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

Sungpack Hong
Pervasive Parallelism Laboratory
Stanford University
hongsup@stanford.edu

Hassan Chafi
Pervaisve Parallelism Laboratory
Stanford University
and Oracle Labs
hassan.chafi@oracle.com

Eric Sedlar
Oracle Labs
eric.sedlar@oracle.com

Kunle Olukotun
Pervasive Parallelism Laboratory
Stanford University
kunle@stanford.edu

Abstract

The increasing importance of graph-data based applications is fueling the need for highly efficient and parallel implementations of graph analysis software. In this paper we describe Green-Marl, a domain-specific language (DSL) whose high level language constructs allow developers to describe their graph analysis algorithms intuitively, but expose the data-level parallelism inherent in the algorithms. We also present our Green-Marl compiler which translates high-level algorithmic description written in Green-Marl into an efficient C++ implementation by exploiting this exposed data-level parallelism. Furthermore, our Green-Marl compiler applies a set of optimizations that take advantage of the high-level semantic knowledge encoded in the Green-Marl DSL. We demonstrate that graph analysis algorithms can be written very intuitively with Green-Marl through some examples, and our experimental results show that the compiler-generated implementation out of such descriptions performs as well as or better than highly-tuned hand-coded implementations.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

General Terms Algorithms, Design, Performance

Keywords Graph, Domain-Specific Language, Parallel Programming

1. Introduction

A graph is a fundamental data structure that captures relationships between different data entities. Graphs are used to represent data sets in a wide range of application domains, such as social science, astronomy and computational biology. In a social graph, for example, nodes correspond to people while friendship relationships between them are represented as edges. In addition, nodes or edges in a graph are typically associated with a certain set of values. For example, the edges of a social graph might be associated with the average number of phone calls per month between two people. Graph analysis involves extracting information from a given data-set which is represented as a graph. For example, one might be interested in finding groups of people who call each other frequently.

The enormous growth in data leads to large underlying graphs which require huge amounts of computational power to analyze. While modern commodity computer systems provide a significant amount of computational power measured in the tera-flops, efficient execution of graph analysis algorithms on large datasets remains difficult on these systems [26]. Some of the challenges include:

- Capacity – for very large datasets, the graph will not fit into a single physical memory address space.
- Performance – Some graph algorithms often perform poorly when applied to large graph instances.
- Implementation – it is not easy to develop a correct and efficient implementation for many graph algorithms.

In this paper, we tackle the performance and implementation challenges, and focus on the case when the graph fits into physical memory. This is practical– as recent work [4] has demonstrated that fairly large-sized graph problems can be processed in the physical memory of a modern high-end server machine. In Section 5, however, we do discuss some future work to deal with the capacity challenge.

Even in a single memory address space, graph applications often suffer from poor performance on large problems. Poor performance is due to the random memory access behavior of most graph analysis algorithms – as soon as the working-set size exceeds the size of the various levels of caches in the system, performance becomes dominated by memory latency.

One common approach to address the performance challenge is to exploit the data parallelism which is typically abundant in analysis algorithms on large graphs. This approach leverages both the increasing number of parallel implementations proposed for common graph-theory algorithms, and the recent advances and proliferation of parallel computing systems By properly utilizing this hardware, we hide this memory latency problem by using parallelism and are only limited by the available memory bandwidth in the system.

However, adopting parallelism exacerbates the third challenge – implementation. Even without parallelism, it is often challenging to implement a graph analysis algorithm in an efficient way. The application developer has to carefully consider which data structure

ASPLOS'12 March 3–7,2012, London, England, UK.

to use and has to reason about the algorithm's memory access patterns. Parallel programming introduces a whole set of other issues to be carefully considered – race-conditions, dead-lock, etc.

Even the same algorithm may show drastically different performance depending on its implementation. For instance, many different parallel implementations [4, 21, 38] have been proposed for a simple breadth-first search algorithm, all with different performance results. As we will show in Section 1.1, achieving an efficient implementation requires both a clear understanding of the graph algorithm and a deep knowledge of the underlying hardware architecture. This also introduces a tight coupling to the underlying architecture which decreases the portability of the resulting implementation.

In this paper, we take a different approach on how to tackle the performance and implementation challenges we have identified. We propose Green-Marl[1], a domain-specific language (DSL) designed specifically for graph analysis algorithms. Users of Green-Marl can describe their graph algorithm intuitively using high-level graph constructs which expose the inherent parallelism in the algorithm. A compiler for Green-Marl can exploit this high-level information by applying a series of high-level optimizations and parallelizing the algorithm, and finally producing a (correct) optimized parallel implementation of the given algorithm. By using the DSL, the users can concentrate on their algorithm rather than its implementation. The Green-Marl compiler final output is an implementation written in a general-purpose language, e.g. C++, rather than a machine language, e.g. x86 instructions (Section 3).

Our specific contributions are as follows:

- Green-Marl, a DSL in which a user can describe a graph analysis algorithm in a very intuitive way. This DSL captures the high-level semantics of the algorithm as well as its inherent parallelism.
- The Green-Marl compiler which applies a set of optimizations and parallelization enabled by the high-level semantic information of the DSL and produces an optimized parallel implementation targetted at commodity SMP machines.
- An interdiscipliary DSL approach to solving computational problems that combines graph theory, compilers, parallel programming and computer architecture.

The rest of this paper is organized as follows: we present the overall language design of Green-Marl in Section 2. In Section 3, we describe the Green-Marl compiler which produces high-performing parallel implementations of algorithms described in Green-Marl. Our experimental results (Section 4) show that while graph analysis algorithms can be written using Green-Marl in a simple way, their compiler-generated implementation performs equally as well as or better than a highly-tuned hand-coded library implementations. In Section 5, we discuss how Green-Marl can be used with minimal disruption in existing development environments, and how we plan to solve the capacity issue using Green-Marl. Section 6 highlights related work and we conclude in Section 7.

1.1 A Motivating Graph Example

Before discussing the details of the Green-Marl language and compiler, we present a well known social network analysis algorithm, "betweenness centrality" (BC), written in Green-Marl. BC measures the centrality (or relative importance) of nodes in a given graph. BC is widely used in social network analysis. Brandes [13] first proposed a fast sequential algorithm to compute BC values for all nodes in a graph.

[1] Green-Marl is a transliteration of Korean words meaning 'pictured language'.

```
1  Procedure Compute_BC(
2   G: Graph, BC: Node_Prop<Float>(G)) {
3    G.BC = 0;        // initialize BC
4    Foreach(s: G.Nodes) {
5     // define temporary properties
6      Node_Prop<Float>(G) Sigma;
7      Node_Prop<Float>(G) Delta;
8      s.Sigma = 1; // Initialize Sigma for root
9     // Traverse graph in BFS-order from s
10     InBFS(v: G.Nodes From s)(v!=s) {
11       // sum over BFS-parents
12       v.Sigma = Sum(w: v.UpNbrs) {w.Sigma};
13     }
14    // Traverse graph in reverse BFS-order
15     InRBFS(v!=s) {
16       // sum over BFS-children
17       v.Delta = Sum (w:v.DownNbrs) {
18          v.Sigma / w.Sigma * (1+ w.Delta)
19       };
20       v.BC += v.Delta @s; //accumulate BC
21  } } }
```

Figure 1. Betweenness Centrality algorithm described in Green-Marl

Although the original BC computation algorithm was written for sequential execution, the algorithm contains an abundant amount of inherent parallelism. Bader and Madurri [7] leveraged this fact and presented an initial parallel implementation. A few years later, the same researchers presented a significantly improved implementation of the same algorithm [28]; the implementation adopted different meta-data structures, used a better iteration scheme, and eliminated lock contention. This resulted in more than 2x speedup over their previous parallel implementation when measured on a Cray XMT machine.

Figure 1 shows the same algorithm written in Green-Marl. The procedure takes two arguments `G`, a graph, and `BC`, a node property (a piece of data associated with each node) of graph `G`, which is to be computed (line 2). At line 3, the `BC` value is initialized to 0 for all nodes in the graph `G`. Line 4 begins an iteration over every node `s` in graph `G`. At each iteration step, two temporary node properties, `Sigma` and `Delta`, are defined. After initializing `Sigma` for node `s` (line 8), we do a breadth-first (BFS) order iteration over the nodes in graph `G`, where `s` is the root of the search (line 10). During the BFS iteration over every node `v` other than `s`, `Sigma` of `v` is computed by summing up `Sigma` values of its BFS-parents (line 12). When the BFS iteration concludes, at line 15 we then perform a reverse order BFS traversal (i.e. we start iterating at the farthest nodes from `s`). During this traversal we compute `Delta` from the BFS-children nodes for each node `v` (line 17). Finally, `Delta` is accumulated into `BC` during every iteration of s (line 20).

Note that the Green-Marl implementation is much shorter than Brandes' original description, which included implementation details related to performing the BFS iteration using queues and lists. Bader and Madduri's implementation [7], on the other hand, available in a library [9] is more than 400 lines long. Note that the library is implemented using OpenMP [31] which is a concise way of writing parallel code.

Although written in an intuitive way, the Green-Marl implementation fully exposes the parallelism inherent in the algorithm. Initialization at Line 3 is a trivial data-parallel operation, while BFS and RBFS (line 10 and line 15 can be parallelized at each level of the iteration. Finally, even the outer loop 4 can be parallelized. The compiler's analysis phase (Section 3) ensures that there are no data accesses that conflict (otherwise the compiler emits an error or a warning). The Green-Marl compiler exploits the available parallelism to generate an efficient implementation in a general-purpose language (e.g. C++). In Section 4, we will show that this Green-Marl implementation performs as well as the hand-optimized implementation [28].

2. Green-Marl Language Design

2.1 Scope of the Language

Mathematically, a graph is an ordered pair $G = (N, E)$ comprising a set, N, of nodes and E, a set of edges or optionally ordered pairs of two nodes. The data associated with each node or edge can be defined as a mapping P from N (or E) to some codomain (e.g. $P_{phone} : E \rightarrow \mathbb{R}$). In this paper, we refer to such a mapping as a *node (or edge) property*.

Given a graph, $G = (V, E)$, and a set of properties defined on the graph, $\Pi = \{P_1, P_2, ...P_n\}$, our language is specifically designed for the following types of graph analysis:

- Computing a scalar value from (G, Π), e.g. the conductance of a sub-graph
- Computing a new property P_{n+1} from (G, Π), e.g. the pagerank of each node of a graph
- Selecting a subgraph of interest out of the original graph $(V', E') \subset (V, E)$, e.g. strongly connected components of a graph

Note that the last type of analysis can be also be formulated as computing two new properties $P_{node} : N \rightarrow \{true, false\}$ and $P_{edge} : E \rightarrow \{true, false\}$ which captures the membership of the original nodes and edges in the resulting subgraph.

The above mathematical descriptions imply two important assumptions that Green-Marl makes:

1. The graph is immutable and is not modified during analysis.
2. There are no aliases between graph instances nor between graph properties.

We assume an immutable graph so that we can focus on the task of graph analysis, rather than worry about orthogonal issues such as how graphs are constructed or modified. Since Green-Marl is designed to be used in re-writting only parts of the user application (Section 3.1), one can construct or modify the graph in their own preferred way (e.g. from data file, from a database, etc.) but when a Green-Marl generated implementation is handed a graph, the assumption is that the graph will not be modified while a Green-Marl procedure is analyizing it.

2.2 Parallelism in Green-Marl

The Green-Marl language design is based on a few paradigms. First, it includes language constructs for *implicit parallelism*, e.g. group assignment (line 3 in figure 1) and in-place reductions (line 12). Second, it allows users to explicitly demarcate parallel execution regions. For example, the `foreach` statement used in line 4 specifies a parallel region. The compiler analysis enabled by the domain-specific knowledge encoded in Green-Marl applications can detect possible conflicts in parallel regions. Finally, the domain-specific data analyses increase the likelihood that the compiler can apply speculative or automatic parallelization. For example, while the `for` statement, in contrast to a `foreach` statement, specifies sequential execution of the iteration steps in no particular order, the compiler may parallelize the iteration as long as the parallelization can guarantee a serializable execution of iteration steps. [2]

The language adopts a fork-join style of parallelism, where iteration steps in a `foreach` are forked and execute in parallel and then are synchronized at a join point which is inserted right after the `foreach`. Note that this is the same widely used mechanism adopted in successful parallel frameworks such as OpenMP [31], CUDA [30], and Pregel [29]. Green-Marl also allows the users to express nested parallelism. The following example[3] shows nested parallel regions: line 23 is a parallel iteration over all the nodes in the graph `G`, while line 26 is a nested parallel iteration over all the neighbors of node s. Forked iteration steps are joined at the end of a parallel iteration. All the forked iteration steps of line 26 from a single `s` have to synchronize before execution proceeds to line 27. However, forked iteration steps do not need to synchronize with other nested parallel execution regions (i.e., those iteration steps which are forked from a different `s`).

```
22  Int sum=0;
23  Foreach(s: G.Nodes) {
24    Int p_sum = u.A;
25    Foreach(t: s.Nbrs)
26       p_sum *= t.B;
27    sum += p_sum;
28  }
29  Int y = sum / 2;
```

Green-Marl uses static scoping rules. Local variables are private to the current iteration step, but are shared by any parallel region in the same scope as the variables. In our previous example, `p_sum` is private to each parallel iteration step of the iteration at line 23 (*s-iteration*) but shared by every parallel iteration step at line 26 (*t-iteration*) originated from the same `s`.

Finally, Green-Marl's memory consistency model for parallel execution is similar to that of OpenMP's:

1. a write to a shared variable is not guaranteed to be visible to other concurrent iteration steps during the parallel execution.
2. a write to a shared variable is guaranteed to be visible to the later statements of the current iteration step, unless another write to the same variable from a concurrent iteration step becomes visible beforehand.
3. a write to a shared variable becomes visible at the end of a parallel iteration; if there have been multiple concurrent iteration steps that wrote to the same, only one write (chosen non-deterministically).

On the other hand, Green-Marl ensures each write is atomic (i.e. no data is partially written) and operations on a collection (i.e. adding an item to a set) are also atomic.

Thus, the following Green-Marl example suffers from data-races under the above consistency model. Line 32 is a write-write conflict because multiple s-iteration steps can write to the same t-node. Similarly the read at line 33 and the write at line 32 is a read-write conflict. We encourage the readers to visualize these cases using Figure 2.(a). A Green-Marl compiler would detect such data-races at compile time (Section 3.2).

```
30  Foreach(s:G.Nodes)
31    Foreach(t:s.OutNbrs)
32      t.A =          // write-write conflict
33         t.A + s.B;  // read-write conflict
```

Fundamentally, Green-Marl prevents users from writing algorithms that communicate between concurrent iteration steps. Instead, it expects users to use proper *reductions* (Section 2.3.4) to get deterministic results. This design decision was made to allow the language to work in distributed environments as well (Section 5.2).

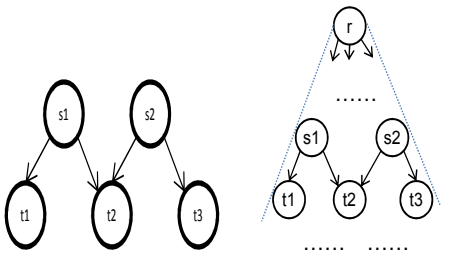
2.3 Language Constructs

2.3.1 Data-Types and Collections

Green-Marl has a simple type system. First, there are five primitive types (`Bool`, `Int`, `Long`, `Float` and `Double`). Green-Marl also

[2] Our current Green-Marl compiler (Section 3) does not attempt to parallelize `for` iterations; however, future implementations may safely parallelize them via conventional schemes such as fine-grained locking, graph coloring, or transactional memory.

[3] In our examples, our convention is to use 'G' for graphs, 'x,y,z' for scalar variables, 'A,B,C' for propertes, 's,t,u' for loop variables, 'F(),G()' for Boolean functions, and 'X(),Y()' for numeric functions.



(a) (b)

Figure 2. Simple graph instances. (a) is a small bipartite graph, while (b) is showing only a portion of the graph in the middle of BFS traversal rooted from r.

		sequential			parallel		
Group	Op-Name	S	O	Q	S	O	Q
Grow	Add	v			v		
	Push(Front/Back)		v	v		v	v
Shrink	Remove	v			v		
	Pop(Front/Back)		v	v		?	?
	Clear	v	v	v	v	v	v
Lookup	Has	v	v	v	v	v	v
	Front(Back)		v	v		v	v
	Size	v	v	v	v	v	v
Copy	=	v	v	v	X	X	X
Iteration	Items	v	v	v	v	v	v

Modification under iteration → Shrink, Grow, or Copy: X

Conflicts under parallel execution →

Grow-Shrink: X Lookup-Shrink: ? Lookup-Grow: ?

Table 1. Operations on Collections: S,O,and Q denotes set, order and sequence, respectively. In the table, 'v', 'X', '?' stands for the operation being valid, invalid, and undefined for the selected collection type under the selected execution context.

defines two graph types (`DGraph` and `UGraph`) which denote a directed graph and an undirected graph respectively[4]. Second, there is a node type and an edge type both of which are always bound to a graph instance, as in the `n1` and `n2` in following code example. Third, there are node properties and edge properties which are bound to a graph but have base-types as well (`A` in the following code example).

Finally, Green-Marl provides three types of collection types (for both node and edge types): Set, Order, and Sequence. Elements in a Set are *unique* while a Set is *unordered*. Elements in an Order are *unique* while an Order is *ordered*. Elements in a Sequence are *not unique* while a Sequence is *ordered*. All of these type are also bound to a graph: e.g., `S` in the following code example.

```
34  Procedure foo(G1, G2:Graph, n:Node(G1)) {
35     Node(G2) n2; // a node of graph G2
36     n2 = n;  // type-error (bound to different graphs)
37     Node_Prop<Int>(G1) A; //integer node property for G1
38     n.A = 0;
39     Node_Set(G1) S; // a node set of G1
40     S.Add(n);
41  }
```

Table 1 summarizes the operations defined on collection types in a sequential and parallel execution context. Here are a few highlights: (a) The semantics of assignment to a collection is to create a copy of the collection. During parallel execution, assignment to a shared collection variable is not allowed. (b) Parallel `Push` to an `Order` is allowed – the relative order of pushes in a parallel context is non-deterministic and the pushed elements may not be visible to other iteration steps. Support for parallel `Pop` is dependent on each compiler implementation. (c) Every collection can be iterated on, in sequential or in parallel. When iterating in parallel on a Sequence or an Order, however, ordering information is lost. (d) It is prohibited to modify the collection while iterating over it. (e) During parallel execution, a collection should either grow or shrink, but cannot do both.

2.3.2 Iterations and Traversals

Iteration in Green-Marl has the following form:

`Foreach` (*iterator*`:`*source*`(-).`*range*) (*filter*)
body_statement

The keyword `Foreach` implies parallel execution; it can be replaced with `For` which implies sequential execution. *iterator* defines an iterator symbol for this iteration, while *source* and *range* determines over what is being iterated on and how. *filter* is an optional boolean expression, which dictates whether to apply *body_statement* on the current iteration step or not. The following table summarizes the possible iteration range defined on each source type.[5]

Source Type	Range	Access
`D/UGraph`	`Nodes`	Linear
`Node(D/UGraph)`	`Nbrs`	Random
`Node(DGraph)`	`OutNbrs`	Random
`Node(DGraph)`	`InNbrs`	Random
`Node(D/UGraph)`	`UpNbrs`	Random/-1
`Node(D/UGraph)`	`DownNbrs`	Random/+1
`Node_Set`	`Items`	Linear
`Node_Order`	`Items`	Linear
`Node_Seq`	`Items`	Random

One can iterate on all the nodes in a graph (`Nodes`), or all the items in a collection (`Items`). From a node, one can iterate on its neighborhood nodes in several different ways. `InNbrs` and `OutNbrs` are defined for directed graphs using the edge directions – neighbors that are connected by incoming/outgoing edges. For undirected graphs `InNbrs` and `OutNbrs` become synonym to `Nbrs`. `UpNbrs` and `DownNbrs` and are defined only during a BFS from a specific node: `UpNbrs` of a node are `InNbrs` of the node whose hop-distance from the root is smaller by one. See Figure 2.(b) to visualize `UpNbrs` and `DownNbrs`.

The *access* column in the previous table indicates the nature of the iteration. *Linear* iteration means that every iterator points to a unique item, i.e. each item will be accessed once and only once via the iterator. On the other hand, *Random* indicates the possibility of aliasing; Random/+-1 will be discussed when we discuss BFS traversal.

Sequential(i.e. `For`) iteration on an `Order` or a `Set` preserves the order in which items were appended to the collection. Reverse order iteration is also possible as shown in the following example:

```
42  Node_Order(G) O; ...
43  For(o: O-.Items) // reverse order iteration on O
44     ...
```

Green-Marl also provides two graph traversal schemes – Breadth-First Search (BFS) order traversal and Depth-First Search (DFS) order traversal. The syntax takes the following form:

`InBFS` (*iter*`:`*src*`^.Nodes From` *root*) `[`*navigator*`]` (*filter1*)
forward_body_statement
`InRBFS` (*filter2*)
backward_body_statement

[4] `Graph` is a type alias for `DGraph`. Both `DGraph` and `UGraph` can be multi-graphs.

[5] For the sake of brevity, the table only shows iteration types for node-wise iteration. Similar iteration types are also defined for edge-wise iteration.

The above syntactic form is similar to that of normal iteration with a few differences. First, *root* defines the root node of the BFS traversal. Second, the optional ^ character means that we first create a transposed version of the graph and then traverse it along the transposed edges. Third, *navigator* is another optional boolean expression that dictates which nodes will be pruned for traversal. For example, if a node does not satisfy the navigator condition, the node is not further expanded during traversal. On the other hand, a node that satisfies the navigator condition is still expanded, even if it does not satisfy the filter condition.

The following code example specifies a BFS traversal on the transposed graph of G, traversing only through nodes whose flag have not been set.

```
45  Node_Prop<Bool>(G) flag;
46  InBFS(s: G^.Nodes From r)[!s.flag]
47  {...}
```

Also, note that the BFS syntax has two body statement blocks. The first body statement block is executed while in forward BFS expansion (i.e. traversing nodes from closest to farthest from the root). If the optional second sentence block is specified, the reverse order BFS traversal (i.e. from the farthest to the closest nodes to the root) is also performed.

DFS has the same syntactic form as BFS except that `InDFS` and `InPost` are used in place of `InBFS` and `InRBFS`; the first body statement block specifies the statements to be executed while in pre-order execution while the second is for post-order execution.

DFS and BFS have different parallel execution semantic. DFS implies sequential execution, while BFS implies level-synchronous parallel execution. That is to say, during BFS, all the nodes that have the same distance from the root node are visited concurrently but parallel execution contexts are synchronized before moving on to the next level. Therefore there are no data conflicts in the following code example, as nodes accessed via `s` are disjoint with repect to the nodes accessed via `t`.

```
48  InBFS(s: G.Nodes From r) {
49    Foreach(t: s.UpNbrs)      // t: Random/-1 access
50      s.A += t.A;  // s.A does not conflict with t.A
51  }
```

2.3.3 Deferred Assignment

Green-Marl also supports bulk synchronous consistency [34] via deferred assignments. Deferred assignments are denoted by <= and followed by a binding symbol as in the following example. When a symbol is written using a deferred assignment, a read from the symbol always gives an 'old' value and a write to the symbol becomes effective at the end of the binding iteration.

```
52  Foreach(s:G.Nodes) {
53      // no conflict. t.X gives 'old' value
54      s.X <= Sum(t:s.Nbrs) {t.X} @ s
55  }
56  // All the writes to X becomes visible simultaneously
57  // at the end of the s iteration.
```

2.3.4 Reductions

Green-Marl heavily relies on reductions to achieve deterministic results despite its non-sequential memory consistency model (Section 2.2). There are two (slightly) different kinds of reductions: one assumes an expression form (or *in-place* from), the other an assignment form. In the following example, `Sum` is in-place reduction and `+=` is reduction assignment. Note that initialization is required for the reduction assignment; in the case of `G` being an empty graph, x becomes zero, but `y` would have become a non-deterministic value without proper initialization.

```
58  Int x,y;
59  x = Sum(t:G.Nodes){t.A}; // equivalent to next 3 lines.
60  y = 0;
61  Foreach(t:G.Nodes)
62    y+= t.A;
```

The following lists all the reduction operators in Green-Marl; we list both in-place form and assignment form[6]. Note that in-place reductions can have filters just like the `foreach` statement.

In-place	Assignment	In-place	Assignment
All	&&=	Sum	+=
Any	||=	Product	*=
Min	min=	Count	++
Max	max=		

Min and Max are especially interesting forms of reduction because they can be accompanied with *argmax* and *argmin* assignment. In the following example, line 67 stores not only the minimum value of the expression `t.A + u.b` but also the arguments minimizing the expression. Note that all three LHS symbols will be written atomically.

```
63  Int X=INF;
64  Node(G) from, to;
65  Foreach(t:G.Nodes)
66      Foreach(u:t.Nbrs)
67        X <from, to> min= (t.A + u.B) <t, u>;
```

Similarly to deferred assignments, reduction assignments can be accompanied with a bound symbol, denoted as an iteration variable followed by `@` character, which indicates the iteration scope where the reduction happens. This syntax is designed to clarify the user's intention in the presence of nested iterations. For example, x+=..@s means that variable x is reduced by addition during the s-iteration and therefore x should not be read or written otherwise inside the s-iteration. The concept is not far from OpenMP's reduction pragma.

However, specifying a bound symbol is optional for reduction assignments, as in the following example; if omitted, a Green-Marl compiler should try finding appropriate bound for the user, or give an error if it can't.

```
68  Int sum = 0;
69  Foreach(s:G.Nodes) {
70      Foreach(t:s.Nbrs)
71         sum += t.A @s; // accumulate over s-iteration
72      if (s.A > THRESHOLD)
73         sum += s.B ; // @s is implied
74      s.C = sum; // this is an read-reduce conflict.
75  }              // (sum is still being reduced)
```

Although this section introduced key features of the Green-Marl language, we are in the process of adding formalism in our language specification. The current draft of Green-Marl language specification is publicly available on our website. [1].

3. Compiler

3.1 Compiler Overview

Figure 3 shows how Green-Marl fits in the overall application development process. We envision that the user application is composed of graph analysis components and also other components like data acquisition and UI. We expect that the application developer would extract the graph analysis components and use Green-Marl to express them. The Green-Marl compiler is used to generate an equivalent implementation of the graph analysis component in a lower level general purpose language such as C++. The generated code assumes certain data structures for graph representation. The definition and implementation of these required data structures is

[6] We're investigating adding custom reduction operators in the next version of the language.

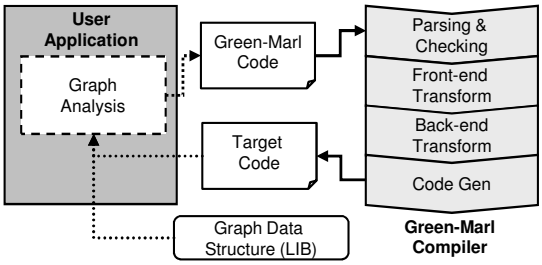


Figure 3. Overview of Green-Marl DSL-compiler Usage

supplied as a library to be linked into the final executable. Therefore each Green-Marl compiler should specify a one-to-one mapping between Green-Marl types and target language types.

The figure also shows the four phases of Green-Marl compilation. In the first phase, the compiler conducts syntactic checks, runs a type-checker, and ensures that the application developer is not violating the semantics of Green-Marl's parallel constructs. If all checks pass successfully, the compiler applies a set of transformations that are target independent. These are followed by target dependent transformations and optimizations. The fourth phase handles the task of code generation. In this paper, we discuss our first implementation of a Green-Marl compiler which targets coherent shared-memory multi-processor systems and emits C++ code. Note that, however, the first two phases can be re-used for our future compiler implementations that will target completely different systems (Section 5.2). Each of the four different phases will be discussed in more detail in the following sections:

3.2 Parsing and Checking

The first phase of compilation involves checking the validity of the user input. The compiler checks for three things: (1) syntax, (2) types, and (3) valid parallel semantics.

Syntax checking and type checking in Green-Marl is no different than what is found in traditional general-purpose compilers. Green-Marl is a statically typed language; the type of each expression is determined at compile time. The type checking phase of a Green-Marl compiler is much simpler than that of a general purpose language,since Green-marl is composed of a handful of primitive and built-in types.[7]

While Green-Marl's type system is quite rudimentary, the high-level semantics it encodes enable powerful program analysis, which can be used to check for incorrect use of parallel constructs. Let us consider the following example Green-Marl code snippet:

```
76  Int y=0;
77  Foreach(s:G.Nodes)(s.C>3){
78    Foreach(t:s.Nbrs){
79      Int x = y * s.B;
80      s.A += t.B * X @ t;
81    }
82    s.B = 4;
83  }
```

Table 2 shows how the above code can be analyzed. The table should be read from top to bottom, where each row represents a step in the semantics analysis phase. The analysis happens through post-order iteration of the abstract syntax tree; thus, we first analyze the body of a `foreach` iteration before finishing the analysis of the `foreach` itself. First, the analyzer identifies at line 76 that scalar symbol `y` is being written. Then the analysis continues two levels deeper and reaches line 79, where it detects a read of `y`, a read of `B` driven by `s`, and a write to `x`. Similar analysis happens at line 80, where `x` is being read, `B` is being read driven-by `t`, but this time symbol `A` is being reduced with a plus, driven-by `s`, and bound to `t`.

[7] User defined types are supported via foreign syntax (Section 5.1).

Line	type	target symbol	driver/ access.	cond?	reduce op	binding symbol
76	W	y	-	N	-	-
79	R	y	-	N	-	-
	R	B	s	N	-	-
	W	x	-	N	-	-
80	R	x	-	N	-	-
	R	B	t	N	-	-
	D	A	s	N	+	t
78	R	y	-	N	-	-
(during	R	B	s	N	-	-
inspect-	R	B	t	N	-	-
ion)	D	A	s	N	+	t
78	R	y	Rand	N	-	-
(after	R	B	s	N	-	-
abst-	R	B	Rand	N	-	-
raction)	W	A	s	N	-	-
82	W	B	s	N	-	-
77	R	y	Rand	Y	-	-
(after	R	B	Linear	Y	-	-
abst-	R	B	Rand	Y	-	-
raction)	R	C	Linear	N	-	-
	W	A	Linear	Y	-	-
	W	B	Linear	Y	-	-

Table 2. Read-Write Analysis Example

When all the statements of the *foreach(t)* have been analyzed, the parallel semantics of this loop are analyzed. First, all the read, write, and reduce sets of body block are merged while references to the local scope variables (e.g. `x`) are eliminated. The compiler checks (1) if the reduced symbol (i.e. `A`) is not being written, read, or reduced by other operations. Then, (2) it compares every symbol in the merged read-set against the write-set, and a check is made for conflicting writes. In this example, there are no errors.

If the data conflict checks do not generate errors, the result of current iteration analysis is computed in the following way: (a) reduction bound to the current iterator is changed into a write and (b) access through the current iterator is abstracted according to the iterator's type. In the example in table 2, the reduction to symbol `A` is changed to a write, and the read of `B` driven by t is now labeled as a *Random* read of `B`, as `t` is a neighborhood iterator.

Now, the analysis goes up one level and reaches line 82, where it sees the write to `B`, driven by `s`. Then, parallel semantic checking can be performed for the iteration at line 77. This time the analysis will find a conflict between the write to `B` driven by `s` (line 82 and the Random read of `B` (line 80). The current default action for a Read-Write conflict is to warn the users – In contrast, Read-Reduce, Write-Reduce, or Reduce-Reduce conflicts cause compilation to terminate in error. When no errors have been detected, the data-access of the s-iteration can be computed, as in the last row of table 2. Note how the filter at line 77 adds a conditional flag.

This analysis proceeds until all the statements in a procedure are analyzed. The parallel semantic analysis is not only used to flag errors, but its results are also used during the code transformation phase as it will become clear in the following section.

3.3 Architecture Independent Optimizations

Once a Green-Marl application successfully emerges from the type-checking and data-access analysis phases, the compiler can apply to it a set of transformations that are target independent.

In this phase, the compiler first transforms all the syntactic sugar into explicit iterations (*Group Assignments* and *In-place Reduction*). Then it applies optimizations that are effective regardless of the target architecture. (*Loop Fusion*, *Hoisting Definitions*, *Reduc-*

tion Bound Relaxation). In addition, the compiler might use additional knowledge on the domain-specific properties of the data-set, possibly controlled by command-line options (*Flipping Edges*).

Group Assignment: Group assignment can be trivially expanded into a parallel or sequential iteration, depending on the type of source collection.

```
Node_Set S(G);
Node_Seq Q(G); //may not be unique
S.A = S.A + S.size();
Q.A = Q.A + Q.size();
```

becomes

```
Foreach(s:S.Items) // parallel iteration
  s.A = s.A + S.Size();
For(q:Q.Items) // sequential iteration
  q.A = q.A + Q.size();
```

In-place Reduction: *In-place Reductions* are expanded into loop and reduction assignments.

```
Int y = Sum(s:G.Nodes){
          Product(t:s.Nbrs)(f(t)){X(t,s)}};
```

becomes

```
Int y;
Int _s0 = 0;
Foreach(s:G.Nodes) {
  Int _p1 = 1;
  Foreach(t: s.Nbrs)(f(t))
      _p1 *= X(t,s) @ u;
  _s0 += _p1 @ s;
}
y = _s0;
```

Loop Fusion: In the following example, two loops `s` and `t` are fused into one even though the two loops have dependencies (through property `A` and `B`), since the access pattern is *Linear*. Fusing loops in general reduces loop overhead and increases locality. Note that procedures that are written with implicit parallel constructs allow for many opportunities to apply loop fusion.

```
Foreach(s: G.Nodes)(f(s))
   s.A = X(s.B);
Foreach(t: G.Nodes)(g(t))
   t.B = Y(t.A)
```

becomes

```
Foreach(s: G.Nodes)(
   if (f(s)) s.A = X(s.B);
   if (g(s)) s.B = Y(s.A);
}
```

Hoisting Definitions: Temporary property definitions can be moved out of sequential loops, which can save repeated large memory allocations and de-allocations in some target systems.

```
For(s:G.Nodes) { //sequential loop
    Node_Prop<Int>(G) A;
    ...
}
```

becomes

```
Node_Prop<Int>(G) A;
For(s:G.Nodes) {
    ...
}
```

Reduction Bound Relaxation: Reduction bounds can be relaxed to the *outmost parallel iteration* that comes after the target symbol is defined. If there is no such iteration, the reduction is transformed back to a normal assignment. Note that in general reductions are more expensive than normal reads and writes.

```
int x = 1;
Foreach(s:G.Nodes) { // par loop
 int y = 0;
 For(t: s.Nbrs) { // seq loop
   x*= s.A @ s;
   y+= s.B + t.C @ t;
} }
```

becomes

```
int x = 1;
Foreach(s:G.Nodes) { // par loop
 int y = 0;
 For(t: s.Nbrs) { // seqloop
   x*= s.A @ s;
   y= y + s.B + t.C; // normal assignment
} }
```

Flipping Edges: Reductions that are applied with reverse edges can be replaced with ones that use forward edges as in the following example. (See Figure 2.(a) to visualize this optimization.) Currently, the compiler only applies this optimization when use of reverse-edges is discouraged via a command-line options; use of reverse edge is often disabled because reverse edges may not be available from the original graph data but would need to be generated via an extra computation step.

```
Foreach(t:G.Nodes)(f(t))
  Foreach(s:t.InNbrs)(g(s))
     t.A += s.B;
```

becomes

```
Foreach(s:G.Nodes)(g(s))
  Foreach(t:s.OutNbrs)(f(t))
     t.A += s.B;
```

3.4 Architecture Dependent Optimizations

In this phase, the compiler applies optimizations based on further available information. It utilizes specific knowledge about the target system (*Selection of Parallel Regions*) as well as about the target language(*deferred assignment* and *Saving BFS Children*). It also takes advantage of the performance characteristics of the underlying data-structures that implement Green-Marl built-in types (*Set-Graph Loop Fusion*).

In addition, architecture independent optimizations (e.g. *Relaxing Reduction Bounds*, and *Hoisting Definitions*) are re-applied since new opportunities for those optimizations can be uncovered as a result of other optimizations (e.g. *Selection of Parallel Regions*).

Set-Graph Loop Fusion: Using the uniqueness property of a set, set iteration can be fused with linear graph iteration. This optimzation is only enabled when the back-end library ensures that the Has() operation is fast (e.g. O(1)).

```
Node_Set S(G); // ...
Foreach(s: S.Items)
   s.A = x(s.B);
Foreach(t: G.Nodes)(g(t))
   t.B = y(t.A)
```

becomes

```
Foreach(s: G.Nodes)(
   if (S.Has(s)) s.A = x(s.B);
   if (g(s)) s.B = y(s.A);
}
```

Selection of Parallel Regions: The compiler determines which parallel iterations are actually going to be parallelized. Currently, the compiler flattens all the nested parallelism but selects only the inner-most graph-wide foreach iteration or BFS traversal. This decision is based on the assumption that the graph instance is large enough to consume all the processor and memory bandwidth of the given system.

```
Foreach(s:G.Nodes) {
  InBFS(x:G.Nodes)
    doX(x);
  Foreach(y:s.Nbrs)
    doY(y);
  Foreach(z:G.Nodes)
    doZ(z);
}
```

becomes

```
For(s:G.Nodes) {   // Seq
  InBFS(x:G.Nodes) // Par
    doX(x);
  For(y:s.Nbrs)    // Seq
    doY(y);
  Foreach(z:G.Nodes) // Par
    doZ(z);
}
```

Deferred Assignment: Deferred assignments are transformed into the definition of temporary properties, initializing them and copying back the final result.

```
Foreach(s:G.Nodes)(f(s))
  s.A = Sum(t:s.Nbrs){t.A}
```

becomes

```
Node_Prop<...>(G) A_new;   // define temp
G.A_new = G.A;             // init temp
Foreach(s:G.Nodes)(f(s))
  s.A_new = Sum(t:s.Nbrs){t.A}  // write to temp
G.A = G.A_new;             // copy back temp
```

However, initialization can be removed if the property is linearly and unconditionally updated inside the binding iteration.

```
Foreach(s:G.Nodes)
  s.A = Sum(t:s.Nbrs){t.A}
```

becomes

```
Node_Prop<...>(G) A_new;   // define temp
Foreach(s:G.Nodes) // linear & unconditonal
  s.A_new = Sum(t:s.Nbrs){t.A}  // write to temp
G.A = G.A_new;             // copy back temp
```

Furthermore, if the binding iteration is inside a sequential loop, the copy back operation can be replaced with pointer swaps, while a final copy back operation is required at the iteration exit. Note that the compiler can generate aliases, whereas users cannot.

```
While (...) {
  // ...
  Foreach(s:G.Nodes)(f(s))
    s.A = Sum(t:s.Nbrs){t.A}
  // ...
}
```

becomes

```
Node_Prop<..>(G) A_new;
// following syntax is for explanation only
Prop* ptr_saved = _alias_ptr(A);
While(...) {
  // ...
  Foreach(s:G.Nodes) {
    s.A_new = Sum(t:s.Nbrs){t.A}
  }
  _swap_ptr(A, A_new);
  // ...
}
If (ptr_saved != A) {
  _swap_ptr(A, A_new);
  A = A_new; // copy back before return
}
```

Saving BFS Children: Our compiler also applies a technique used in Madduri et al.'s work [28]. This optimization checks if the down neighbors are used during the reverse order traversal, in which case the compiler prepares an O(E) array named edge-marker. During forward BFS iteration, if a neighbor is identified in the next level, the edge leading to it is marked. During reverse iteration, next-level neighbors can be identified quickly by looking at the edge information rather than the nodes.

```
InBFS(v:G.Nodes; s) { ... //forward }
InRBFS { // reverse-order traverse
  Foreach(t: v.DownNbrs) { //using DownNbrs
    DO_THING(t);
} }
```

becomes

```
// before BFS
_prepare_edge_marker(); // O(E) array
{     // inside code template of
  ... // forward BFS iteration
  for (e = edges ... ) {
    index_t t = ...node(e);
     ... // normal BFS expansion here
    // [added: mark-down nbrs]
    if (isNextLevel(t)) {
        edge_marker[e] = 1;
}  }  } // end of forward BFS
{ ...  // reverse BFS
 // iterate Down_Nbrs
  for (e = edges ..) {
   // check on edge instead of node
   if (edge_marker[e] ==1) {
     index_t t= ...node(e);
     DO_THING(t);
} }}
```

3.5 Code Generation

In this phase, the compiler emits out target code using code-generation templates that make use of back-end libraries. Currently, we use OpenMP [31] as our threading library. Thus, during code generation, most of the parallel iterations are trivially translated into `#pragma omp parallel for` headers. We also assume gcc as our target compiler.

The graph is represented using the same data format used in a publicly available parallel graph library [9]. The format is essentially equivalent to the CSR (Compressed Sparse Row) format used in sparse matrix computation. A `Set` is implemented using both a bitmap and a vector while an `Order` is implemented using a bitmap and a list. Finally, node and edge properties are implemented as an array of length O(N) and O(E) respectively.

Graph and Neighborhood Iteration: Thus a typical neighborhood expansion iteration is translated into as following form:

```
Foreach(s:G.Nodes)
  For(t: s.Nbrs)
    s.A = s.A + t.B;
```

becomes

```
OMP(parallel for)
for(index_t s = 0; s < G.numNodes(); s++) {
  // iterate over node's edges
  for(index_t t_=G.edge_idx[s]:t_<G.edge_idx[s+1];t_++){
    // get node from the edge
    index_t t = G.node_idx[t];
    A[s] = A[s] + B[t];
} }
```

Efficient DFS and BFS traversals and **Small BFS Instance Optimization**: code generation for DFS and BFS traversals is done using efficient code-generation templates. For BFS traversal we use an efficient parallel implementation [22] as our base template with the additional optimization of delaying initialization of expensive runtime structures (e.g. an O(N) array) until the BFS traversal grows wide enough.

Note that there have been many publications on efficient parallel implementation of BFS traversal [4, 6, 22] and more are to come. Green-Marl users, however, can benefit from any future BFS implementation without modifying their program, as the compiler can simply generate target source code using new code generation templates.

Reduction on Properties: Reductions involving node properties are translated using an atomic compare and swap.

```
Foreach(s:G.Nodes)
  For(t: s.Nbrs)
    t.A += s.B @s;
```

becomes

```
OMP(parallel for)
for( s = ... )
  for( t_ = ... ) {
    t = ...
  // The follwing 4 lines are implemented with C-MACRO
    {int __old;
     int __new;
     do { __old = A[t];
          __new = __old + B[s] ;
     } while (CAS(&t[A]), __old, __new);
} }
```

Reduction on Scalars: Reductions on scalar values use privatization. In other words, the value is first reduced into a thread-local variable and a reduction is done at the end. Although this is similar to OpenMP's reduction clause, OpenMP for C does not support reduction by minimum or maximum.

```
int x = ...;
Foreach(s:G.Nodes)
  x min= s.B @s;
```

becomes

```
int x = ...;
OMP(parallel)
{  // create thread local
  int _x_prv = INT_MAX;
  OMP(for)
  for( s = ... )
    _x_prv = MIN(_x_prv, B[s]);
 // C-macro equivalent to the above example.
  REDUCE_MIN(int, x, _x_prv)
}
```

3.6 Discussion

Before we move on, let us discuss the benefit of using a compiler to apply the optimizations that have been presented in the previous sections. Each of the optimization techniques is not completely novel in itself; it is either an application of a classic compiler optimization (e.g. *Loop Fusion*) or a technique that has been discussed in previous work (e.g. *Saving BFS Children*).

However, using a compiler allows optimization without requiring significant effort from a graph algorithms expert, and allows optimizations that are difficult with fixed function libraries (those that don't participate in compilation). For example, in applying *Saving BFS Children*, the compiler takes a look at the statements that are executed during reverse BFS traversal, and inserts extra lines of codes in forward traversal only if `DownNbrs` are referred to during reverse traversal. Finally, some optimizations are difficult to apply in a low-level language compiler without domain knowledge. For example, it is challenging for a C++ compiler to merge loops in the following example, which is a possible C++ implementation of the Green-Marl source code (line 103 – 106) in the *Loop Fusion* example; C++ compiler cannot ascertain data dependencies between line 264 and line 266 without information about the uniqueness of the loop index.

```
Map<Node, int> A,B;
List<Node>& Nodes = G.getNodes();
List<Node>::iterator t,s;
for(s=Nodes.begin();s!=Nodes.end();s++)
  if (f(*s)) A[*s] = X(B[*s]);
for(t=Nodes.begin();t!=Nodes.end();t++)
  if (g(*t)) B[*t] = Y(A[*t]);
```

4. Experiments

In this section, we demonstrate the productivity benefits of using Green-Marl, and the efficiency of the target-specific implementations it generates. Table 3 lists the popular graph algorithms we used to evaluate Green-Marl. The first three of these algorithms come from a parallel graph library called SNAP [9]. BC denotes the betweenness centrality computation algorithm as explained in section 1.1. Conductance [12] evaluates a single value from a graph partition, by counting edges between nodes in a given partition and nodes in other partitions; the algorithm is frequently used to detect community structures in social graphs. Vertex Cover [17] is a well known approximation to the NP-hard minimum vertex cover problem. Note that each of the above three algorithms belongs to a different type of graph analysis task defined in section 2.1.

In order to exercise more language constructs in Green-Marl, we also used two famous algorithms that are not included in the SNAP library. PageRank [32] is a famous algorithm which evaluates the importance of each node in a graph based on (out-)degrees of its in-coming neighbors. The algorithm can be described naturally with the bulk-synchronous consistency model. Kosaraju's algorithm [17] is one way to find strongly connected components in a directed graph. The algorithm performs two DFS traversals on the graph; one traversal using the original edges and a second one using the reverse edges. Therefore the algorithm is naively sequential. However, the second DFS traversal can be replaced with a BFS traversal that can be parallelized. Green-Marl's syntax makes it easy to switch between DFS and BFS traversals.

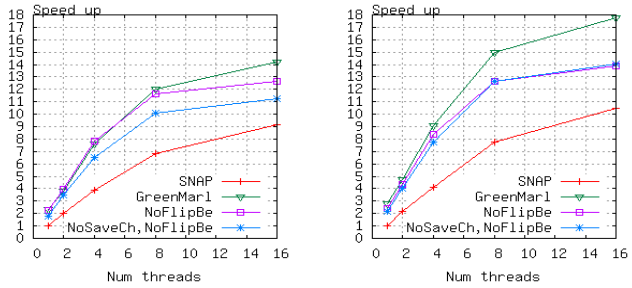
We first consider the productivity gains in using Green-Marl. Table 3 compares the number of lines of code of the above algorithms when they are implemented in both in Green-Marl as well a general-purpose language. For a fair comparison, we counted only the relevant lines of code: we did not count lines of code responsible for data generation, time measurement, or ifdef statements. We also treated every block of comments as a single line.

Overall, Green-Marl enables a much more concise implementation than what can be achieved in a general purpose language. Note that SNAP already uses OpenMP [31], which significantly reduces the lines of code needed to parallelize programs. However, the Green-Marl implementation was much shorter. The extreme example is the BC computation, which is more than 300 lines long as implemented in the SNAP library whereas the Green-Marl implementation is only 24 lines (Figure 1). The main reason for this reduction in LOC is that the SNAP implementation cannot make use of a BFS library call, because its execution was tightly coupled with the BFS iteration code. On the other hand, Green-Marl allows a concise implementation of the algorithm, while the compiler to apply optimizations which ultimately yield better performance than the SNAP version.

Of course, fewer lines of code does not necessarily mean higher productivity. However, we believe that the Green-Marl implementations, in general, are more concise and intuitive than those written in a general-purpose language. For example, the Green-Marl description of Conductance computation (Figure 9) reflects the mathematical definition of Conductance more closely than the SNAP library's implementation. We show the Green-Marl implementations of the algorithms in Figure 9 so the reader can judge for themselves.

Name	LOC Original	LOC Green-Marl	Source
BC	350	24	[9] (C OpenMp)
Conductance	42	10	[9] (C OpenMp)
Vetex Cover	71	25	[9] (C OpenMp)
PageRank	58	15	[2] (C++, sequential)
SCC(Kosaraju)	80	15	[3] (Java, sequential)

Table 3. Graph algorithms used in the experiments and their Lines-of-Code(LOC) when implemented in Green-Marl and in a general purpose language.



(a) RMAT (b) Uniform

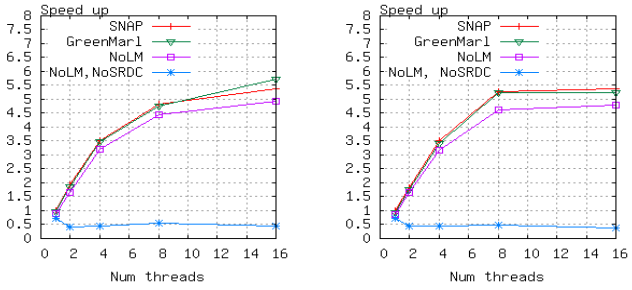
Figure 4. Speed-up of Betweenness Centrality. Speed-up is over the SNAP library [9] version running on a single-thread. NoFlipBE and NoSaveCh means disabling the *Flipping Edges* (Section 3.3) and *Saving BFS Children* (Section 3.5) optimizations respectively.

Next, we measure the performance of the target-specific compiler-generated Green-Marl implementations. For these experiments, we used two kinds of widely-accepted synthetic graph generators [9]: Uniform and RMAT. Uniform generates a random graph based on a simple model where two nodes are randomly selected and edges are inserted between them. On the other hand, RMAT [15] generates a scale-free graph which has a power-law skewed degree distribution. Unless stated otherwise, we used graphs with 32 million nodes and 256 million edges which were generated using default parameters [9]. All the performance was measured on a commodity server-class machine, which has two sockets with an Intel Xeon X5550, each with 4 cores and 2 hardware threads per core. The total last-level-cache size was 16MB.

Figure 4 compares the performance of the Betweenness Centrality computation implemented in Green-Marl to the hand-optimized version included in the SNAP library. The Green-Marl implementation (denoted as GreenMarl in the plot) performs far better than the current SNAP library implementation – the improvement is up to 2.5 times when using 4 cores with the Uniform graph data set. The performance gap is diminished with higher thread counts as the memory bandwidth gets saturated.

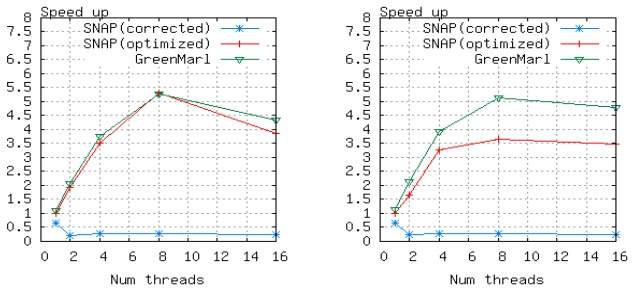
Note that Madduri et al. proposed a more optimized implementation [28], with a reported speed-up of 2.3 over the current SNAP version. However, this implementation was developed specifically for the Cray XMT machine, and thus is not usable on commodity systems. Also note that the Green-Marl compiler applies all of the optimizations discussed in the paper automatically. Furthermore, these optimizations can be automatically applied to other algorithms. This is one of the main benefits of using a DSL; we are able to leverage the insights developed in optimizing a specific application and apply them to a whole class of similar algorithms.

To show the impact of the optimizations, the figure also shows the performance of the Green-Marl implementation when some optimizations are disabled. The NoFlipBe and NoSaveCh curves show the performance of this algorithm when the *Flipping Edges* (Section 3.3) and the *Saving BFS Children* (Section 3.5) optimizations, respectively, are not applied. Figure 4 shows that most of the



(a) RMAT (b) Uniform

Figure 5. Speed-up of Conductance. Speed-up is over the SNAP library [9] version running on a single-thread. NoLM and NoSRDC means disabling the *Loop Fusion* (Section 3.3) and *Reduction on Scalars* (Section 3.5) optimizations, respectively.



(a) RMAT (b) Uniform

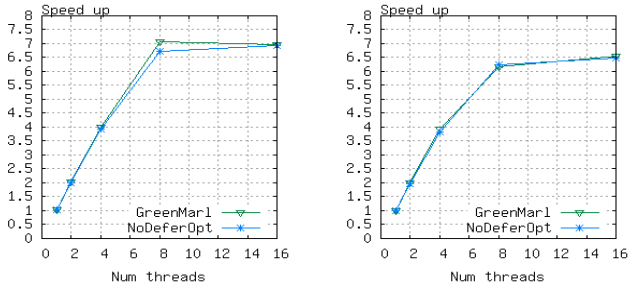
Figure 6. Speed-up of Vertex Cover implemented in Green-Marl and two versions of the corrected SNAP implementation SNAP which had a data-race. The first version, SNAP(correct) utilizes a simple locking approach. The second version, SNAP(optimized), uses a more advanced test and test-and-set scheme. A small instance (100k nodes, 800k edges) was used in this experiment.

parallel speed-up is attained from using an optimized parallel BFS iteration (Section 3.5). The two optimizations mentioned above do however have a measurable contribution to overall performance.

Figure 5 shows the performance of the conductance algorithm implemented in Green-Marl compared to the implementation included in the SNAP library. In this experiment, we randomly partitioned the nodes of each graph into four sets where each set contains 10, 20, 30, and 40% of the nodes. We measured the time to compute the conductance of all these partitions in turn. The figure shows that the Green-Marl implementation performs as well as the hand-tuned SNAP library. Furthermore, we can see that the *Reduction on Scalars* optimization is critical to achieving parallel performance. Without the *Loop Fusion* optimization, we witness some performance loss due to additional synchronization overhead.

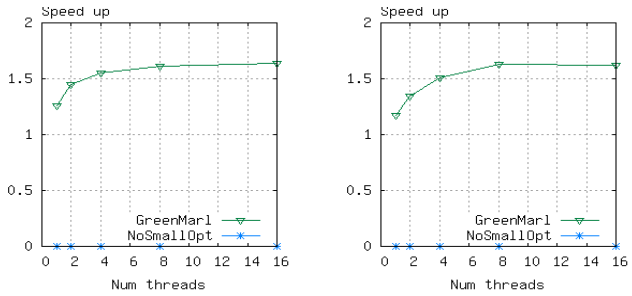
Figure 6 shows the performance result of the vertex cover algorithm. Note that the original implementation in the current version of the SNAP library (ver 0.4) is not correct – a critical region was not properly protected. One simple way to fix this issue is to protect the critical region with a simple `omp critical` pragma. However such a fix completely serializes the execution (SNAP(corrected) in the figure). While a more advanced scheme (test and test-and-set) can improve the performance of this implementation (SNAP(optimized) in the figure), Green-Marl still outperforms the more advanced version due by applying additional optimizations such as *Reduction on Scalars*(Section 3.5).

Figure 7 shows the performance results of the PageRank algorithm implemented in Green-Marl. The Green-Marl compiler successfully parallelized the original sequential algorithm. In the fig-



(a) RMAT (b) Uniform

Figure 7. Speed-up of PageRank implemented in Green-Marl. Speed-up is over a single-threaded implementation of the original algorithm [2]. NoDeferOpt means disabling the *Deferred Assignment* (Section 3.4) optimization. These results are based on running the implementation for 10 iterations.



(a) RMAT (b) Uniform

Figure 8. Speed-up of Kosaraju implemented in Green-Marl. The Speed-up is measured over a single-threaded C++ implementation which performs two DFS iterations. NoSmallOpt means disabling our *Small BFS Instance Optimization*. A small instance (1M nodes, 8M edges) was used in this experiment.

ure, the curve denoted as NoDeferOpt shows the resulting performance when disabling the *Deferred Assignment* (Section 3.4) optimization. Although this optimization eliminates the initialization and copy-back of the temporary array variable, its impact on overall performance was not significant; the performance was governed by the random memory accesses during the neighborhood expansion phase of the algorithm.

Figure 8 shows the performance result of Kosaraju's algorithm implemented in Green-Marl. Note that while the original algorithm involves two DFS (i.e. sequential) traversals on the graph, only the second DFS traversal can be replaced with a BFS (i.e. parallel) traversal. Therefore, as a result of Amdahl's law, the theoretical speed-up limit is 2. The NoBFSOpt curve shows the impact of our small instance optimization which is an improvement over a recently proposed BFS scheme [4] for small sub-graphs. Real-world graphs which are mimicked by our synthetic graphs tend to be composed of a small number of large (i.e. O(N)) components and a large number of small (i.e. O(1)) components. Thus, our optimization ensures that the overhead of the BFS traversal is minimized when iterating on small sub-graphs.

5. Smoothing the Way to Green-Marl Adoption

So far, we have shown the benefits of using a DSL to achieve optimum efficiency with high productivity. However requiring users to adopt a new programming language introduces a new set of challenges. In this section, we show how most of these concerns can be addressed.

5.1 Programmer Productivity

Green-Marl doesn't require the application developer to re-write their whole application. They can isolate graph-analysis routines in their application and only rewrite these using Green-Marl. Our compiler will then generate an efficient implementation of the algorithms in the target source code (e.g. C++). The generated implementation can then be compiled along with the rest of the application with minimal changes to other modules. Any extra effort required to convert between the data types in the original application and those expected by Green-Marl would also be required when using any other graph library.

Green-Marl also allows the embedding of foreign data types and statements. In the example that follows, procedure `foo` accepts an argument with a foreign type (`my_type`). During the type-checking phase, the compiler simply treats all foreign types as equivalent (line 269)[8] Foreign statements are also easily embedded in Green-Marl through a square bracket expression (line 270).

Foreign statements are conceptually similar to the inline assembler (i.e. `asm`) in gcc. During the code generation phase, the compiler copies the text enclosed in square bracket as is, other than properly handling variables with DSL types (denoted by the $ sign inside the back-ticks). For example, *'$a.F'* would be translated to *F[a]* (in our current C++ back-end compiler). At the end of a foreign statement, the application developer may supply a *list of mutated variables*. This allows the Green-Marl compiler to correctly identify read after write (RAW) hazards and prevents it from moving any statements that read the variable beyond the point at which it will be modified. It is also up to the application developer to correctly handle data races inside a foreign function if it is called during parallel execution.

```
267  Procedure foo(G:Graph, my_var:$my_type) {
268     Int x = 0;
269     $my_type2 var2 = my_var;       \\ compiles okay
270     [$my_var->mutate($x)]::[x]; } \\ text replacement
```

Since the Green-Marl compiler performs source-to-source translation, the application developer will still end up with code generated in a more widely-used target language such as C++. This significantly reduces the risk associated with adopting a new language. The generated source from the current Green-Marl compiler is fairly human-readable: variable names and code layouts are preserved to a reasonable degree. We also plan to preserve comment blocks in future versions of our compiler.

While one could always have manually typed in the exact same source code that was emitted by the Green-Marl compiler, the DSL approach provides additional benefits, as was shown in the previous sections. First, more concise and intuitive descriptions of graph algorithms, and secondly, a set of optimizations during translation that span multiple function calls. These optimizations across library entry points are nearly impossible to achieve using a general purpose compiler that lacks any higher level semantic knowledge of the application (Section 3 and 4).

Furthermore, users of Green-Marl may rely on the robust debugging tools of the target language. In general, using Green-Marl leads to an expression of the desired graph algorithms that makes it easy to reason about the algorithm itself, leading to fewer errors. We also plan on implementing an interpreter for Green-Marl applications that will feature step-wise code execution and a visual graph representation.

5.2 Architecture Portability

Although not the main focus of this paper, a domain-specific approach such as that adopted by Green-Marl, in fact, can greatly

[8] The behavior can be changed to treat each type distinctly, via command-line options.

improve the portability of graph analysis applications. By replacing the back-end module of the compiler without modifying the DSL source code, the user can obtain equivalent implementations tailored for systems substantially different from one another, for example GPUs or clusters rather than symmetric shared-memory machines. Each back-end would use different code-templates for the language constructs and different optimizations in order to produce high-performance code for the target platforms. In contrast, the low-level implementation required to achieve performance on one system (e.g. C++ for SMP), rarely achieves similar performance and often cannot be made to work on other systems (e.g. GPU or cluster) without significant code restructuring, such adding CUDA or MPI constructs. A Green-Marl compiler back-end for GPU systems is being developed; this version adopts recent techniques in GPU graph processing [21] which leverage the massive amount of thread-level parallelism and large memory bandwidth available on GPUs.

Green-Marl will ultimately be applicable to the analysis of very large graph instances which cannot fit in a single physical memory space. Currently, the application developer is being encouraged to use libraries such as Pregel [29], a distributed graph processing framework, which consists of a MapReduce-like API that abstracts the details of data communication in the distributed system. However, the Pregel framework also forces the user to restructure their traditional graph algorithms in terms of this API. This can yield non-intuitive expressions of such graph algorithms. We refer the readers to the original paper of Malewicz et al. [29] which shows how the PageRank algorithm is implemented with Pregel APIs. By contrast, in Figure 9 we show the same algorithm written in Green-Marl. The Green-Marl implementation closely resembles the natural way the algorithm was explained in the original PageRank publication [32]. Hence, we are investigating the possibility of adding another back-end that would translate Figure 9 into the Pregel implementation. All of this is possible because considerations for CUDA and distributed environments have influenced the Green-Marl language design. Features like relaxed-memory consistency, support for full bulk-synchronous memory consistency, rich reduction operators and a limited selection of built-in data-types are examples of such considerations.

6. Related Work

DSLs fall into two broad categories, namely external DSLs which are completely independent languages, and internal DSLs, which borrow functionality from a host language [23]. While Green-Marl is currently implemented as an external DSL, we will consider in the future the viability of implementing it as an internal DSL. Indeed there has been a lot of research into how to leverage domain-specific knowledge in the service of more optimized execution. Expression Templates [35] can produce customized generation, and are used by Blitz++ [36]. Active libraries [37], which are libraries that participate in compilation were introduced by Veldhuizen. Kennedy coined the term telescoping languages [24] for efficient DSLs created from annotated component libraries. TaskGraph [11] is a meta-programming library that supports run-time code generation in C++. Delite [14] is a more recent proposal that aims at simplifying the creation of performance oriented DSLs such as Green-Marl.

Noticeably, many features of Green-Marl are reminiscent of those found in OpenMP [31], as both programming models let the user explicitly specify parallel regions of execution. However, Green-Marl provides further performance benefits by using domain specific knowledges in applying optimizations. Previous work has also shown some of these benefits. Guyver et al. present significant performance improvements by annotating library methods with domain-specific knowledge [20], and CodeBoost [10] uses user-defined rules to transform programs using domain knowledge.

There are several publicly available libraries for graph analysis. Popular single-threaded libraries include the Boost Graph Library(BGL) [33] and igraph [18]. There are only a few libraries that support parallel or distributed execution: Parallel BGL [19] is a distributed version of BGL while SNAP [8] is a stand-alone parallel graph analysis package. GraphLab [25] is a framework for machine learning type graph algorithms. We are considering to use GraphLab as one of our future back-ends. No matter how wide the range of fixed functions supported by such libraries, the user still may need to implement a different algorithm not supplied by the library. Green-Marl allows users to implement their own algorithms and still get generated code that performs as well as hand-tuned low-level implementations. Note that Green-Marl can still make use of the efficient data structures supplied by a graph library such as SNAP [8].

The efficient implementation of parallel graph algorithms is a challenging task. The best implementation often is closely coupled to an underlying hardware architecture. For example, a simple BFS traversal has been implemented differently on commodity servers [4, 22], GPUs [21], Cray XMT machines [6] and cluster environments [38]. Techniques for efficient implementation of other algorithms such as betweenness centrality [28], shortest path [27], and minimum spanning tree [16] can be found in the research literature. Green-Marl does not obviate the need for these studies; rather these techniques can easily be reused and applied to other algorithms with similar patterns by means of the Green-Marl DSL and compiler.

Large graph instances are drawing more and more attention from the high performance computing community. Traditional HPC technologies such as vectorization do not provide satisfactory performance in processing large graph instances. Graph500 [5] is an effort to create a benchmark that captures the computational requirements of large graph applications. Pregel [29] is a MapReduce like framework that aims to bring distributed processing to graph algorithms. In Section 5.2, we discussed our plans for targetting such a framework in future versions of Green-Marl.

7. Conclusions and Future Work

The Green-Marl DSL approach to graph analysis demonstrates that developers can generate an implementation via an intuitive high-level language that has competitive performance with hand-written native code. Green-Marl allows users to get the benefit of optimizations specifically for graph data structures, without writing those optimizations directly, and without tying those optimizations to architecture-specific characteristics. The language approach provides benefits over a library approach by optimizing across the call boundaries of currently available graph libraries. Green-Marl can be linked with native C++ programs, allowing graph algorithms written in Green-Marl to be embedded in more complex, general-purpose software. The next steps for Green-Marl are to generate code for alternative architectures, such as clusters and GPUs, to provide more support for the architecture-independent benefits of this approach.

Acknowledgements

We thank Dr. Peter B. Kessler and Dr. Mario Wolczko (Oracle Labs) for reviewing this paper and making helpful suggestions. This work is supported by Army contract AHPCRC W911NF-07-2-0027-1; DARPA contract, Oracle order US1032821; DOE contract, Sandia order 1134180; SEEC: Specialized Extremely Efficient Computing, Contract HR0011-11-C-0007; Stanford PPL affiliates program, Pervasive Parallelism Lab: Oracle, NVIDIA, AMD, NEC, and Intel.

References

[1] Green-marl lanaguage specification. `http://ppl.stanford.edu/main/green_marl.html`.

[2] Pagerank c++ implementation. http://code.grnet.gr/projects/pagerank.

[3] Strongly connected component (kosaraju) java implementation. http://www.keithschwarz.com/interesting/.

[4] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable Graph Exploration on Multicore Processors. In *ACM/IEEE SC 2010*.

[5] M. Anderson. Better benchmarking for supercomputers. *Spectrum, IEEE*, 48(1), 2011.

[6] D. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *ICPP 2006*. IEEE.

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

[8] 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.

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

[10] O. Bagge, K. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*.

[11] O. Beckmann, A. Houghton, M. Mellor, and P. H. Kelly. Runtime code generation in c++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004.

[12] B. Bollobás. *Modern graph theory*. Springer Verlag, 1998.

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

[14] H. Chafi, A. Sujeeth, K. Brown, H. Lee, A. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*. ACM, 2011.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[16] G. Cong and D. Bader. Lock-free parallel algorithms: An experimental study. *High Performance Computing-HiPC 2004*, 2005.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press and McGraw-Hill, 2001.

[18] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal Complex Systems*, 1695, 2006.

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

[20] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*.

[21] S. Hong, S. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *PPoPP*, 2011.

[22] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration for multi-core cpu and gpu. In *IEEE PACT 2011*.

[23] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[24] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3), 2005.

[25] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[26] A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, and J. Guest Editors. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[27] K. Madduri, D. Bader, J. Berry, and J. Crobak. Parallel shortest path algorithms for solving large-scale instances. *9th DIMACS Implementation Challenge-Shortest Paths*, 2006.

[28] 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*.

[29] 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.

[30] NVIDIA. CUDA. `http://developer.nvidia.com/object/cuda.html`.

[31] OpenMP ARB. Openmp. `http://www.openmp.org`.

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

[33] J. Siek, L. Lee, A. Lumsdaine, L. Lee, L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, M. Heroux, et al. The boost graph library: user guide and reference manual. 2002.

[34] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33, August 1990.

[35] T. L. Veldhuizen. Expression templates, C++ gems. SIGS Publications, Inc., New York, NY, 1996.

[36] T. L. Veldhuizen. Arrays in blitz++. In *ISCOPE*, pages 223–230, 1998.

[37] T. L. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Indiana University Computer Science, May 2004.

[38] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *SC 2005 ACM/IEEE*.

```
//------------------------------------------------------------------------------------------------------------
// Computing Conductance
//------------------------------------------------------------------------------------------------------------
Procedure conductance(G: Graph, member: N_P<Int>(G), num: Int) : Double {
  Int Din, Dout, Cross;
  Din  = Sum(u:G.Nodes)(u.member==num){u.Degree()};              // Compute degree sum of inside nodes.
  Dout = Sum(u:G.Nodes)(u.member!=num){u.Degree()};              // Compute degree sum of outside nodes.
  Cross = Sum(u:G.Nodes)(u.member==num){                         // Count number of crossing edges.
            Count(j:u.Nbrs)(j.member!=num)};                    // (Count is a syntactic sugar to Sum(..){1}
  Double m = (Din < Dout) ? Din : Dout;
  If (m ==0) Return (Cross == 0) ? 0.0 : +INF;
  Else Return (Cross / m);
}

//------------------------------------------------------------------------------------------------------------
// Obtaining vertex cover
//------------------------------------------------------------------------------------------------------------
Procedure vertex_cover(G: Graph, VC:Edge_Prop<Bool>(G)): Int {
  Node_Prop<Int>(G) Deg;
  Node_Prop<Bool>(G) covered;
  G.covered = False;                                            // Initialization
  G.Deg = G.InDegree() + G.OutDegree();                         // Deg: the sum of in-degree and out-degree
  G.VC = False;
  Int remain = G.NumEdges()*2;

  Do {
    Int maxVal = 0;
    Node(G) from, to;
    Edge(G) e;
    Foreach(s: G.Nodes)(!G.covered) {                           // Choose an edge that has maximum Deg
      Foreach(t: s.OutNbrs)                                     //   value among remaining nodes
        maxVal <from, to, e> max= (s.Deg + t.Deg) <s, t, t.GetEdge()> @s;
    }
    e.VC = True;                                                // Select this edge.
    from.covered = to.covered = True;                           // Node 'from' and 'to' are now covered.
    from.Deg = to.Deg = 0;
    remain = remain - maxVal;
  } While (remain > 0)                                          // Finish when there is no more edges

  Int C = Count(t:G.Nodes)(t.covered);                          // Count number of covered nodes
  Return C;
}

//------------------------------------------------------------------------------------------------------------
// Computing PageRank
//------------------------------------------------------------------------------------------------------------
Procedure PageRank(G: Graph, e,d: Double, max_iter: Int, PR: Node_Prop<Double>(G)) {
  Double diff =0;                                               // Initialization
  Int cnt = 0;
  Double N = G.NumNodes();
  G.PR = 1 / N;

  Do {                                                          // Main Iteration.
    diff = 0.0;
    Foreach (t: G.Nodes) {                                      // Compute PR from neighbor's current PR.
      Double val = (1-d) / N +
          d* Sum(w: t.InNbrs) (w.OutDegree()>0) {w.PR / w.OutDegree()};
      t.PR <= val @ t;                                          // Modification of PR will be visible after t-loop.
      diff += | val - t.PR |;                                   // Accumulate difference (t.PR is still old value)
    }
    cnt++;                                                      // ++ is a syntactic sugar.
  } While ((diff > e) && (cnt < max_iter));                     // Iterate for max num steps or difference is
}                                                               //   smaller than given threshold.

//------------------------------------------------------------------------------------------------------------
// Obtaining strongly connected components using Kosarajus algorithm
//------------------------------------------------------------------------------------------------------------
Procedure SCC(G: Graph, CompID: Node_Prop<Int>(G)): Int {
  G.CompID = -1;                                                // Initialization

  Node_Order(G) P;                                              // Phase 1: get reverse post-DFS order.
  For(t:G.Nodes)(!P.Has(t))  {                                  // Starting from a non-visited node,
    InDFS(s: G.Nodes From t)[!P.Has(s)] {}                      // Do DFS traversal and
    InPost {P.PushFront(s);}                                    //   store nodes in reverse post DFS order.
  }

  Int numC = 0;                                                 // Phase 2: get strongly connected components.
  For(s: P.Items)(s.compID==-1) {                               // Starting from a non-connected node,
    InBFS(t: G^.Nodes From s) [t.CompID == -1]                  // Do BFS traversal and
    { t.CompID = numC;}                                         //   add visited node into the current component.
    numC++;                                                     // End current component.
  }
  Return numC;
}
```

Figure 9. Algorithms used in the paper