# Green-Marl: A DSL for Easy and Efficient Graph Analysis

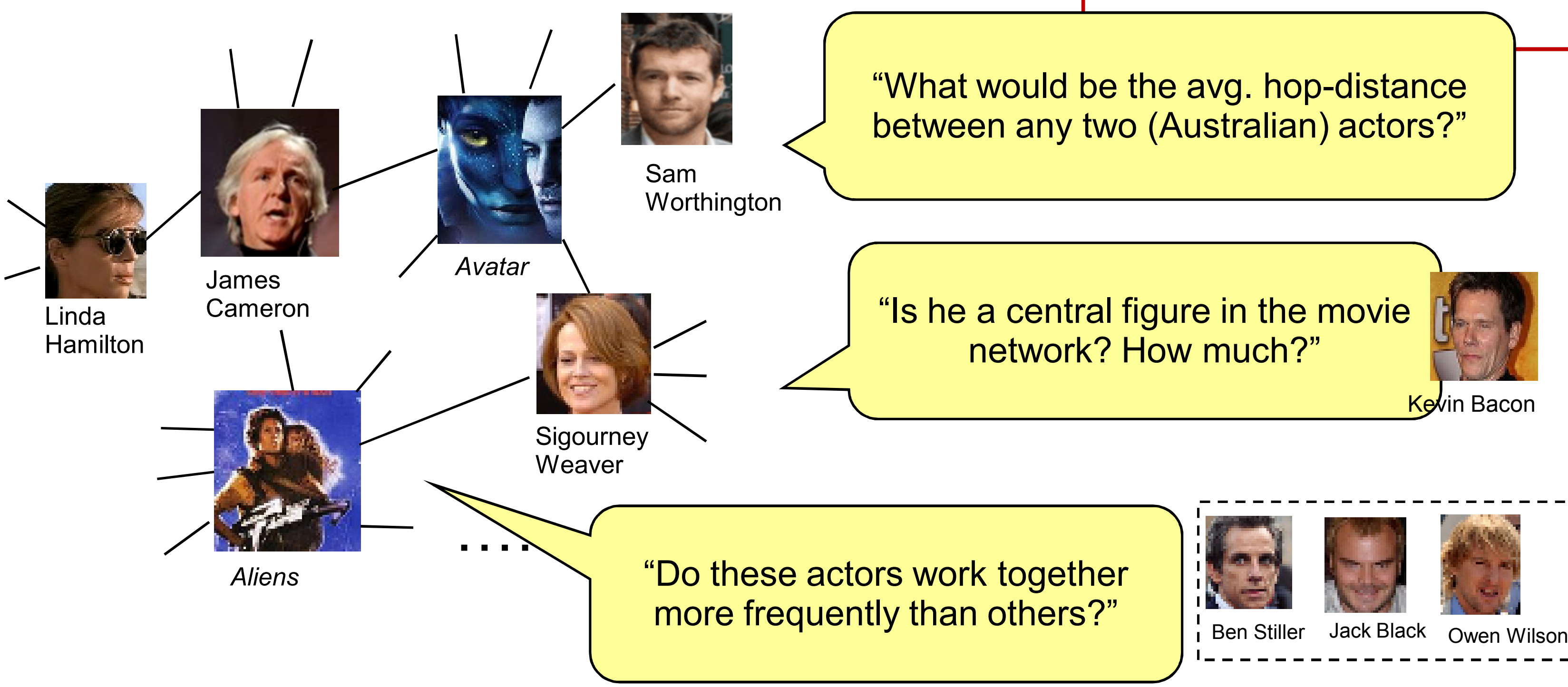
Sungpack Hong*, Hassan Chafi*+, Eric Sedlar+, and Kunle Olukotun*

*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



# 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, Π) = (G, $P_1$, $P_2$, … )
- Graph analysis on (G, Π)
  - 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

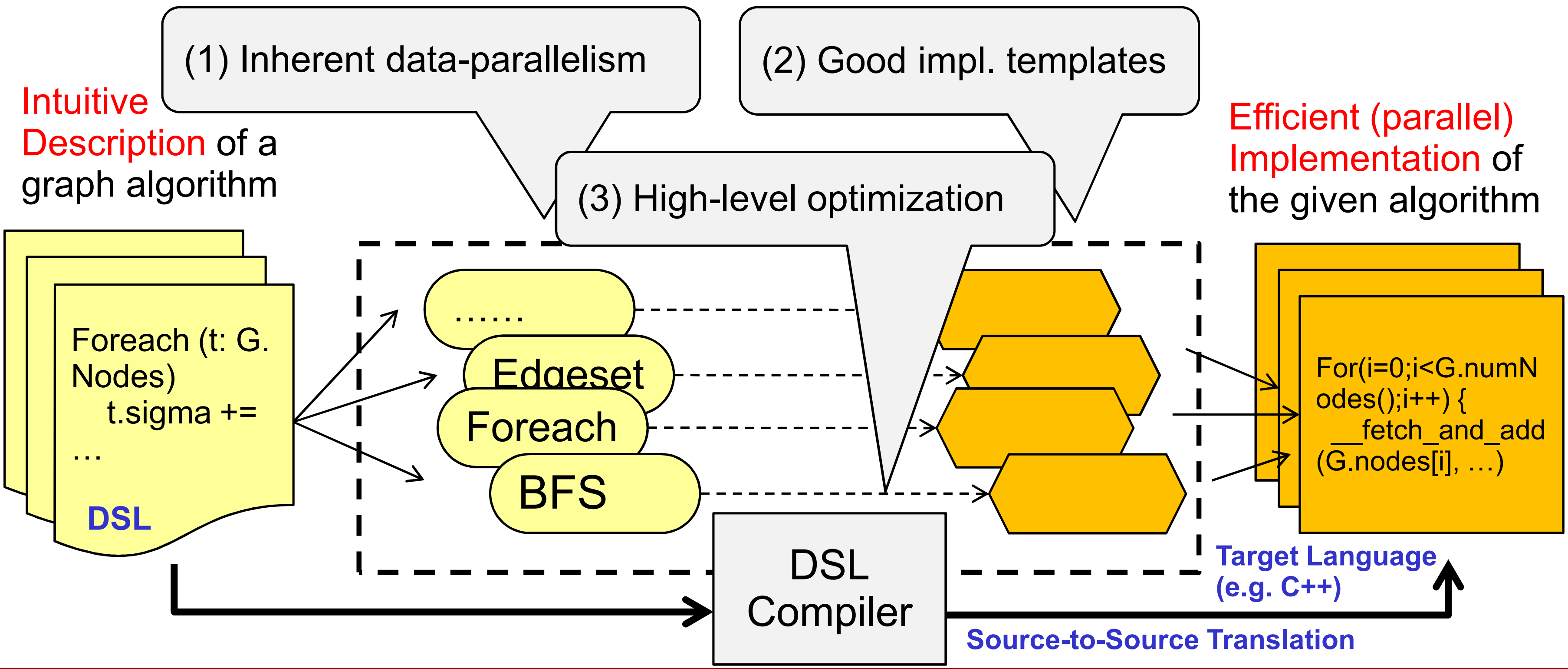
➔ Use parallelism to accelerate graph analysis
  - Plenty of data-parallelism in large graph instances
  - Performance now depends on memory *bandwidth*, not *latency*.
  - Exploit modern parallel computers: Multi-core CPU, GPU, Cray XMT, Cluster, ...

# 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

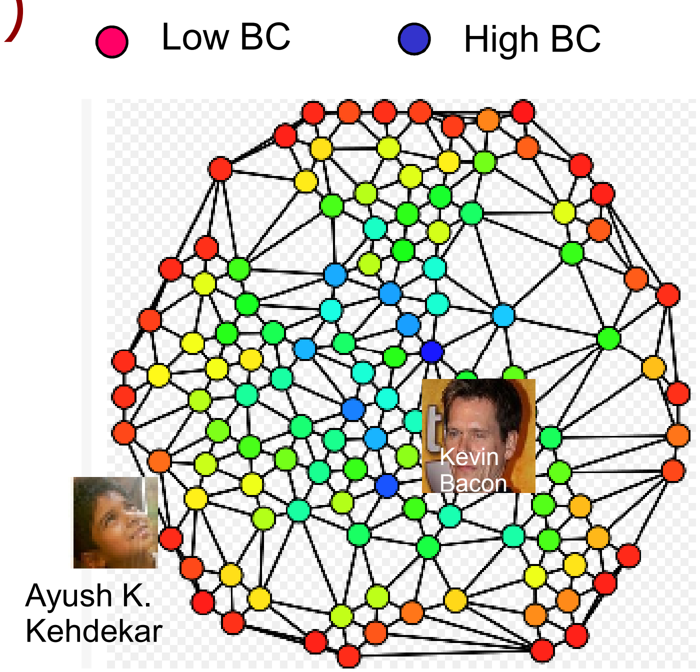
- We design a domain specific language (DSL) for graph analysis
- The user writes his/her algorithm concisely with our DSL
- The compiler translates it into the target language (e.g. parallel C++ or CUDA)



# 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]

# …eenness Centrality

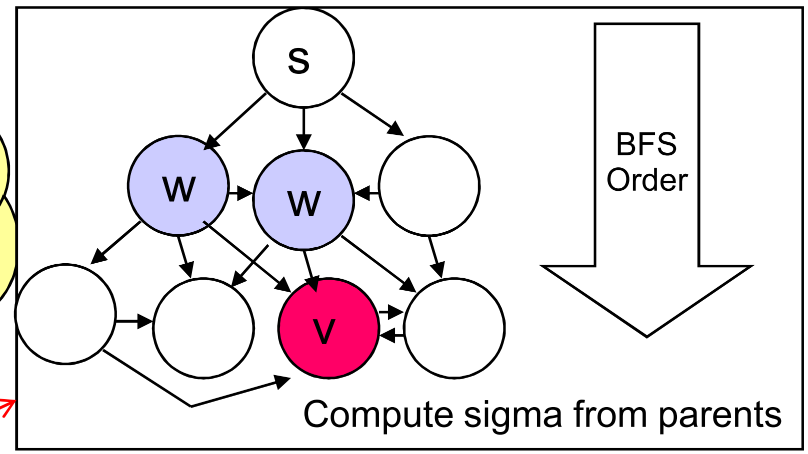
Init BC for every node
and begin outer-loop (s)

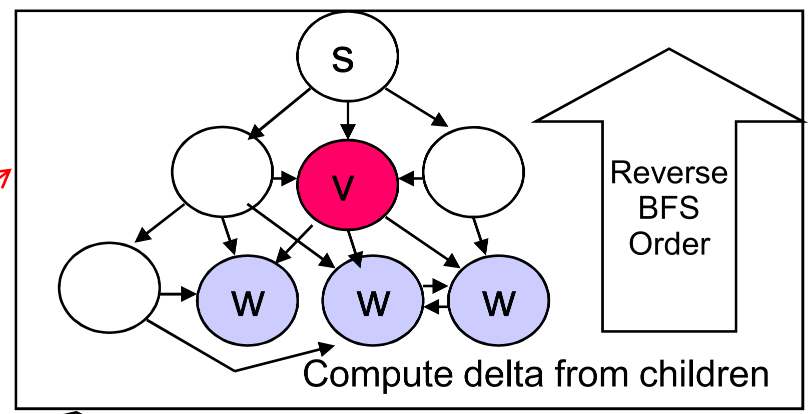
[Brand… 2001]

Looks
complex

Queues, Lists,
Stack…
Is this
parallelizable?

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + σ[v]/σ[w] · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

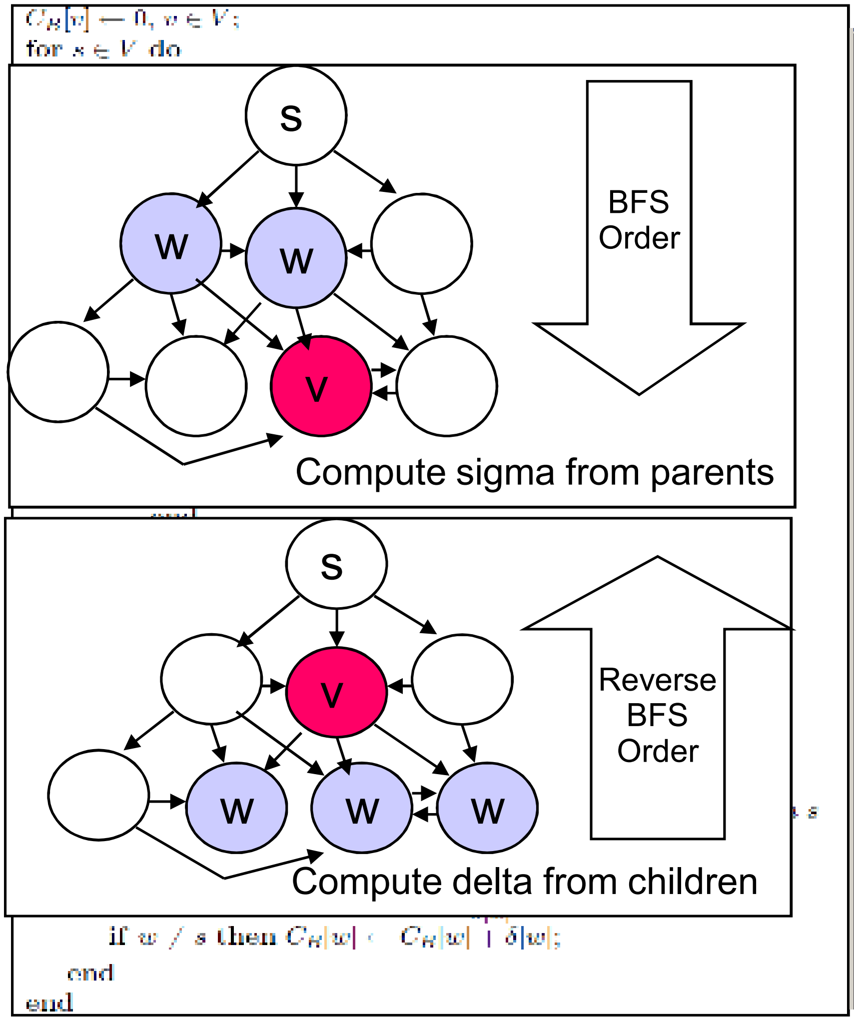



Accumulate delta into BC

# Example: Betweenness Centrality

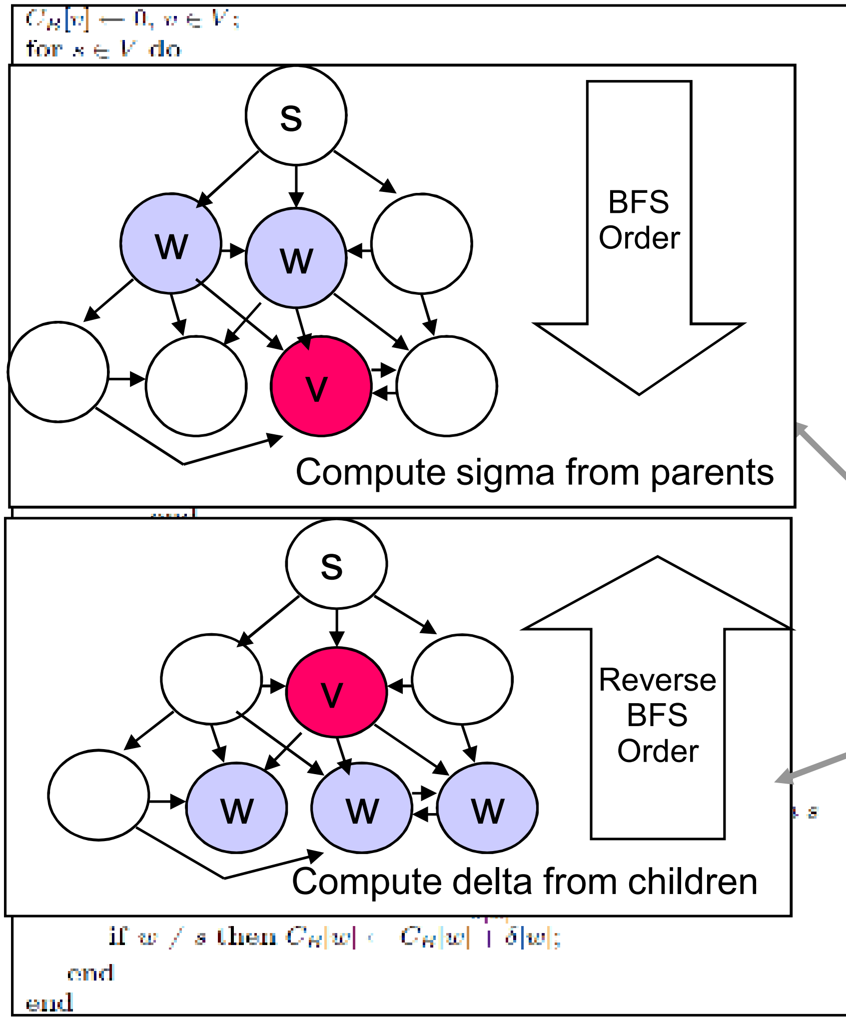
[Brandes 2001]




```
Procedure comp_BC(G: Graph, BC: Node_Property<Float>(G))
{
  G.BC = 0; // Initialize

  Foreach (s: G.Nodes) {
    // temporary values per Node
    Node_Property<Float>(G) sigma;
    Node_Property<Float>(G) delta;

    G.sigma = 0; // Initialize
    G.delta = 0;
    s.sigma = 1;

    // BFS order iteration from s
    InBFS(v: G.Nodes From s) {
      v.sigma =  // Summing over BFS parents
         Sum (w:v.UpNbrs) {w.sigma};
    }

    // Reverse-BFS order iteration to s
    InRBFS(v:G.Nodes To s)(v!=s) {
      v.delta =  // Summing over BFS children
         Sum (w:v.DownNbrs) {
            v.sigma / w.sigma * (1+ w.delta) };

      v.BC += v.delta @ s; // accumulate BC
    }
  }
}
```

# Example: Betweenness Centrality

[Brandes 2001]




```
Procedure comp_BC(G: Graph, BC: Node_Property<Float>(G))
{
  G.BC = 0; // Initialize

  Foreach (s: G.Nodes) {
    // temporary values per Node
    Node_Property<Float>(G) sigma;
    Node_Property<Float>(G) delta;

    G.sigma = 0; // Initialize
    G.delta = 0;
    s.sigma = 1;

    // BFS order iteration from s
    InBFS(v: G.Nodes From s) {
      v.sigma = // Summing over BFS parents
        Sum (w:v.UpNbrs) {w.sigma};
    }

    // Reverse-BFS order iteration to s
    InRBFS(v:G.Nodes To s)(v!=s) {
      v.delta = // Summing over BFS children
        Sum (w:v.DownNbrs) {
          v.sigma / w.sigma * (1+ w.delta) };

      v.BC += v.delta @ s; // accumulate BC
    }
  }
}
```

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

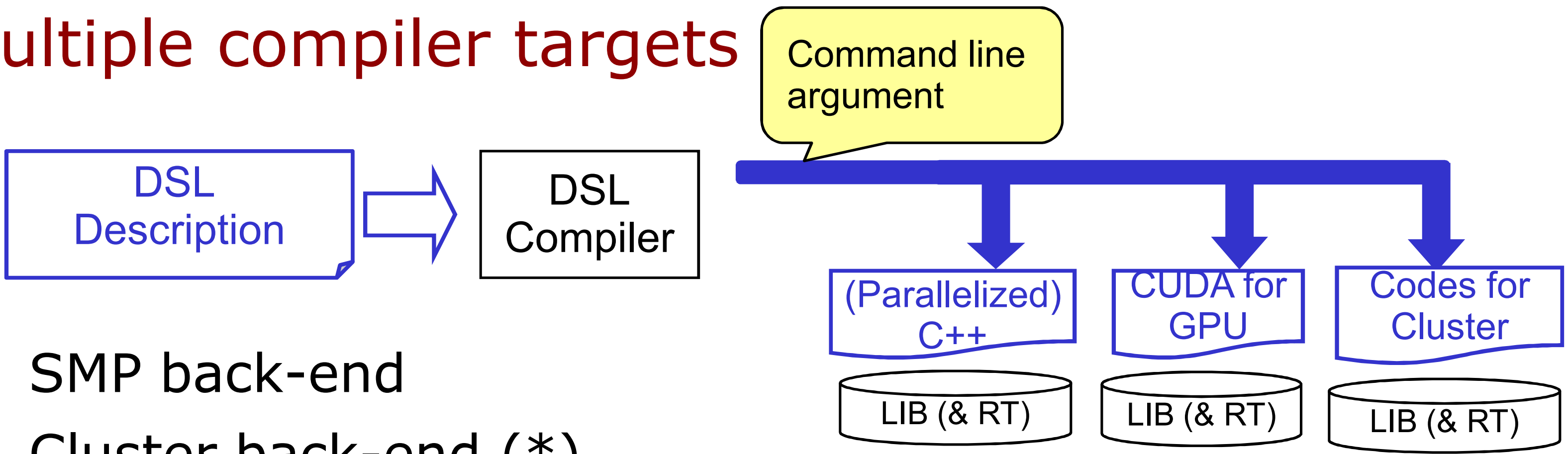
# Productivity Benefits

| Procedure | Manual LOC | Green-Marl LOC | Source | Misc |
|---|---|---|---|---|
| BC | ~ 400 | 24 | SNAP | C++ openMP |
| Vertex Cover | 71 | 21 | SNAP | C++ openMP |
| Conductance | 42 | 10 | SNAP | C++ openMP |
| Page Rank | 75 | 15 | http:// .. | C++ single thread |
| SCC | 65 | 15 | http:// .. | Java single thread |

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

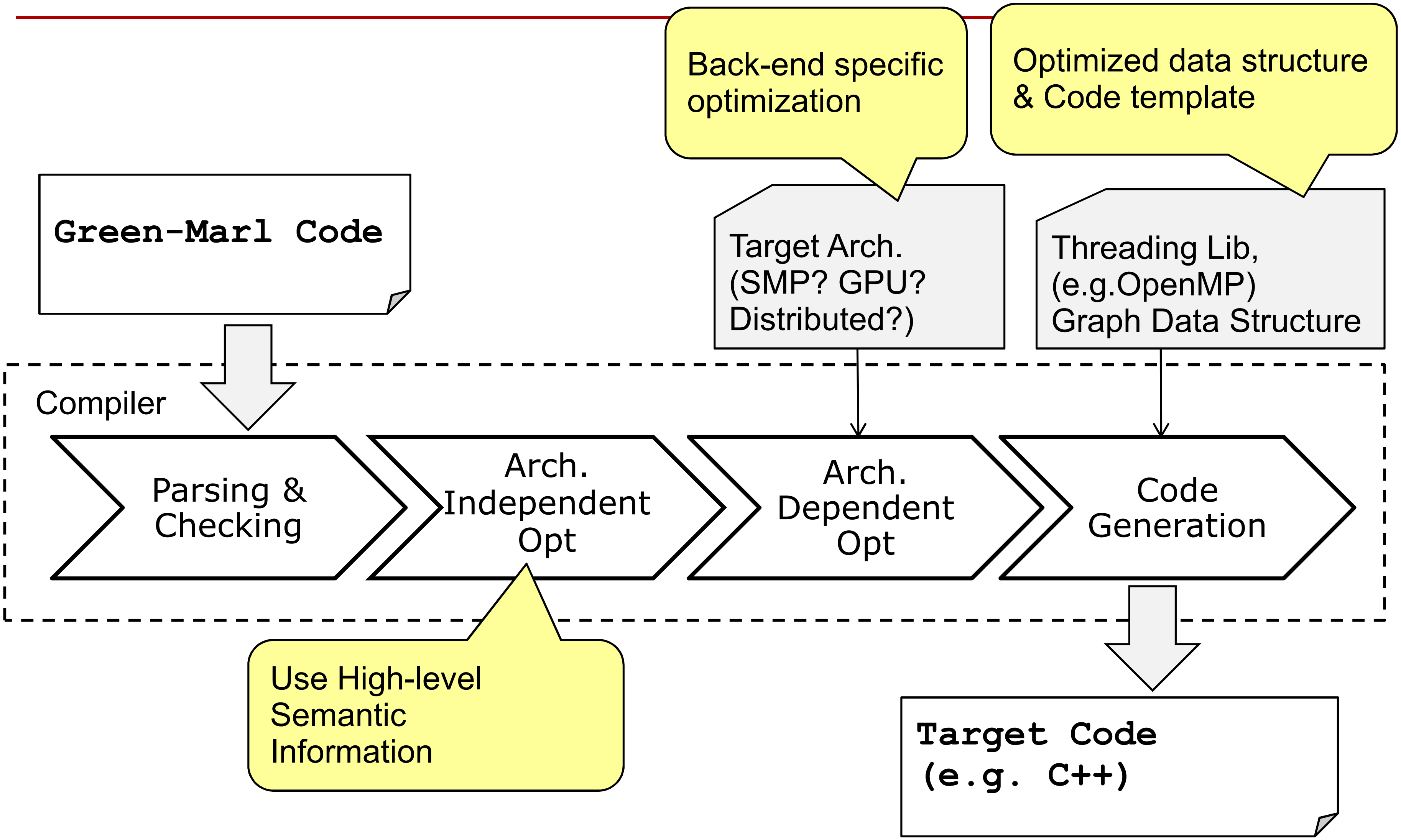
# Portability Benefits (On-going work)

- 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



# Arch-Indep-Opt: Loop Fusion

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

Loop Fusion

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

“set” of nodes
(elems are unique)

```
Map<Node, int> A, B, C;
List<Node>& Nodes = G.getNodes();
List<Node>::iterator t, s;
for(t = Nodes.begin(); t != Nodes.end(); t++)
    A[*t] = C[*t];
for(s = Nodes.begin(); s != Nodes.end(); s++)
    B[*s] = A[*s] + C[*s];
```

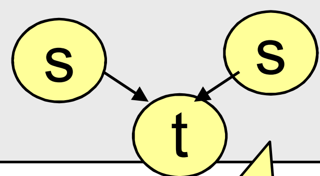
C++ compiler cannot merge loops
(Independence not gauranteed)

Optimization enabled by high-level (semantic) information

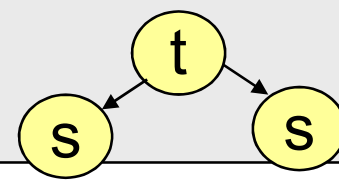
# Arch-Indep-Opt: Flipping Ed

- Graph-Specific Optimization

```
Foreach(t: G.Nodes)
  Foreach(s: t.InNbrs)(s.B>0)
    t.A += 1;
```




```
Foreach(t: G.Nodes)(t.B>0)
  Foreach(s: t.OutNbrs)
    s.A += 1;
```




Adding 1 to for all Outgoing Neighbors, if my B value is positive

Counting number of Incoming Neighbors whose B value is positive

(Why?) Reverse edges may not be available or expensive to compute

Optimization using domain-specific property

# 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 min= t.C
}
```

Three levels of nested parallelism + reductions

```
For (t: G.Nodes) {
  Foreach(s: G.Nodes)(s.X > t.Y) {
    For (r: s.Nbrs) {
       s.A += r.B;
    }
    t.C *= s.A;
  }
  val min= t.C
}
```

Compiler chooses parallel region, heuristically

[Why?]
- Graph is large
- # core is small.
- There is overhead for parallelization

```
For (t: G.Nodes) {
  Foreach(s: G.Nodes)(s.X > t.Y) {
    For (r: s.Nbrs) {
       s.A = s.A + r.B;
    }
    t.C *= s.A;
  }
  val = (t.C < val) ? t.C : val;
}
```

Reductions became normal read & write

Optimization enabled by both architectural and domain knowledge

# Code-Gen: Optimization enabled by code analysis (i.e. no BFS library could do this automatically)

- Prepare data structure for reverse BFS traversal during forward traversal, *only if required*.

```
InBFS(t: G.Nodes From s) {
  …
}
InRBFS {
   Foreach (s: t.DownNbrs)
      …
}
```

Compiler detects that down-nbrs are used in reverse traversal

Generated code can iterate only ***edges to down-nbrs*** during reverse traversal

```
// Preperation of BFS
…

// Forward BFS (generated)
{ …
  // k is an out-edge of s
  for(k … )
     node_t child = get_node(k);
     if (is_not_visited(child)) {
        …;     // normal BFS code here
        edge_bfs_child[k] = true;
  }   }
…}

// Reverse BFS (generated)
{ …
 // k is an out-edge of s
  for(k … )  {
     if (!edge_bfs_child[k]) continue;
     …
} }
```

Generated code saves ***edges to the down-nbrs*** during forward traversal.

# Code-Gen: Code Templates

- Data Structure
  - Graph: similar to a conventional graph library
  - Collections: custom implementation
- Code Generation Template
  - BFS
    - Hong et al. PACT 2011 (for CPU and GPU)
    - Better implementations coming; can be adapted transparently
  - DFS
    - Inherently sequential

Compiler takes any benefits that a (template) library would give, as well

# 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) ~ x2.3 on Cray XMT
  - Parallel implementation available in SNAP library based on (1) not (2) (for x86)

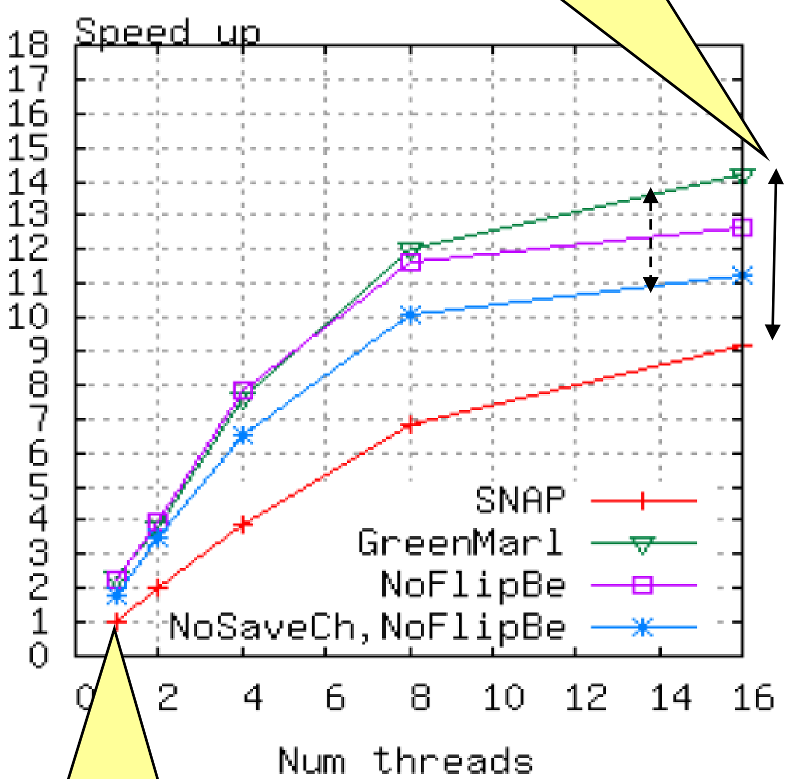
- Our Experiment
  - Start from DSL description (as shown previously)
  - Let the compiler apply the optimizations in (2), *automatically*.

# Results

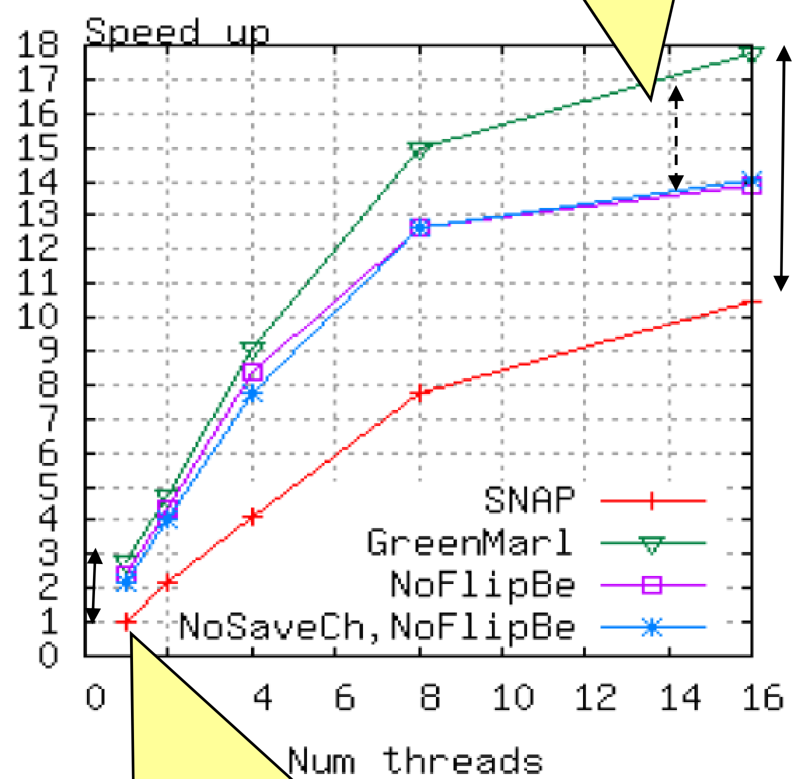
Parallel performance difference

Effects of other optimizations
• Flipping Edges
• Saving BFS children

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




(a) RMAT

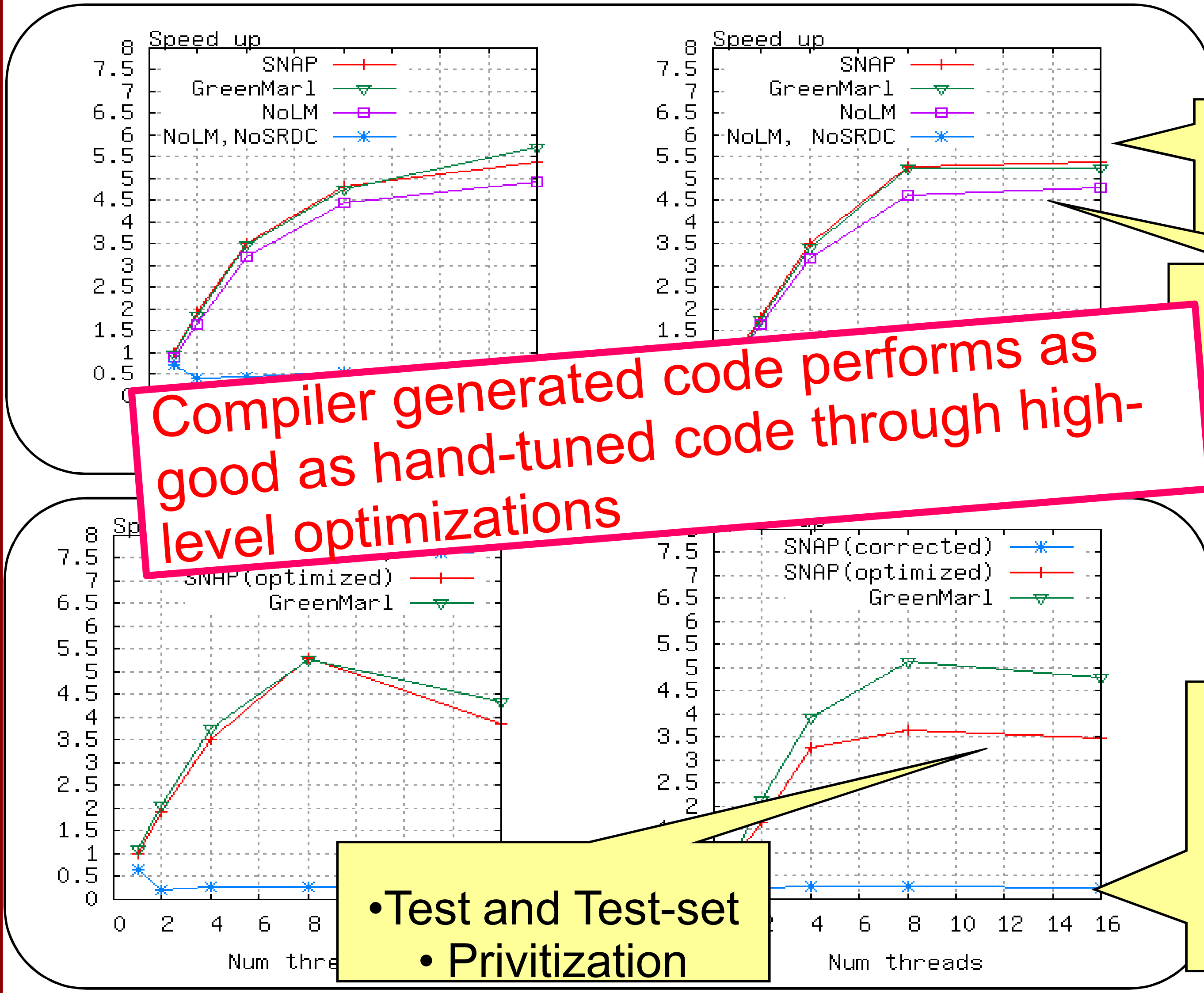



orm

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

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

**Vertex Cover**

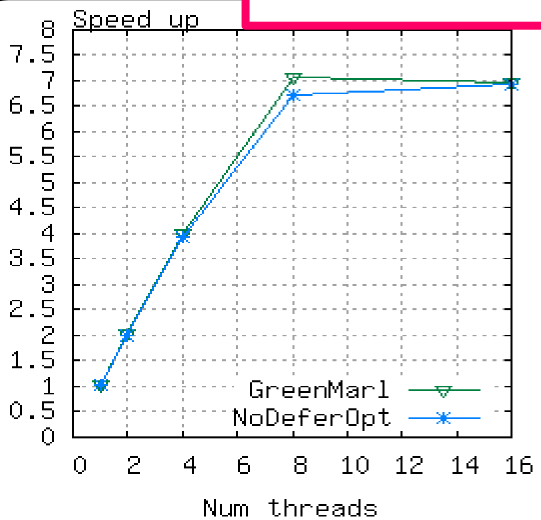
- Test and Test-set
- Privitization

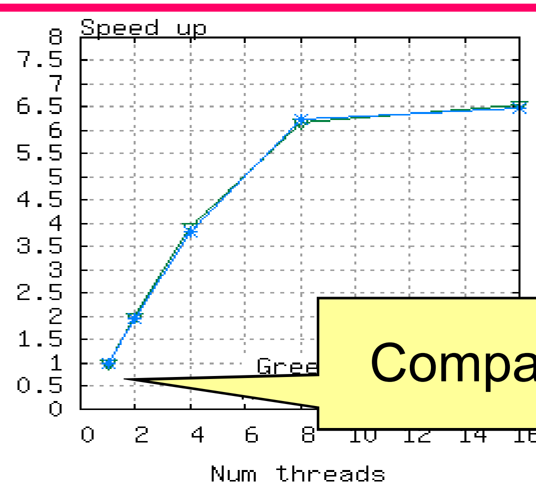
Original code ➔ data race;
Naïve correction (omp_critical) ➔ serialization

# Othe

Automatic parallelization as much as exposed data parallelism (i.e. there is no black magic)

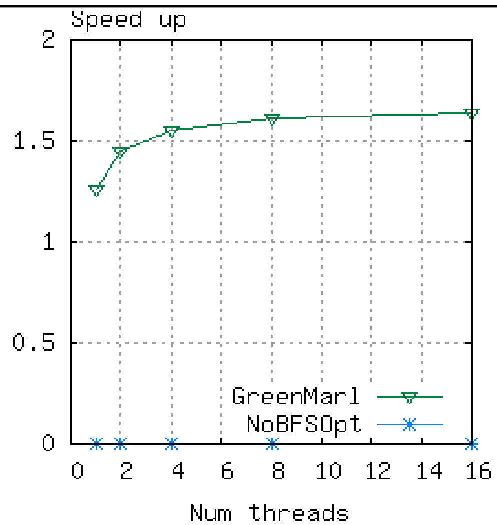
**PageRank**

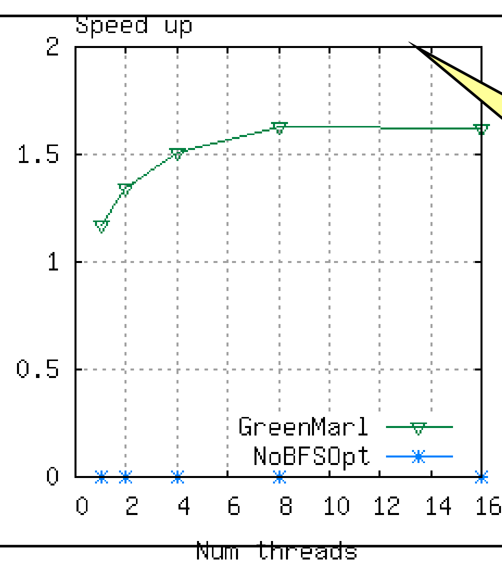



Compare against Seq. Impl

**Strongly Connected Component**




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

# Conclusion

- Green-Marl
  - A DSL designed for graph analysis
- Three benefits
  - Productivity
  - Performance
  - *Portability (soon)*

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