Noname manuscript No.
(will be inserted by the editor)

Towards Systematic Parallelization of Graph Transformations over Pregel

Le-Duc Tung · Zhenjiang Hu

Received: date / Accepted: date

Abstract Graphs are flexible in modelling many kinds of data from traditional datasets to social networks or semi-structured datasets. To process large graphs, many systems have been proposed in which Pregel programming model is popular thanks to its scalability. Although Pregel is simple to understand and use, it is of low-level in programming and requires developers to write programs that are hard to maintain and need to be carefully optimized. On the other hand, structural recursion is a high-level tool to systematically construct efficient parallel programs on lists, arrays and trees, but it is still not scalable for graphs. In this paper, we propose an efficient parallel evaluation to the structural recursion on graphs, which is suitable for Pregel. As a result, we design and implement a high-level parallel programming framework where a domain-specific language (DSL) is provided to ease the programing burden for users, and programs written in our DSL are automatically compiled into Pregel programs that are scalable for large graphs. Experimental results show that our framework outperforms the original evaluation of structural recursion and achieves good scalability and speedup for real datasets.

Keywords Structural recursion · Graph transformation · Parallel programming

Notes

LD. Tung
SOKENDAI (The Graduate University for Advanced Studies),
Shonan Village, Hayama, Kanagawa 240-0193, Japan
E-mail: tung@nii.ac.jp

Z. Hu
SOKENDAI / National Institute of Informatics (NII),
2-1-2 Hitotsubashi, Chiyoda,
Tokyo 101-8430, Japan
E-mail: hu@nii.ac.jp

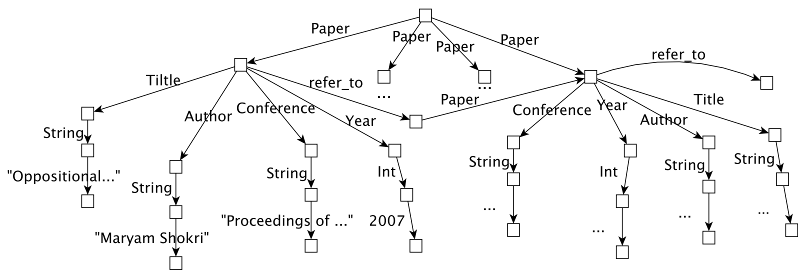

Fig. 1: A Graph of Paper Citation Network

1 Introduction

With the explosion of data on the Internet, graph has recently received much attention due to its flexibility in describing many complex data from traditional datasets to semi-structured data [2]. However, efficiently processing large graphs is challenging. At the system side, many problems need to be considered such as locality issues, failure tolerance and scalability. At the user side, it is difficult to design efficient graph algorithms due to the existence of cycles making graph algorithms easily go to an infinite loop or lead to incorrect results. Moreover, inefficient graph traversals also result in poor performance.

Many graph processing models have been proposed, in which Pregel [8] has emerged as an efficient and scalable model. It was inspired by the *Bulk-Synchronous Parallel* (BSP) model [15] whose computation consists of a sequence of *supersteps*. During a superstep, every vertex executes exactly once a common function *Vertex*.**compute**(). Inside *Vertex*.**compute**(), a vertex receives messages from the previous superstep, does its computation (i.e. updating its value, adding/removing edges and vertices), and sends messages to other vertices in the next superstep. Any changes related to vertices and edges will be available in the next superstep. Supersteps alternate with barriers that are used to synchronize communication data.

Although Pregel is scalable for large graphs, its programming model is of low-level [12]. Users need to care about many low-level things, such as when a vertex sends a message out, to whom the message will be sent, what is the type for vertices in a specific superstep, when the whole computation terminates, etc, this makes Pregel difficult to express complex computations in practice. To see this, let us consider the graph in Fig. 1, which represents a paper citation network. Assume that we would like to know citation relationships among all conferences in the graph. We expect a result of a new graph of conferences whose links denote citation relationships among these conferences. It is not so difficult to extract conference information. However, it is non-trivial to reorganize them into a new graph because, in Pregel, edges are quite independent. Moreover, we need to carefully design graph traversals to make sure that conferences are paired with right conferences.

On the other hand, structural recursion is a high-level programming model that has been shown to be powerful to build high-level parallel programming frameworks for lists, arrays and trees [9]. For graphs, structural recursion plays an important role in manipulating unstructured data; UnCAL, a powerful "calculus" for unstructured data, is based on structural recursion [3]. For instance, the "citation relationship" example can be easily written using structural recursive functions as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{eval}\ \mathit{conf}\ \textbf{where}} \\
\quad \mathit{conf}\,(\{\mathsf{Paper} : \$g\}) & = \{\mathsf{Conference} : (\mathit{cname}(\$g) \cup \mathit{cite}(\$g))\} \\
\quad \mathit{conf}\,(\{\$l : \$g\}) & = \{\} \\
\\
\quad \mathit{cname}\,(\{\mathsf{Conference} : \{\mathsf{String} : \$g\}\}) & = \$g \\
\quad \mathit{cname}\,(\{\$l : \$g\}) & = \{\} \\
\\
\quad \mathit{cite}\,(\{\mathsf{refer\_to} : \$g\}) & = \{\mathsf{Cite} : \mathit{conf}(\$g)\} \\
\quad \mathit{cite}\,(\{\$l : \$g\}) & = \{\}
\end{array}
$$

The function *conf* shows that, for each edge in the graph, if it is labeled Paper and points to a subgraph g, then we extract conference information from the graph g. For each conference, we need a union of two pieces of information: name and corresponding citations. These are returned by two recursive functions *cname* and *cite*. The function *cname* follows the path Conference.String to extract only the conference name. At the same time, the function *cite* follows the edges refer_to to find cited conferences.

The key to implementing the structural recursion in UnCAL, as described in [3], is the use of ε-edges (that work as ε-transitions of automata) for glueing computation results in a bulk computation on edges which enables parallel evaluation and makes it easier to prove termination properties. However, problems come from its scalability. Experiments for a sequential version of structural recursion only dealt with graphs up to ten thousand nodes [3]. Later, a distributed evaluation was proposed in theory, in which a performance guarantee in terms of communication complexity was given [13]. Unfortunately, too much intermediate data is generated during the evaluation, which is a serious problem in practice when processing large graphs. Moreover, for large graphs, there potentially exists a bottleneck since its reachability computation is done at a single site [14].

This paper aims to bridge the gap between structural recursion and Pregel to combine their advantages of high-level programming and practical scalibility. We show that a class of structural recursion can be efficiently mapped to Pregel. As a result, we design and implement a graph transformation framework on top of Pregel, which is actually inspired by high-level frameworks on top of MapReduce such as *Generate-Test-Aggregate* [4]. Our framework accepts graph transformations written in a domain-specific language as its input and automatically compiles these transformations into an efficient Pregel implementation.

Our main contributions in this paper are summarized as follows.

- We propose a domain-specific language (DSL) to help users write their scalable graph transformations as structural recursive functions in a declarative way. The DSL is a subset of the UnCAL language [3] but it covers a wide class of structural recursion that can be efficiently evaluated in parallel. Moreover, fusion rules can be freely applied to the composition of structural recursive functions.
- We improve the bulk semantics in UnCAL in order to efficiently evaluate structural recursive functions written in our DSL. This improvement minimizes the amount of data generated during the evaluation and utilizes the basic Pregel "skeletons" that have been proved to be efficient and scalable.
- We design and implement a high-level parallel processing framework for graphs in which transformations written in our DSL are automatically compiled into Pregel programs. Preliminary results for real-world graphs (citation network, youtube) show that our framework achieves good scalability and speedup compared to the original bulk semantics used in UnCAL.

This paper is organized as follows. Section 2 introduces basic concepts to represent and construct a graph. A domain-specific language and its internal structural recursion are given in Sec. 3. We present our efficient evaluation strategy for programs written in our DSL in Sec. 4. Section 5 shows our framework in detail and how to map our evaluation to Pregel. Experimental results are described in Sec. 6. Finally, we discuss related works in Sec. 7, and conclude our paper in Sec. 8.

2 Preliminary

Our framework is based on UnCAL (Unstructured CALculus), a powerful graph algebra for queries on graph databases [3] and graph-based model transformations [6]. In this model, graphs are up to bisimulation and constructed in a recursive way. Structural recursion on graphs is described as the one on infinite regular trees.

2.1 Graph Data Model

UnCAL's data model is a directed edge-labeled graph extended by *markers* and *ε-edges* [3]. Edge-labeled graphs are in the sense that data are stored on edges, while vertices are unique identity objects without labels. Markers are symbols to designate certain vertices as *input vertices* or *output vertices*. ε-edges are edges labeled with a special symbol ε. One could consider ε-edges as "empty" transitions and markers as initial/final states in automata.

Let *Label* be a set of labels, $\mathcal{M}$ be an infinite set of markers denoted by $\&x$, $\&y$, $\&z$, ... There is a distinguished marker $\& \in \mathcal{M}$ called a *default marker*.

Definition 1 (Graph with Markers [3]) A graph G is a quadruple (V, E, I, O), where V is a set of vertices, $E \subseteq V \times \{\mathsf{Label} \cup \varepsilon\} \times V$ is a set of edges,

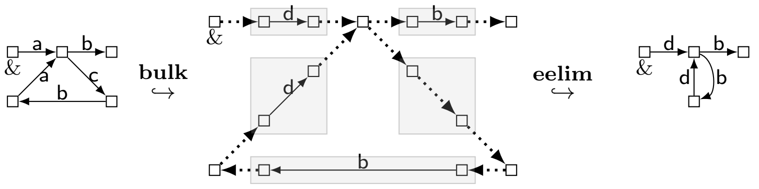

Fig. 2: Transformation **a2d_xc**: Change Edges a to d and Delete Edges c

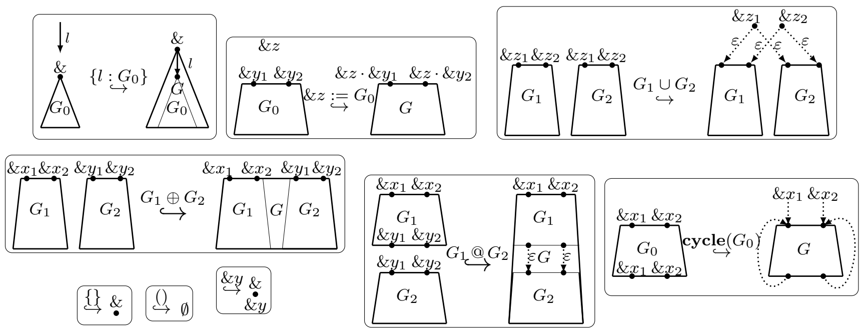

Fig. 3: Graph Constructors

$I \subseteq \mathcal{M} \times V$ is an one-to-one mapping from a set of input markers to V, and $O \subseteq V \times \mathcal{M}$ is a many-to-many mapping from V to a set of output markers.

For $\&x, \&y \in \mathcal{M}$, let $v = I(\&x)$ be the unique vertex such that $(\&x, v) \in I$, we call v an *input vertex*. If there exists a $(v, \&y) \in O$, we call v an *output vertex*. Note that there are no edges coming to input vertices or leaving from output vertices. Let $DB^{\mathcal{X}}_{\mathcal{Y}}$ denote data graphs with sets of input markers $\mathcal{X}$ and output markers $\mathcal{Y}$. When $\mathcal{X} = \{\&\}$, $DB^{\mathcal{X}}_{\mathcal{Y}}$ is abbreviated to $DB_{\mathcal{Y}}$, and $DB_{\emptyset}$ is abbreviated to DB. *A rooted graph* is the one that has only one input marker $\mathcal{X} = \{\&\}$ and no output markers $\mathcal{Y} = \emptyset$, in which the vertex $v = I(\&)$ is called the root vertex of the graph. Graphs with multiple markers are internal data structures for graph constructors.

The left graph in Fig. 2 is an example of a rooted directed edge-labeled graph in which $V = \{1, 2, 3, 4, 5\}$, $E = \{(1, a, 2), (2, b, 3), (2, c, 4), (4, b, 5), (5, a, 2)\}$, $I = \{(\&, 1)\}$, and $O = \{\}$. The vertex marked with & is the root of the graph. In this paper, we ignore vertex ids when drawing graphs.

2.2 Graph Constructors

Before looking at graph constructors in detail, we need to define an additional operation "·" to generate new markers. The operation "·" returns a different marker for every pair of $\&x$ and $\&y$. We assume "·" to be associative, $(\&x \cdot \&y) \cdot \&z = \&x \cdot (\&y \cdot \&z)$, and & to be its identity, $\& \cdot \&z = \&z \cdot \& = \&z$.

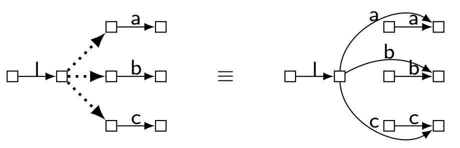

Fig. 4: Meaning of ε-Edges. The left graph with ε-edges is value equivalent to the right graph without ε-edges

There are nine graph constructors in UnCAL. From these constructors, we can build arbitrary directed edge-labeled graphs. Definitions of the constructors are given in Fig. 3 (Dotted arrows denote ε-edges). Informally, $\{\}$ constructs a graph of only one vertex labeled with a default input marker &, $\{l : G\}$ constructs a new graph G' from the graph G by adding the edge l pointing to the root of G. The source vertex of l becomes the root of G'. The operator $\cup$ unions two graphs of the same input markers with the aid of ε-edges. The next two constructors allow us to add input and output markers: $\&z := G$ takes a graph $G \in DB^{\mathcal{X}}_{\mathcal{Y}}$ and relabels input vertices with the input marker $\&z$, thus the result is in $DB^{\mathcal{Z}\cdot\mathcal{X}}_{\mathcal{Y}}$; $\&y$ returns a graph of a single vertex labeled with the default input marker & and the output marker $\&y$. () constructs an empty graph without any markers and vertices. The disjoint union $G_1 \oplus G_2$ requires two graphs G_1 and G_2 have disjoint sets of input markers. The operator $G_1 \mathbin{@} G_2$ vertically constructs a graph by plugging output markers of G_1 to input markers of G_2. It requires $G_1 \in DB^{\mathcal{X}}_{\mathcal{Y}}$ and $G_2 \in DB^{\mathcal{Y}}_{\mathcal{Z}}$, thus $G_1 \mathbin{@} G_2 \in DB^{\mathcal{X}}_{\mathcal{Z}}$. Finally, the last operator allows us to introduce cycles by adding ε-edges from an output marker to the input marker named after it.

Example 1 The left graph in Fig. 2 can be constructed as follows. (but not uniquely)

$$
\begin{aligned}
\&z \mathbin{@} \mathbf{cycle}((&\&z := \{\mathsf{a} : \&z_1\}) \\
&\oplus (\&z_1 := \{\mathsf{b} : \{\}\} \cup \{\mathsf{c} : \{\mathsf{b} : \&z_2\}\}) \\
&\oplus (\&z_2 := \{\mathsf{a} : \&z_1\}))
\end{aligned}
$$

For brevity, we write $\{l_1 : G_1, \ldots, l_n : G_n\}$ to denote $\{l_1 : G_1\} \cup \ldots \cup \{l_n : G_n\}$, and $(G_1, \ldots, G_n)$ to $G_1 \oplus \ldots \oplus G_n$.

2.3 Meaning of ε-edges

Theoretically, an ε-edge from a vertex v to v' means that all edges emanating from v' should be emanating from v [3]. Eliminating an ε-edge (v, ε, v') means removing this ε-edge and for each edge emanating from v', (v', a, w), a new edge (v, a, w) is added. Figure 4 shows an example of two equivalent graphs: one contains ε-edges and the other has no ε-edges. In this paper, we use dotted arrows to denote ε-edges.

$$
\begin{array}{rcll}
\mathit{prog} & ::= & \textbf{eval}\ f\,[\circ\, f]\ \textbf{where}\ \mathit{decl} \cdots \mathit{decl} & \{\ \text{program}\ \} \\
\mathit{decl} & ::= & f(\{l : \$g\})\ =\ t & \{\ \text{structural recursive function}\ \} \\
t & ::= & \{\} \mid \{l : t\} \mid t \cup t \mid \&x := t \mid \&y \mid () & \{\ \text{graph constructors}\ \} \\
 & \mid & t \oplus t \mid t \,@\, t \mid \textbf{cycle}(t) & \{\ \text{graph constructors}\ \} \\
 & \mid & \$g & \{\ \text{graph variable}\ \} \\
 & \mid & f(\$g) & \{\ \text{function application}\ \} \\
l & ::= & a \mid \$l & \{\ \text{label } (a \in \mathit{String}) \text{ and label variables}\ \}
\end{array}
$$

Fig. 5: Syntax of our DSL Language

3 A Domain-Specific Language

In this section, we will show our domain-specific language (DSL) and explain how to map programs written in our DSL to structural recursion. Although the DSL is simplified from the UnCAL language, it covers a wide class of structural recursion that can be efficiently evaluated in parallel. Moreover, thanks to the simplification, fusion rules can be freely applied to the composition of structural recursive functions [3].

3.1 Syntax and Semantics

Figure 5 shows the syntax of our language. A program starts with a header that specifies a composition of functions followed by a sequence of function declarations. Declarations are defined in the way of pattern matching and its body is an expression. For a function f, its argument is in the form of $\{l : \$g\}$ that is one of graph constructors presented before (Fig. 3). Note that, l can be a real label a or a label variable $\$l$. Declarations of f are based on pattern matching for $\{l : \$g\}$. Only one $f(\{\$l : \$g\})$ is allowed and must be located after all other declarations of $f(\{a : \$g\})$. The declaration $f(\{\$l : \$g\})$ will apply for graphs that do not match previous patterns. The body of a declaration is an expression including nine graph constructors, a graph variable and a function application. We require a strict form for function applications in which graph variable is the only argument, which avoids computations that may lead to infinite loop.

The semantics of our language is as follows. Given a set of structural recursive functions (defined by declarations), and a rooted edge-labeled graph, the program returns a new rooted edge-labeled graph by applying a transformation defined by the composition of structural recursive functions. Function composition is denoted by "$\circ$", and, from its definition, we have $(f_2 \circ f_1)\, x = f_2\, (f_1\, x)$. A declaration $f(\{l : \$g\})$ means, for each edge labeled l and its following subgraph $\$g$ in the input graph, we do some computations on l and then apply the structural recursive functions f on $\$g$. Results returned by applying a function f on adjacent edges are automatically combined by the constructor $\cup$ as follows: $f(G_1 \cup G_2) = f(G_1) \cup f(G_2)$.

Example 2 The following specification **a2d_xc** relabels edges a to d and contracts edges c. Figure 2 shows input and output graphs for this specification.

eval $a2d\_xc$ **where**
$a2d\_xc\,(\{\mathsf{a} : \$g\}) \;=\; \{\mathsf{d} : a2d\_xc\,(\$g)\}$
$a2d\_xc\,(\{\mathsf{c} : \$g\}) \;=\; a2d\_xc(\$g)$
$a2d\_xc\,(\{\$l : \$g\}) \;=\; \{\$l : a2d\_xc\,(\$g)\}$

Example 3 The following specification **c_b2d** changes all edges b, that are reachable from the edge c, to edges d. In this example, we need two mutually recursive functions.

eval c_b2d **where**
$c\_b2d\,(\{\mathsf{c} : \$g\}) \;=\; \{\mathsf{c} : b2d\,(\$g)\}$
$c\_b2d\,(\{\$l : \$g\}) \;=\; c\_b2d\,(\$g)$

$b2d\,(\{\mathsf{b} : \$g\}) \;=\; \{\mathsf{d} : b2d\,(\$g)\}$
$b2d\,(\{\$l : \$g\}) \;=\; \{\$l : b2d\,(\$g)\}$

Remark 1 (Our DSL and UnCAL) Our DSL consists of the essential part of UnCAL including graph constructors and function applications. Conditions **isempty** over graphs and the comparison of two label variables are omitted.

Fact 1 *Our language is powerful enough to described many interesting graph queries and graph transformations. For instance, it has been shown that any regular path queries can be described as mutually recursive functions [13], and graph updating can be compiled into structural recursion [6].* □

3.2 Programs as Parallelizable Structural Recursions

Given a function $e :: \{\mathsf{Label} \cup \varepsilon\} \rightarrow DB^{\mathcal{Z}}_{\mathcal{Z}}$, where $\mathcal{Z} = \{\&z_1, \ldots, \&z_n\}$. A function $h :: DB^{\mathcal{X}}_{\mathcal{Y}} \rightarrow DB^{\mathcal{X}\cdot\mathcal{Z}}_{\mathcal{Y}\cdot\mathcal{Z}}$ is called a parallelizable structural recursive function if the following data value equalities for nine graph constructors hold [3]:

$$
\begin{aligned}
h\,(\{\}) &\equiv (\&z_1 := \{\}, \ldots, \&z_n := \{\}) && (1)\\
h\,(\{\$l : \$g\}) &\equiv e(\$l)\,@\,h\,(\$g) && (2)\\
h\,(\$g_1 \cup \$g_2) &\equiv h\,(\$g_1) \cup h\,(\$g_2)\\
h\,(\&x := \$g) &\equiv \&x \cdot h\,(\$g) && (3)\\
h\,(\&y) &\equiv (\&z_1 := \&y \cdot \&z_1, \ldots, \&z_n := \&y \cdot \&z_n)\\
h\,() &\equiv ()\\
h\,(\$g_1 \oplus \$g_2) &\equiv h\,(\$g_1) \oplus h\,(\$g_2)\\
h\,(\$g_1\,@\,\$g_2) &\equiv h\,(\$g_1)\,@\,h\,(\$g_2)\\
h\,(\mathbf{cycle}(\$g)) &\equiv \mathbf{cycle}(h\,(\$g))
\end{aligned}
$$

In Eq. (3), $\&x \cdot (\&z_1 := \$g_1, \ldots, \&z_n := \$g_n)$ denotes $(\&x \cdot \&z_1 := \$g_1, \ldots,$ $\&x \cdot \&z_n := \$g_n)$.

For brevity, we denote the function h by $hom_{\mathcal{Z}}(e)$, and define it by the following equality.

$$hom_{\mathcal{Z}}(e)\,(\{\$l : \$g\}) = e(\$l)\,@\,hom_{\mathcal{Z}}(e)\,(\$g)$$

Structural recursive functions in our language are $hom_{\mathcal{Z}}(e)$ functions whose function e is obtained by transforming pattern matchings into the construct **if** ... **then** ... **else** and substituting recursive calls by markers.

For example, the specification **a2d_xc** is equivalent to $hom_{\{\&\}}(e)$, where,

$$\begin{aligned} hom_{\{\&\}}(e)(\{\$l : \$g\}) &= e(\$l)\,@\,hom_{\{\&\}}(e)(\$g) \\ e(\$l) &= \textbf{if } \$l = \mathsf{a} \textbf{ then } \{\mathsf{d} : \&\} \\ &\qquad \textbf{else if } \$l = \mathsf{c} \textbf{ then } \{\varepsilon : \&\} \textbf{ else } \{\$l : \&\} \end{aligned}$$

The specification **c_b2d** is equivalent to $\&z_1\,@\,hom_{\{\&z_1,\&z_2\}}(e)$ by tupling two mutually recursive functions *c_b2d* and *b2d* as follows. (Note that, here we extract the result of the function *c_b2d* by using the expression $\&z_1\,@$).

$$\begin{aligned} e(\$l) = \;& (\&z_1 := \textbf{if } \$l = \mathsf{c} \textbf{ then } \{\mathsf{c} : \&z_2\} \textbf{ else } \&z_1, \\ & \;\&z_2 := \textbf{if } \$l = \mathsf{b} \textbf{ then } \{\mathsf{d} : \&z_2\} \textbf{ else } \{\$l : \&z_2\}) \end{aligned}$$

Lemma 1 (Fusion rule [3]) *Given two parallelizable structural recursions $hom_{\mathcal{Z}_1}(e_1)$ and $hom_{\mathcal{Z}_2}(e_2)$, the following equality holds.*

$$hom_{\mathcal{Z}_2}(e_2) \circ hom_{\mathcal{Z}_1}(e_1) = hom_{\mathcal{Z}_2}(hom_{\mathcal{Z}_2}(e_2) \circ e_1)$$

Theorem 1 *A program written in our DSL language is equivalent to an expression $\&z_i\,@\,hom_{\mathcal{Z}}(e)$, where $z_i \in \mathcal{Z}$.*

4 Strategy for Deriving an Efficient Evaluation

The definition of parallelizable structural recursive function in Sec. 3 suggests a bulk semantics whose idea is to delay computations by introducing ε-edges. In general, to evaluate an expression $hom_{\mathcal{Z}}(e)$, the bulk semantics first creates a *bulk graph* by using ε-edges to connect results of the application of the function e on each edge. After that, the final result is obtained by computing transitive closure for ε-edges. In particular, the bulk graph is created as follows. For each vertex v, $|\mathcal{Z}|$ disjoint copies of v are created, then the function e is applied to every edge to create subgraphs with $|\mathcal{Z}|$ input markers and $|\mathcal{Z}|$ output markers. The bulk graph is created by, based on input and output markers, connecting disjoint vertices and subgraphs via ε-edges. It is clear that the bulk semantics enables a parallel evaluation. The following equation captures the above computation.

$$hom_{\mathcal{Z}}(e) = eelim \circ bulk_{\mathcal{Z}}(e)$$

where the function $bulk_{\mathcal{Z}}(e)$ is to compute a bulk graph and *eelim* is to eliminate ε-edges. Figure 2 shows an example of the above computation for the specification **a2d_xc**. Subgraphs surrounded by a shaded rectangle are results of the application of the function e on edges.

Now, we consider the situation where we use the bulk semantics to evaluate our program, $\&z_i \,@\, hom_{\mathcal{Z}}(e)$. Recall that the result of $hom_{\mathcal{Z}}(e)$ is a graph with $|\mathcal{Z}|$ input markers, and $\&z_i$ is one of its input markers. Therefore. $\&z_i \,@$ is actually a reachability computation that returns a graph whose edges and vertices are reachable from the vertex v, where $v = I(\&z_i)$. The reachability computation is a basic computation "skeleton" in Pregel model [8], so we consider it an efficient and scalable one from the standpoint of Pregel model. Now, we have

$$\&z_i \,@\, hom_{\mathcal{Z}}(e) = reach_{\{\&z_i\}} \circ eelim \circ bulk_{\mathcal{Z}}(e)$$

where the function $reach_{\{\&z_i\}}$ denotes the reachability computation for $\&z_i \,@$.

To minimize the amount of ε-edges as well as redundant edges produced by $bulk_{\mathcal{Z}}(e)$, we propose a hybrid approach of recursive semantics and bulk semantics. The recursive semantics is to compute a *marker graph* whose each vertex u consists of a set of *possible markers* $\mathcal{X}_u \subseteq \mathcal{Z}$. After that, in the bulk semantics, for each vertex u, we create exactly $|\mathcal{X}_u|$ disjoint vertices, and then the function e, instead of computing a graph of $|\mathcal{Z}|$ disjoint subgraphs, computes a graph of only $|\mathcal{X}_u|$ disjoint subgraphs. It is important to note that intermediate graphs generated are very close to the final result.

In order to express both semantics, we extract two functions $e_{\rightarrow}$ and e_π from the function e of $hom_{\mathcal{Z}}(e)$

$$\begin{aligned} e_{\rightarrow} &:: (\mathcal{M}_t, \{\mathsf{Label} \cup \varepsilon\}) \rightarrow \mathcal{M} \\ e_{\rightarrow}(\&z, \$l) &= \ \textbf{let}\ (vs, es, is, os) = \&z \,@\, e(\$l)\ \textbf{in}\ map\ snd\ os \\ e_\pi &:: (\mathcal{M}_t, \{\mathsf{Label} \cup \varepsilon\}) \rightarrow DB_{\mathcal{Z}}^{\mathcal{M}} \\ e_\pi(\&z, \$l) &= \&z \,@\, e(\$l) \end{aligned}$$

where, $\mathcal{M}_t$ is a type for markers; function $e_{\rightarrow}$ is like a transition function in automaton, where, for each label $\$l$ and input marker $\&z$ we compute a set of (output) markers reachable from $\&z$ in the graph generated by function e; function e_π is simply a project function. Note that these two functions are statically derived from a given function e.

Our program is now evaluated as follows.

$$\&z_i \,@\, hom_{\mathcal{Z}}(e) = reach_{\{\&z_i\}} \circ eelim \circ bulk'_{\mathcal{Z}}(e_\pi) \circ mark_{\{\&z_i\}}(e_{\rightarrow}) \qquad (4)$$

The function $mark_{\{\&z_i\}}(e_{\rightarrow})$ is to compute a marker graph using recursive semantics. It starts from the root of an input graph with the input marker $\&z_i$, and recursively uses $e_{\rightarrow}$ to find all markers a vertex can have. Vertices having no markers will be removed after that. The function $bulk'_{\mathcal{Z}}(e_\pi)$ is similar to $bulk_{\mathcal{Z}}(e)$ but e_π is applied with respect to markers $\&z$ of source vertices. Note that, although the function $bulk'_{\mathcal{Z}}(e_\pi)$ now generates a graph with only one input marker, the function $reach_{\{\&z_1\}}$ is necessary to find a smaller equivalent

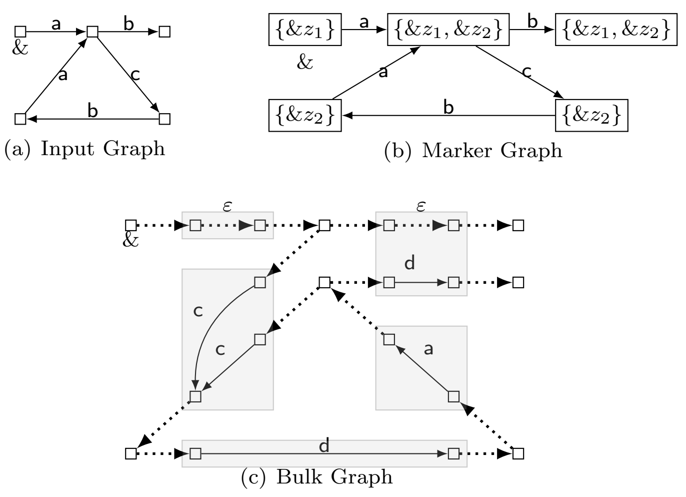

Fig. 6: Intermediate Graphs During the Computation for the Specification **c_b2d**. Values inside a box are vertex values not vertex ids

graph because the ε-edge elimination might produce redundant edges and vertices that are unreachable from the root.

We consider the example **c_b2d** to see in detail how to evaluate it with our approach. From its function e:

$$\begin{aligned} e(\$l) = \ & (\&z_1 := \textbf{if } \$l = \mathsf{c} \textbf{ then } \{\mathsf{c} : \&z_2\} \textbf{ else } \&z_1, \\ & \&z_2 := \textbf{if } \$l = \mathsf{b} \textbf{ then } \{\mathsf{d} : \&z_2\} \textbf{ else } \{\$l : \&z_2\}) \end{aligned}$$

we have two functions $e_{\rightarrow}$ and e_{π} as follows.

```
$e_{\rightarrow} = \lambda(\&z,\ \$l).$
    $(\&z,\ \$l)$ match {
      case $(\&z_1, \mathsf{c}) \Rightarrow \{\&z_2\}$
      case $(\&z_1, \_) \Rightarrow \{\&z_1\}$
      case $(\&z_2, \mathsf{b}) \Rightarrow \{\&z_2\}$
      case $(\&z_2, \_) \Rightarrow \{\&z_2\}$
    }

$e_{\pi} = \lambda(\&z,\ \$l).$
    $(\&z,\ \$l)$ match {
      case $(\&z_1, \mathsf{c}) \Rightarrow \&z_1 := \{\mathsf{c} : \&z_2\}$
      case $(\&z_1, \_) \Rightarrow \&z_1 := \&z_1$
      case $(\&z_2, \mathsf{b}) \Rightarrow \&z_2 := \{\mathsf{d} : \&z_2\}$
      case $(\&z_2, \_) \Rightarrow \&z_2 := \{\$l : \&z_2\}$
    }
```

Figure 6 shows intermediate graphs for the example **c_b2d**. The marker graph in Fig. 6(b) is the result of our recursive semantics and the bulk graph in Fig. 6(c) is the result of our bulk semantics. The marker graph is computed as follows. First, the root vertex is initialized with a singleton set $\{\&z_1\}$, where $\&z_1$ is the marker in our program. We evaluate the first edge (u, a, v) from the root. Its result, $e_{\rightarrow}(\&z_1, \mathsf{a}) = \{\&z_1\}$, is written to the vertex v. Next, we concurently evaluate two edges (v, b, w_1) and (v, c, w_2) emanating from v, and

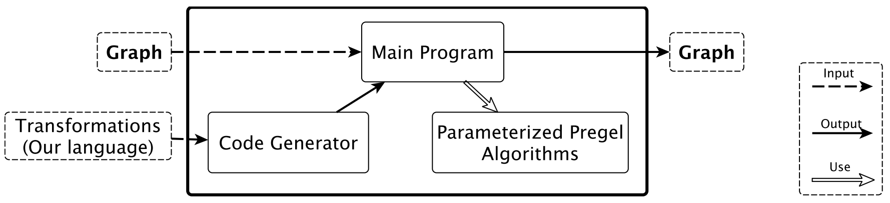
Graph
Transformations (Our language)
Main Program
Code Generator
Parameterized Pregel Algorithms
Graph
Input
Output
Use

Fig. 7: Overview of Our Framework

results are written to respective targets w_1, w_2. This procedure is iterated and then terminated when it can not find new markers to add to vertices. The bulk graph is then computed as follows. For a vertex u, and its set of markers $\mathcal{X}_u$, we create $|\mathcal{X}_u|$ disjoint vertices. Next, we apply the function e_π on each edge (u, l, v) and each marker in $\mathcal{X}_u$, producing a subgraph of $|\mathcal{X}_u|$ input markers. In Fig. 6(c), these subgraphs are surrounded by a shaded rectangle. Finally, we use ε-edges to connect disjoint vertices and subgraphs.

5 A High-Level Parallel Programming Framework for Graphs

In this section, we present our high-level framework that automatically compiles transformations written in our DSL into Pregel programs based on Eq. 4. We formalize some efficient Pregel computations and use them as basic skeletons to describe our evaluation.

5.1 Overview

Figure 7 shows components in our framework. It accepts a rooted directed edge-labeled graph and a specification written in our DSL language as inputs. It returns a rooted directed edge-labeled graph. The component "Code Generator" analyses the specification and generates a main program. In particular, it generates two functions $e_\rightarrow$ and e_π. It is also the place to do optimizations such as fusion rule or tupling rule. The component "Parameterized Pregel Algorithms" contains a set of efficient Pregel algorithms for functions such as *mark*, *bulk*, *eelim*, *reach*. The main program utilizes these functions in order to construct a complete program for the input specification. Next sections will focus on designing efficient Pregel algorithms for the component "Parameterized Pregel Algorithms".

5.2 Basic Skeletons in Pregel Programming Model

In order to describe our evaluation, we first formalize some efficient computations in the Pregel programming model. We refer to these computations as *basic skeletons*. Our formalization is inspired by APIs of GraphX library [18], a functional implementation of the Google Pregel model. However, the significant difference is that GraphX allows users to access to the full information of both the source vertex and the destination vertex of an edge, while our skeletons only access to the full information of the source vertex, and the id of destination vertex. In that sense, our skeletons are closer to the Google Pregel model.

We consider a distributed graph of a type of Graph[VD, ED], where VD is a type for vertex values and ED is a type for edge values. Vertex ID has a type of VID. Our function descriptions are written in functional programming style, in particular we borrow the syntax of *Scala* functional language. The notation `(a => b)` is a lambda abstraction `(λa.b)`. Returned value of a function is the returned value of the last statement in the function. Functions `_i` is to take the *i*-th element in a tuple, i.e. `(a, b)._2` returns the value `b`.

The first skeleton is `pregel`. This skeleton executes in a series of supersteps in which vertices receive an aggregate of incomming messages from the previous superstep, compute a new value for the vertex value, and send messages to neighbouring vertices in the next superstep. Only vertices receiving messages are involved in the next superstep. Its computation finishes when there are no messages in transit. Below is the signature of the `pregel` skeleton:

```
pregel[A](initialMsg : A)(
          vprog = (VID, VD, A) => VD,
          sendMsg = (VID, VD, ED, VID) => Message[(VID, A)],
          mergeMsg = (A, A) => A)
```

where, `vprog` is to update vertex values, its arguments include a vertex (id and value) and an aggregate of incoming messages. In the first superstep, the aggregate of incoming messages is initialized by the message `intializeMsg` and sent to all vertices. The argument of `sendMsg` is an edge including its source vertex, its source vertex value, edge label, and its target vertex id. The function `mergeMsg` is to define an aggregate operator over incoming messages.

In addition to the skeleton `pregel`, we have three one-step skeletons (`mapVertices`, `subgraph`, `mutateGraph`) and one two-step skeleton (`pregel2`):

```
mapVertices(vprog = (VID, VD1) => VD2)
subgraph(vpred = (VID, VD) => Bool,
         epred = (VID, VD, ED, VID) => Bool)
mutateGraph(efunc = (VID, VD1, ED1, VID)
                 => Set[(VID, VD2, ED2, VID)])
pregel2[A](sendMsg = (VID, VD, ED, VID) => Message[(VID, A)],
           mergeMsg = (A, A) => A,
           vprog = (VID, VD, A) => VD)
```

The skeleton `mapVertices` applies a function to vertices to update their values, it might change the type of vertex values but always keep the vertex id unchanged. The skeleton `subgraph` filters a graph based on vertices or edges or both. The skeleton `mutateGraph` generates a new graph from the union of sets of edges generated by applying the function `efunc` on edges. Users must ensure the consistence of ids and values for vertices. The `pregel2` is quite similar to MapReduce computation, vertices send a message to its neighbouring vertices, then, in the next superstep, vertices combine the incoming messages and update their values.

For example, the following code computes a graph whose vertices and edges are reachable from a given vertex, say, a vertex with id 1.

```
g.mapVertices((vid, vd) => (vid==1 ? true : false, false)
 .pregel(false)(
      (vid, (re, flag), ms) => {
            (re && !ms) ? (re, true) : (ms, !re)
      },
      (srcid, (re, flag), l, dstid) => {
             flag ? Message[(dstId, true)]: Message[Empty]
      },
      (a, b) => a | b)
 .subgraph(vpred = (vid, vd) => vd._1)
```

Next sections will show how to use these skeletons to express our evaluation.

5.3 Marker Graph Computation

A marker graph is computed by the skeleton `pregel`. Each vertex maintains a triple (x, ys, zs), where x denotes whether the vertex is the root or not, ys is a set of markers and zs is a set of new markers received from other vertices but not in ys. Let r be the id of the root in the input graph. Following is the computation of the function $\texttt{mark}_{\{\&z\}}(e_{\rightarrow})$.

```
  mark = (&z, e→, g) => {
   g.mapVertices((id,_) => ((id==r)? true:false, ∅, ∅))
    .pregel(∅)(
     (vi, vd, msg) => {  /* vprog */
        let (isRoot, curr_mrk, new_mrk) = vd  in
        if (isRoot) {
           (isRoot, &z, &z)
        } else {
           (isRoot, curr_mrk ∪ msg, msg \ (msg ∩ curr_mrk))
        }
     },
     (si, sd, ed, di) => { /* sendMsg */
        outMrk = sd._3.flatMap(mrk => e→(mrk, ed))
        (!outMrk.empty) ? Message((di, outMrk)) : Message[Empty]
     },
     (a, b) => a ∪ b) /* mergeMsg */
    .mapVertices((id, v) => (v._1, v._2)) }
```

5.4 Bulk Graph Computation

To compute a bulk graph from a marker graph, we need only one superstep and use the skeleton `mutateGraph`.

```
bulk = (eπ, g) => {g.mutateGraph(
  (si, sd, ed, di) => {
    sg = sd._2.map(mrk => eπ(mrk, ed))
              .reduce((a, b) => a ⊕ b) /*disjoint subgraphs*/

    InEps = sg.I._1.map((m,id) => Edge(genId(m, si), sd, ε, id)
    OutEps = sg.O._2.map((m,id)
                    => Edge(id, (false,∅), ε, genId(m, di))

    InEps ∪ sg.E ∪ OutEps
  }).mapVertices((vi, vd) => vd._1) }
```

We apply a function e_π on edges to create a "forest" of disjoint subgraphs. Disjoint vertices are created from markers and source/target vertex ids. We need a function `genId` to generate a unique id for a pair of a marker and a vertex id, which helps Pregel model correctly merge new vertices to form a new graph. Note that, all vertices in the graph `sg` are initialized to a default value `(false, ∅)`, while disjoint vertices of a source vertex are initialized to the value of the source vertex.

5.5 Elimination of ε-edges

We propose a parallel implementation to eliminate ε-edges in a graph. It includes two phases, the first one is to send information of ε-edges and the second one is to contract ε-edges. Consider two consecutive edges (u, ε, v) and (v, l, w), eliminating the edge (u, ε, v) means (1) adding an edge (u, l, w) and (2) removing the edge (u, ε, v). The removing of the edge (u, ε, v) should be done at the vertex u, and the adding of an edge (u, l, w) should be done at the vertex v because only the vertex v has the information of l and w. However, the vertex v has no information of its incoming vertex u, so we need an additional superstep to send the information of the vertex u to v. Note that, this parallel implementation can not deal with cycles of ε-edges (all edges in the cycles are ε-edges), but we believe this problem can be easily solved by computing strongly connected components in advance.

Figure 8 shows a simple example of the ε-edge elimination with 5 supersteps. Values inside a box denote the id of a vertex, and values below a vertex are content of incoming messages. After the first phase (sending ε information), vertices $2, 3$ and 4 know their sets of incoming edges $\{1\}$, $\{2\}$ and $\{3\}$ respectively. Then, they use these sets to contract ε-edges in the second phase. In the fifth superstep, we can not find any ε-edges so we stop the ε-edge elimination.

We use the skeleton `pregel2` to aggregate ε-edges information and use the skeleton `mutateGraph` to contract ε-edges.

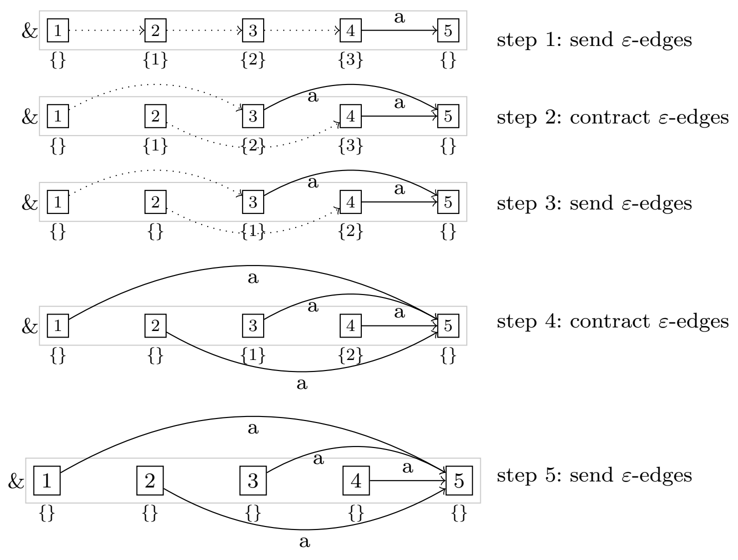

Fig. 8: Parallel Elimination of ε-edges. Here, a value inside a vertex is a vertex id, and a set below a vertex is the union of messgages the vertex received

```
epsElim = g => {
  let epsAgg = pregel2(
    (si,sd,ed,di) => {
      (ed == ε) ? Message((di,{(si,sd)})) : Message[Empty]
    },
    (a, b) => a ∪ b,
    (vi, vd, ms) => (vd._1, ms))
  in
  g = epsAgg(g)
  epsInfo = g.subgraph(vpred = (vi,vd) => !vd._2.empty)
  while (epsInfo.vertices.count() > 0) {
    g = g.mutateGraph(
          (si, sd, ed, di) => {
            res = sd._2.map((vi, vd) => Edge(vi, vd, ed, di))
            if (ed == ε) {
              res                          /* delete ε-edge */
            } else {
              res ∪ Edge((si, sd, ed, di))
            }
          })
    g = epsAgg(g)
    epsInfo = g.subgraph(vpred = (vi,vd) => !vd._2.empty)
  }
}
```

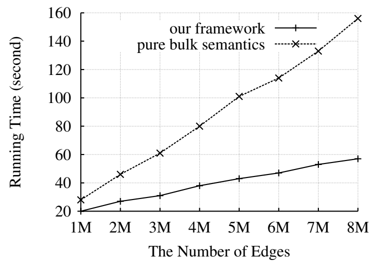

Fig. 9: Varying graph size (Citation)

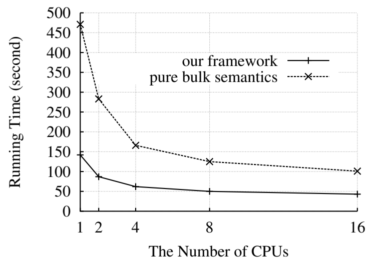

Fig. 10: Varying CPU number (Citation)

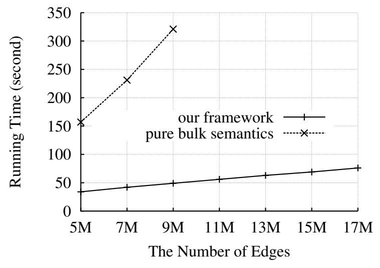

Fig. 11: Varying graph size (Youtube)

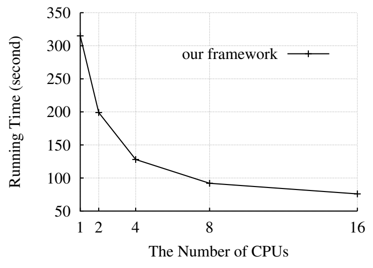

Fig. 12: Varying CPU number (Youtube)

6 Preliminary Results

Apart from our framework, we implemented the pure bulk semantics [3], in which we used our parallel algorithm for the phase of ε-edge elimination. We will compare our framework with the pure bulk semantics to evaluate our proposal. Both the pure bulk semantics and our framework[1] are implemented in GraphX system [18].

The environment of our experiments is the following: Intel(R) Xeon(R) CPU E5620@2.40GHz 16 cores, 48GB memory, GraphX in Spark 1.3.0, Hadoop 2.4.2. We set up GraphX working in a distributed mode.

We used two following datasets: (a) Citation[2], in which data are papers (Title, Authors, Conference, Year, References), (b) Youtube[3], in which data are videos (Uploader, Category, Length, Related IDs, etc.). These datasets need to be converted to rooted edge-labeled graphs. Figure 1 at the beginning of this paper shows a snapshot of a graph for the citation network. The citation graph has 7 462 913 vertices and 8 784 415 edges, and the youtube graph has 13 501 697 vertices and 17 179 944 edges.

[1] source code: http://www.prg.nii.ac.jp/members/tungld/gito-graphxApr1.tar.gz
[2] http://arnetminer.org/billboard/citation, dataset: citation-network V1
[3] http://netsg.cs.sfu.ca/youtubedata/, dataset: 0222, Feb. 22nd, 2007

We did experiments with transformations of four mutually recursive functions. First, we changed graph size to see how our framework performs when increasing the size of input data, while setting the number of cpus to 16. Figure 9 and 11 show results for citation and youtube graphs, respectively. It is clear that, for the citation graph, our framework outperforms the pure bulk semantics though the pure bulk semantics seems linear to the size of graph. However, for the youtube graph, the pure bulk semantics can not deal with a graph of 11 million edges. It can generate a bulk graph, but can not finish the ε-edge elimination due to an “out-of-memory” error. By contrast, our framework works smoothly even for the graph of 17 million edges.

Next, we changed the number of cpus and fix the graph size at 5 million edges for the citation graph (Fig. 10) and 17 million edges for the youtube graph (Fig. 12). Both the pure bulk semantics and our framework follow the same shape. When we double the number of cpus from 1 to 2, or 2 to 4, both achieves a speedup of 2. But, after that, for 8 and 16 cpus, we do not have the same performance. This is because when we increase the number of cpus, we will have more partitions. The local computation time for each partition is decreased but the communication time is increased. Also from these experiments, we can see that our framework is about 2-3 times faster than the pure bulk semantics. This is reasonable for the specifications with four mutually recursive functions, because the pure bulk semantics generates a bulk graph of about 4 times larger than the input, which makes its computation time slower than our framework where there is no duplication data generated.

7 Related Work

Graph Processing: Systematically developing graph algorithms is non-trivial due to the existence of cycles. Some works have tried to reduce problems on graphs to the ones on trees whose systematic solutions have known. Wang et al. [16] proposed a systematic approach for graph problems via tree decomposition. Wei [17] used tree decomposition as an indexing method for answering reachability queries. Another approach is to develop a new calculus for graphs. UnCAL algebras is based on structural recursion [3]. GraphQL [5] is a graph query language whose core is a graph algebra. Compositions of graph structures are allowed by extending the notion of formal languages from strings to the graph domain. Graph grammars have been used for graph transformations in various domains [11]. However, both UnCAL and GraphQL are different in that their focus has been on graph databases. Our DSL is a subset of UnCAL language.
Structural Recursion: Basically, there are two ways to evaluate a structural recursion: recursive semantics and bulk semantics [3]. The idea of recursive semantics is to apply a function on edges recursively from the root of an input graph. Memorization is used to avoid infinite loops in which results of each recursive call are stored at vertices. The advantage of recursive semantics is that its computation produces only necessary data that are involved in the final

result. No redundant data are produced. However, the disadvantage is that we have to do a heavy computation at each step, leading to a slow convergence. On the other hand, bulk semantics is trying to delay computations by introducing ε-edges. For each edge, it computes all possible results and the final result is returned after computing transitive closure of ε-edges. The bulk semantics potentially enables parallel evaluation, but in practice, it generates a lot of redundant data so that it is impractical to large graphs. Our evaluation in this paper is a hybrid approach of recursive and bulk semantics. It has both the advantage of the recursive semantics in producing a small amount of redundant data and the advantage of the bulk semantics in exploiting parallelism.

High-Level Framework: One of the few works on processing queries using Pregel is proposed by Nole et al. [10], in which Brzozowski's derivation of regular expressions are exploited. In consequence, queries are limited to regular path queries. Krause et al. [7] proposes a high-level graph transformation framework on top of BSP model. In particular, they implemented the framework in Giraph, an open-source implementation of Pregel model. The framework is based on graph grammars. Another approach is done by Salihoglu et al. [12], in which they have found a set of high-level primitives that capture commonly appearing operators in large-graph computations. These primitives are also implemented in GraphX library.

8 Conclusion

In this paper, we have proposed a systematic framework for transformations over big graphs. Our approach is to combine the advantages from a solid foundation of graph algebra and a practical scalable graph processing model. Preliminary results show that this combination is very promising where our framework outperforms the original bulk semantics and achieves both good scalability and speedup. We consider this work an important step in order to port the whole UnCAL language to the Pregel model.

In the future, we will extend our framework in two directions. The first direction is to deal with more complex computations such as providing conditions over graphs. In our current language, we only allow to use the variable $\$g$ of a function $f(\{\$l, \$g\})$ in its recursive call. It is possible to allow conditions over the graph $\$g$, i.e. if **isempty**($\$g$) then ... else ... However, we have not known yet the way to efficiently compute such structural recursive functions in the Pregel model. The second direction is to optimize the core of our framework. Although our DSL allows compositions of structural recursion and fusion rules can be freely applied, we have not made fusion mechanism be automatically done yet. Efficient computation of transitive closure (TC) in Pregel environment is also an important improvement to our framework. Afrati et al. [1] proposes an efficient distributed algorithm to compute TC on clusters. That would be interesting to integrate that algorithm into our framework to eliminate ε-edges efficiently. Last but not least, we need to compare our framework with others to see more details of its performance.

References

1. Afrati, F.N., Ullman, J.D.: Transitive Closure and Recursive Datalog Implemented on Clusters. In: Proceedings of the 15th International Conference on Extending Database Technology, EDBT '12 (2012)
2. Buneman, P.: Semistructured Data. In: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '97, pp. 117–121. ACM, New York, NY, USA (1997)
3. Buneman, P., Fernandez, M., Suciu, D.: UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. The VLDB Journal **9**(1) (2000)
4. Emoto, K., Fischer, S., Hu, Z.: Generate, Test, and Aggregate: A Calculation-based Framework for Systematic Parallel Programming with Mapreduce. In: Proceedings of the 21st European Conference on Programming Languages and Systems, ESOP'12, pp. 254–273. Springer-Verlag, Berlin, Heidelberg (2012)
5. He, H., Singh, A.K.: Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08 (2008)
6. Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Towards a Compositional Approach to Model Transformation for Software Development. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, pp. 468–475. ACM, New York, NY, USA (2009)
7. Krause, C., Tichy, M., Giese, H.: Implementing Graph Transformations in the Bulk Synchronous Parallel Model. In: S. Gnesi, A. Rensink (eds.) Fundamental Approaches to Software Engineering, *Lecture Notes in Computer Science*, vol. 8411, pp. 325–339. Springer Berlin Heidelberg (2014)
8. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A System for Large-scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10 (2010)
9. Matsuzaki, K., Iwasaki, H., Emoto, K., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In: Proceedings of the 1st International Conference on Scalable Information Systems, InfoScale '06. ACM, New York, NY, USA (2006)
10. Nolé, M., Sartiani, C.: Processing Regular Path Queries on Giraph. In: EDBT/ICDT Workshops (2014)
11. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
12. Salihoglu, S., Widom, J.: HelP: High-level Primitives For Large-Scale Graph Processing. In: Proceedings of Workshop on GRAph Data Management Experiences and Systems, GRADES'14, pp. 3:1–3:6 (2014)
13. Suciu, D.: Distributed Query Evaluation on Semistructured Data. ACM Trans. Database Syst. **27**(1) (2002)
14. Tung, L.D., Nguyen-Van, Q., Hu, Z.: Efficient Query Evaluation on Distributed Graphs with Hadoop Environment. In: Proceedings of the Fourth Symposium on Information and Communication Technology, SoICT '13. ACM, New York, NY, USA (2013)
15. Valiant, L.G.: A Bridging Model for Parallel Computation. Commun. ACM **33**(8), 103–111 (1990)
16. Wang, Q., Chen, M., Liu, Y., Hu, Z.: Towards Systematic Parallel Programming of Graph Problems via Tree Decomposition and Tree Parallelism. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-performance Computing, FHPC '13, pp. 25–36 (2013)
17. Wei, F.: Efficient Graph Reachability Query Answering Using Tree Decomposition. In: Proceedings of the 4th International Conference on Reachability Problems, RP'10, pp. 183–197 (2010)
18. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, pp. 2:1–2:6 (2013)