Toward Intutive and Efficient Vertex-Centric Graph Processing

by

Yongzhe Zhang

Dissertation

submitted to the Department of Informatics

in partial fulfillment of the requirements for the degree of

Master of Science

SOKENDAI (The Graduate University for Advanced Studies)

June 2017

Abstract

Pregel is a popular parallel 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 programming interfaces, as programmers need to carefully structure an algorithm in terms of supersteps and message passing, which are low-level and hard to reason about. Some domain-specific languages (DSLs) have been proposed to provide more intuitive ways to implement graph algorithms. In these DSLs, vertices can directly access the data from neighbor vertices, while the message passing are hided and generated by the compiler. This approach simplifies the implementation of several algorithms, but it also introduces severe restrictions which cause a wide range of algorithms hard to implement.

To address this problem, we design and implement Palgol, a more declarative and powerful DSL. In particular, when programming in vertex-centric mode, programmers can use a more declarative syntax called *global field access* to directly read data on arbitrary remote vertices. By analyzing the logic patterns of global field access, we provide a novel algorithm for translating remote data access to message passing, and generate high efficient Pregel code from Palgol programs. Furthermore, by structuring the graph algorithms using Palgol's high-level model, we recognize Pregel's weakness in handling graph algorithms with multiple communication channels, and thus extend the Pregel framework with the message channel interfaces, to enable more optimizations on message reduction. We demonstrate the power of Palgol by using it to implement a bunch of practical Pregel algorithms and compare them with hand-written code. The evaluation result shows that the efficiency of Palgol is comparable with that of hand-written code. As a future work, we are going to translate Palgol to the new Pregel framework to achieve better performance.

Contents

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. In this thesis, we focus on a particular type of graph processing system that adopts the "Think like a vertex" paradigm, and indentify the difficulties in programming and optimization. We therefore design a domain-specific language to provide a high-level abstraction and customize a graph processing system for message reduction, trying to give an intuitive and efficient solution for this problem.

1.1 Pregel: Graph Processing in Vertex-Centric Paradigm

Google's Pregel [1] is one of the most popular framework for processing large-scale graphs, which is based on the bulk-synchronous parallel (BSP) model [2]. It adopts the *vertex-centric* computing paradigm to achieve high parallelism and scalability. Following the BSP model, 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. Being simple, Pregel has demonstrated its usefulness in implementing many interesting graph algorithms [1, 3, 4, 5, 6].

Google's Pregel [1] proposed the vertex-centric computing paradigm, which allows programmers to think naturally like a vertex when designing distributed graph algorithms. There are a bunch of open-source alternatives to the official and proprietary Pregel system, such as Apache Hama [12], Apache Giraph [13], Catch the Wind [14], GPS [15], GraphLab [16], PowerGraph [17], Mizan [18] and Pregel+[?].

1.2 Problem in Existing Pregel Frameworks

Despite the power of Pregel, it is a big challenge to implement a graph algorithm correctly and efficiently in Pregel [4], especially when the algorithm consists of multiple stages and complicated data dependencies. The main reason for this is the big gap between graph algorithms and their actual implementations:

- *Exceedingly complicated loop body.* A Pregel program is essentially a loop with a possibly big and complicated loop body defined by the *compute()* function. In contrast, many typical distributed graph algorithms can be seen as the combination (composition) of smaller computation components, each being a simpler graph transformation.

- *Non-declarative message passing style.* In Pregel, each superstep processes the messages sent by other vertex in the previous superstep. This makes a Pregel program hard to understand, because one has to trace where the messages are from and what information they carry. In contrast, it is more natural to describe computation with *remote data access* — reading or writing attributes of other vertices through references — instead of lower-level message passing.

In addition to the difficulties in mapping the graph algorithms to Pregel, the Pregel's interface is actually not suitable for implementing the algorithms containing various stages or requiring data communication for different purposes, and may potentially decrease the performance. This is due to the single message type throughout the whole computation, and the complex side effects of vertex-inactivation interface:

- *Single message type.* In Pregel, programmers need to specify a message type in advance, and the implement the *compute()* function that handles or emits the messages in this type only. However, some graph algorithms actually need different message types for different purposes, so it may consume more space and bring the overhead of data encoding/decoding.

- *Side effects in vertex-inactivation.* Many graph algorithms can be optimized by the vertex-inactivation interface, for example to skip the vertices that are already converged. However, the vertex state is also used to control the termination, and affects the computation of aggregator, which makes it not always applicable in graph algorithms.

1.3 Domain-Specific Languages for Pregel

Domain-Specific Languages (DSLs) are a well-known mechanism for describing solutions in a restricted domain. Therefore, some attempts have been made to bridge this gap by proposing domain-specific languages (DSLs), such as Green-Marl [7], Fregel [8] and s6raph [20]. On one hand, these DSLs allow programmers to write a program in a compositional way to avoid writing a big loop body, and provide neighboring data access to avoid explicit message passing. On the other hand, the programs written in these DSLs can be automatically translated to Pregel code by fusing the components in the programs into a single loop, and mapping neighboring data access into message passing. There are several differences between these languages:

- Green-Marl [21] is a DSL that allows programmers to describe a graph algorithm at a higher level. Originally proposed for graph processing on shared-memory multi-processors, it is extended [7] to support Pregel systems, where the compilation to Pregel relies on discovering "Pregel-canonical" patterns. Since it does not have a Pregel-specific language design, programmers may easily get compilation errors if they are not familiar with the implementation of the compiler. Besides, there is still a wide range of Pregel algorithms that cannot be implemented or efficiently expressed in Green-Marl (detailed discussion is in chapter 4), due to the limitation in expressing remote access.

- Fregel [8] is a functional DSL for declarative programming on big graphs. In Fregel, a vertex-centric computation is represented by a pure function that takes a graph as input and produces a new vertex state; such functions can then be composed using a set of predefined combinators to implement a complete graph algorithm. This approach not only prevents programmers from writing invalid programs, but also helps people to better understand the behavior of a vertex-centric program. However, due to the same reason as Green-Marl, it is also limited in expressiveness. Moreover, when implementing the same graph algorithm, the execution time of Fregel is around an order of magnitude slower than human written code [8], which makes it unsuitable for practice use.

- S6raph [20] is a special graph processing framework with a functional interface. It models a particular type of iterative vertex-centric computation by six functions, which are specified by programmers to implement different algorithms. Compared to Green-Marl and Fregel, s6raph can only represent graph algorithms that contain a single iterative computing (such as PageRank and Shortest Path), while many practical Pregel algorithms are far more complicated. In exchange, s6raph programs can be written concisely and executed efficiently.

However, for efficient implementation, the existing DSLs impose a severe restriction on the data access, allowing data access only from neighboring vertices. In other words, they neither permit reading data from remote vertices (other than neighboring vertices), nor support writing data in remote vertices. This restriction makes it difficult to describe many interesting graph algorithms that need remote data access [5, 4]. For instance, a graph algorithm may maintain a tree structure, and let every vertex use its parent's attribute in computation. In this case, the *parent* is a reference stored on each vertex, and the expected data access is to fetch the parent's attribute through the reference.

It is, in fact, hard to equip DSLs with remote reads that has efficient implementation in Pregel. There are two main reasons. First, remote reads cannot be directly translated into the message passing style. Passing messages to some vertex requires the sender to know the destination. However, a remote read is described from the perspective of the reading vertex, which specifies (and thus knows) which remote vertex to read. To implement the read, the remote vertex should send the data as a message to the reading vertex, but there is no guarantee that the remote vertex knows the reading vertex, so additional rounds of communication are needed to propagate information about the reading vertex to the remote vertex. 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. It requires a more careful design of the translation (to Pregel) that has no redundant communication (i.e., only the least and sufficient data is sent when implementing data reads in Pregel).

It will be even harder to equip DSLs with not only remote reads but also remote writes. Naively adding remote writing capabilities into these languages is dangerous, because it may produce read/write conflicts when having both remote reads and writes in the context of parallel computing. To the best of our knowledge, the problem of translating remote reads/writes to efficient message passing in Pregel has not been well studied.

1.4 Contribution of the Thesis

In this thesis, we tackle all the above problems by proposing a more powerful DSL called Palgol[1], which supports flexible remote data access, a more algorithmic way of representing inter-vertex communication. By structuring supersteps in a high-level vertex-centric computation model and analyzing the logic patterns of global field access, we provide a novel algorithm for compiling Palgol programs to efficient Pregel code. In detail, our main technical contributions are as follows:

- We propose a new high-level model for compositional 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

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

and writes are ordered in a safe way. Using algorithmic supersteps along with *sequence* and *iteration*, graph algorithms can be expressed clearly and concisely.

- Based on the new computation 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 *global field 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 message passing without redundant communications, and an algorithm to translate Palgol programs to Pregel programs where the number of supersteps is reduced whenever possible.
- We demonstrate the power of Palgol by working on a set of representative examples. Palgol is much more powerful than the existing DSLs (such as Green-Marl and Fregel); it covers all what the existing DSLs can do, and can deal with algorithms with more advanced communication patterns such as the Shiloach-Vishkin connectivity algorithm [4], where an internal tree structure needs to be maintained, and data communication is needed between parent and children.
- Based on the Palgol's high-level model, we further study the efficient implementation of graph algorithms in distrubuted environment. We propose an extention of Pregel framework with message channel interfaces, which allows programmers to use different message types and separetedly manage or optimize each communication channel, to reduce the message size in computation.
- 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 the execution time varies from 25.9% speedup to 32.4% slowdown and the number of supersteps can be dramatically reduced up to 51.7% for complicated algorithms.

2 The Palgol Language

In this section, we present our high-level DSL Palgol to describe vertex-centric graph algorithms with flexible remote access. We start from introducing a high-level model, in which an algorithm description has a more direct correspondence with the Pregel model. Then we define the Palgol language based on this model, and explain its syntax and semantics. Finally we use two examples to demonstrate how Palgol can concisely describe some Pregel algorithms using remote access.

2.1 The High-Level Model

The essence of Pregel is its vertex-centric paradigm on top of the BSP model, where programmers basically describe the local computation handling the local vertex states and incoming and outgoing messages, as described in the beginning of ??. This computation model requires low-level desciption of message passing for implementing data commmunication.

The high-level model we will propose is to provide a bridge between an algorithm

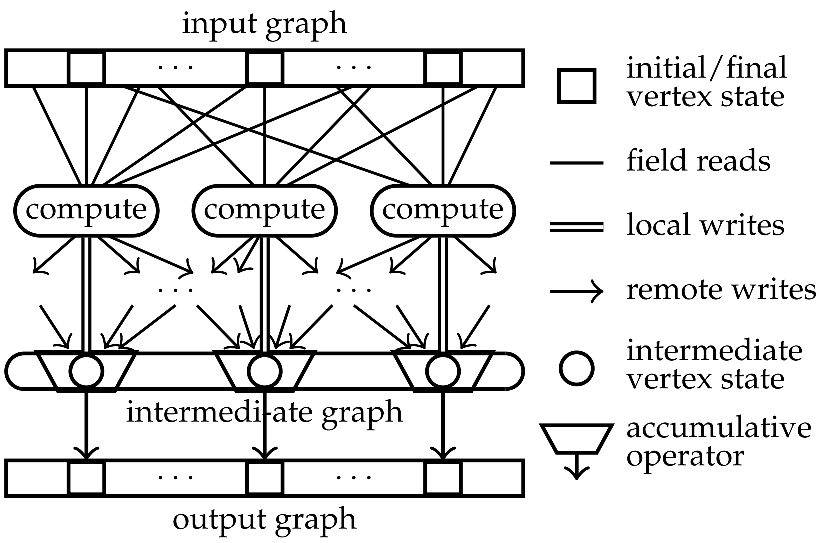


Figure 2.1: In an algorithmic superstep, every vertex performs local computation (including field reads and local writes) and remote updating in order.

and a Pregel implementation. On one hand, it uses remote reads and writes instead of message passing to describe vertex-centric computation, making it more suitable for describing an algorithm. On the other hand, the model is close to the Pregel computation model, in particular using the vertex-centric paradigm and barrier synchronization, and remote reads and writes are arranged in this model to avoid data conflicts, making it possible to automatically derive a valid and efficient Pregel implementation from an algorithm description in this model.

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, vertex-centric computation can be combined or repeated by two operations, *sequence* and *iteration*, to describe sophisticated Pregel algorithms.

The distinguishing feature of algorithmic supersteps is remote access. Within each algorithmic superstep (illustrated in Figure 2.1), 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, to completely avoid 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 the following 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 processing 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.

2.2 An Overview of Palgol

We present our DSL Palgol next, whose design follows the high-level model we introduced in the previous subsection. Figure 2.2 shows the essential part of the syntax of Palgol. 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. 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 conditions, but in this paper we focus on only one kind of termination

$$\begin{array}{lcll}
\mathit{int} & = & \text{integer} & \\
\mathit{float} & = & \text{floating-point number} & \\
\mathit{var} & = & \text{identifier starting with lowercase letter} & \\
\mathit{field} & = & \text{identifier starting with capital letter} & \\
 & & & \\
\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}\ \textbf{mut}_{\mathit{opt}}\ \mathit{var} = \mathit{exp} & \\
 & \mid & \textbf{local}_{\mathit{opt}}\ \mathit{field}\ [\ \mathit{var}\ ]\ \mathit{op}_{\mathit{local}}\ \mathit{exp} & \text{– local write} \\
 & \mid & \textbf{remote}\ \mathit{field}\ [\ \mathit{exp}\ ]\ \mathit{op}_{\mathit{remote}}\ \mathit{exp} & \text{– remote write} \\
\mathit{exp} & ::= & \mathit{int} \mid \mathit{float} \mid \mathit{var} \mid \textbf{true} \mid \textbf{false} \mid \textbf{inf} & \\
 & \mid & \mathit{exp}.\textbf{ref} \mid \mathit{exp}.\textbf{val} \mid \{\mathit{exp}, \mathit{exp}\} & \\
 & \mid & \textbf{fst}\ \mathit{exp} \mid \textbf{snd}\ \mathit{exp} \mid (\mathit{exp},\ \mathit{exp}) & \\
 & \mid & \mathit{exp}\ ?\ \mathit{exp} : \mathit{exp} \mid (\ \mathit{exp}\ ) & \\
 & \mid & \mathit{exp}\ \mathit{op}_{\mathit{binary}}\ \mathit{exp} \mid \mathit{op}_{\mathit{unary}}\ \mathit{exp} & \\
 & \mid & \mathit{field}\ [\ \mathit{exp}\ ] & \text{– 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 \textbf{and} \mid \textbf{or} \mid \ \ldots &
\end{array}$$

Figure 2.2: Essential part of Palgol syntax. Palgol is indentation-based, and two special tokens '⟨' and '⟩' are introduced to represent the increase and decrease of indentation level.

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 remote field using a field access expression of the form *field* [*exp*], and update a remote field using an accumulative assignment statement prefixed with the keyword **remote**.

- A field access expression *field* [*exp*], where *exp* should evaluate to a vertex id, looks similar to an array access, and indeed, one can regard a field as a global array, so that when a vertex id is given, the value of the field on that vertex is returned.

- Remote updating statements take effect only in the RU phase (after the LC phase), regardless of where they occur in the program. Remote updating statements can only be accumulative, whereas local updating statements can be normal or accumulative assignments to the current vertex's fields. To better distinguish the two kinds of updating statements, local updating statements can be prefixed with the **local** keyword.

There are some predefined fields that have special meaning in our language. **Id** is an immutable field that stores the value of the vertex id for each vertex, whose type is user-specified but currently we can simply treat it as an integer. **Nbr**, **In**, **Out** are by default the edge lists, where **Nbr** is intended to be used 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.

The rest of the syntax for Palgol steps is in principle similar to an ordinary programming language, but without mutable variables to make the compilation easy. Some functional programming constructs are also used here, like let-binding and list comprehension. To express more complicated computation, there is also a foreign function interface that allows programmers to invoke functions written in a general-purpose language.

2.3 A Taste of Palgol

In this subsection, we use two examples to demonstrate how to describe vertex-centric algorithms in Palgol. The first example is single-source shortest path (SSSP), and the second one is a connectivity algorithm called Shiloach-Vishkin Practical Pregel Algorithm (S-V) [4], which shows more important features of Palgol.

2.3.1 Single-Source Shortest Path Algorithm

The single-source shortest path problem is among the best known in graph theory and arises in a wide variety of applications. To understand this algorithm, we first have a look at how it is implemented in Google's Pregel system [1] in Figure 2.3.

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int minDist = vertex_id() == 0 ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      minDist = min(minDist, msgs->Value());
    if (minDist < GetValue()) {
      *MutableValue() = minDist;
      OutEdgeIterator iter =
            GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
              minDist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

Figure 2.3: Google's SSSP Pregel program [1]

This program assumes that the initial distance associated with each vertex is infinite (INF). In each superstep, every vertex first receives messages containing the potentially minimum distances from the source computed by its neighbors, and chooses the smallest one among them. If that distance is smaller than the one it currently stores, then it updates its own distance, and sends messages to inform all its neighbors of their potentially minimum distances from its perspective. Every node votes to halt at the end, and in the next superstep only those nodes who have received messages will be activated. The computation is performed iteratively until all vertices become inactive.

Despite the obscure Pregel implementation, which is cluttered with language-specific and low-level details (in particular message passing), the idea of this algorithm is fairly simple, which is an iterative computation until the following equation holds:

$$dist[v] = \begin{cases} 0 & v \text{ is the source} \\ \min_{u \in In(v)} \ (dist[u] \ + \ len(v,\ u)) & \text{otherwise} \end{cases}$$

We can concisely capture the essence of the shortest path algorithm in a Palgol program, as shown in Figure 2.4. In this program, we store the distance of each vertex from the source in the D field, and use a boolean field A to indicate whether the vertex is active. There are two steps in this program. In the first step (lines 1–4), every vertex initializes its own distance and the A field. Then comes the iterative step (lines 6–13) inside **do** . . . **until fix** $[D]$, which runs until every vertex's distance stabilizes. Using a list comprehension (lines 7–8), each vertex iterates over all its active incoming neighbors (those whose A field is true), and generates a list containing the sums of their current distances and the corresponding edge weights. More specifically, the list comprehension goes through every edge e in the incoming edge list **In** $[v]$ such that $A\,[e.\mathbf{ref}]$ is true, and puts $D\,[e.\mathbf{ref}] + e.\mathbf{val}$ in the generated list, where $e.\mathbf{ref}$ represents the neighbor's vertex id and $e.\mathbf{val}$ the edge weight. Finally, we pick the minimum value from the generated list as *minDist*, and update the local fields.

```
1  for v in V
2    Dist[v] := (Id[v] == 0 ? 0 : inf)
3    Active[v] := (Id[v] == 0)
4  end
5  do
6    for v in V
7      let minDist = minimum [ Dist[e.ref] + e.val
8            | e <- In[v], Active[e.ref] ]
9      Active[v] := false
10     if (minDist < Dist[v])
11       Active[v] := true
12       Dist[v] := minDist
13   end
14 until fix[Dist]
```

Figure 2.4: The SSSP program in Palgol

It should be clarified that we do not intend to compile our Palgol program to the Pregel one in Figure 2.3. In fact, the Pregel code generated by the compiling algorithms in chapter 3 is quite different from the manually coded one.

2.3.2 The Shiloach-Vishkin Connectivity Algorithm

The Shiloach-Vishkin Practical Pregel Algorithm (S-V for short) [4] is a more interesting example that uses lots of important features of Palgol. It calculates the connected components (CCs) of an undirected graph in a logarithmic number of supersteps. The connectivity information is maintained using the disjoint-set data structure [9]. 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 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. 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. The description of the algorithm is shown in Figure **??**. Readers interested in the correctness of this algorithm are referred to the original paper [4] for more details.

The implementation of this algorithm is complicated, which contains roughly 120 lines of code[1] for the *compute()* function alone. Here we just explain the message-passing-based Pregel implementation of a small part of the algorithm, to give a sense of its complexity. For example, in the **if** statement, u needs to know whether its parent p is a root node or not, i.e., whether p's pointer points to p itself. The implementation achieves this by making p send its pointer to u in a previous superstep, so that u can check whether p's pointer points to p. Moreover, since p does not maintain a list of its children, u must send its own id to p in an additional superstep before p sends its pointer. We can see that, despite the simplicity of the description at algorithm level, 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 2.5 looks very similar to the algorithm description. This piece of code contains two steps, where the first one (lines 1–3) just performs a simple initialization, and the other (lines 5–13) is inside an iteration as the main computation. We use the field D to store the pointer to the parent vertex. Let us focus on line 6, which corresponds to the **if** statement in the algorithm description that 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. All the magic happens in our compilation algorithm,

[1] The implementation can be accessed in http://www.cse.cuhk.edu.hk/pregelplus/code/apps/basic/svplus.zip

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

Figure 2.5: The S-V program

which analyzes the expression and generates the message passing code.

The rest of the algorithm can be easily 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 (lines 7–8). Now t is either **inf** if the neighbor list is empty or a vertex id; in both cases we can use it to update the parent's pointer (lines 9–10) via a remote updating statement. One important thing is that the parent vertex ($D[u]$) may receive many remote writes from its children, where only one of the children providing the minimum t can successfully perform the updating. Here, the statement `a <?= b` is an accumulative assignment, whose meaning is the same as `a := min(a, b)`. Finally, for the **else** branch, we locally assign u's grandparent's id to u's D field.

The S-V program demonstrates how programmers can naturally describe a complex Pregel algorithm in Palgol, in particular using the field access syntax and the remote updating syntax. Below we will see how such a Palgol program can be efficiently compiled into an efficient Pregel program.

2.4 Vertex Inactivation

In some Pregel algorithms, we may want to inactivate vertices during computation. Typical examples include some matching algorithms like randomized bipartite matching [1] and approximate maximum weight matching [5], where matched vertices are no longer needed in subsequent computation, and the minimum spanning forest algorithm [5] where the graph gradually shrinks during computation.

In Palgol, we model the behavior of inactivating vertices as a special Palgol step, which can be freely composed with other Palgol programs. The syntactic category of *step* is now defined as follows:

$$\begin{array}{rcl} \textit{step} & ::= & \textbf{for } \textit{var} \textbf{ in V} \,\langle\, \textit{block} \,\rangle\, \textbf{end} \\ & | & \textbf{stop } \textit{var} \textbf{ where } \textit{exp} \end{array}$$

The special Palgol step stops those vertices satisfying the condition specified by the boolean-valued expression *exp*, which can refer to the current vertex *var*. The semantics of stopping vertices is different from Pregel's voting to halt mechanism. In Pregel, an inactive vertex can be activated by receiving messages, but such semantics is unsuitable for Palgol, since we already hide message passing from programmers. Instead, a stopped vertex in Palgol will become immutable and never perform any subsequent local computation, but other vertices can still access its fields. This feature is still experimental and we do not further discuss it in this paper; it is, however, essential for achieving the performance reported in chapter 6.

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 main problems are the following two: how to translate Palgol steps into Pregel supersteps, and how to implement sequence and iteration, which will be presented in section 3.2 and section 3.3 respectively. When compiling a single Palgol step, the most challenging part is the remote reads, for which we first give a detailed explanation in section 3.1. We also mention an optimization based on Pregel's combiners in section 3.4.

3.1 Compiling Remote Reads

In current Palgol, our compiler recognizes two forms of remote reads. The first one is called *consecutive field access* (or *chain access* for short), which uses nested field access expressions to acquire remote data. The second one is called *neighborhood communication* where a vertex may use chain access to acquire data from *all* its neighbors. The combination of these two remote read patterns is sufficient to express quite a wide range of practical Pregel algorithms according to our experience (see the discussion in section 4.3). In this section, we present the key algorithms to compile these two remote read patterns to message passing in Pregel.

3.1.1 Consecutive Field Access Expressions

Definition and challenge of compiling: Let us begin from the first case of remote reads, which is consecutive field access expressions (or chain access) 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 consecutive field 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 [10]), 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

$$\begin{tikzcd}
\text{step1:} & u \text{ knows } u & & & {} \arrow[r] & \text{message passing} \\
& u \text{ knows } D[u] \arrow[dr] \arrow[ddr, dashed] & & & {} \arrow[r, dashed] & \text{logical inference} \\
\text{step2:} & & D[u] \text{ knows } u & & & \\
& & D[u] \text{ knows } D[D[u]] \arrow[dl] \arrow[dr] & & & \\
\text{step3:} & u \text{ knows } D[D[u]] \arrow[drr, dashed] & & D[D[u]] \text{ knows } u & & \\
& & & D[D[u]] \text{ knows } D^4[u] \arrow[dll] & & \\
\text{step4:} & u \text{ knows } D^4[u] & & & &
\end{tikzcd}$$

Figure 3.1: Interpretation of the derivation of $\forall u.\, \mathrm{K}_u\, D^4[u]$

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 **??** 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 3.1 is a direct interpretation of the implications in **??**. 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 **??**, 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 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: The algorithm starts from a proposition $\forall u.\, \mathrm{K}_{v(u)}\, e(u)$. 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 consecutive field access expression 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 original 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:

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

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 powerful 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 supersteps in Pregel framework. 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.

3.1.2 Neighborhood Communication

Neighborhood communication is another important communication pattern widely used in Pregel algorithms. Precisely speaking, neighborhood communication refers to those chain access expressions inside a non-nested loop traversing an edge list (**Nbr**, **In** or **Out**), where the chain access expressions start from the neighboring vertex. The following code is a typical example of neighborhood communication, which is a list comprehension used in the S-V algorithm program (Figure 2.5):

```
        let t = minimum [ D[e.ref] | e <- Nbr[v] ]
```

Syntactically, a field access expression $D[e.\mathbf{ref}]$ can be easily identified as a neighborhood communication.

The compilation of such data access pattern is based on the symmetry that if *all* vertices need to fetch the same field of their neighbors, that will be equivalent to making all vertices send the field to all their neighbors. This is a well-known technique that is also adopted by Green-Marl and Fregel, so we do not go into the details and simply summarize the compilation procedure as follows:

1. In the first superstep, we prepare the data from neighbors' perspective. Field access expressions like $D[e.\mathbf{ref}]$ now become neighboring vertices' local fields $D[u]$. Every vertex then sends messages containing those values to all its neighboring vertices.

2. In the next step, every vertex scans the message list to obtain all the values of neighborhood communication, and then executes the loop according to the

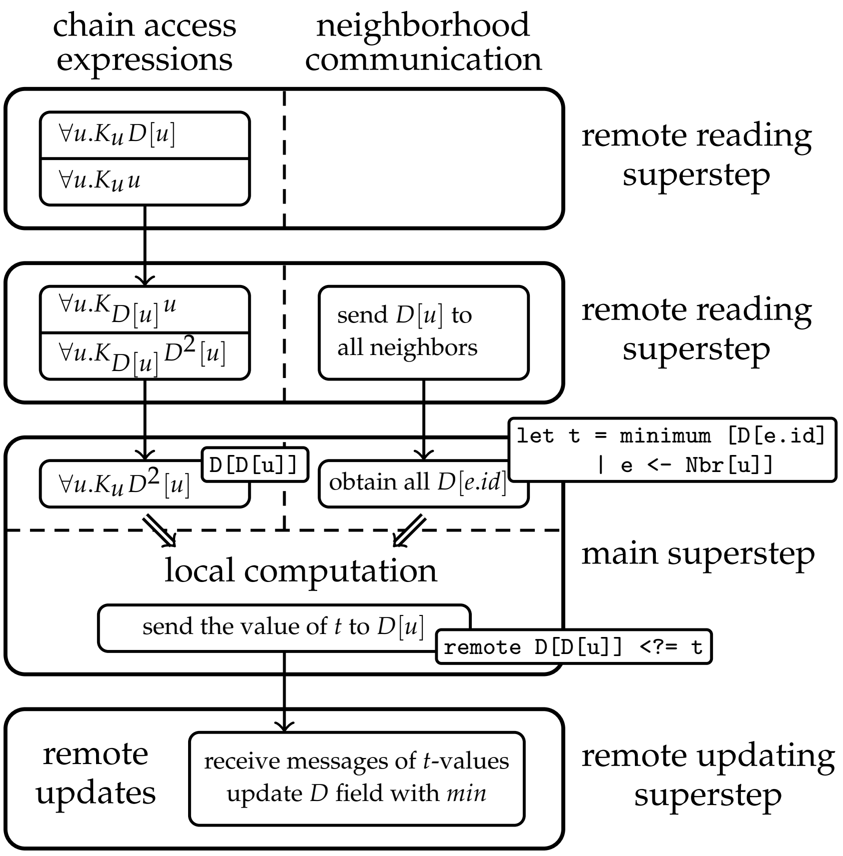


Figure 3.2: Compiling a Palgol step to Pregel supersteps.

Palgol program.

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 3.2. 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 algorithm program (lines 5–13 in Figure 2.5) as a running example for explaining the compilation algorithm, which consists of the following four steps:

1. We first handle neighborhood communications (for S-V algorithm, $D[e.\mathbf{ref}]$ at line 7). As mentioned at the end of subsection 3.1.2, the evaluation of such expressions requires a sending superstep that provides all the remote data for the loops from the neighbors' perspective (for S-V algorithm, sending their D field to all their neighbors). 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 subsection 3.1.1, and corresponding remote reading supersteps are inserted in the front. (For S-V algorithm, the only interesting chain access expression is $D[D[u]]$, which induces two remote reading supersteps realizing a request-reply conversation.) In addition, our handling of neighborhood communication may introduce more chain accesses in the sending superstep, and message passing schemes should also be generated for those accesses. (For S-V algorithm, the chain access introduced by neighborhood communication is $D[u]$, which happens to be trivial.)

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 except for the handling of local writes. In general, we need to create a separate copy of all the involved fields at the beginning of the superstep. Then, during the superstep, all field reads are performed on the original fields, and all updates are on the copies. Finally, we use the (possibly updated) values of the copies to update the original fields at the end of the superstep.

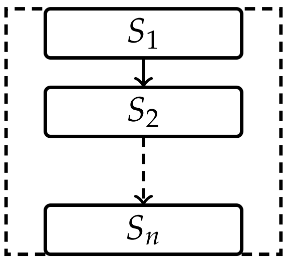
4. What remains to be handled is the remote updating statements, which require sending the updating values as messages to the target vertices in the main superstep. (For S-V algorithm, there is one remote updating statement at line 10, requiring that the value of t be sent to $D[u]$.) Then an additional 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. (For S-V algorithm, the minimum of all the t-values received

is assigned to field D.)

3.3 Compiling Sequences and Iterations

We finally tackle the problem of compiling sequence and iteration, to 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. In the simplest case, every Palgol step is translated into a "linear" STM consisting of a chain of states corresponding to the supersteps like those shown in Figure 3.2. In general, a generated STM may be depicted as:



where there are a start state and an end state, between which there can be more states and transitions, not necessarily having the linear structure.

3.3.1 Sequence

A sequence of two Palgol programs uses the first program to transform an initial graph to an intermediate one, which is then transformed to a final graph using the second program. To compile the sequence, we first compile the two component programs into STMs; a composite STM is then built from these two STMs, implementing the sequence semantics.

We illustrate the compilation in Figure 3.3. The left side is a straightforward way of compiling, and the right side is an optimized one produced by our compiler, with states S_n and S_{n+1} merged together. This is because the separation of S_n and S_{n+1} is unnecessary: every Palgol program describes an independent vertex-centric computation that does not rely on any incoming messages (according to our high-level model); correspondingly, our compilation ensures that the first superstep in

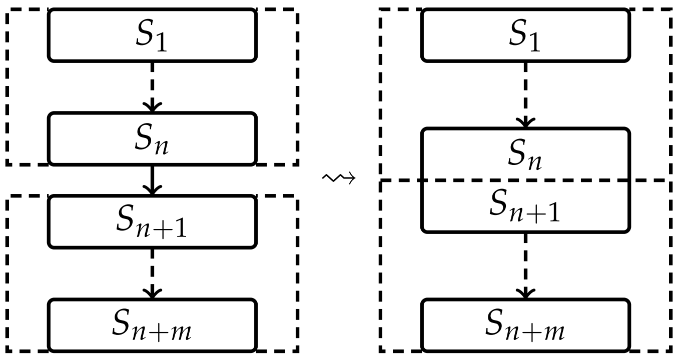


Figure 3.3: The compilation of sequence. A most straightforward way is shown on the left, and our compiler merges the states S_n and S_{n+1} and creates the STM on the right.

the compiled program ignores the incoming messages. We call this the *message-independence* property. Since S_{n+1} is the beginning of the second Palgol program, it ignores the incoming messages, and therefore the barrier synchronization between S_n and S_{n+1} can be omitted.

3.3.2 Iteration

Fixed-point iteration repeatedly runs a program enclosed by '**do**' and '**until** …' until the specified fields stabilize. To compile an iteration, we first compile its body into an STM, then we extend this STM to implement the fixed-point semantics. The output STM is presented in Figure 3.4, where the left one is generated by our general approach, and the right one performs the *fusion optimization* when some condition is satisfied.

Let us start from the general approach on the left. Temporarily ignoring the initialization state, the STM implements a while loop: first, a check of the termination condition takes place right before the state S_1: if the termination condition holds, we immediately enters the state *Exit*; otherwise we execute the body, after which we go back to the check. The termination check is implemented by an OR aggregator to make sure that every vertex makes the same decision: basically, every vertex determines whether its local fields are changed during a single iteration by storing the original values before S_1, and sends the result (as a boolean) to the aggregator, which can then decide globally whether there exists any vertex that has not stabilized. What remains is the initialization state, which guarantees that the termination check will succeed in the first run, turning the while loop into a do-until loop.

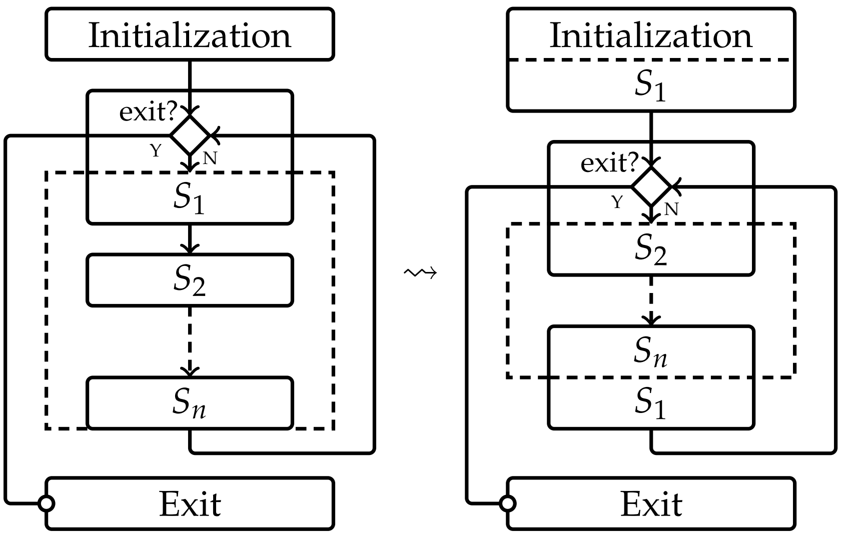


Figure 3.4: An STM for general iteration is shown on the left. The fusion optimization applies when the iteration body begins with a remote reading superstep (S_1), and yields the STM on the right.

There is a chance to reduce the number of supersteps in the loop body of the iteration STM when the first state S_1 of the loop body is a remote reading superstep (see section 3.2). In this case, as shown on the right side of Figure 3.4, the termination check is moved to the beginning of the second state S_2, and then the state S_1 is duplicated and attached to the end of both the initialization state and S_n. This transformation ensures that, no matter from where we reach the state S_2, we always execute the code in S_1 in the previous superstep to send the necessary messages. With this property guaranteed, we can simply iterate S_2 to S_n to implement the iteration, so that the number of supersteps inside the iteration is reduced. The only difference with the left STM is that we execute an extra S_1 attached at the end of S_n when we exit the iteration. However, it still correctly implements the semantics of iteration: the only action performed by a remote reading superstep is sending some messages; although unnecessary messages are emitted, the Palgol program following the extra S_1 will ignore all incoming messages in its first state, as dictated by the message-independence property.

3.4 Combiner Optimization

Combiners are a mechanism in Pregel that may reduce the number of messages transmitted during the computation. Essentially, in a single superstep, if all the messages sent to a vertex are only meant to be consumed by a reduce-operator (e.g., sum or maximum) to produce a value on that vertex, and the values of the individual messages are not important, then the system can combine the messages intended for the vertex into a single one by that operator, reducing the number of messages that must be transmitted and buffered.

In Pregel, combiners are not enabled by default, since "there is no mechanical way to find a useful combining function that is consistent with the semantics of the user's *compute()* method" [1]. On the other hand, Palgol's list comprehension syntax combines remote access and a reduce operator, and naturally represents such type of computation, which can potentially be optimized by a combiner. A typical example is the SSSP program (line 7–8 in Figure 2.4), where the distances received from the neighbors ($D[e.\mathbf{ref}] + e.\mathbf{val}$) are transmitted and reduced by the **minimum** operator. Since the algorithm only cares about the minimum of the messages, and the compiler knows that nothing else is carried by the messages in that superstep, the compiler can automatically implement a combiner with the minimum operator to optimize the program.

4 Expressiveness Analysis

In this section, we give a comprehensive comparison on the expressiveness of Palgol, Green-Marl and Fregel.

4.1 Remote Access Patterns

Palgol is able to express more remote access patterns than Green-Marl and Fregel, particularly chain access and remote writing.

Let us start from a comparison between Palgol and Fregel. A remarkable similarity of Palgol and Fregel is that both of them use a compositional way to construct programs: a Palgol or a Fregel program always consists of several basic units of vertex-centric computation (e.g., Palgol step, or *step function* in Fregel), and put them together using combinators (e.g., sequence and iteration in Palgol, and higher-order functions in Fregel). However, a Palgol step is more expressive than a step function, due to the following two reasons:

- a Palgol step consists of local computation and remote updating phases, whereas a step function can be thought of as only describing local computation, lacking the ability to modify other vertices' states;

- even when considering local computation, Palgol has highly declarative *field access expressions* to express remote reads from an arbitrary vertex, but Fregel can only perform neighborhood communication (see subsection 3.1.2).

As for Green-Marl, 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. Green-Marl suffers the same problem as Fregel, where programmers cannot fetch data from an arbitrary vertex; besides, the remote writes, according to our experience, is quite restricted and unstable in Green-Marl, which at least cannot be used inside a loop iterating over neighbor list.

The remote access abilities of the three languages are summarized in Table 4.1.

Table 4.1: Comparison on Data Access Patterns

Data Access Pattern	Palgol	Green-Marl	Fregel
Neighborhood Reads	✓	✓	✓
Remote Reads	✓	✗	✗
Neighborhood Writes	✓	✗	✗
Remote Writes	✓	✓	✗

4.2 Graph Mutation

Some algorithms need to mutate the graph during computation. For example, they may reverse the edges, shrink the graph, or even build an intermediate graph. Graph mutation can also be used as optimization, for example to remove parallel edges, or delete some vertices or edges that are no longer needed.

Graph mutation is partially supported by Palgol, but has not been supported in Green-Marl or Fregel. Both Green-Marl and Fregel consider the graph as immutable, so we cannot add or delete vertices and edges during graph processing. In Palgol, one can easily perform one kind of graph mutation, which is adding or removing edges from the edge lists. Edge lists are just vertex fields named as *Nbr*, *In* or *Out*, and they can be modified by local or remote updating statements. However, one should make sure

Table 4.2: Comparison on Handling Graph Mutations

Graph Mutation Pattern	Palgol	Green-Marl	Fregel
Adding/Deleting Edges	✓	✗	✗
Adding/Deleting Vertices	✗	✗	✗

that the symmetry of edge lists are kept, or otherwise the generated code will work incorrectly when we further perform neighborhood communication.

There is still one limitation in Palgol (and also in Green-Marl or Fregel) that makes a small class of algorithms unable to implement, which is not having the ability to add or delete vertices. One typical example is the algorithm for calculating bi-connected components (BCC) [4], which builds a dual graph using the edges in the original graph as the new vertices.

4.3 Real-World Graph Algorithms

We further compare the expressiveness of Palgol, Green-Marl and Fregel by real-world graph algorithms. We choose a list of representative Pregel algorithms as follows:

- PageRank (PR) [1]
- Single-Source Shortest Path (SSSP) [1]
- Randomized Bipartite Matching (BM) [1]
- Shiloach-Vishkin Pregel Pregel Algorithm (S-V) [4]
- Bi-Connected Components (BCC) [4]
- Strongly Connected Components (SCC) [4]
- Minimum Spanning Forest (MSF) [5]
- Randomized Graph Coloring (GC) [5]
- Weakly Connected Components (WCC) [5]
- Approximate Maximum Weight Matching (MWM) [5]
- Triangle Counting (TC) [11]

Except for BCC, which requires the ability of adding vertices, all the other algorithms can be implemented in Palgol. On the other hand, many of the listed algorithms cannot be implemented in Green-Marl and Fregel, due to their limitation in remote access and graph mutation.

S-V and MSF are two algorithms that need field access from arbitrary vertex. Both of them maintain an internal tree structure and data communication is needed between parent and children. Such data accessing pattern is definitely not supported in Green-Marl or Fregel. Furthermore, MSF and MWM need to modify the graph by eliminating or reconstructing edges during computation, which is also unable to be done in Green-Marl and Fregel.

The rest of the algorithms only need data fetching between neighbors, so by our understanding, Green-Marl and Fregel should be able to implement them. However, we discover that some algorithms can be implemented more concisely and efficiently in Palgol. For example, an important optimization in SCC and GC is that one can remove unnecessary edges during computation to reduce communication cost, which requires the ability to removing edges.

5
A New Message Channel-Based Pregel System

In this section, we target the problem of Pregel's inefficient message passing interface,

6 Evaluation

We evaluate and compare the performance of the Pregel programs generated by our Palgol compiler and manually coded Pregel programs over large real-world graphs. The experiment is conducted on an Amazon EC2 cluster with 16 nodes (instance type is m4.large), each containing 2 vCPUs and 8G memory.

We compile Palgol code to Pregel+[1], which is an open-source implementation of Pregel, though any Pregel-like system can be used. The reason for choosing Pregel+ is that there is already a bunch of sophisticated Pregel algorithms implemented in this framework (including PageRank, SSSP, S-V algorithm, MSF and SCC), so we can get a fair comparison between our DSL and the hand-written code. For SCC, since the manually coded Pregel program does not completely implement the algorithm and only provides an approximate solution, we are not able to compare them with our Palgol implementation. Consequently, the rest three algorithms are used in the evaluation, and the experiment is conducted in Pregel+'s ordinary mode.

We use 4 real-world graph datasets in our performance evaluation, which are listed

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

Table 6.1: Datasets for Performance Evaluation

Data	Type	$\lvert V\rvert$	$\lvert E\rvert$
LJ-UG	undirected	10,690,276	224,614,770
Facebook	undirected	59,216,214	185,044,032
LJ-DG	directed	4,847,571	68,993,773
Wikipedia	directed	18,268,992	172,183,984

in Table 6.1: (1) LJ-UG[2]: a network of LiveJournal users and their group memberships; (2) Facebook[3]: a friendship network of the Facebook social network; (3) LJ-DG[4]: a friendship network of the LiveJournal blogging community; (4) Wikipedia[5]: the extracted hyperlink network of Wikipedia; To evaluate those manually coded programs on these datasets, we slightly modify the code for graph loading (and data types if necessary), while the computation kernel remains the same. For SSSP, we use a version closer to Figure 2.3, which performs better than the one shipped with Pregel+.

6.1 Performance Comparison

For each experiment, we measure the execution time (which consists of the computation time and communication time) and the number of supersteps of the entire execution. All the results are averaged over 3 repeated experiments.

The runtime results of our experiments are summarized in Table 6.2. Each algorithm is run on the type of input graphs to which it is applicable (PR on directed graphs, for example) with 4 configurations, where the number of cores changes from 4 to 16. As shown, the execution time of the Pregel programs generated by our Palgol compiler vary between 25.9% speedup to 32.4% slowdown, which is comparable with human-written ones.

Remarkably, for S-V algorithm, Palgol code takes up to 25.9% less execution time compared to the manual implementation. This is due to the efficient implementation of the fixed-point termination condition (see section 3.3) by our Palgol compiler, while the manually coded Pregel program contains redundant communication. For PageRank,

[2]http://konect.uni-koblenz.de/networks/livejournal-groupmemberships
[3]http://konect.uni-koblenz.de/networks/facebook-sg
[4]http://konect.uni-koblenz.de/networks/soc-LiveJournal1
[5]http://konect.uni-koblenz.de/networks/dbpedia-link

Table 6.2: Comparison of Execution Time between Palgol and Manual Implementation (in seconds)

Algorithm	**Dataset**	4 cores		8 cores		12 cores		16 cores	
		Palgol	Manual	Palgol	Manual	Palgol	Manual	Palgol	Manual
S-V	LJ-UD	35.46	40.46	16.62	18.82	10.69	12.76	8.06	8.36
	Facebook	146.80	172.33	84.72	105.50	65.67	85.70	56.78	76.61
SSSP	LJ-DG	3.21	2.98	1.83	1.78	1.31	1.23	0.98	0.93
	Wikipedia	10.02	7.57	5.20	4.19	3.61	2.97	2.74	2.30
PR	LJ-DG	76.70	76.54	42.31	42.83	29.83	30.20	23.95	24.10
	Wikipedia	151.26	151.26	80.67	81.05	57.30	57.75	45.44	46.24

we observed highly close execution time on the compiler-generated program and the manually implemented program, where the difference is within 1.7%.

The slowdown of Palgol on SSSP is mainly due to two reasons. First, the manually coded program utilizes Pregel's voting to halt mechanism to inactivate converged vertices during computation; this accelerates the execution since the Pregel system skips executing the *compute()* function for those inactive vertices. Second, due to the simplicity of SSSP, the manually coded program can dispense with the maintenance of a global state with an aggregator, and this further reduces the overhead.

6.2 Number of Supersteps

The number of supersteps used by the compiler-generated programs and manually coded programs are presented in Table 6.3. For PageRank, Palgol takes exactly the same number of supersteps as the manually coded Pregel program, and for SSSP, Palgol needs one additional superstep to terminate the program. The reason is that Palgol needs to check whether all vertices have converged by using an aggregator, while the manual version utilizes the voting to halt mechanism, which can terminate the program immediately when all vertices have converged.

For S-V algorithm, Palgol reduces the number of supersteps by 51.7%. The dramatic improvement for S-V algorithm is due to Palgol's efficient transformation of the fixed-point termination condition. In addition, the state merging and iteration fusion optimizations can generate highly compact code. Although such optimization is applicable to manually implemented code, it takes a lot of effort to do so especially for

Table 6.3: Comparison of the Number of Supersteps

Algorithm	**Dataset**	Palgol	Manual	Comparison
S-V	LJ-UG	14	29	−51.7%
	Facebook	23	43	−46.5%
SSSP	LJ-DG	17	16	6.2%
	Wikipedia	50	49	2.0%
PR	LJ-DG	32	32	0%
	Wikipedia	32	32	0%

large programs, since state merging can easily change the whole implementation (such as message encoding, aggregator, etc). In other words, compact programs are usually harder to maintain for humans.

7 Customizing Pregel with Message Channel Interface

In this section, we present our optimized Pregel framework with the message channel interface, which brings more opportunities for reducing the messages during computation. . Finally, although being an ongoing work, we will explain the relation of Palgol and the customized Pregel framework.

7.1

7.2 Programming Interface

8 Conclusions

8.1 Contributions

This thesis proposes Palgol, a high-level domain-specific language for Pregel systems with flexible remote data access, which allows people to describe vertex-centric graph algorithms in a more natural way. In particular, we tackle the problem of translating remote data read to Pregel's message passing, and by recognizing and compiling two useful remote reading patterns, chain access and neighborhood communication, Palgol can express a wider range of practical Pregel algorithms than existing languages. Moreover, experiment results show that graph algorithms written in Palgol can be compiled to efficient Pregel programs comparable to the human written ones.

8.2 Future Directions

As for future work, on the high-level, we are going to provide a more mature solution for vertex activation or inactivation, and support more remote access patterns in

Palgol. On the aspect of system design.

Bibliography

[1] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *International Conference on Management of Data*, pages 135–146. ACM, 2010.

[2] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[3] Louise Quick, Paul Wilkinson, and David Hardcastle. Using Pregel-like large scale graph processing frameworks for social network analysis. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, pages 457–463. IEEE, 2012.

[4] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[5] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on Pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[6] Miao Xie, Qiusong Yang, Jian Zhai, and Qing Wang. A vertex centric parallel algorithm for linear temporal logic model checking in Pregel. *Journal of Parallel and Distributed Computing*, 74(11):3161–3174, 2014.

[7] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *International Symposium on Code Generation and Optimization*, page 208. ACM, 2014.

[8] Kento Emoto, Kiminori Matsuzaki, Akimasa Morihata, and Zhenjiang Hu. Think like a vertex, behave like a function! a functional dsl for vertex-centric big graph processing. In *International Conference on Functional Programming*, pages 200–213. ACM, 2016.

[9] Harold N Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, 1985.

[10] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.

[11] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. High-level programming abstractions for distributed graph processing. arXiv:1607.02646, 2016.

[12] Apache Hama. http://hama.apache.org/.

[13] Apache Giraph. http://giraph.apache.org/.

[14] Zechao Shang and Jeffrey Xu Yu. Catch the wind: graph workload balancing on cloud. In *International Conference on Data Engineering*, pages 553–564. IEEE, 2013.

[15] Semih Salihoglu and Jennifer Widom. GPS: a graph processing system. In *International Conference on Scientific and Statistical Database Management*, number 22. ACM, 2013.

[16] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, 2012.

[18] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph

processing. In *European Conference on Computer Systems*, pages 169–182. ACM, 2013.

[19] Michael Lesniak. Palovca: describing and executing graph algorithms in Haskell. In *International Symposium on Practical Aspects of Declarative Languages*, pages 153–167. Springer, 2012.

[20] Onofre Coll Ruiz, Kiminori Matsuzaki, and Shigeyuki Sato. s6raph: vertex-centric graph processing framework with functional interface. In *International Workshop on Functional High-Performance Computing*, pages 58–64. ACM, 2016.

[21] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 349–362. ACM, 2012.