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

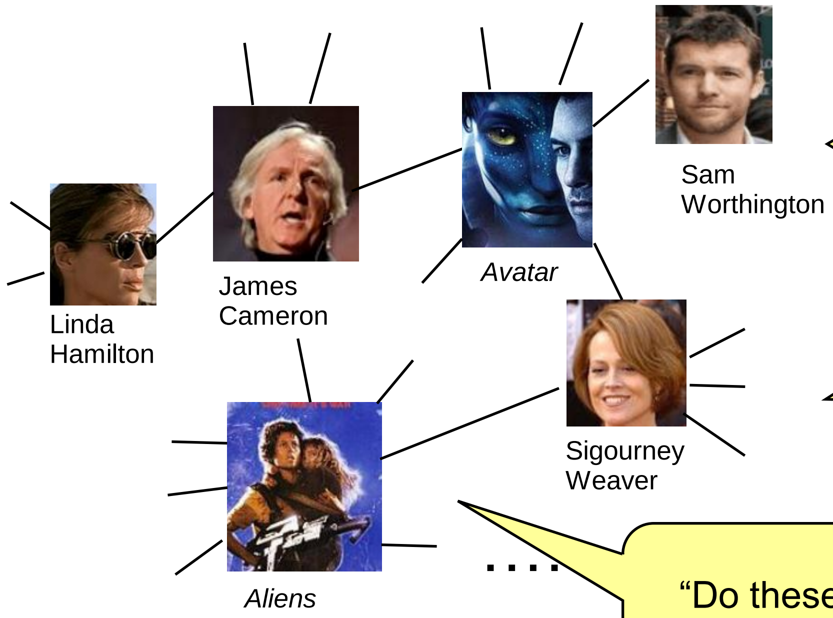
Sungpack Hong*+, Hassan Chafi*+, Eric Sedlar+, and Kunle Olukotun*

*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



"What would be the avg. hop-distance between any two (Australian) actors?"

"Is he a central figure in the movie network? How much?"

"Do these actors work together more frequently than others?"

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

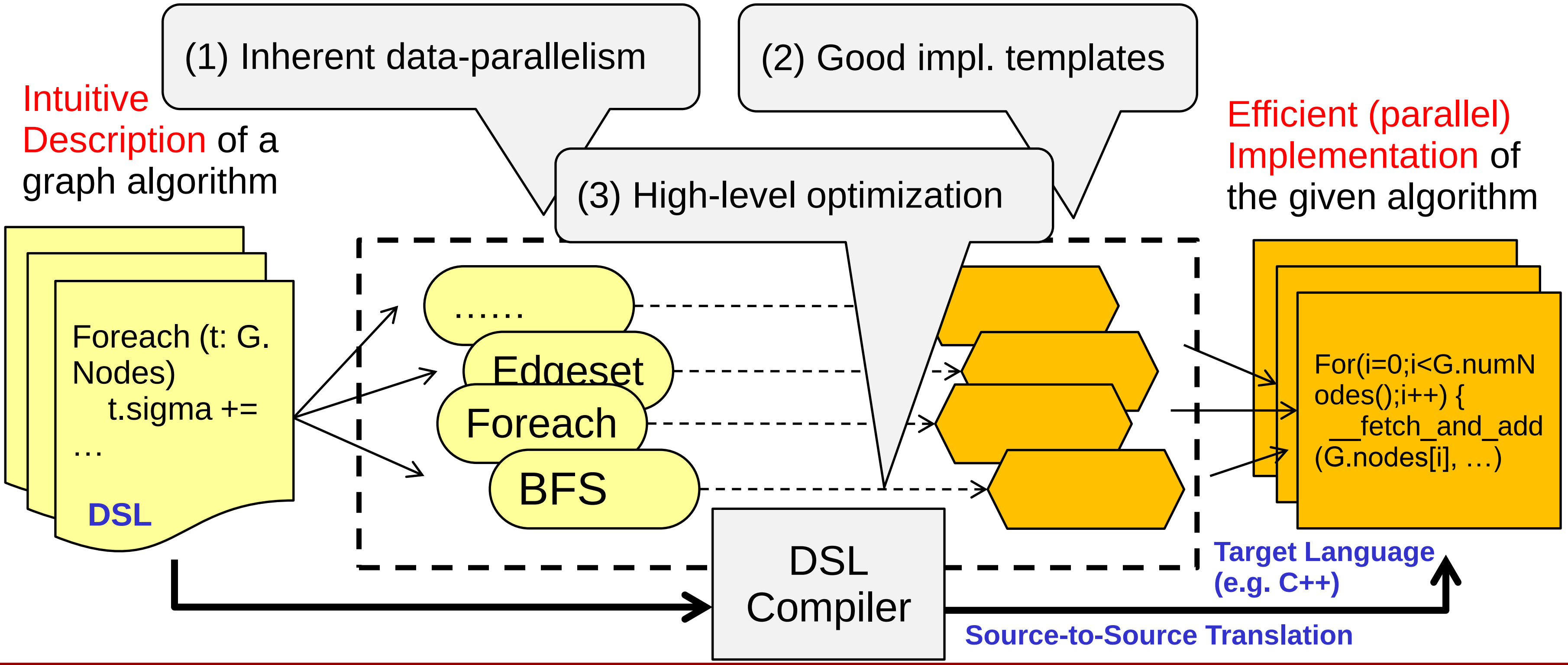
➔ Use parallelism to accelerate graph analysis
  - Plenty of data-parallelism in large graph instances
  - Performance now depends on memory *bandwidth*, not *latency*.
  - Exploit modern parallel computers: Multi-core CPU, GPU, Cray XMT, Cluster, ...

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)

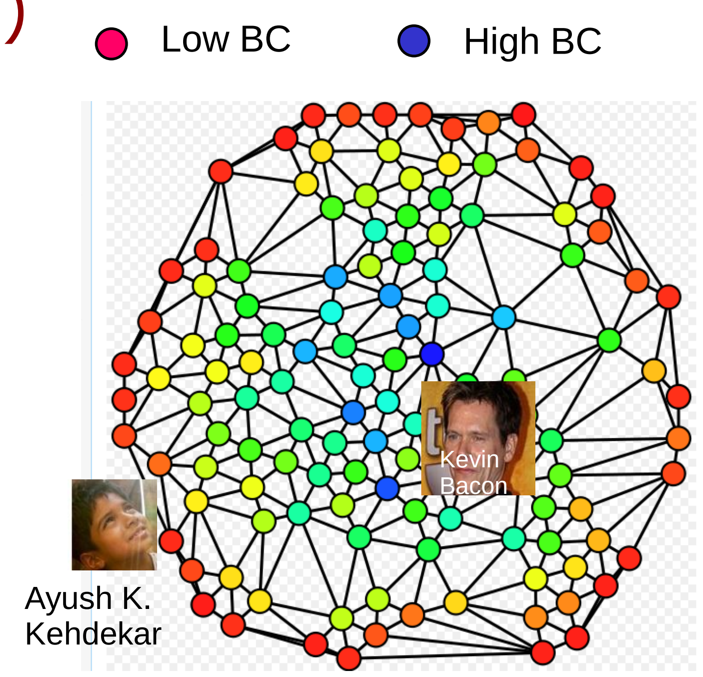


Intuitive Description of a graph algorithm
(1) Inherent data-parallelism
(2) Good impl. templates
(3) High-level optimization
Efficient (parallel) Implementation of the given algorithm
Foreach (t: G. Nodes)
t.sigma +=
…
DSL
……
Edgeset
Foreach
BFS
For(i=0;i<G.numNodes();i++) {
__fetch_and_add (G.nodes[i], …)
Target Language (e.g. C++)
DSL Compiler
Source-to-Source Translation

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]

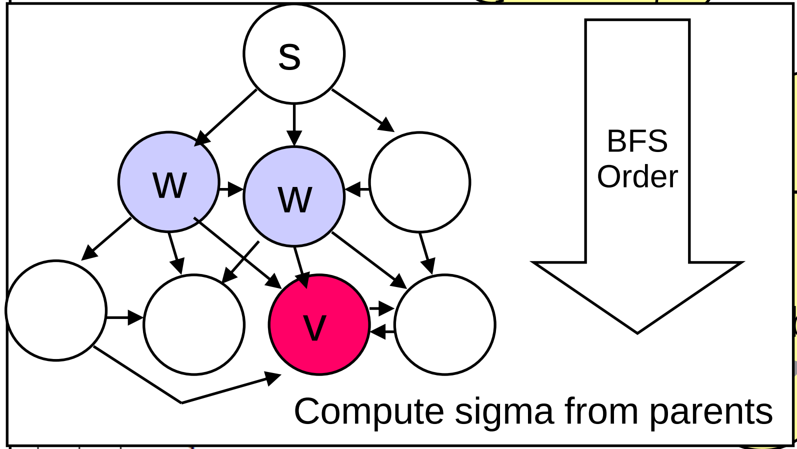
E eenness Centrality

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

[Brandes 01]

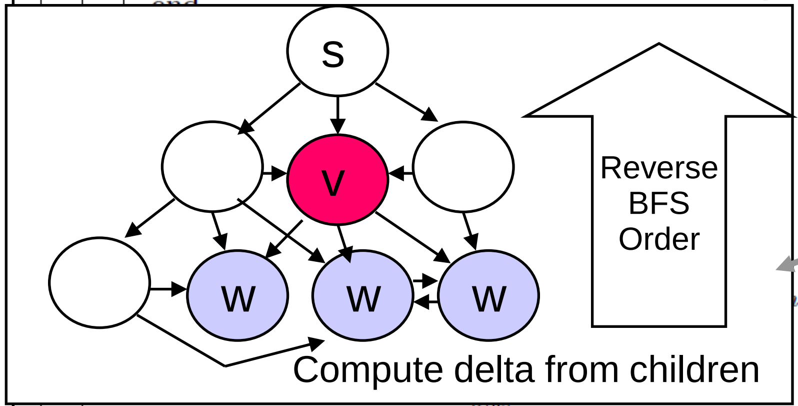
Looks
complex

$C_B[v] \leftarrow 0, v \in V$;
for $s \in V$ do



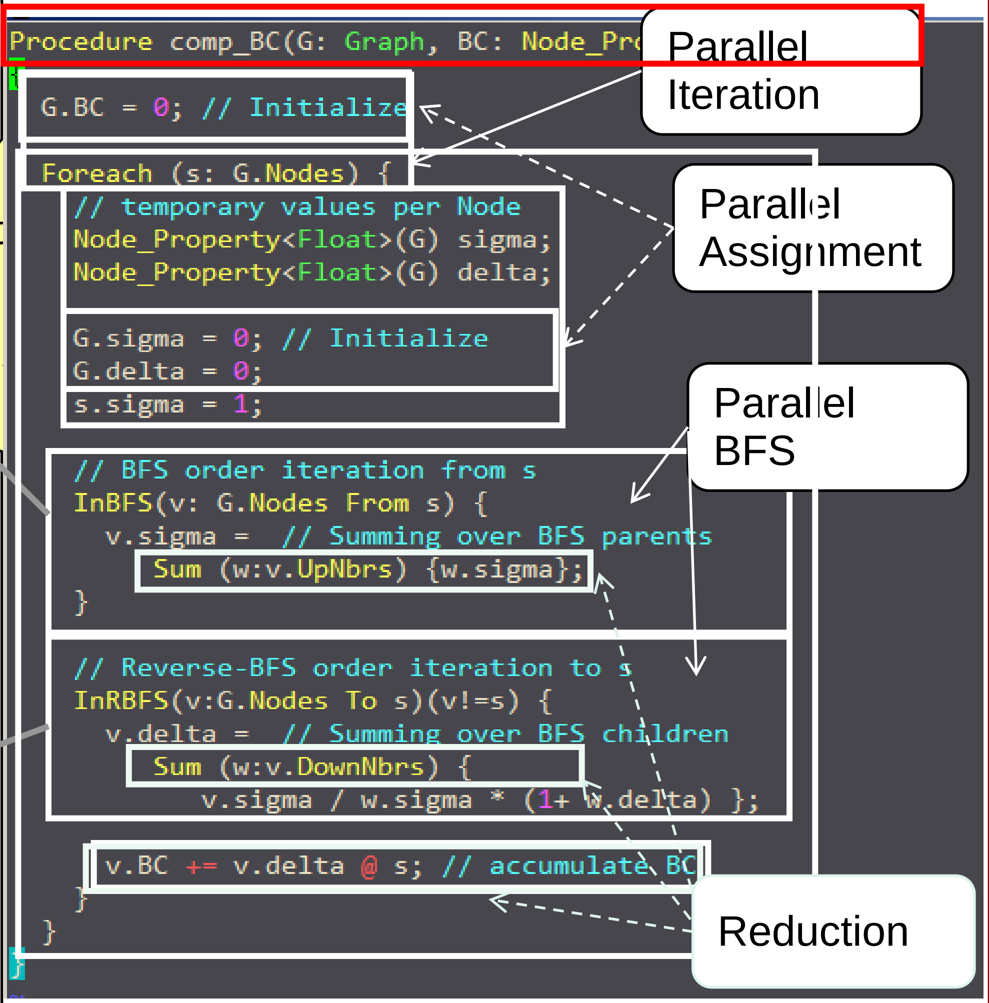
List

ble



```
    if $w \neq s$ then $C_B[w] \leftarrow C_B[w] + \delta[w]$;
  end
```

Accumulate delta into BC



```
Procedure comp_BC(G: Graph, BC: Node_Pro
{
  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

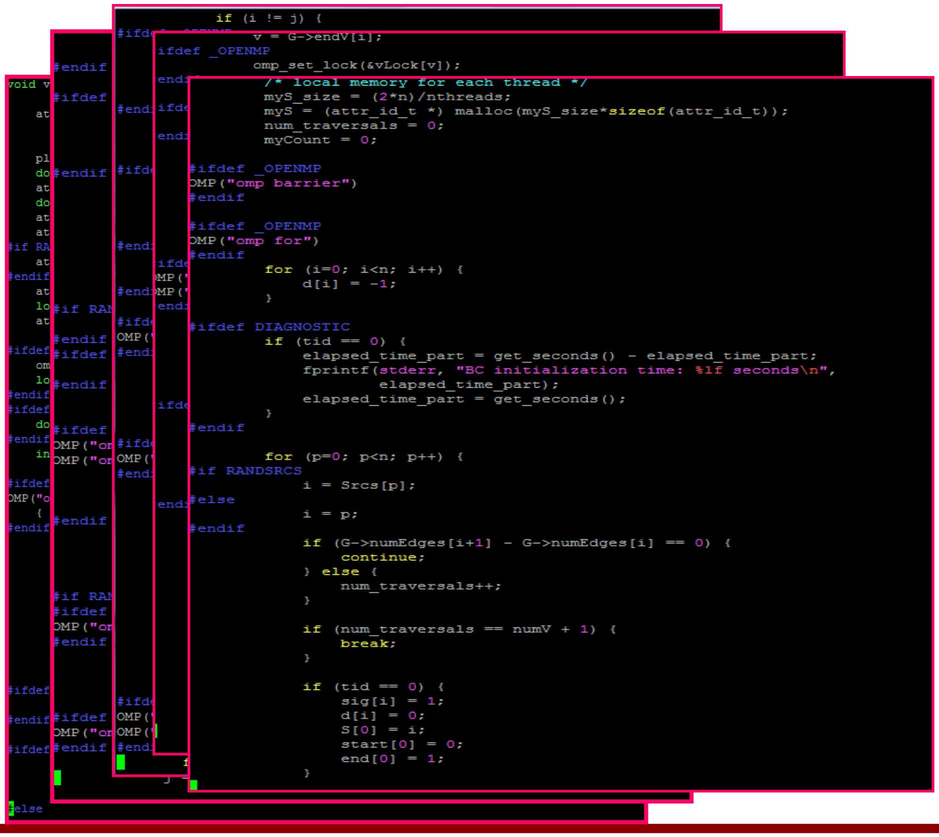
Reduction

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*?)



```
        /* local memory for each thread */
        myS_size = (2*n)/nthreads;
        myS = (attr_id_t *) malloc(myS_size*sizeof(attr_id_t));
        num_traversals = 0;
        myCount = 0;

#ifdef _OPENMP
OMP("omp barrier")
#endif

#ifdef _OPENMP
OMP("omp for")
#endif
        for (i=0; i<n; i++) {
            d[i] = -1;
        }

#ifdef DIAGNOSTIC
        if (tid == 0) {
            elapsed_time_part = get_seconds() - elapsed_time_part;
            fprintf(stderr, "BC initialization time: %lf seconds\n",
                    elapsed_time_part);
            elapsed_time_part = get_seconds();
        }
#endif

        for (p=0; p<n; p++) {
#if RANDSRCS
            i = Srcs[p];
#else
            i = p;
#endif
            if (G->numEdges[i+1] - G->numEdges[i] == 0) {
                continue;
            } else {
                num_traversals++;
            }

            if (num_traversals == numV + 1) {
                break;
            }

            if (tid == 0) {
                sig[i] = 1;
                d[i] = 0;
                S[0] = i;
                start[0] = 0;
                end[0] = 1;
            }
```

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

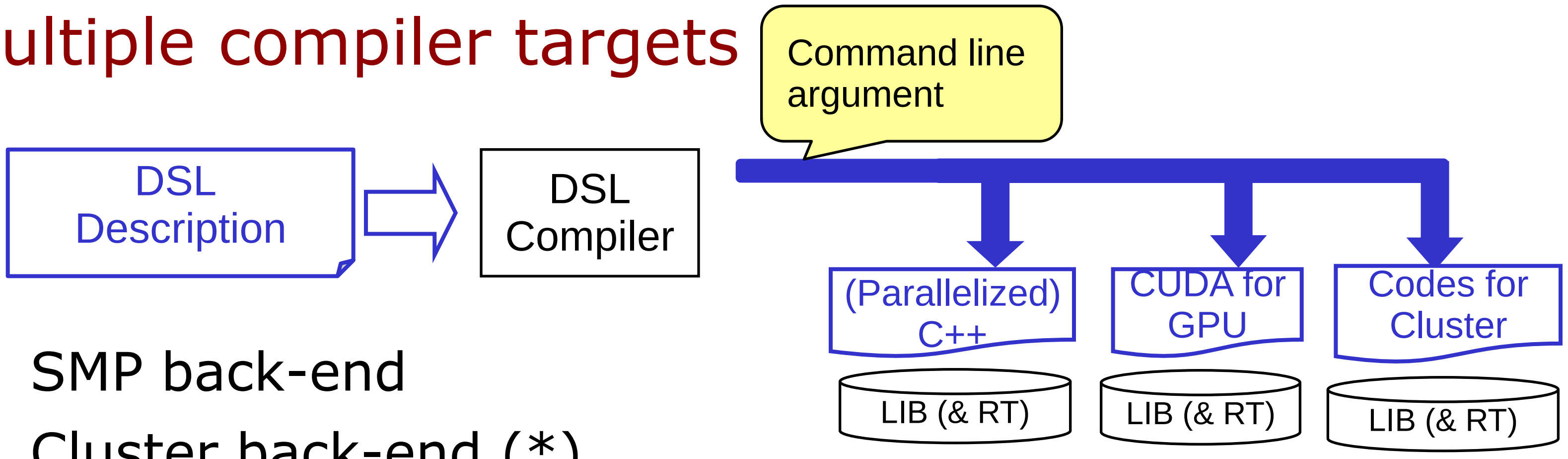
Productivity Benefits

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

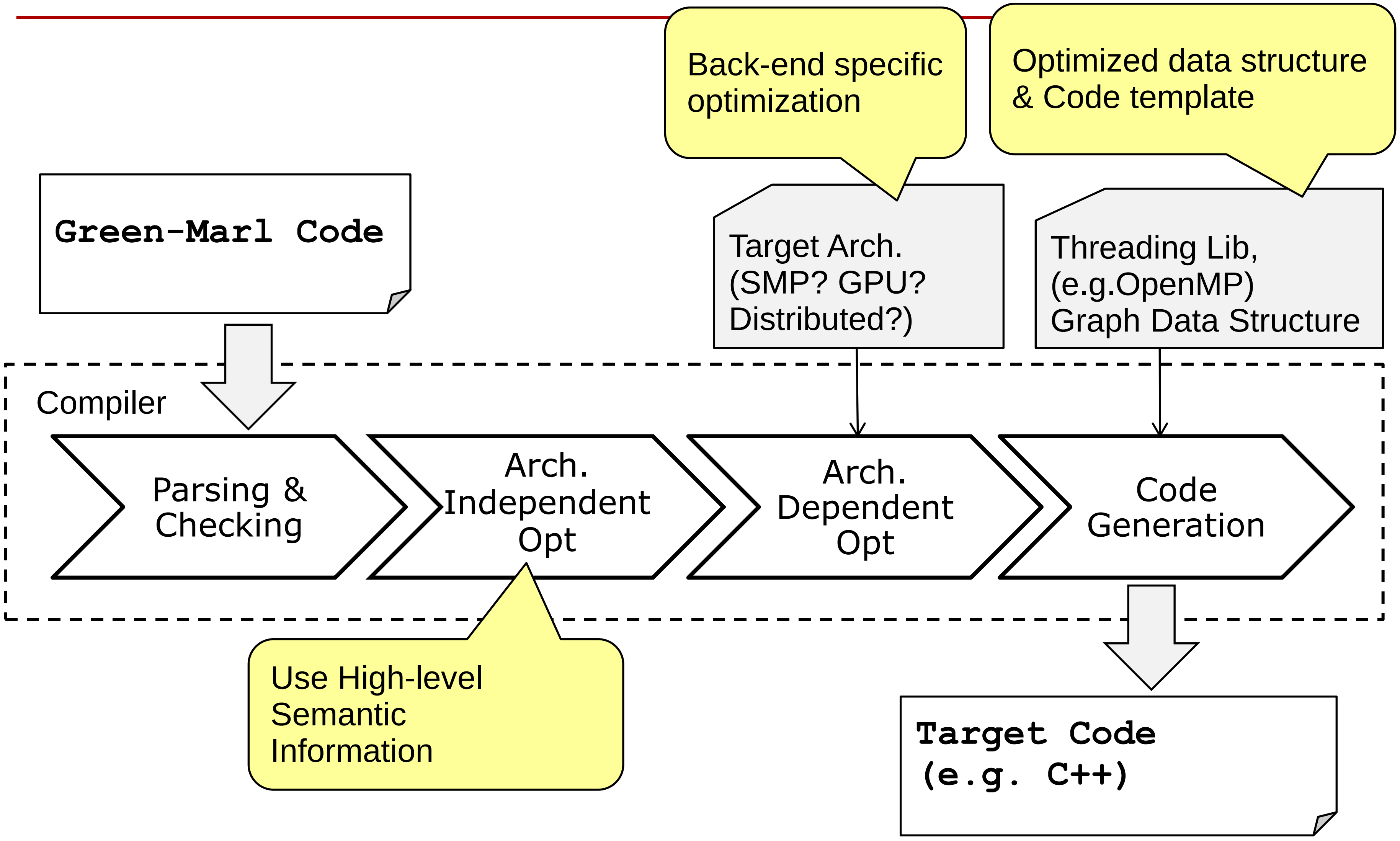
Portability Benefits

- 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


Back-end specific optimization
Optimized data structure & Code template
Green-Marl Code
Target Arch. (SMP? GPU? Distributed?)
Threading Lib, (e.g.OpenMP) Graph Data Structure
Compiler
Parsing & Checking
Arch. Independent Opt
Arch. Dependent Opt
Code Generation
Use High-level Semantic Information
Target Code (e.g. C++)

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.
}
```

les
que)

Syn
Exp

Optimization enabled by high-level (semantic) information

```
G.A = G.C + 1;    // Gro
G.B = G.A + G.C;  // (ve
…
```

```
        ::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];
```

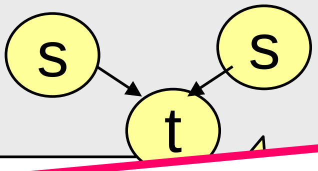
Syntactic sugars may create a lot of independent loops

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

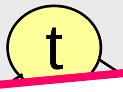
Arch-Indep-Opt: Flipping

- 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

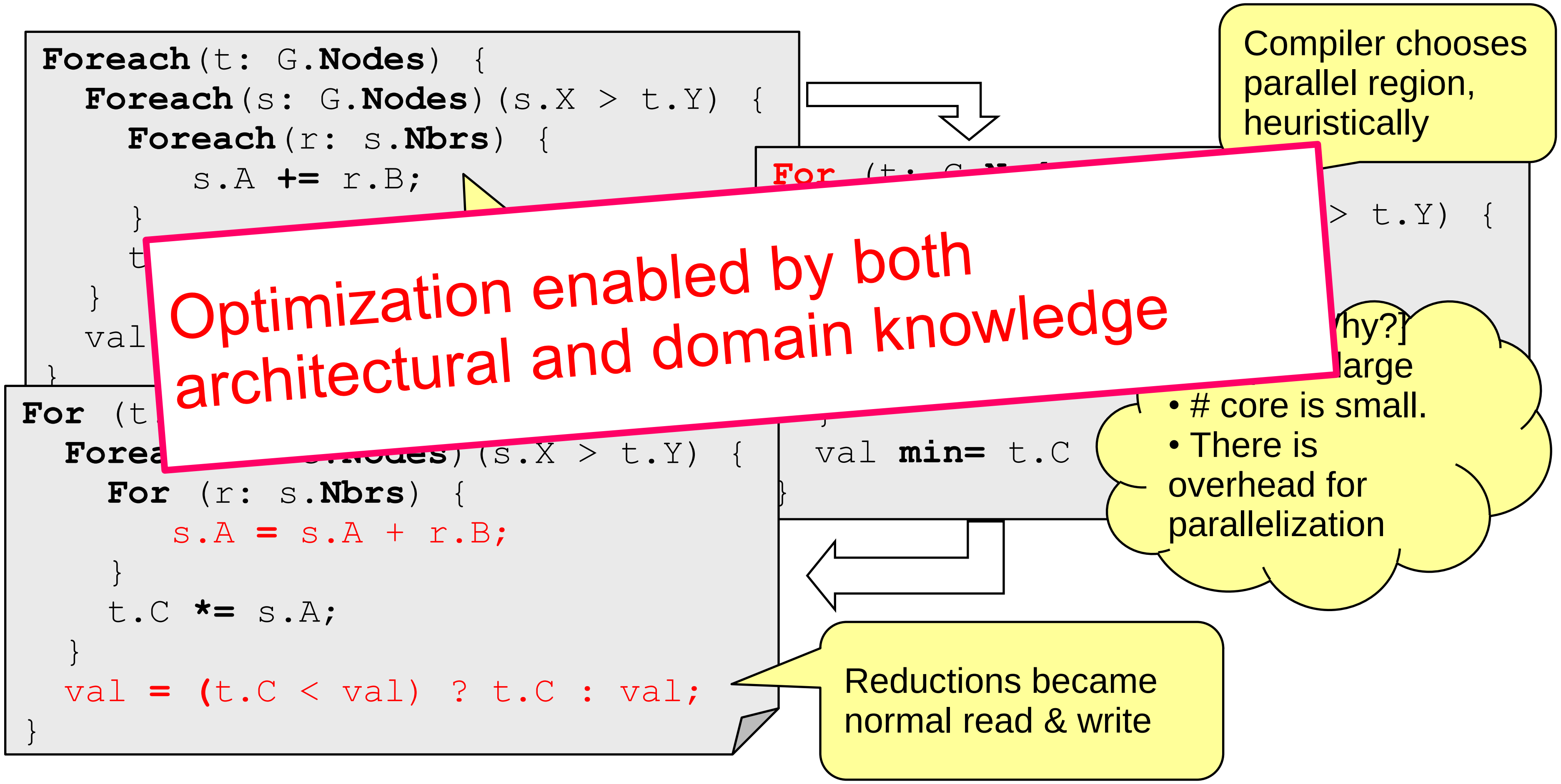
Optimization using domain-specific property

Counting number of Incoming Neighbors whose B value is positive

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

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
  }
  val
}
```

```
For (t: G.N
                  > t.Y) {

}
  val min= t.C
}
```

```
For (t
  Forea          .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;
}
```

Compiler chooses parallel region, heuristically

hy?]
large
- # core is small.
- There is overhead for parallelization

Reductions became normal read & write

Optimization enabled by both architectural and domain knowledge

Code-Gen: Saving DownNbrs in BFS

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

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

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

```
// Preperation of BF
…

// Forward BFS (generated)
{ …
  // k is an out-edge of s

…}

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

Con
dow
reverse traversal

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

Optimization enabled by code analysis
(i.e. no BFS library could do this automatically)

Code-Gen: Reduction

- Reduction to Scalar➔ Privatization

```
// reduction by minimum
Foreach(t: G.Nodes)
  x min= t.A;
```

```
// C++ OpenMP Implementation
#pragma omp parallel
{   // Privatization
    int x_prv = x;
    #pragma omp for
    for(t=G.begin();...)
      x_prv = min(x_prv, A[t]);
                                                   ;
        if (x >= x_prv) break;
        success = CAS(x, x_prv);
      }
    }
}
```

Compiler mimics the way experts code, using high-level information

Code-Gen: Code Templates

- Data Structure
  - Graph: similar to a conventional graph library
  - Collections: custom implementation

Generated code also benefits from optimized libraries

    - Hong et al. PACT 2011 (for CPU and GPU)
    - Better implementations coming; can be adapted transparently
  - DFS
    - Inherently sequential

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)

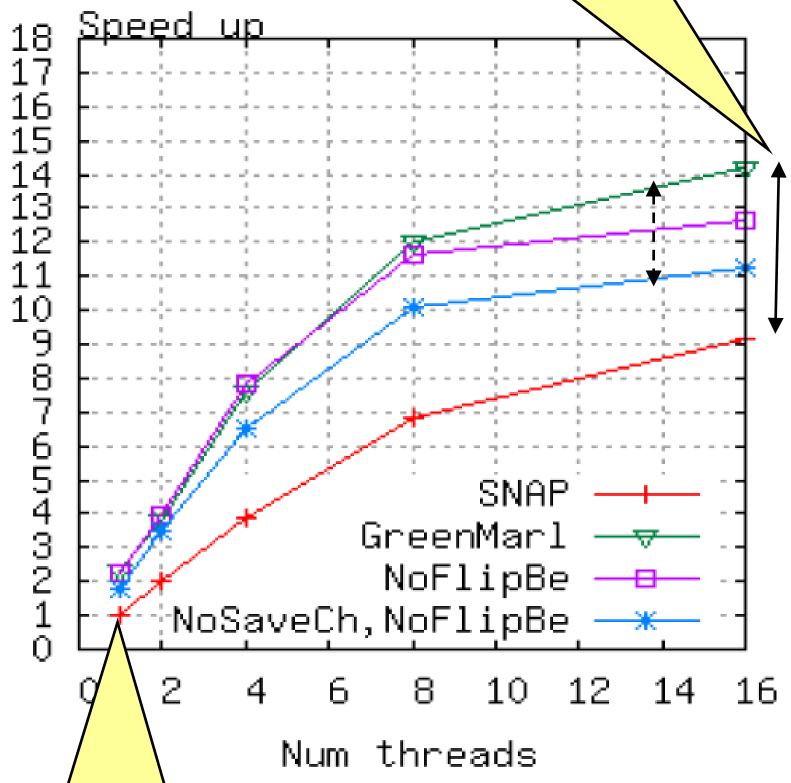
- Our Experiment
  - Start from DSL description (as shown previously)
  - Let the compiler apply the optimizations in (2), *automatically*.

Resul

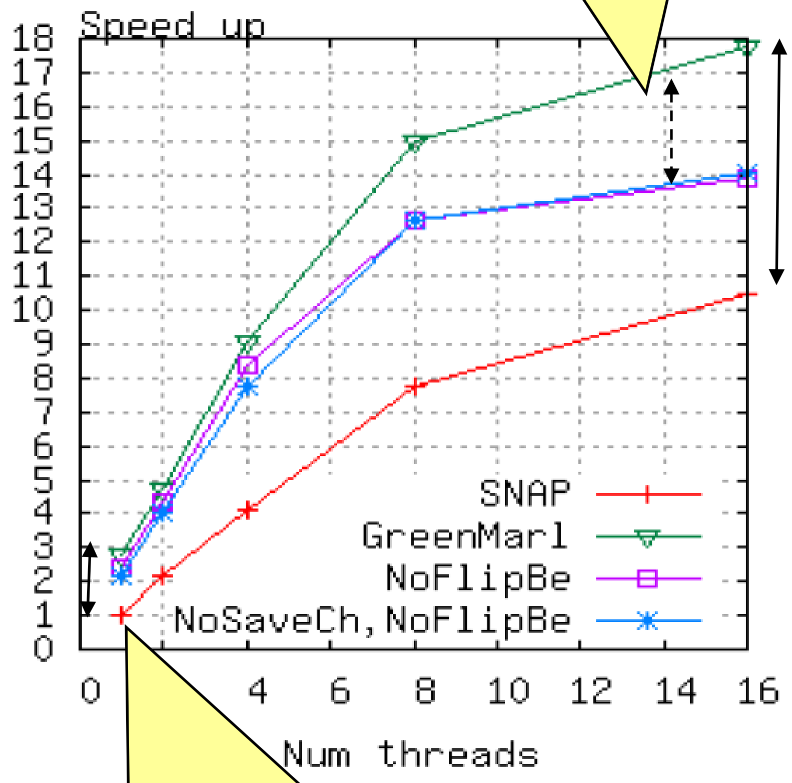
Parallel performance difference

Effects of other optimizations
• Flipping Edges
• Saving BFS children

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



(a) RMAT



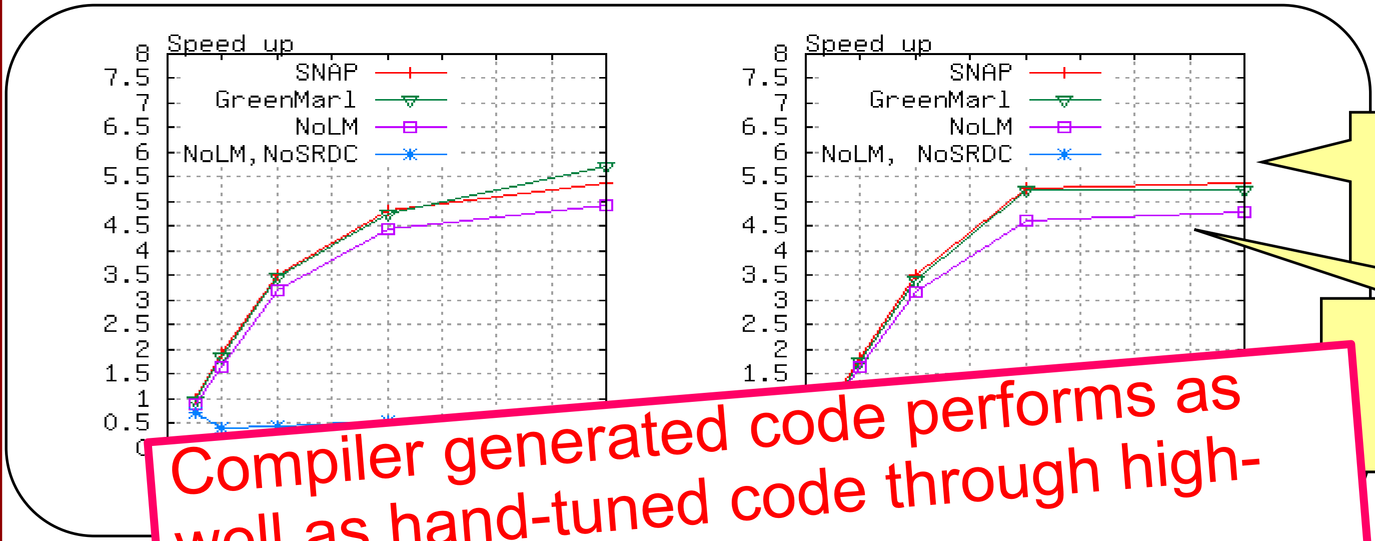
orm

Shows speed up over
Baseline: SNAP
(single thread)

Better single thread performance:
(1) Efficient BFS code
(2) No unnecessary locks

Other Results

Conductance

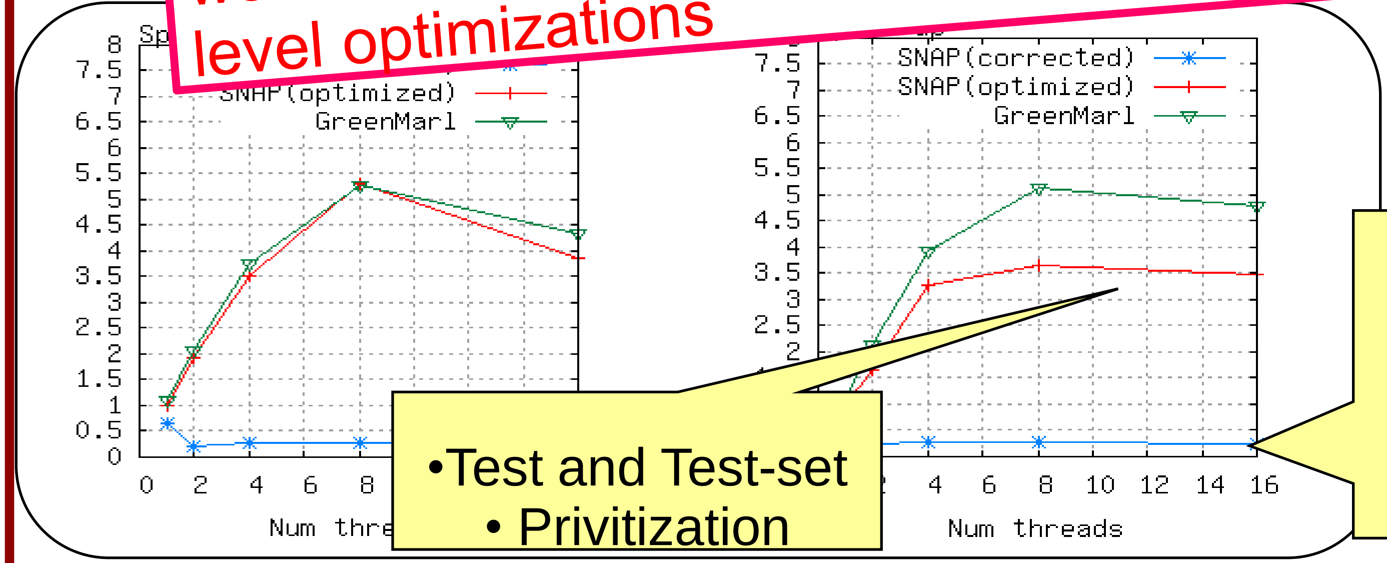


Perf similar to manual impl.

•Loop Fusion
• Privitization

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

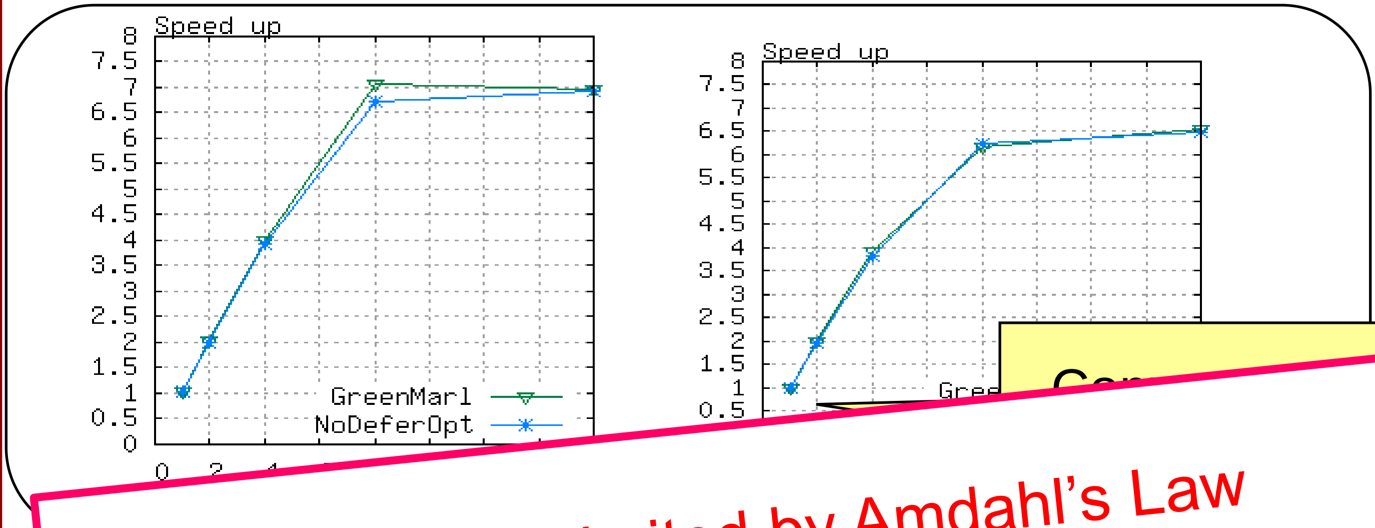
Vertex Cover



•Test and Test-set
• Privitization

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

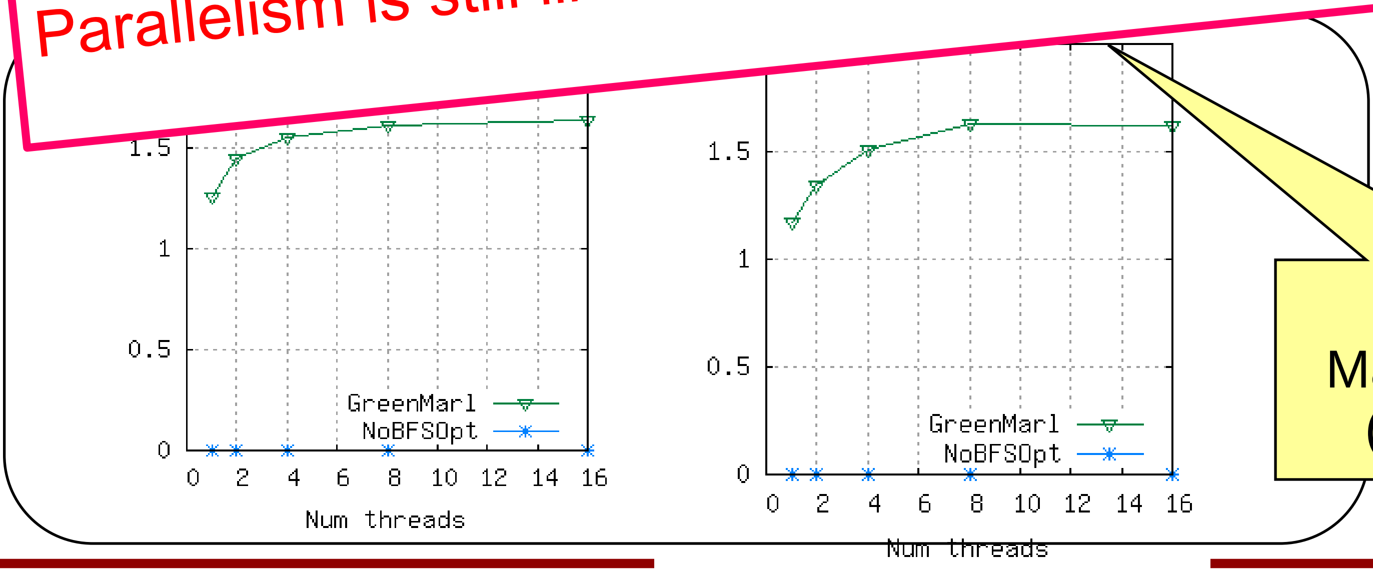
Other Results



PageRank

Co t Seq. Impl

Parallelism is still limited by Amdahl's Law



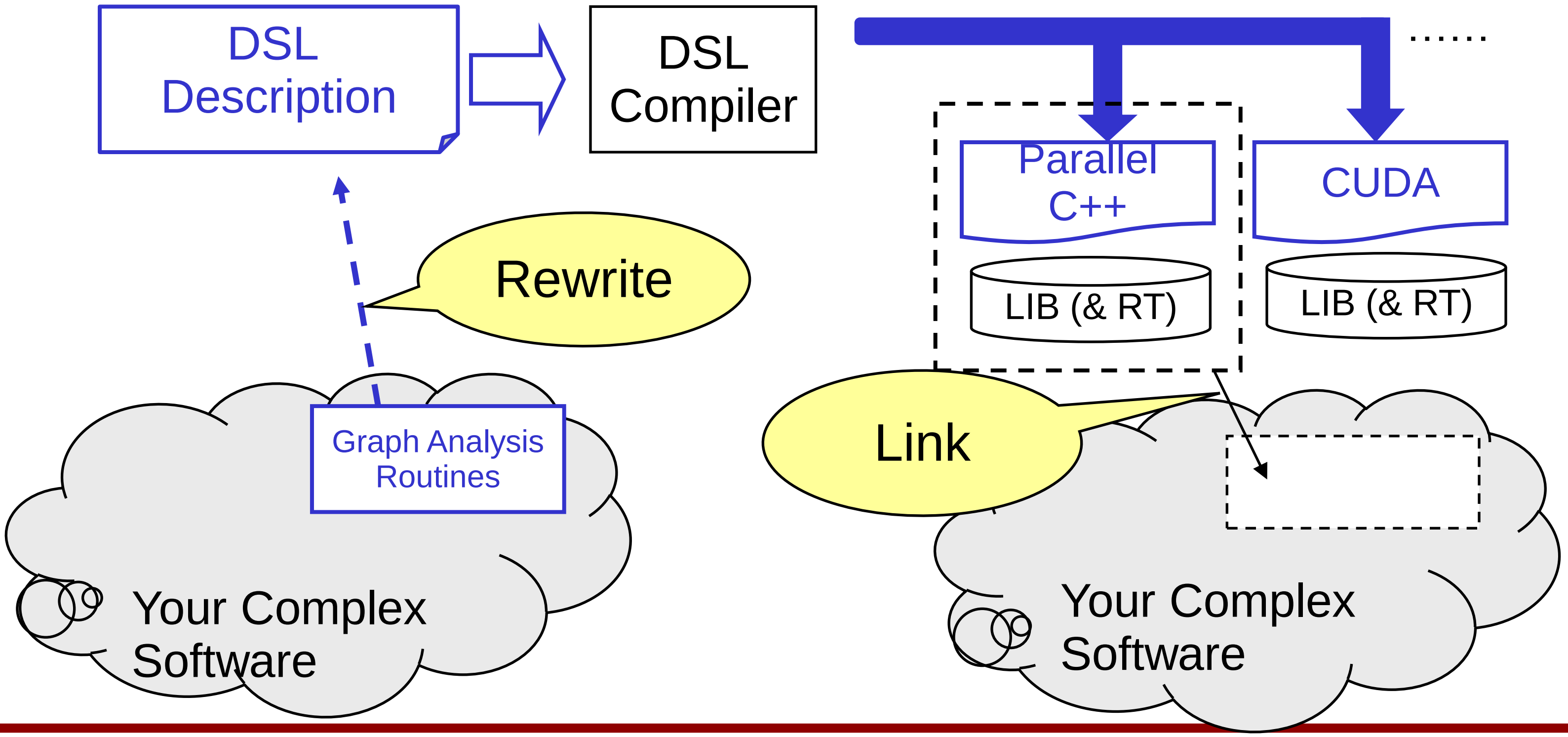
trongly Connected Component

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

Usage Model

"Do you expect me to re-write my whole application with your DSL?"

- No. Our src-to-src translation does not demand it.
- Okay, maybe a little glue code



About Libraries

"Can I still use my custom library inside DSL?"

- Yes, via *foreign syntax*
  - Similar to _asm_ mechanism in gcc
  - Statements inside []

    ➔ Compiler simply keeps the text as-is in the generated code
  - Just tell the compiler what are being read/mutated.

```
Procedure foo(x: Int, U: $User_Type) {
    ……
    //  Read-Set: x and U
    //  Write-Set: x
    [C_function($x, $U.get_val()) ]::[x];
    ……
}
```

Any foreign (e.g. C++) statement inside []

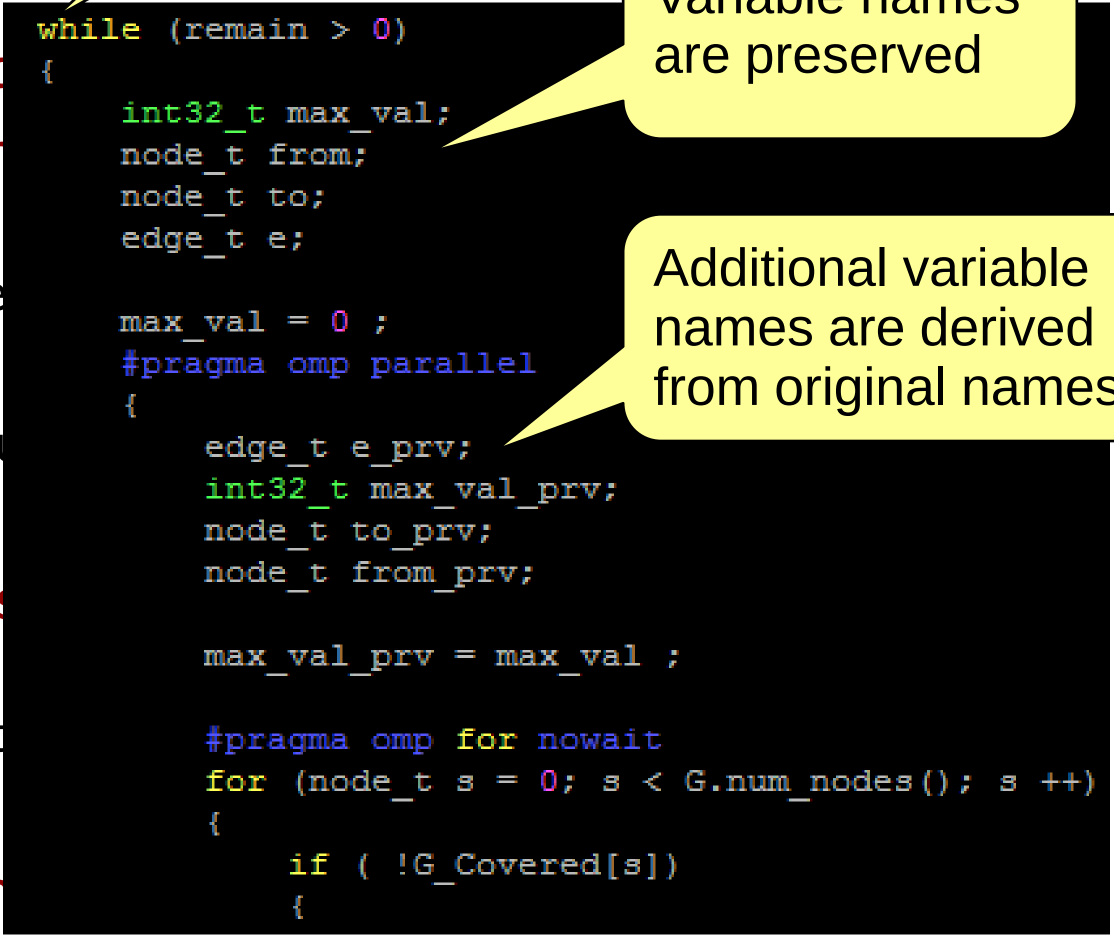
Hand-tuned Codes

“I, as an expert, can create faster code by hand-tuning.”

- Yes, I’m sure you can
    - DSL will be more helpful to non-experts. (Productivity)
- DSL enables rapid exploration of different algorithms
- You can manually enhance compiler-generated code
    - Compiler output is fairly human-readable C++ code
- DSL also provides portability

What about deb

- Yes, another good
- Currently, we're r
  C++ code level.
  - I.e. you can use
  - This is no harde
  - Generated outpu
- The compiler does
  - The compiler ca
    Green-Marl synt
- We also plan to in
  - Will look like a MATLAB for graph.



```
while (remain > 0)
{
    int32_t max_val;
    node_t from;
    node_t to;
    edge_t e;

    max_val = 0 ;
    #pragma omp parallel
    {
        edge_t e_prv;
        int32_t max_val_prv;
        node_t to_prv;
        node_t from_prv;

        max_val_prv = max_val ;

        #pragma omp for nowait
        for (node_t s = 0; s < G.num_nodes(); s ++)
        {
            if ( !G_Covered[s])
            {
```

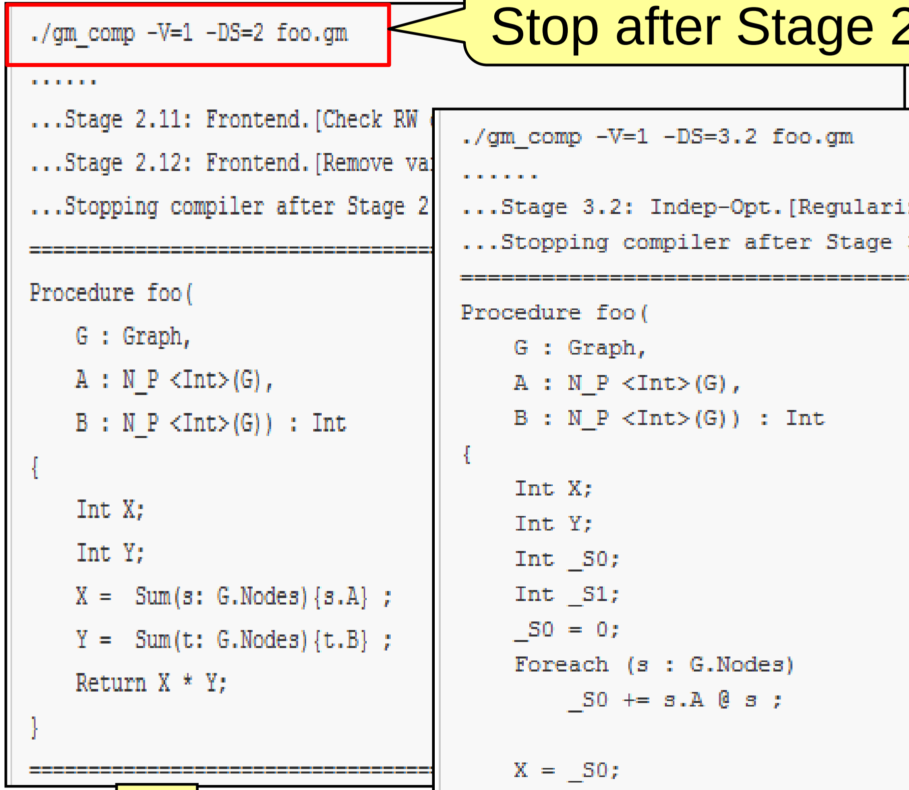
Generated codes are normal C++ program

Variable names are preserved

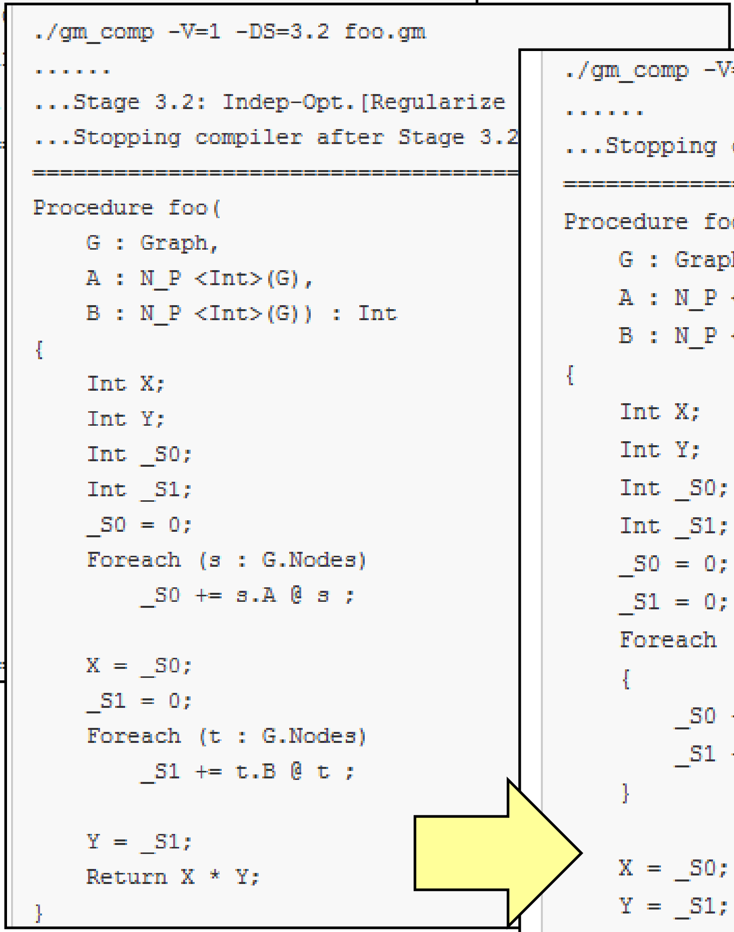
Additional variable names are derived from original names

Tracing the Compiler's Work

Verbose = on
Stop after Stage 2.



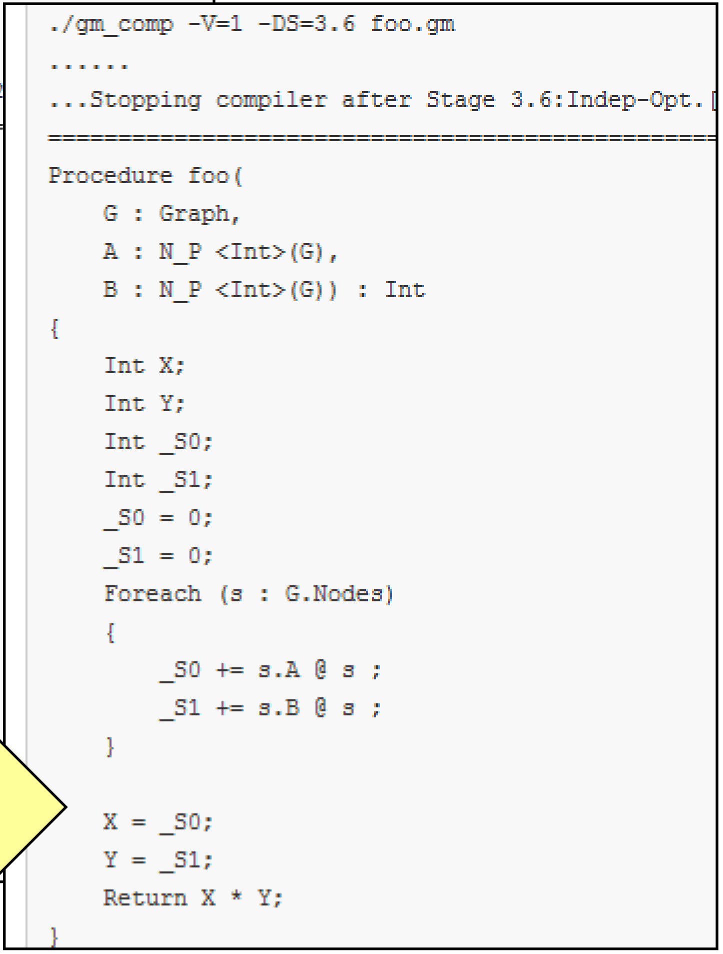
```
./gm_comp -V=1 -DS=2 foo.gm
......
...Stage 2.11: Frontend.[Check RW
...Stage 2.12: Frontend.[Remove va
...Stopping compiler after Stage 2
==================================
Procedure foo(
    G : Graph,
    A : N_P <Int>(G),
    B : N_P <Int>(G)) : Int
{
    Int X;
    Int Y;
    X =  Sum(s: G.Nodes){s.A} ;
    Y =  Sum(t: G.Nodes){t.B} ;
    Return X * Y;
}
==================================
```



```
./gm_comp -V=1 -DS=3.2 foo.gm
......
...Stage 3.2: Indep-Opt.[Regularize
...Stopping compiler after Stage 3.2
=====================================
Procedure foo(
    G : Graph,
    A : N_P <Int>(G),
    B : N_P <Int>(G)) : Int
{
    Int X;
    Int Y;
    Int _S0;
    Int _S1;
    _S0 = 0;
    Foreach (s : G.Nodes)
        _S0 += s.A @ s ;

    X = _S0;
    _S1 = 0;
    Foreach (t : G.Nodes)
        _S1 += t.B @ t ;

    Y = _S1;
    Return X * Y;
}
```



```
./gm_comp -V=1 -DS=3.6 foo.gm
......
...Stopping compiler after Stage 3.6:Indep-Opt.[
================================================
Procedure foo(
    G : Graph,
    A : N_P <Int>(G),
    B : N_P <Int>(G)) : Int
{
    Int X;
    Int Y;
    Int _S0;
    Int _S1;
    _S0 = 0;
    _S1 = 0;
    Foreach (s : G.Nodes)
    {
        _S0 += s.A @ s ;
        _S1 += s.B @ s ;
    }

    X = _S0;
    Y = _S1;
    Return X * Y;
}
```

Sums are expanded into loops

Loops are merged

Portability – Different Backends

- Different back-ends of Green-Marl
  - Cache-coherent shared memory: current
  - Pregel (Distributed Environment) : on-going
  - Cray XMT : early investment
  - GPU : early investment
  - GraphLab (a different run-time): idea brainstroming
  - Custom hardware: idea brainstorming
  - RamCloud: idea brainstorming

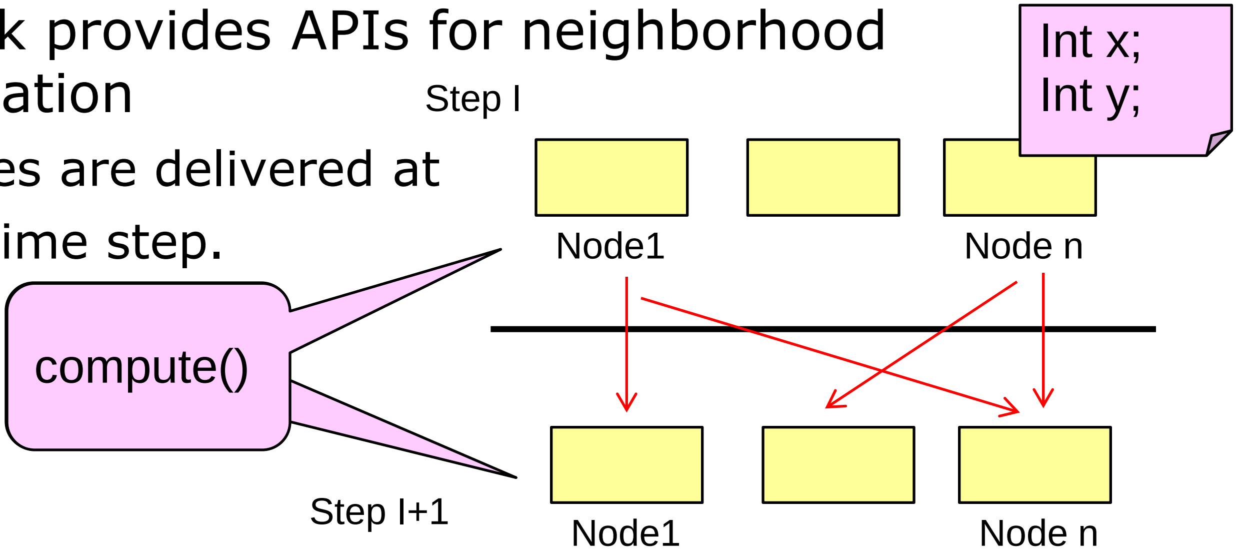
Capacity Issue in Graph Analysis

- Large graph + Associated data
  ≥ Main Memory

- Disk-based system (i.e. virtual memory) ?
  - A lot of *random* accesses ➔ disk latency kills you
- Stand-alone distributed program?
  - Large development overhead
- Map-Reduce (Hadoop)?
  - Unable to keep *state across iterations* ➔ performance loss
- ➔ Pregel (or its replicates)

Pregel (from Google)

- Map-Reduce like framework with enhancement
  - Iterative, Sensitive, Vertex-centric
  - A vertex can maintain its associated data
  - Single *compute*() function
    - Called for every vertex by the system
    - At each time step
  - Framework provides APIs for neighborhood communication
    - Messages are delivered at the next time step.



Implementation Issue

Automatic Translation?

- New Issu our algorithm has to converted for Preg API

```
// Count number of teen followers
// for each node(person) in a SN
Foreach(n: G.Nodes) {
  n.teenCount =
    Count(t:n.InNbrs)
       (t.age>=10 && t.age<20);
}
// Compute average number of
// teen-followers of people of
// certain age
Float avgAgeTeenFollowers =
    Avg(n:G.Nodes)(n.age>K)
       {n.teenCnt};
```

Your algorithm

```
class foo extends … {
……
 public void compute(…){
  if (step == 1) {
    if (this.age >= 10 &&
        this.age <= 20)
      sendNeighbors (
         new IntMessage(1));
  }
  else if (step == 2) {
    this.teenCount = 0;
    for(r: getReceived())
      this.teenCount +=
         r.IntValue();
  }
  else if (step = 3) {
     if (this.age > K) {
         ….
```

Pregel Implementation

Need context management

Need boilerplate code

Message Sending

Message is always *pushed*, not *pulled*

Message Receiving

need some tricks for *global* computation

Imperative

Some global-scoped sequential computation

Based on random *reading*

Issues to be solved

- Sequential computation
- Globally scoped variables
- Management of Execution Context
- Communication (message sending/receiving)
- Enforcing Push-based messaging

……

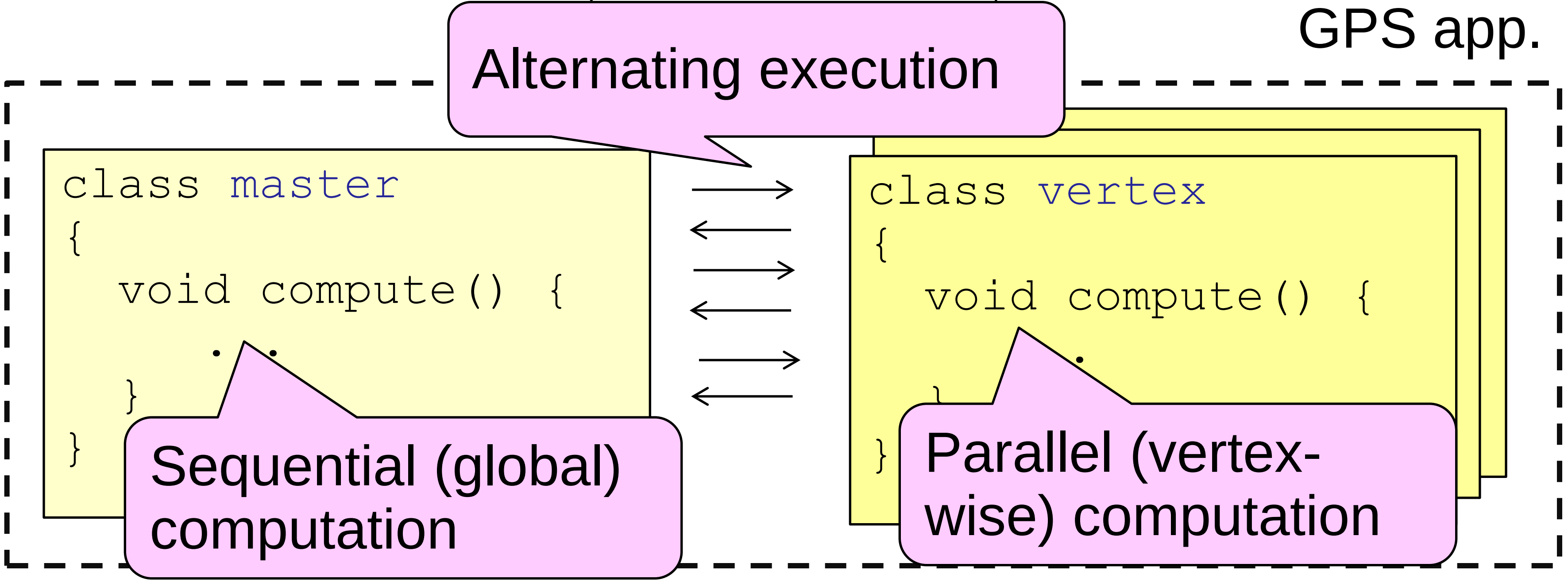
Our framework

- Pregel (from Google) is not open to public.
- GPS: an implementation of Pregel from Stanford, with Semih Salihoglu
- With enhancements
  - Optimized for performance

    ➔ x5~10 faster than Giraph (a popular Pregel implementation from Yahoo/Apache)
  - Elegant API for *global* objects and sequential computation

Handling Sequential Portion

- Your algorithm may include sequential portion
  - E.g. termination based on global sum of difference in page rank algorithm
- GPS provides a nice API for this:
  - `master class, master.compute()`


GPS app.
Alternating execution
class master { void compute() { . . } }
class vertex { void compute() { . } }
Sequential (global) computation
Parallel (vertex-wise) computation

Globally shared variables

- Another useful API: Global object map

```
class master {
 void compute() {
   ......
   global.put("x",
      new IntVal(1));

  }
}
```

Master puts an value object to the map

```
class vertex {
  void compute() {
   …
   int x=
     global.get("x")
        .intVal();
  }
}
```

Map is cleared at the end of each computation step

The object is broadcast to every vertices at following vertex-compute()

Compiler Translation: Global Object Management

```
Procedure foo(age,teenCnt:N_P<Int>,
   K: Int) {
…
Int S=0; // globally scoped
Foreach (n:G.Nodes)
  If (n.age>K)
    S += n.teenCnt;
```

Node property

```
class vertex {
 int age;
 int teenCnt;
 void compute() {
  …
  int K=
    global.get("K").intVal();
  if (this.age > K){
    global.put("S", new
     IntSumVal(this.teenCnt);
  }
  …
}
```

Compiler knows when the variable is used

master copy of global variables

```
class mast
 int S;
 int K;
 void comp  e() { …
  S = 0; …
  global.put("K", new IntVal(K));
 …
  S+= global.get("S").intVal();
… }
```

Reduction is implemented via special API

Compiler Translation: Execution Context & Sequential Portion

```
Foreach(n: G.Nodes) {
  n.teenCnt = …
}
```
(1)

```
Int S=0;
```
(2)

```
Foreach (n:G.Nodes) {
  If (n.age>K)
    S += n.teenCnt;
}
```
(3)

(4)

Compiler can figure out phases of algorithm

Compiler generates state-machine at master

Current state is broadcast to vertices

```
class master {
 int _state;
 void compute() {
  switch(_state) {
   case 1: do_state_1();
     ...
  } }
  void do_state_3() {
   global.put("K", new IntVal(K));
   startVertex = true;
   _stateNxt = 4; }
```

```
class vertex {
 ...
 void compute(..) { …
  int _state =
   global.get("_state")
               .intVal();

  switch(_state) {
   case 1: ..
  }
 }
 void do_state_3() {
  int K= …
  if (this.age > K)
    ...
 }
```

Compiler Translation: Communication

```
…
Foreach(n: G.Nodes){
  If (n.age >= 10 ...)
    Foreach(t: n.Nbrs) {
      t.teenCnt += 1;
    }
}
```

Nested loop implies communication

```
class vertex { ...
  void do_state_1() {
   if (this.age >= 10 … ) {
     sendNbrs(new Msg(…));
   }

  void do_state_2() {
   for(Msg r: getRcvd()) {
     this.teenCnt += 1;
   }
```

Communication is split into two consecutive states:
sending + *receiving*

Outer-loop becomes sending side

Inner-loop becomes receiving side

Enforcing Push-based algorithm

```
Foreach(n: G.Nodes)
  Foreach(t: n.Nbrs)
    t.X += f(t.Y, n.Z);
```

This nested loop is a *push.*

For every n, push n.Y to out-neighbor t to update t.X

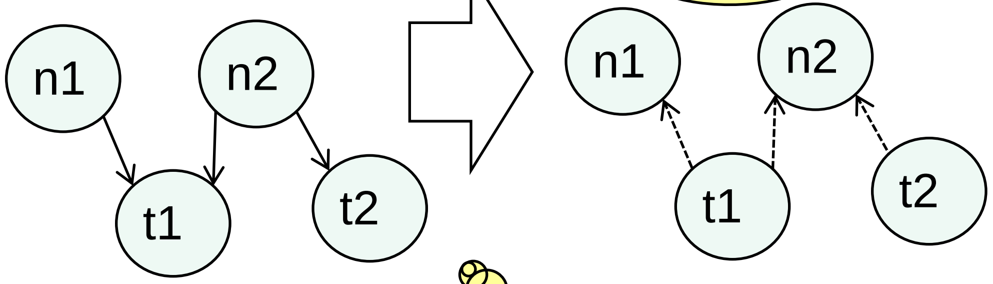
```
Foreach(n: G.Nodes)
  Foreach(t: n.Nbrs)
    n.X += g(t.Y, n.Z);
```

This nested loop is a *pull.* (cannot be implemented with API)

For every n, pull t.Y from out-neighbor t to update n.X

```
Foreach(t: G.Nodes)
  Foreach(n: t.InNbrs)
    n.X += g(t.Y, n.Z);
```

Compiler transforms it into *push* by *flipping edges*



For every t, push t.Y to *in-neighbor* n to update n.X

Compiler Transformation: Applying edge-flipping

```
Foreach(n: G.Nodes)
  n.teenCnt =
   Sum(t:n.InNbrs)(...){1};
```

Compiler changes Sum into Foreach

```
Foreach(n: G.Nodes)
  Int _S = 0;
  Foreach(t: n.InNbrs)(...)
    _S += 1;
  n.teenCnt = _S;
```

Replace scalar S with temporary node property Stmp

```
Node_Prop<Int> _Stmp;
Foreach(n: G.Nodes)
  n._Stmp = 0;
  Foreach(t: n.InNbrs)(...)
    n._Stmp += 1;
  n.teenCnt = n._Stmp;
```

Split Loops

```
Node_Prop<Int> _Stmp;
Foreach(n: G.Nodes)
  n._Stmp = 0;

Foreach(n: G.Nodes)
  Foreach(t: n.InNbrs)(...)
    n._Stmp += 1;

Foreach(n: G.Nodes)
  n.teenCnt = n._Stmp;
```

Edge Flipped

```
Node_Prop<Int> _Stmp;
Foreach(n: G.Nodes)
  n._Stmp = 0;

Foreach(t: G.Nodes)(...)
  Foreach(n: n.Nbrs)
    n._Stmp += 1;

Foreach(n: G.Nodes)
  n.teenCnt = n._Stmp;
```

There are still other details …

- Defining message class
- Merging states together
- Optimizing temporary node properties
- Merging congruent message classes
- ……

→ Current State:
  - → Can transform many algorithms into Pregel
  - → Compiler-generated code exhibits little overhead compared to hand-written code
  - → Still improving.

Conclusion

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

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

Thank you for attention

- Questions?

“*Programs must be written for people to read, and only incidentally for machines to execute*.”
-- Abelson & Sussman

Language Features

- For graph analysis
  - Built-in data types
  - Node and edge property
  - Collections
  - Graph iteration and traversal
- For parallel and distributed execution
  - Implicit parallelism
  - Consistency Model
  - Reduction
- For extensibility
  - Embedded foreign syntax

Types and Properties

- Green-Marl is statically-typed languages
  - Primitive types
  - Graphs (directed, undirected),
  - Node/Edge, Node/Edge properties
  - Collections
  - Foreign types (later)

```
Procedure foo(G: Graph,                    // Graph
              s: Node(G),                  // Node of G
              A,B: Node_Prop<Int>(G),      // Node Property of G
              C: Edge_Prop<Float>(G))
{
  // Property definition inside a scope
  Node_Prop<Int>(G) T;
  …
}
```

Types and Properties

- Node/Edge
  - **Node**(*graph*)
  - Bound to a graph instance
- Node/Edge Property
  - **Node_Prop**< *prim_type* > (*graph*)
- Collection Types
  - **Node_Set** (*graph*)
  - **Node_Order** (*graph*)
  - **Node_Seq** (*graph*)
  - **Node_Multiset** (*graph*)

```
Graph G1, G2;
Node(G1) n;
Node(G2) m;
n = m; // type error
```

	Unique-ness	Ordered-ness
Set	Y	N
Order	Y	Y
Sequence	N	Y
Multiset	N	N

Graph Iteration and Traversal

- Graph Iteration

```
Foreach(n : G.Nodes) (n.A > 0)
    …
```

`For` ➔ Sequential consistency

`Foreach` ➔ Parallel consistency

Iterator and Range

Graph. `Nodes/Edges`

Node. `Nbrs/InNbrs/OutNbrs`

`(UpNbrs/DownNbrs)` …

Set.`Items`

Filter; do not execute body if false

- Graph Traversal

Root

```
InBFS(n:G.Nodes From r)
    (n.A > 0) [n.color == 0]
{…}
```

`InDFS`➔ Depth-First Search Order

`InBFS` ➔ Breadth-First Search Order

`InRDFS,InRBFS` ➔ Reverse order traversal

Filter

Navigator; do not go further if false

Implicit Parallelism

- Parallel assignment
- Reduction expression

```
Graph G;
Node_Prop<Int>(G) x, y;

// parallel assignment
G.x = G.y + 1;
// Reduction (expression form)
Int z = Sum (t: G.Nodes) {t.x};
```

They are
Syntax sugars

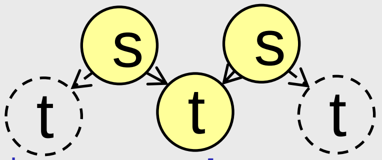
```
Foreach (n: G.Nodes)
  n.x = n.y + 1;

Int z = 0;
Foreach (t: G.Nodes)
   z += t.x; // Reduction (assignment form)
```

Consistency Model

- Sequential Consistency (For)
- Parallel Consistency (Foreach)
  - Things happen in parallel …
  - No ordering is guaranteed btwn concurrent loops
  - No visibility is guaranteed btwn concurrent loops
  - Use *reductions!*

```
Foreach (s: G.Nodes) {
  Foreach (t: s.Nbrs) {
    // Error (Warning)
    // (w-w conflict) multiple s can write to the same t.A
    // (r-w conflict) t.A can be read and written by different s.
    t.A = t.A + s.B*2;
  }
}
```



```
Foreach (s: G.Nodes) {
  Foreach (t: s.Nbrs) {
    // But compiler understands reduction
    t.A += s.B*2 ;
  }
}
```

Reductions

- Assignment Form

```
Int z = 0;
Foreach(n : G.Nodes)
    z += n.X;
```

- Expression Form

```
Int z = Sum(n:G.Nodes){n.X};
```

+=	Sum{}
*=	Product{}
&=	All{}
|=	Any{}
min=	Min{}
max=	Max{}

- Argmax/Argmin

```
Int x,z;
Node(G) m;
Foreach(n : G.Nodes)
    z <x, m> max= f(n.A) + n.B  <f(n.A), n> ;
```

z: Max
x, m: Argmax

Bulk Synchronous Consistency

- Deferred assignment

```
Foreach (s: G.Nodes) {
   // Reading t.A gives 'old' value
   s.A <= Sum (t: s.Nbrs) {t.A} @ s;
}
// modification to property A becomes
// visible at the end of s-loop
```

Loop bound indicator: tells to which loop this assignment is bound.
(e.g. nested loop)

Note

- A note on parallel/sequential consistency and parallel execution
  - The compiler (runtime) may execute a *foreach* loop sequentially.
  - The compiler (runtime) may execute a *for* loop in parallel, as long as it can guarantee sequential consistency.
    - E.g. transactional memory or locks

Data Access Analysis

Wset: (Z, -, always)
Rset: (B, Linear, always)
(A, Linear, cond)

```
Procedure foo (G:Graph, A,B: N_P<Int>(G); Z:INT)
{
   Int Y = 0;
   Foreach(x: G.Nodes)
   {
      If(x.B > 3)
         Y += x.A ;
   }
   Z = Y;
}
```

WSet: (Y, -, always),

Wset: (Y, -, cond)
Rset: (B, Linear, always)
(A, Linear, cond)

DSet: (Y, -, cond, (+=, x))
Rset: (B, x, always)
(A, x, cond)

DSet: (Y, -, always, (+=, x))
Rset: (A, x, always)

RSet: (Y, -, always),
WSet: (Z, -, always),

I'm not a graph guy. Do you suggest that I create my own DSL?

- Yes, I encourage you.
- Green-Marl is a stand-alone DSL, created from the scratch
  - This paper is written with 3 of my managers.
  - Current compiler was implemented in less than 6 months.
  - It is a doable job : (1) Type checker is simple. (2) Code generation is also not very complicated as you emit C++ code
  - Designing a good language is challenging, though.
- There are easier ways, though.
  - Innovations in Embedded DSL
  - Delite [H. Chafi et al., PPoPP'11] ➔ a framework for DSL creation
  - Green-Marl on Delite is also being developed.

Can every graph algorithm be written in Green-Marl?

- Good question. We hope so, don’t have proof.
  - We think we have all the necessary basic blocks
  - Basic node/edge iteration; graph traversal
  - Four collections (set/seq/order/bag)
  - Reductions
- Foreign syntax / Foreign type may help you
- Still, we are improving our language specification
  - We’re hearing from users, including professionals
  - Your opinion is valuable to us