Describing Pregel Algorithms with Non-adjacent Data Communication

Yongzhe Zhang Hsiang-Shang Ko Zhenjiang Hu

Pregel is a popular parallel computing model for dealing with large-scale graphs. However, it can be tricky to implement traditional graph algorithms correctly and efficiently in Pregel's vertex-centric model, as the programmer needs to carefully restructure an algorithm in terms of supersteps and message passing, which are low-level and detached from the algorithm description. There exist domain-specific languages that hide message passing behind the scenes, and let the programmer express, at a higher level, their intention of fetching data from adjacent vertices. But only allowing communication between neighbors is insufficient to express a still wide range of graph algorithms. In this paper, we design a new domain-specific language for Pregel that allows the programmer to fetch or modify data on non-adjacent vertices. We give several examples to show that this feature allows many graph algorithms to be expressed more naturally and concisely.

1 Introduction

The rapid increase of graph data in real world calls for efficient analysis on massive graphs. However, graph computation is in general difficult to parallelize or scale, due to the inherent interdependencies in graph data. Google's Pregel [11] is a framework proposed to process such large graphs, which runs on shared nothing architecture and implements the bulk-synchronous parallel (BSP) model [14] to handle large-scale graphs using the *vertex-centric* computing paradigm. In Pregel, computation on graphs is split into supersteps, in each of which the vertices run in parallel performing some local computation, and data communication takes place between supersteps. Pregel is equipped with a failure recovery mechanism, which is important for programs running in a distributed environment.

Although Pregel is well-suited for parallelism and have good scalability, implementing graph algorithms in Pregel is challenging. As summarized in several papers [3] [6], the following reasons make programming in Pregel inconvenient to programmers:

- Explicit state control: In Pregel, programmers write a function *compute()* that describes the local computation performed on the vertices. Usually that function needs to refer to a global state to decide what part of the computation should be performed, and global communication is needed to perform state transition when necessary.
- Explicit message passing: The utilization of message passing between vertices is hard to understand, because it is hard to see where the message is from and how the message is consumed without searching through the whole

Yongzhe Zhang, Zhenjiang Hu, 総合研究大学院大学複合科学研究科, School of Multidisciplinary Sciences, SOKENDAI.

Hsiang-Shang Ko, 国立情報学研究所, National Institute of Informatics.

compute() function. In addition, Pregel only allows a single message type, which usually contains different contents for different communication purposes, and is hard to comprehend as well.

- Manually arranged supersteps: In Pregel, which is a specialization of the Bulk-Synchronous Parallel model, one conceptual step in a high-level algorithm description may have to be arranged manually into multiple supersteps, which can be drastically different from the original conceptual step and obscure the logic.

It is inconvenient and error-prone to let programmers keep all those detailed issues in mind when implementing a graph algorithm in Pregel. Furthermore, having lots of low-level details in the code will greatly hurt the readability of the program, and make the program hard to maintain when some changes need to be made at the algorithm level.

Several domain-specific languages (DSLs) have been proposed to provide more intuitive ways of implementing graph algorithms in Pregel, such as Green-Marl [7] and Fregel [3]. These DSLs all aim to partially solve the problems listed above. Specifically, they aim to simplify programming in Pregel by allowing programmers to write the program compositionally to avoid explicit state control, hide explicit message passing by data fetching, generate supersteps according to data dependencies, and automatically produce the message type.

These two DSLs still have several severe restrictions, however: They only allow data fetching along the edges in a graph, so that only a simple "flipping" from data fetching to message sending is needed when compiling their programs to Pregel. But in general, a step in an algorithm may need to fetch data from arbitrary vertices. Then, there is no way for a vertex to modify the state of an arbitrary vertex, which is also a common pattern in graph computing. These two restrictions preclude a wide range of Pregel algorithms from being implemented in those DSLs.

As an example, let us consider the Shiloach-Vishkin Practical Pregel Algorithm (S-V PPA) [15], which calculates the connected components of an undirected graph in a logarithmic number of supersteps. This algorithm builds the connectivity information using the disjoint-set data structure [4]. Specifically, the data structure is a forest, and the vertices in the same tree are regarded as belonging to the same connected component. Each vertex maintains a 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. S-V PPA 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 merges the trees together. The description of the algorithm is shown in Figure 1.

To implement this algorithm in Pregel, one challenge is to correctly translate the intention of fetching or modifying non-local data into message passing and arrange the supersteps. For example, one specific intention is for u to compare its parent's ID and pointer to know whether its parent is a root node. Before doing the comparison, u's parent should send its pointer to u, but it is u that maintains the parent-child relationship, so u's parent actually does not know to which vertex to send that message. Therefore, u should first send a request to u's parent, and then in the next superstep, u's parent replies to all its children. In the third superstep, u can finally do the comparison to check whether its parent is a root node or not. We see that this intention, despite its simplicity, has to be translated into three supersteps containing a query-reply conversation between each vertex and its parent, and it is not until the last superstep that the comparison can be done.

```
Input: an undirected graph
Output: connected components represented
        as disjoint sets
set all vertices as root nodes;
repeat
    for each vertex u do
        if u's parent is a root node then
            choose a neighbor v whose parent's
            ID is smaller than u's parent's ID;
            if v exists, let u's parent point to v's
            parent;
        else
            let u point to its own grandparent;
        end
    end
until the disjoint-set structure does not
change;
```

図 1 S-V PPA algorithm description

Unfortunately, such pattern actually cannot be implemented in the DSLs mentioned above. The reason is that S-V PPA maintains a dynamic internal data structure that is orthogonal to the original graph and requires communication along this internal graph, but in those DSLs, there is no interface for directly sending messages to other vertices, and the only way to fetch non-local data is restricted between neighbors in the original graph. For the same reason, the intention of modifying u's parent's pointer in the *else* clause cannot be implemented in those DSLs either, since u and u's parent are not neighbors in the original graph, but modifying non-local data relies on message passing as well. From this example, we see that the existing DSLs are not expressive enough for implementing still a wide range of algorithms.

To address these problems, we first propose in this paper a high-level model for vertex-centric computing on Pregel systems. During each step in this model, each vertex can fetch local or non-local data, perform local computation, and modify local or non-local state. This notion is more high-level than Pregel supersteps, in which the vertices have to explicitly handle the messages, and is more natural for describing Pregel algorithms. Then, we design a new DSL to allow programmers to express algorithms in our model, and sketch a transformation from our DSL to Pregel. The concise syntax of our DSL makes it easy to implement more complicated graph algorithms. The main contributions of this work are:

- We introduce the concept of *algorithmic superstep*, which is more natural for describing Pregel algorithms at a higher level.
- We design a DSL that allows programmers to describe graph algorithms in terms of algorithmic supersteps, and provide concise syntax for fetching or modifying non-local state, improving on the expressiveness of the existing DSLs.
- We give a representative example to illustrate how to use the DSL and sketch a transformation from our DSL into Pregel.

The organization of the rest of the paper is as follows. Section 2 gives a brief introduction of Pregel, and Section 3 introduce our high-level model and the DSL. Then, Section 4 sketches the transformation from our DSL to Pregel. Section 5 discusses related work, and Section 6 concludes the paper.

2 Background

We give a brief overview of the Pregel system [11] here.

2.1 The Computation Model

In Pregel, the input is in general a directed graph, and each vertex is associated with a mutable user-defined state. The computation is split into supersteps, and terminates when all vertices are inactive in a particular superstep. A vertex is active at the

beginning of execution, but can deactivate itself by "voting to halt". When inactive, it can only be activated by receiving external messages.

Within each superstep, the vertices compute in parallel, each executing the same user-defined function. A vertex can read the messages sent to it in the previous superstep, modify its own state, and send messages to other vertices. Computation assumes the Bulk-Synchronous Parallel (BSP) model [14], where global barrier synchronization happens at the end of each superstep.

Vertices can communicate with each other by sending messages, each of which contains the ID of the destination vertex and the message value. If a vertex sends a message in superstep n, that message will reach the destination vertex in superstep $n+1$. In general, a vertex can send any number of messages to any other vertex, but there is no guaranteed order among the messages. In Pregel, only a single message type is allowed, so programmers should encode different types in one if they need to pass multiple types of messages.

In addition, Pregel provides *aggregators*, a mechanism for global communication. Every vertex can provide a value to an aggregator in superstep n; the system then combines these values by a reduction operator and makes the result available at the beginning of superstep $n+1$. Pregel has a bunch of predefined aggregators for various types, like *max*, *min* for numeric values, but programmers may also define their own aggregators by providing the default value and a reduction operator. Whether build-in or user-defined, aggregation operators should be commutative and associative.

2.2 Pregel-like Systems

There are a bunch of open-source alternative to the official and proprietary Pregel system, such as Apache Hama [2], Apache Giraph [1], Catch the Wind [13], GPS [12], GraphLab [10], PowerGraph [5] and Mizan [8]. This paper does not target a specific Pregel-like system. Instead, our DSL can be transformed to any Pregel-like system, as long as the system supports message passing between arbitrary pairs of vertices, and provides the mechanism of aggregators. In Section 4, we use physical supersteps, message passing and aggregators to represent a Pregel program, and sketch how to transform our DSL to this internal representation.

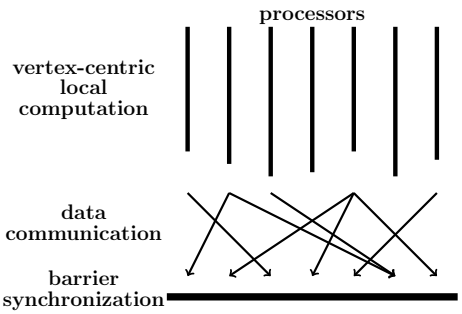

図 2 The Pregel computation model

3 The Domain-Specific Language

In this section, we introduce the concept of *algorithmic superstep*, and our domain-specific language for describing Pregel algorithms.

3.1 The Algorithmic Superstep

In Pregel systems, one particular difficulty for programmers is to write vertex-centric programs in terms of supersteps and message passing. Existing DSLs have already shown that thinking in terms of data fetching is more natural for Pregel programmers, because that allows algorithms to be expressed as more compact Pregel programs, from which the necessary supersteps and message passing can be automatically derived. However, it is also important to have the concept of superstep in the algorithm level, otherwise it is hard to see whether an algorithm can be properly parallelized in Pregel. Therefore, the design of our DSL tries to

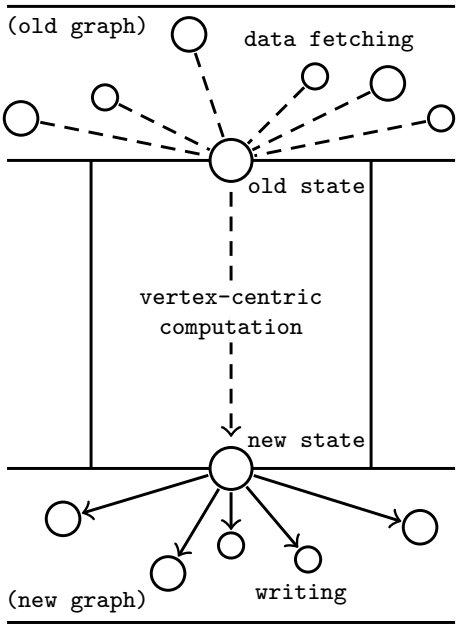

図 3 Algorithmic superstep

give a proper abstraction of superstep, to let programmers ignore unnecessary low-level details and retain the ability to reason about the algorithm.

We model the basic unit of vertex-centric computation as an *algorithmic superstep*, which contains the following three phases (illustrated in Figure 3):

- A local computation (LC) phase that produces local results from the state of the current graph. The computation can fetch data from any other vertices, either adjacent or non-adjacent.
- A local update (LU) phase that replaces the local states of the current vertex with new values computed by the local computation phase.
- A remote update (RU) phase that modifies the states of other vertices.

The LU and RU phases are optional, because a vertex may not need to change local and remote states, but at least one of the two phases should be specified in an algorithmic superstep.

The notion of algorithmic supersteps is better for describing algorithms for the following reasons. First, it tries to describe the most general intention of data fetching, that is, to get a field of another vertex whose ID is known. In this way, many computations can be expressed by more naturally and concisely. Second, it allows a vertex to update other vertices at the end of a superstep, which is also a common and necessary pattern of communication in graph algorithms. Third, it clearly separates a superstep into three phases, which gives programmers a standard template to describe the algorithms. The well-controlled side effects and the high-level semantics make algorithmic supersteps easier to be transformed into efficient Pregel code.

Recall S-V PPA in Figure 1, which we can clearly describe in algorithmic supersteps. First, setting all vertices as root nodes is simply a superstep that modifies every vertex's pointer to its own ID. Second, within the iteration there is a single superstep, in which a vertex fetches the pointers of its parent and all its neighbors, and then modifies its own pointer or the pointers of other vertices according to the checking performed locally on that vertex.

3. 2 Basic Vertex-Centric Programs

From now on, we propose a domain-specific language (DSL) for writing algorithmic supersteps concretely. In our DSL, an algorithmic superstep is described by a *basic vertex-centric program*, which always starts with **for** (*var* **in V**), followed by a code block containing a list of statements, where *var* is a freely chosen name for the current vertex. The syntax of basic vertex-centric programs is shown in Figure 4.

This simple language contains the basic elements in a normal imperative programming language, such as assignment, branching, loop, and arithmetic operations. It also supports the basic data types and data structures including integers, booleans, floating-point numbers, pairs, and lists.

int	=	integer
float	=	floating-point number
var	=	identifier starting with lowercase letter
field	=	identifier starting with capital letter
vcprog	::=	**for** (*var* **in V**) *block*
block	::=	$stmt_1 \ldots stmt_n$ (with same indentation)
stmt	::=	**if** *exp block* **else** *block*
	|	**for** (*var* ← *exp*) *block*
	|	*var* op_{assign} *exp*
	|	*local* | *remote* | *agg*
exp	::=	*int* | *float* | *var* | **true** | **false**
	|	*pair* | *list* | **fst** *exp* | **snd** *exp*
	|	*var*.**Id** | *var*.**W** (edge ID and weight)
	|	*exp* op_{binary} *exp* | (*exp*)
	|	(*exp* ? *exp* : *exp*)
	|	op_{accum} *list* | *fetch*
pair	::=	(*exp*, *exp*)
list	::=	[$exp_1, \ldots, exp_n$]
	|	[*exp* | *var* ← *exp*, $exp_1, \ldots, exp_n$]
fetch	::=	*field*(*exp*)
local	::=	**new** [*field*] op_{local} *exp*
remote	::=	**update** [*fetch*] op_{remote} *exp*
agg	::=	**agg** [*var*] op_{agg} *exp*

図 4 Syntax of basic vertex-centric program

There are four additional special constructs listed at the end of Figure 4, whose semantics are closely related to Pregel: the snapshot operator, the local update statement, the remote update statement, and the aggregator.

Snapshot Operator. The snapshot operator is to fetch data from local or non-local vertices, and the value is the result of the previous logic superstep. The syntax is simply *field*(*exp*), where the field name is an identifier starting with a capital letter, and the expression specifies a vertex ID. This operator returns the value of the specified field of some vertex, as the result of the previous algorithmic superstep.

```
for (u in V)
  t := A(u) + B(C(u))
```

In this example, suppose that A, B, and C are fields, and that C stores the ID of another vertex. This piece of code calculates the sum of the current vertex's A field and vertex $C(u)$'s B field, and then stores the result in the temporary variable t of the current vertex.

Basically, fetching non-local data will introduce additional communication with other vertices, and with proper restrictions on fetching patterns, it can be automatically transformed into Pregel. The transformation will be discussed in Section 4.

Local Update. A local update statement (rule *local* in Figure 4) is to update a local field by some expression, where the operators are listed in Table 1. At first glance, the local update statement looks similar to an assignment, but in our DSL it is actually different. Recall the definition of algorithmic supersteps, in which the local update always happens after all local computation is completed. To follow this specification, the local update statement can be seen as a deferred operation, which expresses the intention of local data modification, and takes effect after all local computation is finished. Local update statements can be put anywhere in the program. Since our DSL is imperative, the final result depends on the actual execution. For example:

表 1 Local Update Operators

Name	Syntax	Description
simple assignment	`a := b`	$a \leftarrow b$
ADD assignment	`a += b`	$a \leftarrow a + b$
SUB assignment	`a -= b`	$a \leftarrow a - b$
MUL assignment	`a *= b`	$a \leftarrow a \times b$
DIV assignment	`a /= b`	$a \leftarrow a \,/\, b$
AND assignment	`a &&= b`	$a \leftarrow a \wedge b$
OR assignment	`a ||= b`	$a \leftarrow a \vee b$
MAX assignment	`a >?= b`	$a \leftarrow max(a, b)$
MIN assignment	`a <?= b`	$a \leftarrow min(a, b)$

```
for (u in V)
  new[A] := B(u)
  new[B] := A(u)
```

This program swaps the values of fields A and B

of the current vertex, so B should be updated with current $A(u)$ and vice versa. In a traditional programming language, we usually have to use a temporary variable to achieve the swapping, but in our DSL, we only need to write two local update statements, since local update is deferred.

Remote Update. The remote update statement is to update a non-local field by some expression. The state to be modified is represented by a snapshot operator, which specifies the field name and the vertex, and the available assignment operators are listed in Table 1 without the simple assignment.

The remote update statement (rule *remote* in Figure 4) is also deferred. It expresses the intention of remote update and takes effect after all local computation and update are performed. More precisely, an additional physical superstep is added for remote update, since communication is needed. However, updating a local vertex and a remote vertex are essentially different, not only because of the message passing, but also about the meaning of the operation. When we modify a local field, we just use a new value to replace the original one; however, when we try to update a field of another vertex, potentially there may be other vertices trying to do the same. Eventually, remote update should be implemented by message passing (e.g., by sending the new value and the operation to another vertex), so in the updating phase, a vertex may receive more than one update requests, which is different from the local modification. To make programmers aware of this situation, we forbid a remote update statement to be a normal assignment. Instead, programmers should consider how to produce a final result from the original value and a (possibly empty) list of new values.

Unlike the imperative local update statements, we do not specify the order of performing remote updates, since Pregel does not guarantee that messages are ordered in some specific way. Although it is possible for the receiver to order the update requests by adding an additional value in the message specifying the priority of execution, the price is increasing the message size as well as having additional local computation. Besides, by investigating various practical Pregel algorithms, we have not seen the necessity of keeping the order of remote updates.

Since the order of remote updates is not specified, as a consequence, it is not reasonable to use different operators in a basic vertex-centric program to update a particular remote field.

Aggregator. Pregel aggregators are a mechanism for global communication, in which every vertex provides a value, before the Pregel system collects those values and use some reduction operator to produce a final value. Then, this result will be broadcast to all vertices in the graph, and will be available in the next physical superstep.

Such mechanism is provided in our DSL. In a basic vertex-centric program, programmers can use the rule *agg* in Figure 4, where each vertex evaluates the expression to a value, and the system reduces those values by a given operator. Currently, we support all the default aggregators [11] in Pregel, including *maximum*, *minimum*, *sum*, *and*, and *or*. In addition, we provide a new aggregator *random*, which randomly selects an element according to a programmer-specified probability. In this case, each vertex provides a pair (v_i, w_i) with $w_i \geq 0$, and the probability that v_i is selected as the final result is $w_i/(\sum_{i=0}^{n-1} w_i)$.

3.3 Concatenation and Iteration

In the previous subsection, we have presented the design of basic vertex-centric programs, with which we can access local or non-local data, do pure local computation, and modify the fields on related vertices. In this subsection, we introduce how to write

prog	::=	*vcprog*
	|	$prog_1 \ldots prog_n$ (with same indentation)
	|	**do** *exp* **until** *term*
term	::=	**iter** *int*
	|	**fix** [$field_1, \ldots, field_n$]
	|	*exp* (no local variables)

図 5 Syntax of concatenation and iteration

sophisticated algorithms using two combinators on basic vertex-centric programs: concatenation and iteration.

Concatenation combines two vertex-centric programs into one sequentially, so that the second program can take the result of the first program as input. Iteration, on the other hand, runs a vertex-centric program repeatedly until some termination condition is satisfied. In our DSL, programmers can enclose a vertex-centric program with **do** ... **until** *term*, and three types of termination conditions can be used:

1. Iterate a fixed number of supersteps.
2. Iterate until the values of some fields do not change anymore.
3. Iterate until an expression evaluates to true, where the expression should not refer to local variables and fields.

3.4 Example: Shiloach-Vishkin Practical Pregel Algorithm

Here, we give the complete implementation of SV-PPA (Figure 1):

```
for (u in V)
  new[D] := u
do
  for (u in V)
    if D(D(u)) == D(u)
      update[D(D(u))] <?= minimum[ D(e.Id)
          | e <- Nbr(u), D(e.Id) < D(u) ]
    else
      new[D] := D(D(u))
until fix[D]
```

In the first two lines, we write an initialization program which assigns each vertices' own ID to its D field, which stores the pointer. Next is an iteration containing a single basic vertex-centric program, which tries to find new neighboring pair of vertices that are not in the same tree yet. In this program, we first see whether u's parent is a root, by checking the condition `D(D(u)) == D(u)`. If this condition is satisfied, it fetches all its neighbors' parents and chooses the ones whose ID is smaller than u's parent, which is represented by the list comprehension. Then, the build-in function *minimum* further picks out the one with the smallest ID among them and updates its parent $D(u)$'s D field. The operator used here is `<?=`, which means if the provided value is smaller than $D(D(u))$'s original value, then perform the assignment. For the other case, where u's parent is not a root, we just update the local field D with its grandparent. The algorithm terminates when all vertices' pointers no longer change.

4 Transformation to Pregel

In this section, we sketch how to transform our DSL to physical supersteps in Pregel. We first consider the transformation of basic vertex-centric programs in Section 4.1, and then concatenation and iteration in Section 4.2. Finally, in Section 4.3, we show concretely how the SV-PPA is transformed to Pregel.

4.1 Basic Vertex-Centric Program

Let us first consider how a basic vertex-centric program can be translated into a sequence of physical supersteps. In general, a basic vertex-centric program will be transformed into several supersteps for data fetching, one superstep for main computing and local state updating, and then one superstep for remote updating. The main challenge of the transformation lies in the data fetching phase, where some other vertices may be involved and play the role of data provider, and some data communi-

cation is needed; this will take most of this subsection to explain.

Data fetching. In the DSL, data fetching is specified by the snapshot operator, in which the source vertex and the field are specified. The strategy is a query-reply conversation, where a vertex first sends a message to the source to acquire the field, and then the source vertex replies the value of that field. For example, suppose that the expression has the form *field*(*exp*), and that the current vertex u (potentially recursively) evaluates *exp* to a vertex ID x in superstep n, and that the field we need is A. Then, u sends a message to vertex x in superstep n including its own ID, and attaches a unique label l_1 to that message. At the beginning of superstep $n+1$, all vertices scan the messages, and then reply each request having label l_1 with the value of its own field A to the sender whose ID is included in that request, and also attaches a unique label l_2 to the reply. In superstep $n+2$, u scans the messages and find the one with label l_2, which should contain the value of *field*(*exp*).

When the snapshot operator contains a loop variable (from either a *for* statement or a list comprehension), the situation becomes slightly more complicated. In our DSL, a restriction is that a snapshot may only contain a loop variable from a top-level *for* statement or list comprehension; otherwise the operation is considered invalid. One reason for this choice is that, a use of the snapshot operator that contains multiple loop variables is very expensive, which may cause a large number of messages passing through the system, and may be also space consuming. If, however, we only consider the snapshot operator in a loop containing no inner loops, then the translation is easy. Since the loop is represented by iterating over a certain list (either a *for* loop or list comprehension), we first evaluate the list as l, and then in superstep n, we evaluate the *exp* and create a loop to send the request to the vertex whose ID is the result of evaluating *exp*. In superstep $n+1$, every vertex replies to all requests. Next, in superstep $n+2$, every vertex gets the results, and then recreates the results for further computation.

There is a special case where l is actually a list of edges in the graph, and by evaluate *exp* we find that we need to fetch some field alone the edges. In this case, we do not need to send request to all neighbors. Instead, we just send the vertex's field directly to neighbors, which saves one superstep.

In the transformation procedure above, we have only shown how to transform each snapshot into several steps of query-reply conversation, but finally we should combine them together, so that different data fetching procedures can actually happen in parallel, and the code in some supersteps should be merged. When to perform each data fetching procedure is decided by a dependency check, while combining code in supersteps for fetching is always safe, because all communications use different channels by attaching unique labels to messages. Since we separate a single program into multiple supersteps, some local variables may be referred to by the code in different supersteps. In this case, we should scan the program one more time, and change such kind of local variables into local fields, so that they can be stored on the vertex.

Local and remote updates. Finally, transforming the local update and remote update is trivial. In general, we should store the new value into the vertex state, and then add an assignment in the local update superstep after all local computation is done. Then, a remote update statement should be translated to a message sending containing the ID of the destination vertex and the value. Since a remote update statement in our DSL does not necessary happen just before the remote update superstep, in which case we should also keep the information in the vertex state, and send the

messages before the remote update phase. Finally, in the remote update phase, the only thing to do is reading the update requests, and update the corresponding fields of the vertex.

4.2 Concatenation and Iteration

In the previous subsection, we have shown how to transform a basic vertex-centric program into physical supersteps. Here, we introduce the transformation rules for concatenation and iteration, and try to derive a complete Pregel program with as small number of supersteps as possible.

A basic vertex-centric program may possibly be transformed into several sending supersteps, one computing superstep, and possibly one remote update superstep. The sending supersteps are pure in the sense that no local state would change; the single computing superstep may update the local state; and in the remote update superstep, a vertex only scans the messages and updates its own state. In our transformation for concatenation and iteration, we use a slightly different representation, in which a program may possibly have the first sending superstep, followed by other sending supersteps and the computation superstep, and an optional remote update superstep. For simplicity, we represent the program in a concise way like $[FS, SC, RU]$, where FS is the first sending superstep, SC is the other sending supersteps along with the local computation superstep, and RU is the remote update superstep. Since FS and RU are optional, we simply write $\emptyset$ when they are omitted. The following equation shows the concatenation rules for combining two programs:

$$
\begin{aligned}
& concat([\ FS_1,\ SC_1,\ \emptyset\ ],\ [\ \emptyset,\ SC_2,\ RU_2\ ]) \\
= \ & [\ FS_1,\ [\ SC_1 +\!\!+\ SC_2\ ],\ RU_2\ ] \\
& concat([\ FS_1,\ SC_1,\ \emptyset\ ],\ [\ FS_2,\ SC_2,\ RU_2\ ]) \\
= \ & [\ FS_1,\ [\ SC_1 +\!\!+ FS_2,\ SC_2\ ],\ RU_2\ ] \\
& concat([\ FS_1,\ SC_1,\ RU_1\ ],\ [\ \emptyset,\ SC_2,\ RU_2\ ]) \\
= \ & [\ FS_1,\ [\ SC_1, RU_1 +\!\!+ SC_2\ ],\ RU_2\ ] \\
& concat([\ FS_1,\ SC_1,\ RU_1\ ],\ [\ FS_2,\ SC_2,\ RU_2\ ]) \\
= \ & [\ FS_1,\ [\ SC_1, RU_1 +\!\!+ FS_2,\ SC_2\ ],\ RU_2\ ]
\end{aligned}
$$

In these equations, the $+\!\!+$ operator combines two groups of supersteps such that the last superstep in the first group and the first superstep in the second group are merged together. The resulting program also contains the three parts. It is obvious that the rules defined above are correct, because the order of supersteps in those two programs are kept in the combined program, and the order of the two programs are also kept.

Secondly, we show the transformation rules for iteration with three different types of termination conditions. Each type of termination rule can be easily translated to a 4-tuple (I, T, J, E), where I is an initialization superstep, T is a superstep containing termination check, J is to generate global information about whether to exit the iteration and jump to superstep T, and E is the exit superstep doing nothing. The transformation rules for iteration is as follows:

$$
\begin{aligned}
& iterate((I,\ T,\ J,\ E),\ [\ FS,\ SC,\ RU\ ]) \\
= \ & [\ \emptyset,\ [\ FS +\!\!+ I,\ T +\!\!+ SC,\ RU +\!\!+ FS +\!\!+ J\ ],\ E\ ] \\
& iterate((I,\ T,\ J,\ E),\ [\ FS,\ SC,\ \emptyset\ ]) \\
= \ & [\ \emptyset,\ [\ FS +\!\!+ I,\ T +\!\!+ SC +\!\!+ FS +\!\!+ J\ ],\ E\ ] \\
& iterate((I,\ T,\ J,\ E),\ [\ \emptyset,\ SC,\ RU\ ]) \\
= \ & [\ \emptyset,\ [\ I,\ T +\!\!+ RU +\!\!+ SC +\!\!+ J\ ],\ E\ ] \\
& iterate((I,\ T,\ J,\ E),\ [\ \emptyset,\ SC,\ \emptyset\ ]) \\
= \ & [\ \emptyset,\ [\ I,\ T +\!\!+ SC +\!\!+ J\ ],\ E\ ]
\end{aligned}
$$

Now let us see how the three types of termination rules can be translated to an (I, T, J, E) tuple. First is the condition of iterating a fixed number of times. This requires an additional field as the counter. The initialization step I initializes the counter to 0, and the termination check step T reads the counter, and goes to superstep J (by changing the global state) if the counter is greater than a pre-determined value. In superstep J, we just increase the counter by one. Second is the condition of iterating a program until some specified fields remain unchanged during the iteration. In this case, we need to trace all involved fields,

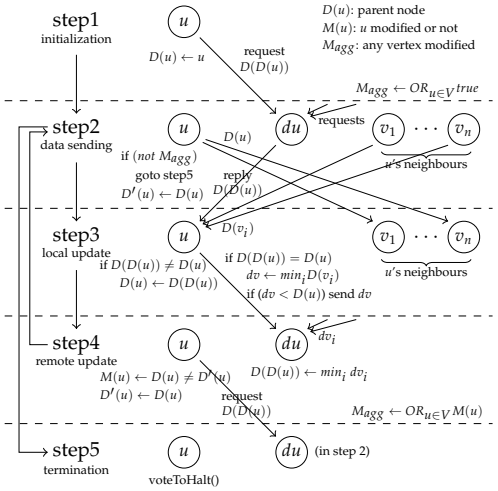

図 6 S-V PPA transformation result

so we insert code to keep a copy of old values in superstep T which is prior to any state modification, so that in the end of iteration J, every vertex knows whether its own state is modified during the iteration. Besides, in superstep J, the vertices use global communication (aggregator in Pregel) to obtain a global variable indicating whether any of the vertices is modified in an iteration, so that in superstep T, the vertices look at this variable to decide whether to exit the iteration. Third is the termination condition using a general expression that contains no local variable. In this case, we simply evaluate the expression in superstep J, and then perform checking in T.

4.3 Example: Shiloach-Vishkin Practical Pregel Algorithm

Now let us see concretely how S-V PPA is translated to Pregel's physical supersteps. We visualize the transformation result in Figure 6.

First of all is the initialization step, in which all vertices set their pointers (D field) to their own IDs. Then, M is an additional boolean field, which is used to trace whether D changes in the following iteration. In the iteration, we can see that the main computation happens in Step 3, and before that, all data fetching can be done. There are two non-local expressions that are required by u's computation: the parent's pointer $D(D(u))$ and some neighbor's pointer $D(e.Id)$. The parent's pointer is a simple pattern that fetches data from a single source $D(u)$, but the fetching of neighbor's pointer is inside a list comprehension, so they are transformed differently. For $D(D(u))$, u sends the request to $D(u)$ in Step 1, and the parent replies in Step 2, so that all vertices can get $D(D(u))$ in Step 3. Then, for $D(e.Id)$ in the list comprehension expression, we can see that it is passed along the edges, according to the generator $e \leftarrow \mathrm{Nbr}(u)$. To translate this pattern, we simply let every vertex send their $D(u)$ to all their neighbors in Step 2, so that in Step 3, every vertex will receive $D(v_i)$ from all its neighbors $\{v_1, \ldots, v_n\}$. After all data have been sent, the computation is performed in Step 3: Every vertex first check whether its parent is a root node, by checking the equation $D(D(u)) = D(u)$. Then, it either performs a local update in Step 3, or perform a remote update, which happens at the beginning of Step 4. Next, in Step 4, we trace the D field to see whether it changes in the iteration. We use an aggregator to generate a global value M_{agg} indicating whether there is any modification, so we jump to Step 2, check the value of M_{agg}, and decide whether to exit the iteration. Finally we reach Step 5, where we just invoke the function voteToHalt().

In this figure, each intention of data fetching is transformed separately, but finally they can be easily put together, since there is no vertex state change in the data fetching phase, and the message types can be distinguished by adding an additional label. In implementation, we just combine the code in each superstep, handle the messages properly, and perform the state transition at the end of the superstep.

5 Related Work

There are lots of DSLs proposed for big graph processing on Pregel. Fregel [3] is a functional language for writing Pregel algorithms, in which the vertex-centric computation is represented by pure functions returning new vertex state. However, it cannot access data on non-adjacent vertices, or modify the state of other vertices. Green-Marl [6] is an imperative language for shared-memory multiprocessor system, and can be compiled to Pregel [7], but the compilation relies on discovering Pregel-canonical pattern, which means that programmers may easily get compilation errors if they are not familiar with the implementation of the compiler. It also has the same expressiveness problem as Fregel. Palovca [9] is an embedded domain specific-language using Haskell as the underlying host language for Pregel algorithms. It uses a monad to hide side effects and define vertex-centric program in a low-level way like in typical Pregel systems.

6 Conclusion

This paper introduces the concept of algorithmic superstep, a high-level model for Pregel computation, and presents a novel domain-specific language that allows programmers to implement Pregel algorithms in a more natural way. Our DSL has high expressiveness that can represent complicated Pregel algorithms concisely, and the high-level semantics makes it much easier to be transformed to Pregel with small number of supersteps. As future work, we are interested in the further optimization of reducing the number of supersteps and the size of internal data, to produce more efficient Pregel programs.

Acknowledgment We thank Dr. Kento Emoto (from Kyushu Institute of Technology) for his insightful comments to improve this work.

参 考 文 献

[1] : Apache Giraph, http://giraph.apache.org/.
[2] : Apache Hama, http://hama.apache.org/.
[3] Emoto, K., Matsuzaki, K., et al.: Think like a vertex, behave like a function!, *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, ACM, 2016.
[4] Gabow, H. N. and Tarjan, R. E.: A linear-time algorithm for a special case of disjoint set union, *Journal of computer and system sciences*, Vol. 30, No. 2(1985), pp. 209–221.
[5] Gonzalez, J. E., Low, Y., et al.: Powergraph: distributed graph-parallel computation on natural graphs, *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012.
[6] Hong, S., Chafi, H., et al.: Green-Marl: a DSL for easy and efficient graph analysis, *ACM SIGARCH Computer Architecture News*, ACM, 2012, pp. 349–362.
[7] Hong, S., Salihoglu, S., Widom, J., and Olukotun, K.: Simplifying scalable graph processing with a domain-specific language, *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ACM, 2014, pp. 208.
[8] Khayyat, Z., Awara, K., et al.: Mizan: a system for dynamic load balancing in large-scale graph processing, *Proceedings of the ACM European Conference on Computer Systems*, 2013.
[9] Lesniak, M.: Palovca: describing and executing graph algorithms in haskell, *International Symposium on Practical Aspects of Declarative Languages*, 2012.
[10] Low, Y., Bickson, D., et al.: Distributed GraphLab: a framework for machine learning and data mining in the cloud, *Proceedings of the VLDB Endowment*, Vol. 5, No. 8(2012), pp. 716–727.
[11] Malewicz, G., Austern, M. H., et al.: Pregel: a system for large-scale graph processing, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.
[12] Salihoglu, S. and Widom, J.: GPS: a graph processing system, *Proceedings of the International Conference on Scientific and Statistical Database Management*.
[13] Shang, Z. and Yu, J. X.: Catch the Wind: graph workload balancing on cloud, *Proceedings of the International Conference on Data Engineering*, 2013.
[14] Valiant, L. G.: A bridging model for parallel computation, *Communications of the ACM*, (1990), pp. 103–111.
[15] Yan, D., Cheng, et al.: Pregel algorithms for graph connectivity problems with performance guarantees, *Proceedings of the VLDB Endowment*, (2014), pp. 1821–1832.