Graph DSLs : A Survey on Green-Marl & Sparql

A Report Submitted in Partial Satisfaction of the Requirements for an Independent Research 2014.

Thejaka Amila Kanewala
thejkane@iu.edu
School of Informatics & Computing
Indiana University
Bloomington, IN. USA

ABSTRACT

Many real world problems are formulated as graphs and standard graph processing algorithms are used to search solutions. Applications of graphs and related algorithms can be found in many domains. Domains vary from standard scientific applications to social media applications such as facebook. Creating and processing graphs in HPC environments adds lot of complexities. Hiding detail complexities and yet achieving the necessary performance enables lot of people to develop graph applications. In this paper we discuss about 2 such graph specific programming languages; 1. **Green-Marl** - an imperative programming language for graph processing, 2. **SPARQL** - An SQL like query language for graph processing.

General Terms

DSL, Graphs, SQL, HPC

Keywords

Domain Specific Languages, High Performance Computing, Graph Processing, Algorithms

1. INTRODUCTION

There are established parallel graph processing frameworks. **Parallel Boost Graph Library (PBGL)** [9] and **Pregel** [17] are two such frameworks. Using such systems need knowledge about underlying programming languages. For example using PBGL needs thorough understanding about C++ Template Programming in addition to **Boost Graph Library (BGL)** [21]. Domain Specific Languages for Graph processing hides these complexities in underlying processing frameworks and let user works on the graph problem based on the mathematical definition of the graph. (Mathematical definition from [26]: Graph is an ordered pair G = (V, E) comprising a set V of vertices or nodes together with a set E of edges or lines.). In this paper we discuss about 2 domain specific programming languages for graph processing, from 2 different paradigms.

In this paper we (authors) do not wish to present new findings or concepts. Our main focus is to analyse 2 existing Graph DSLs and summarise their features and compare pros and cons.

Green-Marl (GM) [13] was developed at Standford University in collaboration with Oracle corporation. Green-Marl consists of a front-end compiler and a back-end. As per now GM have several backend implementations. The front-end compiler is capable of converting Green -Marl code to a preferred backend implementation. GM supports following backend implementations; 1. Sequential C++ 2. Concurrent C++ 3. Giraph [2] 4. Pregel [17]

In Section 2 of this paper we will do an in-depth analysis of Green-Marl DSL. We will discuss Green-Marl in-terms of language features offered by GM, Parallelism offered by GM and finally a performance comparison between GM vs. BGL.

SPARQL [20] is more like a query language rather than a standard programming language. SPARQL is close to SQL interms of behaviour and syntax. SPARQL is originally developed as the query language for **Resource Description Framework (RDF)** [16] graphs. In Section 3 we will analyse the usability of SPARQL as a general purpose graph processing framework.

In Section 4 we will discuss about advantages and disadvanteges of SPARQL and Green-Marl in the context of large scale graphs.

2. GREEN-MARL

Figure 1 shows the high-level architecture of Green-Marl compiler. The Graph logic is expressed using Green-Marl language constructs. Then front-end compiler translates Green-Marl program into C++, CUDA [18] or Java depending upon the back end requirement. During translation Green-Marl compiler applies code level optimisations to Green-Marl logic (We will discuss these optimisations in Section 2.3). Green-Marl tries to generate code that achieves maximum performance through parallelism. For backend, Green Marl is capable of generating code compatible with Pregel and Giraph. But in this paper our main focus is C++. Green Marl is capable of generating C++ code that achieves

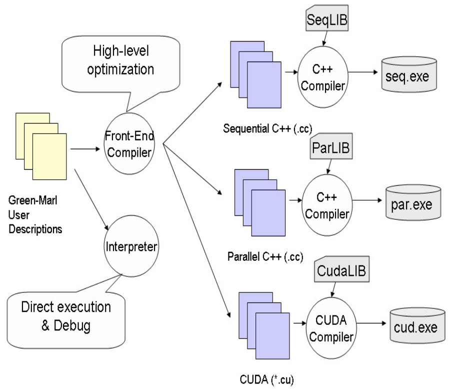

Figure 1: An Overview of Green-Marl Compiler (Source: Green-Marl Language Specification v 0.5.0 [14])

thread level parallelism. GM uses OpenMP [19] threading framework to achieve thread level parallelism. But Green-Marl does not provide any process level parallelism in their C++ implementation. (i.e. No MPI [10] support)

In the next sub-section we will discuss Green-Marl language constructs in detail.

2.1 Green-Marl Programming Language

Green-Marl is a **statically typed** (Types are known at compile time) and **lexically scoped** programming language. It consists of several graph specific built-in data types; **Graph, Node, Edge**.

Further GM provides several control flow mechanisms to do iterations ForEach, InBFS, etc. Also GM provides several data structure types and types to store properties of a graph; Node_Property, Edge_Property, etc.

The Single Source Shortest Path (SSSP) [29] algorithm is coded in Green-Marl as follows;

Listing 1: Bellman Ford like SSSP Algorithm written in GM.

```
Procedure sssp(G:Graph, dist:N_P<Int>, len:E_P<Int>, root: Node) // defining procedure
{
    Node_Property<Bool> updated;     // defining a node property
    Node_Property<Bool> updated_nxt; // defining another node property
    Node_Property<Int>  dist_nxt;    // more node properties
    Bool fin = False;

    G.dist = (G == root) ? 0 : +INF; // Initialize all nodes to +INF except source node
    G.updated = (G == root) ? True: False; // Set update to False for all nodes except for source
    G.dist_nxt = G.dist; // Update dist_nxt in the same way as dist
    G.updated_nxt = G.updated; // Update update_nxt with updated

    While(!fin) {
        fin = True;

        Foreach(n: G.Nodes)(n.updated) { // For each node n where ``updated`` is true in the graph do
            Foreach(s: n.Nbrs) { // For each neighbor s of n
                Edge e = s.ToEdge(); // the edge to s
                // updated_nxt becomes true only if dist_nxt is actually updated
                    // s.dist_nxt = min(s.dist_nxt, (n.dist + e.len))
                    // if (s.dist_nxt > (n.dist + e.len)) {
                    //         s.dist_nxt = n.dist + e.len;
                    // s.updated_nxt = true;
                    // }
                <s.dist_nxt; s.updated_nxt> min= <n.dist + e.len; True>;
            }
        }

        G.dist = G.dist_nxt;
        G.updated = G.updated_nxt;
        G.updated_nxt = False;
        fin = ! Exist(n: G.Nodes){n.updated};
    }
}
```

We will use above example as a basis and go through language constructs in Green-Marl.

GM is not designed as a complete programming language (like C++ or Java). In the sense one cannot write a whole application using GM. We use GM only for graph processing part. The following example describes this behaviour; Let‘s say we have an application which needs to process web page connectivity. We need to calculate minimum cost for each connecting web page from a designated web page. For this we write an application using C/C++ and use GM graph API to construct the graph of connecting web pages and weight of each edge. Then we can use GM to write the shortest path algorithm (As depicted in Listing 1; Let‘s name the file as sssp.gm). Then we use Green-Marl compiler to generate C++ code as follows;

```
>gm_comp -t=cpp_omp sssp.gm -o ./src
```

Above command will generate C++ code for sssp.gm program. The code will use OpenMP [19] threading model and output files are written to src directory. Above command mainly generates sssp.h file and sssp.cc file. In sssp.h file you will find a function as follows;

```
void sssp(gm_graph& G, int32_t* G_dist,
  int32_t* G_len, node_t& root);
```

Above C++ function is generated using Green-Marl procedure signature (i.e. Procedure sssp(G:Graph, dist:N_P <Int>, len:E_P<Int>, root: Node) in Listing 1). In above function, G represents the graph constructed using Green-Marl API, G_dist is the distance array for each vertex in the graph, G_len contains weights of each edge indexed by edge number (GM graph gives an id to each edge and ids are generated sequentially), root is the source vertex.

2.1.1 *Procedures*

Green-Marl defines two types of procedures. 1. Entry Procedure 2. Local Procedure

An entry procedure starts with keyword “Procedure“ or “Proc“. what we saw in Listing 1 is an entry procedure. An entry procedure is the main entrance point to the algorithm. Within the algorithm it can define local procedures. A local procedure can only be called from an entry procedure or another local procedure. An entry procedure must be called from the main application code. The entry procedure should only be called from a “virtually sequential context“ i.e. at the time of invocation we need to make sure there are no concurrent threads which modifies arguments to procedure.

2.1.2 Types

All the types in GM are determined at compile time. It does not allow user defined types, also there is no notion of inherited types. GM categorises types mainly into 3. 1. Primitive Types 2. Graph Types 3. Collection Types.

As the primitive types, GM defines, Int, Long, Float, Double and String. For example if you have GM code in Listing2, the equivalent C++ code is in Listing3.

Listing 2: Sample code on use of primitive types.

```
Proc example(b: Int)
{
        Int i = 3;
        Float f = 0.1;
        Double d = 0.2;
        Bool k = False;
}
```

Listing 3: Equivalent C++ code for above GM code.

```
void example(int32_t b)
{
    //Initializations
    gm_rt_initialize();

    int32_t i = 0 ;
    float f = 0.0 ;
    double d = 0.0 ;
    bool k = false ;

    i = 3 ;
    f = ((float)(0.100000)) ;
    d = ((float)(0.200000)) ;
    k = false ;
}
```

Type conversions for Int, Long, Float, Double and Bool are carried out as in C and String defines set of in built operations which are quite similar to C++ strings.

GM defines 2 graph types. **DGraph** (also named **Graph**) which represents a directed graph and **UGraph** which represents an undirected graph. In GM all graphs are immutable. Graphs can only be passed as arguments to procedures and cannot be instantiated within GM program code. So the graph construction must be done at the application using GM graph API. Further GM defines Node and Edge types. These types are bound to graphs. An example of Node usage is given in Listing 4.

Listing 4: Use of Node type in function parameters.

```
Proc f(G1, G2: Graph, n : Node(G1), m:
    Node(G2)) {
...
}
```

But the generated code does not check whether node n is a node of graph G1 and node m is a node of graph G2. For code in Code in Listing 4 the generated code is given in Listing 5.

Listing 5: Generated C++ code for GM code in Listing 4.

```
void f(gm_graph& G1, gm_graph& G2,
    node_t& n, node_t& m)
{
...
```

As depicted in Listing 5, both n and m are defined with “node_t“. Therefore at application run time we cannot distinguish n as a node of G1 and m is a node of G2. This we see as a drawback of GM.

Further GM defines set of operations on Node and Edge types. (E.g :- Nbrs (neighbours), OutNbrs (out edges) etc.)

GM provides property types to associate values with nodes and edges.

Listing 6: Passing graph property A as a parameter and defining a new graph property B.

```
Proc g(G: Graph, A: Node_Prop<Int>(G)) {
        Node_Prop<Int>(G) B;
}
```

Listing 7: The generated C++ code for Green-Marl code in Figure 6.

```
void g(gm_graph& G, int32_t* G_A)
{
    //Initializations
    gm_rt_initialize();
    G.freeze();

    int32_t* G_B =
        gm_rt_allocate_int(G.num_nodes(),
        gm_rt_thread_id());

    gm_rt_cleanup();
}
```

The C++ code generated for Listing 6 is shown in Listing 7.

Algorithm in Listing 1 defines 3 property maps (updated, updated_nxt, dist_nxt).

In addition to property types GM also provides collection types to achieve orderedness and uniqueness properties. Primarily GM defines collections; Set, Order and Seq. Further each collection type is defined for nodes and for edges. A set preserves uniqueness, while a sequence preserve only orderedness. To preserve both orderedness and uniqueness we can use “Order“ collections. Green-Marl also supports collection of collections and map types to store key value pairs.

2.1.3 Parameter Passing & Return Values

In GM parameter passing mechanism depends on the type of the argument. All the primitive types and Node, Edge types are passed by value. All other types (i.e. Graph, Property, Map types) are passed by reference. GM also support the concept of output arguments. For example, in following code “min“ is an output argument. Output arguments are stated after input arguments, separate by a “;“.

Listing 8: An example of an output argument.

```
Local get_min(a,b : Int; min: Int) : Int {
}
```

In the generated C++ code the min is passed by reference.

Listing 9: Generated C++ code for GM code in Figure 8.

```
static int32_t get_min(int32_t a, int32_t b,
    int32_t&  min) {
}
```

2.1.4 *Iteration Constructs*

GM provides 2 language constructs for iterations. They are “for“ and “foreach“. The “for“ loop is always executed in sequentially. The “foreach“ iteration tries to parallelize using OpenMP [19] threads. But parallelization is carried out with synchronization locks when necessary. We will discuss about parallelization more detail in section 2.2.

We illustrate the difference between “for“ loop and “foreach“ loop using code Listings in 10 and 11 ;

Listing 10: Difference between “for“ and “foreach“ - GM code.

```
Proc foo(G: Graph) {
        Map<Node(G), Int> map;

        Foreach(n: G.Nodes) {
                map[n]++;
        }
}
Proc boo(G: Graph) {
        Map<Node(G), Int> map;

        For(n: G.Nodes) {
                map[n]++;
        }
}
```

Listing 11: Difference between “for“ and “foreach“ - C++ generated code for GM code in 10.

```
void foo(gm_graph& G) {
    gm_rt_initialize();
    G.freeze();
    gm_map_medium<node_t, int32_t>
        map(gm_rt_get_num_threads(),  0);

    #pragma omp parallel for
    for (node_t n = 0; n < G.num_nodes(); n
        ++) {
        map.changeValueAtomicAdd(n, 1);
    }
}
void boo(gm_graph& G)
{
    gm_rt_initialize();
    G.freeze();
    gm_map_medium<node_t, int32_t>
        map(gm_rt_get_num_threads(),  0);

    for (node_t n = 0; n < G.num_nodes(); n
        ++) {
        map.changeValueAtomicAdd(n, 1);
    }
}
```

2.1.5 *Implicit Bindings*

Nodes, Edges and properties are bound to a graph. But when we work with a single graph we do not need to mention the graph specifically. As an example, in the code in Listing 1 we do not specify to which graph they bind. Since there is a single graph the properties are bound to that implicitly.

2.2 Thread Parallelization in GM

In GM the execution becomes parallel at the beginning of a parallel region and at the end of the parallel region all those parallel executions are merged and becomes sequential again. GM parallelize “Foreach“ loops and “InBFS“ iterations. GM also supports parallelism in nested loops.

2.2.1 *Memory Consistency*

The basic “memory consistency model“ [12] supported by GM is “sequential memory consistency“ [1] . i.e. change made to a memory location is visible to any sentence comes after. “While“ loops and “For“ loops assure sequential consistency.

GM follows a different memory consistency model for “Foreach“ and “InBFS“ loops. The model does not guarantee anything about visibility of concurrent writes. But model gurantees following features; 1. Self visibility : A write by an instance is always visible to current instance later in the program. 2. Eventual Visibility : At the end of parallel region, when all concurrent executions are merged, every write made by concurrent execution becomes visible to subsequent code.

Listing 12: An example of using parallel consistency model.

```
1. Int x = 0;
2. Foreach (n: G.nodes) {
3.      x = n.Color;
4. }
```

In above code, at line 4 the variable “x“ will have one of color values written.

GM also supports Bulk Synchronous memory consistency [8] model. In Bulk Synchronous memory consistency it is guaranteed that a write to a memory location is not visible to every concurrent execution instance, until the synchronization point is reached. Listing 13 shows an example GM code which uses Bulk Synchronous Memory consistency.

Listing 13: Sample GM code which demonstrates Bulk synchronous memory consistency.

```
Proc boo3(G: Graph) {
        Node_Prop<Int>(G) A;
        Foreach(n: G.Nodes) {
                n.A <= Sum (t: n.Nbrs) { t.A };
        }
}
```

GM supports bulk synchronous consistency through deferred assignment. Deferred assignments are identified by “<=“ symbol. In the example given in Listing 13, value of n.A is not written until loop is completed. Until the loop ends, calculated value of n.A is stored in a temporary variable.

2.2.2 *Reductions*

Reduction [28] calculates a single value from a set of values in a deterministic way. In GM reductions are implemented in 2 forms. 1. As assignments 2. As expressions

First we will consider assignment form. Consider following code;

Listing 14: GM code which demonstrates reductions.

```
Proc boo4(G: Graph) {
        Int x = 0;
        Int y = 0;
        Node_Prop<Int>(G) A;
        Foreach(n: G.Nodes) {
                x += n.A;
                y = y + n.A;
        }
}
```

In above code the "+=" operator is treated as a reduction. But "=" is not considered as an reduction. Similarly there are number of reduction operators available in GM. (E.g := *=, max=, min=.... etc.). In the expression form we can have following like statements as reductions;

Listing 15: Expression based reductions.

```
Int x;
x = Sum (n: G.Nodes) :{ n.A }
```

2.3 GM Performance

We compared GM performance against Boost Graph Library (BGL). We executed "dijkstra's single source shortest path" [22] and compared results.

All tests were carried out in Indiana University "Big Red II" [24] system. We used Hybrid (x86_64 CPUs/NVIDIA Kepler GPUs) machine types and host Operating System is Cray Linux (Based on Suse Linux), with distributed memory. Experiments were run on GPU machines with 16 cores and 32 GB memory per node.

2.3.1 *GM and BGL*

We ran tests for scale 18-25 for both GM and BGL (We were unable to run tests beyond scale 25, as we ran out of memory in a single node) . BGL was run in serial. For each scale we ran GM with a single thread and compared the performance. Figure 2 shows the time taken to complete SSSP algorithm in Boost and GM. GM algorithm was run in a single thread. As per the graph, both implementations have more or less equal performance. But Boost was leading in performance (with a very small number) for all scales.

Next we ran tests by increasing number of OpenMP threads in GM. Figure 3 shows the results for scale 25.

Pattern is similar for other scales. As per the graph, in Figure 3, with one thread, Boost performance is better, but as the number of threads increases GM performance improves rapidly. Then when the number of threads reaches 16 the performance becomes more stable.

Figure 4 shows the effect of number of threads on performance of SSSP algorithm for scales 20-25. As per the graph in Figure 4, for higher scales when number of threads increase, effect on the performance is significant. For example, for scales 25 and 24 when number of threads increased from 1 to 3, algorithm gains a significant performance improvement. This performance increase is not significant for scales 20, 21, 23.

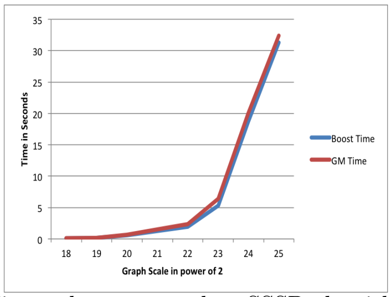

Figure 2: Time taken to complete SSSP algorithm in Boost and GM

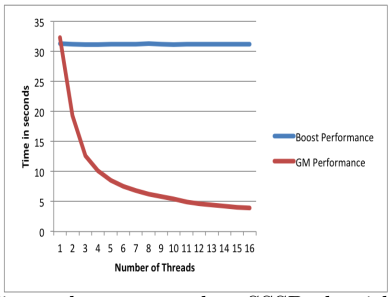

Figure 3: Time taken to complete SSSP algorithm in Boost and GM for a graph of scale 25.

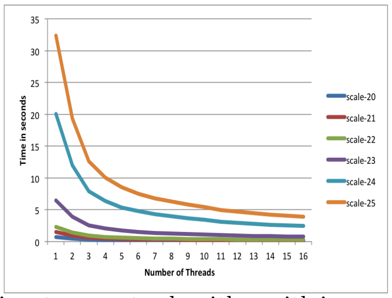

Figure 4: Time to execute algorithm with increasing number of threads for scales 20 - 25.

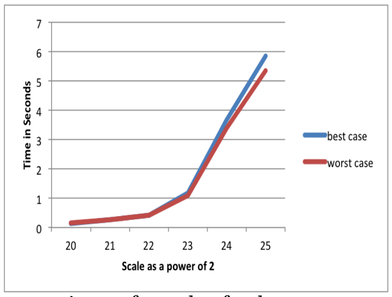

Figure 5: Comparison of results for best case and worst case NUMA effect tests.

2.3.2 *NUMA Effect on GM*

We also wanted to find "non-uniform memory access" (NUMA) support in GM. Specially, BR II is based on AMD processors so NUMA has a significant effect on programs running on AMD processors as per [4]. We did not find any written information claiming that GM supports NUMA. Therefore we designed our NUMA tests as follows; we chose a worst case scenario and a best case scenario. We also ran tests against 8 threads (8 is enforced by BR II architecture; in BR II 8 cpu's are bound to a single NUMA node). So in the best case all 8 threads ran in 0 to 7 consecutive cpus (cores). In the worst case each thread was scheduled to run in a cpu (core) on a different NUMA node. This way each thread runs in a different NUMA node.

As per Figure 5 there is not much of a difference in the results for best case and worst case. Therefore we conclude that GM performs well in NUMA architectures.

3. SPARQL

SPARQL [20] is quite different from Green-Marl. SPARQL is a query language and its behaviour is quite similar to SQL. In this section we will analyse SPARQL as a Graph DSL. SPARQL is tightly bound with **"Resource Description Framework"** (RDF) [16]. Therefore we will briefly discuss about RDF and RDF graphs. Then we will discuss about SPARQL query language with the help of a SPARQL implementation; **"Apache Jena"** [15]. Finally we will conclude the section by discussing suitabiity of SPARQL as a general purpose Graph DSL.

3.1 RDF Graphs

Resource Description Framework or RDF is used to represent information about web resources. A web resource is a particular web content. For example a web page is a web resource. A PDF document in web is a web resource. Such resources have metadata associated. For example for a web page we can find following metadata; 1. Title, 2. Author, 3. Modification Date. For web document we may interest in metadata such as copyright or licensing information.

The main objective of this effort is to let applications process the metadata of web resources and do some actions based on those metadata. In Order to achieve this we need to have a formal method to define web resources. RDF formalizes this by describing each resource as a set of pairs of properties and values. In RDF each resource and resource attribute is identified using an "Internationalized Resource Identifier" [11] (IRI).

The RDF data model is a set of **triples**. Each triple includes following; 1. Subject 2. Predicate 3. Object

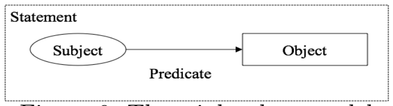

Figure 6: The triples data model

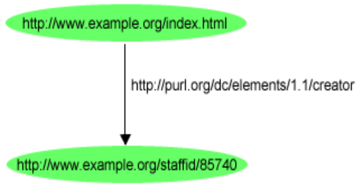
http://www.example.org/index.html
http://purl.org/dc/elements/1.1/creator
http://www.example.org/staffid/85740

Figure 7: RDF graph relevant to the example given above (Source: [3])

A set of such triples is called a **RDF graph**. Each triple gives a meaningful information about subject, object and predicate. Such a triple also known as a statement. Statement is graphically depicted in Figure 6

Example : Consider following statement about resource "http://www.example.org/index.html". (Note: This example is based on reference [3])

"http://www.example.org/index.html has a creator whose value is John Smith"

Above statement can be modeled in RDF as follows;
Subject - http://www.example.org/index.html
Predicate - http://purl.org/dc/elements/1.1/creator
(This IRI represents create action)
Object - http://www.example.org/staffid/85740

The RDF graph relevant to above statement is in Figure 7.

Further we can enhance above graph by adding more information. For example we can add information about staff member 85740. We can add properties like "First Name", "Age" etc. Then we will get a graph in Figure 8.

RDF graphs are always modified in the granularity of a triple. i.e. graph is expanded by adding a new triple or shrinked by removing an existing triple.

3.2 SPARQL Syntax & Semantics

This section briefly introduce SPARQL language and its capabilities. To experiment with queries we mainly used Apache Jena. Our queries will be based on RDF graph in Figure 9.

3.2.1 *Simple Query (also called Triple Pattern)*

Let's say we want to find all subjects who's name is "John Smith". For this we can write a SPARQL query as shown in Listing 16.

Listing 16: Find all subjects who's name is "John Smith".

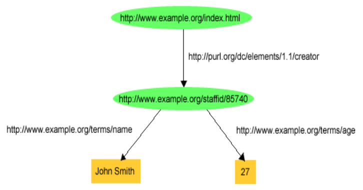
http://www.example.org/index.html
http://purl.org/dc/elements/1.1/creator
http://www.example.org/staffid/85740
http://www.example.org/terms/name
http://www.example.org/terms/age
John Smith
27

Figure 8: Extended RDF graph, added details about staff member 85740 (Source: [3])

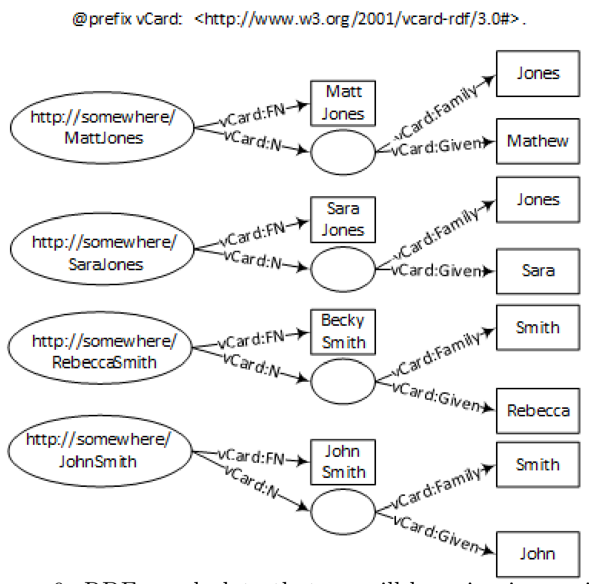
@prefix vCard: <http://www.w3.org/2001/vcard-rdf/3.0#> .
http://somewhere/ MattJones
vCard:FN
vCard:N
Matt Jones
vCard:Family
vCard:Given
Jones
Mathew
http://somewhere/ SaraJones
Sara Jones
Jones
Sara
http://somewhere/ RebeccaSmith
Becky Smith
Smith
Rebecca
http://somewhere/ JohnSmith
John Smith
Smith
John

Figure 9: RDF graph data that we will be using in queries (Source: [15])

```
SELECT ?x
WHERE
 { ?x
      <http://www.w3.org/2001/vcard-rdf/3.0#FN>
      "John Smith" }
```

The result of above query would look like follows;

Listing 17: Results of query 16.

```
---------------------------------
| x                             |
=================================
| <http://somewhere/JohnSmith/> |
---------------------------------
```

In above query the item enclosed within <>is an IRI. If we want to match a plain literal we have to use quotes (““). “?x“ defines a variable called “x“ and it is not part of the variable name. Therefore we only see “x“ in the output.

Let‘s briefly look at how query execution works. The basic mechanism used here is pattern matching. The triple pattern in query’s “WHERE“ clause is matched against the triple patterns in RDF graph. The subject is a variable, therefore subject will be matched to any subject in the RDF graph. But object and predicate are fixed. Therefore the only matching triple is in Listing 18.

Listing 18: Only matching triple for query in Listing 16.

```
<http://somewhere/JohnSmith/>
    <http://www.w3.org/2001/vcard-rdf/3.0#FN>
    "John Smith"
```

Further example in Listing 19 matches only the predicate and returns all matching values for subject and object in variables x and fname.

Listing 19: Query that matches only the predicate.

```
SELECT ?x ?fname
WHERE {?x
    <http://www.w3.org/2001/vcard-rdf/3.0#FN>
    ?fname}
```

Above types of queries are called “Triple Pattern“ queries since there is only one triple in the where clause.

3.2.2 Basic Patterns

A basic pattern is a set of triple patterns. i.e. in the where clause we have more than one triple patterns separated by a “.“. An example query is given in Listing 20.

Listing 20: Query with basic pattern.

```
SELECT ?givenName
WHERE
  { ?y
      <http://www.w3.org/2001/vcard-rdf/3.0#Family>
       "Smith" .
    ?y
        <http://www.w3.org/2001/vcard-rdf/3.0#Given>
         ?givenName .
  }
```

The “WHERE“ clause becomes true when all triple patterns match with the same variable value.

3.2.3 Filters

Filters allow us to limit results in the output.

Listing 21: Query without a filter.

```
PREFIX vcard:
    <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?g
WHERE
        { ?y vcard:Given ?g . }
```

If we execute above query we would get results in Listing 22;

Listing 22: Results of a query without a filter.

```
-------------
| g         |
=============
| "Matthew" |
| "Sarah"   |
| "Rebecca" |
| "John"    |
-------------
```

If we want to retrieve only results which has letter “r“ in the given name, we can apply a filter as in Listing 23.

Listing 23: Query with a filter.

```
PREFIX vcard:
    <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?g
WHERE
{ ?y vcard:Given ?g .
  FILTER regex(?g, "r", "i") }
```

Then we would get following result set;

Listing 24: Reuslts of a filter query.

```
-------------
```

```
| g           |
=============
| "Sarah"   |
| "Rebecca" |
-------------
```

3.2.4 Optionals

Let‘s say we want to retrieve people names and their ages. But retrieving age is optional. Using the knowledge we have so far (i.e. without optionals), we would try to write a query as in 25.

Listing 25: Query without optionals.

```
PREFIX info:   <http://somewhere/peopleInfo#>
PREFIX vcard:
    <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    ?person info:age ?age .
}
```

But in-order for above query to return results we need to match both triples in the “WHERE“ clause. Therefore if age information is optional we will not get certain results from above query. Therefore to retrieve age information only if it is their we can use “OPTIONAL“ clause. Then the query would look like Listing 26.

Listing 26: Query with optionals.

```
PREFIX info:    <http://somewhere/peopleInfo#>
PREFIX vcard:
    <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name ?age
WHERE
{
    ?person vcard:FN  ?name .
    OPTIONAL { ?person info:age ?age }
}
```

3.2.5 Union

Union performs union operation on retrieved result sets. For example consider that in the input graph data, the name is represented in 2 ways as in Listing 27.

Listing 27: Graph data is represented in 2 formats(foaf and vcard).

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix vcard:
    <http://www.w3.org/2001/vcard-rdf/3.0#> .

_:a foaf:name   "Matt Jones" .
_:b foaf:name   "Sarah Jones" .
_:c vcard:FN    "Becky Smith" .
_:d vcard:FN    "John Smith" .
```

Let‘s say we want to retrieve names from both formats. Then we can execute a query similar to 28.

Listing 28: An example UNION query.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vCard:
    <http://www.w3.org/2001/vcard-rdf/3.0#>
SELECT ?name
WHERE
{
   { [] foaf:name ?name } UNION { [] vCard:FN
        ?name }
}
```

In query depicted in Listing 28, we do not consider the subjects we are matching. So the result of above query would look like in 29. First 2 names (Matt and Sarah) coming from “foaf“ format and second 2 coming from “vcard“ format. First 2 names (Matt and Sarah) coming from “foaf“ format and second 2 coming from “vcard“ format.

Listing 29: Results of UNION query in 28.

```
-----------------
| name          |
=================
| "Matt Jones"  |
| "Sarah Jones" |
| "Becky Smith" |
| "John Smith"  |
-----------------
```

3.2.6 Property Paths

Property paths is an important concept when it comes to implementing algorithms using SPARQL. Therefore we will discuss about property paths in briefly. As per the specification [11] property path is a possible route through a graph between two graph nodes. A triple pattern is a property path of length 1. The ends of the path may be RDF terms or variables. Some interesting examples are as follows; 1.***elt****- A path that connects the subject and object of the path by zero or more matches of *elt*. 2.***elt1 / elt2*** - A sequence path of *elt*1 followed by *elt2*. For a detailed list please refer the specification [11]. An example query that uses property paths is given in 30

Listing 30: Query that uses property paths.

```
PREFIX :   <http://example/>
SELECT *
{  ?s :item/:price ?x . }
```

Above query retrieves all nodes that connects
<http://example/item>predicate and
<http://example/price>predicate. This behaviour is graphically depicted in Figure 10

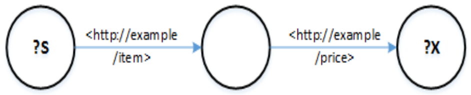

Figure 10: Selecting subjects and objects connecting concatenated path :item/:price

3.2.7 Named Graphs

So far we discussed about queries pertaining to a single graph. A data set in SPARQL is a collection of graphs. In the collection we mainly find 2 types of graphs; 1. The default graph 2. Named graphs

In SPARQL we can specify how to query a specific graph. But if we do not specify which graph to query SPARQL will execute the query on default graph. Further we could execute query on the whole dataset. For that we need to use **"GRAPH"** keyword as shown in Listing 31. The first union clause is executed on the default graph and second query is executed for each named graph and results are union together.

Listing 31: Querying multiple graphs.

```
PREFIX  xsd:
    <http://www.w3.org/2001/XMLSchema#>
PREFIX  dc:
    <http://purl.org/dc/elements/1.1/>
PREFIX  :       <.>
SELECT *
{
    { ?s ?p ?o } UNION { GRAPH ?g { ?s ?p ?o }
        }
}
```

Also we can execute a query against a specific named graph as shown in Listing 32. The query is executed against ds-ng-2.ttl named graph.

Listing 32: Query against a specific graph in dataset.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <.>
SELECT ?title
{
  GRAPH :ds-ng-2.ttl
    { ?b dc:title ?title }
}
```

3.3 Writing Algorithms in SPARQL

One of the main features we expect from a Graph DSL is ability to implement algorithms based on graphs. It turns out implementing algorithms (such as SSSP [29]) using SPARQL is not trivial. The main reason is; SPARQL is a query language, therefore using SPARQL we state what data we need, but we dont specify how to retrieve those data. But an algorithm implementation needs a way to encode "how to instructions" as well as data structure support. In this section we will discuss 2 different approaches to implement algorithms in SPARQL.

"YarcData" [30] discuss about writing an "iterative algorithms" [27] in SPARQL. We will not go into details of that algorithm. However we will discuss their approach.

In an iterative solution, algorithm first starts with an approximate solution then in each iteration it improves the solution and converges to the optimal solution. Achieving such an iterative behaviour in SPARQL is not straightforward. The way YarcData achieves this by having nested loops and using property paths (discussed in section 3.2.6). The depth of the nesting queries will depend on the actual nature of the algorithm. But it can even go up to 9 levels. Having that much of nesting will result in poor performance also the code will get more complicated. As per YarcData reference the poor performance is mainly due to the inner "SELECT" queries. The inner "SELECT" queries increases exponentially with the depth. Also this approach is not strong when we want to process large graphs in a distributed setting. Mainly due to the fact how SPARQL engine going to distribute processing based on the query data.

"Bigdata" [5] approach to implement SPARQL based algorithms is slightly different. The model they adopted is called **"Gather Apply Scatter"** (GAS) [6]. Briefly GAS is a 3 phase execution. Each phase briefly described below (as per [23]); 1. Gather - Collect information from 1-hop neighborhood of a vertex using a binary operator such as Max, Sum etc. 2. Apply - The Apply phase integrates the information from the Gather phase, updating the state of the vertex 3. Scatter - The Scatter phase redistributes information to the 1-hop neighborhood of a vertex.

Bigdata has an implementation of above model and that implementation is exposed to the SPARQL implementation as a service. So the algorithms are implemented using GAS model and SPARQL queries can execute those implemented algorithms using query language. An example query which uses this service is given in Listing 33

Listing 33: Bigdata Breadth First Search implementation using SPARQL.

```
PREFIX gas: <http://www.bigdata.com/rdf/gas#>
SELECT ?depth ?predecessor ?linkType ?out {
  SERVICE gas:service {
      gas:program gas:gasClass
          "com.bigdata.rdf.graph.analytics.BFS"
          .
      gas:program gas:in <ip:/112.174.24.90> .
          # one or more times, specifies the
          initial frontier.
      gas:program gas:out ?out . # exactly
          once - will be bound to the visited
          vertices.
      gas:program gas:out1 ?depth . # exactly
          once - will be bound to the depth of
          the visited vertices.
      gas:program gas:out2 ?predecessor . #
          exactly once - will be bound to the
          predecessor.
      gas:program gas:maxIterations 4 . #
          optional limit on breadth first
          expansion.
      gas:program gas:maxVisited 2000 . #
          optional limit on the #of visited
          vertices.
  }
  ?predecessor ?linkType ?out . # figure out
      what link type(s) connect a vertex with
      a predecessor
}
limit 100
```

As per above example the actual "Breadth First Search" (BFS) [25] implementation resides in `com.bigdata.rdf.graph.analytics.BFS` Java class. The parameter names for BFS are in, out, out1, out2, maxIterations and maxVisited. These parameters are passed to the algorithm from the query.

In summary Bigdata approach is to implement algorithms into the query engine itself and allow SPARQL interpreter invoke those algorithms. They have already implemented several algorithms (including SSSP). More details about algorithms can be found in [7] . Also Bigdata claims, introducing a new algorithm is fairly straightforward task.

One of the main advantages of this approach is the ability to have more control over the graph. Algorithm code optimization will also be easy since the algorithm code is written using Java.

4. DISCUSSION

SPARQL is the defacto query language designed to query RDF graphs. Our focus is more towards to a general purpose graph DSL. But SPARQL is bound to RDF structure. Mainly it is bound to the concept of a triple (Subject, Predicate and Object). Further almost all the SPARQL implementations uses pattern matching on triples to extract results. Therefore this methodology may not perfectly suit to bi-directional graphs and undirected graphs (Simply because, formalizing triple pattern will not be straightforward for those graphs).

Also writing algorithms using SPARQL is inherently complicated; due to the nature of a query language. As per section 3.3 one good approach is to build algorithm support into SPARQL query engine and let user invoke those algorithms, rather than writing those himself. But then to introduce new algorithms user has to write Java/C++ code and also he/she needs to learn about the framework provided by the query engine. This nature kind of limits what we are trying to achieve by a graph DSL.

On the other hand Green-Marl is a well designed Domain Specific Language for graph processing. It hides details about complex programming structures from user and attempts to achieve maximum thread level parallelism. GM has identified necessary programming language constructs (data structures, traversal mechanisms) for graph processing. Further memory consistency models used in sequential regions and parallel regions assures safe operation of the application. Operator based reductions and BSP support is quite useful and provides synchronization of data structures when needed. Green-Marl is also capable of generating high performance code. How ever we experienced following limitations/drawbacks when using Green-Marl.

1. Green-Marl provides thread level parallelism using OpenMP. But to process large scale graphs (scale 28 and over) we need to distribute processing. In other words we need process level parallelism to process large scale graphs. For this GM compiler needs to generate code that is compatible MPI [10] like framework. As per current implementation GM does not support this.

2. GM provides few data structures (set, sequence for edges and nodes and map, collection for others). When implementing certain algorithms we need data structure support significantly. An example is “Dijkstras Single Source Shortest Path (SSSP) Algorithm“ [22]. For Dijkstra‘s SSSP we need to have a priority queue. GM does not provide an in-built data structure to implement a priority queue. So we need to use existing data structure in GM and custom implement a priority queue. As an example we could use a Map type together with a set. i.e. `i.e. Map<int, Set >PriorityQueue`. But such implementation will not perform like a standard priority queue implementations.

3. During Green-Marl code compilation it does type checking of parameters. But Green-Marl does not generate code which is always safe. Consider following example; Let‘s say we have following procedure;

   Listing 34: Procedure definition. n and m are from 2 different graphs.

   ```
   Proc f(G1, G2: Graph, n : Node(G1), m:
       Node(G2)) {
   }
   ```

   Green-Marl compiler make sure n is a node of graph G1 and m is a node of graph G2. But generated C++ code for above code would look like follows;

   Listing 35: Generated code for procedure definition in 34.

   ```
   void f(gm_graph& G1, gm_graph& G2,
       node_t& n, node_t& m)
   {
       //Initializations
       gm_rt_initialize();
       G1.freeze();
       G2.freeze();
   }
   ```

   As per generated code both “n“ and “m“ are “node_t“ type. Further generated code does not have a way to enforce n is a node of G1 and m is a node of G2 (For example if we used C++ metaprogramming we could enforce such rules). Therefore in the application one can call “f“ as follows (mistakenly)
   `f(G1, G2, n, n);`
   In above call, n is a node of G1 but we are allowed to pass as a node of G2 also (as both node types in generated code is “node_t“). So such erroneous code is not detected by generated code.

4. In GM the graph must always be immutable. Therefore GM does not support streaming or dynamic graphs.

5. ACKNOWLEDGMENTS

This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0521433.

6. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[2] C. Avery. Giraph: Large-scale graph processing infrastruction on hadoop. *Proceedings of Hadoop Summit. Santa Clara, USA:[sn]*, 2011.

[3] D. Beckett and T. Berners-Lee. Rdf primer, turtle version, 2009.

[4] L. Bergstrom. Measuring numa effects with the stream benchmark. *arXiv preprint arXiv:1103.3225*, 2011.

[5] Bigdata. Bigdata, 2014. [Online; accessed 3-August-2014].

[6] Bigdata. Gather apply scatter, 2014. [Online; accessed 3-August-2014].

[7] bigdata. Rdf gas api, 2014. [Online; accessed 3-August-2014].

[8] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In *IFIP Congress (1)*, pages 509–514. Citeseer, 1994.

[9] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.

[10] W. D. Gropp, A. Lumsdaine, S. Huss-Lederman, B. Nitzberg, and E. Lusk. *MPI*. The MIT Press, 1998.

[11] S. Harris and A. Seaborne. Sparql 1.1 query language. *W3C working draft*, 14, 2010.

[12] M. D. Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, 1998.

[13] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 349–362. ACM, 2012.

[14] S. Hong and M. Sevenich. The green-marl language specification, 2014. [Online; accessed 1-August-2014].

[15] A. Jena. Apache jena, 2013.

[16] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[17] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[18] C. Nvidia. Programming guide, 2008.

[19] C. OpenMP. C++ application program interface, 2002.

[20] E. PrudâĂŹHommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

[21] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[22] S. Skiena. Dijkstra's algorithm. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pages 225–227, 1990.

[23] L. SYSTAP. Rdf, graphs, graph databases, graph analytics on gpus, 2014. [Online; accessed 3-August-2014].

[24] I. University. Big red ii at indiana university, 2014. [Online; accessed 3-August-2014].

[25] Wikipedia. Breadth-first search — wikipedia, the free encyclopedia, 2014. [Online; accessed 3-August-2014].

[26] Wikipedia. Graph (mathematics) — wikipedia, the free encyclopedia, 2014. [Online; accessed 1-August-2014].

[27] Wikipedia. Iterative method — wikipedia, the free encyclopedia, 2014. [Online; accessed 3-August-2014].

[28] Wikipedia. Reduction (complexity) — wikipedia, the free encyclopedia, 2014. [Online; accessed 3-August-2014].

[29] Wikipedia. Shortest path problem — wikipedia, the free encyclopedia, 2014. [Online; accessed 1-August-2014].

[30] YarcData. Yarcdata, 2014. [Online; accessed 3-August-2014].