Early Experiences in Using a Domain-Specific Language for Large-Scale Graph Analysis

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Jan Van Der Lugt
Oracle Labs
jan.van.der.lugt@oracle.com

Adam Welc
Oracle Labs
adam.welc@oracle.com

Raghavan Raman
Oracle Labs
raghavan.raman@oracle.com

Hassan Chafi
Oracle Labs
hassan.chafi@oracle.com

ABSTRACT

Large-scale graph analysis has recently been drawing lots of attention from both industry and academia. Although there are already several frameworks designed for scalable graph analysis, e.g. Giraph [1], all these frameworks adopt non-traditional programming models and APIs. This can significantly lower the productivity of the framework user. This paper discusses the feasibility of using an intuitive Domain-Specific Language (DSL) for graph analysis. Specifically, we use a compiler to translate Green-Marl [5] programs into an equivalent Giraph application, automatically bridging between very different programming models. We observe that the DSL programs are concise and intuitive, and that the compiler generated Giraph implementations exhibit performance on par with that of hand-written ones. However, the DSL compilation cannot but fail if the algorithm is fundamentally not compatible with the target framework. Overall, we believe that the DSL-based approach will provide great productivity benefits once it matures.

1. INTRODUCTION

A graph is a fundamental data representation that captures relationships (edges) between data entities (vertices). *Graph analysis* is a procedure which examines such relationships and extracts certain information that is not immediately available from a given data-set. Examples of graph analysis include assigning weights to the data entities based on their relative importance, predicting future relationship between data entities, and identifying groups of entities that are more closely related than others. Furthermore, graph analysis algorithms can take as input the results of other analyses, e.g. counting number of (un-)closed triads or sampling the vertices in a largegraph.

Large-scale graph analysis is often conducted in an off-line (batch) manner using throughput-oriented systems. This is due to the fact that such an analysis typically involves (repeated) inspection of nearly the entire graph instance, and thus naturally takes a lot of time. Consequently, the usage model of graph analysis is somewhat different from that of simple *graph querying* which mainly aims to identify the specified subset of the graph, using the most up-to-date information, within a certain latency. This is analogous to the difference between OLAP systems and OLTP systems in classic database design.

Several different off-line systems have been used for large-scale graph analysis. Hadoop [4], the most popular MapReduce engine, has been widely used to solve graph analysis problems [14] even though its performance for graph analysis has been put in question [9]. There are also systems that are specially designed for graph analysis. For example, Pregel [10], and its open-source implementation Giraph [1], provide a graph-specific message-passing API on top of a bulk-synchronous [17] computation engine. GraphLab [9] performs distributed asynchronous vertex-wise computation triggered by messages exchanged between vertices. Trinity [16] performs graph computation using a distributed in-memory key-value store.

Noticeably, all the graph analysis systems above adopt different computation models, not to mention different APIs. Consequently, it is up to the user to *implement* a graph algorithm that is compatible with each system's computation model and API. Such an *implementation*, however, can impose a non-trivial programming overhead to the user and thus lowers productivity as substantial modifications to the original graph algorithm are often required.(Section 2)

In order to address this issue, we introduce a high-level domain-specific language (DSL) for graph analysis. That is, we let the users program the graph algorithms intuitively using the DSL, and then let the compiler translate it into the target system with its different programming model and API, in an efficient way. Note that it is not a new idea to use a DSL for this purpose. SQL is a classic example of an intuitive interface language for accessing a relational database. Pig [11] and Hive [15] are also good examples that give significant productivity benefits for MapReduce environments.

This paper reports our early experience of using a DSL for large-scale graph analysis. Specifically, we encode a few popular graph algorithms with the Green-Marl [5] DSL. We then use the Green-Marl compiler to generate an equivalent Giraph application out of the given Green-Marl program. We discuss the following topics regarding this approach:

- **Productivity:** How intuitive or how hard is it to program graph algorithms with Green-Marl? (Section 3)
- **Performance:** How is the quality of the compiler-generated code? How much performance overhead does it induce?

Proceedings of the First International Workshop on Graph Data Management Experience and Systems (GRADES2013) Jun 23, 2013, New York, NY, USA

(Section 4)

- **Other issues:** What are other practical benefits and issues in using a DSL? Especially, how can the DSL approach be seamlessly integrated into a conventional data analysis system? (Section 5)

We draw some conclusions in Section 6 while we use the next section (Section 2) to provide some background information.

2. BACKGROUND

2.1 Pregel, Giraph and Their Programming Model

Pregel [10] is a distributed, in-memory graph analysis framework, in which the vertices of the graph are distributed across multiple worker machines. The original publication showed that the framework is highly scalable.

The Pregel framework adopts a special programming model:

- **vertex-centric**: The user implements a single method (`vertex.compute()`) which describes the behavior of each vertex. The method is applied to every vertex in the graph in parallel.
- **stateful and iterative**: The same `vertex.compute()` function is applied over multiple time-steps. Vertex-private data is maintained between time-steps.
- **message-passing**: For inter-vertex communication, a vertex can explicitly send messages to other vertices. Globally shared data can be implemented via a special `aggregator` API.
- **bulk-synchronous** [17]: Conceptually, every vertex computation happens in parallel at each time-step, while global barrier synchronization is enforced at the end of the time-step. All the messages generated in a time step are instantly delivered at the beginning of the next time-step.

Giraph [1] is an open-source implementation of Pregel, with a few enhancements. First, Giraph workers are implemented as *Hadoop Mappers*; therefore, Giraph tasks can be trivially integrated with all the other Hadoop infrastructure components, such as HDFS (Hadoop Distributed File System), Pig, and Hive. Note that this is very useful when a large graph instance is generated from the even larger raw data-set in HDFS as there is no need to move a large amount of data only for graph analysis. Second, Giraph adopts the concept of `master.compute()` [13], which allows master-side, sequential computation between each parallel vertex computation.

2.2 Compiling Green-Marl into Giraph

In this study, we use Green-Marl [5], a DSL for describing graph analysis algorithms. In contrast to Pregel's programming model, Green-Marl adopts an imperative, shared-memory style programming model. It also provides various built-in data types, operators, and functions which allows for intuitive programming by users, while still exposing important semantic information to the compiler. We omit providing details of the language here, but the language specification is publicly available [2].

In order to accommodate large-scale graph analysis, we extend the existing Green-Marl compiler such that it transforms the given Green-Marl program into an equivalent Giraph program. Note that such a transformation is not trivial because these two programs assume very different programming models; Green-Marl programs are written in a shared-memory imperative style, while Pregel programs are written in a bulk-synchronous, message-passing, vertex-centric style. Here we present an overview of this compiler transformation. The details about this transformation are outside the scope of this paper, but available in another manuscript [6].

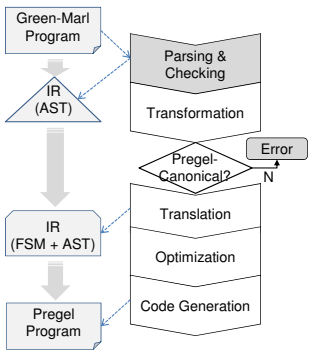

Figure 1: Compilation steps: Initially, the compiler uses an annotated abstract syntax tree (AST) as the internal representation (IR). After transformation steps, the compiler uses another IR that is composed of both a finite state machine (FSM) and an AST.

Figure 1 depicts the overall flow of our compilation steps. Once the input Green-Marl program is parsed and type-checked, the program is first internally represented as an annotated syntax tree. Then the compiler applies multiple transformation rules, while trying to turn the given program into Pregel-compatible form. For instance, the following program is not Pregel-compatible, because each vertex `n` is reading its neighbors `t`'s `bar` value while Pregel does not allow remote data reading:

```
Foreach(n: G.Nodes)
  Foreach(t: n.Nbrs)
    n.foo += t.bar
```

The compiler transforms the above program into the following form, which is Pregel-compatible; now each vertex `t` pushes its own `bar` value to incoming neighbors, which can be implemented as messages.

```
Foreach(t: G.Nodes)
  Foreach(n: t.InNbrs)
    n.foo += t.bar
```

Note that, however, some graph algorithms are inherently sequential (e.g. Tarjan's strongly connected algorithm) and thus not compatible with Pregel in any realistic way. If the compiler fails to transform the given program into Pregel-compatible form, it simply reports an error and stops.

On the other hand, in the case of a Pregel-compatible Green-Marl program, The compiler identifies parallel and sequential execution phases of the algorithm and analyzes the control structure between them. From this information, the compiler creates a finite state machine (FSM) in which each state corresponds to a time-step in Pregel execution. The compiler examines information that is exchanged between states and converts them into messages. The compiler also applies several optimization rules to minimize the number of time-steps and the size of the messages. Finally, the compiler creates the resulting Giraph programs with the appropriate API calls, including all the required boilerplate code.

2.3 Example Graph Algorithms

In this study, we perform an early evaluation of using Green-Marl for large scale graph analysis, using a few popular, important graph algorithms as our example algorithms. The algorithms used in our study are as follows:

1. PageRank [12]: PageRank is a very famous algorithm to compute a weight of a vertex based on the link structure of

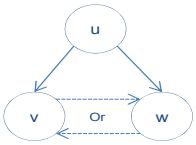

Figure 2: The Counted Triangle Pattern in Directed Graphs

the graph. The PageRank value of each vertex is determined by the weighted sum of PageRank value of its neighborhood vertices; the computation is repeated until all the PageRank values have converged.

2. Triangle Counting [14]: Counting the number of triangles (or closed triads) is a critical step for computing clustering coefficient and detecting community structures. In this experiment, we use a variant of the algorithm which counts only a specific pattern of triangles (Fig. 2) assuming that the input graph is *directed*.

3. Random Walk Sampling with Random Jumps [8]: Sampling is used to obtain a representative (vertex) set of a large graph. Here, we implement a sampling algorithm that performs random walking on the graph with probabilistic random jumping for the sake of escaping from local clusters.

3. PROGRAMMABILITY AND PRODUCTIVITY

In this section, we discuss the benefits of using a DSL and its impact on programmability and productivity. For this purpose, we implement the example algorithms in Section 2 both with Green-Marl as well as directly with the Giraph API. Figure 3 shows the resulting Green-Marl programs. Note that the figure shows the entire programs and not excerpts.

3.1 Current Benefits

Intuitive Programming Model

Graph analysis algorithms implemented in Green-Marl are quite intuitive, as can be seen from Figure 3. Most noticeably, Green-Marl programs are written in an imperative, shared-memory style, which is similar to how the original graph algorithms were specified. Indeed, the Green-Marl implementation of PageRank and Triangle Counting are almost identical to the abstract algorithm description in their original publications [12, 14].

To the contrary, the Pregel programming model may require modification of graph algorithms. For instance, in the original PageRank algorithm, each vertex reads values from its incoming neighbors (line 10). However, in the Pregel implementation, each vertex sends out its PageRank value (divided by its degree) to the outgoing neighbors:

```
// vertex class
public void compute(Iterable<DoubleMsg> msgs) {
  // received messages;
  for (DoubleMsg m : msgs)
     sum += m.get();
  double newV = ((1.0 - d)/N + d*sum); // new PageRank
  // send messages to 'out-nbrs'
  sendMsgToNbrs(new DoubleMsg(newV/numEdges()));
  ...
}
```

In other words, the original algorithm uses *data-pulling* while the Pregel API only allows *data-pushing*; therefore the algorithm has to be modified accordingly. The Green-Marl compiler automatically handles such transformation between *data-pulling* and *data-pushing*

```
1  Procedure PageRank (G: Graph, e,d:Double, max:Int
2    ; pg_rank: Node_Prop<Double>) {
3    Double diff;
4    Double N = G.NumNodes();
5    Int cnt = 0;
6    Do {
7      diff = 0;
8      Foreach(t: G.Nodes) {
9        Double val = (1-d) / N +
10         d * Sum(w:t.InNbrs) {w.pg_rank / w.Degree()};
11       diff += | val - t.pg_rank |;
12       // pg_rank updated synchronously after t-loop
13       t.pg_rank <= val @ t;
14     }
15     cnt ++;
16   } While ((diff > e) && (cnt < max));
17 }
```

(a) PageRank

```
18 Procedure Triangle_Counting(G: Graph) : Long {
19   Long T = 0;
20   Foreach(v: G.Nodes) {
21     Foreach(u: v.Nbrs) {
22       Foreach (w: v.Nbrs) (w > u) {
23         If (w.HasEdgeFrom(u) || w.HasEdgeTo(u))
24             T ++;
25   } } }
26   Return T;
27 }
```

(b) Triangle Counting

```
28 Procedure Random_Walk_Sampling(G: Graph,
29   p_sample: Float, p_jump: Float,
30   num_tokens:Int ; Selected : Node_Prop<Bool> ) {
31
32   // temporary properties
33   Node_Prop<Int> Token, TokenNxt;
34
35   // Initialize
36   G.Token = 0;
37   G.TokenNxt = 0;
38   G.Selected = False;
39   Long N = G.NumNodes() * p_sample; // sample size
40
41   // Choose initial nodes and give them tokens
42   Node_Set S;
43   While (S.Size() < num_tokens) {
44      Node n = G.PickRandom();
45      S.Add(n);
46   }
47   S.Token = 1;
48
49   // Repeat random walk until sample size reaches N
50   Long size = 0;
51   While (size < N) {
52     Foreach(n : G.Nodes) (n.Token > 0) {
53       // Increase sample size at first token reception
54       If (!n.Selected) {
55         n.Selected = True;
56         size++;
57       }
58       While (n.Token > 0) { // randomly shoot out tokens
59         n.Token--;
60         If ((n.Degree() == 0) || (Uniform() < p_jump)) {
61           // global random jump
62           Node m = G.PickRandom();
63           m.TokenNxt ++;
64         } Else {
65           // local random jump
66           Node m = G.PickRandomNbr();
67           m.TokenNxt ++;
68     } } }
69     G.Token = G.TokenNxt;
70     G.TokenNxt = 0;
71 } }
```

(c) Random Walk Sampling

Figure 3: Green-Marl Implementation of Sample Graph Algorithms

Moreover, the compiler automatically generates code for computational state management, which is also required for the vertex-centric computation model. For instance, the manual Giraph implementation of the random walk sampling algorithm contains the following state machine:

```
class RandomWalkMaster ... {
  public void compute(...) {
    switch(state) {
      //phase 1: initialization
      case 1: init_token_set(); break;
      //phase 2: main computation
      case 2: aggregate_sample_size(); break;
    }
    if (sample_sz > N) halt();
} }
```

A similar state machine has to be implemented for the vertex class. Such state management code grows even longer if the target algorithm is composed of multiple computation kernels.

Finally, intuitive programming is also very valuable for the sake of software maintenance. Green-Marl programs like in Figure 3 are short and intuitive so that they can be easily understood by people who didn't originally write the program and thus can be maintained over a long period of time.

Automatic Application of Optimizations

While generating the Giraph programs, the Green-Marl compiler automatically applies a set of optimizations that (1) enable expressing certain functionalities with the Giraph API or (2) enhance the performance of the generated code.

For example, the Triangle Counting algorithm requires the incoming neighbors as well as the outgoing neighbors (Line 23 in Figure 3). However, the original Pregel(Giraph) API only provides outgoing neighbors. In this case, the compiler automatically inserts extra states in the FSM that compute incoming neighbors. That is, at the first timestep each vertex sends its own ID to its outgoing neighbors, and at the next step each vertex construct the incoming neighbor list from those messages. Note that our compiler inserts these extra states only if the given program requires incoming neighbors.

As another example, for PageRank and Random Walk Sampling, the compiler identifies that there are message passing computation kernels inside a `while` loop. In such a case, the compiler automatically combines the message receiving of the last kernel with the message sending of first kernel in the `while`-loop. This optimization reduces the number of time-steps in the generated program.

Certainly, the programmers could have applied the same programming tricks manually when they hand-code the Giraph application. However, since these techniques are automatically applied by our compiler, even the programmers who are not aware of these techniques can have the same benefits.

Free of Boilerplate Code

Table 1 compares the size of Green-Marl programs and hand-written Giraph programs using both Lines-of-Code and number of bytes. Noticeably, Giraph programs are a lot longer than Green-Marl programs, even though they implement the same algorithm. Most of the size difference comes from boilerplate code required by the Giraph API. For instance, the program has to define a message and vertex/edge data class, implement their serialization methods, register aggregators, declare input graph loaders and output writers, and specify job configurations. In our approach, the Green-Marl compiler automatically generates all this lengthy boilerplate, thereby providing additional productivity benefits.

3.2 To be Improved

The Learning Curve

	Green-Marl		Manual Giraph	
Algorithm	LOC	#Bytes	LOC	#Bytes
PageRank	19	516	188	6633
Triangle Counting	14	300	168	6272
Random-Walk Sampling	53	1168	444	16766

Table 1: Comparison of Line of Codes (LOC) and number of bytes between Green-Marl and manual Giraph implementation of the same algorithms

Being a DSL, Green-Marl requires the user to undergo a certain amount of learning. The learning curve is not unreasonably steep; the language looks like just another descendant of C and doesn't add too many new concepts other than graph-specific built-in types and operators

However, since Green-Marl is a research-level language under development, it certainly lacks a detailed documentation or a user community, which hampers the learning experience quite a bit.

Non-Giraph-Compatible Algorithms

When the given algorithm is an inherently sequential one, however, the Green-Marl compiler cannot turn it into a Giraph program. The compiler simply reports an error when it fails to transform the given program into a Pregel-compatible form (Section 2). In such a case, it is up to the programmer to create a new algorithm that is compatible with Pregel.

Among our examples, the original Random Walk Sampling algorithm [8] is inherently sequential and thus cannot be compiled into Giraph. We created a parallel version of this algorithm by introducing the concept of *tokens*; we let multiple agents, as many as `numTokens`, walk the graph in parallel, where any vertex having a token becomes an agent. Note that the resulting Green-Marl program (Figure 3.(c)) is still written in an imperative, shared-memory style without any boilerplate code.

Although a user intervention is disappointing as it diminishes the benefit of the DSL, it is inevitable for those algorithms that are inherently not Pregel-compatible. The user has to create a new algorithm which can be targeted to the underlying computational system. Indeed, the two sampling algorithms (the original algorithm and our parallelized one) are not strictly equivalent; the original algorithm produces a sample of the exact size, while the parallel one produces a sample that is larger (by the number of tokens) than the requested size.

Currently, we are taking a practical approach to this issue. (1) The compiler recognizes frequent (non-Pregel-compatible) patterns in graph algorithms and transforms them into Pregel-compatible ones automatically. (2) We define a subset of Green-Marl programming patterns that are guaranteed to be compilable into Giraph. Therefore, the compiler points out non-Pregel-compatible portions of the program so that users can re-write these portions using the Pregel-compatible Green-Marl subset; thus the compiler is helping the user with the inevitable algorithm modification.

Compiler Maturity

Conversely, one can ask about the completeness of our approach: can every Pregel-compatible algorithm be compiled from its Green-Marl description? Fundamentally, we believe that the answer is yes [1]; however, the current compiler needs more improvement as it does not support all the features required for the Pregel-compatible subset. Specifically, the current compiler only provides limited support for collection data types and aggregation on collections for Giraph compilation. User-defined data types are also not supported yet. Nevertheless, these are not fundamental obstacles but mostly

[1] we can make a mapping from every message-passing call in Pregel to a write statement in Green-Marl

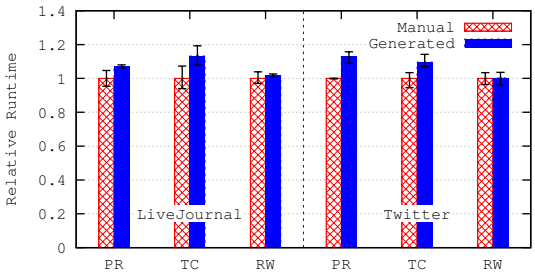

Figure 4: Runtime Comparison Between Hand-coded and Compiler-Generated Giraph Implementations: PR, TC and RW stands for Pagerank, Triangle Counting and Random Walk Sampling, respectively

engineering issues that can be improved upon over time.

Overall, we believe that programming in Green-Marl can provide considerable productivity benefits for graph analysis as it matures. Ease of programming is especially useful when the user wants to explore different graph algorithms in a short time and quickly try out new graph algorithms, or when they wish to customize an existing graph algorithm for their own purpose.

4. QUALITY OF THE COMPILER GENERATED IMPLEMENTATION

In this section, we discuss the quality of the compiler-generated Giraph implementations, focusing on their performance. For this purpose, we evaluate both the hand-coded and compiler-generated Giraph implementation of the three example algorithms in Section 2.3[2] on an x86 cluster and examine their performance characteristics.

Our cluster is composed of 8 machines, with each machine having 16 cores (32 HW threads) and 256GB of memory, running CentOS 6. We used the 1.0.4 release of Hadoop and a snapshot of Giraph 0.2.203 taken on Feb 28th, 2013. All applications were compiled with Oracle jdk 1.7.017 and ran with the corresponding Oracle HotSpot 64-bit JVM. We only instantiate 10 workers per machine, i.e. 80 workers in total; since each worker process already consists of a few threads (for computation and communication), adding more workers per machine impacted performance negatively.

We ran the algorithms on two public graph instances; one is a web graph instance (LiveJournal) from the SNAP graph repository [3], and the other is a social graph extracted from Twitter user relationships [7]. LiveJournal is moderate in size (4 million vertices and 34 million edges), while Twitter is larger (40 million vertices and 1.8 billion edges). Figure 4 depicts the resulting execution time of each algorithm on these graph instances while the y-axis shows the relative execution time, i.e. normalized by execution time of the hand-coded giraph application.

4.1 Current Benefits

Decent Performance of Compiler-Generated Implementations

As can be seen in Figure 4, the compiler-generated Giraph implementations perform fairly close to the hand-written ones. Even though the compiler-generated implementations were slower than manual ones for Pagerank and Triangle Counting, the difference was only about 10 $\sim$ 15%. In the case of Random Sampling, there was virtually no difference at all.

The small difference is the result of several compiler optimizations applied during code generation. For instance, the compiler merges multiple states into one as long as there is no data dependency between them, reducing the number of global barriers. We refer interested readers to a more detailed report that explains how our compiler works [6]. Reasons for the remaining performance difference will be discussed in the next subsection.

The case of the Triangle Counting algorithm is worth a closer look as the first hand-coded version was actually slower than the compiler-generated one. In fact, it did not even finish for the Twitter graph instance in a reasonable time. This was because the programmer used (from habit) `EdgeListVertex` as base class, which has a small memory footprint and provides fast iterate over neighbors, but is very slow when checking whether there is neighborhood relationship. This mistake was not evident during the initial *testing* phase as small graph instances are typically used. Note that with DSL-apporach, the user doesn't have to worry about such issues but can rely on the compiler to handle them.

Readability of Generated Code

We remind the readers that the compiler-generated code is just a plain Java program that uses the Giraph API. In fact, the generated code is fairly readable. The code is appropriately indented and variable names in the original Green-Marl programs are preserved as much as possible. Moreover, each generated Java code block contains the comments of the original Green-Marl source from which the Java code is generated.

For practical reasons, we strive to ensure that the programmer is able to read the generated code and even to edit the generated code; for instance if Giraph introduces a new feature or API, the programmer can manually apply *hot-fixes* to the generated code, before the compiler gets updated.

4.2 To Be Improved

Sub-optimal Performance

Still, there is a certain performance difference between the hand-coded Giraph implementations and the compiler-generated ones, for PageRank and Triangle Counting algorithm in Figure 4. In both cases, the performance overhead came from the more generic approach taken by the compiler.

For the case of PageRank, the compiler is not utilizing the message combiner API at all, since in general there can be multiple message types that are used (even inside one kernel), while the Giraph API only allows one combiner class which will be applied to every message. However, it is possible to extend the compiler such that it can handle *special* cases differently. In this particular case, the complier can recognize that there is only one type of message being used for summing values at the destination, and thus it can use the summation combiner.

For the case of Triangle Counting, the manual implementation uses `HashMapVertex`, which has a large memory footprint, is slow for iteration, but fast in checking neighborhood relationships. On the other hand, the compiler-generated implementation, as a trade-off, uses `EdgeListVertex` accompanied with a memory-efficient side structure (i.e. a sorted primitive array), which is good for both iteration and neighborhood checking.

Again, the compiler can recognize that the given program is a special case which has only one kernel that does nothing but neighborhood checking, and it can decide to use `HashMapVertex` instead. Fundamentally, the Green-Marl DSL exposes enough semantic information for the compiler the apply these specialization techniques.

[2]We modified the triangle counting algorithm for the sake of this experiment, since there is a fundamental issue related to the BSP computation model for this algorithm. See Section 4.2 for details.

Limitation of the Target Runtime System

The Green-Marl generated implementations are still bound by the fundamental limitations of the underlying Giraph runtime. For instance, a Giraph implementation of the original Triangle Counting algorithm in Figure 3 when used with the Twitter graph instance, (or even with the LiveJournal instance), simply fails to execute.

This is because these graphs show a highly skewed degree distribution such that there exist a small number of high-degree nodes, i.e. nodes that have millions of neighbors. Note that the Giraph implementation of the original algorithm produces O($degree^2$) number of messages per each vertex. For vertices with a millions of neighbors , this number reaches the trillions. Worker processes run out of memory, since Giraph keeps these messages in-memory until the next time-step begins. In our experiment, we only counted triangles that originated from small degree nodes (i.e. smaller than 100) so that we can still evaluate our approach.

It would also be possible, though, for the compiler to warn the users in advance, if it sees a pattern that might result in a low-performing implementation due to message count explosion.

5. OTHER BENEFITS AND ISSUES

Debugging New Algorithms

Green-Marl can provide additional benefits for debugging new graph algorithms. First of all, we believe that programming using Green-Marl would be less error-prone than directly programming Giraph, since it is more intuitive and more succinct.

Moreover, since the Green-Marl compiler can generate a fast running (parallel) shared-memory C++ program from the same input [5], the algorithm developer can test their algorithm in a fast way with small-sized graphs on a single machine, without having to use a Hadoop cluster at all. This allows for much shorter turnaround time in testing a new algorithm.

However, the user may still suspect that the compiler injects errors in code generation. Currently, we allow the users to inspect the generated code since it is very readable. The compiler can also be asked to print the code at each intermediate compilation step.

Future Benefits for Portability

This effort focused on the translation of abstract Green-Marl programs into one specific analysis framework: Giraph. However, Green-Marl is not defined only for the Giraph framework, but for general graph algorithms.

Therefore one can imagine a Green-Marl compiler that translates the same Green-Marl program into other different graph processing frameworks (e.g. GraphLab [9] and Trinity [16]) as well as Giraph. Since these systems generally show different performance characteristics depending on the algorithm or size/shape of input graph, users may choose the appropriate runtime for their purpose.

Indeed, as of now, the user can generate a shared-memory implementation and Giraph implementation from the same Green-Marl program. If the size of the target graph fits in a single machine's memory space, one can use the shared-memory implementation which is generally faster than Giraph execution as it does not suffer from communication cost.

System Integration

Another practical issue is how to integrate the Green-Marl DSL with the rest of a complete data analysis system, in which graph analysis is only one phase in the analysis flow. A complete data analysis system may include persistent storage of graph instances or the materialization of a temporary graph out of raw data. It may also include other off-line analysis engines (e.g. Hadoop jobs for filtering out uninteresting data) as well as a connection mechanism to on-line systems that make use of the result of off-line analysis.

Currently, our compiler-generated programs (both Giraph and shared-memory applications) load data files either from the local file system or from HDFS assuming that the graph is available as one of a few popular formats such as an adjacency list. Supporting more formats or user-defined loaders would make it more flexible.

On the other hand, we can also think of extending the DSL such that it can declare inputs and outputs of a graph analysis in a more abstract and declarative way. For instance, the input can be loaded from a data file, database or even directly pipelined from other analysis. Similar approaches of other successful DSLS (e.g. Pig and LINQ) can be applied for extending Green-Marl as well.

6. CONCLUSION

This paper reported on our early experience with using Green-Marl as a front-end language for large-scale graph analysis, by compiling Green-Marl programs into Giraph applications. We observed that Green-Marl programs are concise and intuitive while the performance of the compiler-generated code closely matches hand-tuned applications. Overall, we think this approach could provide considerable productivity benefits as the compiler matures.

Acknowledgements

We appreciate Sam Shah, Roshan Sumbaly and Evion Kim at LinkedIn for their collaboration in this study.

7. REFERENCES

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

[2] Green-Marl DSL. http://github.com/stanford-ppl/Green-Marl.

[3] Stanford network analysis library. http://snap.stanford.edu/snap.

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

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

[6] 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.

[7] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? WWW '10, 2010.

[8] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, 2006.

[9] 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.

[10] 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.

[11] 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.

[12] L. Page. Method for node ranking in a linked database, Sept. 4 2001. US Patent 6,285,999.

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

[14] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*. ACM, 2011.

[15] 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.

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

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