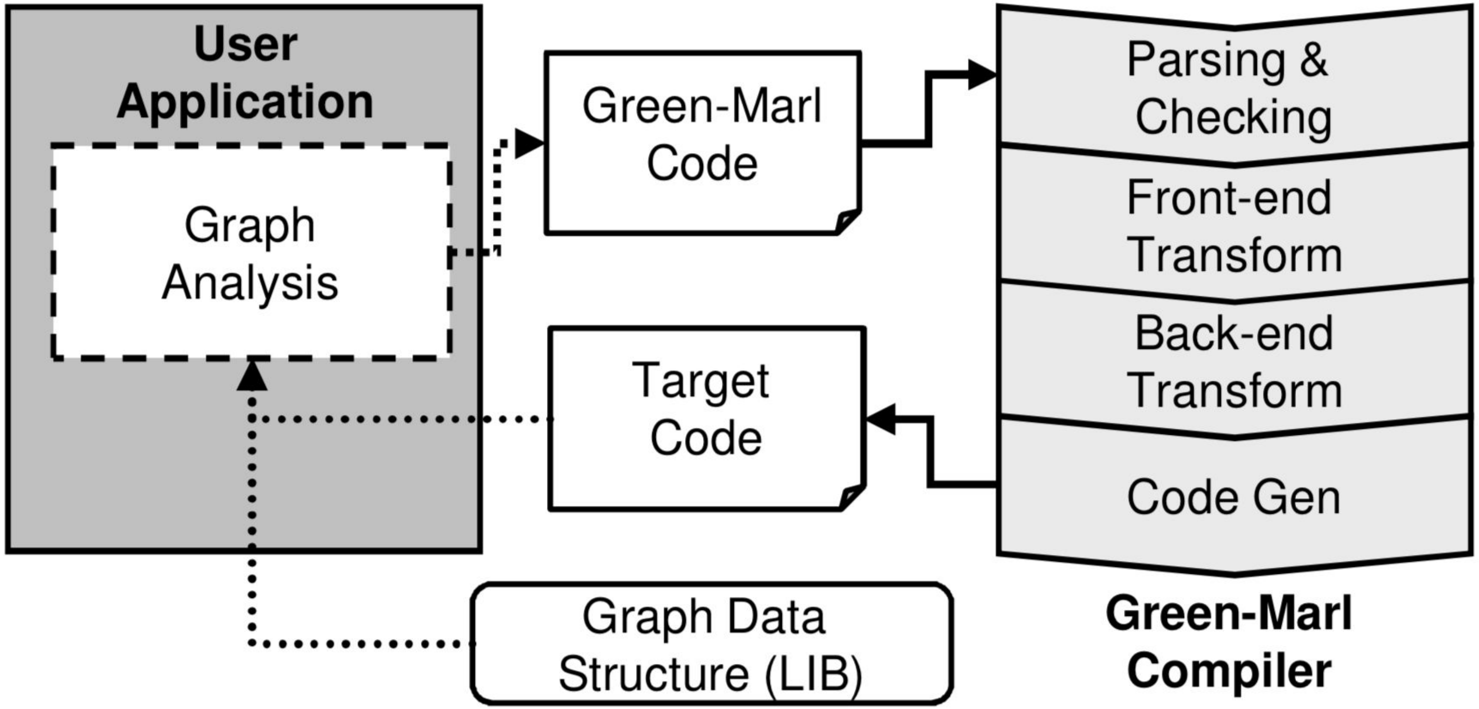
Traversals

- Can traverse graphs in either **BFS** or **DFS** order
- Each allows both a **forwards** and a **backwards** pass
- Can **prune** the search tree using a boolean navigator
- For DFS the execution is **sequential**
- BFS has **level-synchronous** execution
    - Nodes at same level can be processed in parallel
    - But parallel contexts are synchronised before next level
- During a BFS traversal each node exposes a collection of its **upwards** and **downwards** neighbours



```
InBFS (iter:src^.Nodes From root) [navigator] (filter1)
    forward_body_statement
InRBFS (filter2)
    backward_body_statement
```

The compiler


User Application
Graph Analysis
Green-Marl Code
Target Code
Graph Data Structure (LIB)
Parsing & Checking
Front-end Transform
Back-end Transform
Code Gen
Green-Marl Compiler

Structure

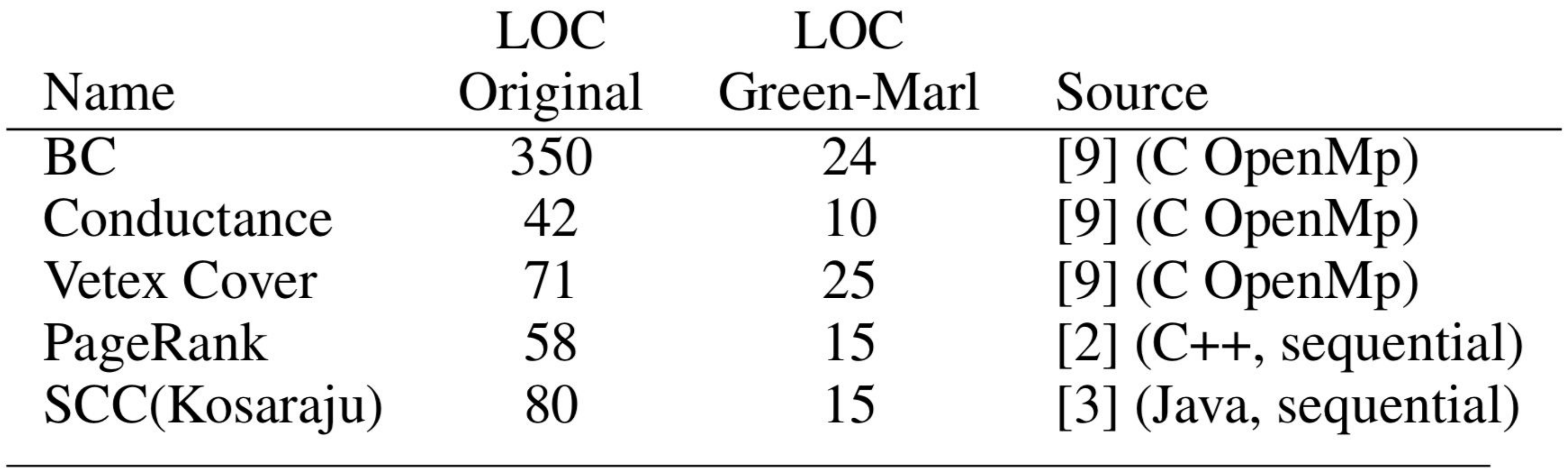
- **Parsing & checking:**
  - Can detect some data conflicts (Read-Write, Read-Reduce, Write-Reduce, Reduce-Reduce)
- **Architecture independent optimisations:**
  - Loop fusion, code hoisting, flipping edges (uses domain knowledge)
- **Architecture dependent optimisations:**
  - **NOTE:** currently the compiler only parallelises the inner-most graph-wide iteration
- **Code generation:**
  - Assumes gcc as compiler, uses OpenMP as threading library
  - Uses efficient code-generation templates for DFS and BFS

Evaluation

Methodology

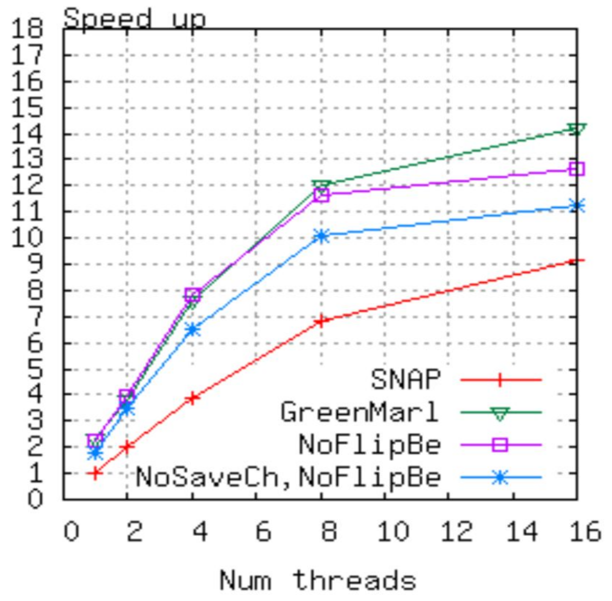
- Use **synthetically generated** graphs (generally 32 million nodes, 256 million edges):
    - **uniform** degree distribution
    - **power-law** degree distribution
- Test on a number of graph algorithms:
    - Betweenness centrality
    - Conductance
    - Vertex Cover
    - PageRank
    - Kosaraju (strongly connected components)
- Compare with implementations using the SNAP library

Productivity gains

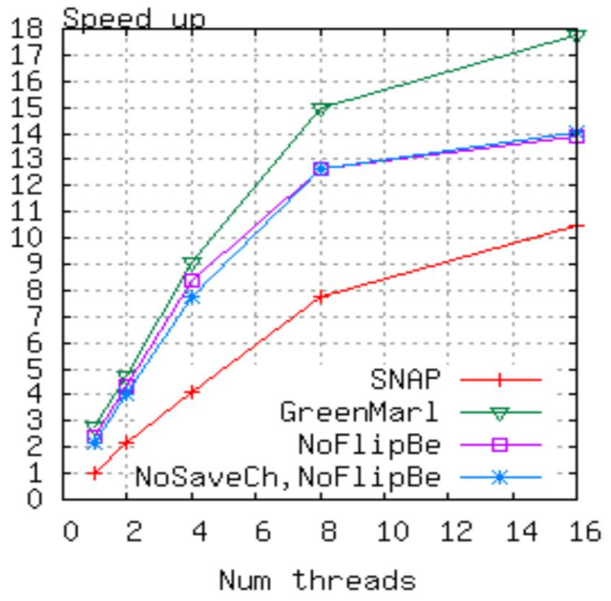


Name	LOC Original	LOC Green-Marl	Source
BC	350	24	[9] (C OpenMp)
Conductance	42	10	[9] (C OpenMp)
Vetex Cover	71	25	[9] (C OpenMp)
PageRank	58	15	[2] (C++, sequential)
SCC(Kosaraju)	80	15	[3] (Java, sequential)

Performance gains (BC)

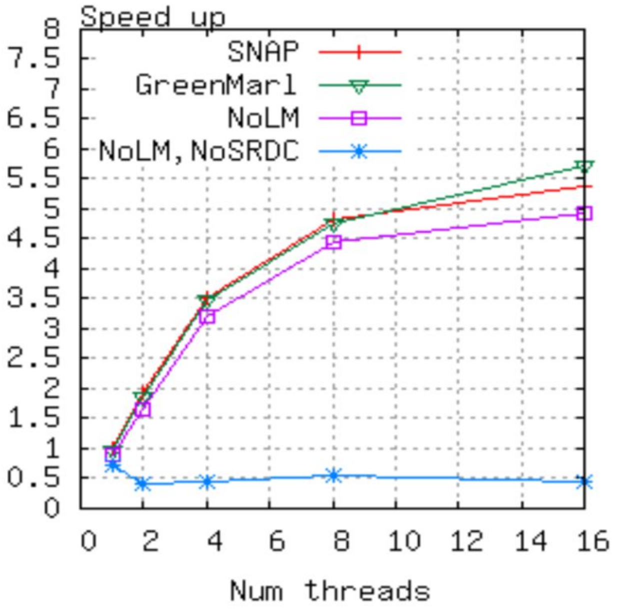


(a) RMAT

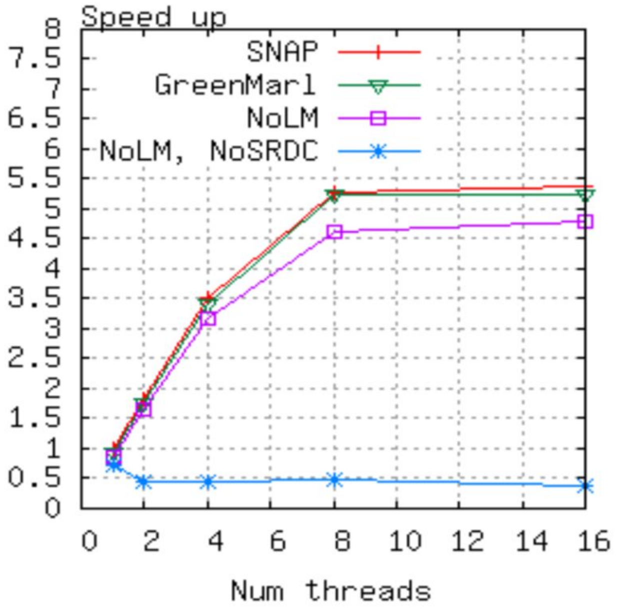


(b) Uniform

Performance gains (Conductance)



(a) RMAT



(b) Uniform

Opinion

What’s neat

- Language is easy to use
- Using a compiler means:
    - Users don’t have to worry about applying optimisations themselves
    - Programs can target multiple architectures
- Producing high-level code (like C++) means the graph analysis code can be integrated in existing applications with minimal changes
- Further work could even support out-of-memory graphs
    - E.g. compile Green-Marl to Pregel
- Or using GPUs

But...

- The ecosystem is very limited (for now, at least):
    - Cannot modify the graph structure
    - Can only compile to C++
    - Only inner-most graph-wide loops are parallelised
- Keep in mind none of the optimisations are novel
- Also, measuring productivity gains in lines of code seems very subjective and the claims should be taken with a pinch of salt

References

[1] S. Hong, H. Chafi, E. Sedlar, K.Olukotun: *Green-Marl: A DSL for Easy and Efficient Graph Analysis*, ASPLOS, 2012.

All code snippets and evaluation plots in this presentation are extracted from the paper above.

Questions

Thank you!