Palgol: A High-Level DSL for Vertex-Centric Graph Processing with Remote Data Access

Yongzhe Zhang[1,2], Hsiang-Shang Ko[2], and Zhenjiang Hu[1,2]

[1] Department of Informantics, SOKENDAI
Shonan Village, Hayama, Kanagawa 240-0193 Japan
[2] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan
{zyz915,hsiang-shang,hu}@nii.ac.jp

Abstract. Pregel is a popular distributed computing model for dealing with large-scale graphs. However, it can be tricky to implement graph algorithms correctly and efficiently in Pregel's vertex-centric model, especially when the algorithm has multiple computation stages, complicated data dependencies, or even communication over dynamic internal data structures. Some domain-specific languages (DSLs) have been proposed to provide more intuitive ways to implement graph algorithms, but due to the lack of support for *remote access* — reading or writing attributes of other vertices through references — they cannot handle the above mentioned dynamic communication, causing a class of Pregel algorithms with fast convergence impossible to implement.

To address this problem, we design and implement Palgol, a more declarative and powerful DSL which supports remote access. In particular, programmers can use a more declarative syntax called *chain access* to naturally specify dynamic communication as if directly reading data on arbitrary remote vertices. By analyzing the logic patterns of chain access, we provide a novel algorithm for compiling Palgol programs to efficient Pregel code. We demonstrate the power of Palgol by using it to implement several practical Pregel algorithms, and the evaluation result shows that the efficiency of Palgol is comparable with that of hand-written code.

1 Introduction

The rapid increase of graph data calls for efficient analysis on massive graphs. Google's Pregel [9] is one of the most popular frameworks for processing large-scale graphs. It is based on the bulk-synchronous parallel (BSP) model [14], and adopts the *vertex-centric* computing paradigm to achieve high parallelism and scalability. Following the BSP model, a Pregel computation is split into *supersteps* mediated by *message passing*. Within each superstep, all the vertices execute the same user-defined function *compute()* in parallel, where each vertex can read the messages sent to it in the previous superstep, modify its own state, and send messages to other vertices. Global barrier synchronization happens at the end of each superstep, delivering messages to their designated receivers

before the next superstep. Despite its simplicity, Pregel has demonstrated its usefulness in implementing many interesting graph algorithms [9,10,12,15,17].

Despite the power of Pregel, it is a big challenge to implement a graph algorithm correctly and efficiently in it [17], especially when the algorithm consists of multiple stages and complicated data dependencies. For such algorithms, programmers need to write an exceedingly complicated *compute()* function as the loop body, 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. Some attempts have been made to ease Pregel programming by proposing domain-specific languages (DSLs), such as Green-Marl [7] and Fregel [2]. These DSLs allow programmers to write a program in a compositional way to avoid writing a complicated loop body, and provide neighboring data access to avoid explicit message passing. Furthermore, programs written in these DSLs can be automatically translated to Pregel by fusing the components in the programs into a single loop, and mapping neighboring data access into message passing. However, for efficient implementation, the existing DSLs impose a severe restriction on data access — each vertex can only access data on their neighboring vertices. In other words, they do not support general *remote data access* — reading or writing attributes of other vertices through references.

Remote data access is, however, important for describing a class of Pregel algorithms that aim to accelerate information propagation (which is a crucial issue in handling graphs with large diameters [17]) by maintaining a dynamic internal structure for communication. For instance, a parallel pointer jumping algorithm maintains a tree (or list) structure in a distributed manner by letting each vertex store a reference to its current parent (or predecessor), and during the computation, every vertex constantly exchanges data with the current parent (or predecessor) and modifies the reference to reach the root vertex (or the head of the list). Such computational patterns can be found in algorithms like the Shiloach-Vishkin connected component algorithm [17] (see Section 2.3 for more details), the list ranking algorithm (see Section 2.4) and Chung and Condon's minimum spanning forest (MSF) algorithm [1]. However, these computational patterns cannot be implemented with only neighboring access, and therefore cannot be expressed in any of the existing high-level DSLs.

It is, in fact, hard to equip DSLs with efficient remote reading. First, when translated into Pregel's message passing model, remote reads require multiple rounds of communication to exchange information between the reading vertex and the remote vertex, and it is not obvious how the communication cost can be minimized. Second, remote reads would introduce more involved data dependencies, making it difficult to fuse program components into a single loop. Things become more complicated when there is *chain access*, where a remote vertex is reached by following a series of references. Furthermore, it is even harder to equip DSLs with remote writes in addition to remote reads. For example, Green-Marl detects read/write conflicts, which complicate its programming model; Fregel has a simpler functional model, which, however, cannot support remote writing

without major extension. A more careful design is required to make remote reads and writes efficient and friendly to programmers.

In this paper, we propose a more powerful DSL called Palgol[3] that supports remote data access. In more detail:

- We propose a new high-level model for vertex-centric computation, where the concept of *algorithmic supersteps* is introduced as the basic computation unit for constructing vertex-centric computation in such a way that remote reads and writes are ordered in a safe way.
- Based on the new model, we design and implement Palgol, a more declarative and powerful DSL, which supports both remote reads and writes, and allows programmers to use a more declarative syntax called *chain access* to directly read data on remote vertices. For efficient compilation from Palgol to Pregel, we develop a logic system to compile chain access to efficient message passing where the number of supersteps is reduced whenever possible.
- We demonstrate the power of Palgol by working on a set of representative examples, including the Shiloach-Vishkin connected component algorithm and the list ranking algorithm, which use communication over dynamic data structures to achieve fast convergence.
- The result of our evaluation is encouraging. The efficiency of Palgol is comparable with hand-written code for many representative graph algorithms on practical big graphs, where execution time varies from a 2.53% speedup to a 6.42% slowdown in ordinary cases, while the worst case is less than a 30% slowdown.

The rest of the paper is organized as follows. Section 2 introduces algorithmic supersteps and the essential parts of Palgol, Section 3 presents the compiling algorithm, and Section 4 presents evaluation results. Related work is discussed in Section 5, and Section 6 concludes this paper with some outlook.

2 The Palgol Language

This section first introduces a high-level vertex-centric programming model (Section 2.1), in which an algorithm is decomposed into atomic vertex-centric computations and high-level combinators, and a vertex can access the entire graph through the references it stores locally. Next we define the Palgol language based on this model, and explain its syntax and semantics (Section 2.2). Finally we use two representative examples — the Shiloach-Vishkin connected component algorithm (Section 2.3) and the list ranking algorithm (Section 2.4) — to demonstrate how Palgol can concisely describe vertex-centric algorithms with dynamic internal structures using remote access.

[3] Palgol stands for **P**regel **algo**rithmic **l**anguage. The system with all implementation code and test examples is available at `https://bitbucket.org/zyz915/palgol`.

2.1 The High-Level Model

The high-level model we propose uses remote reads and writes instead of message passing to allow programmers to describe vertex-centric computation more intuitively. Moreover, the model remains close to the Pregel computation model, in particular keeping the vertex-centric paradigm and barrier synchronization, making it possible to automatically derive a valid and efficient Pregel implementation from an algorithm description in this model, and in particular arrange remote reads and writes without data conflicts.

In our high-level model, the computation is constructed from some basic components which we call *algorithmic supersteps*. An algorithmic superstep is a piece of vertex-centric computation which takes a graph containing a set of vertices with local states as input, and outputs the same set of vertices with new states. Using algorithmic supersteps as basic building blocks, two high-level operations *sequence* and *iteration* can be used to glue them together to describe more complex vertex-centric algorithms that are iterative and/or consist of multiple computation stages: the *sequence* operation concatenates two algorithmic supersteps by taking the result of the first step as the input of the second one, and the *iteration* operation repeats a piece of vertex-centric computation until some termination condition is satisfied.

The distinguishing feature of algorithmic supersteps is remote access. Within each algorithmic superstep (illustrated in Figure 1), all vertices compute in parallel, performing the same computation specified by programmers. A vertex can read the fields of any vertex in the input graph; it can also write to arbitrary vertices to modify their fields, but the writes are performed on a separate graph rather than the input graph (so there are no read-write conflicts). We further distinguish *local writes* and *remote writes* in our model: local writes can only modify the current vertex's state, and are first performed on an intermediate graph (which is initially a copy of the input graph); next, remote writes are propagated to the destination vertices to further modify their intermediate states. Here, a remote write consists of a remote field, a value and an "accumulative" assignment (like `+=` and `|=`), and that field of the destination vertex is modified by executing the assignment with the value on its right-hand side. We choose to support only accumulative assignments so that the order of performing remote writes does not matter.

More precisely, an algorithmic superstep is divided into two phases:

- a *local computation* (LC) phase, in which a copy of the input graph is created as the intermediate graph, and then each vertex can read the state of any vertex in the input graph, perform local computation, and modify its own state in the intermediate graph, and
- a *remote updating* (RU) phase, in which each vertex can modify the states of any vertices in the intermediate graph by sending remote writes. After all remote writes are processed, the intermediate graph is returned as the output graph.

Among these two phases, the RU phase is optional, in which case the intermediate graph produced by the LC phase is used directly as the final result.

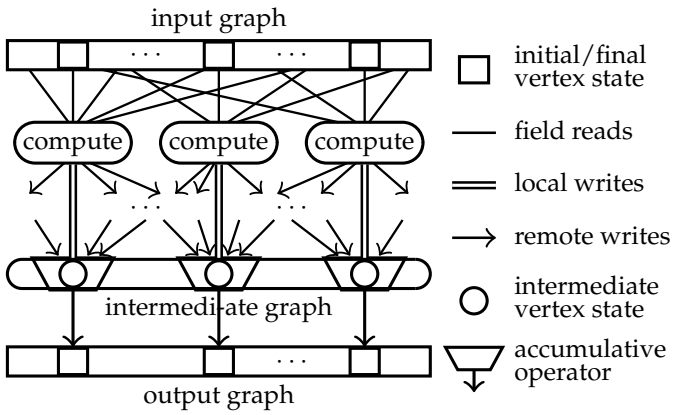

Fig. 1. In an algorithmic superstep, every vertex performs local computation (including field reads and local writes) and remote updating in order.

2.2 An Overview of Palgol

Next we present Palgol, whose design follows the high-level model we introduced above. Figure 2 shows the essential part of Palgol's syntax. As described by the syntactic category *step*, an algorithmic superstep in Palgol is a code block enclosed by "**for** *var* **in V**" and "**end**", where *var* is a variable name that can be used in the code block for referring to the current vertex (and **V** stands for the set of vertices of the input graph). Such steps can then be composed (by sequencing) or iterated until a termination condition is met (by enclosing them in "**do**" and "**until** ..."). Palgol supports several kinds of termination condition, but in this paper we focus on only one kind of termination condition called *fixed point*, since it is extensively used in many algorithms. The semantics of fixed-point iteration is iteratively running the program enclosed by **do** and **until**, until the specified fields stabilize.

Corresponding to an algorithmic superstep's remote access capabilities, 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 updated by local or remote assignments, where an assignment to a remote vertex should always be accumulative and prefixed with the keyword **remote**. One more thing about remote assignments is that they take effect only in the RU phase (after the LC phase), regardless of where they occur in the program.

There are some predefined fields that have special meaning in our language. **Nbr** is the edge list in undirected graphs, and **In** and **Out** respectively store incoming and outgoing edges for directed graphs. Essentially, these are normal fields of a predefined type for representing edges, and most importantly, the compiler assumes a form of symmetry on these fields (namely that every edge is stored consistently on both of its end vertices), and uses the symmetry to produce more efficient code.

$$
\begin{array}{lll}
\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 \ ] \\
\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}_{\mathit{opt}} \ \mathit{field} \ [\ \mathit{var} \ ] \ \mathit{op}_{\mathit{local}} \ \mathit{exp} \qquad \text{– local write} \\
 & \mid & \textbf{remote} \ \mathit{field} \ [\ \mathit{exp} \ ] \ \mathit{op}_{\mathit{remote}} \ \mathit{exp} \qquad \text{– remote write} \\
\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}\} \qquad \text{– specialized pair} \\
 & \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} \ ] \qquad \text{– global field access} \\
 & \mid & \mathit{func}_{\mathit{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}
$$

Fig. 2. Essential part of Palgol's syntax. Palgol is indentation-based, and two special tokens '⟨' and '⟩' are introduced to delimit indented blocks.

The rest of the syntax for Palgol steps is similar to an ordinary programming language. Particularly, we introduce a specialized pair type (expressions in the form of $\{exp, exp\}$) for representing a reference with its corresponding value (e.g., an edge in a graph), and use **.ref** and **.val** respectively to access the reference and the value respectively, to make the code easy to read. Some functional programming constructs are also used here, like let-binding and list comprehension. There is also a foreign function interface that allows programmers to invoke functions written in a general-purpose language, but we omit the detail from the paper.

2.3 The Shiloach-Vishkin Connected Component Algorithm

Here is our first representative Palgol example: the *Shiloach-Vishkin (S-V) connected component algorithm* [17], which can be expressed as the Palgol program in Figure 3. A traditional HashMin connected component algorithm [17] based on neighborhood communication takes time proportional to the input graph's diameter, which can be large in real-world graphs. In contrast, the S-V algorithm can calculate the connected components of an undirected graph in a logarithmic number of supersteps; to achieve this fast convergence, the capability of accessing data on non-neighboring vertices is essential.

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 [17] for more details.

The implementation of this algorithm is complicated, which contains roughly 120 lines of code[4] for the *compute()* function alone. Even for detecting whether the parent vertex $D[u]$ is a root vertex for each vertex u, it has to be translated into three supersteps containing a query-reply conversation between each vertex and its parent. In contrast, the Palgol program in Figure 3 can describe this algorithm concisely in 13 lines, due to the declarative remote access syntax. This piece of code contains two steps, where the first one (lines 1–3) performs simple initialization, and the other (lines 5–12) is inside an iteration as the main computation. We also use the field D to store the pointer to the parent vertex. Let us focus on line 6, which checks whether u's parent is a root. Here we simply check $D[D[u]] == D[u]$, i.e., whether the pointer of the parent vertex $D[D[u]]$ is equal to the parent's id $D[u]$. This expression is completely declarative, in the sense that we only specify what data is needed and what computation we want to perform, instead of explicitly implementing the message passing scheme.

The rest of the algorithm can be straightforwardly associated with the Palgol program. 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 8–9) 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

[4] `http://www.cse.cuhk.edu.hk/pregelplus/code/apps/basic/svplus.zip`

```
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] | e <- Nbr[u] ]
8        if (t < D[u])
9          remote D[D[u]] <?= t
10     else
11       D[u] := D[D[u]]
12   end
13 until fix[D]
```

Fig. 3. The S-V algorithm in Palgol

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.

2.4 The List Ranking Algorithm

Another example is the *list ranking* algorithm, which also needs communication over a dynamic structure during computation. Consider a linked list L with n elements, where each element u stores a value $val(u)$ and a link to its predecessor $pred(u)$. At the head of L is a virtual element v such that $pred(v) = v$ and $val(v) = 0$. For each element u in L, define $sum(u)$ to be the sum of the values of all the elements from u to the head (following the predecessor links). The list ranking problem is to compute $sum(u)$ for each element u. If $val(u) = 1$ for every vertex u in L, then $sum(u)$ is simply the rank of u in the list. List ranking can be solved using a typical pointer-jumping algorithm in parallel computing with a strong performance guarantee. Yan et al. [17] demonstrated how to compute the pre-ordering numbers for all vertices in a tree in $O(\log n)$ supersteps using this algorithm, as an internal step to compute bi-connected components (BCC).[5]

We give the Palgol implementation of list ranking in Figure 4 (which is a 10-line program, whereas the Pregel implementation[6] contains around 60 lines of code). $Sum[u]$ is initially set to $Val[u]$ for every u at line 2; inside the fixed-point iteration (lines 5–9), every u moves $Pred[u]$ toward the head of the list and updates $Sum[u]$ to maintain the invariant that $Sum[u]$ stores the sum of a sublist from itself to the successor of $Pred[u]$. Line 6 checks whether u points to the

[5] BCC is a complicated algorithm, whose efficient implementation requires constructing an intermediate graph, which is currently beyond Palgol's capabilities. Palgol is powerful enough to express the rest of the algorithm, however.

[6] `http://www.cse.cuhk.edu.hk/pregelplus/code/apps/basic/bcc.zip`

```
1  for u in V
2    Sum[u] := Val[u]
3  end
4  do
5    for u in V
6      if (Pred[Pred[u]] != Pred[u])
7        Sum[u] += Sum[Pred[u]]
8        Pred[u] := Pred[Pred[u]]
9    end
10 until fix[Pred]
```

Fig. 4. The list ranking program

virtual head of the list, which is achieved by checking $Pred[Pred[u]] == Pred[u]$, i.e., whether the current predecessor $Pred[u]$ points to itself. If the current predecessor is not the head, we add the sum of the sublist maintained in $Pred[u]$ to the current vertex u, by reading $Pred[u]$'s *Sum* and *Pred* fields and modifying u's own fields accordingly. Note that since all the reads are performed on a snapshot of the input graph and the assignments are performed on an intermediate graph, there is no need to worry about data dependencies.

3 Compiling Palgol to Pregel

In this section, we present the compiling algorithm to transform Palgol to Pregel. The task overall is complicated and highly technical, but the most challenging problem is how to translate chain access (like $D[D[u]]$) into Pregel's message passing model. We describe the compilation of chain access in Section 3.1, and then the compilation of a Palgol step in Section 3.2, and finally how to combine Palgol steps using sequence and iteration in Section 3.3.

3.1 Compiling Remote Reads

Our compiler currently recognizes two forms of remote reads. The first form is *chain access* expressions like $D[D[u]]$. The second form is *neighborhood access* where a vertex may use chain access to acquire data from *all* its neighbors, and this can be described using the list comprehension (e.g., line 7 in Figure 3) or for-loop syntax in Palgol. The combination of these two remote read patterns is already sufficient to express quite a wide range of practical Pregel algorithms. Here we only present the compilation of chain access, which is novel, while the compilation of neighborhood access is similar to what has been done in Fregel.

Definition and challenge of compiling: A chain access is a consecutive field access expression starting from the current vertex. As an example, supposing that the current vertex is u, and D is a field for storing a vertex id, then $D[D[u]]$ is a chain access expression, and so is $D[D[D[D[u]]]]$ (which we abbreviate to

$D^4[u]$ in the rest of this section). Generally speaking, there is no limitation on the depth of a chain access or the number of fields involved in the chain access.

As a simple example of the compilation, to evaluate $D[D[u]]$ on every vertex u, a straightforward scheme is a request-reply conversation which takes two rounds of communication: in the first superstep, every vertex u sends a request to (the vertex whose id is) $D[u]$ and the request message should contain u's own id; then in the second superstep, those vertices receiving the requests should extract the sender's ids from the messages, and reply its D field to them.

When the depth of such chain access increases, it is no longer trivial to find an efficient scheme, where efficiency is measured in terms of the number of supersteps taken. For example, to evaluate $D^4[u]$ on every vertex u, a simple query-reply method takes six rounds of communication by evaluating $D^2[u]$, $D^3[u]$ and $D^4[u]$ in turn, each taking two rounds, but the evaluation can actually be done in only three rounds with our compilation algorithm, which is not based on request-reply conversations.

Logic system for compiling chain access: The key insight leading to our compilation algorithm is that we should consider not only the expression to evaluate but also the vertex on which the expression is evaluated. To use a slightly more formal notation (inspired by Halpern and Moses [5]), we write $\forall u.\, \mathrm{K}_{v(u)}\, e(u)$, where $v(u)$ and $e(u)$ are chain access expressions starting from u, to describe the state where every vertex $v(u)$ "knows" the value of the expression $e(u)$; then the goal of the evaluation of $D^4[u]$ can be described as $\forall u.\, \mathrm{K}_u\, D^4[u]$. Having introduced the notation, the problem can now be treated from a logical perspective, where we aim to search for a derivation of a target proposition from a few axioms.

There are three axioms in our logic system:

1. $\forall u.\, \mathrm{K}_u\, u$
2. $\forall u.\, \mathrm{K}_u\, D[u]$
3. $(\forall u.\, \mathrm{K}_{w(u)}\, e(u)) \wedge (\forall u.\, \mathrm{K}_{w(u)}\, v(u)) \implies \forall u.\, \mathrm{K}_{v(u)}\, e(u)$

The first axiom says that every vertex knows its own id, and the second axiom says every vertex can directly access its local field D. The third axiom encodes message passing: if we want every vertex $v(u)$ to know the value of the expression $e(u)$, then it suffices to find an intermediate vertex $w(u)$ which knows both the value of $e(u)$ and the id of $v(u)$, and thus can send the value to $v(u)$. As an example, Figure 5 shows the solution generated by our algorithm to solve $\forall u.\, \mathrm{K}_u\, D^4[u]$, where each line is an instance of the message passing axiom.

Figure 6 is a direct interpretation of the implications in Figure 5. To reach $\forall u.\, \mathrm{K}_u\, D^4[u]$, only three rounds of communication are needed. Each solid arrow represents an invocation of the message passing axiom in Figure 5, and the dashed arrows represent two logical inferences, one from $\forall u.\, \mathrm{K}_u\, D[u]$ to $\forall u.\, \mathrm{K}_{D[u]}\, D^2[u]$ and the other from $\forall u.\, \mathrm{K}_u\, D^2[u]$ to $\forall u.\, \mathrm{K}_{D^2[u]}\, D^4[u]$.

The derivation of $\forall u.\, \mathrm{K}_u\, D^4[u]$ is not unique, and there are derivations that correspond to inefficient solutions — for example, there is also a derivation for the

$$
\begin{aligned}
&(\forall u.\, \mathrm{K}_{u} \quad u \quad) \wedge (\forall u.\, \mathrm{K}_{u} \quad D[u]\ ) \implies \forall u.\, \mathrm{K}_{D[u]}\ u \\
&(\forall u.\, \mathrm{K}_{D[u]}\ u \quad) \wedge (\forall u.\, \mathrm{K}_{D[u]}\ D^2[u]) \implies \forall u.\, \mathrm{K}_{D^2[u]}\ u \\
&(\forall u.\, \mathrm{K}_{D[u]}\ D^2[u]) \wedge (\forall u.\, \mathrm{K}_{D[u]}\ u \quad) \implies \forall u.\, \mathrm{K}_{u} \quad D^2[u] \\
&(\forall u.\, \mathrm{K}_{D^2[u]}\ D^4[u]) \wedge (\forall u.\, \mathrm{K}_{D^2[u]}\ u \quad) \implies \forall u.\, \mathrm{K}_{u} \quad D^4[u]
\end{aligned}
$$

Fig. 5. A derivation of $\forall u.\, \mathrm{K}_u\ D^4[u]$

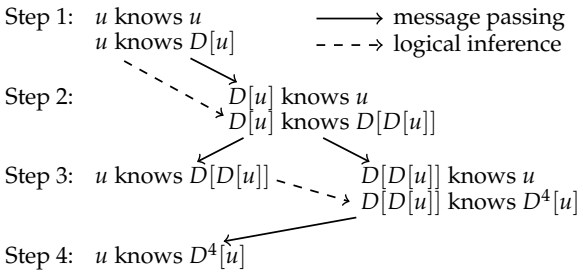

Fig. 6. Interpretation of the derivation of $\forall u.\, \mathrm{K}_u\ D^4[u]$

six-round solution based on request-reply conversations. However, when searching for derivations, our algorithm will minimize the number of rounds of communication, as explained below.

The compiling algorithm: Initially, the algorithm sets as its target a proposition $\forall u.\, \mathrm{K}_{v(u)}\ e(u)$, for which a derivation is to be found. The key problem here is to choose a proper $w(u)$ so that, by applying the message passing axiom backwards, we can get two potentially simpler new target propositions $\forall u.\, \mathrm{K}_{w(u)}\ e(u)$ and $\forall u.\, \mathrm{K}_{w(u)}\ v(u)$ and solve them respectively. The range of such choices is in general unbounded, but our algorithm considers only those simpler than $v(u)$ or $e(u)$. More formally, we say that a is a *subpattern* of b, written $a \preceq b$, exactly when b is a chain access starting from a. For example, u and $D[u]$ are subpatterns of $D[D[u]]$, while they are all subpatterns of $D^3[u]$. The range of intermediate vertices we consider is then $\mathrm{Sub}(e(u), v(u))$, where Sub is defined by

$$\mathrm{Sub}(a, b) = \{\, c \mid c \preceq a \text{ or } c \prec b \,\}$$

We can further simplify the new target propositions with the following function before solving them:

$$\mathit{generalize}(\forall u.\, \mathrm{K}_{a(u)}\ b(u)) = \begin{cases} \forall u.\, \mathrm{K}_{u}\ (b(u)/a(u)) & \text{if } a(u) \preceq b(u) \\ \forall u.\, \mathrm{K}_{a(u)}\ b(u) & \text{otherwise} \end{cases}$$

where $b(u)/a(u)$ denotes the result of replacing the innermost $a(u)$ in $b(u)$ with u. (For example, $A[B[C[u]]]/C[u] = A[B[u]]$.) This is justified because the orig-

inal proposition can be instantiated from the new proposition. (For example, $\forall u.\, \mathrm{K}_{C[u]}\, A[B[C[u]]]$ can be instantiated from $\forall u.\, \mathrm{K}_u\, A[B[u]]$.)

It is now possible to find an optimal solution with respect to the following inductively defined function *step*, which calculates the number of rounds of communication for a proposition:

$$\begin{aligned}
\mathit{step}(\forall u.\, \mathrm{K}_u\, u) &= 0 \\
\mathit{step}(\forall u.\, \mathrm{K}_u\, D[u]) &= 0 \\
\mathit{step}(\forall u.\, \mathrm{K}_{v(u)}\, e(u)) &= 1 + \min_{w(u) \in \mathrm{Sub}(e(u), v(u))} \max(x, y) \\
&\quad \text{where } x = \mathit{step}(\mathit{generalize}(\forall u.\, \mathrm{K}_{w(u)}\, e(u))) \\
&\qquad\quad\; y = \mathit{step}(\mathit{generalize}(\forall u.\, \mathrm{K}_{w(u)}\, v(u)))
\end{aligned}$$

It is straightforward to see that this is an optimization problem with optimal and overlapping substructure, which we can solve efficiently with memoization techniques.

With this compiling algorithm, we are now able to handle any chain access expressions. Furthermore, this algorithm optimizes the generated Pregel program in two aspects. First, this algorithm derives a message passing scheme with a minimum number of supersteps, thus reduces unnecessary cost for launching Pregel supersteps during execution. Second, by extending the memoization technique, we can ensure that a chain access expression will be evaluated exactly once even if it appears multiple times in a Palgol step, avoiding redundant message passing for the same value.

3.2 Compiling Palgol Steps

Having introduced the compiling algorithm for remote data reads in Palgol, here we give a general picture of the compilation for a single Palgol step, as shown in Figure 7. The computational content of every Palgol step is compiled into a *main superstep*. Depending on whether there are remote reads and writes, there may be a number of *remote reading supersteps* before the main superstep, and a *remote updating superstep* after the main superstep.

We will use the main computation step of the S-V program (lines 5–12 in Figure 3) as an illustrative example for explaining the compilation algorithm, which consists of the following four steps:

1. We first handle neighborhood access, which requires a sending superstep that provides all the remote data for the loops from the neighbors' perspective. This sending superstep is inserted as a remote reading superstep immediately before the main superstep.
2. We analyze the chain access expressions appearing in the Palgol step with the algorithm in Section 3.1, and corresponding remote reading supersteps are inserted in the front. (For the S-V algorithm, the only interesting chain access expression is $D[D[u]]$, which induces two remote reading supersteps realizing a request-reply conversation.)

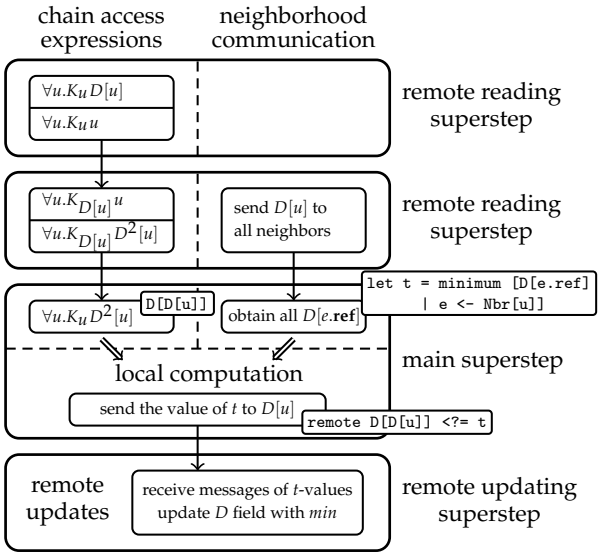

Fig. 7. Compiling a Palgol step to Pregel supersteps.

3. Having handled all remote reads, the main superstep receives all the values needed and proceeds with the local computation. Since the local computational content of a Palgol step is similar to an ordinary programming language, the transformation is straightforward.
4. What remain to be handled are the remote assignments, which require sending the updating values as messages to the target vertices in the main superstep. Then an additional remote updating superstep is added after the main superstep; this additional superstep reads these messages and updates each field using the corresponding remote updating operator.

3.3 Compiling Sequences and Iterations

Finally, we look at the compilation of sequence and iteration, which assemble Palgol steps into larger programs. A Pregel program generated from Palgol code is essentially a *state transition machine* (STM) combined with computation code for each state. Every Palgol step is translated into a "linear" STM consisting of a chain of states corresponding to the supersteps like those shown in Figure 7, and the compilation of a Palgol program starts from turning the atomic Palgol steps into linear STMs, and implements the sequence and iteration semantics to construct more complex STMs.

Compilation of sequence: To compile the sequence, we first compile the two component programs into STMs, then a composite STM is constructed by simply adding a state transition from the end state of the first STM to the start state of the second STM.

Compilation of iteration: We first compile the loop body into an STM, which starts from some state S_{start} and ends in a state S_{end}, then we extend

this STM to implement the fixed-point semantics. Here we describe a generalized approach which generates a new STM starting from state $S_{start'}$ and ending in state $S_{end'}$:

1. First, a check of the termination condition takes place right before the state S_{start}: if it holds, we immediately enters a new exit state $S_{exit'}$; otherwise we execute the body, after which we go back to the check by adding a state transition from S_{end} to S_{start}. This step actually implements a while loop.
2. The termination check is implemented by an OR aggregator to make sure that every vertex makes the same decision: basically, all vertices determine whether their local fields stabilize during a single iteration by storing the original values beforehand, and the aggregator combines the results and makes it available to all vertices.
3. We add a new start state $S_{start'}$ and make it directly transit to S_{start}. This state is for storing the original values of the fields, and also to make the termination check succeed in the first run, turning the while loop into a do-until loop.

Optimizations: In the compilation of sequence and iteration, two optimization techniques are used to reduce the number of states in the generated STMs and can remove unnecessary synchronizations. Due to space restrictions, we will not present all the details here, but these techniques share similar ideas with Green-Marl's "state merging" and "intra-loop state merging" optimizations [7]:

- *state merging*: whenever it is safe to do so, the Green-Marl compiler merges two consecutive states of vertex computation into one. In the compilation of sequence in Palgol, we can always safely merge the end state of the first STM and the start state of the second STM, resulting in a reduction of one state in the composite STM.
- *intra-loop state merging*: this optimization merges the first and last vertex-parallel states inside Green-Marl's loops. In Palgol, we can also discover such chance when iterating a linear STM inside a fixed-point iteration.

4 Experiments

In this section, we evaluate the overall performance of Palgol and the state-merging optimisations introduced in the previous section. We compile Palgol code to Pregel+[7], which is an open-source implementation of Pregel written in C++.[8] We have implemented the following six graph algorithms on Pregel+'s basic mode, which are:

[7] `http://www.cse.cuhk.edu.hk/pregelplus`

[8] Palgol does not target a specific Pregel-like system. Instead, by properly implementing different backends of the compiler, Palgol can be transformed into any Pregel-like system, as long as the system supports the basic Pregel interfaces including message passing between arbitrary pairs of vertices and aggregators.

Table 1. Datasets for performance evaluation

Dataset	**Type**	$\|V\|$	$\|E\|$	Description
Wikipedia	Directed	18,268,992	172,183,984	the hyperlink network of Wikipedia
Facebook	Undirected	59,216,214	185,044,032	a friendship network of the Facebook
USA	Weighted	23,947,347	58,333,344	the USA road network
Random	Chain	10,000,000	10,000,000	a chain with randomly generated values

Table 2. Comparison of execution time between Palgol and Pregel+ implementation

Dataset	**Algorithm**	4 nodes		8 nodes		12 nodes		16 nodes		Comparison
		Pregel+	Palgol	Pregel+	Palgol	Pregel+	Palgol	Pregel+	Palgol	
Wikipedia	SSSP	8.33	10.80	4.47	5.61	3.18	3.83	2.41	2.85	18.06% – 29.55%
	PageRank	153.40	152.36	83.94	82.58	61.82	61.24	48.36	47.66	−1.62% – 2.26%
	SCC	177.51	178.87	85.87	86.52	61.75	61.89	46.64	46.33	−0.66% – 0.77%
Facebook	S-V	143.09	142.16	87.98	86.22	67.62	65.90	58.29	57.49	−2.53% – −0.65%
Random	LR	56.18	64.69	29.58	33.17	19.76	23.48	14.64	18.16	12.14% – 24.00%
USA	MSF	78.80	82.57	43.21	45.98	29.47	31.07	22.84	24.29	4.79% – 6.42%

- PageRank [9]
- Single-Source Shortest Path (SSSP) [9]
- Strongly Connected Components (SCC) [17]
- Shiloach-Vishkin Connected Component Algorithm (S-V) [17]
- List Ranking Algorithm (LR) [17]
- Minimum Spanning Forest (MSF) [1]

Among these algorithms, SCC, S-V, LR and MSF are non-trivial ones which contain multiple computing stages. Their Pregel+ implementations are included in our repository for interested readers.

4.1 Performance Evaluation

In our performance evaluation, we use three real-world graph datasets (Facebook[9], Wikipedia[10], USA[11]) and one synthetic graph, and some detailed information is listed in Table 1. The experiment is conducted on an Amazon EC2 cluster with 16 nodes (whose instance type is m4.large), each containing 2 vCPUs and 8G memory. Each algorithm is run on the type of input graphs to which it is applicable (PageRank on directed graphs, for example) with 4 configurations, where the number of nodes changes from 4 to 16. We measure the execution time for each experiment, and all the results are averaged over three repeated experiments. The runtime results of our experiments are summarized in Table 2.

Remarkably, for most of these algorithms (PageRank, SCC, S-V and MSF), we observed highly close execution time on the compiler-generated programs and the manually implemented programs, with the performance of the Palgol programs varying between a 2.53% speedup to a 6.42% slowdown.

[9] https://archive.is/o/cdGrj/konect.uni-koblenz.de/networks/facebook-sg
[10] http://konect.uni-koblenz.de/networks/dbpedia-link
[11] http://www.dis.uniroma1.it/challenge9/download.shtml

Table 3. Comparison of the compiler-generated programs before/after optimization

Dataset	**Algorithm**	Number of Supersteps			Execution Time		
		Before	After	Comparison	Before	After	Comparison
Wikipedia	SSSP	147	50	−65.99%	5.36	2.85	−46.83%
	PageRank	93	32	−65.59%	45.57	47.66	4.58%
	SCC	3819	1278	−66.54%	106.03	46.33	−56.30%
Facebook	S-V	31	23	−25.81%	52.37	57.49	9.78%
Random	LR	77	52	−32.47%	17.54	18.16	3.51%
USA	MSF	318	192	−39.62%	26.67	24.29	−8.95%

For SSSP, we observed a slowdown up to 29.55%. The main reason is that the human-written code utilizes Pregel's *vote_to_halt()* API to deactivate converged vertices during computation; this accelerates the execution since the Pregel system skips invoking the *compute()* function for those inactive vertices, while in Palgol, we check the states of the vertices to decide whether to perform computation. Similarly, we observed a 24% slowdown for LR, since the human-written code deactivates all vertices after each superstep, and it turns out to work correctly. While voting to halt may look important to efficiency, we would argue against supporting voting to halt as is, since it makes programs impossible to compose: in general, an algorithm may contain multiple computation stages, and we need to control when to end a stage and enter the next; voting to halt, however, does not help with such stage transition, since it is designed to deactivate all vertices and end the whole computation right away.

4.2 Effectiveness of Optimization

In this subsection, we evaluate the effectiveness of the "state merging" optimization mentioned in Section 3.3, by generating both the optimized and unoptimized versions of the code and executing them in the same configurations. We use all the six graph applications in the previous experiment, and fix the number of nodes to 16. The experiment results are shown in Table 3.

The numbers of supersteps in execution are significantly reduced, and this is due to the fact that the main iterations in these graph algorithms are properly optimized. For applications containing only a simple iteration like PageRank and SSSP, we reduce nearly 2/3 supersteps in execution, which is achieved by optimizing the three supersteps inside the iteration body into a single one. Similarly, for SCC, S-V and LR, the improvement is around 2/3, 1/4 and 1/3 due to the reduction of one or two superstep in the main iteration(s). The MSF is a slightly complicated algorithm containing multiple stages, and we get an overall reduction of nearly 40% supersteps in execution.

While this optimization reduces the number of supersteps, and thus the number of global synchronizations, it does not necessarily reduce the overall execution time since it incurs a small overhead for every loop. The optimization produces a tighter loop body by unconditionally sending at the end of each iteration the necessary messages for the next iteration; as a result, when exiting the loop, some

redundant messages are emitted (although the correctness of the generated code is ensured). This optimization is effective when the cost of sending these redundant messages is cheaper than that of the eliminated global synchronizations. In our experiments, SSSP and SCC become twice as fast after optimization since they are not computationally intensive, and therefore the number of global synchronizations plays a more dominant role in execution time; this is not the case for the other algorithms though.

5 Related Work

Google's Pregel [9] 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 [16] 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 [2] and Green-Marl [7]. We briefly introduce each of them below.

Palovca [8] exposes the Pregel APIs in Haskell using a monad, and a vertex-centric program is written in a low-level way like in typical Pregel systems. Since this language is still low-level, programmers are faced with the same challenges in Pregel programming, mainly having to tackle all low-level details.

At the other extreme, the s6raph system [11] is a special graph processing framework with a functional interface, which models a particular type of graph algorithms containing a single iterative computation (such as PageRank and Shortest Path) by six programmer-specified functions. However, many practical Pregel algorithms are far more complicated.

A more comparable and (in fact) closely related piece of work is Fregel [2], which is a functional DSL for declarative programming on big graphs. In Fregel, a vertex-centric computation is represented by a pure step function that takes a graph as input and produces a new vertex state; such functions can then be composed using 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 put them together using two combinators, namely sequence and iteration. Compared with Fregel, the main strength of Palgol is in its remote access capabilities:

- a Palgol step consists of local computation and remote updating phases, whereas a Fregel step function can be thought of as only describing local computation, lacking the ability to modify other vertices' states;
- even when considering local computation only, Palgol has highly declarative *field access expressions* to express remote reading of arbitrary vertices, whereas Fregel allows only neighboring access.

These two features are however essential for implementing the examples in Section 2, especially the S-V algorithm. Moreover, when implementing the same graph algorithm, the execution time of Fregel is around an order of magnitude slower than human written code; Palgol shows that Fregels combinator-based design can in fact achieve efficiency comparable to hand-written code.

Another comparable DSL is Green-Marl [6], which lets programmers describe graph algorithms in a higher-level imperative language. This language is initially proposed for graph processing on the shared-memory model, and a "Pregel-canonical" subset of its programs can be compiled to Pregel. 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. In contrast, Palgol (and Fregel) programs are by construction vertex-centric and distinguish the current and previous states for the vertices, and thus have a closer correspondence with the Pregel model. For remote reads, Green-Marl only supports neighboring access, so it suffers the same problem as Fregel where programmers cannot fetch data from an arbitrary vertex. While it supports graph traversal skeletons like BFS and DFS, these traversals can be encoded as neighborhood access with modest effort, so it actually has the same expressiveness as Fregel in terms of remote reading. Green-Marl supports remote writing, but according to our experience, it is quite restricted, and at least cannot be used inside a loop iterating over a neighbor list, and thus is less expressive than Palgol.

6 Concluding Remarks

This paper has introduced Palgol, a high-level domain-specific language for Pregel systems with flexible remote data access, which makes it possible for programmers to express Pregel algorithms that communicate over dynamic internal data structures. We have demonstrated the power of Palgol's remote access by giving two representative examples, the S-V algorithm and the list ranking algorithm, and presented the key algorithm for compiling remote access. Moreover, we have shown that Fregel's more structured approach to vertex-centric computing, our compilation algorithm for remote access and Green-Marl's optimization techniques can work together perfectly, and the experiment results show that graph algorithms written in Palgol can be compiled to efficient Pregel programs comparable to human written ones.

We expect Palgol's remote access capabilities to help with developing more sophisticated vertex-centric algorithms where each vertex decides its action by looking at not only its immediate neighborhood but also an extended and dynamic neighborhood. The S-V and list ranking algorithms are just a start — for a differently flavored example, graph pattern matching [3] might be greatly simplified when the pattern has a constant size and can be translated declaratively as a remote access expression deciding whether a vertex and some other "nearby" vertices exhibit the pattern.

Algorithm design and language design are interdependent, with algorithmic ideas prompting more language features and higher-level languages making it

easier to formulate and reason about more sophisticated algorithms. We believe that Palgol is a much-needed advance in language design that can bring vertex-centric algorithm design forward.

Acknowledgements. We thank the reviewers for their insightful comments to improve this paper. This work was supported by JSPS KAKENHI Grant Numbers 26280020 and 17H06099.

References

1. Chung, S., Condon, A.: Parallel implementation of Borůvka's minimum spanning tree algorithm. In: IPPS. pp. 302–308. IEEE (1996)
2. Emoto, K., Matsuzaki, K., Morihata, A., Hu, Z.: Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing. In: ICFP. pp. 200–213. ACM (2016)
3. Fard, A., Nisar, M.U., Ramaswamy, L., Miller, J.A., Saltz, M.: A distributed vertex-centric approach for pattern matching in massive graphs. In: BigData. pp. 403–411. IEEE (2013)
4. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. *J. Comput. System Sci.* 30(2), 209–221 (1985)
5. Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. *J. ACM* 37(3), 549–587 (1990)
6. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In: ASPLOS. pp. 349–362. ACM (2012)
7. Hong, S., Salihoglu, S., Widom, J., Olukotun, K.: Simplifying scalable graph processing with a domain-specific language. In: CGO. p. 208. ACM (2014)
8. Lesniak, M.: Palovca: describing and executing graph algorithms in Haskell. In: PADL. pp. 153–167. Springer (2012)
9. 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: SIGMOD. pp. 135–146. ACM (2010)
10. Quick, L., Wilkinson, P., Hardcastle, D.: Using Pregel-like large scale graph processing frameworks for social network analysis. In: ASONAM. pp. 457–463. IEEE (2012)
11. Ruiz, O.C., Matsuzaki, K., Sato, S.: s6raph: vertex-centric graph processing framework with functional interface. In: FHPC. pp. 58–64. ACM (2016)
12. Salihoglu, S., Widom, J.: Optimizing graph algorithms on Pregel-like systems. *PVLDB* 7(7), 577–588 (2014)
13. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From think like a vertex to think like a graph. *PVLDB* 7(3), 193–204 (2013)
14. Valiant, L.G.: A bridging model for parallel computation. *Commun. ACM* 33(8), 103–111 (1990)
15. Xie, M., Yang, Q., Zhai, J., Wang, Q.: A vertex centric parallel algorithm for linear temporal logic model checking in Pregel. *J. Parallel Distrib. Com.* 74(11), 3161–3174 (2014)
16. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB* 7(14), 1981–1992 (2014)
17. Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W., Bu, Y.: Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB* 7(14), 1821–1832 (2014)