Extending Fregel for Functional Vertex-centric Graph Processing with Remote Access

Yongzhe Zhang　Kento Emoto　Zhenjiang Hu

Fregel provides a functional interface and a clear functional semantics to let users easily write vertex-centric programs. Unfortunately, Fregel is limited in its lack of support for remote access (reading or writing attributes of other vertices through reference), which makes it hard to describe a class of important graph algorithms that communicate over dynamic internal data structures. In this paper, we propose to extend Fregel to support remote access by introducing more language constructs to concisely represent remote reads and writes. We show how this extension can be efficiently implemented and evaluate it through some practical examples.

1 Introduction

Nowadays, more and more real-world applications rely on the big graph data, which makes parallel processing of large-scale graphs very important. To handle graphs in such scale, Google's Pregel [10] system adopts the *vertex-centric* programming model (also known as "think like a vertex" paradigm), which supports scalable big graph processing through iterative supersteps that execute in parallel a user-defined vertex program over each vertex of a graph. There are several open-source graph processing frameworks that are influenced by Pregel, such as GraphLab [9], Giraph [1] and Pregel+ [15].

Despite the power of Pregel, it is however unintuitive for programmers to implement graph algorithms in Pregel's imperative and message-passing style [3, 6, 17], especially when the algorithm consists of multiple stages and complicated data dependencies. For such algorithms, programmers need to write an exceedingly complicated vertex program, which encodes all the stages of the algorithm. Message passing makes the code even harder to maintain, because one has to trace where the messages are from and what information they carry in each superstep. To address this problem, several domain-specific languages are proposed to provide a more intuitive way of Pregel programming, such as Green-Marl [7], Fregel [3], s6raph [11] and Palgol [17]. In these DSLs, instead of the message passing interface, programmers are provided with a more convenient abstraction so that they can directly access data from a vertex's *neighborhood*, then the compiler helps to generate a correct vertex-centric Pregel program in terms of supersteps and message passing.

Although the core idea of these DSLs are similar, they have different focuses and designs, which makes them distinguishable from each other.

Yongzhe Zhang, Zhenjiang Hu, 総合研究大学院大学複合科学研究科情報学専攻/国立情報学研究所, Department of Informatics, School of Multidisciplinary Sciences, SOKENDAI; National Institute of Informatics (NII).

Kento Emoto, 九州工業大学, Kyushu Institute of Technology.

Among them, Fregel provides a purely *functional* model for Pregel, modeling the Pregel computation as a special higher-order function *fregel*, and letting users to declaratively specify the step functions (used in fregel) to update a vertex's state using the information from the vertex's neighborhood. In spite of the functional interface and the clear semantics that make Fregel programs suitable for reasoning, its restriction of data access only to the neighborhood makes it hard to describe many interesting graph algorithms that need global communicate, say through dynamic internal data structures.

Fortunately, Palgol [18], a new domain specific language for vertex-centric computation, introduces a good mechanism for remote (data) access, which highly improves the expressiveness of the excising languages with powerful communication capability. This remote access makes it possible to describe those efficient algorithms that need the pointer jumping technique (a representative example will be presented in Section 3. 4). However, Palgol is an imperative language and its semantics with remote access is not clearly defined, which makes Palgol programs difficult to reason about.

A natural question arises: can we have a DSL that combines the advantages of both Fregel and Palgol? Or put it in another way, is it possible to design a DSL that can have both a functional interface (for easy reasoning) and the remote access capacity (for high expressiveness)? A straightforward idea is to extend Fregel with the remote access capacity Palgol has. However, there are two challenges in this extension. One challenge is how to express the remote access (remote reads and remote writes) as a pure function of a vertex updating function. Remote writing, a destructive action, does not seem to fit to a functional setting. The other challenge is how to transform this high-level remote access to the lower-level message passing in Pregel.

In this paper, we propose an extension of Fregel which fully integrates the remote access capacities to Fregel, and thus provides a more powerful functional interface for vertex-centric graph processing. Our technical contributions can be summarized as follows.

- We extend Fregel to support remote access by allowing each vertex to store and update reference values. To deal with remote writing in a functional manner, we propose a two-step method showing that a remote write can be realized by composition of two remote reads.
- We show that the extended Fregel can be efficiently implemented by defining a transformation from this extened Fregel to Palgol. As an application, we demonstrate its use in parallel computation of a class of interesting graph algorithms using the pointer jumping technique.

The organization of the rest of the paper is as follows. Section 2 explains the background of Pregel, Fregol and remote access. Section 3 describes the extension of Fregel, then Section 4 explains the transformation from the extended Fregel to Palgol. We discuss related work in Section 5, and conclude the paper in Section 6.

2 Background

In this section, after briefly reviewing the Pregel-like systems, we give an overview of Fregel, the target language to be extended, and explain the remote access capability in Palgol.

2. 1 The Pregel-like Systems

Pregel [10] is a framework proposed for processing large-scale graphs in the vertex-centric manner, and there are several Pregel-like open-source frameworks [1, 5, 9, 12, 15] that have been implemented.

In Pregel, the computation is divided into finite number of supersteps, and within in each superstep,

every vertex computes the same user-defined function in parallel to accomplish the logic of a given algorithm. The same user-defined function typically does the following things in a superstep:

- read the messages sent from the previous superstep;
- perform the local computation and change current vertex's state;
- send messages to other vertices.

When all the vertices finish the local computation in each superstep, all vertices perform the bulk synchronization, where the messages sent in the last superstep are handled by the system, and are delivered before the next superstep. In addition to the message exchange, Pregel provides an interface called *aggregator*, where users can provide an operation, such as sum or max, to collectively processing global information from all vertices. Furthermore, vertex inactivation is also provided for terminating the computation or optimizing the program by skipping some vertices.

2.2 Fregel: Functional DSL for Pregel

Fregel [3] is a high-level functional DSL for Pregel, but it does not use explicit message passing like Pregel. Instead, it provides a more convenient abstraction that enables a vertex to update itself by accessing its neighbors' state. Fregel is equipped with a compiler that can transform Fregel programs with neighbor access to Pregel codes with message passing. Fregel is a subset of Haskell, so it is possible to execute a Fregel program using an Haskell intepreter for debugging or testing.

2.2.1 Graph Model

In Fregel, a vertex has the following type, containing its own unique id, the vertex value, and its incoming neighbors:

```
data Vertex a b = Vertex {
  vid :: Int,
  val :: a,
  is :: [Edge a b]
}
type Edge a b = (b, Vertex a b)
```

where an edge is a pair of its label and the vertex it points to. Then, a graph is represented by a list of vertices:

```
type Graph a b = [Vertex a b]
```

For instance, the following is an example of a graph g containing 5 vertices:

```
g :: Graph Bool Int
g = let
  v0 = Vertex 0 True []
  v1 = Vertex 1 True [(1,v0)]
  v2 = Vertex 2 True [(1,v0)]
  v3 = Vertex 3 True [
          (1,v1),(1,v2),(1,v4)]
  v4 = Vertex 4 False []
in [v0, v1, v2, v3, v4]
```

Since every vertex stores its neighbors, such representation forms a potentially cyclic data structure.

2.2.2 Core of Fregel

Figure 1 presents the core part of the syntax of Fregel. A Fregel program defines a function that takes a single input graph and returns a resultant graph. The expressions in Fregel are standard ones in Haskell: a combine function applied to a comprehension with specific generators, table access on a vertex v followed by the field selection operator denoted by "`.^`"), and a graph expression with higher-order functions on graphs such as ***fregel***. There are three generators in Fregel: a graph variable to generate all vertices, and ***is*** v / ***rs*** v to generate every pair of v's adjacent vertex u connected by an incoming or reversed edge and the value on this edge.

There are two kinds of definitions in **let**-expressions: ones for a definition of a normal function or constant including an initialization function and ones for a step function, which is the only higher-order function that a user can define. A step function takes two tables, ***prev*** and ***curr***, as well as vertex v, which repeatedly executes the step function. There is another table ***val*** that is used to

$$
\begin{array}{lcl}
prog & := & f\ g\ =\ e \\
e & := & \textbf{let}\ decl_1\ \cdots\ decl_n\ \textbf{in}\ e \\
 & | & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
 & | & f\ e_1\ \cdots\ e_n \\
 & | & comb\ [\ e\ |\ gen,\ e_1,\ \ldots,\ e_n\ ] \\
 & | & table\ v\ .\hat{}\ fld_1\ .\hat{}\ \cdots\ .\hat{}\ fld_n \\
 & | & \textbf{\textit{fregel}}\ f_1\ f_2\ tc\ g \\
 & | & \textbf{\textit{gmap}}\ f\ g \\
 & | & \textbf{\textit{gzip}}\ g_1\ g_2 \\
 & | & \textbf{\textit{giter}}\ f_1\ f_2\ tc\ g \\
decl & := & f\ x_1\ \cdots\ x_n\ =\ e \\
 & | & f\ v\ \textbf{\textit{prev curr}}\ =\ e \\
gen & := & u\ \leftarrow\ g \\
 & | & (ed, u)\ \leftarrow\ \textbf{\textit{is}}\ v \\
 & | & (ed, u)\ \leftarrow\ \textbf{\textit{rs}}\ v \\
table & := & \textbf{\textit{prev}}\ |\ \textbf{\textit{curr}}\ |\ \textbf{\textit{val}} \\
tc & := & \textbf{\textit{Fix}}\ |\ \textbf{\textit{Until}}\ (\lambda\ g.e)\ |\ \textbf{\textit{Iter}}\ e \\
comb & := & \texttt{max}\ |\ \texttt{or}\ |\ \cdots \\
 & & \\
f & := & \text{variable representing a function/operator} \\
g & := & \text{variable representing a graph} \\
u, v & := & \text{variable representing a vertex} \\
ed & := & \text{variable representing an edge} \\
fld & := & \text{field name}
\end{array}
$$

図 1 Core part of Fregel syntax.

access the values in the input graph. There are three kinds of termination conditions for higher-order functions ***fregel*** and ***giter***.

It is worth noting that Fregel is equipped with four higher-order functions for graphs that provide ways to concisely write various graph computations. Function **fregel** corresponds to Pregel's execution logic, which execute an initialization step f_1 at the beginning, then execute a step function f_2 until a termination condition is satisfied. Function **giter** is similar to **fregel** except that f_2 is now a graph transformation. Function **gzip** pairs values on every corresponding vertices in two graphs of the same shape, and **gmap** applies a given vertex function to every vertex.

As a simple example, the Fregel program in Figure 2 is to mark *True* of those vertices that are reachable from vertex 0, and *False* otherwise. Note that since the step function for this problem returns a Boolean value indicating whether the vertex that calls the step function is currently reachable or not,

```
1  data RVal = RVal { rch :: Bool }
2
3  reAll g =
4    let init v = RVal (vid v == 0)
5        step v prev curr =
6          let newrch =
7                prev v .^ rch ||
8                or [ prev u .^ rch
9                   | (e, u) ← is v ]
10         in RVal newrch
11   in fregel init step Fix g
```

図 2 A Fregel Program

we thus define a record *RVal* that contains only this Boolean value at the *rch* field in this record (Line 1).

The function *reAll* is the main part of the program, and it defines the initialization and step functions. The initialization function, *init*, returns an *RVal* record in which the *rch* field is *True* only if the vertex is the staring point (vertex identifier is zero). The vertex identifier can be obtained by using a special predefined function, ***vid***. The step function, *step*, collects data at the previous clock from every adjacent vertex connected by an incoming edge. This is done by using the syntax of comprehension, in which the generator is ***is*** v. For every adjacent vertex u, this program obtains the result at the previous clock by ***prev*** u and accesses its *rch* field. Then *step* combines the results of all adjacent vertices by using the *or* function and returns the disjunction of the combined value and its own *rch* value at the previous clock.

2.3 Remote Access

Remote access is introduced in Palgol [17] for supporting flexible communications between non-neighboring vertices. Essentially, it extends the peek-based functional model by introducing the following two abstract constructs:

- *global field access*: a vertex can access the state of an arbitrary vertex as long as it holds a reference of that vertex. Moreover, users can

express *chain access*, which accesses some remote data by following the references several times.

- *remote write*: a vertex can modify the state of another vertex by specifying the update value, the destination vertex, the field to update, as well as a unique reduce operator. Then, an additional superstep is attached at the end of current logic superstep to transfer the messages and perform the update accordingly.

With these two construct, more Pregel algorithms with non-ordinary communication patterns can be implemented, including a class of efficient Pregel algorithms that apply the pointer jumping technique [2, 16].

3 Extension of Fregel

In this section, we describe our extension of Fregel to support remote access. We first show how to extend Fregel to describe remote reads and remote writes. Then, we present a Haskell implementation of our extended Fregel to help user better understand the semantics of this extension. Finally, we provide an example to show how we can easily describe algorithms using our extension of remote access.

3.1 Remote Reads through Reference

Palgol introduces a special syntax called *field access* for representing remote reads through *reference*, which is in the form of `FieldName[exp]`. In this expression, `FieldName` is a user-specified field, and `exp` is an expression with a special type called reference (and physically, it is just the vertex identifier) which refers to some vertex. Then, the semantics of such field access expression is to fetch the specified field from the designated vertex. Could we add the concepts of field access and reference to Fregel?

The solution turns out to be natural. Initially, Fregel has the table lookup interface for representing data access, but only with its list comprehension syntax we can extract its neighbors' vertex identifier and perform neighborhood data access. To enable remote read from an arbitrary vertex, we slightly modify our model that allow a vertex to store a reference to some other vertex, and provide a table lookup interface to enable remote data access through reference. To achieve so, we extend the definition of the step function $f^{\dagger 1}$ in Figure 1 from

$$f\ v\ \boldsymbol{prev}\ =\ e$$

to

$$f\ v\ \boldsymbol{prev}\ \boldsymbol{refs} =\ e$$

where the step function has the new type of `StepFunc` defined sa follows:

```
data VRef = VRef { idx :: Int }
type StepFunc a b r = Vertex a b
   -> (Vertex a b -> a)
   -> (VRef -> a)
   -> r
```

First, we introduce a new data type `VRef`, which is essentially a simple wrap of some other vertex's identifier (which has type `Int`). A value with this type exactly represents a reference to other vertex. Then, two tables are passed into the step function, which are:

- A `prev` table with type `Vertex a b -> a` allows a vertex to access the state of a neighboring vertex in the previous logical step. This is exactly the same as the original Fregel's `prev` table.
- A `refs` table with type `VRef -> a` allows a vertex to access the state of an arbitrary vertex in the previous logical step via a reference.

To see how this new interface works, consider an input graph having the form of a rooted tree where each vertex initially has a reference to is parent (root vertices point to themselves), and we want ev-

†1 We omit the argument ***curr*** in f, since ***curr*** is planned to be removed in Fregel.

ery vertex to compute its grandparent (the parent of parent). Suppose the input data structure has a field `parent` to store the reference to the parent, we can obtain the grandparent using the following step function:

```
data VD = VertexData {
  parent :: VRef,
  grandp :: VRef
}
step :: StepFunc VD Int VD
step v prev refs =
  let
    p = prev v.^parent
    g = refs p.^parent
  in VertexData p g
```

3. 2 Remote Writes

In the Fregel's functional model, each vertex basically executes a pure function which collects data from its neighbors and generates a new vertex state, therefore it is hard to model remote writes in Fregel where a vertex directly modifies other vertices' states. To support Pregel algorithms that require the remote write capacity, we introduce a new approach that uses two steps to simulate the remote writes in the functional model.

The basic idea is simple. Recall that a remote write essentially requires a vertex to specify the destination vertex, the value to update, and the operation that is performed on the destination vertex to handle the updates. So, in our functional model, we can use two logical steps to achieve this:

- In the previous logical step, a vertex stores the reference to the remote vertex and the update value in the vertex value.
- In the consequent logical step, we use the list comprehension syntax with a special global generator (`u <- g`) to let a vertex v update itself (i.e., write) by collecting all those vertices that store a reference to v.

We will see a concrete example in Section 3. 4.

3. 3 Haskell Implementation

As mentioned in Section 2. 2, Fregel is a subset of Haskell, and a concrete implementation of Fregel in Haskell given by the authors can not only help programmers understand the behavior of Fregel better, but also make them easier to test and debug the Fregel programs. Following this methodology, we provide a Haskell implementation for our extension on Fregel. The new implementation turns out to be incremental: we just slightly change the definition of `fregel`, introduce a new high-order function called `gmaps` to convert a step function to a single-step graph transformation, and add an auxiliary function `getVertexRef` to create a reference from a vertex, as seen in Figure 3.

The semantics of `gmaps` is explained in terms of the slightly revised higher-order function `fregel` that is adapted from the original definition in [3] for coping with the new step function having the reference table. Essentially, the `fregel` function simulates a pregel computation. It takes four arguments as input (`fregel h f t g`) and produces an output graph. More detailed, h is an initialization function that executes just once beginning, and f is a step function that is applied to the graph again and again. Finally, the `t` provides different ways of terminating a computation, and we can just use "Iter 1" to say that the step function executes exactly once.

The importance of introducing such `gmaps` is to better support the remote writes: when specifying two consecutive step functions, the `gmaps` function can turn each step into a graph transformation, and functions in such types are much easier to be composed together. In the following, we will see how this function helps writing concise and intuitive Fregel code.

```
fregel :: (Vertex a b -> r) -> StepFunc a b r -> Term (Graph r b) -> Graph r b
fregel h f t g =
  let rs0 = map h g
      step rs = let rs' = map (\v -> f v prev refs) g
                    prev u = rs !! (getVertexId u)
                    refs r = rs !! (idx r)
                in rs'
      rss = iterate step rs0
  in termination t (map (graphy g) rss)

gmaps :: StepFunc a1 b a2 -> Graph a1 b -> Graph a2 b
gmaps f g = fregel default_init f (Iter 1) g
    where default_init Vertex { val = v } = v

getVertexRef :: Vertex a b -> Ref
getVertexRef Vertex { idx = r } = VRef r
```

図 3 Defintions of revised `fregel` and `gmaps`.

3.4 An Application

As an application, let us consider a slightly more complex example called Shiloach-Vishkin connected component algorithm [16], which requires both remote reads and writes in its algorithm description.

In the S-V algorithm, the connectivity information is maintained using the classic disjoint set data structure [4]. Specifically, the data structure is a forest, and vertices in the same tree are regarded as belonging to the same connected component. Each vertex maintains a parent pointer that either points to some other vertex in the same connected component, or points to itself, in which case the vertex is the root of a tree. We henceforth use $D[u]$ to represent this pointer for each vertex u. The S-V algorithm is an iterative algorithm that begins with a forest of n root nodes, and in each step it tries to discover edges connecting different trees and merge the trees together. In a vertex-centric way, every vertex u performs one of the following operations depending on whether its parent $D[u]$ is a root vertex:

- **tree merging:** if $D[u]$ is a root vertex, then u chooses one of its neighbors' current parent (to which we give a name t), and makes $D[u]$ point to t if $t < D[u]$ (to guarantee the correctness of the algorithm). When having multiple choices in choosing the neighbors' parent p, or when different vertices try to modify the same parent vertex's pointer, the algorithm always uses the "minimum" as the tiebreaker for fast convergence.
- **pointer jumping:** if $D[u]$ is not a root vertex, then u modifies its own pointer to its current "grandfather" ($D[u]$'s current pointer). This operation reduces u's distance to the root vertex, and will eventually make u a direct child of the root vertex so that it can perform the above tree merging operation.

The algorithm terminates when all vertices' pointers do not change after an iteration, in which case all vertices point to some root vertex and no more tree merging can be performed. Readers interested in the correctness of this algorithm are referred to the original paper [16] for more details.

Figure 6 shows the implementation of this algorithm in our extended Fregel. The whole program is represented by the expression `giter init ss Fix g`, where `init` is the initialization function that sets the parent pointer to itself at the beginning, and the `ss` is a graph transformation which implements

```
data VD1 = VD1 { p1 :: VRef }
data VD2 = VD2 { p2 :: VRef, p3 :: VRef, gr :: VRef }
sv g = let
         init v = VD1 (getVertexRef v);
         step1 g = let
                     s1 v prev refs =
                       let gr = refs (prev v.^p1).^p1
                         t  = minimum ( (getVertexRef v) :
                                        [(prev w.^p1) | (e, w) <- is v])
                       in VD2 (prev v.^p1) t gr
                   in gmaps s1 g
         step2 g = let
                     s2 v prev refs =
                        let t = minimum ((prev v.^gr) :
                                         [ prev u.^p3
                                         | u <- g,
                                           prev u.^p2 == getVertexRef v,
                                           prev v.^p2 == getVertexRef v])
                        in VD1 t
                   in gmaps s2 g
         ss = step2 . step1
       in giter init ss Fix g
```

図 4 The S-V algorithm in extended Fregel

the main function in the iteration. This program terminates when the graph does not change after an iteration.

Let us first focus on the `ss`, which is composed by two smaller graph transformations `step1` and `step2`, each containing a single logical step. `step1` is firstly performed to collect the id of grandparent as well as all neighbors' parents. The expression `gr = refs (prev v.^p1).^p1` calculates the grandparent: `prev v.^p1` is current vertex v's parent, which is stored as a reference on v; then we fetch the parent node's state via the `refs` table, and further get the parent field `p1`, and bind it to a variable `gr`. According to the algorithm, if a vertex is a root, we need to perform a remote write to modify the parent `pr`'s pointer, but in our implementation we delay the checking and just prefetch the neighbors' minimum parent value, which is achieved by the expression `minimum ( (getVertexRef v) : [(prev w.^p1) | (e, w) <- is v])`. Finally, we store the grand parent, the update value, and the current parent in a temporary data type `VD2`.

Next, we look at `step2`, which exactly follows `step1` and finishs the remote write described in the algorithm. In detail, every root vertex collects those vertices that are trying to write data to itself, and further takes the minimum of those update values as the final vertex state. This computation is exactly encoded in the calculation of variable `t`. Semantically, every vertex uses a list comprehension with generator `u <- g` to look at the whole graph, and uses the filters to find those ones pointing to it (`prev u.^p2 == getVertexRef v`) as well as checks whether the vertex itself is a root vertex (`prev u.^p2 == getVertexRef v`). In the case where v is not a root vertex, the vertex performs the pointer jumping that sets its parent as the grandparent calculated in the previous step.

4 Code Transformation

In this section, we sketch how to transform the major part of our extended Fregel to Pregel code,

based on the fact that Palgol is equipped with remote access capacity with high efficiency, and that we can use Palgol's compiler to generate the Pregel code. The purpose of this transformation is just to verify the feasibility of our extended Fregel, and show its potential to be compiled to efficient Pregel code.

4.1 The Palgol Language

Palgol [18] is a high-level imperative DSL for Pregel, where the vertex-centric graph applications are described in terms of atomic Palgol steps and combinators. A Palgol step is basically a simple graph transformation described in a vertex-centric way, where a vertex can use several pre-defined constructs to fetch data from neighbors or through reference, perform local computation just like an ordinary imperative programming language, generate a new vertex state and/or write to the other vertices. Then, such atomic steps can be composed or iterated, so that control flows including sequences and iterations can be implemented.

Figure 5 presents the essential syntax of Palgol. Regarding its unique remote access capacities, in Palgol we can read a field of an arbitrary vertex using a global field access expression of the form *field* [*exp*], where *field* is a user-specified field name and *exp* should evaluate to a vertex id. Such expression can be further updated by an remote assignment, which consists of a remote field, a value and an "accumulative" assignment (like += and |=). According to Palgol's high-level model, such remote assignment are handled in an additional superstep attached at the end of current step, and the field of the destination vertex is then modified by executing the assignment with the value on its right-hand side.

To give a feeling of the Palgol language, we present the original Palgol code for the S-V algorithm, which is shown in Figure 6. This piece of code contains two steps, where the first one (lines 1–3) performs simple initialization, and the other (lines 5–13) is inside an iteration as the main computation. We also use the field D to store the pointer to the parent vertex. First, line 6 checks whether u's parent is a root, by just writing $D[D[u]] == D[u]$, i.e., whether the pointer of the parent vertex $D[D[u]]$ is equal to the parent's id $D[u]$. If u's parent is a root, we generate a list containing all neighboring vertices' parent id ($D[e.\mathbf{ref}]$), and then bind the minimum one to the variable t (line 7). Now t is either **inf** if the neighbor list is empty or a vertex id; in both cases we can use it to update the parent's pointer (lines 9–10) via a remote assignment. One important thing is that the parent vertex ($D[u]$) may receive many remote writes from its children, where only one of the children providing the minimum t can successfully perform the updating. Here, the statement `a <?= b` is an accumulative assignment, whose meaning is the same as `a := min(a, b)`. Finally, for the **else** branch, we (locally) assign u's grandparent's id to u's D field.

4.2 Compiling Extended Fregel to Palgol

The transformation from our extended Fregel to Palgol is conceptually direct, because of the similarity in the structures of these two languages. In the extended Fregel, a vertex-centric computation is represented by a pure step function that takes a graph as input and produces a new vertex state; these functions are further combined by a set of predefined higher-order functions to implement a complete graph algorithm. Palgol borrows this idea in the language design by letting programmers write atomic vertex-centric computations called Palgol steps, and similarly, we can put them together using two combinators, namely sequence and iteration.

The transformation can be generally divided into two steps. First, we need to recover the dependen-

$$
\begin{array}{lcl}
\mathit{prog} & ::= & \mathit{step} \mid \mathit{prog}_1 \ldots \mathit{prog}_n \mid \mathit{iter} \\
\mathit{iter} & ::= & \textbf{do} \langle\ \mathit{prog}\ \rangle\ \textbf{until fix}\ [\ \mathit{field}_1, \ldots, \mathit{field}_n\ ] \\
 & \mid & \textbf{iter}\ \mathit{var}\ \textbf{in range}\ (\mathit{int},\ \mathit{int})\ \langle\ \mathit{prog}\ \rangle\ \textbf{end} \\
\mathit{step} & ::= & \textbf{for}\ \mathit{var}\ \textbf{in V}\ \langle\ \mathit{block}\ \rangle\ \textbf{end} \\
\mathit{block} & ::= & \mathit{stmt}_1 \ldots \mathit{stmt}_n \\
\mathit{stmt} & ::= & \textbf{if}\ \mathit{exp}\ \langle\ \mathit{block}\ \rangle \mid \textbf{if}\ \mathit{exp}\ \langle\ \mathit{block}\ \rangle\ \textbf{else}\ \langle\ \mathit{block}\ \rangle \\
 & \mid & \textbf{for}\ (\mathit{var} \leftarrow \mathit{exp})\ \langle\ \mathit{block}\ \rangle \\
 & \mid & \textbf{let}\ \mathit{var} = \mathit{exp} \\
 & \mid & \textbf{local}_{opt}\ \mathit{field}\ [\ \mathit{var}\ ]\ \mathit{op}_{local}\ \mathit{exp} \\
 & \mid & \textbf{remote}\ \mathit{field}\ [\ \mathit{exp}\ ]\ \mathit{op}_{remote}\ \mathit{exp} \\
\mathit{exp} & ::= & \mathit{int} \mid \mathit{float} \mid \mathit{var} \mid \textbf{true} \mid \textbf{false} \mid \textbf{inf} \\
 & \mid & \textbf{fst}\ \mathit{exp} \mid \textbf{snd}\ \mathit{exp} \mid (\mathit{exp}, \mathit{exp}) \\
 & \mid & \mathit{exp}.\textbf{ref} \mid \mathit{exp}.\textbf{val} \mid \{\mathit{exp}, \mathit{exp}\} \mid \{\mathit{exp}\} \\
 & \mid & \mathit{exp}\ ?\ \mathit{exp} : \mathit{exp} \mid (\ \mathit{exp}\ ) \mid \mathit{exp}\ \mathit{op}_b\ \mathit{exp} \mid \mathit{op}_u\ \mathit{exp} \\
 & \mid & \mathit{field}\ [\ \mathit{exp}\ ] \\
 & \mid & \mathit{func}_{opt}\ [\ \mathit{exp} \mid \mathit{var} \leftarrow \mathit{exp}, \mathit{exp}_1, \ldots, \mathit{exp}_n\ ] \\
\mathit{func} & ::= & \textbf{maximum} \mid \textbf{minimum} \mid \textbf{sum} \mid \ldots
\end{array}
$$

図 5 Essential part of Palgol's syntax.

```
1  for u in V
2    D[u] := u
3  end
4  do
5    for u in V
6      if (D[D[u]] == D[u])
7        let t = minimum [ D[e.ref]
8                | e <- Nbr[u] ]
9        if (t < D[u])
10         remote D[D[u]] <?= t
11     else
12       D[u] := D[D[u]]
13   end
14 until fix[D]
```

図 6 The S-V algorithm in Palgol

cies of graphs from the higher-order functions, then we transform each step function to a corresponding Palgol step, and combine them in a proper order with Palgol's combinator. Rather than diving into formal definition of the transformation, we shall use the code in Figure 4 to illustrate how the transformation is performed.

Let us starting by condiering the code

```
giter init ss Fix g
```

in Figure 4 which says that, from the original graph g, the *init* function is first executed, then we iteratively execute the graph transformation *ss* until the graph stabilize, which corresponds to Palgol's iteration construct

```
do ..  until fix[..]
```

. We just need to put *init* before the iteration, put *ss* inside the iteration and put all of g's fields inside `fix[..]`, which means the iteration ends when the specified fields stabilize.

Next, according to the code `ss = step2 . step1`, we know that *ss* is actually composed by two smaller steps, while *step*1 and *step*2 cannot be further divides since each of them contains just a step function.

Eventually, we obtain the structure of the Palgol program, which is shown below:

```
[init is compiled here]
do
  [s1 is compiled here]
  [s2 is compiled here]
```

```
until fix[..]
```

The compilation of each step function is performed by converting the communication primitives in the extended Fregel to the corresponding ones in Palgol, and transforming the functional code to equivalent imperative ones. Let us return to the code in Figure 4. For example, the neighborhood access expression in Fregel

```
[(prev w.^p1) | (e, w) <- is v]
```

iterates over the neighboring list and collects their previously calculated `p1` field; in Palgol, the representation is basically the same. It is transformed to Palgol as

```
[ P1[e.ref] | e <- Nbr[u] ]
```

where `Nbr` is the neighboring list, `e.ref` is the reference to a neighboring vertex, and `P1[e.ref]` fetch the value of `P1` field[†2] of the neighboring vertex. Then, the expression for remote read `refs (prev v.^p1).^p1` (which can also be written as `p1 (refs (p1 (prev v)))`) is transformed to a Palgol expression `P1[P1[u]]`, which follows the reference to parent (`P1` field) twice to get the grandparent. In Fregel, local access and remote reads use two different tables `prev` and `refs`, but Palgol just has a unified interface `P1[..]` for both local and remote field access.

One tricky transformation is caused by the remote access in our extended Fregel, and an example is the following expression:

```
[ prev u.^p3 | u <- g,
  prev u.^p2 == getVertexRef v,
  prev v.^p2 == getVertexRef v ]
```

Briefly speaking, a vertex is now collecting data from all vertices in the graph, which is usually not so realistic in real graph applications, but from the filter `prev u.^p2 == getVertexRef v` we can recover the real intension: u is going to perform a remote write to v and stores a link in its `P2` field in the vertex. Having recognized such intension, we naturally transform this expression by two steps. In the first step, all vertices write a value (which is `P3` according to the Fregel code) to vertex `P2`, and the remote write is made to a temporary field `T` with a `minimum` operation which can be found in the Fregel program. Then in the second step, all vertices just read the `T` field to collect the result of remote write, and then move on to the consequent computation.

Finally, we get the following transformed result for the S-V program. Although it looks different from the Palgol's original implementation of the S-V algorithm, it actually does the same job.

```
for u in V
  let t = minimum [ e.ref | e <- Nbr[u] ]
  P1[u] := (u < t ? u : t)
end
do
  for u in V
    let t = minimum [ P1[e.ref]
          | e <- Nbr[u] ]
    P3[u] := (u < t ? u : t)
    Gr[u] := P1[P1[u]]
    P2[u] := P1[u]
  end
  for u in V
    T[u] := inf
    if (P2[u] == Gr[u])
      remote T[P2[u]] <?= P3[u]
  end
  for u in V
    P1[u] := (Gr[u] < T[u] ? Gr[u] : T[u])
  end
until fix[P1]
```

It is worth noting that despite the similarity in these two languages, there are several differences between them which should be carefully taken into consideration when designing the transformation algorithm. Here, we summarize them and give an idea of how they can be solved.

- Dependency: Palgol program has a clear execution order, while the extended Fregel can specify the computation in a declarative way, for example, the `gzip` function combines two

†2 in Palgol, field name should be capital letter

graphs together without saying which one evaluates first. Therefore, dependency check should be perform to generate a valid structure of the Palgol program.

- Vertex fields: Palgol is an imperative language that exposes all vertex fields to the programmers at the beginning, and a vertex can manipulate arbitrary vertex fields in any part of the program. In contrast, Fregel has a functional model where each step function can only access the fields of the input graph feed to it. To fill this gap, we should calculate a mapping from the graph in Fregel to a set of corresponding fields in Palgol in advance.

It should also be noted that there is still a few constructs in Fregel that cannot be directly translated to Palgol, such as the termination condition `Until`. Nevertheless, this is mainly due to the limitation in Palgol's high level combinator and can be easily fixed by hand-written some code. Based on the transformation algorithm described in this subsection as well as Palgol's own compiling algorithm [17], it is our future work to directly compile our extended Fregel to Pregel code.

5 Related Work

Google's Pregel [10] proposed the vertex-centric computing paradigm, which allows programmers to think naturally like a vertex when designing distributed graph algorithms. Some graph-centric (or block-centric) systems like Giraph++ [13] and Blogel [14] extends Pregel's vertex-centric approach by making the partitioning mechanism open to programmers, but it is still unclear how to optimize general vertex-centric algorithms (especially those complicated ones containing non-trivial communication patterns) using such extension.

Domain-Specific Languages (DSLs) are a well-known mechanism for describing solutions in specialized domains. To ease Pregel programming, many DSLs have been proposed, such as Palovca [8], s6raph [11], Fregel [3] and Green-Marl [7]. We briefly introduce each of them below.

Green-Marl [6] is a DSL that allows programmers to describe a graph algorithm at a higher level. Originally proposed for graph processing on shared-memory multi-processors, it is extended [7] to support Pregel systems, where the compilation to Pregel relies on discovering " Pregel-canonical " patterns. Since it does not have a Pregel-specific language design, programmers may easily get compilation errors if they are not familiar with the implementation of the compiler. Besides, there is still a wide range of Pregel algorithms that cannot be implemented or efficiently expressed in Green-Marl, due to the limitation in expressing remote access.

Fregel [3] is a functional DSL for declarative programming on big graphs. In Fregel, a vertex-centric computation is represented by a pure function that takes a graph as input and produces a new vertex state; such functions can then be composed using a set of predefined combinators to implement a complete graph algorithm. This approach not only prevents programmers from writing invalid programs, but also helps people to better understand the behavior of a vertex-centric program. However, due to the same reason as Green-Marl, it is also limited in expressiveness.

S6raph [11] is a special graph processing framework with a functional interface. It models a particular type of iterative svertex-centric computation by six functions, which are specified by programmers to implement different algorithms. Compared to Green-Marl and Fregel, s6raph can only represent graph algorithms that contain a single iterative computing (such as PageRank and Shortest Path), while many practical Pregel algorithms are far more complicated. In exchange, s6raph programs can be written concisely and executed efficiently.

Palgol [17] is an imperative DSL which is inte-

grated with the remote access capacities. Similar to Fregel, a Palgol program is composed by vertex programs and high-level combinators, but a vertex program in Palgol contains an additional remote write phase, which allows a vertex to send the updates to other vertices. In addition, it proposes a new abstraction, *chain access*, for representing fetching data from a reference, which is useful for a class of Pregel algorithms.

6 Conclusion

In this paper, we report our work-in-progress to extend Fregel with remote access to enhance its expressiveness. The basic idea is to enrich vertex attributes with reference and to treat the reference as the first class value during each iteration. We formally define the semantics of this extension, demonstrate a nontrivial application, and informally show how this extension can be efficiently compiled to Pregel though Palgol. We are now working on the implementation and will give more practical evaluation in the future.

参 考 文 献

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

[2] Chung, S. and Condon, A.: Parallel implementation of Borůvka's minimum spanning tree algorithm, *IPPS*, IEEE, 1996, pp. 302–308.

[3] Emoto, K., Matsuzaki, K., Morihata, A., and Zhenjiang, H.: Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing, *ICFP*, ACM, 2016, pp. 200–213.

[4] Gabow, H. N. and Tarjan, R. E.: A linear-time algorithm for a special case of disjoint set union, *J. Comput. System Sci.*, Vol. 30, No. 2(1985), pp. 209–221.

[5] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs, *OSDI*, 2012, pp. 17–30.

[6] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis, *ASPLOS*, ACM, 2012, pp. 349–362.

[7] Hong, S., Salihoglu, S., Widom, J., and Olukotun, K.: Simplifying scalable graph processing with a domain-specific language, *CGO*, ACM, 2014, pp. 208.

[8] Lesniak, M.: Palovca: describing and executing graph algorithms in Haskell, *PADL*, Springer, 2012, pp. 153–167.

[9] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud, *PVLDB*, Vol. 5, No. 8(2012), pp. 716–727.

[10] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G.: Pregel: a system for large-scale graph processing, *SIGMOD*, ACM, 2010, pp. 135–146.

[11] Ruiz, O. C., Matsuzaki, K., and Sato, S.: s6raph: vertex-centric graph processing framework with functional interface, *FHPC*, ACM, 2016, pp. 58–64.

[12] Salihoglu, S. and Widom, J.: GPS: a graph processing system, *SSDBM*, No. 22, ACM, 2013.

[13] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J.: From think like a vertex to think like a graph, *PVLDB*, Vol. 7, No. 3(2013), pp. 193–204.

[14] Yan, D., Cheng, J., Lu, Y., and Ng, W.: Blogel: A block-centric framework for distributed computation on real-world graphs, *PVLDB*, Vol. 7, No. 14(2014), pp. 1981–1992.

[15] Yan, D., Cheng, J., Lu, Y., and Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation, *WWW*, ACM, 2015, pp. 1307–1317.

[16] Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., and Bu, Y.: Pregel algorithms for graph connectivity problems with performance guarantees, *PVLDB*, Vol. 7, No. 14(2014), pp. 1821–1832.

[17] Zhang, Y., Ko, H.-S., and Hu, Z.: Palgol: A High-Level DSL for Vertex-Centric Graph Processing with Remote Data Access, *Asian Symposium on Programming Languages and Systems*, 2017.

[18] Zhang, Y., Ko, H.-S., and Hu, Z.: Palgol: A High-Level DSL for Vertex-Centric Graph Processing with Remote Data Access, *15th Asian Symposium on Programming Languages and Systems*, LNCS, Spinger, 2019.