Tech Report: Compiling GreenMarl into GPS

Sungpack Hong
Oracle Labs and Stanford University
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

30 October 2012

Abstract

The massive size of the data in large graph processing requires distributed processing. However, conventional frameworks for distributed graph processing, such as Pregel, use programming models that are well-suited for scalability but inconvenient for programming graph algorithms. In this paper, we use Green-Marl, a Domain-Specific Language for graph analysis, to describe graph algorithms intuitively and extend its compiler to generate equivalent Pregel programs. Using the semantic information exposed by Green-Marl, the compiler applies the same kinds of transformation rules that programmers apply when manually implementing graph algorithms with Pregel. Our experiments show that the Pregel programs generated by Green-Marl compiler perform similarly to native Pregel implementations of the same algorithms. The compiler is even able to generate a Pregel implementation of a complicated graph algorithm whose native Pregel implementation is very challening.

1 Introduction

Large-scale graph processing often involves data sets that exceed the memory capacity of a single machine. Example of such data sets include the webgraph, various popular social networks and genomic data. This led to various distributed graph processing frameworks chiefly among them Google's *Pregel* [21] and its open source implementations [2, 24, 15, 23] Pregel is a scalable distributed graph processing framework in which the vertices of a graph are distributed across multiple machines in a cluster. These vertices communicate via sending each other messages in order to perform a graph computation.

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). Many graph algorithms (e.g. PageRank and Single-Source Shortest Path) can be implemented using this programming model in a straightforward manner as these algorithms are composed of a single vertex-centric computation kernel. Moreover, this vertex computation always requires the same data fields from neighboring vertices at every timestep; such a regular data access pattern can be easily mapped into Pregel's message passing API.

However, for more sophisticated graph algorithms, their implementation in Pregel becomes more challenging. This is especially the case when the algorithm is composed of multiple computation kernels connected by non-trivial control flow as is the case for Betweenness Centrality [8]. One can summarize the challenges as follows:

1. When implementing such multi-kernel algorithms using Pregel's programming model, the programmer needs to explicitly keep track of the global execution state inside the single computation function. The overhead of writing state management code grows as the control flow of the algorithm becomes more complicated.

2. In these algorithms, each kernel may access different fields, each having a different data type. Furthermore, a single kernel may access different fields depending on non-trivial control flow and since Pregel's message passing API only allows a single message type, the programmers must implement explicit message-type management code for varying data-access patterns.

3. Pregel's bulk-synchronous computation (BSP) model [27]
forces the programmer to ensure that all the remote data required in a given timestep be sent in the previous timestep via messages. In the presence of multiple kernels and non-trivial control flow, it is not easy to identify exactly which data will be required in the next timestep.

4. Algorithms may include certain operations which are not naturally expressed as vertex-centric: for example, global computation, random vertex access, reverse edge iteration, (reverse) breadth-first order traversal and so on. The programmer has to alter these operations into a vertex-centric programming model.

In addition, optimizing the performance of the Pregel implementation of these sophisticated algorithms is more challenging due to resulting increase in complexity. Consequently, Pregel programmers have to put considerable amount of effort in resolving all of the challenges just discussed (Section 2.2).

In this paper, we show how one can express graph algorithms intuitively using *Green-Marl* [16], 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 worrying about the challenges we have discussed, specifically:

1. Green-Marl adopts an imperative programming style which naturally describes non-trivial global control flow.
2. Green-Marl assumes random memory access (as in shared memory environments), and thus there is no need to reason about messages at all.
3. In Green-Marl, data access is not bulk-synchronous but immediate; the programmer need not speculate which data will be required in the next timestep.
4. Green-Marl programs need not be vertex-centric. Rather Green-Marl programs feel like natural descriptions of graph algorithms. This results from the inclusion of graph-specific language constructs such as global computation, random vertex access, reverse edge iteration, and (reverse) breadth-first order traversal.

The Green-Marl compiler can then produce equivalent Pregel implementations, dealing with the challenges we have discussed in place of the programmer. That is, the compiler automatically generates an implementation that manages execution state and message passing. The compiler ensures that all the data (potentially) required in the next timestep is sent in current timestep. The compiler transforms the graph operations that are not vertex-centric into the vertex-centric form. The compiler also tries to optimize the performance of the generated implementation. The high-level semantic knowledge captured by the Green-Marl language provide sufficient information for the compiler to apply these transformations.

Our compiler implementation is based on the observation that there are several patterns in graph algorithms that have natural implementations in 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 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 experiments show that the performance of the compiler generated programs are as good as manually coded Pregel implementations of the same algorithms. Our compiler is even able to to compile a Green-Marl implementation of approximate Betweenness Centrality, for which a manual coded 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 applied by our 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 applied by our Green-Marl 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 manually coded Pregel implementation of the same algorithms. We also show that our compiler is able to compile a Green-Marl implementation of approximate Betweenness Centrality [6, 7], a complicated graph algorithm, into Pregel.

2 Background

2.1 Pregel, GPS, and Green-Marl

The Pregel Framework and its Programming Model

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

Figure 1: Green-Marl Implementation of Calculating the Average Teenage Followers

We give an overview of Pregel here and refer the reader to [21] for details. Broadly, the input is a directed graph, and each vertex of the graph maintains a local user-defined state, and a *flag* indicating whether or not the vertex is active. Optionally, edges may also have states. The computation is inherently iterative, terminating when all vertices are inactive in a particular iteration. Within an iteration *i*, each active vertex *u* in parallel: (a) looks at the messages that were sent to *u* in iteration $i - 1$; (b) modifies its state; (c) sends messages to neighborhood vertices or vertices with known ids and optionally becomes inactive. A message sent in iteration *i* from vertex *u* to vertex *v* becomes available for *v* to use in iteration $i + 1$. Similar to the *map()* and *reduce()* functions in the MapReduce framework, the behavior of each vertex is encapsulated in a function *vertex.compute()*, which is executed exactly once in each iteration. Computation proceeds in a bulk-synchronous parallel way (BSP) [27], where global barrier synchronization is enforced at the end of each iteration – i.e. no vertex can proceed into iteration *i + 1* until all the other vertices finish executing *vertex.compute()* in iteration *i*. For global sharing of data, vertices use *aggregators*. Each vertex can create and update the value of an aggregator. Multiple updates to the same aggregator in iteration *i* are merged by a user-defined reduction function and the merged value becomes available to all vertices in the beginning of iteration $i + 1$.

As a result of BSP model, execution of `compute()` does not wait for message delivery; thus computation is overlapped with communication. In addition, since the message delivery can be delayed until the end of timestep, the framework can buffer messages to enable the use of large-sized network packets for their delivery; thereby utilizing underlying network bandwidth more efficiently.

GPS: An open-source implementation of Pregel

In this paper, we used GPS [24], an open-source implementation of Pregel. GPS has an extension to Pregel's API. In addition to *vertex.compute()* within which vertex-centric, parallel computations can be expressed, GPS has a second function *master.compute()* to express sequential parts of graph algorithms. Programmers can optionally define a *Master* object, which has its own local state and *compute()* function. *master.compute()* is invoked at the beginning of each iteration before *vertex.compute()* functions are called. This ensures serializability between *master.compute()* and *vertex.compute()* function calls. GPS maintains a map of *global objects*, which are implementations of aggregators of Pregel, and can be accessed both by the master and the vertices. Master and vertices can modify the global objects map by creating new or updating existing key-value pairs. If the master puts a key-value pair into the map during *master.compute()* before the beginning of iteration *i*, this pair will be available for vertices to use during iteration *i*. Similar to Pregel, a user-defined reduction function merges multiple updates to value of the same key.

Green-Marl: a DSL for graph algorithms

Green-Marl [16] is a DSL that is designed for implementing graph algorithms in an intuitive way. We omit providing details of the language but refer the interested reader to publicly available language specification [3]. In a nutshell, Green-Marl is an imperative language with built-in data types, operators and functions tailored for the implementation of graph algorithms. The domain-specific constructs allows for the high-level graph-specific semantics to be captured and exposed to the Green-Marl compiler. Using this semantic information, the original version of the Green-Marl compiler generates a parallelized C++ implementation of a given graph algorithm [16].

In this paper, we extend the open-source Green-Marl compiler to generate an equivalent Pregel (GPS) implementation as well. The remainder of this paper explains the mechanisms of such a program transformation. If our compiler encounters a Green-Marl program that cannot be translated into an equivalent Pregel implementation, it will simply reports an error (Section A).

2.2 A Programming Model Comparison

Pregel's programming model differs from the traditional imperative style, shared-memory programming model in which many graph algorithms are typically expressed. 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 1 and Figure 2 show the Green-Marl and Pregel implementations of this algorithm respectively.

The Green-Marl implementation in Figure 1 feels like a more intuitive translation of the proposed graph algorithm. The program

```
13 public class TeenCnt extends ... {
14  public void compute(int iterationNo, ...) {
15   if (iterationNo == 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 (iterationNo == 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 (iterationNo == 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 2: Pregel Implementation of Calculating the Average Teenage Followers

```
37 Procedure bc_approx(G:Graph, BC:Node_Prop<Float>,
38                     K:Int) {
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 3: Green-Marl Implementation of Approximating Betweenness Centrality

is written in an imperative style; it describes a sequence of statements to be executed in order but without any notion of timesteps. The program statements are 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).

In contrast, the Pregel implementation of the same algorithm in Figure 2 looks quite different. First, the Pregel implementation has to consider the notion of timestep [1]. 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 1 to its outgoing neighbors (Line 15 – 26). This change results from Pregel only allowing 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). Fourth, the Pregel implementation requires more boilerplate code than shown in the figure, such as the definition of master and message classes.

Although implementing algorithms traditionally described in an imperative, shared-memory style into Pregel's Programming model is manageable for a simple algorithm as in Figure 1, it becomes more challenging as the algorithm gets more complicated. Consider for instance the *Betweenness Centrality* (BC) algorithm [8], which measures the relative importance of nodes in a graph;

[1]In this paper, we use a slightly modified API to simplify our explanation. For instance we assume that vertex data is stored directly in the vertex class, rather than in its companion class.

Figure 3 is the Green-Marl implementation of the approximate Betweenness Centrality algorithm as described in the SNAP graph library [7]. 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.[2]

It is not obvious how the algorithm shown in Figure 3 can be implemented using Pregel's programming model. Note that, due to the complexity of this algorithm, even its multi-threaded implementations were considered worthy of publication [5, 20] and no manually coded Pregel implementaion has been reported. There are a few challenges in implementing this algorithm with Pregel. 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 the Pregel API would require significant amount of code complexity to explicitly manage the execution state and message types.

That being said, the Green-Marl compiler can successfully translate the program in Figure 3 into an equivalent Pregel implementation (Section 5) which contains several vertex-centric computation kernels and different types of messages, with a significant amout of state management code. 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 Automatic Translation of Graph Algorithm Patterns

In this section, we explain how certain patterns that emerge in graph algorithms can be directly translated into using the Pregel programming model. We also show how these translations are applied by the Green-Marl to Pregel 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 snippet goes through a sequential, vertex-parallel, and finally another sequential computation phase.

```
  // sequential computation
  Int S = 0;
  Int C = 0;
  // vertex-parallel computation
  Foreach(n: G.Nodes) {
    if (n.age > K) {  // K,S,C: global variables
        S += n.cnt;
        C += 1;
  } }
  // sequential computation
  Float val = (C == 0) ? 0 : S / (float) C;
```

Using the extended Pregel API provided by GPS (Section 2.1), sequential and vertex-parallel computation can be naturally mapped into *master compute* and *vertex compute* methods respectively. However, the example above contains two different sequential computation phases. Since the same `compute()` method is invoked at each timestep, 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. The compiler then automatically creates a corresponding state machine. For instance, the compiler generates the following code from the example above:

```
class master extends ... {
  int _state, _next_state;
  public void compute(..) {
    ...
    switch(_state){ // state machine managed by master
      case 0: do_state_0(..);break; // S=0,C=0;
      case 1: do_state_1(..);break; // vertex parallel
      case 2: do_state_2(..);break; // val=(C==0)?...
    }
    ...
    // broadcast current state to the vertex objects
    Global.put("_state", new IntValue(_state));
  }
  ...
```

[2]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 social graphs and identify highly central nodes.

```
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. Note that the compiler-generated state machine is managed entirely by the master. The master broadcasts the current state number to the vertices via `Global` (Line 81) so that every vertex gets the same state number (Line 91).

Vertex and Global Object Construction

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

The compiler identifies global and vertex-local variables in the input program and implements each as field members in the master and vertex class of the generated Pregel program, respectively. Moreover the compiler also ensures visibility of global variables using the `Global` map (Section 2.1). For instance, the compiler generates the following code from the previous example (Lines 59 – 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 to 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` value into all of its neighbors' `foo` variable.

```
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 sends its `bar` value to its neighbors as messages while neighbors receive those messages and accumulate those values into `foo`. Due to BSP computing model, this message-passing takes two timesteps.

The compiler recognizes these nested-loop patterns and generates corresponding message passing Pregel patterns. The example code (Lines 123– 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 payloads of the messages via data 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 a 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.

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. (See **Edge-Flipping** in Section 4.1 for further discussion on how the compiler can automatically transform message pulling programs into message pushing ones.)

Multiple Communication

It is possible that an outer-loop contains multiple inner-loops where each inner-loop performs a different computation. In such cases, the compiler attaches a tag to the message which indicates to the receivers which computation to perform. Thus the following example

```
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   } }
```

becomes

```
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 random (i.e. possibly non-neighboring) vertices. In Green-Marl programs, random modification happens when a vertex is referenced not by a loop-iterator but by a random variable as in the following example:

```
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       // pick any random neighborhood of node n
170       Node s = n.PickRandomNbr();
171       s.foo += 1; // random write
172     }
173   }
```

The compiler identifies such random writes (e.g. Line 167) and creates the corresponding message passing code. Message payloads are determined in a similar way to neighborhood communication. Since the random messaging API is only available for the vertex class in Pregel, random-writes are allowed only in the vertex-parallel phase of a Green-Marl program. The compiler generates the following code from the example above:

```
class vertex extends ... { ...
  private void do_state_1(..) {
    if(getNumNbrs()==0||Random.nextDouble()<0.2){
      node s = Ramdom.netxInt(getGraphSize());
      Message m = new Message();
      m.type = 1;
      m.intVal = this.bar;
      sentToNode(s, m);
    } else {
      node s = getNbrs()[Random.nextInt(getNumNbrs())];
      Message m = new Message();
      m.type = 2;
      sentToNode(s, m);
  } }
  private void do_state_2(..) {
    for(Message m : rcvdMsgs()) {
      if (m.type == 1)
          this.foo += m.intVal;
      else if (m.type == 2)
          this.foo += 1;
} } }
```

Edge Properties

The Green-Marl language specification allows edges to have properties just like vertices. However, the compiler allows only certain patterns in edge property access since Pregel makes edge properties only accessible from the source vertex. Specifically, the compiler allows edge properties to be accessed only by an edge variable which points to a neighborhood iterator; for example, `e.len` in the following code snippet:

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

The compiler generates the following code from the example above:

```
class vertex extends ... { ...
  private void do_state_1(..) {
    for (Edge e: getEdges()) {
      Message m = new Message();
      m.intVal = this.dist + e.len;
      sentToNode(e.getDest(), m)
  } }
  private void do_state_2(..) {
    for(Message m : rcvdMsgs()) {
      int i0 = m.intVal;
      if (this.dist < i0) this.dist = i0;
} } }
```

3.2 Pregel-canonical Programs

A Green-Marl program is Pregel-canonical if it only consists of patterns that have a direct mapping to the Pregel API. Specific conditions for Pregel-canonical program 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** Inner `Foreach` loops do not modify *outer-loop scoped variables* (Section 3.1).
- **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** Edge properties are only accessed via edge variables that point to neighborhood iterators.

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 translate programs that are not Pregel-canonical.

4 Advanced Compiler Transformations

4.1 Converting 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.

Desugaring

The Green-Marl language provides syntactic sugar for user convenience that does not meet the conditions for Pregel-canonical form. However, statements can be desugared into Pregel-canonical statements by the compiler. For instance, the syntactic sugar in the following example

```
212  G.A = G.B + 1;  // Group assignment;
213  x = y * Sum{n:G.Nodes}{n.C}; // Reduction expression
```

is desugared into explicit loops

```
214  Foreach(n:G.Nodes) // Group assignment
215    n.A = n.B + 1;
216  Int _S = 0; // Reduction expression
217  Foreach(t:G.Nodes)
218    _S += t.C;
219  Int x = y * _S;
```

which makes the program Pregel-canonical.

Flipping Edges: Converting Message Pulling into Message Pushing

The previous section explained that only *Message Pushing* is supported by Pregel-canonical Green-Marl programs. 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:

```
220  Foreach(n:G.Nodes)
221   Foreach(t:G.InNbrs)   // maximum of in-nbrs
222     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 `t`. However, the compiler transforms the program into a functionally equivalent Pregel-canonical program:

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

Note that, in the above example, the complier has flipped the direction of information flow as each `t` now writes its `bar` value to its *outgoing* neighbors `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[3] 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).[4]

Dissecting Nested Loops: Preprocessing for Edge Flipping

Since condition (1) of the **Flipping Edges** transformation is rather restrictive, the compiler applies the following pre-processing transformations.

Consider the following example:

```
227  Foreach(n: G.Nodes){
228   Int _C = 0;
229   Foreach(t: n.InNbrs)(t.age>=13&&t.age<=19)
230     _C += 1;
231   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 230). Whenever the compiler recognizes this pattern, it introduces a temporary vertex property to replace the scalar variable. Therefore the above code becomes:

[3] Optionally, there can be a filter, i.e. an `if` statement between the inner-loop and the outer-loop.

[4] In the presence of a filtering `if` statement in between the two original loops, the `if` statement is pushed into the inner loop after this transformation.

```
232 Node_Prop<Int> _tmp;
233 Foreach(n: G.Nodes){
234  n._tmp = 0;
235  Foreach(t: n.InNbrs)(t.age>=13&&t.age<=19)
236    n._tmp += 1;
237  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 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.

```
238 Node_Prop<Int> _tmp;
239 Foreach(n: G.Nodes)
240  n._tmp = 0;
241 Foreach(n: G.Nodes)
242  Foreach(t: n.InNbrs)(t.age>=13&&t.age<=19)
243    n._tmp += 1;
244 Foreach(n: G.Nodes)
245   n.cnt = n._tmp;
```

Note that the compiler can now apply the edge-flipping transformation to the second loop (Lines 241 – 243), which makes the entire program Pregel-canonical.

Arbitrary Access in Sequential Phase

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

```
246 s.dist = 0;  // a random write in seq phase
```

becomes,

```
247 Foreach(n:G.Nodes) {
248   if (n==s) n.dist = 0;
249 }
```

BFS-order Graph Traversal

Green-Marl provides a language construct for a BFS-order traversal of the graph with the user supplying their own custom computation to occur during the traversal (e.g. *Approximate Betweenness Centrality* in Section B). The original Green-Marl compiler [16] uses a pre-built template in order to generate multi-threaded BFS traversals. The Green-Marl to Pregel compiler transforms a BFS traversal statement into a set of Pregel-canonical statements; specifically, the BFS traversal is transformed into iterative frontier expansions. The following, original Green-Marl code

```
250 InBFS(n: G.Nodes From s) { // BFS-order traversal
251   ... // user code
252 }
```

is transformed into the following, Pregel-canonical Green-Marl code

```
253 {
254  Node_Prop<Int> _lev; // hop-distance from root
255  Bool _fin=False;
256  Int _curr = -1;
257
258  G._lev=+INF; // initialize distance as INF
259  s._lev=0;    // root node has zero-distance
260  While(!_fin){ // level-wise frontier expansion
261   _fin=True; _curr++;
262   Foreach(n:G.Nodes)(n._lev==_curr){
263     Foreach(t:n.Nbrs)(t._lev==+INF){
264       // expand unreached nbrs
265       t._lev=_curr+1; _fin=False;
266     }
267     ... // user code
268 } } }
```

Note that the user-provided custom computation in the original program (Line 251) is fused with the expanded BFS code (Line 267). 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,

```
269 InBFS(n: G.Nodes From s) { // BFS-order traversal
270   Foreach(t:n.DownNbrs) // iterate over BFS children
271    ... // more user code
272 }
```

becomes

```
273  ...
274   Foreach(n:G.Nodes)(n._lev==curr){
275     Foreach(t:n.Nbrs)(t._lev==+INF){
276       // expand unreached nbrs
277       t._lev=_curr+1; _fin=False;
278   } }
279   Foreach(n:G.Nodes)(n._lev==curr){ // a separate loop
280     Foreach(t:n.Nbrs)(t._lev==_curr+1){ // BFS children
281       ... // more user-code
282   } }
```

Notice how _lev is used to implement iteration over the BFS children.

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. For instance,

```
283  InBFS(n: G.Nodes From s) {...} // Forward BFS
284  InReverse {
285   ... // user code
286  }
```

becomes

```
287  {
288    ... // Forward BFS
289    While (_curr >= 0) { // Reverse BFS
290      Foreach(n:G.Nodes)(n._lev==_curr){
291       ... // user code
292      }
293      _curr--;
294  } }
```

4.2 Performance Optimizations

We have identified two performance optimization techniques which are applied in manually coded Pregel programs and implemented them in our compiler.

State Merging

Whenever it is safe to do so, the compiler merges two consecutive states of vertex computation into one. Consider the following example. According to the translation rules in Section 3.1, two states are inferred for the first nested loop (Line 295) and another state is inferred for the other loop (Line 299).

```
295  Foreach(n: G.Nodes)(n.age>=10&&n.age<20){
296    Foreach(t: n.Nbrs)  // nbr iteration -> state 1,2
297      t.cnt += 1;
298  }
299  Foreach(n: G.Nodes)(n.age>K){ // loop --> state 3
300    _sum += n.cnt;
301  }
```

However, the compiler merges the second state from the nested loop (the message receiving state) with the state from the other loop (the computing sum state) together, as follows:

```
302  private void do_state_1(..) { // sending messages
303    if (this.age>=10 && this.age<20){
304      Message m = new Message();
305      sendToNbrs(m);
306  } }
307  private void do_state_4(..) {  // state 2, 3 merged
308    // (merged) state2: receiving messges
309    for(Message m : rcvdMsgs()) {
310       this.cnt += 1;
311    }
312    // (merged) state3: updating global sum
313    if (this.age > K) {
314      Global.put("_sum", new intSum(this.cnt));
315  } }
```

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.

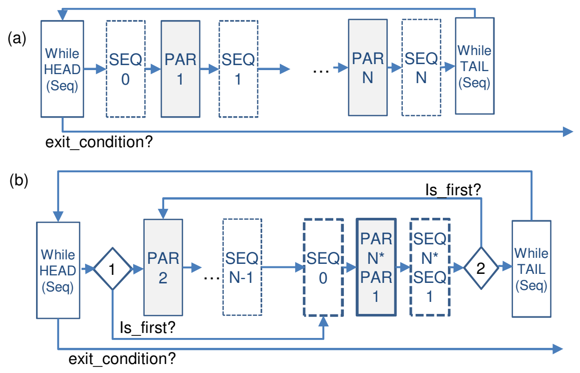

Figure 4: Intra-loop State Merging: (a) is the state machine before merging, (b) is the one after merging. The merged states PAR N* and SEQ N* are not executed when `is_first` is true. `is_first` is set during the first entry into the while-loop and cleared after state `2` (in the figure) is first executed.

Intra-Loop State Merging

As a related optimization to state merging, consider the following example:

```
316  While(condition) {
317   Foreach(n: G.Nodes) {
318     Foreach(t: n.Nbrs) {
319       t.foo += n.bar;
320   } }
321   ...  // update condition
322  }
```

Again, two states are inferred for the nested loop (Line 317) – one state for message sending, another state for message receiving. 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:

```
323  private void do_state_1(..) {
324    if (!_is_first) { // (merged): receiving messages
325      for(Message m : rcvdMsgs())
326       this.foo += m.intVal;
327    }
328    { // (merged): sending messges
329      Message m = new Message();
330      m.intVal = this.bar;
331  } }
```

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 4, where there are N vertex-parallel states and N+1 sequential states inside a while-loop, while 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 states are executed in the correct order.

However, the merged state machine (Fig 4.(b)) differs from the original (Fig 4.(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 executed one more time than in the original program. The compiler checks whether these differences do not alter the semantics of the original program via data 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 an 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:

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

Table 1: Input graphs: Twitter and Sk-2005 graphs were provided by *The Laboratory for Web Algorithmics* [4].

```
private void do_state_0(..) { // send ID to nbrs
  Message m = new Message();
  m.intValue = this.getID();
  sendToNbrs(m);
}
private void do_state_1(..) { int i=0;
  // create list of incoming neigbors
  this.in_nbrs = new int[rcvdMsgs().length];
  for(Message m : rcvdMsgs()) {
    this.in_nbrs[i++] = m.intValue;
} }
```

The compiler uses this list to generate code that sends messages from a vertex to its incoming neighbors as shown in the following example:

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

Message Class and 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 de-serialization 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`:

```
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 (`type`) to distinguish different message payloads used in the program, 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 input/output (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.

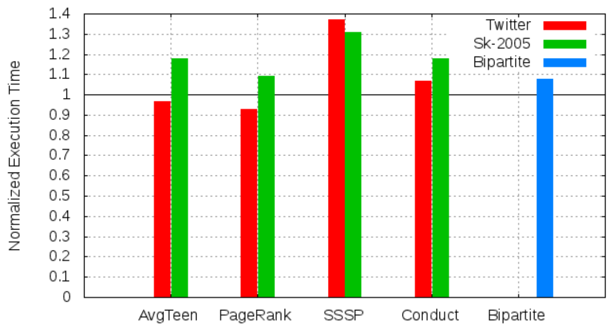

Figure 5: Run-time Comparison of Compiler-Generated and Manually Coded Pregel Programs.

Algorithm Name	Origin	Green-Marl LOC	Manual Pregel Impl. LOC	Description
Average Teenage Follower (*AvgTeen*)	-	13	130	A simple example described in Section 2.2 (Fig 1)
PageRank	[16, 21]	19	110	A famous algorithm to rank vertices
Conductance (*Conduct*)	[16]	12	149	An algorithm to compute the community structure
Single Source Shortest Paths With Edge Values (*SSSP*)	[21]	29	105	Finding shortest paths from a source *s* to all vertices
Random Bipartite Matching (*Bipartite*)	[21]	47	225	A matching algorithm for a bipartite graph
Approximate Betweenness Centrality	[16]	25	N/A	An algorithm to compute relative importance of vertic

Table 2: Graph algorithms.

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 [21] and Green-Marl [16] 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 we present are the averages across all runs ignoring the initial data loading stage. Performance across multiple runs varied by only small amounts.

5.1 Productivity Benefits

Table 2 shows the lines of code (LOC) required to implement 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.

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. Since such a claim is inherently subjective, we provide the Green-Marl implementations of the random bipartite matching, conductance, SSSP, and PageRank algorithms in the Appendix. We also provide the manual Pregel implementations of the random bipartite matching and conductance algorithms in our tech report [17] for comparison.

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 3) into Pregel-canonical by applying the following transformation rules: **BFS-order Graph Traversal**, **Desugaring**, **Nested-Loop Dissection**, **Flipping Edges**, and **Incoming Neighbors**. 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 is over 800 lines of code, consisting of nine vertex-centric kernels (after **State Merging**) 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 whenever possible. 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 21 lines of Green-Marl code in Figure 3.

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 5. 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. For example, a height of 1.1 indicates that the compiler-generated Pregel implementation of an algorithm was 10% slower than its manual implementation for a particular algorithm and input graph. As shown, the run-time performance 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. However the absolute run-time difference was only 24 seconds, 122 seconds for the compiler-generated program and 98 seconds for the manual implementation.

The run-time overhead of the compiler-generated programs is due to 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 vetices. 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 human programmers typically do when implementing algorithms manually using the Pregel API. Consequently, the compiler-generated implementaions run in virtually the same way as the manual implementations do and exhibit similar performance behavior.

6 Related Work

This paper shows how we can translate Green-Marl programs into the Pregel API [21]. Pregel is not the only large-scale distributed graph analysis framework. Other frameworks include ActivePebbles [28] and GraphLab [19] which also have a vertex-centric computation model. The Parallel Boost Graph Library (PBGL) [13] is an extension of the C++ Boost Graph Library that runs on clusters. Trinity [26] is a proprietary graph computation system at Microsoft which is built on top of a distributed RAM-based key-value store. HipG [18] is another distributed graph framework, where vertices are Java objects which use an RPC mechanism for distributed computation. One can also use non graph-specific distributed frameworks, with support for iterative computations, to do large-scale graph processing. For example, SPARK [29] is a general in-memory cluster computing framework that can also handle iterative computation. HaLoop [9], iMapReduce [30], Twister [12], and Surfer [10] are systems built on top of Hadoop [14] which can do iterative MapReduce [11] 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 [16], an existing DSL for graph processing. DSLs have also been used to provide high-level abstractions for different application domains. For instance, Pig Latin [22] and Hive [25] 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 DSL compiler 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 for which manual Pregel implementation is very challenging.

Our current solution also suggests a few opportunities for future work. First, with proper static analysis, we can improve the performance of our compiler-generated implementation for simple single-kernel algorithms. The compiler should be able to leverage more of Pregel's convenience API e.g *votetoHalt()* for execution context management. Second, by identifying more transformation rules that turn the graph algorithm into Pregel-canonical form, the compiler should be able to compile more Green-Marl programs into Pregel. Finally, our compiler can be extended to simplify the programming of other graph processing systems with different programming models or API.

References

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

[2] Apache giraph project. http://giraph.apache.org.

[3] Green-marl dsl. http://github.com/stanford-ppl/Green-Marl.

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

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

[6] D. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IEEE IPDPS*, 2008.

[7] D. A. Bader and K. Madduri. Snap: small-world network analysis and partitioning. http://snap-graph.sourceforge.net.

[8] U. Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the International Conference on Very Large Databases*, pages 285–296, 2010.

[10] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *Proceedings of the International Conference on Management of Data*, pages 1123–1126, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818. ACM, 2010.

[13] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

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

[15] Apache Hama. http://incubator.apache.org/hama/.

[16] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *ASPLOS*. ACM, 2012.

[17] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun. Compiling green-marl into gps. ppl.stanford.edu/papers/tr_gm_gps.pdf.

[18] E. Krepska, T. Kielmann, W. Fokkin, and H. Bal. Hipg: parallel processing of large-scale graphs. *ACM SIGOPS Operating Systems Review*, 45(2):3–13, 2011.

[19] 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. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[20] 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 *IEEE IPDPS 2009*.

[21] 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.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

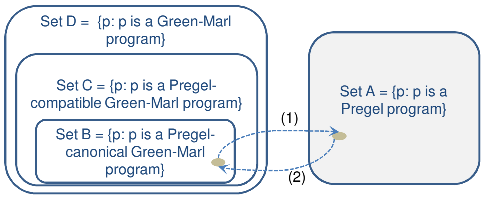
Set D = {p: p is a Green-Marl program}
Set C = {p: p is a Pregel-compatible Green-Marl program}
Set B = {p: p is a Pregel-canonical Green-Marl program}
Set A = {p: p is a Pregel program}
(1)
(2)

Figure 6: Relationships between different set of possible programs

[23] Phoebus. https://github.com/xslogic/phoebus.

[24] S. Salihoglu. Gps: Graph processing system. http://infolab.stanford.edu/gps.

[25] 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. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

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

[27] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[28] J. Willcock, T. Hoefler, N. Edmonds, and A. Lumsdaine. Active pebbles: A programming model for highly parallel fine-grained data-driven computations. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 305–306. ACM, 2011.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10. USENIX Association, 2010.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapreduce: A distributed computing framework for iterative computation. *DataCloud*, 2011.

A Discussions on Completeness

In this section, we discuss the completeness issue of our approach. For the time being, we focus on the case of graph algorithms that deal with only one graph and do not modify the graph, which is the typical case in graph analysis. As an illustration, Figure 6 shows the relationships between different sets of possible programs: Set A is the set of Pregel Programs, Set B is Pregel-canonical Green-Marl programs, Set C is Pregel-compatible Green-Marl programs, and Set D is any Green-Marl programs. (We say a Green-Marl program is Pregel-compatible if there exists an equivalent Pregel-canonical program for it.)

First, is there a Pregel program for each Pregel-canonical Green-Marl program and vice versa (i.e. Set A vs. Set B)? Since the forward direction (i.e. arrow (1)) has been already shown in Section 3.1, the remaining question is the converse direction (i.e. arrow (2)): can any Pregel program be described as an functionally equivalent Pregel-canonical Green-Marl program? Without formal proof, we claim yes because every Pregel API has a direct mapping to Pregel-canonical condition. Java-specific computation (e.g. library call) can be embedded in Green-Marl program with a *foreign syntax* [16] 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 program pattern whose translation rule is not known, the compiler can sequentialize it and execute it on the master. During *simulation*, every vertex data access inside 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 whose translation is not known. Since we do not know if a program is fundamentally compatible with Pregel (without *simulation*), the answer to the third question is *not yet*. On the other hand, once a manual Pregel implementation of such a program is developed, a new transformation rules based on it could be developed for the compiler. Green-Marl should be able to provide the required semantic information.

Finally, our compiler currently does not cover the algorithms where case if the underlying graph is modified or multiple graphs are involved. 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 regarding these matters, they will be covered in the future.

To summarize, we claim that a Pregel-canonical Green-Marl program is functionally as powerful as a Pregel program but more succinct and are easier to comprehend. By adding more transformation rules, the compiler expands its capabilities until it translates every Pregel-compatible Green-Marl program.

B Green-Marl Programs of the Algorithms

Single-Source Shortest Path (SSSP)

The Green-Marl program below computes shortest distance from a single source vertex (`root`) to every other vertex in the graph. Note that the original Pregel paper [21] used the exactly same algorithm as an example.

```
367 Procedure sssp(G:Graph, root:Node,
368    dist:Node_Prop<Int>, len:Edge_Prop<Int>)
369 {
370   Node_Prop<Bool> updated;
371   Node_Prop<Bool> updated_nxt;
372   Node_Prop<Int>  dist_nxt;
373   Bool fin = False;
374   // Initialize
375   G.dist = (G == root)? 0 : +INF;
376   G.updated = (G == root)? True: False;
377   G.dist_nxt = G.dist;
378   G.updated_nxt = G.updated;
379   // Main loop
380   While(!fin) {
381     fin = True;
382     // Propagate changes of updated node
383     Foreach(n: G.Nodes)(n.updated) {
384       Foreach(s: n.Nbrs) {
385         Edge e = s.ToEdge(); // the edge to s
386         // updated_nxt becomes true
387         // only if dist_nxt is actually updated
388         <s.dist_nxt; s.updated_nxt> min=
389                <n.dist + e.len; True>;
390     } }
391     G.dist = G.dist_nxt;
392     G.updated = G.updated_nxt;
393     G.updated_nxt = False;
394     fin = ! Exist(n: G.Nodes){n.updated};
395 } }
```

The program above first initalizes vertex-private variables (Lines 375 – 378), before entering the main while-loop (Line 380). Inside the while-loop, every vertex whose distance has been updated (Line 383) tries to update its neighbors distance via minimum reduction (Lines 384 – 389). Deterministic results are gauranteed via two-phase updates (Line 391 – 394). The while-loop terminates when no vertex is updated (Line 394).

Random Bipartite Matching

The following is a Green-Marl program thats finds a matching in a bipartite graph. Note that the original Pregel paper used the exactly same algorithm as an example [21].

```
396 Procedure bipartite_matching(
397  G: Graph, isBoy : Node_Prop<Bool>, // bipartite graph
398  Match: N_P<Node(G)>): Int { // match and count of match
399   Int count = 0;
400   Bool finished = False;
401   N_P<Node> Suitor;
402   G.Match = NIL;
403   G.Suitor = NIL;
404   While (!finished) {
405    finished = True;
406    // boys propose to every unmatched girl nearby
407    Foreach(b: G.Nodes)(b.isBoy && b.Match==NIL) {
408     Foreach(g: b.Nbrs)(g.Match == NIL) {
409       g.Suitor = b;    // intended write-write conflict.
410                        // Only one will be make effect.
411       finished &= False;
412    }}
413    // girls accept only one and reply
414    Foreach(g: G.Nodes)(!g.isBoy && g.Match==NIL) {
415      If (g.Suitor != NIL) {
416        Node b = g.Suitor; // the lucky chosen one
417        b.Suitor = g;      // Reply: "I'm available"
418        g.Suitor = NIL;    // clear suitor
419    } }
420    // boy accept only one reply
421    Foreach(b: G.Nodes)(b.isBoy && b.Match == NIL) {
422      If (b.Suitor != NIL) {
423        Node g = b.Suitor; // the lucky chosen one
424         b.Match = g;
425         g.Match = b;
426         count++; // increase match count
427   } } }
428   Return count; }
```

The program above takes a bipartite graph as an input, where edges exist only from *boy* vertices to *girl* vertices (Line 397). `Match` and `Suitor` are initialized as as `NIL` (Line 402 – 403) before the main while-loop begins at Line 404. 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 neighborhood *girl* vertices. (Lines 407– 412) 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 414 – 419), every *girl* vertex replies to the effective `Suitor` vertex, by writing back her ID to the `Suitor` vertex. However, the replying writes also happen in parallel and thus only one of them becomes effective. In the third phase (Lines 421 – 427), 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 computes PageRank of the given graph. Note that the original Pregel paper used the exactly same algorithm as an example.

```
429 Procedure PageRank(G: Graph, e,d: Double,
430   max_iter: Int, PR: Node_Prop<Double>(G)) {
431   Double diff =0;
432   Int cnt = 0;
433   Double N = G.NumNodes();
434   G.PR = 1 / N; // Init PageRank
435   Do { // Main iteration
436     diff = 0.0;
437     Foreach (t: G.Nodes) {
438       Double val = (1-d) / N + d* Sum(w: t.InNbrs){
439           w.PR / w.OutDegree()};
440       t.PR <= val @ t;
441       diff += | val - t.PR |; }
442     cnt++;
443 } While ((diff > e) && (cnt < max_iter)); }
```

Line 434 initializes PageRank before the main iteration begins at Line 435. During the main iteration, the PageRank of each node is computed by the PageRank of neighboring nodes (Lines 437–439), while the new value is updated synchronously at the end of the `t`-loop (Line 440). The iteration is repeated until convergence condition is met or given number of iterations has passed.

Conductance

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

```
444 Procedure conductance(G: Graph,
445   member: N_P<Int>(G), num: Int) : Double {
446   Int Din, Dout, Cross;
447   // compute degree sum of inside/outside nodes
448   // and number of crossing edges
449   Din= Sum(u:G.Nodes)(u.member==num){u.Degree()};
450   Dout= Sum(u:G.Nodes)(u.member!=num){u.Degree()};
451   Cross=Sum(u:G.Nodes)(u.member==num){
452           Count(j:u.Nbrs)(j.member!=num)};
453   Double m = (Din < Dout) ? Din : Dout;
454   If (m ==0) Return (Cross == 0) ? 0.0 : +INF;
455   Else Return (Cross / m); }
```

The algorithm computes the degree sum among nodes that belong to the subset and that do not (Lines 449–450), and counts the number of edges that cross the boundary of the subset. (Lines 451–452). The final conductance value is computed from these values (Lines 453–454).