Simplifying Scalable Graph Processing with a Domain-Specific Language

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Semih Salihoglu
Stanford University
semih@stanford.edu

Jennifer Widom
Stanford University
widom@stanford.edu

Kunle Olukotun
Stanford University
kunle@stanford.edu

ABSTRACT

Large-scale graph processing, with its massive data sets, requires distributed processing. However, conventional frameworks for distributed graph processing, such as Pregel, use non-traditional programming models that are well-suited for parallelism and scalability but inconvenient for implementing non-trivial graph algorithms. In this paper, we use Green-Marl, a Domain-Specific Language for graph analysis, to intuitively describe graph algorithms and extend its compiler to generate equivalent Pregel implementations. Using the semantic information captured by Green-Marl, the compiler applies a set of transformation rules that convert imperative graph algorithms into Pregel's programming model. Our experiments show that the Pregel programs generated by the Green-Marl compiler perform similarly to manually coded Pregel implementations of the same algorithms. The compiler is even able to generate a Pregel implementation of a complicated graph algorithm for which a manual Pregel implementation is very challenging.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Code generation*

1. INTRODUCTION

Large-scale graph processing often involves data sets that exceed the memory capacity of a single machine. Examples of such data sets include the web-graph, social networks and genomics data. Working with these large graph data sets requires a distributed graph processing framework, such as Google's *Pregel* [16] or its open source implementations [2, 18]. Pregel is a scalable distributed graph processing framework, in which the vertices of a graph are distributed across multiple machines in a cluster. Vertices, in synchronized timesteps, communicate with each other via messages to perform a graph computation.

CGO '14 February 15 - 19 2014, Orlando, FL, USA

ACM 978-1-4503-2670-4/14/02$15.00.
http://dx.doi.org/10.1145/2544137.2544162

Pregel adopts a non-traditional programming model, for the sake of maximizing parallelism and scalability. A graph algorithm is implemented as a single computation function written in a vertex-centric, message-passing and *bulk-synchronous* way. This function is invoked by the framework at every timestep (Section 2.1).

However, implementing sophisticated graph algorithms in Pregel is challenging, especially when the algorithm is composed of multiple computation kernels connected by non-trivial control flows, e.g. the Betweenness Centrality computation algorithm [6]. We summarize the challenges for implementing multi-kernel algorithms in Pregel as follows:

1. In Pregel's programming model, programmers must explicitly keep track of the global execution state inside the single computation function.
2. Pregel API allows only a sinlge message types. Therefore the programmer has to encode and decode different types of messages explicitly, if multiple types of messages are required by the algorithm.
3. Pregel's bulk-synchronous computation (BSP) model [21] only allows messages to be pushed. However, many conventional graph algorithms relies on data reading or pulling.
4. In Pregel, the graph algorithms should be mapped into vertex-centric functions explicitly, while many conventional graph algorithms are not designed in this model.

In addition, optimizing the performance of Pregel programs becomes more challenging as the vertex-centric computations, local vertex values, and message-types get more complex. Consequently, developing complicated algorithms using Pregel is a daunting task (Section 2.2).

In this paper, we show how one can express graph algorithms intuitively using *Green-Marl* [11], a high-level domain-specific language (DSL), and then automatically generate equivalent Pregel implementations. Using our approach, graph algorithms can be intuitively implemented in Green-Marl without addressing the challenges above. Specifically:

1. Green-Marl adopts an imperative programming style, allowing global control flows.
2. Green-Marl assumes random memory access, and thus there is no need to reason about messages at all.
3. In Green-Marl, data access is not bulk-synchronous but immediate; programmers need not speculate about which data will be required in the next timestep.

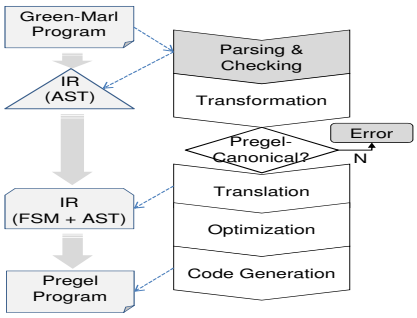

Figure 1: Compilation steps: Initially, the compiler uses annotated abstract syntax tree (AST) as internal representation (IR). After transformation step, the compiler uses another IR that is composed of both a finite state machine (FSM) and AST.

4. Green-Marl programs need not be vertex-centric, but are composed of high-level graph operations.

Given a Green-Marl program, our compiler first examines if the program is composed of certain frequent patterns that can be mapped into Pregel's programming model. If a Green-Marl program is composed solely of such patterns, we call such a program *Pregel-canonical*, the compiler translates it into an equivalent Pregel implementation that combines each pattern with proper state and message management code (Section 3). If the Green-Marl program is not Pregel-canonical, the compiler attempts to apply a set of transformations to turn the original Green-Marl program into its equivalent Pregel-canonical form. In addition, the compiler automatically applies a few performance optimization techniques that are typically applied to simple algorithms by Pregel programmers (Section 4). Our compilation steps are summarized in Fig 1.

Our experiments show that the performance of the compiler generated programs is as good as manual Pregel implementations of the same algorithms. Our compiler is even able to compile a Green-Marl implementation of Approximate Betweenness Centrality, whose manual Pregel implementation is prohibitively difficult (Section 5). Although this paper focuses on generating Pregel programs, our approach can be applied to other distributed graph analysis frameworks that have similar programming model characteristics and challenges (Section 6).

Our specific contributions are as follows:

- We identify common patterns in graph algorithms designed in an imperative, shared memory style that have a direct translation into Pregel's vertex-centric, message-passing programming model. We show how these translations are implemented in the Green-Marl compiler.
- We devise several transformation rules that can convert a subset of the Green-Marl programs that are not composed of the above common patterns (i.e. not Pregel-canonical) into equivalent Green-Marl programs consisting only of those patterns (i.e. Pregel-canonical). We show how these transformations are implemented in the compiler.
- We show two compiler optimizations which decrease the number of timesteps that the compiler generated Pregel programs take.
- We show experimentally that the performance of Green-Marl programs that are compiled into Pregel is similar to the manual Pregel implementations of the same algorithms. We also show that our compiler is able to compile a Green-Marl implementation of Approximate Betweenness Centrality, a complicated graph algorithm, into Pregel.

2. BACKGROUND

2.1 The Pregel Programming Model

We give a brief overview of Pregel [16] here. Broadly, the input is a directed graph, and each vertex of the graph maintains a local user-defined state. The computation is broken down into timesteps and terminates when all vertices are inactive in a particular timestep. Within a timestep *i*, each active vertex *u* in parallel: (a) looks at the messages that were sent to *u* in timestep $i-1$; (b) modifies its state; (c) sends messages to other vertices and optionally becomes inactive. A message sent in timestep *i* from vertex *u* to vertex *v* becomes available for *v* to use in timestep $i+1$. The behavior of each vertex is encapsulated in a function `vertex.compute()`, which is executed exactly once in each timestep. Computation assumes bulk-synchronous parallel model (BSP) [21], where global barrier synchronization is enforced at the end of each timestep. The framework buffers the messages sent during the execution of `vertex.compute()`, and as a result: (1) utilizes the underlying network bandwidth efficiently with large network packets; (2) overlaps computation with communication.

In this paper, we used GPS [18], an open-source implementation of Pregel. Noticeably, GPS introduces a second function `master.compute()`, in addition to `vertex.compute()`, which allows expressing of sequential parts of graph algorithms. Note that, however, this might make the Pregel programming even more challenging since the algorithm now has to be implemented in two single functions. We refer to GPS publication for details [18].[1]

2.2 Comparison of Programming Models

Pregel's programming model differs from the imperative style, shared-memory programming model in which many graph algorithms are typically expressed. In this paper, we use a high-level DSL, namely Green-Marl [11] to describe the same algorithm in a more intuitive or conventional way.

As an illustrative example, consider a graph algorithm for computing two statistical facts on a Twitter-like social network: (1) the number of teenage followers of every user, and (2) the average number of teenage followers of users over *K* years old. Figure 2 and Figure 3 show the Green-Marl and Pregel implementations of this algorithm respectively.

The Green-Marl implementation in Figure 2 is a more intuitive translation of the proposed graph algorithm. The program is written in an imperative style; it describes a sequence of statements to be executed in order without any notion of timesteps. The program is composed of a vertex-parallel computation (Lines 6–7) and a globally scoped computation (lines 10–11). The implementation assumes a shared-memory environment and performs random memory accesses (e.g. `t.age` in line 7).

The Pregel implementation of the same algorithm is shown in Figure 3. First, the Pregel implementation has to consider the notion of timestep In particular, the program has to manage the execution state based on the current timestep (lines 15, 21, and 27). Second, shared-memory data access is replaced with explicit message passing. Note that, due to BSP model, messages are received only in the next timestep (Line 19 and 24). Moreover, the direction of information flow is different. For example, instead of reading the incoming neighbors' age and counting the number of teenagers, each teenage node sends a message (containing the value 1) to its outgoing neighbors (lines 15 – 26). This is because Pregel only

[1]There also exists a variant [13] of this work where Green-Marl programs are compiled into Giraph, another Pregel implementation that adopts `master.compute()` API.

```
1  Procedure teenCnt (G: Graph,
2    K: Int, age: Node_Prop<Int>; // int-type node property
3    teenCnt: Node_Prop<Int>) : Float {
   // returns a float value
4  // For each node n in graph G, count the number
5  // of followers whose age is between 13 and 19
6    Foreach(n: G.Nodes)
7      n.teenCnt = Count(t:n.InNbrs)(t.age>=13 && t.age<=19);
8  // Compute the average of teenCnt,
9  // among node n whose is age larger K
10   Float avg = Avg(n:G.Nodes)(n.age>K){n.teenCnt};
11   Return avg;
12 }
```

Figure 2: Green-Marl Implementation of Calculating the Average Teenage Followers

```
13 public class TeenCnt extends ... {
14  public void compute(int timestepNo, ...) {
15   if (timestepNo == 1) {
16   // check my age, send 1 to nbrs
17    if (getValue().age>=13 && getValue().age<=19) {
18       message M = new message(1);
19       sendToNbrs(M);
20   }}
21   else if (timestepNo == 2) {
22   // add up ones from received messages
23    this.cnt = 0;
24    for(message M : rcvdMsgs())
25     this.cnt += M.intValue;
26   }
27   else if (timestepNo == 3) {
28   // check my age, send this.cnt
29    int k = getGlobalObject("k").intValue();
30    if (this.age > k) {
31     // increment sum by this.cnt, and num by 1. master
32     // will divide total sum by num to compute the avg
33     Global.put("sum", new intSum(this.cnt));
34     Global.put("num", new intSum(1));
35     ...
36 }}}} ...
```

Figure 3: Pregel Implementation of Calculating the Average Teenage Followers

allows messages to be *pushed* but not to be *pulled*. Third, computation of globally scoped data is implemented by defining and using special global objects (lines 29, 33, and 34). Finally, the Pregel implementation requires more boilerplate code not shown in the figure, such as the definition of master and message classes.

Although implementing algorithms in Pregel's programming model is manageable for simple algorithms as in Figure 2, it becomes very challenging for more complicated algorithms Consider for instance the *Betweenness Centrality* (BC) algorithm [6], which measures the relative importance of nodes in a graph. Note that even the multi-threaded implementations of this algorithm were considered worthy of publication [4, 15].

Figure 4 is the Green-Marl implementation of the Approximate Betweenness Centrality algorithm as described in the SNAP graph library [5]. The algorithm picks a random node `s` in the graph, and assigns a `sigma` value of 1 to `s` and a `sigma` value of 0 to other nodes. The algorithm then traverses the graph in breadth-first search (BFS) order from `s`, updating the `sigma` value of each visited node `u` by aggregating the `sigma` values of `u`'s BFS parents. The algorithm then traverses the graph in reverse BFS order, computing two other values for each node `u` (`delta` and `bc`) by aggregating values from `u`'s BFS children. These steps are repeated `K` times. In order to compute the exact BC value, one can replace the iteration over K random starting nodes with an iteration over all nodes in the graph. Note that this approximation is fast to compute – $O(Km)$ instead of $O(nm)$, where n is number of nodes and m is the number of edges – but only accurate enough to rank the vertices in large

```
37 Procedure bc_approx(G:Graph, K: Int ;
38                     BC:Node_Prop<Float>) {
39   G.BC = 0;   // Initialize BC as 0 per each node
40   Int i = 0;
41   Do {
42    Node_Prop<Float> sigma;
43    Node_Prop<Float> delta;
44    Node s = G.PickRandom();
45    G.sigma = 0;
46    s.sigma = 1;
47    InBFS(v: G.Nodes From s) { // BFS-order traversal
48       // Summing over BFS parents
49       v.sigma = Sum(w:v.UpNbrs) { w.sigma }; }
50    }
51    InReverse { // Reverse-BFS order traversal
52      v.delta =  // Summing over BFS children
53         Sum (w:v.DownNbrs) {
54            v.sigma / w.sigma * (1+ w.delta) };
55      v.BC += v.delta;  // accumulate delta into BC
56    }
57    i++;
58   } While (i < K); }
```

Figure 4: Green-Marl Implementation of Approximating Betweenness Centrality

social graphs and identify highly central nodes.

There are several challenges in implementing Approximate Betweenness Centrality with Pregel's API. First, it is not obvious how the control structures (e.g. `while` and graph traversal operations (e.g. `BFS`) can be expressed in a vertex-centric way. Second, it is also not obvious how the algorithm should be split into multiple timesteps and what messages should be exchanged at each step. Third, even after resolving the previous issues, implementing the algorithm in Pregel would require significant amount of code complexity to explicitly manage execution state and message types.

As we describe in Section 5, the Green-Marl compiler can successfully compile the program in Figure 4 into an equivalent Pregel implementation. Our compiler analyzes the program and applies several transformation and translation rules until an equivalent Pregel program is generated. The next two sections explain the details of these transformation and translation rules.

3. DIRECT COMPILER TRANSLATIONS

3.1 Translation of Pregel-Canonical Patterns

In this section, we explain how certain patterns that emerge in graph algorithms can be directly translated into Pregel's programming model. We also show how these translations are implemented in the Green-Marl compiler.

State Machine Construction

Pregel-compatible graph algorithms are decomposed into two alternating phases of computation: sequential computation and vertex-parallel computation. For instance, the following Green-Marl code consists of a sequential, a vertex-parallel, and finally another sequential computation phase.

```
59   // sequential computation
60   Int S = 0;
61   Int C = 0;
62   // vertex-parallel computation
63   Foreach(n: G.Nodes) {
64     if (n.age > K) {  // K,S,C: global variables
65         S += n.cnt;
66         C += 1;
67   } }
68   // sequential computation
69   Float val = (C == 0) ? 0 : S / (float) C;
```

Using the extended Pregel API of GPS, sequential and vertex-parallel computation can be naturally mapped into `master.compute()` and `vertex.compute()` methods, respectively. However, the two sequential phases in the example above do different computations and need to be executed inside the same `compute()` method. As a result, explicit state management code is required in the `master`.

`compute()` method to ensure that the correct computation is executed at the appropriate timestep.

The compiler recognizes these phases in a given input program by using control-flow analysis and creates a state machine. For instance, the compiler generates the following code from the example above:

```
70  class master extends ... {
71    int _state, _next_state;
72    public void compute(..) {
73      ...
74      switch(_state){ // state machine managed by master
75        case 0: do_state_0(..);break; // S=0,C=0;
76        case 1: do_state_1(..);break; // vertex parallel
77        case 2: do_state_2(..);break; // val=(C==0)?...
78      }
79      ...
80      // broadcast current state to the vertex objects
81      Global.put("_state", new IntValue(_state));
82    }
83    ...
84    private void do_state_1(..) { ...
85        _next_state = 2;  // sets the next state
86    } ...
87  }
88  class vertex extends ... {
89    public void compute(..) {
90      // receive current state from master
91      int _state = Global.get("_state").intVal();
92      switch(_state){
93        case 1: do_state_1(..);break;
94    } } ...
95    private void do_state_1(..) {
96     // actual vertex-parallel computation
97    ... }
98  }
```

Essentially, each parallel `Foreach` loop that iterates over the vertices in the graph is treated as a vertex-parallel state. The compiler-generated state machine is managed entirely by the master. The master broadcasts the current state number to the vertices via the global objects map (line 81), and vertices read the same state number from the map inside `vertex.compute()` (line 91).

Vertex and Global Object Construction

Variables in Pregel-compatible graph algorithms are grouped into global and vertex-local variables. Global variables are defined in the sequential phases and are visible to every vertex. On the other hand, unless explicitly communicated via messages, vertex-local variables are only visible to a single vertex.

The compiler analyzes the variables in the input program and implements global and vertex-local variables as field members in the generated master and vertex class, respectively. The compiler also ensures visibility of global variables using the global objects map. For instance, the compiler generates the following code from the Green-Marl code shown between lines 59 and 69:

```
99  class master extends ... { ...
100   int K, C, S; // global variables
101   private void do_state_1(..) {
102     Global.put("K", intValue(K)); // broadcast K
103     ...
104   }
105   private void do_state_2(...) { ...
106     // finalize reduction for state 1
107     S = S + Global.get("S").intVal();
108     C = C + Global.get("C").intVal();
109     // original code for state 2
110     float val = (C == 0) ? 0 : S / (float) C;
111   }
112 }
113 class vertex extends ... { ...
114   int age; // vertex-local variable
115   private void do_state_1(..) {
116     // receive broadcast from K
117     int K = Global.get("K").intValue();
118     if (this.age > K) {
119       // write to a global variable with sum reduction
120       Global.put("S", new intSum(this.count));
121       Global.put("C", new intSum(1));
122 } } }
```

Neighborhood Communication

Pregel provides an API for sending messages from a vertex to its neighboring vertices. This API naturally corresponds to a nested loop in Green-Marl programs where each vertex iterates over its neighbors. Consider the following example, where every vertex adds its `bar` variable into all of its neighbors' `foo` variables.

```
123 Foreach(n:G.Nodes)
124   Foreach(t:n.Nbrs)
125     t.foo += n.bar;
```

When this example is implemented using the Pregel API each vertex u sends its `bar` value to its neighbors as a message at a timestep. At the next timestep, each neighbor of u receives this message and accumulates it into its `foo` variable. As a result, in Pregel's BSP model, this computation takes two timesteps.

The compiler recognizes these nested-loop patterns and generates the corresponding message-passing Pregel code. The example code between lines 123 and 125 is translated as below:

```
126 class vertex extends ... { ...
127   private void do_state_1(..) {
128       Message m = new Message();
129       m.intValue = this.bar;
130       sendToNbrs(m);
131   }
132   private void do_state_2(..) {
133     for(Message m : rcvdMsgs()) {
134       int bar = m.intValue;
135       this.foo += bar;
136 } } }
```

The compiler determines the payload of the messages, via dataflow analysis of the nested loops. We define the following kinds of variables as *outer-loop scoped* variables: (1) a scalar variable defined in the outer-loop and (2) a vertex property accessed via an outer-loop iterator (e.g. `n.bar` in line 125). When an outer-loop scoped variable is accessed on the right-hand side (RHS) of a statement inside the inner-loop, the compiler adds it to the message payload. The compiler does not put the same variable multiple times in a message. In contrast, if an outer-loop scoped variable is accessed on a left-hand side (LHS) inside the inner-loop (i.e. modified inside the inner-loop), the compiler reports an error since the program requires messages to be pulled instead of pushed. We discuss how the compiler automatically transforms message pulling programs into message pushing ones by the **Edge-Flipping** rule in Section 4.1.

Multiple Communication

It is possible that an outer-loop contains multiple inner-loops. Since the inner-loops perform different computations, the Pregel translation of each inner-loop requires different message types. In such cases, the compiler attaches a tag to the message to identify the computation the message is used for. For example, the example between lines 137 and 144 will be compiled as the Pregel code shown between lines 145 and 162.

```
137 Foreach(n:G.Nodes){
138   if ((n.foo%2)==0) {
139     Foreach(t:n.Nbrs) // type1
140       t.even_cnt += 1;
141   } else {
142     Foreach(t:n.Nbrs) // type2
143       t.odd_cnt += 1;
144 } }
```

```
145 class vertex extends ... { ...
146   private void do_state_1(..) {
147     if ((this.foo%2)==0) {
148       Message m = new Message();
149       m.type = 1;
150       sendToNbrs(m);
151     } else {
152       Message m = new Message();
153       m.type = 2;
154       sendToNbrs(m);
155   } }
156   private void do_state_2(..) {
157     for(Message m : rcvdMsgs()) {
158       if (m.type == 1)
```

```
159             this.even_cnt += 1;
160           else if (m.type == 2)
161             this.odd_cnt += 1;
162   } } }
```

Random Writing

The Pregel API allows a vertex to send messages to any vertex with a known ID. In graph algorithms, this API corresponds to modifying properties of possibly non-neighboring vertices. The Green-Marl compiler distinguishes such random writes from previous **Neighborhood Communication** by checking whether a vertex is referenced by a random variable or a neighborhood iterator. In the following code, for example, line 167 is a random write but line 170 is neighborhood communication.

```
163  Foreach(n:G.Nodes)
164    If (n.Degree()==0||Uniform()<0.2) {
165      // pick any random node in the graph G
166      Node s = G.PickRandom();
167      s.foo += n.bar; // random write
168    } Else {
169      Foreach(s2:n.Nbrs)
170        s2.foo += 1;  // neighborhood communication
171    }
172  }
```

The message payload for random writes is determined by the same dataflow analysis as in the neighborhood communication pattern. Therefore, The compiler generates the following code from the example above:

```
173  class vertex extends ... { ...
174    private void do_state_1(..) {
175      if(getNumNbrs()==0||Random.nextDouble()<0.2){
176        node s = Ramdom.netxInt(getGraphSize());
177        Message m = new Message();
178        m.type = 1;
179        m.intVal = this.bar;
180        sentToNode(s, m);
181      } else {
182        Message m = new Message();
183        m.type = 2;
184        sentToNbrs(m);
185    } }
186    private void do_state_2(..) {
187      for(Message m : rcvdMsgs()) {
188        if (m.type == 1)
189            this.foo += m.intVal;
190        else if (m.type == 2)
191            this.foo += 1;
192  } } }
```

Edge Properties

The Green-Marl language specification allows edges to have properties. Pregel makes the property of an edge (u, v) accessible only from the source vertex u. As a result, the compiler allows edge properties to be accessed only when u is iterating over its neighbors as in line 196 below.

```
193  Foreach(n:G.Nodes){
194    Foreach(t:n.Nbrs){
195      Edge e = t.ToEdge(); // e is the edge from n to t
196      n.dist min= t.dist + e.len; // min-reduction
197  }}
```

The compiler generates the following code from the example above:

```
198  class vertex extends ... { ...
199    private void do_state_1(..) {
200      for (Edge e: getEdges()) {
201        Message m = new Message();
202        m.intVal = this.dist + e.len;
203        sentToNode(e.getDest(), m)
204    } }
205    private void do_state_2(..) {
206      for(Message m : rcvdMsgs()) {
207        int i0 = m.intVal;
208        if (this.dist < i0) this.dist = i0;
209  } } }
```

3.2 Pregel-canonical Programs

A Green-Marl program is Pregel-canonical if it only consists of patterns of Section 3.1. Specific conditions for Pregel-canonical programs are as follows:

- **Finite State Management**: The program is non-recursive and has at most one directed graph as an argument in the entry function. It can have any number of `If-Then-Else` and `While` constructs.
- **Parallel Vertex and Neighborhood Iteration**: Parallel `Foreach` can be doubly nested (not any deeper) in which case the outer-loop iterates over all the vertices in the graph and the inner-loop iterates over the neighboring vertices of the outer-loop iterator. `Return` is not allowed in such loops.
- **Message Pushing**: Inside a `Foreach` loop that iterates over a vertex u's outgoing neighbors, u's neighbors do not modify u's value.
- **Random Writing**: Writing a property of a random vertex occurs only in a vertex-parallel phase of the program. Random reading of a vertex property is not allowed.
- **Edge Property**: The property of an edge (u, v) is accessed only through the source vertex u.

The compiler recognizes Pregel-canonical Green-Marl programs and translates them into equivalent Pregel (GPS) programs by applying the rules discussed in this section. The next section discusses how the compiler attempts to transform programs that are not Pregel-canonical into equivalent Pregel-canonical forms.

4. OTHER COMPILATION STEPS

4.1 Transformation for Non-Pregel-Canonical Programs

In case where a given input Green-Marl program is not Pregel-canonical, the compiler applies various program transformations to try to convert it into Pregel-canonical form. If successful, the transformed program can then be directly translated into a Pregel implementation using the translation rules explained in the previous section. Otherwise, the compiler reports an error.

Flipping Edges: Converting Message Pulling into Message Pushing

Pregel-canonical Green-Marl programs satisfy the *message pushing* condition from Section 3.2. However, there are some graph algorithms that are naturally described using message pulling, instead of pushing. Consider the following example where each node finds the maximum value of `bar` across its incoming neighbors:

```
210  Foreach(n:G.Nodes)
211   Foreach(t:G.InNbrs)   // maximum of in-nbrs
212     n.foo max= t.bar;   // this requires message pulling
```

Note that the above program requires message pulling as each `n` reads the `bar` value of its *incoming* neighbors. In such cases, the compiler transforms the program into a functionally equivalent Pregel-canonical program:

```
213  // loop iterators are exchanged with each other.
214  Foreach(t:G.Nodes)
215   Foreach(n:G.Nbrs)   // (Out)Nbr instead of InNbr
216     n.foo max= t.bar; // this requires message pushing
```

Note that, in the above example, the complier has swapped the two `Foreach` loop and flipped the direction of information flow. Each `t` now writes its `bar` value to each of its *outgoing* neighbor `n`.

The exact mechanism of the above transformation is as follows:

1. The compiler identifies nested `Foreach` loops (i.e. neighborhood iterating pattern) where the outer-loop does not have any statements other than the inner-loop and the inner-loop *only* updates outer-loop scoped variables.

2. The compiler switches the iterators of the two loops and flips the edge direction of the inner loop iteration (e.g. (Out)Nbrs instead of InNbrs).

Dissecting Nested Loops: Preprocessing for Edge Flipping

Note that condition (1) of the **Flipping Edges** transformation is rather restrictive. Thus, the compiler applies following pre-processing transformations to make it easier applying **Flipping Edges**. Consider the following example:

```
217  Foreach(n: G.Nodes){
218   Int _C = 0;
219   Foreach(t: n.InNbrs)(t.age>=13&&t.age<=19)
220     _C += 1;
221   n.cnt = _C;
```

The above example is not Pregel-canonical, since an outer-loop scoped scalar variable `_C` is modified inside the inner-loop (line 220). Whenever the compiler recognizes this pattern, it introduces a temporary vertex property to replace the scalar variable. Therefore the above code becomes:

```
222  Node_Prop<Int> _tmp;
223  Foreach(n: G.Nodes){
224   n._tmp = 0;
225   Foreach(t: n.InNbrs)(t.age>=13&&t.age<=19)
226     n._tmp += 1;
227   n.cnt = n._tmp;
```

Next, the compiler sees that an outer-loop scoped vertex property (`n._tmp`) is modified in the inner-loop; however the outer-loop still contains statements other than the inner-loop. In this case, the compiler splits the outer loop into multiple loops so that the inner-loop becomes the sole member of a newly created outer-loop. For instance, the nested loop in the above example is split into three loops.

```
228  Node_Prop<Int> _tmp;
229  Foreach(n: G.Nodes)
230   n._tmp = 0;
231  Foreach(n: G.Nodes)
232   Foreach(t: n.InNbrs)(t.age>=13&&t.age<=19)
233     n._tmp += 1;
234  Foreach(n: G.Nodes)
235    n.cnt = n._tmp;
```

Note that the compiler can now apply the edge-flipping transformation to the second loop between lines 231 and 233), which makes the entire program Pregel-canonical.

Random Access in Sequential Phase

While Pregel has no native support for the reading and writing of an random node's properties, the compiler allows such patterns by transforming the random access into an extra parallel loop as in the following example:

```
236  s.dist = 0;  // a random write in sequential phase


237  Foreach(n:G.Nodes) {
238    if (n==s) n.dist = 0;
239  }
```

BFS-order Graph Traversal

Green-Marl provides a language construct for a BFS-order traversal of the graph with the users supplying their own custom computation to occur during the traversal. The compiler transforms a BFS traversal statement into a set of Pregel-canonical statements. Specifically, the BFS traversal is transformed into iterative frontier expansions. For example, consider the following Green-Marl code:

```
240  InBFS(n: G.Nodes From s) { // BFS-order traversal
241    ... // user code
242  }
```

This above code is transformed into the following Pregel-canonical code:

```
243  {
244   Node_Prop<Int> _lev; // hop-distance from root
245   Bool _fin=False;
246   Int _curr = -1;
247
248   G._lev=+INF; // initialize distance as INF
249   s._lev=0;    // root node has zero-distance
250   While(!_fin){ // level-wise frontiner expansion
251    _fin=True; _curr++;
252    Foreach(n:G.Nodes)(n._lev==_curr){
253      Foreach(t:n.Nbrs)(t._lev==+INF){
254        // expand unreached nbrs
255        t._lev=_curr+1; _fin=False;
256      }
257      ... // user code
258  } } }
```

In the transformed code `_lev` is a compiler-inserted node property which stores the depth of BFS tree. Also note that the user-provided custom computation in the original program (line 241) is fused with the expanded BFS code (line 257). However, if the *user code* iterates over the BFS-parents or the BFS-children, the compiler generates an extra loop to implement the user code. For instance, lines 259–262 is transformed into lines 263–272.

```
259  InBFS(n: G.Nodes From s) { // BFS-order traversal
260    Foreach(t:n.DownNbrs) // iterate over BFS children
261      ... // more user code
262  }


263  // after transformation
264   Foreach(n:G.Nodes)(n._lev==curr){
265     Foreach(t:n.Nbrs)(t._lev==+INF){
266       // expand unreached nbrs
267       t._lev=_curr+1; _fin=False;
268    } }
269   Foreach(n:G.Nodes)(n._lev==curr){ // a separate loop
270     Foreach(t:n.Nbrs)(t._lev==_curr+1){ // BFS children
271       ... // more user-code
272    } }
```

A reverse BFS traversal, or a traversal of connected vertices in a graph in reverse topological order from the root vertex, is implemented in a similar fashion. Since a reverse BFS traversal is always preceded by a forward BFS traversal, it is converted into an additional `While` loop after the forward BFS traversal which checks `_lev` in decreasing order, as in the following example:

```
273  InBFS(n: G.Nodes From s) {...} // Forward BFS
274  InReverse {
275    ... // user code
276  }
```

The above code is tranformed into the following Pregel-canonical code.

```
277  {
278    ... // Forward BFS
279    While (_curr >= 0) { // Reverse BFS
280      Foreach(n:G.Nodes)(n._lev==_curr){
281        ... // user code
282      }
283      _curr--;
284  } }
```

4.2 Performance Optimizations

State Merging

Whenever it is safe to do so, the compiler merges two consecutive states of vertex computation into one. Consider the following example.

```
285  Foreach(n: G.Nodes)(n.age>=13&&n.age<20){
286    Foreach(t: n.Nbrs)  // nbr iteration -> state 1,2
287      t.cnt += 1;
288  }
289  Foreach(n: G.Nodes)(n.age>K){ // loop --> state 3
290    _sum += n.cnt;
291  }
```

According to the translation rules in Section 3.1, the compiler generates a state machine consisting of three states: 1) nodes with age

values between 13 and 20 send an integer value of 1 to their outgoing neighbors; 2) each node aggregates received messages from state 1 into their `cnt` variables; 3) nodes with age values over K add their `cnt` values to a global `_sum` variable. The first two states come from the first nested loop, and the third state comes from the last loop in line 289. However, states two and three can be executed in a single timestep because there is no data dependency between them. After observing data dependencies, the compiler merges states two and three into a single state (state 4 in line 297) below.

```
292  private void do_state_1(..) { // sending messages
293    if (this.age>=10 && this.age<20){
294      Message m = new Message();
295      sendToNbrs(m);
296  } }
297  private void do_state_4(..) {  // state 2, 3 merged
298    // (merged) state2: receiving messges
299    for(Message m : rcvdMsgs()) {
300       this.cnt += 1;
301    }
302    // (merged) state3: updating global sum
303    if (this.age > K) {
304      Global.put("_sum", new intSum(this.cnt));
305  } }
```

Note that the Pregel framework incurs a certain amount of overhead at the end of each timestep due to global barrier synchronization across the cluster. This optimization reduces this overhead.

Intra-Loop State Merging

This optimization merges the first and the last vertex-parallel states inside a while-loop. The simple case is when there is only one neighborhood communication pattern inside the while-loop. Consider the following example:

```
306  While(condition) {
307   Foreach(n: G.Nodes) {
308     Foreach(t: n.Nbrs) {
309       t.foo += n.bar;
310   } }
311   ...  // update condition
312  }
```

The neighborhood communication pattern inside the while loop is translated into two states: (1) for sending `bar` values and (2) receiving those values and summing them up into `foo`. Normally, these two states cannot be merged together because message delivery takes two timesteps under BSP. However, leveraging the fact that the nested-loop is inside a while-loop, the compiler merges the two states in the following way:

```
313  private void do_state_1(..) {
314    if (!_is_first) { // (merged): receiving messages
315      for(Message m : rcvdMsgs())
316       this.foo += m.intVal;
317    }
318    { // (merged): sending messges
319      Message m = new Message();
320      m.intVal = this.bar;
321  } }
```

In the above code, `_is_first` is a compiler-inserted flag that is set during the first entry into the while-loop and unset after the first iteration. Essentially, the compiler has merged the receiving state for the current iteration of the while-loop, with the sending state for the next iteration of the while-loop.

The generalized mechanism of this intra-loop state merging is illustrated in Figure 5. There are N vertex-parallel states and (N+1) sequential states inside a while-loop. Any of these sequential states can be empty. The compiler merges the last states (PAR N and SEQ N) of the current while-iteration with the first states (SEQ 0, PAR 1 and SEQ 1) of the next while-iteration. The compiler also inserts additional control structures in the state machine (e.g. `is_first`) so that the states are executed in the correct order.

However, the merged state machine (Fig 5.(b)) differs from the original (Fig 5.(a)) in two ways: (1) PAR 1 state and SEQ N state are executed out-of-order. (2) SEQ 0, PAR 1, and SEQ 1 states are

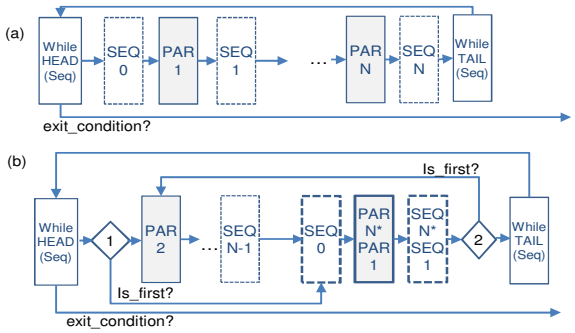

Figure 5: Intra-loop State Merging: (a) is the state machine before merging, (b) is the one after merging. The merged states PAR N* and SEQ N* are not executed when `is_first` is true. `is_first` is set during the first entry into the while-loop and cleared after state `2` (in the figure) is first executed.

executed one more time than in the original program. Also dangling messages are sent at the last-stage of the loop but are safely dropped by the system as they have no side effect. The compiler checks whether these differences do not alter the semantics of the original program via data flow analysis before applying this optimization.

4.3 Pregel Code Generation

Incoming Neighbors

The Pregel API only allows sending messages to outgoing neighbors. However, there are graph algorithms that iterate over incoming neighbors as well. The compiler automatically resolves this issue with the following compiler steps. First, the compiler analyzes the Green-Marl program to see if incoming neighbor iteration is used. If so, the compiler inserts two extra steps at the beginning of the generated Pregel program to create a list of incoming neighbors for each vertex. These two extra steps are as follows:

```
322  private void do_state_0(..) { // send ID to nbrs
323    Message m = new Message();
324    m.intValue = this.getID();
325    sendToNbrs(m);
326  }
327  private void do_state_1(..) { int i=0;
328    // create list of incoming neigbors
329    this.in_nbrs = new int[rcvdMsgs().length];
330    for(Message m : rcvdMsgs()) {
331      this.in_nbrs[i++] = m.intValue;
332  } }
```

After these two steps, the incoming neighbors of each vertex are stored as a local vertex value, inside an integer array, called `in_nbrs`. Then, the compiler uses this array to generate code that sends messages from a vertex to its incoming neighbors as demonstrated in the following example:

```
333  private void do_state_2(..) {
334    for (int n: this.in_nbrs) {
335      Message m = new Message();
336      ... // fill in massage payload
337      sendToNode(n, m)
338  } }
```

Message Class and Input/Output (I/O) Methods

The compiler generates all the boilerplate code required for Pregel applications. One example is the serializable message class which every Pregel application must declare. The compiler automatically generates the definition of this class, including serialization and deserialization methods. The compiler uses the same semantic information it uses when determining the payload of messages (Section 3.1). For example, here is a generated message class in which the payload can either be an `int` or a `double`:

Name	Nodes	Edges	Description
Sk-2005	51M	1.9B	Web graph of .sk domain
Twitter	42M	1.5B	Twitter follower network
Bipartite	75M	1.5B	Synthetic (Uniform Random)

Table 1: Input graphs: Twitter and Sk-2005 graphs were provided by *The Laboratory for Web Algorithmics* [3].

```
class Message extends ... {
  byte type; // can be short or long
  int intValue;
  double doubleValue;
  public void serialize(Buffer B) {
    B.putByte(type);
    if (type==0)
      B.putInt(intValue);
    else if (type==1)
      B.putFloat(intValue);
  } ...
  public void deserialize(Buffer B) {
    type = B.getByte();
    if (type==0) {
      intValue = B.getInt();
    } else if (type==1) {
      floatValue = B.getFloat();
} } }
```

Notice that the compiler has inserted a tag field called `type` to distinguish different message payloads used in the program. This is because a Pregel application can declare only one message class. This tag field can be optimized out if the following conditions are met: (1) if every message uses the same payload type and (2) there is no **Multiple Communication** (Section 3.1) in the program.

The compiler automatically generates I/O methods in the Pregel program as well. These I/O methods are generated from the parameter declaration of the entry procedure of the Green-Marl program. Scalar input variables are mapped to command-line arguments while scalar output variables are printed to standard output. Input properties are read from the input file at initialization, and output properties are dumped into the output file at finalization.

5. EXPERIMENTS

In this section, we compare the Green-Marl and manual Pregel implementations of six algorithms. We first demonstrate the productivity benefits of programming in Green-Marl by comparing the lines of code each implementation of our algorithms took. Then, we present experiments comparing the performance of the compiler-generated and manual implementations of our algorithms on GPS.

The input graphs and the algorithms we used in our experiments are listed in Table 1 and Table 2. The algorithms were selected from the original Pregel [16] and Green-Marl [11] papers in order to exercise various aspects of each programming model. We ran all of our experiments on the Amazon EC2 cluster using 20 large instances (4 virtual cores and 7.5GB of RAM) running Red Hat Linux OS. We repeated each experiment five times. The results are the averages across all runs ignoring the initial data loading stage. Measured performance varied by only small amounts across multiple runs.

5.1 Productivity Benefits

Table 2 shows the lines of code (LOC) required for implementing each algorithm in Green-Marl and manually with the Pregel API. As shown, Green-Marl programs are between 4x and 12x shorter than their manual Pregel implementations. Manually coded Pregel programs are significantly longer because they require extra code for state management and message sending/receiving, which are avoided in Green-Marl. Manually coded Pregel programs also require lengthy boilerplate code, such as defining vertex and message types, global objects, and serialization/deserialization functions for the vertex and message types.

Algorithm	Origin	Green-Marl	Native GPS
Average Teenage Follower (*AvgTeen*)	-	13	130
PageRank	[16]	19	110
Conductance (*Conduct*)	[11]	12	149
Single Source Shortest Paths (*SSSP*)	[16]	29	105
Random Bipartite Matching (*Bipartite*)	[16]	47	225
Approximate Betweenness Centrality(*BC*)	[11]	25	N/A

Table 2: Comparisons of line of codes for implementing Graph algorithms: Green-Marl and native GPS implementation

Transformation	*AvgTeen*	*PageRank*	*Conduct*	*SSSP*	*Bipartite*	*BC*
State Machine Const.	✓	✓	✓	✓	✓	✓
Global Object	✓	✓	✓	✓	✓	✓
Multiple Comm.						✓
Random Writing					✓	
Edge Property				✓		
Flipping Edge		✓		✓		✓
Dissecting Loops		✓		✓		✓
Random Access(Seq.)						✓
BFS Traversal						✓
State Merging	✓	✓	✓	✓	✓	✓
Intra-Loop Merge		✓		✓	✓	✓
Incoming Nbrs					✓	✓
Message Class Gen	✓	✓	✓	✓	✓	✓

Table 3: List of Compiler Transformations Applied for Each Algorithm

The Green-Marl programs are not only shorter but also more intuitive. Our main argument is that, as a DSL, Green-Marl provides a higher-level of abstraction for designing and implementing graph algorithms. We provide the Green-Marl and implementations in the Appendix. The manual Pregel implementation of the same algorithms (other than Betweenness Centrality) can be found in our tech report [12] for reader's comparison.

Table 3 summarizes the list of transformation steps applied when compiling each algorithm. Basic steps such as **State Machine Construction**, **Global Object**, and **Message Class Generation** are commonly applied to all of the algorithms. **State Merging** transformation is also common; all the algorithms contain consecutive parallel loops to which the transformation can be applied. Other transformation steps are, however, applied only when certain patterns exist in the algorithm. For instance **Edge Properties** or **Incoming Neighbors** are maintained only when they are used by the algorithm.

The most challenging algorithm to implement manually in Pregel was *Approximate Betweenness Centrality*, for which we did not have an existing Pregel implementation. The compiler transformed the program (Figure 4) into Pregel-canonical by applying multiple transformation rules in Table 3. The complexity of the compiler-generated Pregel program substantiate our claim that implementing this algorithm in Green-Marl is much more convenient than manually implementing with Pregel's API. The generated Pregel program consists of nine vertex-centric kernels and four different message types. Code for state management and message passing is accordingly non-trivial. Each kernel performs a different computation and communication. The BFS traversal kernels are fused with the user-provided computation. Programmers can manually implement this algorithm in Pregel's programming model, essentially by repeating the same steps our compiler applies. However, such a task would require a lot more effort than writing the short Green-Marl code in Figure 4.

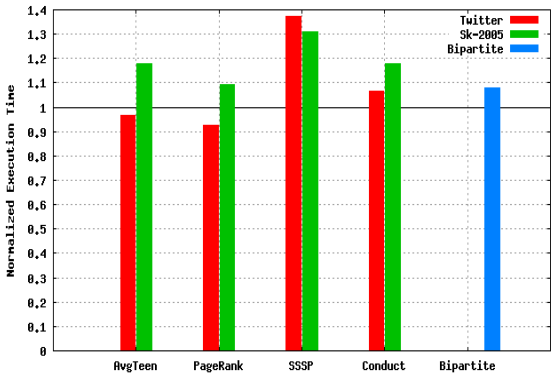

Figure 6: Run-time Comparison of Compiler-Generated and Manually Coded Pregel Programs.

5.2 Performance Comparison

In this section we evaluate the performance of Pregel programs generated by the Green-Marl compiler. For five of our algorithms, we executed both the manually coded Pregel implementation and the compiler-generated Pregel program on the same input graphs. In each experiment, we measured the run-time, the network I/O due to sending/receiving messages, and the number of timesteps of the entire execution.

The run-time results of our experiments are shown in Figure 6. Each bar in the figure represents the result of a single experiment on a particular algorithm and an input graph. The height of each bar is the run-time of the compiler-generated Pregel program normalized against its manual Pregel implementation. As shown, the run-time performances of the compiler-generated Pregel programs are comparable to the manual implementations across all our algorithms. The run-time performances of the compiler-generated implementations varied between 8% speedups to 35% slowdown. We note that the 35% slowdown was observed for the experiment of finding the shortest paths from a single source on the Twitter graph.

Surprisingly, there were some cases when the compiler-generated programs even outperformed hand-tuned Pregel implementation. To our understanding, this was due to the effect of garbage collection behavior of the JVM. However, we did not find enough generalization to make an optimization step out of this phenomenon.

On the other hand, the run-time overhead of the compiler-generated programs can be explained from two reasons. First, the compiler-generated programs manage their state machines via broadcasting global objects. However, for simple algorithms it is possible to manage states using the timestep number, which is made available to the vertices by the Pregel framework without any overhead. Second, the compiler does not yet utilize Pregel's `voteToHalt()` API to inactivate converged vertices. Inactivating vertices speeds up the execution time because the framework skips calling the `compute()` function on inactive vertices. The compiler currently does not use this API because in the general case, when an algorithm has multiple vertex-parallel phases, inactive vertices in one phase may become active again in another phase. The overhead due to lack of support of `voteToHalt()` is more visible in single-kernel algorithms that have many timesteps with few active vertices. For example, when running SSSP on the Twitter graph, less than 1.5% of the vertices were active in the last 30 timesteps.

The compiler-generated programs took the exact same number of timesteps and incurred the exact same network I/O as the manually coded Pregel programs. This exact match is explained by our observation that all the translation and transformation rules that our compiler applies to Green-Marl programs are what programmers typically do when implementing algorithms manually using the Pregel API. Consequently, the compiler-generated implementations run in virtually the same way as the manual implementations do and exhibit similar performance behavior.

Finally, we leave some comments about the completeness issue of our approach. There are inherently sequential algorithms (e.g. Tarjan's SCC algorithm) that can be described in Green-Marl but not with Pregel. To the contrary, since the Pregel-canonical syntax is a direct mapping to Pregel API, it is straightforward to write Pregel programs with Green-Marl. See the related discussion in Appendix for details.

6. RELATED WORK

This paper shows how we can translate Green-Marl programs into the Pregel API [16]. Pregel is not the only large-scale distributed graph analysis framework. Other frameworks include ActivePebbles [22] and GraphLab [14] which also have a vertex-centric computation model. Trinity [20] is a proprietary graph computation system at Microsoft which is built on top of a distributed RAM-based key-value store. There are also general distributed computation framework which can be used for graph analysis. For example, SPARK [23] is a general in-memory cluster computing framework that can also handle iterative computation. HaLoop [7] and Surfer [8] are systems built on top of Hadoop [10] which can do iterative MapReduce [9] computations. All of the above frameworks introduce different programming models and APIs and demand the programmers to re-implement their graph algorithms accordingly. Large-scale graph processing on these systems can also be simplified by our approach of compiling Green-Marl programs into their respective APIs.

This work uses Green-Marl [11], an existing DSL for graph processing. Many DSLs have been used to provide high-level abstractions for different application domains. For instance, Pig Latin [17] and Hive [19] provide a higher level programming interface for creating MapReduce jobs. However, these languages do not provide graph-specific semantic information which is essential to our approach. In contrast, Gremlin [1] is a graph traversal language that is designed for querying on-line graph databases. However, we think that Green-Marl provides a more intuitive and convenient abstraction for describing graph analysis algorithms. Nevertheless, our approach is not specific to Green-Marl and can be used with any DSL that has enough semantic information about a given graph algorithm.

7. CONCLUSION AND FUTURE WORK

We presented a solution to the programmability problem associated with Pregel. We use Green-Marl, a graph DSL, to describe graph algorithms intuitively and then let our compiler automatically generate an equivalent and optimized Pregel implementation. This automatic translation is enabled by the semantic information captured by the Green-Marl language. We expect that our compiler based approach will be most useful for complicated graph algorithms whose manual Pregel implementation is very challenging.

As for future work, we are adding more static analyses, which can improve the performance of our compiler-generated programs as well as more transformation rules that allows compilation of more non-Pregel-canonical Green-Marl programs.

Acknowlegement

This work was funded by DARPA Contract, Xgraphs; Language and Algorithms for Heterogeneous Graph Streams, FA8750-12-2-

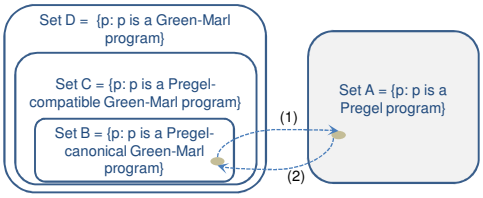

Figure 7: Relationships between different set of possible programs

0335; Army contract AHPCRC W911NF-07-2-0027-1; the National Science Foundation (IIS-0904497) and a KAUST research grant; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, AMD, Intel, NVIDIA, and Huawei. Authors also acknowledge additional support from Oracle.

8. REFERENCES

[1] http://github.com/tinkerpop/gremlin/wiki.

[2] Apache Giraph Project. http://giraph.apache.org.

[3] The laboratory for web algorithmics. http://law.dsi.unimi.it/datasets.php.

[4] D. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *IEEE ICPP 2006*.

[5] D. A. Bader and K. Madduri. SNAP: Small-world Network Analysis and Partitioning. http://snap-graph.sourceforge.net.

[6] U. Brandes. A Faster Algorithm for Betweenness Centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[7] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *VLDB*, pages 285–296, 2010.

[8] R. Chen, X. Weng, B. He, and M. Yang. Large Graph Processing in the Cloud. In *SIGMOD*, 2010.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] Apache Hadoop. http://hadoop.apache.org/.

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

[12] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Tech Report: Compiling Green-Marl into GPS . http://ppl.stanford.edu/papers/tr_gm_gps.pdf.

[13] S. Hong, J. Van Der Lugt, A. Welc, R. Raman, and H. Chafi. Early experiences in using a domain-specific language for large-scale graph analysis. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13. ACM.

[14] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *VLDB*, 5(8), 2012.

[15] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS*, 2009.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD '10*. ACM.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *SIGMOD*. ACM, 2008.

[18] S. Salihoglu and J. Widom. GPS: Graph Processing System. http://infolab.stanford.edu/gps.

[19] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A Warehousing Solution Over a Map-Reduce Framework. *VLDB*, 2(2), 2009.

[20] Trinity. http://research.microsoft.com/en-us/projects/trinity/default.aspx.

[21] L. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[22] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. Active Pebbles: A Programming Model for Highly Parallel Fine-grained Data-driven Computations. In *PPoPP*, pages 305–306. ACM, 2011.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. HotCloud'10, 2010.

APPENDIX

A. DISCUSSIONS ON COMPLETENESS

In this section, we discuss the completeness of our approach. Currently, we focused on graph algorithms that deal with only one graph and do not modify the graph, this is the typical case in graph analysis. To illustrate our points, Figure 7 shows the relationship between different sets of possible programs: Set A is the set of valid Pregel Programs, Set B is the set of valid Pregel-canonical Green-Marl programs, Set C is the set of Pregel-compatible Green-Marl programs, and Set D represents all valid Green-Marl programs (a Green-Marl program is Pregel-compatible if there exists an equivalent Pregel-canonical program for it).

First, is there an equivalence between the set of Pregel programs and the set of Pregel-canonical Green-Marl programs (i.e. Set A vs. Set B)? Since the forward direction (i.e. arrow (1)) has been shown in Section 3.1, the remaining thing to show is the converse direction (i.e. arrow (2)): can any Pregel program be described as a functionally equivalent Pregel-canonical Green-Marl program? Without formal proof, we claim yes because every Pregel API has a direct mapping to a Pregel-canonical pattern. Java-specific expressions (e.g. library call) can be embedded in Green-Marl programs with a *foreign syntax* [11] extension.

Second, is every Green-Marl program Pregel-compatible (i.e. Set D = Set C)? Theoretically, the answer is *yes*, but only because we can *simulate* the Green-Marl program sequentially – if there is a pattern for which no translation rule is known, the compiler can generate a Pregel program that does computation only on the master. During *simulation*, every vertex data access is replaced with an extra vertex-parallel step (**Random Access in Sequential Phase** in Section 4.1). Excluding *simulation*, the answer would be *no*, but we are still discovering what is the exact boundary of Set C.

Third, can the compiler transform every Pregel-compatible program into Pregel-canonical one? Currently, the compiler simply fails when the input program contains a pattern for which no transformation rule is known. Since we do not know if a program is fundamentally compatible with Pregel (without *simulation*), the answer to this question would be *not yet*. On the other hand, once a manual Pregel implementation of such a program is developed, new transformation rules based on the implementation could be added to the compiler.

Finally, our compiler currently does not cover algorithms in which the underlying graph is modified or which involve multiple graphs.

The former is not supported by Green-Marl, the latter is not well-supported by Pregel. However, since both the Green-Marl language and the Pregel framework are being improved with regard to these issues, one can expect upcomming support.

To summarize, we claim that a program implemented in Pregel-canonical Green-Marl is functionally equivalent to the same program implemented in Pregel, but is more succinct and easier to understand. By adding more transformation rules, the compiler will expands its coverage until it can translate every Pregel-compatible Green-Marl program.

B. GREEN-MARL PROGRAMS OF THE ALGORITHMS

Single-Source Shortest Path (SSSP)

The Green-Marl program below computes the shortest distance from a single source vertex (`root`) to every other vertex in the graph. Note that the original Pregel paper [16] used the exact same algorithm as an example.

```
357  Procedure sssp(G:Graph, root:Node,
358     len:Edge_Prop<Int>; dist:Node_Prop<Int> )
359  {
360    Node_Prop<Bool> updated;
361    Node_Prop<Bool> updated_nxt;
362    Node_Prop<Int>  dist_nxt;
363    Bool fin = False;
364    // Initialize
365    G.dist = (G == root)? 0 : +INF;
366    G.updated = (G == root)? True: False;
367    G.dist_nxt = G.dist;
368    G.updated_nxt = G.updated;
369    // Main loop
370    While(!fin) {
371      fin = True;
372      // Propagate changes of updated node
373      Foreach(n: G.Nodes)(n.updated) {
374        Foreach(s: n.Nbrs) {
375          Edge e = s.ToEdge(); // the edge to s
376          // updated_nxt becomes true
377          // only if dist_nxt is actually updated
378          <s.dist_nxt; s.updated_nxt> min=
379                <n.dist + e.len; True>;
380      } }
381      G.dist = G.dist_nxt;
382      G.updated = G.updated_nxt;
383      G.updated_nxt = False;
384      fin = ! Exist(n: G.Nodes){n.updated};
385  } }
```

The program above first initalizes the vertex-private variables (lines 365 – 368), before entering the main while-loop (line 370). Inside the while-loop, every vertex for which distance has been updated (line 373) tries to update its neighbors' distance via a min reduction (lines 374 – 379). Deterministic results are guaranteed via two-phase updates (line 381 – 384). The while-loop terminates when no vertex is updated (line 384).

Random Bipartite Matching

The following is a Green-Marl program that finds a match in a bipartite graph. Note that the original Pregel paper used the exactly same algorithm as an example [16].

```
386  Procedure bipartite_matching(
387   G: Graph, isBoy : Node_Prop<Bool>; // bipartite graph
388   Match: N_P<Node(G)>): Int { // match and count of match
389    Int count = 0;
390    Bool finished = False;
391    N_P<Node> Suitor;
392    G.Match = NIL;
393    G.Suitor = NIL;
394    While (!finished) {
395     finished = True;
396     // boys propose to every unmatched girl nearby
397     Foreach(b: G.Nodes)(b.isBoy && b.Match==NIL) {
398      Foreach(g: b.Nbrs)(g.Match == NIL) {
399        g.Suitor = b;    // intended write-write conflict.
400                         // Only one will be make effect.
401        finished &= False;
402     }}
403     // girls accept only one and reply
404     Foreach(g: G.Nodes)(!g.isBoy && g.Match==NIL) {
405       If (g.Suitor != NIL) {
406         Node b = g.Suitor; // the lucky chosen one
407         b.Suitor = g;      // Reply: "I'm available"
408         g.Suitor = NIL;    // clear suitor
409     } }
410     // boy accept only one reply
411     Foreach(b: G.Nodes)(b.isBoy && b.Match == NIL) {
412       If (b.Suitor != NIL) {
413         Node g = b.Suitor; // the lucky chosen one
414         b.Match = g;
415         g.Match = b;
416         count++; // increase match count
417   } } }
418   Return count; }
```

The program above takes a bipartite graph as an input, where edges exist only from *boy* vertices to *girl* vertices (line 387). `Match` and `Suitor` are initialized as `NIL` (line 392 – 393) before the main while-loop begins at line 394. The main loop implements a three-phase handshaking protocol. In the first phase, every unmatched *boy* vertex writes its ID to the `Suitor` field of all the unmatched neighboring *girl* vertices. (lines 397– 402) Note that in this phase multiple *boy* vertices write their ID in parallel to the same *girl* vertex, but only one of those writes becomes effective at the end. In the second phase (lines 404 – 409), every *girl* vertex replies to the effective `Suitor` vertex, by writing back her ID to the `Suitor` vertex. However, these writes also happen in parallel and thus only one of them becomes effective. In the third phase (lines 411 – 417), the *boy* vertices finalize the matching by checking their `Suitor` field. This three-phase protocol is repeated until no further matches are available.

PageRank

The following is a Green-Marl program that computes PageRank of a given graph. Note that the original Pregel paper used the exact same algorithm as an example.

```
419  Procedure PageRank(G: Graph, e,d: Double,
420    max_iter: Int; PR: Node_Prop<Double>(G)) {
421    Double diff =0;
422    Int cnt = 0;
423    Double N = G.NumNodes();
424    G.PR = 1 / N; // Init PageRank
425    Do { // Main iteration
426      diff = 0.0;
427      Foreach (t: G.Nodes) {
428        Double val = (1-d) / N + d* Sum(w: t.InNbrs){
429            w.PR / w.OutDegree()};
430        t.PR <= val @ t;
431        diff += | val - t.PR |; }
432      cnt++;
433  } While ((diff > e) && (cnt < max_iter)); }
```

line 424 initializes PageRank before the main iteration begins at line 425. During the main iteration, the PageRank of each node is computed by the PageRank of neighboring nodes (lines 427–429), while the new value is updated synchronously at the end of the `t`-loop (line 430). The iteration is repeated until convergence conditions are met or a given number of iterations has passed.

Conductance

The following Green-Marl program computes the conductance of a subset of a graph.

```
434  Procedure conductance(G: Graph,
435    member: N_P<Int>(G), num: Int) : Double {
436    Int Din, Dout, Cross;
437    // compute degree sum of inside/outside nodes
438    // and number of crossing edges
439    Din= Sum(u:G.Nodes)(u.member==num){u.Degree()};
440    Dout= Sum(u:G.Nodes)(u.member!=num){u.Degree()};
441    Cross=Sum(u:G.Nodes)(u.member==num){
442          Count(j:u.Nbrs)(j.member!=num)};
443    Double m = (Din < Dout) ? Din : Dout;
444    If (m ==0) Return (Cross == 0) ? 0.0 : +INF;
445    Else Return (Cross / m); }
```

The algorithm computes the degree sum among nodes that belong to the subset and those that do not (lines 439–440), and counts the number of edges that cross the boundary of the subset. (lines 441–442). The final conductance value is computed from these values (lines 443–444).