JFP **32**, e4, 69 pages, 2022.

doi:10.1017/S0956796821000277

Fregel: a functional domain-specific language for vertex-centric large-scale graph processing

HIDEYA IWASAKI
The University of Electro-Communications, Tokyo, Japan
(*e-mail:* iwasaki@cs.uec.ac.jp)

KENTO EMOTO
Kyushu Institute of Technology, Fukuoka, Japan
(*e-mail:* emoto@csn.kyutech.ac.jp)

AKIMASA MORIHATA
The University of Tokyo, Tokyo, Japan
(*e-mail:* morihata@graco.c.u-tokyo.ac.jp)

KIMINORI MATSUZAKI
Kochi University of Technology, Kochi, Japan
(*e-mail:* matsuzaki.kiminori@kochi-tech.ac.jp)

ZHENJIANG HU
Peking University, Beijing, China
(*e-mail:* huzj@pku.edu.cn)

Abstract

The vertex-centric programming model is now widely used for processing large graphs. User-defined vertex programs are executed in parallel over every vertex of a graph, but the imperative and explicit message-passing style of existing systems makes defining a vertex program unintuitive and difficult. This article presents Fregel, a purely functional domain-specific language for processing large graphs and describes its model, design, and implementation. Fregel is a subset of Haskell, so Haskell tools can be used to test and debug Fregel programs. The vertex-centric computation is abstracted using compositional programming that uses second-order functions on graphs provided by Fregel. A Fregel program can be compiled into imperative programs for use in the Giraph and Pregel+ vertex-centric frameworks. Fregel's functional nature without side effects enables various transformations and optimizations during the compilation process. Thus, the programmer is freed from the burden of program optimization, which is manually done for existing imperative systems. Experimental results for typical examples demonstrated that the compiled code can be executed with reasonable and promising performance.

1 Introduction

The rapid growth of large-scale data is driving demand for efficient processing of the data to obtain valuable knowledge. Typical instances of large-scale data are large graphs such as social networks, road networks, and consumer purchase histories. Since such large

graphs are becoming more and more prevalent, highly efficient large-graph processing is becoming more and more important. A quite natural solution for dealing with large graphs is to use parallel processing. However, developing efficient parallel programs is not an easy task, because subtle programming mistakes lead to fatal errors such as deadlock and to nondeterministic results.

From the programmer's point of view, there are various models and approaches to the parallel processing of large graphs, including the MapReduce model (Bu *et al.*, 2012), the matrix model (Kang *et al.*, 2011, 2012), the data parallelism programming model with a domain-specific language (Hong *et al.*, 2012; Nguyen *et al.*, 2013), and the vertex-centric model (Malewicz *et al.*, 2010; McCune *et al.*, 2015). The *vertex-centric* model is particularly promising for avoiding mistakes in parallel programming. It has been intensively studied and has served as the basis for a number of practically useful graph processing systems (McCune *et al.*, 2015; Khan, 2017; Liu & Khan, 2018; Song *et al.*, 2018; Zhuo *et al.*, 2020). We thus focus on the vertex-centric model in this article.

In vertex-centric graph processing, all vertices in a graph are distributed among computational nodes that iteratively execute a series of computations in parallel. The computations consist of communication with other vertices, aggregation of vertex values as needed, and calculation of their respective values. Communication is typically between adjacent vertices; a vertex accepts messages from incoming edges as input and sends the results of its calculations to other vertices along outgoing edges.

Several vertex-centric graph processing frameworks have been proposed, including Pregel (Malewicz *et al.*, 2010; McCune *et al.*, 2015), Giraph,[1] GraphLab (Low *et al.*, 2012), GPS (Salihoglu & Widom, 2013), GraphX (Gonzalez *et al.*, 2014), Pregel+ (Yan *et al.*, 2014b), and Gluon (Dathathri *et al.*, 2018). Although they release the programmer from the difficulties of parallel programming for large-graph processing to some extent, there still exists a big gap between writing a *natural*, intuitive, and concise program and writing an *efficient* program. As discussed in Section 2, a naturally written vertex-centric program tends to have inefficiency problems. To improve efficiency, the programmer must describe explicit and sometimes complex controls over communications, execution states, and terminations. However, writing these controls is not only an error-prone task but also a heavy burden on the programmer.

In this article, we present a functional domain-specific language (DSL) called *Fregel* for vertex-centric graph processing and describe its model, design, and implementation.

Fregel has two notable features. First, it supports declarative description of vertex computation in functional style without any complex controls over communications, execution states, and terminations. This enables the programmer to write a vertex computation in a natural and intuitive manner. Second, the compiler translates a Fregel program into code runnable in the Giraph or Pregel+ framework. The compiler inserts optimized code fragments into programs generated for these frameworks that perform the complex controls, thereby improving processing efficiency,

Our technical contributions can be summarized as follows:

- We abstract and formalize synchronous vertex-centric computation as a second-order function that captures the higher-level computation behavior using recursive

[1] http://giraph.apache.org.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

execution corresponding to dynamic programming on a graph. In contrast to the traditional vertex-centric computation model, which pushes (sends) information from a vertex to other vertices, our model is *pull-based* (or peek-based) in the sense that a vertex "peeks" on neighboring vertices to get information necessary for computation.

- We present Fregel, a functional DSL for declarative-style programming on large graphs that is based on the pulling-style vertex-centric model. It abstracts communication and aggregation by using comprehensions. Fregel encourages concise, compositional-style programming on large graphs by providing four second-order functions on graphs. Fregel is purely functional without any side effects. This functional nature enables various transformations and optimizations during the compilation process. As Fregel is a subset of Haskell, Haskell tools can be used to test and debug Fregel programs. The Haskell code of the Fregel interpreter in which Fregel programs can be executed is presented in Section 5. Though sequential, this interpreter is useful for checking Fregel programs.
- We show that a Fregel program can be compiled into a program for two vertex-centric frameworks through an intermediate representation (IR) that is independent of the target framework. We also present optimization methods for automatically removing inefficiencies from Fregel programs. The key idea is to use modern constraint solvers to identify inefficiencies. The declarative nature of Fregel programs enables such optimization problems to be directly reduced to constraint-solving problems. Fregel's optimizing compilation frees programmers from problematic programming burdens. Experimental results demonstrated that the compiled code can be executed with reasonable and promising performance.

Fregel currently has a couple of limitations compared with existing Plegel-like frameworks, Giraph and Pregel+. First, the target graph must be a static one that does not change shape or edge weights during execution. Second, a vertex can communicate only with adjacent vertices. Third, each vertex handles only fixed-size data. These mean that algorithms that change the topology of the target graph, update edge weights, or use a variable-length data structure in each vertex cannot be described in Fregel. Removing these limitations by addressing the need to handle dynamism, for example, changing graph shapes and handling variable-length data on each vertex, is left for future work.

The remainder of this article is structured as follows. We start in Section 2 by explaining vertex-centric graph processing and describing its problems. In Section 3, we present our functional vertex-centric graph processing model. On the basis of this functional model, Section 4 describes the design of Fregel with its language constructs and presents many programming examples. In Section 5, we present an interpretive implementation of Fregel in Haskell. In Section 6, we present a detailed implementation of the Fregel compiler, which translates a given Fregel program into Giraph or Pregel+ code. Section 7 discusses optimization methods that remove inefficiencies in the compiled code. Section 8 presents the results of a wide-range evaluation using various programs for both Giraph and Pregel+. Related work is discussed in Section 9, and Section 10 concludes with a summary of the key points, concluding remarks, and mention of future work.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

This article revises, expands, and synthesizes materials presented at the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016) (Emoto *et al.*, 2016) and the 14th International Symposium on Functional and Logic Programming (FLOPS 2018) (Morihata *et al.*, 2018). New materials include many practical program examples of Fregel, redesign and implementation of the Fregel compiler that can generate both Giraph and Pregel+ code, and a wide-range evaluation of the Fregel system from the viewpoints of the performance and the memory usage through the use of both Giraph and Pregel+.

2 Vertex-centric graph processing

Vertex-centric computation became widely used following the emergence the Pregel framework (Malewicz *et al.*, 2010). Pregel enables synchronous computation on the basis of the bulk synchronous parallel (BSP) model (Valiant, 1990) and supports procedural-style programming. Hereafter, we use "Pregel" both as the name of the framework and as the name of the BSP-based vertex-centric computation model.

2.1 Overview of vertex-centric graph processing

We explain vertex-centric computation by using Pregel for procedural-style programming through several small examples.

In Pregel, the vertices distributed on computational nodes iteratively execute one unit of their respective computation, a *superstep*, in parallel, followed by a global barrier synchronization. A superstep is defined as a common user-defined *compute function* that consists of communication between vertices, aggregation of values on all active vertices, and calculation of a value on each vertex. Since the programmer cannot specify the delivery order of messages, operations on delivered messages are implicitly assumed to be commutative and associative. After execution of the compute function by all vertices, global barrier synchronization is performed. This synchronization ensures the delivery of communication and aggregation messages. Messages sent to other vertices in a superstep are received by the destination vertices in the *next* superstep. Thus, only deadlock-free programs can be described.

As an example, let us consider a simple problem of marking all vertices of a graph reachable from the source vertex, for which the identifier is one. We call it the *all-reachability problem* hereafter.

We start with a naive definition of the compute function, which is presented in Figure 1. Here, `vertex.compute` represents a compute function that is repeatedly executed on each vertex. Its first argument, `v`, is a vertex that executes this compute function, and its second argument, `messages`, is a list of delivered messages sent to `v` in the previous superstep. `superstep` is a global variable that holds the number of the current superstep, which begins from 0. The compute function is incomplete in the sense that its iterative computation never terminates. Nevertheless, it suffices for the explanation of vertex-centric computation. Termination control is discussed in Section 2.2.

Every vertex has a Boolean member variable `rch` that holds the marking information, that is, whether the vertex is judged to be reachable at the current superstep. The compute

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
1 vertex.compute(v, messages) {
2   if (superstep == 0) {
3     v.rch = v.vid == 1;
4   } else {
5     newrch = v.rch || or(messages);
6     v.rch = newrch;
7   }
8   sendToNeighbors(v.rch);
9 }
```

Fig. 1. Incomplete and naive Pregel-like code for all-reachability problem.

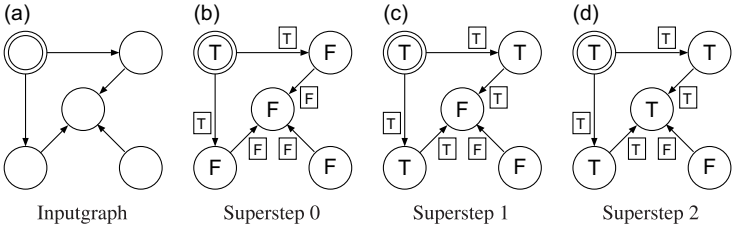


Fig. 2. First three supersteps of the naive program for all-reachability problem.

function accepts a vertex and its received messages as input. At the first superstep, only the source vertex for which the identifier is one is marked `true` and the other vertices are marked `false`. Then each vertex sends its marking information to its neighboring vertices. At the superstep other than the first, each vertex receives incoming messages by "or"ing them, which means that the vertex checks if there is any message containing `true`. Finally, it "or"s the result and the current `rch` value, stores the result as the new marking information, and sends it to its neighboring vertices.

Figure 2 demonstrates how three supersteps are used to mark all reachable vertices for an input graph with five vertices. The T and F in the figure stand for `true` and `false`, respectively, and the double circle indicates the starting vertex.

Though the definition of the compute function is quite simple and easy to understand, the compute function has three *apparent* inefficiency problems in addition to the non-termination problem.

1. A vertex need not send `false` to its neighboring vertices, because `false` never switches a neighbor's `rch` value to `true`.
2. A vertex need not send `true` more than once, because sending it only once suffices for marking its neighbors as `true`.
3. It is not necessary to process all vertices at every superstep except the first one. Only those that receive messages from neighbors need to be processed.

The compute function also has two *potential* inefficiency problems.

4. Global barrier synchronization after every superstep might increase overhead. Though Pregel uses synchronous execution, iteration of the compute function could be performed *asynchronously* without global barrier synchronization.
5. Though the compute function is executed independently by every vertex, a set of vertices placed on the same computational node could cooperate for better performance in the computation of vertex values in the set.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
1  vertex.compute(v, messages) {
2    if (superstep == 0) {
3      v.rch = v.vid == 1;
4      if (v.rch) sendToNeighbors(v.rch);
5      voteToHalt();
6    } else {
7      newrch = v.rch || or(messages);
8      if (newrch != v.rch) {
9        v.rch = newrch;
10       sendToNeighbors(v.rch);
11     }
12     voteToHalt();
13   }
14 }
```

Fig. 3. Improved Pregel-like code for all-reachability problem.

The last two inefficiencies have already been recognized, and mechanisms have been proposed to remove them (Gonzalez *et al.*, 2012; Yan *et al.*, 2014a).

2.2 Inactivating vertices

To address the apparent inefficiencies, Pregel and many Pregel-like frameworks such as Giraph and Pregel+ introduced an "active" property for each vertex. During iterative execution of the compute function, each vertex is either *active* or *inactive*. Initially, all vertices are active. If nothing needs to be done on a vertex, the vertex can become inactive explicitly by *voting to halt*, which means inactivating itself. At each superstep, only active vertices take part in the calculation of the compute function. An inactive vertex becomes active again by being sent a message from another vertex. The entire iterative processing for a graph terminates when all vertices become inactive and there remain no unreceived messages. Thus, inactivating vertices are used to control program termination.

Figure 3 presents Pregel code for the all-reachability problem that remedies the apparent inefficiencies and also terminates when the `rch` values on all vertices no longer change.

At the first superstep, only the source vertex is marked `true`, and it sends its `rch` value to its neighbors. Then all vertices inactivate themselves by voting to halt. At the second and subsequent supersteps, only those vertices that have messages reactivate, receive the messages, and calculate their `newrch` values. If `newrch` and the current `rch` are not the same, the vertex updates its `rch` value and sends it to its neighboring vertices. Then, all vertices inactivate again by voting to halt. If `newrch` and the current `rch` are the same on all vertices, they inactivate simultaneously, and the iterative computation of the compute function terminates.

As can be seen from the code in Figure 3, to remove the apparent inefficiencies, a compute function based on the Pregel model describes communications and termination control explicitly. This makes defining compute functions unintuitive and difficult.

When aggregations are necessary, the situation becomes worse. For example, suppose that we want to mark the reachable vertices and stop when we have a sufficient number (N) of them. For simplicity, we assume that there are more than 100 reachable vertices in the target graph. We call this problem in which $N = 100$ the *100-reachability problem*. At each superstep, the compute function needs to count the number of currently reachable

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
1  vertex.compute(v, messages) {
2    if (superstep == 0) {
3      v.rch = v.vid == 1;
4      if (v.rch) sendToNeighbors(v.rch);
5      else voteToHalt();
6    } else {
7      if (superstep % 2 == 1) {
8        v.newrch = v.rch || or(messages)
9        aggregate(v.newrch ? 1 : 0);
10     } else {
11       cnt = read_agg();
12       if (cnt > 100) {
13         if (v.newrch != v.rch)
14           v.rch = v.newrch;
15         voteToHalt();
16         return;
17       }
18       if (v.newrch != v.rch) {
19         v.rch = v.newrch;
20         sendToNeighbors(v.newrch);
21       }
22     }
23   }
24 }
```

Fig. 4. Pregel-like code for 100-reachability problem.

vertices to determine whether it should continue or halt. To enable acquiring such global information, Pregel supports a mechanism called *aggregation*, which collects data from all active vertices and aggregates them by using a specified operation such as sum or max. Each vertex can use the aggregation result in the next superstep. By using aggregation to count the number of vertices that are marked `true`, we can solve the 100-reachability problem, as shown in the vertex program in Figure 4.

Note that aggregation should be done before the check for the number of reachable vertices. This order is guaranteed by using the odd supersteps to compute aggregation and the even supersteps to check the number. The programmer must explicitly assign states to supersteps so that different supersteps behave differently. The value of `newrch` is set in an odd superstep and read in the next even superstep. Since the extent of a local variable is one execution of the compute function in a superstep, `newrch` has to be changed from a local variable in Figure 3 to a member variable of a vertex in Figure 4.

Only active vertices participate in the aggregation, because inactive vertices do not execute the compute function. Thus, vertices marked `true` should not inactivate, that is, should not vote to halt, in order to determine the precise number of reachable vertices. This subtle control of inactivation is error-prone no matter how careful the programmer.

The program for the 100-reachability problem shows that explicit state controls and subtle termination controls make the program difficult to describe and understand.

2.3 Asynchronous execution

For the fourth potential inefficiency, asynchronous execution in which vertex computations are processed without global barrier synchronization can be considered instead of synchronous execution. Removing barriers could improve the efficiency of the vertex-centric

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

computation. For the all-reachability problem, both synchronous and asynchronous executions lead to the same solution. Generally speaking, however, both executions do not always yield the same result; this depends on the algorithm. In addition, even if they yield the same result, which execution style of the two is more efficient depends on the situation.

Some vertex-centric frameworks, for example, GiraphAsync (Liu *et al.*, 2016), use asynchronous execution. There are also frameworks that support both synchronous and asynchronous executions, such as GraphLab (Low *et al.*, 2012), GRACE (Wang *et al.*, 2013), and PowerSwitch (Xie *et al.*, 2015).

2.4 Grouping related vertices

For coping with the fifth potential inefficiency, placing a group of related vertices on the same computational node and executing all vertex computation as a single unit of processing could improve efficiency. This means enlarging the processing unit from a single vertex to a set of vertices. Many frameworks have been developed on the basis of this idea. For example, NScale (Quamar *et al.*, 2014), Giraph++ (Tian *et al.*, 2013), and GoFFish (Simmhan *et al.*, 2014) are based on subgraph-centric computation, and Blogel (Yan *et al.*, 2014a) is based on block-centric computation. Again, which computation style of the two, vertex-centric or group-based, is more efficient depends on the program.

2.5 Fregel's approach

Fregel enables the programmer to write vertex-centric programs without the complex controls described in Section 2.2 from the declarative perspective and automatically eliminates the apparent inefficiencies of naturally described programs. Since explicit, complex, and imperative controls over communications, terminations, and so forth are removed from a program, the vertex computation proceeds to a functional description with "peeking" on neighboring vertices to obtain information necessary for computation.

To solve the all-reachability problem in Fregel, the programmer writes a natural functional program that corresponds to the Pregel program presented in Figure 1 with a separately specified termination condition. Depending on the compilation options specified by the programmer, the Fregel compiler applies optimizations for reducing inefficiencies in the program and generates a program that can run in a procedural vertex-centric graph processing framework.

As a solution for the fourth potential inefficiency, we propose a method for removing the barrier synchronization and thereby enabling asynchronous execution. This optimization also enables removing the fifth potential inefficiency. In asynchronous execution, the order of processing vertices does not matter; therefore, a group of related vertices can be processed independently from other groups of vertices. To improve the efficiency of processing vertices in a group, we propose introducing priorities for processing vertices.

3 Functional model for synchronous vertex-centric computation

We first modeled the synchronous vertex-centric computation as a higher-order function. Then, on the basis of this model, we designed Fregel, a functional DSL. In this section,

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

we introduce our functional model by using Haskell notation. The Fregel language will be described in Section 4.

In the original Pregel, data communication is viewed as *explicit pushing* in which a vertex sends data to another vertex, typically to its adjacent vertex along an *outgoing* edge. Thus, a Pregel program describes data exchange between two vertices explicitly, for example, by using `sendToNeighbors` in Figure 1, which results in a program with an imperative form. Since our aim is to create a *functional* model of vertex-centric computation, the explicit-pushing style, which has a high affinity with imperative programs, is inappropriate.

We thus designed our functional model so that data communication is viewed as *implicit pulling* in which a vertex pulls (or "peeks at") data in an adjacent vertex connected by an *incoming* edge. The iterative computation at each vertex is defined in terms of a function, and its return value, that is, the result of a single repetition, is implicitly sent to the adjacent vertices. Every adjacent vertex also implicitly receives the communicated value via an argument of the function.

3.1 Definition of datatypes

First, we define the datatypes needed for our functional model. Let *Graph a b* be the directed graph type, where *a* is the vertex value type and *b* is the edge weight type. The vertices have type *Vertex a b*, and the edges have type *Edge a b*. A vertex of type *Vertex a b* has a unique vertex identifier (a positive integer value), a value of type *a*, and a list of *incoming* edges of type [*Edge a b*]. An edge of type *Edge a b* is a pair of the edge weight of type *b* and the source vertex of this edge. *Graph a b* is a list of all vertices, each of which has the type *Vertex a b*.

The definitions of these datatypes are as follows, where *vid*, *val*, and *is* are the identifier, value, and incoming edges of the vertex, respectively,

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Vertex}\ a\ b & = & \mathit{Vertex}\ \{\ \mathit{vid}\ ::\ \mathit{Int},\ \mathit{val}\ ::\ a,\ \mathit{is}\ ::\ [\ \mathit{Edge}\ a\ b\ ]\ \} \\
\textbf{type}\ \mathit{Edge}\ a\ b & = & (b,\ \mathit{Vertex}\ a\ b) \\
\textbf{type}\ \mathit{Graph}\ a\ b & = & [\ \mathit{Vertex}\ a\ b\ ]
\end{array}
$$

For simplicity, we assume that continuous identifiers starting from one are assigned to vertices and that all vertices in a list representing a graph are ordered by their vertex identifiers. As an example, the graph in Figure 2(d) can be defined by the following data structure, where *v1*, *v2*, *v3*, *v4*, and *v5* are the upper-left, upper-right, lower-left, middle, and lower-right vertices, respectively. We assume that all edges have weight 1:

$$
\begin{array}{lcl}
g & :: & \mathit{Graph}\ \mathit{Bool}\ \mathit{Int} \\
g & = & \textbf{let}\ \mathit{v1} = \mathit{Vertex}\ 1\ \mathit{True}\ [\,] \\
 & & \quad\ \mathit{v2} = \mathit{Vertex}\ 2\ \mathit{True}\ [(1, \mathit{v1})] \\
 & & \quad\ \mathit{v3} = \mathit{Vertex}\ 3\ \mathit{True}\ [(1, \mathit{v1})] \\
 & & \quad\ \mathit{v4} = \mathit{Vertex}\ 4\ \mathit{True}\ [(1, \mathit{v2}),\ (1, \mathit{v3}),\ (1, \mathit{v5})] \\
 & & \quad\ \mathit{v5} = \mathit{Vertex}\ 5\ \mathit{False}\ [\,] \\
 & & \textbf{in}\ [\mathit{v1},\ \mathit{v2},\ \mathit{v3},\ \mathit{v4},\ \mathit{v5}]
\end{array}
$$

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

3.2 Description of our model

In synchronous vertex-centric parallel computation, each vertex periodically and synchronously performs the following processing steps, which collectively we call a *logical superstep*, or *LSS* for short.

1. Each vertex receives the data computed in the previous LSS from the adjacent vertices connected by incoming edges.
2. In accordance with the problem to be solved, the vertex performs its respective computation using the received data, the data it computed in the previous LSS, and the weights of the incoming edges. If necessary, the vertex acquires global information using aggregation during computation.
3. The vertex sends the result of the computation to all adjacent vertices along its outgoing edges. The adjacent vertices receive the data in the next LSS.

These three processing steps are performed in each LSS. An LSS represents a semantically connected sequence of actions at each vertex. Each vertex repeatedly executes this "sequence of actions." An LSS is "logical" in the sense that it might contain aggregation and thus might take more than one Pregel superstep. We represent an LSS as a single function and call it an *LSS function*. As explained earlier, an LSS function does not explicitly describe sending and receiving data between a vertex and the adjacent vertices.

The arguments given to an LSS function are an integer value called the *clock* and the vertex on which the LSS function is repeatedly performed. A clock represents the number of iterations of the LSS function. Note that the result of an LSS function may have a type different from that of the vertex value. Thus, the type of an LSS function is $Int \rightarrow Vertex\ a\ b \rightarrow r$, where a is the vertex value type and r is the result type.

We express the LSS function using two functions. One is an *initialization* function, which defines the behavior when the clock is 0, and the other is a *step* function, which defines the behavior when the clock is greater than 0. Let t be a clock value. The initialization function takes as its argument a vertex and returns the result for $t = 0$. Thus, its type is $Vertex\ a\ b \rightarrow r$. The step function takes three arguments: the result for the vertex at the previous clock, a list of pairs, each of which is composed by the weight of an incoming edge and the result of the adjacent vertex connected by the edge at the previous clock, and the vertex itself. Thus, its type is $r \rightarrow [(b, r)] \rightarrow Vertex\ a\ b \rightarrow r$. On the basis of these two functions, a general form of the LSS function is defined in terms of *lssGeneral*, which can be defined as a fold-like second-order function as follows:

$$
\begin{array}{lcl}
\textbf{type}\ Clock & = & Int \\
lssGeneral & :: & (Vertex\ a\ b \rightarrow r) \rightarrow (r \rightarrow [(b, r)] \rightarrow Vertex\ a\ b \rightarrow r) \\
 & & \rightarrow Clock \rightarrow Vertex\ a\ b \rightarrow r \\
lssGeneral\ linit\ lstep\ 0\ v & = & linit\ v \\
lssGeneral\ linit\ lstep\ t\ v & = & lstep\ (lssGeneral\ linit\ lstep\ (t-1)\ v) \\
 & & \quad [\ (e,\ lssGeneral\ linit\ lstep\ (t-1)\ u) \mid (e, u) \leftarrow is\ v\ ] \\
 & & \quad v
\end{array}
$$

An LSS function, *lss*, for a specific problem is defined by giving appropriate initialization and step functions, *ainit* and *astep*, as actual arguments to *lssGeneral*, that is, $lss = lssGeneral\ ainit\ astep$.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
fixedValue          ::  (Eq r, Eq b) ⇒ [ Graph r b ] → Graph r b
fixedValue gs       =   fst (head (dropWhile (λ (a, b) → a ≠ b) (zip gs (tail gs))))
untilValue          ::  (Graph r b → Bool) → [ Graph r b ] → Graph r b
untilValue pred gs  =   head (dropWhile (not . pred) gs)
nthValue            ::  Int → [ Graph r b ] → Graph r b
nthValue n gs       =   head (drop n gs)
```

Fig. 5. Termination functions.

Let $g = [\, v_1,\ v_2,\ v_3,\ \ldots\,]$ be the target graph of type *Graph a b* of the computation, where we assume that the identifier of v_k is k. The list of computation results of LSS function *lss* on all vertices in the graph at clock t is $[\, lss\ t\ v_1,\ lss\ t\ v_2,\ lss\ t\ v_2,\ \ldots\,] :: [\, r\,]$. Further, let g_t be a graph constructed from the results of *lss* on all vertices at clock t, that is, $g_t = makeGraph\ g\ [\, lss\ t\ v_1,\ lss\ t\ v_2,\ lss\ t\ v_3,\ \ldots\,]$. Here, $makeGraph\ g\ [\, r_1,\ r_2,\ \ldots\,]$ returns a graph with the same shape as g for which the i-th vertex has the value r_i and the edges have the same weights as those in g:

```
makeGraph          ::  Graph a b → [ r ] → Graph r b
makeGraph g xs     =   newg
                       where newg =
                               [ Vertex k (xs !! (k − 1))
                                       [ (e, newg !! (k′ − 1)) | (e, Vertex k′ _ _) ← es ]
                                 | Vertex k _ es ← g ]
```

Then the infinite stream (list) of graphs $[\, g_0,\ g_1,\ g_2,\ \ldots\,]$ represents infinite iterations of LSS function *lss*. This infinite stream can be produced by using the higher-order function *vcIter*, which takes as its arguments initialization and step functions and a target graph represented by a list of vertices:

```
vcIter                  ::  (Vertex a b → r) → (r → [(b, r)] → Vertex a b → r)
                            → Graph a b → [ Graph r b ]
vcIter linit lstep g    =   [ makeGraph g (map (lssGeneral linit lstep t) g) | t ← [ 0.. ] ]
```

Though *vcIter* produces an infinite stream of graphs, we want to terminate its computation at an appropriate clock and return the graph at this clock as the final result. We can give a termination condition to the infinite sequence *from outside* and obtain the desired result by using *term* (*vcIter linit lstep g*), where *term* selects the desired final result from the sequence of graphs to terminate the computation.

Figure 5 presents example termination functions. A typical termination point is when the computation falls into a steady state, after which graphs in the infinite list never change. The termination function *fixedValue* returns the graph of the steady state of a given infinite list. Another termination point is when a graph in the stream comes to satisfy a specified condition. We can use the higher-order termination function *untilValue* for this case. It takes a predicate function specifying the desired condition and returns the first graph that satisfies this predicate from a given infinite stream. Finally, *nthValue* retrieves the graph at a given clock.

We define *vcModel* as the composition of a termination function and *vcIter*. We regard the function *vcModel* as representing functional vertex-centric graph processing:

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
reInit              ::  Vertex a b → Bool
reInit v            =   vid v == 0
reStep              ::  Bool → [(b, Bool)] → Vertex a b → Bool
reStep p eqs v      =   p || or [ q | (e, q) ← eqs ]
reAllPregelModel    ::  Graph a b → Graph Bool b
reAllPregelModel    =   vcModel reInit reStep fixedValue
re100PregelModel    ::  Graph a b → Graph Bool b
re100PregelModel    =   vcModel reInit reStep (untilValue ((> 100) . numTrueVertices))
                        where numTrueVertices vs = length (filter val vs)
```

Fig. 6. Formulation of reachability problems in our model.

```
vcModel             ::  (Vertex a b → r) → (r → [(b, r)] → Vertex a b → r) →
                        ([ Graph r b ] → Graph r b) → Graph a b → Graph r b
vcModel linit lstep term  =  term . vcIter linit lstep
```

An LSS function defined in terms of *lssGeneral* has a recursive form on the basis of the structure of the input graph. Although a graph has a recursive structure, a recursive call of an LSS function does not cause an infinite recursion, because a recursive call always uses the prior clock, that is, $t - 1$.

3.3 Simple example

Figure 6 presents the formulation of the reachability problems on the basis of the proposed functional model, where *reAllPregelModel* is for the all-reachablity problem and *re100PregelModel* is for the 100-reachability problem. Variable *numTrueVertices* is the number of vertices with a value of *True* for the target graph. The only difference between these two formulations is the termination condition; the all-reachability problem formulation uses *fixedValue*, while the 100-reachability problem one uses *untilValue*. Note that the LSS function characterized by *reInit* and *reStep* has no description for the aggregation that appears in the original Pregel code (Figure 4).

3.4 Limitations of our model

Our model suffers the following limitations:

- Data can be exchanged only between adjacent vertices.
- A vertex cannot change the shape of the graph or the weight of an edge.

In the Pregel model, a vertex can send data to a vertex other than the adjacent ones as long as it can specify the destination vertex. In our model, unless global aggregation is used, data can be exchanged only between adjacent vertices directly connected by a directed edge. A vertex-centric graph processing model with this limitation, which is sometimes called the *GAS* (gather-apply-scatter) model, has been used by many researchers (Gonzalez *et al.*, 2012; Bae & Howe, 2015; Sengupta *et al.*, 2015).

Furthermore, in our model, computation on a vertex cannot change the shape of the graph or weight of an edge. This limitation makes it is impossible to represent some algorithms including those based on the pointer jumping technique. However, even under this additional limitation, many practical graph algorithms can be described.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

The Fregel language inherits these limitations because it was designed on the basis of our model. As mentioned in Section 1, removing these limitations from the Fregel language is left for future work.

3.5 Features of our model

Our model has four notable features.

First, our model is purely *functional*; computation that is periodically and synchronously performed at every vertex is defined as an LSS function without any side effects that have the form of a structural recursion on the graph structure. The recursive execution of such an LSS function is regarded as dynamic programming on the graph on the basis of memorization.

Second, an LSS function does not have explicit descriptions for sending or receiving data between adjacent vertices. Instead, it uses recursive calls of the LSS function for adjacent vertices, which can be regarded as an implicit pulling style of communication.

Third, an LSS function enables the programmer to describe a series of processing steps as a whole that could be unwillingly divided into small supersteps due to barrier synchronization in the BSP model if we used the original Pregel model.

Fourth, the entire computation for a graph is represented as an infinite list of resultant graphs in ascending clock time order. The LSS function has no description for the termination of the computation. Instead, termination is described by a function that appropriately chooses the desired result from an infinite list.

4 Fregel functional domain-specific language

Fregel is a functional DSL for declarative-style programming on large-scale graphs that uses computation based on *vcModel* (defined in Section 3). A Fregel program can be run on Haskell interpreters like GHCi, because Fregel's syntax follows that of Haskell. This ability is useful for testing and debugging a Fregel program. After testing and debugging, the Fregel program can be compiled into a program for a Pregel-like framework such as Giraph and Pregel+.

4.1 Main features of Fregel

Fregel captures data access, data aggregation, and data communication in a functional manner and supports concise ways of writing various graph computations in a compositional manner through the use of four second-order functions. Fregel has three main features.

First, Fregel abstracts access to vertex data by using three *table*s indexed by vertices. The ***prev*** table is used to access vertex data (i.e., results of recursive calls of the step function) at the previous clock. The ***curr*** table is used to access vertex data at the current clock. These two tables explicitly implement the memorization of calculated values. The third table, ***val*** is used to access vertex initial values, that is, the values placed on vertices when the computation started. An index given to a table is neither the identifier of a vertex nor the position of a vertex in a list of incoming edges but rather is a vertex itself. This enables the programmer to write in a more "direct" style for data accesses.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Second, Fregel abstracts aggregation and communication by using a *comprehension* with a specific generator. Aggregation is described by a comprehension for which the generator is the entire graph (list of all vertices), while communication with adjacent vertices is described by a comprehension for which the generator is the list of adjacent vertices.

Third, Fregel is equipped with four second-order functions for graphs, which we call *second-order graph functions*. A Fregel program can use these functions multiple times. Function ***fregel*** corresponds to functional model *vcModel* defined in Section 3. Function ***gzip*** pairs values for the corresponding vertices in two graphs of the same shape, and ***gmap*** applies a given function to every vertex. Function ***giter*** abstracts iterative computation.

In the following sections, we first introduce the core part of the Fregel language constructs and then explain Fregel programming by using some specific examples.

4.2 Fregel language constructs

A vertex in the functional model described in Section 3.1 has a list of adjacent vertices connected by incoming edges. However, some graph algorithms use edges for the reverse direction. For example, the min-label algorithm (Yan *et al.*, 2014b) for calculating strongly connected components of a given graph, which is described in Section 4.6, needs backward propagation in which a vertex sends messages toward its neighbors connected by its incoming edges. In our implicit pulling style of communications, this means that a vertex needs to peek at data in an adjacent vertex connected by an *outgoing* edge. Thus, though different from the functional model, we decided to let every vertex have two lists of edges: one contains incoming edges in the original graph and the other contains incoming edges in the reversed (transposed) graph. An incoming edge in the reversed graph is an edge produced by reversing an outgoing edge in the original graph. This makes it easier for the programmer to write programs in which part of the computation needs to be carried out on the reversed graph. Hereafter, a "reversed edge" means an edge in the latter list.

Figure 7 presents the syntax of Fregel. Other than the normal reserved words in bold font, the tokens in bold-slant font are important reserved words like identifier names and data constructor names in Fregel. Program examples of Fregel can be found from Sections 4.3 to 4.6. Please refer to these examples as needed.

A Fregel program defines the main function, ⟨mainFn⟩, which takes a single input graph and returns a resultant graph. In the program body, the resultant graph is specified by a graph expression, ⟨graphExpr⟩, which can construct a graph using the four second-order graph functions.

Second-order graph function ***fregel***, which is probably the most frequently used function by the programmer, corresponds to *vcModel* and defines the iterative behavior of an LSS. As described above, it is abstracted as two functions: the initialization function (the first argument) and the step function (the second argument), which is repeatedly executed.

The initialization function of ***fregel*** is the same as that of *vcModel*. It takes a vertex of type *Vertex a b* as its only argument and returns an initial value of type *r* for the iteration carried out by the step function. On the other hand, a step function of ***fregel*** is slightly different.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
⟨program⟩      ::= ⟨recordDef⟩* ⟨mainFn⟩
⟨recordDef⟩    ::= data ⟨constr⟩ = ⟨constr⟩ { ⟨field⟩ :: ⟨type⟩ (, ⟨field⟩ :: ⟨type⟩)* }
⟨mainFn⟩       ::= ⟨func⟩ ⟨graphVar⟩ = let ⟨groundDef⟩ (; ⟨groundDef⟩)* in ⟨graphExpr⟩
⟨groundDef⟩    ::= ⟨defInitFn⟩ | ⟨defStepFn⟩ | ⟨defGraphFn⟩ | ⟨defGraphVar⟩ | ⟨DefSimple⟩
⟨defInitFn⟩    ::= ⟨func⟩ ⟨vertexVar⟩ = ⟨bodyExpr⟩
⟨defStepFn⟩    ::= ⟨func⟩ ⟨vertexVar⟩ prev curr = ⟨bodyExpr⟩
⟨defGraphFn⟩   ::= ⟨func⟩ ⟨graphVar⟩ = ⟨bodyGExpr⟩
⟨defGraphVar⟩  ::= ⟨graphVar⟩ = ⟨graphExpr⟩
⟨defSimple⟩    ::= ⟨defFn⟩ | ⟨defVar⟩
⟨defFn⟩        ::= ⟨func⟩ ⟨var⟩+ = ⟨bodyExpr⟩
⟨defVar⟩       ::= ⟨var⟩ = ⟨bodyExpr⟩
⟨bodyExpr⟩     ::= ⟨expr⟩ | let ⟨defSimple⟩ (; ⟨defSimple⟩)* in ⟨expr⟩
⟨bodyGExpr⟩    ::= ⟨graphExpr⟩ | let ⟨defGraphVar⟩ (; ⟨defGraphVar⟩)* in ⟨graphExpr⟩
⟨graphExpr⟩    ::= fregel ⟨func⟩ ⟨func⟩ ⟨termination⟩ ⟨graphVar⟩
                 | gmap ⟨func⟩ ⟨graphVar⟩ | gzip ⟨func⟩ ⟨graphVar⟩ ⟨graphVar⟩
                 | giter ⟨func⟩ ⟨func⟩ ⟨termination⟩ ⟨graphVar⟩ | ⟨graphVar⟩
⟨termination⟩  ::= Fix | ( Until ⟨predicate⟩ ) | ( Iter ⟨expr⟩ )
⟨expr⟩         ::= if ⟨expr⟩ then ⟨expr⟩ else ⟨expr⟩
                 | ⟨funAp⟩ | ⟨fieldAccess⟩ | ⟨comAggr⟩ | ⟨var⟩ | ⟨constant⟩
                 | ( ⟨expr⟩ )
⟨funAp⟩        ::= ⟨func⟩ ⟨expr⟩+ | ⟨constr⟩ ⟨expr⟩+ | ⟨expr⟩ ⟨binOp⟩ ⟨expr⟩
⟨fieldAccess⟩  ::= ⟨tableExpr⟩ (.^ ⟨field⟩)+
⟨tableExpr⟩    ::= curr ⟨vertexVar⟩ | prev ⟨vertexVar⟩ | val ⟨vertexVar⟩
⟨comAggr⟩      ::= ⟨aggOp⟩ [⟨expr⟩ | ⟨gen⟩ (, ⟨expr⟩)*]
⟨gen⟩          ::= ⟨vertexVar⟩ ← ⟨graphVar⟩
                 | ( ⟨edgeVar⟩ , ⟨vertexVar⟩ ) ← is ⟨vertexVar⟩
                 | ( ⟨edgeVar⟩ , ⟨vertexVar⟩ ) ← rs ⟨vertexVar⟩
⟨pred⟩         ::= ( \ ⟨graphVar⟩ → ⟨expr⟩ )
⟨aggOp⟩        ::= minimum | maximum | sum | prod | and | or
⟨binOp⟩        ::= + | − | < | ...
⟨type⟩         ::= type name
⟨func⟩         ::= function name
⟨constr⟩       ::= constructor name
⟨graphVar⟩     ::= variable name for a graph such as g
⟨vertexVar⟩    ::= variable name for a vertex such as v and u
⟨edgeVar⟩      ::= variable name for an edge such as e
⟨var⟩          ::= variable name
```

Fig. 7. Core part of Fregel syntax.

- First, the step function of *vcModel* executed on every vertex is passed its own result and those of adjacent vertices at the previous clock, together with the weights of incoming edges, through its arguments. In contrast, the step function of ***fregel*** takes a ***prev*** table from which the results of every vertex at the previous clock can be obtained. Edge weights are not explicitly passed to the step function. They can be obtained by using a comprehension for which the generator is the list of adjacent vertices.
- Second, ***fregel***'s step function takes another table called ***curr***, which holds the results at the current clock for the cases in which these values are necessary for computing the results for the current LSS. We show an example of using the ***curr*** table in Section 4.5.
- Third, while the termination judgment of *vcModel* is made using a function that chooses a desired graph from a stream of graphs, that of ***fregel*** is not a function.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Since the initialization and step functions return multiple values in many cases, the programmer must often define a record, ⟨recordDef⟩, for them before the main function and let each vertex hold the record data. Fregel provides a concise way to access a record field by using the field selection operator denoted by .^, which resembles the ones in Pascal and C.

Second-order graph function ***giter*** iterates a specified computation on a graph. Similar to ***fregel***, it takes two functions: the initialization function *iinit* as its first argument and the iteration function *iiter* as its second argument. Let a and b be the vertex value type and edge weight type in the input graph, respectively, and let r be the vertex value type in the output graph. The following iterative computation is performed by ***giter***, where g is the input graph:

$$g :: \mathit{Graph}\ a\ b \xrightarrow{\mathit{iinit}} g_0 :: \mathit{Graph}\ r\ b \xrightarrow{\mathit{iiter}} g_1 :: \mathit{Graph}\ r\ b \xrightarrow{\mathit{iiter}} \cdots \xrightarrow{\mathit{iiter}} g_n :: \mathit{Graph}\ r\ b$$

First, before entering the iteration, *iinit* is applied to every vertex in input graph g to produce the initial graph g_0 of the iteration. Then *iiter* is repeatedly called to produce successive graphs, $g_1, \ldots, g_n$. The iteration terminates when the termination condition given as the third argument of ***giter*** is satisfied. The Haskell definition of ***giter*** in the Fregel interpreter, which may help the reader understand the behavior of ***giter***, is presented in Section 5. Different from ***fregel***'s step function, ***giter***'s iteration function, *iiter*, takes a graph and returns the next graph, possibly by using second-order graph functions. Since ***giter*** is used for repeating ***fregel***, ***gmap***, etc., it takes only a graph. Section 4.6 presents an example of using ***giter***.

The termination condition, ⟨termination⟩, is specified for the third argument of ***fregel*** and ***giter***. This is not a function like *fixedValue* in the functional model, but a data represented by a data constructor like ***Fix***, ***Until***, or ***Iter***, where ***Fix*** means a steady state, ***Until*** means a termination condition specified by a predicate function, and ***Iter*** specifies the number of iterations to perform.

The expressions in Fregel are standard expressions in Haskell, field access expressions on a vertex (⟨fieldAccess⟩), and aggregation expressions (⟨comAggr⟩) each of which applies a combining function to a comprehension with specific generators. There are three generators in Fregel; (1) a graph variable to generate all vertices in a graph, (2) ***is*** v where v is a vertex variable to generate all pairs of v's adjacent vertices connected by incoming edges and the edge weights, and (3) ***rs*** v where v is a vertex variable to generate all pairs of v's adjacent vertices connected by reversed edges and the edge weights. A combining function is one of the six standard functions that have both commutative and associative properties such as *minimum*.

Though Fregel is syntactically a subset of Haskell, Fregel has the following restrictions:

- Recursive definitions are not allowed in a **let** expression. This means that the programmer cannot define (mutually) recursive functions nor variables with circular dependencies.
- Lists and functions cannot be used as values except for functions given as arguments to second-order graph functions.
- A user-defined record has to be non-recursive.
- A specified data obtained from the ***curr*** table have to be already determined.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

(a)

```
data RVal = RVal { rch :: Bool }
reAll g = let reInit v = RVal (vid v == 1);
              reStep v prev curr = let rch' = prev v .^ rch || or [prev u .^ rch | (e, u) ← is v]
                                   in RVal rch'
          in fregel reInit reStep Fix g
```

Program for all-reachability problem

(b)

```
data RVal = RVal { rch :: Bool }
re100 g = let reInit v = RVal (vid v == 1);
              reStep v prev curr = let rch' = prev v .^ rch || or [prev u .^ rch | (e, u) ← is v]
                                   in RVal rch'
          in fregel reInit reStep (Until (λ g → sum [1 | u ← g, val u .^ rch] > 100)) g
```

Program for 100-reachability problem

Fig. 8. Fregel programs for solving reachability problems.

Due to these restrictions, circular dependent values cannot appear in a Fregel program. Thus, Fregel programs do not rely on laziness. In fact, the Fregel compiler compiles a Fregel program into a Java or C++ program that computes non-circular dependent values one by one without the need for lazy evaluation.

4.3 Examples: reachability problems

Our first example Fregel program is one for solving the all-reachability problem (Figure 8(a)). Since the LSS for this problem calculates a Boolean value indicating whether each vertex is currently reachable or not, we define a record *RVal* that contains only this Boolean value at the *rch* field in this record.

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

In *reAll*, *reInit* and *reStep* are given to the ***fregel*** function. Its third argument, ***Fix***, specifies the termination condition, and the fourth argument is the input graph.

Figure 8(b) presents a Fregel program for solving the 100-reachability problem. This program is the same as that in Figure 8(a) except for the termination condition. The termination condition in this program uses ***Until***, which corresponds to *untilValue* in our functional model. ***Until*** takes a function that defines the condition. This function gathers the number of currently reachable vertices by aggregation. Fregel's aggregation takes the form of a comprehension for which the generator is the input graph, that is, a list of all vertices.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data SVal = SVal { dist :: Int }
data MVal = MVal { maxv :: Int }
diameter g =
  let ssspInit v = SVal (if vid v == 1 then 0 else ∞) ;
      ssspStep v prev curr =
        let dist' = prev v .^ dist `min` minimum [prev u .^ dist + e | (e, u) ← is v]
        in SVal dist' ;
      maxvInit v = MVal (val v .^ dist) ;
      maxvStep v prev curr =
        let maxv' = prev v .^ maxv `max` maximum [prev u .^ maxv | (e, u) ← is v]
        in MVal maxv' ;
      g1 = fregel ssspInit ssspStep Fix g ;
      g2 = fregel maxvInit maxvStep Fix g1
  in g2
```

Fig. 9. Fregel program for calculating diameter.

Note that both the initialization and step functions are common to both *reAll* and *re100*. The only difference between them is the termination condition: *reAll* specifies ***Fix*** and *re100* specifies ***Until***. The common step function describes only how to calculate the value of interest (whether or not each vertex is reachable). A description related to termination is not included in the definition of the step function. Instead, it is specified as the third argument of ***fregel***. This is in sharp contrast to the programs in the original Pregel (Figures 3 and 4), in which each vertex's transition to the inactive state is explicitly described in the compute function.

4.4 Example: calculating diameter

The next example calculates the diameter of a graph whose endpoints include the vertex with identifier one. This example sequentially calls two ***fregel*** functions, each of which is similar to the reachability computation. The input is assumed to be a connected undirected graph. In Fregel, an undirected edge between two vertices v_1 and v_2 is represented by two directed edges: one from v_1 to v_2 and the other from v_2 to v_1.

The first call uses values on edges to find the shortest path length from the source vertex (vertex identifier one) to every vertex. This is known as the *single-source shortest path problem.* The second one finds the maximum value of the shortest path lengths of all vertices.

Figure 9 presents the program. The LSS for the first ***fregel*** calculates the tentative shortest path length to every vertex from the source vertex, so record *SVal* consists of an integer field *dist*. The step function *ssspStep* of the first ***fregel*** uses the edge weights, that is, the first component *e* of the pair generated in the comprehension, to update the tentative shortest path for a vertex. It takes the minimum sum of the tentative shortest path of every neighbor vertex (***prev*** *u* .^ *dist*) and the edge length (*e*) from the neighbor vertex.

In the second ***fregel***, every vertex holds the tentative maximum value in the record *MVal* among the values transmitted to the vertex so far. In its step function, *maxvStep*, every vertex receives the tentative maximum values of the adjacent vertices connected by incoming edges, calculates the maximum of the received values and its previous tentative value, and updates the tentative value.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data RRVal = RRVal { rch :: Bool, ranking :: Int }
reRanking g =
  let rerInit v = if vid v == 1 then RRVal True 1 else RRVal False (−1);
      rerStep v prev curr =
        let rch′ = prev v .^ rch || or [prev u .^ rch | (e, u) ← is v];
            r = sum [1 | u ← g, curr u .^ rch];
            ranking′ = if prev v .^ rch == False && rch′ then r else prev v .^ ranking
        in RRVal rch′ ranking′
  in fregel rerInit rerStep Fix g
```

Fig. 10. Fregel program for solving reachability with ranking problem.

The output graph of the first ***fregel***, *g1*, is input to the second ***fregel***, and its resultant graph is the final answer, in which every vertex has the value of the diameter.

4.5 Example: reachability with ranking

Next, we present an example of using the ***curr*** table. The *reachability with ranking problem* is essentially the same as the all-reachability problem except that it also determines the *ranking* of every reachable vertex, where ranking r means that the number of steps to the reachable vertex is ranked in the top r among all vertices. A Fregel program for solving this problem is presented in Figure 10.

We define a record *RRVal* with two fields: *rch* (which is the same as that in *RVal* in the other reachability problems) and *ranking*. For the source vertex, the initialization function, *rerInit*, returns an *RRVal* record in which the *rch* and *ranking* fields are *True* and 1, respectively. For every other vertex, it returns an *RRVal* record in which *rch* is *False* and *ranking* is -1, which means that the ranking is undetermined. The step function, *rerStep*, calculates the new *rch* field value in the same manner as for the other reachability problems. In addition, it calculates the number of reachable vertices at the *current* LSS by using the global aggregation, for which the generator is the entire graph with the *sum* operator. To do this, it filters out the vertices that have not been reached yet. Writing this aggregation as:

$$[\,1 \mid u \leftarrow g,\ rch'\,]$$

is incorrect because rch' is not a local variable on a remote vertex u but rather a local variable on the vertex v that is executing *rerStep*. To enable v to refer to the rch' value of the current LSS on a remote vertex u, it is necessary for u to store the value in an *RRVal* structure by returning an *RRVal* containing the current rch' as the result of *rerStep*. Vertex v can then access the value by ***curr*** u .^ *rch*.

4.6 Example: strongly connected components

As an example of a more complex combination of second-order graph functions, Figure 11 presents a Fregel program for solving the strongly connected components problem. The output of this program is a directed graph with the same shape as the input graph; the value on each vertex is the identifier of the component, that is, the minimum of the vertex identifiers in the component to which it belongs.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data MN = MN { minv :: Int, notf :: Bool }
data C = C { sccId :: Int }
scc g = let finit v = if val v .^ sccId < 0 then MN (vid v) True else MN (val v .^ sccId) False ;
            f0 v = val v ;
            fw v prev curr = let c' = (prev v .^ minv) `min`
                                      minimum [prev u .^ minv | (e, u) <- is v, prev u .^ notf ]
                             in if prev v .^ notf then MN c' (prev v .^ notf) else prev v ;
            bw v prev curr = let c' = (prev v .^ minv) `min`
                                      minimum [prev u .^ minv | (e, u) <- rs v, prev u .^ notf ]
                             in if prev v .^ notf then MN c' (prev v .^ notf) else prev v ;
            detect v = if val v .^ fst .^ minv == val v .^ snd .^ minv
                       then C (val v .^ fst .^ minv) else C (-1) ;
            sccInit v = C (-1) ;
            sccIter g = let ga = gmap finit g ;
                            gf = fregel f0 fw Fix ga ;
                            gb = fregel f0 bw Fix ga ;
                            gfb = gzip gf gb ;
                            g' = gmap detect gfb
                        in g' ;
            gr = giter sccInit sccIter Fix g
        in gr
```

Fig. 11. Fregel program for solving strongly connected components problem.

This program is based on the min-label algorithm (Yan *et al.*, 2014b). It repeats four operations until every vertex belongs to a component.

(1) Initialization: Every vertex for which a component has not yet been found sets the *notf* flag value. This means that the vertex must participate in the following computation.

(2) Forward propagation: Each *notf* vertex first sets its *minv* value as its identifier. Then it repeatedly calculates the minimum value of its (previous) *minv* value and the *minv* values of the adjacent vertices connected by incoming edges. This is repeated until the computation falls into a steady state.

(3) Backward propagation: This is the same as forward propagation except that the direction of *minv* propagation is reversed; each *notf* vertex updates its *minv* value through the reversed edges.

(4) Component detection: Each *notf* vertex judges whether the results (identifiers) of forward propagation and backward propagation are the same. If they are, the vertex belongs to the component represented by the identifier.

The program in Figure 11 has a nested iterative structure.

The outer iteration in terms of ***giter*** repeatedly performs the above operations for the remaining subgraph until no vertices remain. In this outer loop, each vertex has a record *C* that has only the *sccId* field. This field has the identifier of the component, which is the minimum identifier of the vertices in the component, or -1 if the component has not been found yet.

In the processing of operations (1)–(4), each vertex has a record *MN* with two fields. The *minv* field holds the minimum of the propagated values, and the *notf* field holds the flag value explained above. The initialization uses ***gmap*** to create a graph *ga*. There are two inner iterations by the ***fregel*** function: one performs forward propagation and the other performs backward propagation. Both take the same graph created in the initialization. Their results, *gf* and *gb*, are combined by using ***gzip*** and passed to component detection, which is simply defined by ***gmap***.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data Vertex a b = Vertex { vid :: Int, val :: a, is, rs :: [Edge a b] }
type Edge a b = (b, Vertex a b)
type Graph a b = [Vertex a b]
data Termination a = Fix | Until (a → Bool) | Iter Int
data Pair a b = Pair { fst :: a, snd :: b }

termination :: (Eq a) ⇒ Termination a → [a] → a
termination Fix gs = fst (head (dropWhile (λ (a, b) → (a ≠ b)) (zip gs (tail gs))))
termination (Until p) gs = head (dropWhile (not . p) gs)
termination (Iter n) gs = head (drop n gs)

fregel :: (Vertex a b → r) → (Vertex a b → (Vertex a b → r) → (Vertex a b → r) → r) →
          Termination (Graph r b) → Graph a b → Graph r b
fregel init step term g = let rs0 = map init g
                              f rsold = let rsnew = map (λ v → step v prev curr) g
                                            prev u = rsold !! (vid u)
                                            curr u = rsnew !! (vid u)
                                        in rsnew
                              rss = iterate f rs0
                          in termination term (map (makeGraph g) rss)

gmap :: (Vertex a b → r) → Graph a b → Graph r b
gmap f g = makeGraph g (map f g)

gzip :: Graph a1 b → Graph a2 b → Graph (Pair a1 a2) b
gzip g1 g2 = makeGraph g1 (zipWith (λ u v → Pair (val u) (val v)) g1 g2)

giter :: (Eq r, Eq b) ⇒ (Vertex a b → r) → (Graph r b → Graph r b) →
         Termination (Graph r b) → Graph a b → Graph r b
giter iinit iiter term g = let g0 = gmap iinit g
                               gs = iterate iiter g0
                           in termination term gs
```

Fig. 12. Haskell implementation of Fregel.

The four second-order graph functions provided by Fregel abstract computations on graphs and thereby enable the programmer to write a program as a combination of these functions. This functional style of programming makes it easier for the programmer to develop a complicated program, like one for solving the strongly connected components problem.

5 Fregel interpreter

As stated at the beginning of Section 4, a Fregel program can be run on Haskell. We implemented the Fregel interpreter as a library of Haskell. Though this Haskell implementation is used only in the testing and debugging phases during the development of Fregel programs, we describe it here to help the reader understand the behaviors of Fregel programs.

Figure 12 shows the core part of the implementation. The datatypes for the graphs are the same as those described in Section 3.1 except that each vertex has a list of reversed edges in its record under the field name ***rs***. The termination point is defined by the *Termination* type. It has three data constructors: ***Fix*** means a steady state, ***Until*** means a termination condition specified by a predicate function, and ***Iter*** specifies the number of LSS iterations to perform. Function *termination* applies a given termination point to an infinite list of graphs.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

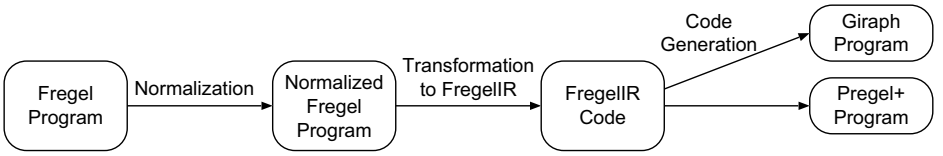


Fig. 13. Compilation flow of Fregel program.

The second-order graph function ***fregel*** takes as its arguments an initialization function, a step function, a termination point, and an input graph and returns the resultant graph of its computation. As explained in Section 4.2, the definition of ***fregel*** here differs somewhat from that of *vcModel*, because it has to implement the memorization mechanism. It does this by using two lists of computation results for all vertices, which are accessed via the vertex identifiers.

Function ***gmap*** applies a given function to every vertex in the target graph and returns a new graph with the same shape in which each vertex has the application result. This is simply defined in terms of *makeGraph*, for which the definition was presented in Section 3.

Function ***gzip*** is given two graphs of the same shape and returns a graph in which each vertex has a pair of values that correspond to those of the vertices of the two graphs. A pair is defined by the *Pair* type with *_fst* and *_snd* fields. This function can also be defined in terms of *makeGraph*.

Function ***giter*** is given four arguments: *iinit*, *iiter*, *term*, and an input graph. It first applies ***gmap*** *iinit* to the input graph and then repeatedly applies *iiter* to the result to produce a list of graphs. Finally, it uses *term* to terminate the iteration and obtain the final result. It can be defined by using a standard function, *iterate*.

6 Fregel compiler

This section describes the basic compilation flow of Fregel programs. Optimizations for coping with the apparent inefficiency problems described in Section 2.2 are described in Section 7.

6.1 Overview of Fregel compiler

The Fregel compiler is a source-to-source translator from a Fregel program to a program for a Pregel-like framework for vertex-centric graph processing. Currently, our target frameworks are Giraph, for which the programs are in Java, and Pregel+, for which the programs are in C++. The Fregel compiler is implemented in Haskell. Figure 13 presents the compilation flow of a Fregel program.

First, a Fregel program is parsed into an abstract syntax tree (AST). Then the AST is transformed into another AST for a *normalized* Fregel program. Since ASTs are internal representations of Fregel programs, we show Fregel programs instead of their ASTs hereafter.

As we have seen in Sections 4.4 and 4.6, a Fregel program can contain multiple uses of second-order graph functions. We do not naively compile each second-order graph

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

function into a Pregel computation, because each invocation of a Pregel computation may start up the Pregel system, which is costly. Instead, we normalize the AST for a Fregel program with (possibly) multiple uses of second-order graph functions into an equivalent one of the following form that uses ***fregel*** with ***Fix*** as the only use of a second-order graph function:

$$prog\ g\ =\ \textbf{let}\ \ldots\ \textbf{in}\ \textbf{\textit{fregel}}\ newInit\ newStep\ \textbf{\textit{Fix}}\ g$$

We call this process and the resulting ASTs *normalization* and *normalized ASTs*, respectively. The normalized AST is transformed into an IR called *FregelIR*. FregelIR is a framework-independent representation in rather procedural style that is close to the target languages, Java (for Giraph) and C++ (for Pregel+). On the one hand, programs in these target languages have many common features such as control structures and styles of function (method) definitions. On the other hand, there are big differences that originate from the design of individual Pregel-like frameworks, such as how to define the compute function, how to exchange messages between vertices, and how to perform aggregations. Thus, we designed FregelIR as an appropriate abstraction layer that represents common features of the two frameworks and moreover absorbs the above-mentioned big differences.

Finally, Giraph or Pregel+ code is generated from a FregelIR representation depending on the option specified by the programmer. The Fregel compiler judges whether a given Fregel program uses reversed edges, ***rs***, and records the judgment into the FregelIR representation of the program. If the program does not use ***rs***, the compiler generates Giraph or Pregel+ code in which the vertices do not have a data structure for unnecessary reversed edges.

6.2 Normalization of Fregel programs

6.2.1 Simple example of normalization

Essentially, normalizing a Fregel program entails building a single-step function that emulates program execution. This step function is basically a *phase* transition machine. Before formerly describing the normalization algorithm, we explain the normalized program by using *diameter* in Figure 9 as an example. Recall that *diameter* contains two occurrences of ***fregel***. The normalization results in a program of the following form:

$$\begin{aligned} diameter\ g\ =\ &\textbf{let}\ newInit\ =\ \ldots\ ; \\ &\quad\ newStep\ =\ \ldots \\ &\textbf{in}\ \textbf{\textit{fregel}}\ newInit\ newStep\ \textbf{\textit{Fix}}\ g \end{aligned}$$

The program consists of a single ***fregel*** function. Its step function, that is, *newStep*, performs the essential computation in two phases followed by the termination phase. These two phases correspond to the two occurrences of ***fregel*** in the original program.

1. At the beginning of the first phase, the same initialization as that of *ssspInit* is performed. Then, the same computation as that of *ssspStep* for finding the shortest path length is repeatedly performed, and whether the computation has fallen into a steady state is detected. If a steady state is detected, the program moves on to the second phase.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

2. At the beginning of the second phase, the same initialization as that of *maxvInit* is performed. During the second phase (except at the beginning), the same computation as that of *maxvStep* is performed and, similar to the first phase, whether the computation has fallen into a steady state is detected. If a steady state is detected, the program moves on to the termination phase.

Since *newStep* executes the computations of both ***fregel*** functions, it is necessary to combine the two records, namely *SVal* and *MVal*, into a single record. In addition, *newStep* has to determine what to execute in the current LSS. We thus let the combined record possess the current phase number and the current counter, that is, the elapsed clock, in the current phase. Thus, the combined record has the following definition:

$$\textbf{data}\ ND\ =\ ND\ \{\, phase :: Int,\ counter :: Int,\ datSVal :: SVal,\ datMVal :: MVal \,\}$$

The initialization function, *newInit*, initializes this record appropriately.

Since *newStep* uses the combined record, record field accesses in the original program before normalization are replaced with the corresponding field accesses to the combined record as follows:

- In *ssspStep*: ***prev*** v .^ *dist* $\longrightarrow$ ***prev*** v .^ *datSVal* .^ *dist*
- In *maxvInit*: ***val*** v .^ *dist* $\longrightarrow$ ***prev*** v .^ *datSVal* .^ *dist*
- In *maxvStep*: ***prev*** v .^ *maxv* $\longrightarrow$ ***prev*** v .^ *datMVal* .^ *maxv*

Please note that since ***val*** v .^ *dist* in *maxvInit* refers to the result of the first ***fregel*** in the original program, it corresponds to the *dist* field in *SVal* in the combined record at the *previous* clock. Thus, it is replaced with ***prev*** v .^ *datSVal* .^ *dist*.

The termination point of every ***fregel*** in the original program is examined explicitly in the *newStep*, because it advances the phase if the condition is satisfied. To this end, *newStep* uses an aggregation. Since ***Fix*** means a steady state, every vertex determines whether the previous and current values of the current phase's computation are the same. For the first phase, previous and current values of vertex u are obtained by ***prev*** u .^ *datSVal* .^ *dist* and ***curr*** u .^ *datSVal* .^ *dist*, respectively. Thus, when the current counter is positive, the result of the aggregation:

$$and\ [\ \textbf{prev}\ u\ .\hat{}\ datSVal\ ==\ \textbf{curr}\ u\ .\hat{}\ datSVal\ |\ u \leftarrow g\ ]$$

represents whether the computation has reached a steady state, where g represents the target graph. If it has, *newStep* advances the *phase* field of the combined record. In addition, *counter* is advanced every time LSS in the current phase is executed and is reset to zero when a new phase begins. The new values of *phase* and *counter* are specified in the *ND* record returned by *newStep*.

Figure 14 presents the pseudocode of the normalized *diameter*. We suppose that the phase numbers of the first, second, and termination phases are one, two, and three, respectively. In addition, in the definition of *newInit*, *defaultSVal* and *defaultMVal*, respectively, represent appropriate default values of *SVal* and *MVal* for which the definitions are omitted. In the definition of *newStep*, d_1 is defined as the value of the *datSVal* field in the combined record at the next clock. Variable e_1 is a Boolean value representing whether the first phase has reached the termination point. Variables d_2 and e_2 are similarly defined.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data SVal = SVal { dist :: Int }
data MVal = MVal { maxv :: Int }
data ND = ND { phase :: Int, counter :: Int, datSVal :: SVal, datMVal :: MVal }
diameter g =
  let newInit v = ND 1 0 defaultSVal defaultMVal ;
      newStep v prev curr =
        let d1 = if prev v .^ phase == 1 then
                   if prev v .^ counter == 0 then inline code of ssspInit
                   else inline code of ssspStep with substitution
                 else prev v .^ datSVal ;
            d2 = if prev v .^ phase == 2 then
                   if prev v .^ counter == 0 then inline code of maxvInit with substitution
                   else inline code of maxvStep with substitution
                 else prev v .^ datMVal ;
            e1 = prev v .^ phase == 1 && prev v .^ counter > 0 &&
                 and [ prev u .^ datSVal == curr u .^ datSVal | u ← g ] ;
            e2 = prev v .^ phase == 2 && prev v .^ counter > 0&&
                 and [ prev u .^ datMVal == curr u .^ datMVal | u ← g ] ;
            phase' = if e1 || e2 then next phase number else prev v .^ phase ;
            counter' = if prev v .^ phase ≠ 3 && prev v .^ phase == curr v .^ phase
                       then prev v .^ counter + 1 else 0
        in ND phase' counter' d1 d2
  in fregel newInit newStep Fix g
```

Fig. 14. Pseudocode of normalized Fregel program for calculating diameter.

6.2.2 Normalization algorithm

We assume that the following preprocessings have already been done on the target Fregel program. They are easily performed using standard techniques such as α-conversion.

1. Bind every call of a second-order graph function to a distinct variable, which we call a *graph variable*.
2. Make variable names unique throughout the program, especially making sure that the variable name of the input graph given to the entire program is g as g is regarded as a special instance of a graph variable.
3. Make the function arguments of ***giter*** unique throughout the program. If two ***giter***s uses the same function, the function should be duplicated with distinct names.
4. Inline user-defined variables and functions within step functions.
5. Infer types of subexpressions and make remaining type-variables monomorphic.

The normalization process consists of five steps.

Step 1: Enumerate phases. The first step is to enumerate each phase corresponding to a use of a second-order graph function. Given the first assumed preprocessing, this is essentially the same as enumerating graph variables except the one for the input graph. Thus, we use graph variables and phases interchangeably.

Let P be the set of graph variables except the input graph. Since ***giter***s need special treatment later, we define a subset I of P, where $I = \{ p \mid p \in P,\ p$ binds a ***giter*** result $\}$.

For *scc* in Figure 11, we have $P = \{ gr, ga, gf, gb, gfb, g' \}$ and $I = \{ gr \}$.

Step 2: Define new record type. The next step is to define a new record type, *ND*, for use in the normalized program. We assume that $P = \{ p_1, \ldots, p_n \}$ and $I = \{ p_{i_1}, \ldots, p_{i_m} \}$

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

($m \leq n,\ i_1 < i_2 < \cdots < i_m$) and that T_p denotes the vertex type of a graph variable p. As stated in Section 6.2.1, we let *ND* possess the current phase number and the current counter in the current phase:

$$\begin{aligned}\textbf{data}\ ND\ =\ ND\ \{\, &phase :: Int,\ counter :: Int,\ d_{p_1} :: T_{p_1},\ \ldots,\ d_{p_n} :: T_{p_n},\\ &ictr_{p_{i_1}} :: Int,\ \ldots,\ ictr_{p_{i_m}} :: Int\,\}\end{aligned}$$

In the above definition of *ND*, dat_{p_j} is used to hold the result of the computation of phase $p_j \in P$ and $ictr_{p_{i_j}}$ is used to hold the number of iterations of the ***giter*** bound to $p_{i_j} \in I$. The new record data for *scc* is shown at the head of Figure 16.

Step 3: Build code pieces for each phase. The new step function for the only ***fregel*** function in the normalized program needs two code pieces for every phase $p \in P$: step function body $comp_p$ for implementing the computation in the phase and termination judgment expression $texp_p$ for detecting the end of the computation in p.

During the building process of $comp_p$ and $texp_p$, ***prev***, ***curr***, and ***val*** used in the original components must be replaced with suitable counterparts. To this end, we define two substitutions, σ_p^1 and $\sigma_{p'}^2$. The former defines the substitution of ***prev*** x and ***curr*** x, while the latter defines the substitution of ***val*** x. Their subscripts (p and p') specify which member in the combined record *ND* is used in the substitution:

$$\begin{aligned}\sigma_p^1\ &=\ \{\,\textbf{\textit{prev}}\ x\ \mapsto\ \textbf{\textit{prev}}\ x\ .\hat{}\ d_p,\ \ \textbf{\textit{curr}}\ x\ \mapsto\ \textbf{\textit{curr}}\ x\ .\hat{}\ d_p\,\}\\ \sigma_{p'}^2\ &=\ \textbf{if}\ p' == g\ \textbf{then}\ \{\ \}\ \textbf{else}\ \{\,\textbf{\textit{val}}\ x\ \mapsto\ \textbf{\textit{prev}}\ x\ .\hat{}\ d_{p'}\,\}\\ \sigma_{p,p'}\ &=\ \sigma_p^1\ \cup\ \sigma_{p'}^2\end{aligned}$$

Both $comp_p$ and $texp_p$ depend on the second-order graph function for which the result is bound to the graph variable corresponding to p. In the following cases, we assume that v is the formal parameter for the vertex given to the new step function we are building.

Case 1: $p\ =$ ***fregel*** *init step term* p'

In this case, $comp_p$ performs the computation of *init* at the beginning of the phase, that is, when *counter* is zero, or the computation of *step* afterward. Thus, $comp_p$ is defined as:

$$comp_p\ =\ \textbf{if}\ \textbf{\textit{prev}}\ v\ .\hat{}\ counter\ ==\ 0\ \textbf{then}\ \sigma_{p,p'}\ (init\ v)\ \textbf{else}\ \sigma_{p,p'}\ (step\ v\ \textbf{\textit{prev}}\ \textbf{\textit{curr}}),$$

where $\sigma_{p,p'}\ (init\ v)$ means applying substitution $\sigma_{p,p'}$ after inlining function application *init* v. Other applications of a substitution in the rest of this section are done in the same manner.

Termination judgment expression $texp_p$ depends on the termination condition, *term*.

When *term* is ***Fix***, judgment is done by checking whether the value of this phase remains unchanged on all vertices. Considering that this judgment is possible after running *step* at least once, we have the following definition of $texp_p$:

$$texp_p\ =\ \textbf{\textit{prev}}\ v\ .\hat{}\ counter\ >\ 0\ \&\&\ and\ [\,\textbf{\textit{prev}}\ u\ .\hat{}\ d_p\ ==\ \textbf{\textit{curr}}\ u\ .\hat{}\ d_p\ \mid\ u \leftarrow p'\,]$$

When *term* is ***Until*** $(\lambda\, p'' \rightarrow e)$, $texp_p$ is defined as e with a suitable substitution applied:

$$texp_p\ =\ \sigma_{p''}^2\ (e)$$

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

When *term* is ***Iter*** k, the judgment is done simply by checking the current counter:

$$texp_p \;=\; \textbf{\textit{prev}}\ v\ .\hat{}\ counter \;==\; k$$

Case 2: $p = \textbf{\textit{gmap}}\ f\ p'$

In this case, $comp_p$ simply applies substitution $\sigma_{p,p'}$ to the inlining result of $f\ v$. Since ***gmap*** does not perform iterative computation, $texp_p$ is always true:

$$\begin{aligned} comp_p &= \sigma_{p,p'}\ (f\ v) \\ texp_p &= True \end{aligned}$$

Case 3: $p = \textbf{\textit{gzip}}\ p_1\ p_2$

In this case, $comp_p$ pairs up the components corresponding to graph variables p_1 and p_2. Similar to Case 2, $texp_p$ is always true:

$$\begin{aligned} comp_p &= Pair\ (\textbf{\textit{prev}}\ v\ .\hat{}\ d_{p_1})\ (\textbf{\textit{prev}}\ v\ .\hat{}\ d_{p_2}) \\ texp_p &= True \end{aligned}$$

Case 4: $p = \textbf{\textit{giter}}\ iinit\ iiter\ term\ p'$

In this case, $comp_p$ performs initialization by *iinit* for the first time, that is, when $ictr_p$ is 0. Note that $ictr_p$ holds the number of iterations of the corresponding ***giter***. Otherwise, since the computation of $comp_p$ has already been done by *iiter*, $comp_p$ can simply obtain the result of *iiter* by $d_{p''}$, where p'' is the output graph of *iiter*:

$$comp_p \;=\; \textbf{if}\ \textbf{\textit{prev}}\ v\ .\hat{}\ ictr_p \;==\; 0\ \textbf{then}\ \sigma_{p,p'}(iinit\ v)\ \textbf{else}\ \textbf{\textit{prev}}\ v\ .\hat{}\ d_{p''}$$

Similar to Case 1, termination judgment expression $texp_p$ depends on termination condition *term*. The difference is that $ictr_p$ is used instead of *counter* for ***giter***. Specifically, when *term* is ***Fix***, $texp_p$ is as follows:

$$texp_p \;=\; \textbf{\textit{prev}}\ v\ .\hat{}\ ictr_p > 0\ \&\&\ and\ [\ \textbf{\textit{prev}}\ u\ .\hat{}\ d_p \;==\; \textbf{\textit{curr}}\ u\ .\hat{}\ d_p \mid u \leftarrow p'\ ]$$

When *term* is ***Until*** $(\lambda\, p'' \rightarrow e)$,

$$texp_p \;=\; \sigma^2_{p''}\ (e).$$

When *term* is ***Iter*** k,

$$texp_p \;=\; \textbf{\textit{prev}}\ v\ .\hat{}\ d_p \;==\; k.$$

Step 4: Build a phase transition machine. Now we define a phase transition machine by using two functions.

One, $next :: P \rightarrow P$, is used to indicate which phase is to be executed next when the computation of the current phase terminates (i.e., when the termination judgment expression returns *True*.) This is defined by a topological sort determined by the dependencies of graph variables. For a program that uses ***giter***, since the output graph of *iiter* is bound to the graph variable corresponding to the ***giter***, this dependency also has to be taken into account.

The other, $stay :: P \rightarrow P$, is used to indicate which phase is to be executed to continue the computation in the current phase (i.e., when the termination judgment expression returns

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
prog g =
  let newInit v = ND r_{p_1} 0 defval_{p_1} ··· defval_{p_n} 0 ··· 0;
      newStep v prev curr =
        let d_{p_1} = if prev v .^ phase == r_{p_1} then comp_{p_1} else prev v .^ d_{p_1};
            ···
            d_{p_n} = if prev v .^ phase == r_{p_n} then comp_{p_n} else prev v .^ d_{p_n};
            e_{p_1} = prev v .^ phase == r_{p_1} && texp_{p_1};
            ···
            e_{p_n} = prev v .^ phase == r_{p_n} && texp_{p_n};
            pend = e_{p_1} || ··· || e_{p_n};
            phase' = if pend then next (prev v .^ phase) else stay (prev v .^ phase);
            counter' = if prev v .^ phase ≠ r_{p_e} && prev v .^ phase == curr v .^ phase
                       then prev v .^ counter + 1 else 0;
            ictr'_{p_{i_1}} = if prev v .^ phase == r_{p_{i_1}} then if pend then 0 else prev v .^ ictr_{p_{i_1}} + 1
                              else prev v .^ ictr_{p_{i_1}};
            ···
            ictr'_{p_{i_m}} = if prev v .^ phase == r_{p_{i_m}} then if pend then 0 else prev v .^ ictr_{p_{i_m}} + 1
                              else prev v .^ ictr_{p_{i_m}}
        in ND phase' counter' d_{p_1} ··· d_{p_n} ictr'_{p_{i_1}} ··· ictr'_{p_{i_m}}
  in fregel newInit newStep Fix g
```

Fig. 15. Template of normalized Fregel program.

False.) Basically, $stay\ p = p$ for most phases, but for a phase that corresponds to ***giter***, *stay* returns the entry phase of the iterative computation by the ***giter***.

For example, graph variables of *scc* have the following dependencies:

- gf and gb depend on ga by ***fregel***.
- gfb depends on gf and gb by ***gzip***.
- g' depends on gfb by ***gmap***.
- gr depends on g' because gr corresponds to ***giter*** and g' is the output of *sccIter*.
- ga depends on gr because ga is the input graph of ***giter***.

Thus, we can define $next(ga) = gf$, $next(gf) = gb$, $next(gb) = gfb$, $next(gfb) = g'$, and $next(g') = gr$. It should be noted that we can swap gf and gb in the above definition of *next* because there is no dependency between them. For *stay*, we define $stay(gr) = ga$ because ga is the entry phase of ***giter***, and $stay(p) = p$ for other phases.

Step 5: Build a normalized program. A normalized program is built by using the components built so far. We assign a unique phase number (integer) r_p to each phase p. We also introduce a special phase p_e and its phase number r_{p_e} to indicate the termination of the entire computation and let $stay(p_e) = p_e$ and $next(gr) = p_e$, where gr is the output graph variable in the original program.

Figure 15 shows the template of a normalized Fregel program. The main part is the new step function, *newStep*, to emulate the original computation. When the current phase number obtained by ***prev*** v .^ *phase* is r_{p_j}, it executes the step function body $comp_{p_j}$. The phase transition is controlled by the termination judgments, $texp_{p_j}$, and the transition functions, *next* and *stay*. Note that *newStep* returns the same value as before once ***prev*** v .^ *phase* becomes n_{p_e}, because $stay(p_e)$ returns p_e and $counter'$ is always bound to 0. Thus, the computation terminates. The initialization function, *newInit*, simply initializes the current

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data ND = ND { phase :: Int, counter :: Bool, d_gr :: C, d_ga :: MN, d_gf :: MN, d_gb :: MN,
               d_gfb :: Pair MN MN, d_g' :: C, ictr_gr :: Int }
scc g =
  let newInit v = ND 1 0 (C 0) (MN 0 False) (MN 0 False) (MN 0 False)
                    (Pair (MN 0 False) (MN 0 False)) (C 0) 0 ;
      newStep v prev curr =
        let d_gr = if prev v .^ phase == 1 then
                     if prev v .^ ictr_gr == 0 then C (−1) else prev v .^ d_g' else prev v .^ d_gr ;
            d_ga = if prev v .^ phase == 2 then
                     if prev v .^ d_gr .^ sccId < 0 then MN (vid v) True
                     else MN (prev v .^ d_gr .^ sccId) False else prev v .^ d_ga ;
            d_gf = if prev v .^ phase == 3 then
                     if prev v .^ counter == 0 then prev v .^ d_ga else
                       let c' = (prev v .^ d_gf .^ minv) `min`
                                  minimum [prev u .^ d_gf .^ minv | (e, u) ← is v, prev u .^ d_gf .^ notf ]
                       in if prev v .^ d_gf .^ notf then MN c' (prev v .^ d_gf .^ notf) else prev v .^ d_gf
                   else prev v .^ d_gf ;
            d_gb = if prev v .^ phase == 4 then
                     if prev v .^ counter == 0 then prev v .^ d_ga else
                       let c' = (prev v .^ d_gb .^ minv) `min`
                                  minimum [prev u .^ d_gb .^ minv | (e, u) ← rs v, prev u .^ d_gb .^ notf ]
                       in if prev v .^ d_gb .^ notf then MN c' (prev v .^ d_gb .^ notf) else prev v .^ d_gb
                   else prev v .^ d_gb ;
            d_gfb = if prev v .^ phase == 5 then Pair (prev v .^ d_gf) (prev v .^ d_gb)
                    else prev v .^ d_gfb ;
            d_g' = if prev v .^ phase == 6 then
                     if prev v .^ d_gfb .^ fst .^ minv == prev v .^ d_gfb .^ snd .^ minv
                     then C (prev v .^ d_gfb .^ fst .^ minv) else C (−1)
                   else prev v .^ d_g' ;
            e_gr = prev v .^ phase == 1 && prev v .^ ictr_gr > 0 &&
                   and [prev u .^ d_gr == curr u .^ d_gr | u ← g ] ;
            e_ga = prev v .^ phase == 2 && True ;
            e_gf = prev v .^ phase == 3 && prev v .^ counter > 0 &&
                   and [prev u .^ d_gf == curr u .^ d_gf | u ← g ] ;
            e_gb = prev v .^ phase == 4 && prev v .^ counter > 0 &&
                   and [prev u .^ d_gb == curr u .^ d_gb | u ← g ] ;
            e_gfb = prev v .^ phase == 5 && True ;
            e_g' = prev v .^ phase == 6 && True ;
            pend = e_gr || e_ga || e_gf || e_gb || e_gfb || e_g' ;
            phase' = if pend then next (prev v .^ phase) else stay (prev v .^ phase) ;
            counter' = if prev v .^ phase ≠ 0 && prev v .^ phase == curr v .^ phase
                       then prev v .^ counter + 1 else 0 ;
            ictr'_gr = if prev v .^ phase == r_1 then if pend then 0 else prev v .^ ictr_{p_i1} + 1
                                                 else prev v .^ ictr_gr
        in ND phase' counter' d_gr d_ga d_gf d_gb d_gfb d_g' ictr'_gr
  in fregel newInit newStep Fix g
```

Fig. 16. Normalized Fregel program for *scc*.

phase to r_{p_1}, counters (*counter*, $ictr_{p_{i_1}}, \ldots, ictr_{p_{im}}$) to 0, and other members in *ND* to their default values, $defval_{p_{ij}}$.

Figure 16 presents the normalized Fregel program for *scc* in Figure 11.

6.2.3 Simple optimization in normalization process

For brevity, the transformation explained so far did not take the efficiency of the normalized program into account and introduced much redundancy. Standard optimizations such

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data IRProg  =  IRProg String                          — program name
                [ IRTypeDecl ]                         — datatypes
                IRVertexStruct                         — datatype for vertices
                IREdgeStruct                           — datatype for edges
                IRMsgStruct                            — datatype for messages
                IRAggStruct                            — datatype for aggregations
                IRCompute                              — computation in phases
type IRNameType  =  (String, IRType)        — name and type (definition of IRType is omitted)
data IRTypeDecl  =  IRTypeDecl String [ IRNameType ]        — struct name and members
data IRVertexStruct  =  IRVertexStruct String String String [ IRNameType ]
                                  — struct name, phase name, sub-phase name, and members
data IREdgeStruct  =  IREdgeStruct String [ IRNameType ]    — struct name and members
data IRMsgStruct  =  IRMsgStruct String [ IRNameType ]      — struct name and members
data IRAggStruct  =  IRAggStruct String [ (IRNameType, IRAggOp) ]
                                                       — struct name, members and operators
type IRCompute  =  IRCompute [ IRComputeProcess ]           — computation in a state
type IRComputeProcess  =  IRComputeProcess IRComputeState      — state
                                           [ IRNameType ]      — local variables
                                           IRBlock             — body
                                           [ (IRExpr, IRComputeState, IRBlock) ]
                          — condition for state transition, next state, and communications
type IRComputeState  =  (Int, Int)                     — phase and sub-phase
```

Fig. 17. Simplified type definitions of FregelIR in Haskell.

as inlining and simplification can reduce redundancy. For example, on the right-hand side of $d_{g'}$ of the normalized program in Figure 16, the redundant pair introduced by ***gzip*** can be eliminated by replacing ***prev*** v .^ d_{gfb} .^ *_fst* and ***prev*** v .^ d_{gfb} .^ *_snd* with ***prev*** v .^ d_{gf} and ***prev*** v .^ d_{gb}, respectively. This simple optimization has been implemented in the normalization process.

6.3 Transforming normalized Fregel into FregelIR

6.3.1 Design of FregelIR

FregelIR is specialized to express Fregel programs. It bridges the gap between the functional style of Fregel programs and the imperative style of programs in the Giraph and Pregel+ frameworks. To this end, we designed FregelIR as a *state transition machine* with two key features. First, every phase in a normalized Fregel program is further split into *subphases*, each of which corresponds to a superstep in Pregel. As a result, a phase that performs communications including aggregations necessarily consists of multiple subphases. Each state is a pair of a phase and its subphase. Second, computation is imperative in a state where processing order is important. This makes generating Java and C++ programs from a FregelIR representation a straightforward process.

Figures 17 and 18 present simplified type definitions of FregelIR in Haskell.

Type *IRProg* is the top-level representation for the entire program. It consists of datatypes used in phases, datatypes for vertices, edges, messages and aggregators, and *IRCompute* data that represents the computation. Each datatype has a name and members; *IRVertexStruct* has additional members for phase and subphase, and *IRAggStruct* has information about the aggregation operator for every aggregator. Type *IRCompute*

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data IRBlock = IRBlock [ IRNameType ] [ IRStmt ]          -- local variables and body
data IRStmt = IRStmtLocal IRVar IRExpr                    -- assignment to a variable
            | IRStmtMsg IRVar IRAggOp IRExpr              -- gather messages from neighbors
            | IRStmtReturn IRExpr                         -- return statement
            | IRStmtVTH                                   -- vote-to-halt statement
            | IRStmtAggr IRNameType IRExpr                -- submit a value to an aggregator
            | IRStmtSendN IRNameType IRExpr               -- send a value
data IRVar = IRVarLocal IRNameType                                          -- local variable
           | IRVarVertex IRNameType IRPrevCurr [ IRNameType ]               -- member in a vertex
           | IRVarEdge IRNameType [ IRNameType ]                            -- member in an edge
           | IRVarAggr IRNameType                                           -- aggregator
data IRPrevCurr = IRPrev | IRCurr | IRNone                -- prev, curr, or none of them
data IRExpr = IRIf IRExpr IRExpr IRExpr                   -- if-then-else
            | IRFunAp IRFun [ IRExpr ]                    -- function/operator application
            | IRVExp IRVar                                -- variable
            | IRCExp IRType IRConst                       -- constant
            | IRMVal IRNameType                           -- message
            | IRAggr IRNameType                           -- aggregated value
data IRFun = IRFun String | IRBinOp String                -- function or binary operator
data IRAggOp = IRAggMin | IRAggMax | IRAggSum | IRAggProd | IRAggAnd | IRAggOr
                                                          -- aggregation operators
data IRConst = IRCInt Int | IRCDouble Double | IRCString String | IRCBool Bool
```

Fig. 18. Simplified type definitions of FregelIR in Haskell, continued.

is essentially a list of *IRComputeProcess*'es. Each *IRComputeProcess* represents the computation for its corresponding state with the following information:

- state, that is, a pair of a phase and subphase,
- local variables,
- a block for the computation including receiving messages,
- conditions for state transitions and next states, and
- a block for sending messages to neighbors.

A block consists of statements represented in *IRStmt* form, which has enough levels of abstraction to absorb the differences between frameworks. FregelIR contains minimum functionalities for expressing programs obtained from Fregel programs. For example, it does not have a structure corresponding to a general-purpose while-loop, because while-loops are unnecessary for transformed framework code.

We next explain the abstraction of FregelIR by using an example of the all-reachability problem, for which a program was presented in Figure 8. In the Fregel program, each vertex collects Boolean values sent from neighboring vertices by using a comprehension and takes their "or" value. This part is represented as the following type *IRStmt* data:

IRStmtMsg (*IRVarLocal* ("*agg*", *irBool*)) *IRAggOr* (*IRMVal* ("*agg*", *irBool*))

Here, "*agg*" is a local variable name to which the result is assigned. The same name is also used as the member name in the message structure. *IRAggOr* represents the disjunction operation used in combining received data, and *irBool* represents the Boolean type. This representation is abstract enough to express the computation in a framework-independent manner. From this *IRStmtMsg* structure, the following Java code for Giraph is generated, where `MsgData` is the typename for messages:

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
agg = false;
for (MsgData msg : messages) agg = (agg || (msg.agg).get());
```

For Pregel+, the following C++ code is generated. Here, `messages` is a vector for messages incoming to the vertex:

```
agg = false;
for (int i = 0; i < messages.size(); i++)
  agg = (agg || messages[i].agg_X425);
```

Note that in the above *IRStmt* data, there is no explicit description of iterating over messages or of obtaining a Boolean value from each message.

6.3.2 Generating FregelIR

Through normalization, a Fregel program is transformed into a program that contains a single ***fregel*** function. However, there remain three essential differences between a normalized Fregel program and FregelIR code:

- A normalized Fregel program is functional, while FregelIR code is imperative.
- A normalized Fregel program describes an LSS, while FregelIR code is composed of supersteps in the Pregel sense.
- A normalized Fregel program describes communications, that is, message exchanges between vertices and aggregations, based on comprehensions and values of other vertices found in a look-up table. In contrast, FregelIR code explicitly describes these communications.

For generating imperative FregelIR code, the FregelIR generator identifies the dependencies of **let**-bound variables and reorders computation of values for these variables so as not to refer to not-yet-computed values.

For every phase p, it is necessary to split the LSS composed by the step function body $comp_p$ and termination judgment $texp_p$ into multiple supersteps at the points where communications occur. Each superstep is referred to as a subphase. As a concrete example, consider the generation of FregelIR code from the normalized *scc* program in Figure 16.

In the expression bound to d_{gf}, communications between adjacent vertices are performed using the following comprehension:

$$minimum\ [\ \textbf{\textit{prev}}\ u\ .\hat{}\ d_{gf}\ .\hat{}\ minv\ \mid (e,u) \leftarrow \textbf{\textit{is}}\ v,\ \textbf{\textit{prev}}\ u\ .\hat{}\ d_{gf}\ .\hat{}\ notf\ ]$$

FregelIR code for this comprehension uses *IRStmtSendN* to send the *minv* value and then transits to the next subphase. From every *IRStmtSendN*, an appropriate code that uses a message-sending API for the target framework (Giraph or Pregel+) is generated. In the next subphase, the FregelIR code gathers the messages sent from neighbors in the previous subphase by using *IRStmtMsg*.

Similarly, an aggregation for termination detection can be found in the expression bound to e_{gf}:

$$and\ [\ \textbf{\textit{prev}}\ u\ .\hat{}\ d_{gf}\ ==\ \textbf{\textit{curr}}\ u\ .\hat{}\ d_{gf}\ \mid u \leftarrow g\ ]$$

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

FregelIR code for this aggregation submits the result of equality test by using *IRStmtAggr* and then transits to the next subphase. The code receives the submitted values and combines them by the *and* function using *IRAggr* in the next subphase.

On the basis of the split subphases, FregelIR code is generated as a state transition machine. In the termination detection of each phase, if termination of the computation at the current phase is detected, the execution state at the next superstep is set to the entrance subphase of the next phase. Otherwise, it is set to the beginning of the iteration of the current phase.

By splitting a phase into multiple subphases, local (non-vertex) variables might be used over successive subphases, that is, supersteps. Such variables should be moved as member variables in the data structure held by each vertex.

6.4 Generating framework programs from FregelIR

From an *IRProg* structure for the entire program in PregeIR, a program for the target framework is generated. For every datatype in *IRProg*, a class (for Giraph) or a struct (for Pregel+) is defined. The target framework may require members that are not explicitly described in FregelIR, and such members are automatically added. For example, Pregel+ requires that the vertex struct has a vector of outgoing edges.

The compute function is built from *IRComputeProcess* datatypes, each of which describes a computation for its corresponding state. The compute function at each vertex dispatches its execution on the basis of the current phase and subphase obtained from its vertex struct.

For generating framework-dependent code, we used Haskell's type classes. To illustrate the basic idea, we describe the generation of framework code for the following *IRStmtMsg* structure, which was presented in Section 6.3.1:

IRStmtMsg (*IRVarLocal* ("*agg*", *irBool*)) *IRAggOr* (*IRMVal* ("*agg*", *irBool*))

To enable framework-dependent code generation, we define a type class called *PregelGenerator* (Figure 19(a)). This type class is a collection of function and variable definitions used for generating framework-dependent code. For each framework, an instance of *PregelGenerator* is defined: *GiraphGenerator* for Giraph and *PregelPlusGenerator* for Pregel+.

For the above example of *IRStmtMsg*, we generate framework code using *ggIRStmtMsg*, for which the definition is presented in Figure 19(c). Framework code consists of an initialization of the destination variable generated by *ggAssign* and a loop generated by *gRecvMsgLoop*, which successively takes a delivered message and performs a value-combining operation. In this code, since the loop structure is framework-dependent, *PregelGenerator* requires every instance to define *gRecvMsgLoop*, which generates a code fragment for the loop structure. Thus, *GiraphGenerator* and *PregelPlusGenerator* define *gRecvMsgLoop* so as to return a string containing a suitable `for`-statement (Figure 19(c)).

We do not convert the IR into the AST of the target language (Java or C++). This is because the IR itself is sufficiently low-level to enable program strings of the target language to be directly generated from the IR without going through an AST.

We defined every function that generates framework-dependent code to take an instance of *PregelGenerator* type class as its argument. By defining a suitable instance in this way,

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

(a)

```
class PregelGenerator gg where
  gRecvMsgLoop :: gg → IRAggOp → String → String → String
  ...        — Other definitions are omitted
```

Definition of type class *PregelGenerator*.

(b)

```
data GiraphGenerator = GiraphGenerator
instance PregelGenerator GiraphGenerator where
  gRecvMsgLoop GiraphGenerator aggop lvar mvar =
    "for (MsgData msg : messages) " ++ ggAssign lvar rhs ++ ";"
    where rhs = "(" ++ lvar ++ ggBinOp aggop ++ m ++ ")"
          m = "(msg." ++ mvar ++ ").get()"
  ...        — Other definitions are omitted

data PregelPlusGenerator = PregelPlusGenerator
instance PregelGenerator PregelPlusGenerator where
  gRecvMsgLoop PregelPlusGenerator aggop lvar mvar =
    "for (int i = 0; i < messages.size(); i++) " ++ ggAssign lvar rhs ++ ";"
    where rhs = "(" ++ lvar ++ ggBinOp aggop ++ m ++ ")"
          m = "messages[i]." ++ mvar
  ...        — Other definitions are omitted
```

Instance definitions of *GiraphGenerator* and *PregelPlusGenerator*.

(c)

```
ggIRStmtMsg :: (PregelGenerator gg) ⇒ gg → IRVar → IRAggOp → IRExpr → String
ggIRStmtMsg gg (IRVarLocal (lvar, lty)) aggop (IRMVal (mvar, mty)) =
  ggAssign lvar (ggUnit aggop) ++ ";" ++ gRecvMsgLoop gg aggop lvar mvar
...        — Other pattern matches are omitted

ggAssign :: String → String → String
ggAssign rhs lhs = rhs ++ " = " ++ lhs

ggUnit :: IRAggOp → String
ggUnit IRAggOr = "false"
...

ggBinOp :: IRAggOp → String
ggBinOp IRAggOr = "||"
...
```

Framework independent functions.

Fig. 19. Generating framework dependent code.

parts of the Fregel compiler for framework-dependent code generation can be packaged within the instance definition.

7 Code optimization

At this point, we have introduced the Fregel programming language and its basic compilation. Although this approach facilitates the development of runnable graph processing programs, as discussed in Section 2.1, it is still difficult to achieve efficiency. Natural programs tend to be slow.

To see the problem, recall the programs for the all-reachability problem (reAll) shown in Figure 8 and the single-source shortest path problem (sssp), which is the first half of the diameter problem in Figure 9. We use these two problems as running examples of the optimizations newly proposed in this section.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

These two programs are based on the following algorithm:

- First, the source vertex is assigned *True* (reAll) or 0 (sssp), and the other vertices are assigned *False* (reAll) or ∞ (sssp). For reAll, this value is the flag indicating whether each vertex is reachable or not at the current LSS. For sssp, this value is the tentative distance from the source vertex to each vertex at the current LSS.
- Then, each vertex sends the flag (reAll) or tentative distance (sssp) to its neighbors and updates its value if it receives *True* (reAll) or a shorter distance (sssp).
- The second step is repeated until all vertex values are no longer changed.

While these programs are clear and reasonable, they also suffer from the following inefficiency problems discussed in Section 2.1. Some communications are apparently unnecessary (it is sufficient to process only those vertices for which values are updated), and global barrier synchronization for every superstep may bring overhead. Moreover, for sssp, there is an additional source of inefficiency: the algorithm is essentially the Bellman–Ford algorithm, for which the time complexity is $O(n^2)$, where n is the size of the graph, and processing near-source vertices prior to distant ones as in Dijkstra's algorithm may reduce the amount of work to possibly $O(n \log n)$.

We developed a method for automatically removing these inefficiencies that incorporates four optimizations:

- Eliminate unnecessary communications. (Section 7.2)
- Inactivate vertices that do not need to be processed. (Section 7.3)
- Remove barrier synchronization, thereby enabling asynchronous execution. (Section 7.4)
- Introduce priorities for processing vertices. (Section 7.5)

These optimizations can be implemented by focusing on specific program patterns (Kato & Iwasaki, 2019), but this ad hoc approach is sensitive to the program details. Our proposed method is based on a more robust approach that uses *constraint solvers* for identifying possible optimizations. We discuss the use of two constraint solving methods: *quantifier elimination* (QE) (Caviness & Johnson, 1998) and *satisfiability modulo theories* (SMT) (de Moura & Bjørner, 2011). The former enables the use of arbitrary quantifier nesting and can generate the program fragments that are necessary for the optimizations. Therefore, it is suitable for formalizing optimizations. However, it is somewhat impractical because of its high computational cost. We thus use SMT solvers as a practical implementation method that captures typical cases.

The first two optimizations listed above were implemented in the Fregel compiler. Implementation of the other two is left for future work because they need a graph processing framework that supports asynchronous execution. Nevertheless, we discuss them here in consideration of the possibility that they may be lead to further optimizations.

7.1 Target programs for optimization

The targets for the optimizations are programs written using the ***fregel*** function. We refer to its second parameter (a step function) as *fStep* and assume that it is written in the

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 1. *Step functions for all-reachability and* sssp *problems*

	reStep	*ssspStep*
n	1	1
p_k	$p_1(pu) = True$	$p_1(pu) = True$
f_k	$f_1(e, pu) = pu\ .\hat{}\ rch$	$f_1(e, pu) = pu\ .\hat{}\ dist + e$
$\oplus_k$	$\oplus_1 = \|\|$	$\oplus_1 = `min`$
g	$g(pv, c_1) = RVal\ ((pv\ .\hat{}\ rch)\ \|\|\ c_1)$	$g(pv, c_1) = SVal\ ((pv\ .\hat{}\ dist)\ `min`\ c_1)$

$$
\begin{aligned}
fStep\ v\ \textbf{\textit{prev curr}} = \textbf{let}\ & c_1 = \oplus_1\,[\,f_1(e,\ \textbf{\textit{prev}}\ u)\mid (e,u) \leftarrow \textbf{\textit{is}}\ v,\ p_1(\textbf{\textit{prev}}\ u)\,] \\
& \vdots \\
& c_n = \oplus_n\,[\,f_n(e,\ \textbf{\textit{prev}}\ u)\mid (e,u) \leftarrow \textbf{\textit{is}}\ v,\ p_n(\textbf{\textit{prev}}\ u)\,] \\
\textbf{in}\ & g(\textbf{\textit{prev}}\ v,\ c_1, \ldots,\ c_n)
\end{aligned}
$$

Fig. 20. Target program for optimization.

form shown in Figure 20. In the program, f_i, p_i, and $\oplus_i$ $(1 \le i \le n)$, respectively, represent computation over each neighbor's value, the condition showing the necessity of sending the value, and the operator used for combining received values. Here, for convenience, ⟨aggOp⟩ in the Fregel's aggregation syntax (Figure 7) is represented by its commutative and associative binary operator $\oplus_i$. For example, the aggregation operation "*sum*" is represented by its binary operator "+". Function g denotes the calculation of the new value of a vertex. For simplicity, we assume the termination condition is ***Fix***, and only the ***is*** function is used as a generator. We discuss these limitations in Section 7.6.

The *fStep* corresponds to *reStep* for the reAll problem and *ssspStep* for the sssp problem, as presented in Table 1.

We use $\bar{u}$ and $\bar{\bar{u}}$ for the following meanings in this section:

- $\bar{u}$ denotes the current value of vertex u, and
- $\bar{\bar{u}}$ denotes the previous value of vertex u.

7.2 Eliminating unnecessary communications

Since accesses to a neighbor's information are compiled to message exchange, modifying the condition p_k and thereby avoiding unnecessary accesses reduces the amount of communication. In the following discussion, we focus on reducing communications caused by the computation of c_k. Our strategy is to formalize the situation in which optimization is possible and then to use constraint solvers to implement the optimization.

7.2.1 Formulation

Consider formulating the necessity of sending $\bar{u}$ to neighboring vertices. The following property naturally formulates the situation in which the sending of $\bar{u}$ does not affect computation on the destination vertex:

$$
\begin{aligned}
&\forall\, pv,\ e,\ c_1,\ \ldots,\ c_n\ . \\
&g(pv,\ c_1,\ \ldots,\ c_n) = g(pv,\ c_1,\ \ldots,\ c_{k-1},\ c_k \oplus_k f_k(e,\ \bar{u}),\ c_{k+1},\ \ldots,\ c_n)
\end{aligned}
\tag{7.1}
$$

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

For *reStep*, Property (7.1) is instantiated as:

$$\forall\, pv,\ c\ .\ RVal\ (pv\ .\hat{}\ rch\ ||\ c)\ =\ RVal\ (pv\ .\hat{}\ rch\ ||\ (c\ ||\ \bar{u}\ .\hat{}\ rch)).$$

This is equivalent to $\bar{u}\ .\hat{}\ rch\ =\ False$. It means that a vertex can skip message sending if its *rch* value is *False*.

For *ssspStep*, Property (7.1) is instantiated as:

$$\forall\, pv,\ e,\ c\ .\ SVal\ (pv\ .\hat{}\ dist\ `min`\ c)\ =\ SVal\ (pv\ .\hat{}\ dist\ `min`\ (c\ `min`\ (\bar{u}\ .\hat{}\ dist + e))).$$

This is equivalent to $\bar{u}\ .\hat{}\ dist\ =\ \infty$, which means that a vertex can skip message sending if its *dist* value is infinity.

This property avoids the sending of apparently useless messages, a solution for the first inefficiency problem described above. Note that the "value of useless" derived from Property (7.1) is the unit value of $\oplus_k$: *False* for *or* and ∞ for '*min*'. We call optimization on the basis of this property "unit values elimination."

For both reAll and sssp, even more message sending can be avoided. A vertex need not send a message if its *rch* (reAll) or *dist* (sssp) value is unchanged from the previous step. To capture this case, we need another formulation that takes the previous value into account. A vertex may be able to skip message sending if sufficient information had been sent at the previous step. The following formula captures this idea:

$$\begin{array}{l}\forall\, pv,\ e,\ c_1,\ \ldots,\ c_n,\ c'_1,\ \ldots,\ c'_n\ .\\ g(pv',\ c'_1,\ \ldots,\ c'_n)\ =\ g(pv',\ c'_1,\ \ldots,\ c'_{k-1},\ c'_k\ \oplus_k\ f_k(e,\ \bar{u}),\ c'_{k+1},\ \ldots,\ c'_n)\\ \textbf{where}\ pv'\ =\ g(pv,\ c_1,\ \ldots,\ c_{k-1},\ c_k\ \oplus_k\ f_k(e,\ \bar{\bar{u}}),\ c_{k+1},\ \ldots,\ c_n)\end{array}\qquad (7.2)$$

The necessity of $\bar{u}$ is checked on the basis of the premise that the message-receiving vertex (which has value pv') took into account the previous value $\bar{\bar{u}}$ of the message-sending vertex. We call this optimization "redundant values elimination."

For *reStep*, Property (7.2) is instantiated to

$$\begin{array}{l}\forall\, pv,\ c,\ c'\ .\ RVal\ (pv'\ .\hat{}\ rch\ ||\ c') = RVal\ (pv'\ .\hat{}\ rch\ ||\ (c'\ ||\ \bar{u}\ .\hat{}\ rch))\\ \qquad\textbf{where}\ pv'\ =\ RVal\ (pv\ .\hat{}\ rch\ ||\ (c\ ||\ \bar{\bar{u}}\ .\hat{}\ rch)).\end{array}$$

This means that a vertex can skip communication when $\bar{u}\ .\hat{}\ rch\ =\ \bar{\bar{u}}\ .\hat{}\ rch$, that is, the *rch* values of $\bar{u}$ and $\bar{\bar{u}}$ are the same.

Similarly for *ssspStep*, Property (7.2) is instantiated to

$$\begin{array}{l}\forall\, pv,\ e,\ c,\ c'\ .\\ SVal\ (pv'\ .\hat{}\ dist\ `min`\ c') = SVal\ (pv'\ .\hat{}\ dist\ `min`\ (c'\ `min`\ (\bar{u}\ .\hat{}\ dist + e)))\\ \textbf{where}\ pv'\ =\ SVal\ (pv\ .\hat{}\ dist\ `min`\ (c\ `min`\ (\bar{\bar{u}}\ .\hat{}\ dist + e))).\end{array}$$

This is equivalent to $\bar{u}\ .\hat{}\ dist\ \geq\ \bar{\bar{u}}\ .\hat{}\ dist$: a vertex can skip communication when the current *dist* value is not smaller than the previous one. Since the current *dist* value is never larger than the previous one, this is essentially equivalent to $\bar{u}\ .\hat{}\ dist\ =\ \bar{\bar{u}}\ .\hat{}\ dist$.

7.2.2 Remarks on implementation

We could implement this optimization by dynamically checking Properties (7.1) and (7.2) for each vertex. However, because these properties consist of quantifiers, their evaluation is likely impossible or very slow. To obtain efficient codes, we need a method for

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

synthesizing a simple (especially quantifier-free) formula that is equivalent to (or expressing a sufficient condition of) the property. For this purpose, we can use constraint solvers.

QE translates a formula into a quantifier-free equivalent one. For example, it may translate $\forall x.\ x^2 + ax + b \geq 0$ into $4b - a^2 \geq 0$. While QE is theoretically ideal for our purpose, QE solvers are impractical for three reasons. First, there are only a few formal systems for which QE procedures are known. Second, QE procedures are usually very slow. Third, current implementations of QE tend to be experimental. Nevertheless, it is worthwhile to formulate the optimizations as QE, because these problems may one day be solved.

As a more practical implementation, we propose using SMT instead of QE. Given a closed formula consisting of only one kind of quantifier, SMT checks (i.e., does not translate) whether it is satisfiable. For example, it may answer "yes" for $\forall x,\ a.\ x^2 + ax + a^2 \geq 0$. Efficient SMT solvers have recently been developed and are now used in many applications.

There are two problems in using SMT for checking Properties (7.1) and (7.2). They contain free variables, $\bar{u}$ and $\bar{\bar{u}}$, and moreover, SMT solvers are unable to synthesize a simple formula. To overcome these problems, we prepare *templates* of simple reasonable formulae, such as $\bar{u} = \infty$ (e.g., $\bar{u}\ .\hat{}\ dist = \infty$) or $\bar{u} = \bar{\bar{u}}$. If the SMT solver guarantees that a template is a sufficient condition of these properties, we insert the negation of the template into p_k. The effectiveness of this approach relies on the generality of the template.

The most common case that satisfies Property (7.1) is one in which the message value is the unit of $\oplus_k$. Since Fregel's syntax allows only a limited operator such as *minimum* and *or* as ⟨aggOp⟩, we can know the unit value of an ⟨aggOp⟩ without using constraint solvers. However, if a user-defined combining operation were able to be specified as ⟨aggOp⟩, we would use an SMT solver to check whether one of the template values is the unit of the operation.

For the case of Property (7.2), several templates can be considered. We believe that comparing values in $\bar{u}$ and $\bar{\bar{u}}$ captures most practical cases.

Considering sssp, for Property (7.1), we have already found that sending ∞ is unnecessary because it is the unit of '*min*'. For Property (7.2), we instruct an SMT solver to check the following formula:

$$\begin{array}{l}\forall\ \bar{u},\ \bar{\bar{u}},\ pv,\ e,\ c,\ c'\ .\\ \bar{u}\ =\ \bar{\bar{u}}\ \longrightarrow\ SVal\ (pv'\ .\hat{}\ dist\ `min`\ c') = SVal\ (pv'\ .\hat{}\ dist\ `min`\ (c'\ `min`\ (\bar{u}\ .\hat{}\ dist + e)))\\ \qquad\qquad \textbf{where}\ pv'\ =\ SVal\ (pv\ .\hat{}\ dist\ `min`\ (c\ `min`\ (\bar{\bar{u}}\ .\hat{}\ dist + e))).\end{array}$$

The solver verifies the condition. We thus modify the program as follows. We instruct each vertex to check and remember the truth of the template. Then, we modify p_1 so that it checks the remembered truth. Letting *notChanged* be the vertex variable for remembering the truth of the template, we modify *ssspStep* to a code that is essentially equivalent to the one presented in Figure 21.

7.3 Inactivating vertices

Next, we discuss inactivating vertices. A vertex u is inactivated if the following condition holds; unless the vertex receives a message, its value $\bar{u}$ does not change and it need not send a message. The optimization condition is thus formalized as:

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

```
data SVal = SVal { dist :: Int, notChanged :: Bool }
sssp g =
  let ssspInit v = SVal (if vid v == 1 then 0 else ∞) False ;
      ssspStep v prev curr =
        let dist' = prev v .^ dist `min`
                    minimum [prev u .^ dist + e | (e, u) ← is v, not (prev u .^ notChanged)]
        in SVal dist' (dist' == prev v .^ dist)
  in fregel ssspInit ssspStep Fix g
```

Fig. 21. sssp program for eliminating redundant communications.

$$\left(\bigwedge_{1\le i\le n} \neg p_i(\bar{u})\right) \wedge (g(\bar{u}, \iota_1, \ldots, \iota_n) = \bar{u}), \tag{7.3}$$

where ι_i $(1 \le i \le n)$ is the unit of $\oplus_i$ and corresponds to the absence of messages. In Property (7.3), "$\neg p_i(\bar{u})$" corresponds to the fact that the current vertex need not send a message for the i-th aggregation, and "$g(\bar{u}, \iota_1, \ldots, \iota_n) = \bar{u}$" means that the vertex's value is unchanged unless the vertex received a message. Since this property contains no quantifier, this optimization can be implemented without the use of a constraint solver. We call this optimization "vertices inactivation."

For effective vertices inactivation, the predicate p_i, which specifies the necessity of sending messages, should result in "false" as much as possible. Hence, vertices inactivation should be applied after communication reduction optimization described in Section 7.2.

For sssp, Property (7.3) is instantiated to

$$\bar{u} \mathbin{.\hat{}} notChanged \wedge (SVal\ d'\ (d' == \bar{u} \mathbin{.\hat{}} dist)) = \bar{u} \textbf{ where } d' = \bar{u} \mathbin{.\hat{}} dist\ `min`\ \infty,$$

which is equivalent to $\bar{u} \mathbin{.\hat{}} notChanged$. In short, a vertex can be inactivated if its value is the same as before.

7.4 Removing barriers

Recall that the execution of Fregel is based on the BSP model. Each local computation is followed by barrier synchronization. Though this makes program behaviors deterministic and deadlock-free, barriers can make execution slower, especially when there are many computational nodes. For most graph algorithms including reAll and sssp for which asynchronous barrier-less execution and synchronous execution yield the same result, barrier synchronization is unnecessary.

The flexibility of asynchronous execution enables further optimizations such as *vertex splitting* (also known as vertex mirroring) (Yan *et al.*, 2015; Verma *et al.*, 2017). Practical graphs often contain vertices that have too many edges, and such vertices form a bottleneck in vertex-centric computation. Vertex splitting resolves the bottleneck by splitting these vertices and distributing their edges among the computational nodes. With synchronous execution, vertex splitting requires an additional superstep to merge the messages sent to the split vertices. With asynchronous execution, an additional superstep is unnecessary because message delay does not matter. Another possible optimization is to repeatedly process vertices in the same computational node before sending messages to other nodes. This

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

optimization is related to subgraph-centric (or neighborhood-centric) approaches (Tian *et al.*, 2013; Quamar *et al.*, 2016) in which subgraphs rather than vertices are the target of parallel processing.

7.4.1 Formulation

We have developed a method that automatically guarantees equivalence between synchronous and asynchronous execution. We first present the following lemma.

Lemma 7.1. *For functions h and h' and a binary relation $\preceq$, three conditions are assumed:*

- ***Monotonicity of** h:* $\forall x, y.\ (x \preceq y) \to (h(x) \preceq h(y))$.
- ***Ordering of** h **and** h':* $\forall x.\ (x \preceq h'(x)) \wedge (h'(x) \preceq h(x))$.
- ***Antisymmetry of** $\preceq$:* $\forall x, y.\ (x \preceq y \wedge y \preceq x) \to (x = y)$.

*Then, $h^*(x) = h^*(h'(x))$ holds for any x, where h^* is defined by $h^*(x) =$ **if** $h(x) = x$ **then** x **else** $h^*(h(x))$.*

Proof From the monotonicity and the ordering of h and h', we have $x \preceq h'(x) \preceq h(x) \preceq h(h'(x))$. Now let $h^0(x) = x$ and $h^n(x) = h^{n-1}(h(x))$ for $n > 1$. By induction, we have $h^n(x) \preceq h^n(h'(x)) \preceq h^{n+1}(x)$ for any n. When $h^*(x)$ terminates, there exists an integer m such that $h^*(x) = h^m(x) = h^{m+1}(x)$. Then, $h^m(x) = h^m(h'(x)) = h^{m+1}(x)$ follows from the inequality mentioned above and the antisymmetry of $\preceq$, and hence $h^*(h'(x)) = h^m(h'(x))$. When $h^*(x)$ is non-terminating, so is $h^*(h'(x))$. We prove it by contradiction. Suppose $h^m(h'(x)) = h^{m+1}(h'(x))$ for some m. Recall that $h^m(h'(x)) \preceq h^{m+1}(x) \preceq h^{m+1}(h'(x)) \preceq h^{m+2}(x) \preceq h^{m+2}(h'(x))$ holds. This inequality and $h^m(h'(x)) = h^{m+1}(h'(x)) = h^{m+2}(h'(x))$ imply $h^{m+1}(x) = h^{m+2}(x)$, which contradicts the non-termination of $h^*(x)$. □

We apply Lemma 7.1 as follows. We regard h as a complete one-step processing of the graph. Similarly, we regard h' as a partial processing in which some vertices and messages are skipped. We regard asynchronous execution as a series of partial processings. Lemma 7.1 guarantees that a partial processing does not change the result; then, by induction, asynchronous execution does not change the result as well.

Lemma 7.1 requires an appropriate binary relation, $\preceq$. From the ordering between h and h', a natural candidate is comparison of the progress in computation: $g_1 \preceq g_2$ indicates that graph g_2 can be obtained by processing computation from g_1. Another requirement is bridging the gap between graph processing and vertex processing. While h, h', and $\preceq$ deal with graphs, we would like to consider vertex-processing functions. The following lemma bridges the gap. For simplicity, we assume that the *fStep* function contains only one access to a neighbor's information by a combining operator $\oplus$.

Lemma 7.2. *For fStep, let $\preceq$ be a binary relation defined by $x \preceq y \iff (\exists m.\ y = g(x, m))$. Three conditions are assumed:*

- $\forall x, m, m'.\ g(x, m \oplus m') = g(g(x, m), m')$.
- $\forall x, y.\ (x \preceq y \wedge y \preceq x) \to (x = y)$.
- $\forall x, y, z.\ (x \preceq y) \to (g(z, x) \preceq g(z, y))$.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Then, h_{fStep}, h'_{fStep}, and $\preceq_{\mathrm{G}}$ satisfy the premise of Lemma 7.1: the first two are respectively complete and partial one-step processing (here, "partial" means processing some of the vertices using some of the messages) over the graph by fStep and the last one compares graphs on the basis of vertex-wise comparison using $\preceq$.

Proof [proof sketch] The first condition and the definition of $\preceq$ guarantee the ordering between h_{fStep} and h'_{fStep}. The antisymmetry of $\preceq_{\mathrm{G}}$ easily follows from the second condition. The third condition together with the first one and the commutativity of $\oplus$ guarantees the monotonicity of h_{reStep}. □

The first condition of Lemma 7.2 can be taken to mean that message delay is not harmful. This is a natural requirement for asynchronous execution.

For sssp, the definition of the relation $\preceq$ is instantiated as:

$$x \preceq y \iff \exists w.\, x\ .\hat{}\ dist\ \text{`}min\text{`}\ w\ .\hat{}\ dist = y,$$

which is equivalent to $x\ .\hat{}\ dist\ \geq\ y\ .\hat{}\ dist$. Therefore, confirming the three conditions is easy.

7.4.2 Remarks on Implementation

The first and second conditions can be checked using either QE or SMT. Note that the second is equivalent to $\forall x,\ m,\ w\,.\,(g(g(x,m),w) = x) \to (g(x,m) = x)$, where y is expressed as $g(x,m)$. Since the definition of $\preceq$ contains an existential quantifier, the third condition cannot be directly checked using SMT. When using an SMT solver, we may instead check the following sufficient condition:

$$\forall x,\ y,\ z\,.\,(x \preceq y) \to (g(g(z,x),y) = g(z,y)).$$

This can be read to mean that the previous result, x, can be "overwritten" by the newer result, y. This is also natural in asynchronous execution.

7.5 Prioritized execution

Another interesting optimization that asynchronous execution enables is prioritized execution (Prountzos *et al.*, 2015; Cruz *et al.*, 2016; Liu *et al.*, 2016). For example, in sssp, a prioritized execution may more intensively process vertices nearer the source, like Dijkstra's algorithm.

Prioritized execution typically focuses on vertices for which the values are *nearer* the final outcome and thus likely contribute to the final outcome for other vertices. Therefore, it is natural to use $\preceq$ defined in Lemma 7.1, which essentially compares progress in computation, as a priority for processing vertices. For sssp, $\preceq$ is equivalent to $\geq$ and thus is a perfect candidate.

However, there are two problems with using $\preceq$ for prioritized execution. First, since its definition contains an existential quantifier, it is essentially not executable unless QE is used. The other, more essential problem is that $\preceq$ may not be a linear order. Nonlinear orders are less effective for prioritized execution and make it difficult to process vertices efficiently using priority queues. A practical solution to these problems is to check whether

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

a known linear order, $\geq$ for example, is consistent with $\preceq$, that is, $\forall x, y.\ (x \preceq y) \rightarrow (x \geq y)$. If it is, the linear order can be used for prioritization. The condition can be checked by an SMT solver.

7.6 Limitations and generalization

We have assumed that information read from neighbors is expressed using the ***is*** generator. Use of other kinds of generators, including the one for expressing an aggregator, generally does not introduce any difficulty. We did not assume anything about communication except that the communication topology does not change during computation.

A notable exception is the case of vertex inactivation. Since the results of aggregation may change regardless of message arrival, if the k-th communication is an aggregator, the following condition should be checked instead of Property (7.3):

$$\left(\bigwedge_{1 \leq i \leq n} \neg p_i(\bar{u})\right) \wedge (\forall w_k.\ g(\bar{u}, \iota_1, \ldots, w_k, \ldots, \iota_n) = \bar{u}).$$

Namely, the vertex value should not change regardless of the aggregator's value if the vertex does not receive a message. Since it contains a quantifier, unless QE is used, an executable sufficient condition is needed. A natural candidate is the following condition:

$$\forall \bar{u}.\ \left(\bigwedge_{1 \leq i \leq n} \neg p_i(\bar{u})\right) \rightarrow (\forall w_k.\ g(\bar{u}, \iota_1, \ldots, w_k, \ldots, \iota_n) = \bar{u}).$$

If it holds, a vertex having $\bar{u}$ can be inactivated if $(\bigwedge_{1 \leq i \leq n} \neg p_i(\bar{u}))$ holds. The condition can be checked using SMT.

We have considered only a certain form of programs. For example, termination conditions other than ***Fix*** and second-order graph functions other than ***fregel*** were neglected. This limitation is theoretically inconsequential. As discussed in Section 6.2, the Fregel compiler normalizes other forms of programs into the one in Figure 15. Nevertheless, from the practical perspective, since the normalization complicates programs, it is questionable whether normalized programs can be effectively optimized.

7.7 Implementation of optimizations

We implemented unit values elimination and redundant values elimination described in Section 7.2 and vertices inactivation described in Section 7.3 in the Fregel compiler. We left implementation of the last two optimizations described in Sections 7.4 and 7.5 as future work because the target frameworks of the current Fregel compiler are based on synchronous execution.

For the unit values elimination optimization, as described in Section 7.2.2, we did not use an SMT solver because specifiable message-combining operators are limited, and their unit values to be eliminated can be easily determined.

For both the redundant values elimination and vertices inactivation optimization, we used the Z3 SMT solver.[2] Implementation using Z3 is mostly straightforward. It is worth noting that the units for minimum and maximum, $-\infty$ and ∞, are necessary for vertices

[2] https://z3.codeplex.com/.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

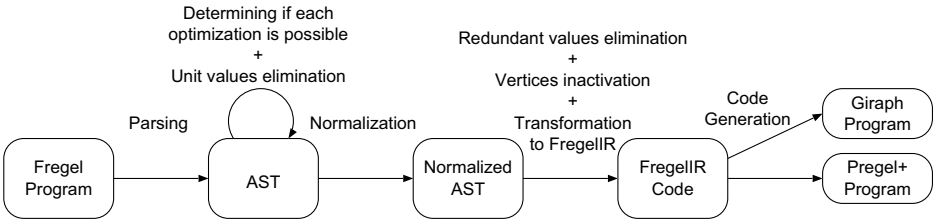


Fig. 22. Optimizations in compilation flow of Fregel program.

inactivation. We prepared numerals with $-\infty$ and ∞ and used them instead of the ones conventionally used, such as *Int*.

Figure 22 illustrates how the proposed optimizations are carried out during compilation of a Fregel program. After parsing the program and constructing an AST for the program, the compiler checks in turn on the basis of the optimizing options given by the user whether or not each specified optimization can be applied.

First, the compiler checks unit values elimination by identifying a combining operator used in a comprehension and modifies its AST so as to contain checking code at the top of its predicate part, if this optimization is possible. For example, the comprehension part of *reStep* is modified to

$$or\ [\ \textbf{\textit{prev}}\ u\ .\hat{}\ rch\ |\ (e, u) \leftarrow \textbf{\textit{is}}\ v,\ \textbf{\textit{prev}}\ u\ .\hat{}\ rch\ /{=}\ False\ ].$$

Next, the compiler checks the possibility of redundant values elimination by generating a Z3 program that corresponds to Property (7.2), invoking Z3, and storing the result, that is, *True* (possible) or *False* (impossible), in a flag variable. Similarly, the compiler checks the possibility of vertices inactivation by using Z3 on the basis of Property (7.3) and stores the result in another flag variable. These flag variables are referred to during transformation from a normalized AST to FregelIR code, resulting in optimized FregelIR code.

If redundant values elimination is possible, the compiler extends the vertex record so as to contain a *notChanged* variable that records whether the vertex value of the current LSS is the same as that of the previous LSS. In addition, the compiler generates code that sets *notChanged* properly and eliminates message sending to neighboring vertices if *notChanged* on a vertex is *True*.

If vertices inactivation optimization is possible, the compiler generates the following code:

- Instead of performing an aggregation to detect termination of the computation, the generated code refers to *notChanged* and votes to halt if its value is *True*.
- Since an aggregation for termination detection is removed, it is not necessary to separate the computations before and after the aggregation into different supersteps. Thus, the generated code executes these computations successively in a single superstep.

8 Evaluation

In this section, we will report our experimental results on the performance of Fregel programs. We used as the parallel computation hardware a PC cluster consisting of 16 nodes,

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

each of which had a four-core CPU (Intel®Core™i5-6500) and 16 GB memory. Thus, the maximum number of worker processes was 64. The software consisted of Ubuntu 18.04.5 LTS (x86_64), JDK 1.8.0_131-b11, Hadoop 1.2.1, Giraph 1.2.0, and Pregel+ (for Hadoop 1.x). We used Giraph and Pregel+ as our compilation targets.

Six computations were used as benchmarks:

- sssp Single-source shortest path (the first part of the diameter computation in Figure 9).
- reAll All-reachability from a given node (Figure 8(a)).
- re100 100-reachability from a given node (Figure 8(b)).
- reRanking Reachability with ranking (Figure 10).
- diameter Diameter from a given node (Figure 9).
- scc Strongly connected components (Figure 11).

For each benchmark, we implemented a Fregel program and two kinds of handwritten programs in the compilation target (Giraph or Pregel+). This resulted in four kinds of programs for each benchmark:

- **handwc** Handwritten program with the use of combiners. It was directly written by hand in Java (Giraph) or C++ (Pregel+). The implementation of each benchmark is explained below.
- **hand** Handwritten program without combiners. The code was the same as for **handwc**, but without combiners.
- **naive** Program generated by a naive compilation from the Fregel program.
- **opt** Program generated by a compilation with all available optimizations from the Fregel program.

Here, *combiners* are objects used to combine messages delivered to a vertex when individual (raw) messages are not important. A message-combining mechanism using combiners is provided by both Giraph and Pregel+. Combining generally improves program efficiency.

The handwritten code for Pregel+ was as follows:

- sssp Pregel+'s sample code with small modifications. Each active vertex did the following in a superstep: (1) compute the minimum value of the messages received, (2) update its current distance if necessary, (3) send the distance to its neighbors if it was updated, and (4) vote to halt. Only the source vertex was active at the beginning.
- reAll Almost the same code as for sssp, but Boolean values were used instead of numbers.
- re100 Made by adding two modifications to reAll: (1) a summation aggregator was added to count the number of reached vertices, and (2) active vertices did not vote to halt unless the aggregator's value exceeded 100.
- reRanking Similar to re100 but another mechanism was used to stop the computation. Two aggregators were used: a summation aggregator was used to count the number of reached vertices, and a logical disjunction aggregator was used to check if there was a newly reached vertex. Active vertices voted to halt when the aggregator returned false (i.e., there was no newly reached vertex in the previous superstep).

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

In addition, two fields were added to each vertex: one for storing the rank and one for indicating whether it was newly reached in the superstep.

- diameter Since this computation performed two different vertex-centric computations, each vertex used two fields to control the switching of the computation phases: one for storing the current computation phase and one for indicating whether its value was updated in the superstep. The vertex first executed, as the first phase, the same computation as reRanking until the disjunction aggregator on the second field returned "false." Then, instead of voting to halt, it switched its phase to the second, and executed the second computation similar to that for sssp.
- scc Similar to diameter, the same mechanism was used to switch between the forward and backward computation phases. Both phases did the same computation as that for sssp, but the backward phase used the reversed edges.

For every benchmark, the implementation strategy of the handwritten code for Giraph was the same as that for Pregel+'s.

The input graphs were three random graphs based on the Watts–Strogatz model (Watts & Strogatz, 1998) with three parameters: N (the number of vertices), K (the mean degree), and P (the probability of reconnection):

- **ws10m2** $N = 10 \times 10^6$, $K/2 = 2$, $P = 0.2$
- **ws10m4** $N = 10 \times 10^6$, $K/2 = 4$, $P = 0.2$
- **ws20m2** $N = 20 \times 10^6$, $K/2 = 2$, $P = 0.2$

We used the Watts–Strogatz model because it generates graphs with the *small-world* property, that is, a high clustering coefficient and a low average shortest path length among vertices, which is often seen in real-world graphs such as social networks. **ws10m2** is the smallest input graph with 10 M vertices and 40 M edges. **ws10m4** has more edges and the same number of vertices, so a comparison of the results for **ws10m2** and **ws10m4** reveals the effect of an increase in degree. Similarly, **ws20m2** has more vertices and the same average degree, so a comparison of the results for **ws10m2** and **ws20m2** reveals the effect of an increase in the number of vertices.

8.1 Compilation target: Giraph

This section reports the experimental results for Giraph.

Tables 2–7 show the measured execution times (the median of five runs) for the programs with 4, 8, 16, 24, 32, 48, and 64 worker processes as well as the number of supersteps ("# SS") and the number of messages ("# messages"). Since the input graphs were too big for runs on a single worker process, we selected four as the minimum number of processes. Note that for each program, the number of supersteps equalled the number of messages for all runs. Also note that the number of messages was counted *before* the use of combiners; the number of messages for **handwc** was the same as that for **hand**.

Figures 23 and 24 show the execution time of each program relative to that of **handwc** with 4 and 64 worker processes, respectively.

The naively compiled Fregel program **naive** was about 4–6 times slower than **handwc** with 4 worker processes and about 2–3 times slower with 64 worker processes. This was due to greater numbers of messages and supersteps. The number of messages was about

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

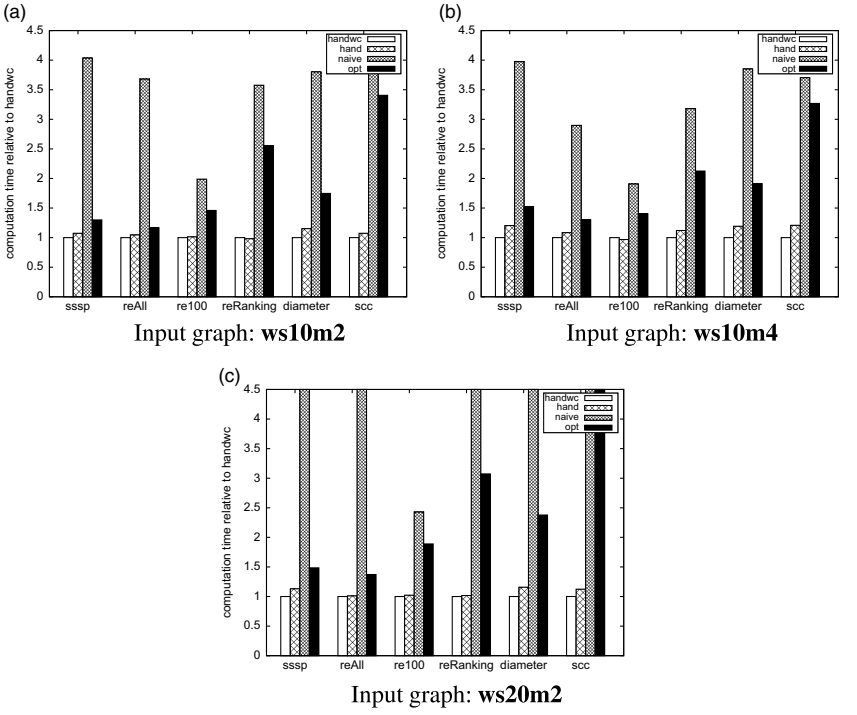


Fig. 23. Computation times compared with that for **handwc** with four worker processes on Giraph.

2–4 times more for scc and diameter, about 10–25 times more for sssp, reAll, and reRanking, and much more for re100, which needed only a few vertices to be active. The number of supersteps was four times more for scc, which was complex enough to need many phases in the normalized program (Section 6.2), and twice as many for the other computations.

The Fregel program **opt** (compiled with the proposed optimizations) achieved better performance than **naive**. The message reduction and vertex inactivation optimizations worked especially well to make the number of messages the same as that of **handwc**. In addition, the simple optimization to run multiple phases in a single superstep made the number of supersteps the same as that of **handwc**. As a result, **opt** was about 1.5 times slower than **handwc** with 4 worker processes and only 1.1 times slower with 64 worker processes. The remaining inefficiency was due to (1) **opt** not using combiners while **handwc** did and to (2) each vertex in **opt** having more data fields, for example, the phase number and total number of supersteps, than **handwc**.

For re100, **opt** used fewer messages and more supersteps than **handwc**. This was because **handwc** sent values to the aggregator and messages to its neighbors simultaneously in a single superstep to reduce the total number of supersteps, while **opt** performed these communications separately in two successive supersteps to reduce the number of messages.

The optimizations also worked in the more complex computations for reRanking, diameter, and scc, in which a part of the whole computation was improved by the proposed optimizations so that **opt** had in general fewer messages and supersteps than **naive**.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

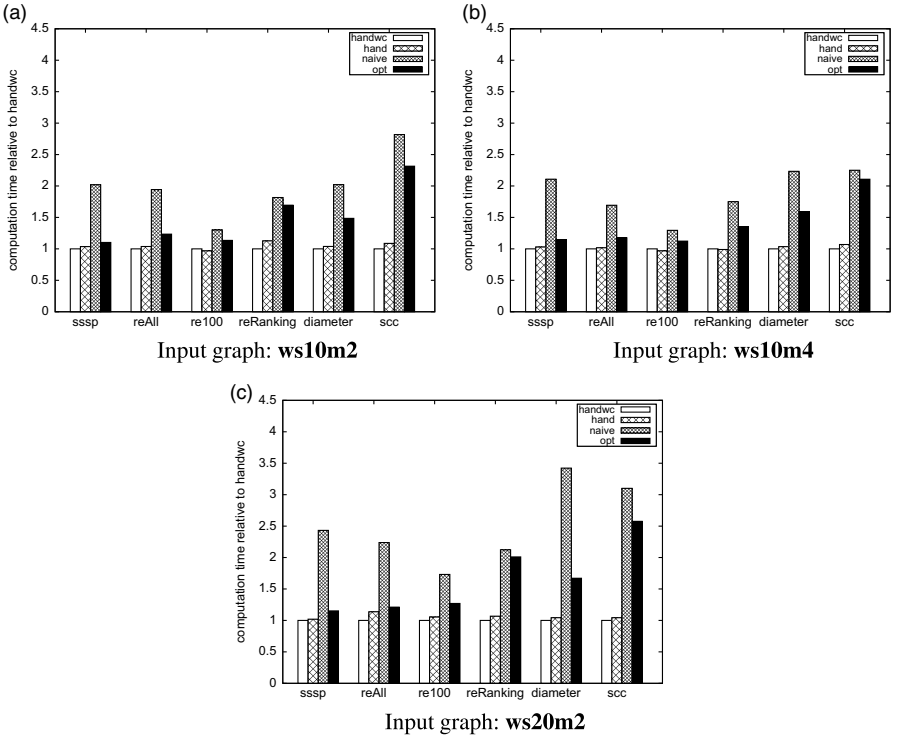


Fig. 24. Computation times compared with that for **handwc** with 64 worker processes on Giraph.

Figures 25–30 show the parallel performance, that is, the ratio of the actual parallel speedup to its ideal value: $(t_4/t_p)/(p/4)$, where t_p is the execution time with p worker processes. First, the parallel performances of both **naive** and **opt** were not worse than that of **handwc**. In some cases, **naive** and **opt** achieved superlinear performance (> 1.0) when the number of worker processes was not large. This was because a vertex in **naive** and **opt** had more data than **handwc** and because there was a lack of memory when running on a small number of worker processes. In general, their performance improved as the input graph became larger.

To sum up, the proposed optimizations achieved reasonably good performance for both simple and complex computations.

Finally in this section, we compare memory consumption. Basically, the programs compiled from Fregel code (**naive** and **opt**) used more memory than the handwritten versions (**handwc** and **hand**). Table 8 shows the memory footprints of the vertex data fields, excluding those defined in the base class of vertices.

In the handwritten versions (**handwc** and **hand**), every vertex held only user-defined fields: 4 bytes for an integer for the shortest distance in sssp, 1 byte for the Boolean value for the flag in reAll, 12 bytes for three integers for the rank, the diameter, and the phase (1 or 2) in diameter, and so on. Fregel's naively compiled program (**naive**) needed an additional 17–51 bytes for each vertex, which included

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 2. *Execution times of* sssp *with 4–64 worker processes on Giraph (in seconds)*

(a) Input graph: **ws10m2**

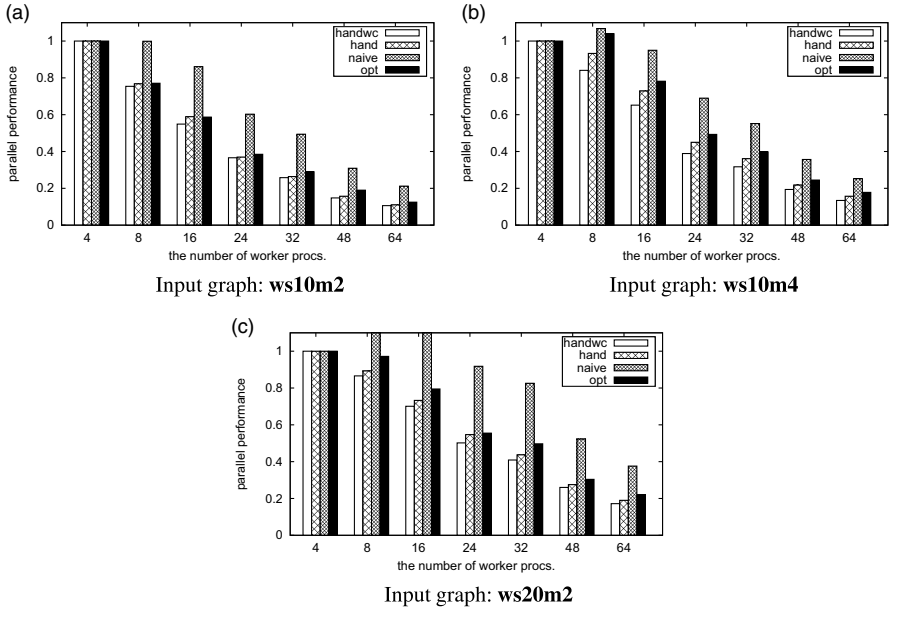
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	60.7	40.3	27.7	27.7	29.4	34.1	35.8	49	174,593,726
hand	65.3	42.5	27.7	29.4	30.9	34.7	37.1	49	174,593,726
naive	245.2	122.7	71.2	67.7	62.1	66.2	72.3	98	1,920,000,000
opt	79.0	51.2	33.6	34.2	34.0	34.6	39.5	49	174,593,726

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	76.7	45.6	29.4	32.9	30.2	32.9	35.7	44	521,880,562
hand	92.4	49.5	31.7	34.2	32.0	35.3	36.8	44	521,880,562
naive	304.8	142.7	80.2	73.6	69.1	71.1	75.4	88	3,440,000,000
opt	116.6	56.1	37.3	39.4	36.6	39.7	41.0	44	521,880,562

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	116.7	67.4	41.6	38.7	35.7	37.4	42.5	49	342,144,658
hand	131.9	73.9	45.0	40.2	37.8	40.0	43.3	49	342,144,658
naive	621.0	266.3	127.2	112.7	94.0	98.9	103.3	98	3,840,000,000
opt	173.3	89.2	54.5	52.1	43.6	47.4	49.0	49	342,144,658



Input graph: **ws10m2** Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 25. Parallel performance of sssp on Giraph.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 3. *Execution times of reAll with 4–64 worker processes on Giraph (in seconds)*

(a) Input graph: **ws10m2**

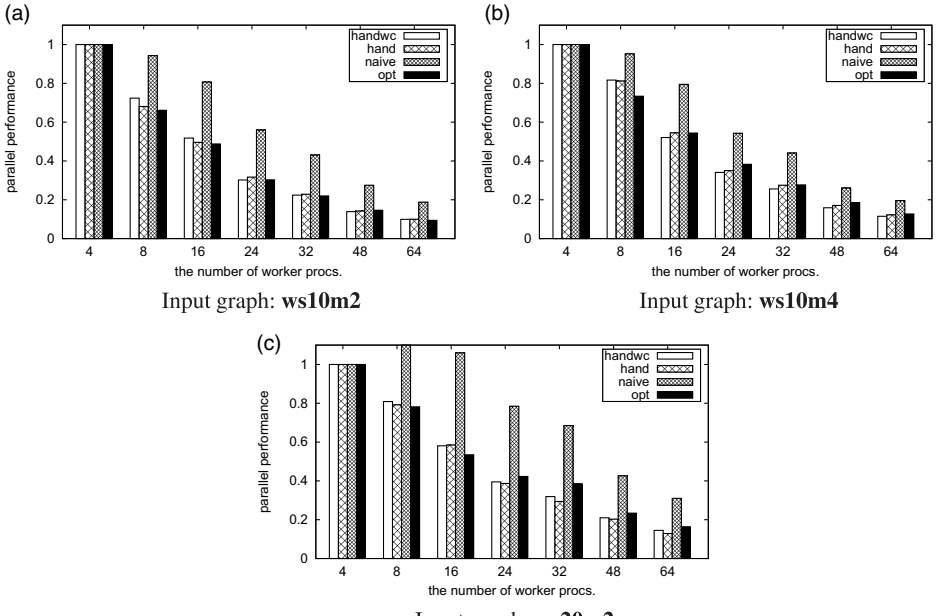
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	40.3	27.8	19.4	22.2	22.4	24.1	25.4	28	40,000,000
hand	42.1	30.9	21.2	22.2	23.0	24.7	26.4	28	40,000,000
naive	148.2	78.6	45.9	44.1	42.9	44.9	49.4	56	1,080,000,000
opt	47.0	35.5	24.1	25.9	26.7	26.8	31.4	28	40,000,000

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	40.1	24.5	19.2	19.6	19.5	20.9	21.9	15	80,000,000
hand	43.4	26.7	19.9	20.7	19.7	21.2	22.2	15	80,000,000
naive	116.1	61.0	36.5	35.6	32.9	37.0	37.0	30	1,120,000,000
opt	52.2	35.6	24.0	22.7	23.6	23.4	25.7	15	80,000,000

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	70.9	43.8	30.5	29.9	27.8	28.2	30.6	28	80,000,000
hand	71.7	45.2	30.6	30.9	30.5	29.4	34.8	28	80,000,000
naive	339.4	154.1	80.1	72.1	62.0	66.4	68.5	56	2,160,000,000
opt	97.3	62.2	45.5	38.3	31.5	34.7	37.1	28	80,000,000



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 26. Parallel performance of reAll on Giraph.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 4. *Execution times of* re100 *with 4–64 worker processes on Giraph (in seconds)*

(a) Input graph: **ws10m2**

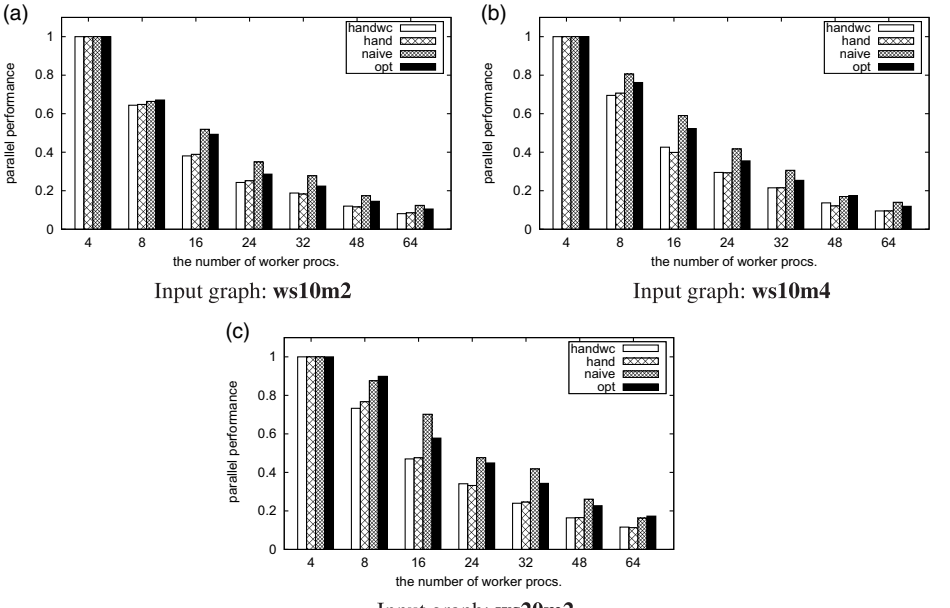
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	24.3	18.8	15.9	16.6	16.1	16.8	18.6	7	577
hand	24.6	19.0	15.8	16.3	16.8	17.7	18.0	7	577
naive	48.3	36.4	23.3	23.0	21.7	23.1	24.2	12	200,000,000
opt	35.5	26.4	18.0	20.7	19.8	20.3	21.1	12	272

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	27.8	20.0	16.3	15.7	16.2	16.9	18.3	5	2,366
hand	27.0	19.1	16.9	15.3	15.6	18.5	17.7	5	2,366
naive	53.1	32.9	22.5	21.2	21.7	26.0	23.7	8	240,000,000
opt	39.1	25.7	18.7	18.4	19.2	18.8	20.6	8	516

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	36.2	24.7	19.3	17.7	18.9	18.5	19.5	6	462
hand	37.0	24.1	19.5	18.6	18.8	18.8	20.5	6	462
naive	88.0	50.3	31.3	30.8	26.3	28.2	33.7	10	320,000,000
opt	68.4	38.0	29.6	25.4	24.9	25.1	24.7	10	205



Input graph: **ws10m2** Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 27. Parallel performance of re100 on Giraph.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 5. *Execution times of reRanking with 4–64 worker processes on Giraph (in seconds)*

(a) Input graph: **ws10m2**

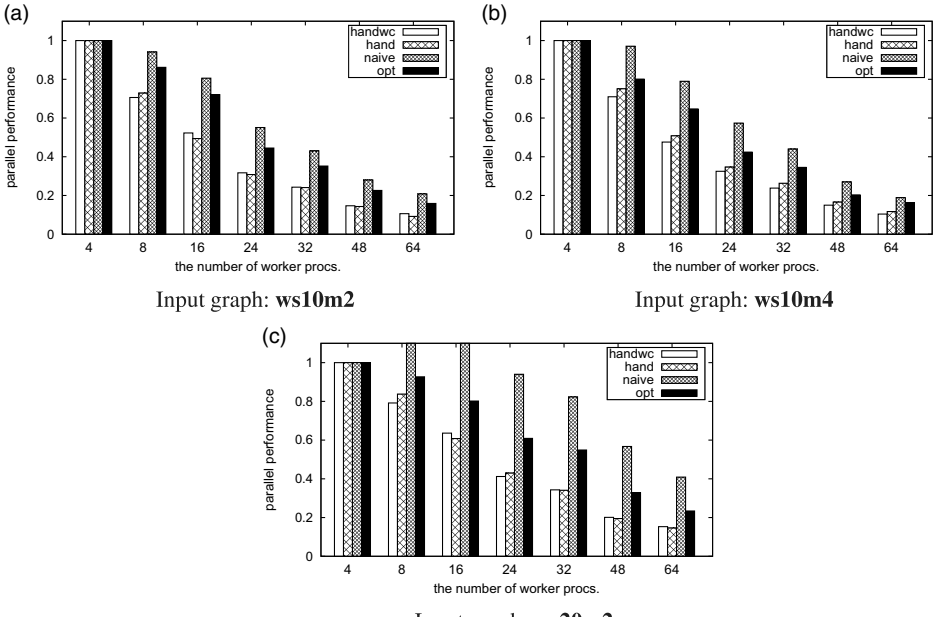
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	44.2	31.3	21.1	23.2	22.7	25.1	26.1	29	40,000,000
hand	43.3	29.7	21.9	23.5	22.5	25.2	29.4	29	40,000,000
naive	157.9	83.9	49.0	47.8	45.9	47.0	47.4	56	1,080,000,000
opt	112.8	65.4	39.1	42.2	40.0	41.5	44.3	56	357,244,816

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	39.4	27.8	20.7	20.2	20.7	22.0	23.8	16	80,000,000
hand	44.2	29.4	21.7	21.2	21.0	22.2	23.5	16	80,000,000
naive	125.5	64.6	39.7	36.5	35.7	38.8	41.6	30	1,120,000,000
opt	83.8	52.3	32.4	32.9	30.3	34.4	32.1	30	310,250,240

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	78.0	49.3	30.7	31.6	28.4	32.4	31.9	29	80,000,000
hand	79.4	47.4	32.6	30.7	29.1	33.9	34.0	29	80,000,000
naive	442.7	171.2	88.8	78.5	67.3	65.0	67.6	56	2,160,000,000
opt	239.7	129.3	74.8	65.6	54.6	61.0	64.0	56	719,102,699



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 28. Parallel performance of reRanking on Giraph.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 6. *Execution times of* diameter *with 4–64 worker processes on Giraph (in seconds)*

(a) Input graph: **ws10m2**

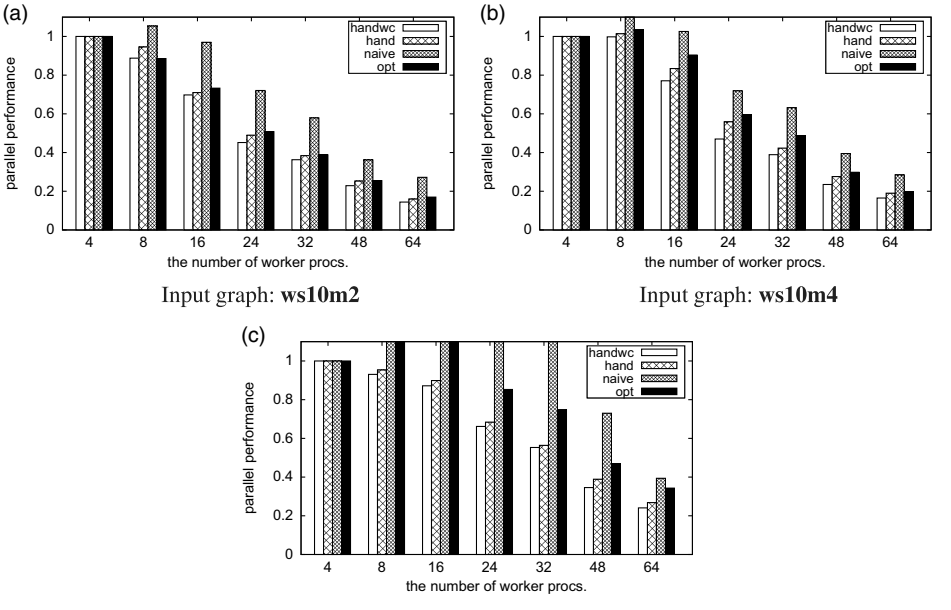
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	116.3	65.5	41.7	42.9	40.0	42.3	50.4	80	766,722,947
hand	133.9	70.7	47.1	45.5	43.6	44.0	52.3	80	766,722,947
naive	442.4	209.8	114.1	102.3	95.5	101.8	101.8	160	3,120,000,000
opt	203.3	114.7	69.4	66.7	65.4	66.5	74.8	129	806,722,943

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	123.7	62.0	40.1	43.9	39.7	43.9	46.8	60	1,298,415,856
hand	147.5	72.7	44.2	44.0	43.6	44.6	48.5	60	1,298,415,856
naive	476.3	215.9	116.1	110.4	94.2	100.5	104.6	120	4,640,000,000
opt	236.6	114.4	65.4	66.1	60.7	66.2	74.6	104	1,378,415,845

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	252.6	135.6	72.4	63.6	57.1	60.9	65.5	84	1,664,378,402
hand	292.5	153.2	81.4	71.3	64.8	62.7	68.3	84	1,664,378,402
naive	1412.2	490.0	224.6	197.7	158.1	161.2	224.3	168	6,560,000,000
opt	600.6	247.9	127.3	117.4	100.3	106.5	109.5	133	1,744,378,397



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 29. Parallel performance of diameter on Giraph.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 7. *Execution times of* scc *with 4–64 worker processes on Giraph (in seconds)*

(a) Input graph: **ws10m2**

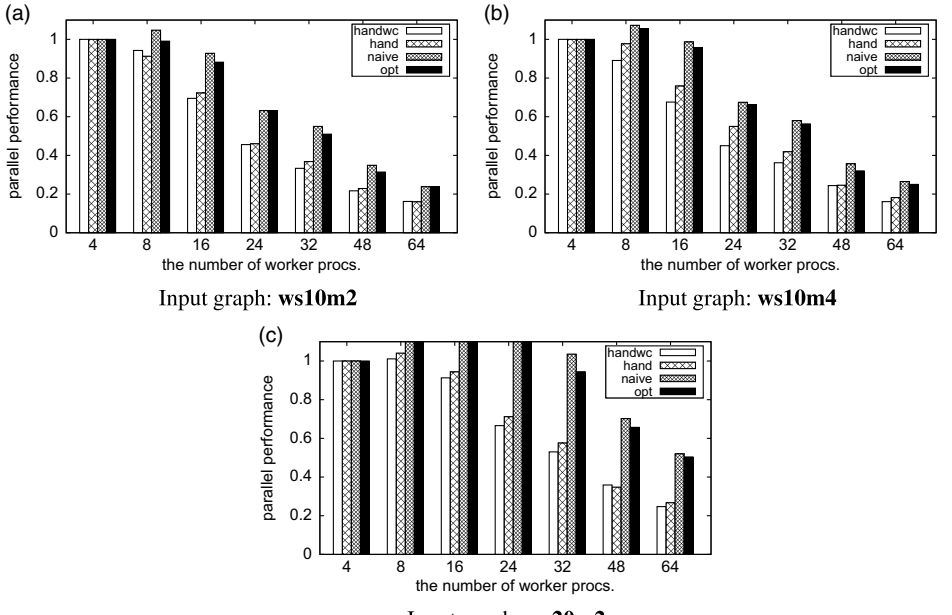
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	95.0	50.4	34.2	34.7	35.7	36.4	36.7	32	829,920,554
hand	102.1	55.9	35.3	37.0	34.7	37.1	40.0	32	829,920,554
naive	394.1	188.0	106.2	104.0	89.6	94.1	103.5	132	2,160,000,000
opt	323.7	163.3	91.7	85.4	79.3	85.8	85.0	130	1,499,841,116

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	89.0	49.9	32.9	33.0	30.7	30.4	34.5	19	1,047,609,058
hand	107.6	55.1	35.4	32.6	32.1	36.4	37.0	19	1,047,609,058
naive	329.3	153.5	83.4	81.4	70.9	76.8	77.7	80	2,240,000,000
opt	290.9	137.7	75.9	73.2	64.6	75.7	72.8	78	1,775,218,138

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	190.5	94.2	52.2	47.7	44.9	44.3	48.1	32	1,658,070,383
hand	214.2	102.9	56.7	50.1	46.5	51.3	50.2	32	1,658,070,383
naive	1242.4	398.7	200.3	183.5	149.9	147.5	149.2	132	4,320,000,000
opt	998.9	356.0	168.7	148.8	132.1	126.7	123.9	130	2,996,140,776



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 30. Parallel performance of scc on Giraph.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 8. *Memory footprint (bytes) of vertex's data fields in Giraph programs, excluding fields defined in the base class of vertices*

Impl.	sssp	reAll	re100	reRanking	diameter	scc
handwc	4	1	1	5	12	16
hand	4	1	1	5	12	16
naive	24	18	18	26	32	67
opt	25	19	19	27	33	58

- integers for the current phase, subphase, and superstep,
- the initial value in the input graph,
- the previous values of the user-defined fields computed in the previous phase, and
- data used to control the phase transition (Section 6.2) caused by the use of ***giter***, which was necessary only in scc.

For all benchmarks except scc, Fregel's optimized program (**opt**) needed another byte compared with **naive** for the Boolean value indicating whether its user-defined fields had been changed in the superstep. For scc, the size of **opt** was less than that of **naive** because some fields were eliminated by the optimizations.

The memory consumptions for edges were the same for all benchmarks.

In summary, for a simple computation like reAll, the Fregel vertices needed much more memory than the ones in the handwritten programs due to the additional fields used for controlling the phase transition. However, this increase in the vertex memory footprint did not matter as it did not substantially increase maximum memory consumption. This is more clearly evident in the results for maximum memory consumption for Pregel+ presented in the next section. (Since Giraph uses Java, it is difficult to observe the maximum memory consumptions for Giraph.)

8.2 Compilation target: Pregel+

This section reports the experimental results for Pregel+.

Tables 9–14 show the measured execution times (the median of five runs) for the programs with 4, 8, 16, 24, 32, 48, and 64 worker processes, as well as the number of supersteps (# SS) and the number of messages (# messages). Note that the number of messages was counted *after* the use of combiners. Thus, the number of messages of **handwc** differed from that of **hand**.

Figures 31 and 32 show the execution time of each program relative to that of **handwc** with 4 and 64 worker processes, respectively. Figures 33–38 show the parallel performance.

In general, the results show the same tendency as those for Giraph. The performance degradation of **naive** from **handwc** was much more than that for Giraph. This was because Pregel+ runs more efficiently than Giraph, so the overhead of Fregel programs was emphasized when running on Pregel+. For the same reason, no superlinear parallel performance was observed.

Table 15 shows the memory footprints of the vertex data fields, excluding those defined in the base class of vertices. The results are similar to those for Giraph (Table 8). The

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

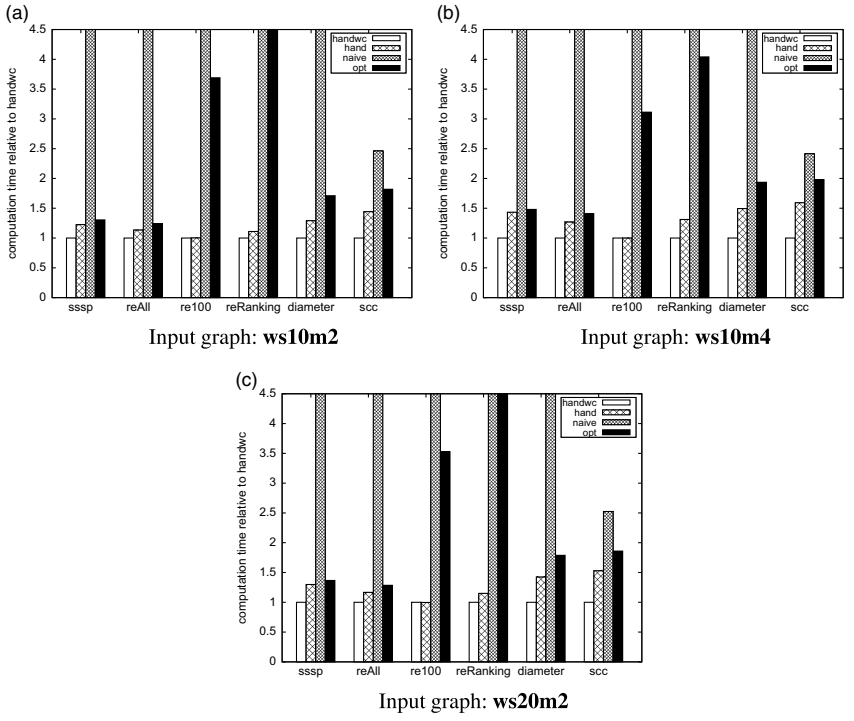


Fig. 31. Computation times compared with that for **handwc** with four worker processes on Pregel+.

reason Pregel+ had a little more vertex data in many cases was that additional fields were needed for the aggregators.

Similar to the results for Giraph, memory consumption for the edges was the same for all benchmarks.

Table 16 shows the maximum memory consumption of a worker process for **ws20m2**. This input graph had the largest ratio of the number of vertices against that of edges among the three input graphs, and hence the effect of the vertex memory footprint on memory consumption was the largest. Each figure shows the median for five runs of the program. For each run, we took the median memory usage of all worker processes except the master process. The results show that even in the worse case (**naive** for re100 with four worker processes), the program compiled from Fregel code consumed only 53.1% more memory than **handwc** although its vertex footprint was much bigger. The increase in the amount of memory consumption decreased as the number of processes increased. These results show that the increase in the vertex memory footprint in Fregel did not cause a serious problem in terms of maximum memory consumption.

In addition, for simple computations like sssp, reAll, and re100, **opt** consumed less memory than **naive** even though **opt** had a bigger footprint than **naive**. This was because **opt** used fewer messages and less memory space for processing messages. These results clearly show that reducing the number of messages is also effective for reducing memory consumption.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

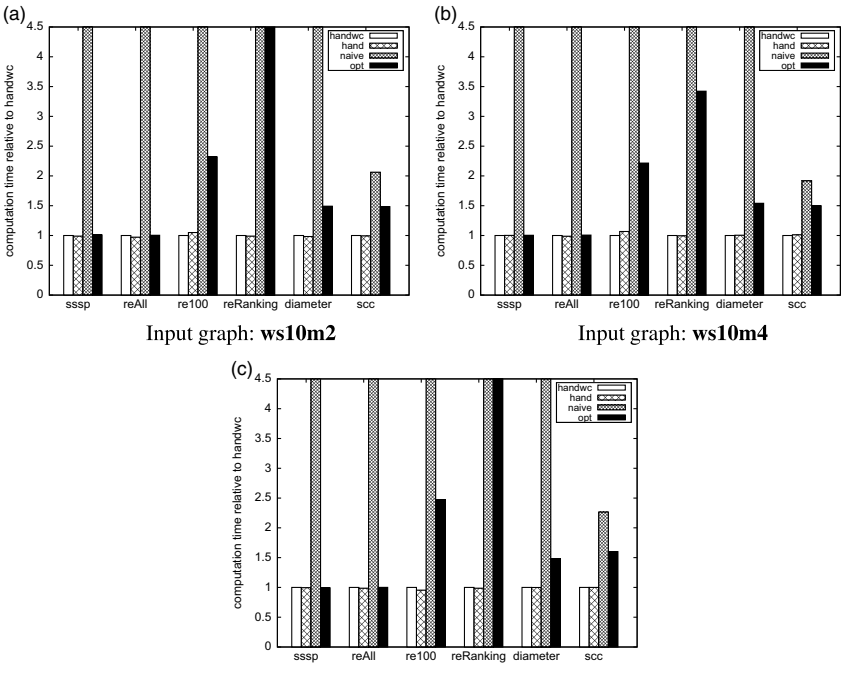


Fig. 32. Computation times compared with that for **handwc** with 64 worker processes on Pregel+.

9 Related work

Vertex-centric graph processing, pioneered by Google's Pregel (Malewicz *et al.*, 2010), is now a major approach to efficient large-scale graph processing. Many vertex-centric graph processing frameworks have been proposed, including Giraph,[3] GraphLab (Low *et al.*, 2012), GPS (Salihoglu & Widom, 2013), GraphX (Gonzalez *et al.*, 2014), and Pregel+ (Yan *et al.*, 2014b). Many other frameworks can be found from extensible surveys on large graph processing (Khan & Elnikety, 2014; McCune *et al.*, 2015; Yan *et al.*, 2016; Khan, 2017; Yan *et al.*, 2017; Kalavri *et al.*, 2018; Liu & Khan, 2018) and experimental evaluations of these frameworks (Han *et al.*, 2014; Lu *et al.*, 2014; Guo *et al.*, 2014; Satish *et al.*, 2014; Capota *et al.*, 2015; Gao *et al.*, 2015; Verma *et al.*, 2017).

Among vertex-centric graph processing frameworks, Fregel has two key features. First, its deterministic functional style makes programs concise, compositional, and easy to develop and test. Second, its optimizations eliminate major inefficiencies in naively written vertex-centric graph processing programs.

Most vertex-centric graph processing frameworks are based on sequential programming. In Section 2, we compared an existing approach with Fregel. Because of Fregel's high-level declarative nature, programmers can write graph processing programs concisely without careful control over communications, execution states, and terminations. Second-order graph functions, ***fregel*** in particular, provide clear separation between initialization, the computation applied in each step, and the termination condition. For supporting the

[3] http://giraph.apache.org.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 9. *Execution times of* sssp *with 4–64 worker processes on Pregel+ (in seconds)*

(a) Input graph: **ws10m2**

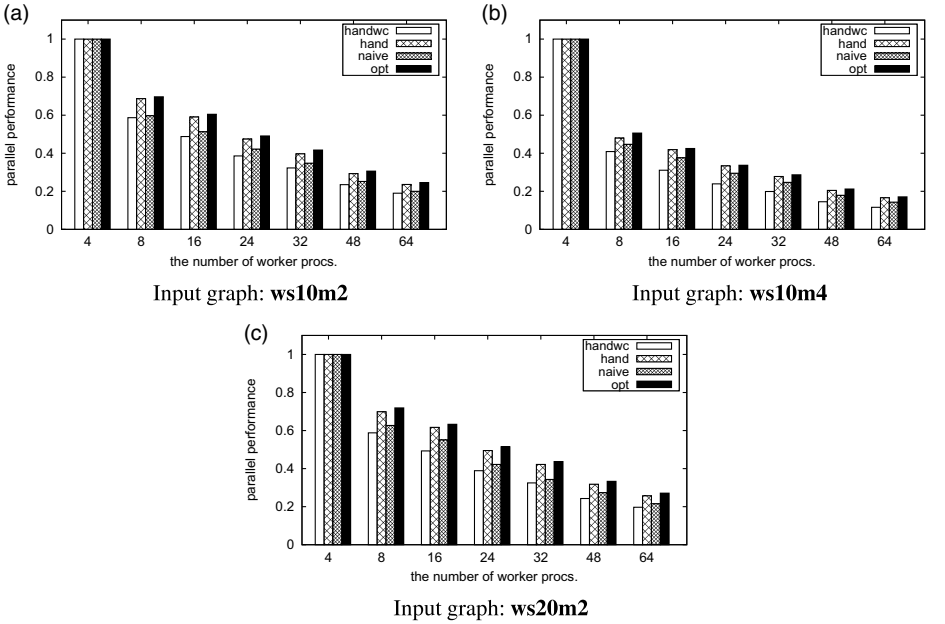
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	12.29	10.47	6.30	5.31	4.75	4.35	4.03	49	173,763,321
hand	15.05	10.96	6.37	5.28	4.73	4.28	3.99	49	174,593,726
naive	118.69	99.39	57.82	46.83	42.64	39.27	37.13	98	1,920,000,000
opt	16.03	11.50	6.63	5.44	4.80	4.37	4.07	49	174,593,726

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	22.67	27.70	18.21	15.80	14.26	13.04	12.23	44	515,487,212
hand	32.51	33.88	19.42	16.23	14.60	13.22	12.26	44	521,880,562
naive	173.49	194.15	115.38	98.00	87.67	80.72	76.01	88	3,440,000,000
opt	33.56	33.15	19.76	16.58	14.63	13.21	12.28	44	521,880,562

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	24.74	21.03	12.56	10.59	9.53	8.47	7.85	49	340,513,631
hand	32.14	22.99	13.02	10.82	9.52	8.42	7.81	49	342,144,658
naive	260.89	208.08	118.45	103.07	94.98	79.77	75.72	98	3,840,000,000
opt	33.79	23.50	13.34	10.93	9.66	8.47	7.79	49	342,144,658



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 33. Parallel performance of sssp on Pregel+.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 10. *Execution times of reAll with 4–64 worker processes on Pregel+ (in seconds)*

(a) Input graph: **ws10m2**

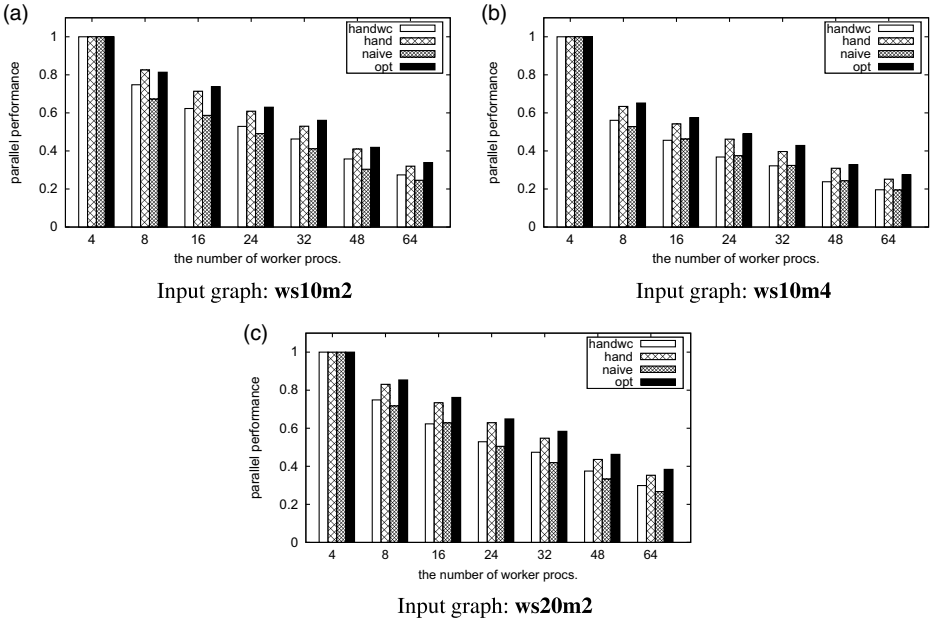
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	4.26	2.85	1.71	1.34	1.15	0.99	0.97	28	39,878,618
hand	4.83	2.92	1.69	1.32	1.14	0.98	0.94	28	40,000,000
naive	63.25	46.98	26.95	21.47	19.22	17.34	16.06	56	1,080,000,000
opt	5.29	3.25	1.79	1.40	1.18	1.05	0.98	28	40,000,000

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	4.70	4.18	2.57	2.13	1.82	1.64	1.50	15	79,351,311
hand	5.96	4.70	2.75	2.15	1.88	1.61	1.48	15	80,000,000
naive	55.47	52.58	29.96	24.68	21.42	19.03	17.77	30	1,120,000,000
opt	6.62	5.08	2.88	2.25	1.93	1.68	1.51	15	80,000,000

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	8.56	5.71	3.43	2.70	2.26	1.90	1.79	28	79,757,890
hand	9.98	6.01	3.40	2.64	2.28	1.91	1.77	28	80,000,000
naive	140.52	98.02	55.92	46.37	41.89	35.15	32.93	56	2,160,000,000
opt	10.99	6.44	3.61	2.82	2.36	1.98	1.79	28	80,000,000



Input graph: **ws10m2** Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 34. Parallel performance of reAll on Pregel+.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 11. *Execution times of* re100 *with 4–64 worker processes on Pregel+ (in seconds)*

(a) Input graph: **ws10m2**

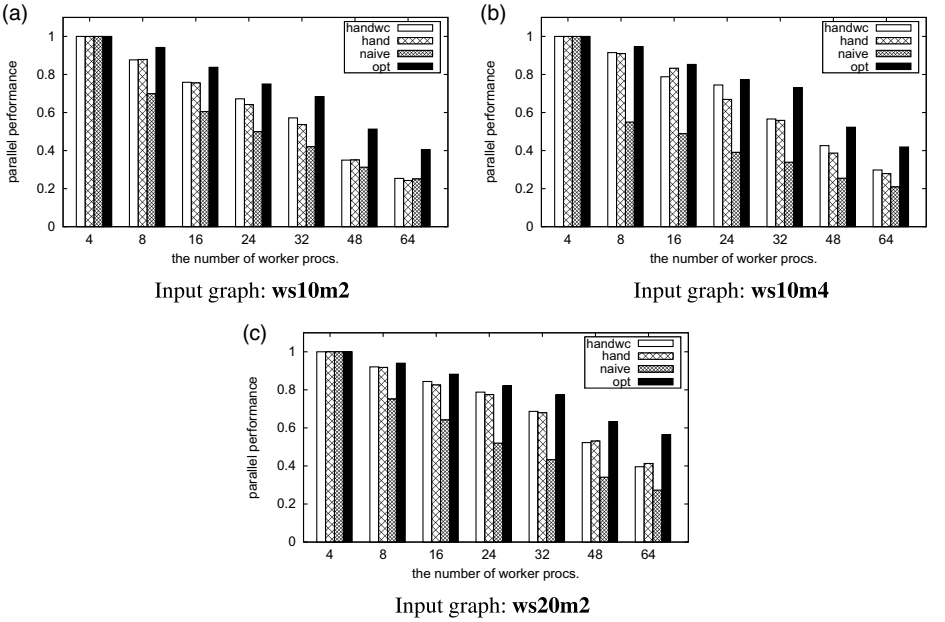
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	0.41	0.23	0.13	0.10	0.09	0.10	0.10	7	575
hand	0.41	0.23	0.14	0.11	0.09	0.10	0.10	7	577
naive	12.44	8.90	5.14	4.16	3.71	3.31	3.10	12	200,000,000
opt	1.50	0.80	0.45	0.33	0.28	0.24	0.23	12	272

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	0.35	0.19	0.11	0.08	0.08	0.07	0.07	5	2,355
hand	0.35	0.19	0.11	0.09	0.08	0.08	0.08	5	2,366
naive	12.79	11.63	6.54	5.44	4.72	4.18	3.83	8	240,000,000
opt	1.10	0.58	0.32	0.24	0.19	0.17	0.16	8	516

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	0.71	0.38	0.21	0.15	0.13	0.11	0.11	6	461
hand	0.71	0.38	0.21	0.15	0.13	0.11	0.11	6	462
naive	22.20	14.76	8.64	7.12	6.41	5.43	5.08	10	320,000,000
opt	2.50	1.33	0.71	0.51	0.40	0.33	0.28	10	205



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 35. Parallel performance of re100 on Pregel+.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 12. *Execution times of reRanking with 4–64 worker processes on Pregel+ (in seconds)*

(a) Input graph: **ws10m2**

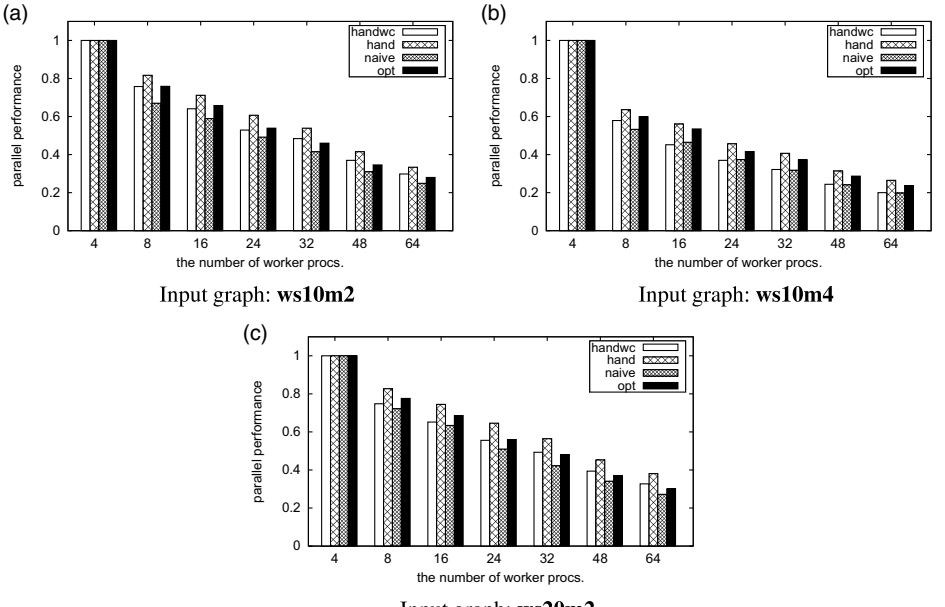
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	5.06	3.34	1.97	1.59	1.31	1.14	1.06	29	39,878,618
hand	5.61	3.43	1.97	1.54	1.30	1.13	1.05	29	40,000,000
naive	63.85	47.67	27.10	21.61	19.23	17.14	16.04	56	1,080,000,000
opt	27.29	17.97	10.37	8.44	7.41	6.58	6.09	56	357,244,816

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	5.01	4.33	2.77	2.26	1.94	1.72	1.56	16	79,351,311
hand	6.58	5.17	2.93	2.40	2.02	1.75	1.55	16	80,000,000
naive	55.92	52.47	30.05	24.93	22.01	19.30	17.67	30	1,120,000,000
opt	20.26	16.92	9.46	8.11	6.79	5.88	5.35	30	310,250,240

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	10.14	6.78	3.89	3.04	2.57	2.14	1.94	29	79,757,890
hand	11.66	7.05	3.91	3.01	2.58	2.14	1.91	29	80,000,000
naive	142.35	98.57	56.15	46.56	42.13	34.77	32.77	56	2,160,000,000
opt	58.71	37.84	21.39	17.46	15.24	13.20	12.19	56	719,102,699



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 36. Parallel performance of reRanking on Pregel+.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 13. *Execution times of* diameter *with 4–64 worker processes on Pregel+ (in seconds)*

(a) Input graph: **ws10m2**

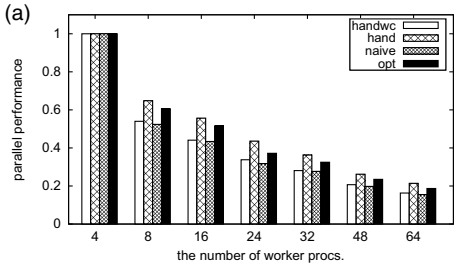
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	41.69	38.64	23.64	20.55	18.55	16.81	16.02	80	761,733,549
hand	53.82	41.55	24.15	20.57	18.56	17.09	15.74	80	766,722,947
naive	211.09	201.40	121.71	110.53	95.09	88.71	85.04	160	3,120,000,000
opt	71.35	58.87	34.57	31.96	27.45	25.26	23.90	129	806,722,943

(b) Input graph: **ws10m4**

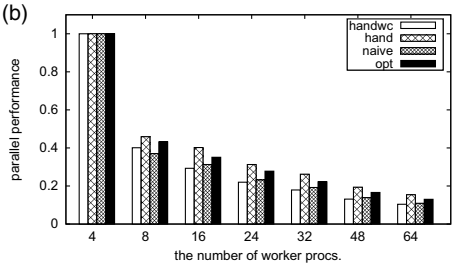
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	49.45	61.64	42.25	37.51	34.53	31.39	29.83	60	1,278,987,924
hand	73.89	80.47	45.90	39.34	35.27	31.70	29.99	60	1,298,415,856
naive	260.42	351.53	208.24	186.69	169.78	156.24	149.53	120	4,640,000,000
opt	95.74	110.68	68.23	57.45	53.87	47.98	45.98	104	1,378,415,845

(c) Input graph: **ws20m2**

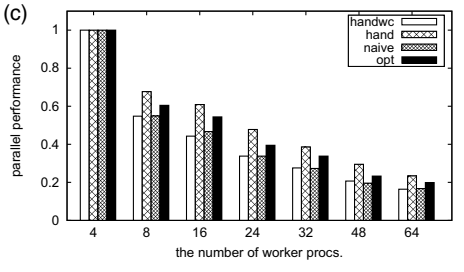
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	91.93	83.86	51.88	45.39	41.67	36.98	34.95	84	1,653,750,466
hand	130.90	96.63	53.76	45.67	42.38	37.02	34.90	84	1,664,378,402
naive	482.90	439.63	258.55	237.90	220.94	205.36	180.70	168	6,560,000,000
opt	164.36	135.87	75.51	69.31	60.76	58.67	51.72	133	1,744,378,397



Input graph: **ws10m2**



Input graph: **ws10m4**



Input graph: **ws20m2**

Fig. 37. arallel performance of diameter on Pregel+.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 14. *Execution times of* scc *with 4–64 worker processes on Pregel+ (in seconds)*

(a) Input graph: **ws10m2**

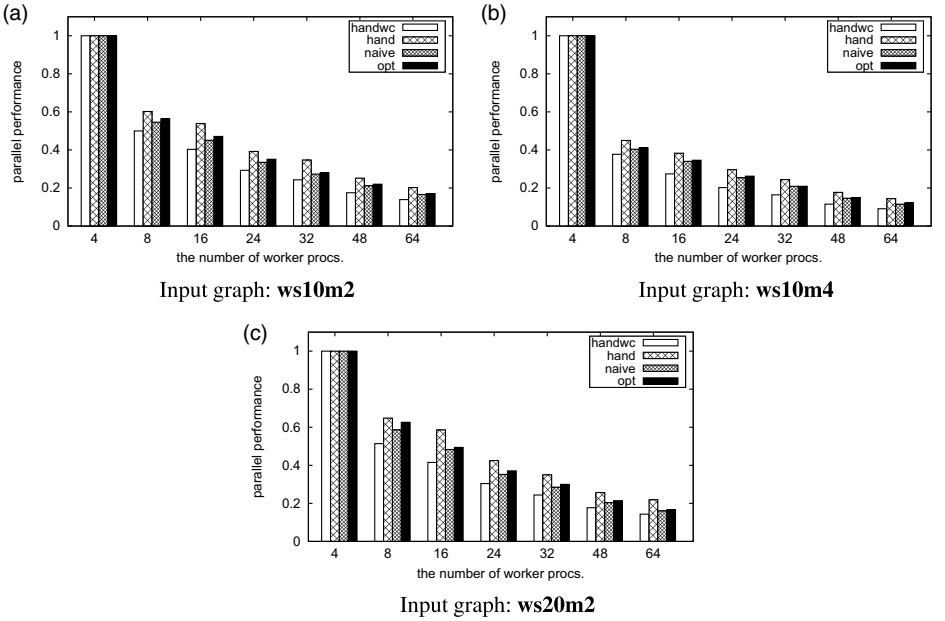
Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	65.08	65.03	40.36	37.06	33.44	30.96	29.22	58	1,485,886,562
hand	93.92	78.00	43.58	39.98	33.84	31.04	29.02	58	1,499,841,116
naive	160.30	146.86	88.80	79.77	73.37	63.06	60.28	132	2,160,000,000
opt	118.35	105.16	62.87	56.22	52.97	44.81	43.37	130	1,499,841,116

(b) Input graph: **ws10m4**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	57.00	75.64	52.03	47.14	43.50	41.35	38.98	32	1,739,688,640
hand	90.72	100.90	59.24	51.13	46.56	42.77	39.40	32	1,775,218,138
naive	137.74	170.46	101.14	90.51	82.55	78.49	74.80	80	2,240,000,000
opt	112.92	137.32	81.63	71.74	67.65	62.80	58.44	78	1,775,218,138

(c) Input graph: **ws20m2**

Impl.	4	8	16	24	32	48	64	# SS	# messages
handwc	136.37	132.70	82.14	74.88	69.86	64.16	59.40	58	2,968,308,280
hand	208.34	160.68	88.94	81.62	74.49	67.51	59.40	58	2,996,140,776
naive	344.21	293.54	178.03	163.21	150.90	140.61	134.56	132	4,320,000,000
opt	253.67	202.65	128.27	113.96	105.79	98.82	95.17	130	2,996,140,776



Input graph: **ws10m2**

Input graph: **ws10m4**

Input graph: **ws20m2**

Fig. 38. Parallel performance of scc on Pregel+.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Table 15. *Memory footprint (bytes) of vertex data fields for Pregel+ programs, excluding fields defined in base class of vertices*

Impl.	sssp	reAll	re100	reRanking	diameter	scc
handwc	4	1	1	6	13	17
hand	4	1	1	6	13	17
naive	26	20	23	32	35	71
opt	27	21	24	33	36	62

Table 16. *Maximum memory consumption (MB) of a worker process for Pregel+ programs for* **ws20m2**

(a) Computation with four worker processes

Impl.	sssp	reAll	re100	reRanking	diameter	scc
handwc	1260.7	1245.2	1055.4	1324.0	1456.0	1810.7
hand	1353.6	1272.3	1055.1	1349.1	1633.7	2040.1
naive	1726.5	1616.9	1616.2	1616.8	1983.7	2592.8
opt	1508.7	1428.9	1211.8	1617.4	2084.2	2514.7

(b) Computation with 16 worker processes

Impl.	sssp	reAll	re100	reRanking	diameter	scc
handwc	423.1	407.7	361.1	440.9	451.5	535.9
hand	440.5	407.6	361.0	440.4	496.7	583.7
naive	497.2	482.1	481.9	494.2	543.4	696.7
opt	479.0	446.4	400.1	508.5	579.9	692.7

(c) Computation with 64 worker processes

Impl.	sssp	reAll	re100	reRanking	diameter	scc
handwc	180.0	166.0	152.4	183.3	197.3	210.7
hand	177.8	166.0	152.3	183.6	198.7	210.7
naive	194.9	186.6	186.0	199.0	219.4	246.1
opt	187.4	175.8	162.1	200.8	224.8	245.3

expressive power of Fregel as a functional vertex-centric framework, a high-level DSL (Emoto & Sadahira, 2020) that is able to manipulate vertex subsets has been developed: a program written in this DSL is compiled into a Fregel program on the basis of second-order graph functions.

Several graph processing frameworks provide declarative programming interfaces, including Elixir (Prountzos *et al.*, 2012, 2015), Distributed SociaLite (Seo *et al.*, 2013), and CLM (Coordinated Linear Meld) (Cruz *et al.*, 2016). Elixir automatically derives an efficient distributed graph processing code from the declarative specification of the output graph. Distributed SociaLite is a graph processing language similar to Datalog. It accelerates single-source-shortest-path-like computation by processing vertices in accordance with a special priority if a certain kind of monotonicity property is detected. CLM is based

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

on linear logic and provides control over scheduling and data layout using coordination. Interestingly, all of these frameworks are concurrent; that is, by default, the underlying graph is processed nondeterministically. In contrast, Fregel is based on BSP and therefore *deterministic*.

We believe that Fregel's deterministic nature makes it easier to develop and test non-trivial graph processing programs. Moreover, Fregel's optimizer can automatically detect possibilities of nondeterministic, that is, asynchronous, evaluation. Another difference is that existing frameworks require programmers to provide clues for optimization. For instance, with Elixir, programmers should specify the conditions for sending messages and the priorities for processing vertices. With Distributed SociaLite, prioritized execution is applied only if programmers use certain operators. CLM can generate efficient code only when programmers provide appropriate annotations called "coordination facts."

Several recently proposed frameworks take dynamic optimization approaches. SLFE (Song *et al.*, 2018) reduces redundancies in vertex computation by utilizing a graph's topological knowledge on the fly. SympleGraph (Zhuo *et al.*, 2020) eliminates unnecessary computations and communications by propagating loop-carried dependency dynamically. Unlike these frameworks, Fregel takes a static optimization approach, but the optimization methods used for Fregel are not new. Vertex inactivation is a part of the core functionality of Pregel (Malewicz *et al.*, 2010). The communication reduction technique for the single-source shortest path problem has been reported (Malewicz *et al.*, 2010). Many vertex-centric graph processing frameworks support asynchronous execution (Gonzalez *et al.*, 2012; Low *et al.*, 2012; Wang *et al.*, 2013; Han & Daudjee, 2015); moreover, some combine asynchronous and synchronous execution to further improve efficiency (Xie *et al.*, 2015; Liu *et al.*, 2016). Several frameworks (Prountzos *et al.*, 2012, 2015; Salihoglu & Widom, 2014; Cruz *et al.*, 2016; Liu *et al.*, 2016) support prioritized execution as well. The effectiveness of these optimizations has been intensively studied. Our contribution is their automation using constraint solvers.

Some frameworks are based on variants of vertex-centric graph processing, including subgraph-centric ones (Tian *et al.*, 2013; Simmhan *et al.*, 2014; Quamar *et al.*, 2014, 2016; Quamar & Deshpande, 2016), block-centric ones (Yan *et al.*, 2014a), edge-centric ones, (Zhou *et al.*, 2017), and path-centric ones (Yuan *et al.*, 2016). The motivation behind these variants is that the vertex-centric approach is sometimes too fine-grained and thus potentially misses opportunities for optimization based on localities and graph structures. For example, the subgraph-centric approach processes subgraphs, rather than vertices, so a specialized algorithm can be used for determining the order and necessity of processing vertices and edges in the subgraph. To enable potential tuning of the substructures, programming with these variants tends to be more difficult than that with the vertex-centric approach because programmers need to carefully control the processing over substructures and the communications between substructures. Though Fregel is based on a vertex-centric approach, the combination of asynchronous and prioritized execution in Fregel may bring efficiency improvement similar to that obtained by using these variants. For instance, in a vertex-centric program for the single-source shortest path problem, these optimizations lead to a code that processes each subgraph by using the Dijkstra algorithm. It is not known whether our optimizations are sufficient for efficient graph processing for practical cases. Investigating this is left for future work.

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

Many researchers have investigated recursive approaches to programming graph algorithms in functional languages (Fegaras & Sheard, 1996; Erwig, 1997, 2001; Hamana, 2010; Oliveira & Cook, 2012; Hidaka *et al.*, 2013; Bahr & Axelsson, 2017). They regarded cyclic and shared structures as (possibly infinite) trees and provided a way of structural-recursive processing of the tree representations. Unfortunately, all of them are for sequential computation. Except for its focus on parallel computation, the Fregel language follows a direction similar to that of previous studies, with special attention to memorization of calculated values and termination control by observing a possibly infinite sequence of graphs.

10 Conclusion

We have presented a functional formalization of synchronous vertex-centric graph processing and proposed Fregel, a domain-specific language based on the proposed formalized model. The Fregel compiler translates a Fregel program into one that can be run in the Giraph or Pregel+ framework for parallel vertex-centric graph processing. The compiler has two key features. One is automatic division of an LSS at every communication point into Pregel supersteps to generate a normalized program, which is then transformed into a program for the target framework via framework-dependent IR. The other is automatic removal of inefficiencies, for example, unnecessary communication between vertices, by the use of a constraint solver. These features enable the Fregel programmer to develop a vertex-centric program intuitively and concisely without being concerned with how to properly control and terminate the computation on each vertex.

Our main focus has been to investigate the effects of a declarative approach to vertex-centric graph processing, for example, how the approach relieves the programmer of the complicated programming tasks when using imperative languages, for which various controls over computation have to be explicitly described. Thus, although Fregel currently has limited capabilities regarding the use of list data structures and recursive definitions, this is not a drawback because the purpose of this research is *not* to develop a compiler for a *full-set* functional language. Nevertheless, future work includes overcoming these limitations to make Fregel more practical.

Future work also includes implementing and evaluating two potential optimizations described in Sections 7.4 and 7.5. This might require developing a framework that supports both synchronous and asynchronous execution.

The latest version of the Fregel system is available via the web at https://fregel.ipl-lab.org/.

Acknowledgments

This work was partly supported by JSPS KAKENHI Grant Numbers JP26280020, JP15K15965, and JP19K11901.

Conflicts of Interest

None

https://doi.org/10.1017/S0956796821000277 Published online by Cambridge University Press

References

Bae, S. & Howe, B. (2015) Gossipmap: A distributed community detection algorithm for billion-edge directed graphs. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15–20, 2015, pp. 27:1–27:12.

Bahr, P. & Axelsson, E. (2017) Generalising tree traversals and tree transformations to dags: Exploiting sharing without the pain. *Sci. Comput. Program.* **137**, 63–97.

Bu, Y., Howe, B., Balazinska, M. & Ernst, M. D. (2012) The Haloop approach to large-scale iterative data analysis. *VLDB J.* **21**(2), 169–190.

Capota, M., Hegeman, T., Iosup, A., Prat-Pérez, A., Erling, O. & Boncz, P. A. (2015) Graphalytics: A big data benchmark for graph-processing platforms. In Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31–June 4, 2015, pp. 7:1–7:6.

Caviness, B. F. & Johnson, J. R. (eds). (1998) *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Springer Vienna.

Cruz, F., Rocha, R. & Goldstein, S. C. (2016) Declarative coordination of graph-based parallel programs. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12–16, 2016, pp. 4:1–4:12.

Dathathri, R., Gill, G., Hoang, L., Dang, H., Brooks, A., Dryden, N., Snir, M. & Pingali, K. (2018) Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018, Foster, J. S. & Grossman, D. (eds). ACM, pp. 752–768.

de Moura, L. M. & Bjørner, N. (2011) Satisfiability modulo theories: Introduction and applications. *Commun. ACM* **54**(9), 69–77.

Emoto, K. & Sadahira, F. (2020) A DSL for graph parallel programming with vertex subsets. *J. Supercomput.* **76**(7), 4998–5015.

Emoto, K., Matsuzaki, K., Hu, Z., Morihata, A. & Iwasaki, H. (2016) Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016, pp. 200–213.

Erwig, M. (1997) Functional programming with graphs. In Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, ICFP 1997, Peyton Jones, S. L., Tofte, M. & Berman, A. M. (eds). Amsterdam, The Netherlands, June 9–11, 1997. ACM, pp. 52–65.

Erwig, M. (2001) Inductive graphs and functional graph algorithms. *J. Funct. Program.* **11**(5), 467–492.

Fegaras, L. & Sheard, T. (1996) Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 1996. ACM, pp. 284–294.

Gao, Y., Zhou, W., Han, J., Meng, D., Zhang, Z. & Xu, Z. (2015) An evaluation and analysis of graph processing frameworks on five key issues. In Proceedings of the 12th ACM International Conference on Computing Frontiers, CF 2015, Ischia, Italy, May 18–21, 2015, pp. 11:1–11:8.

Gonzalez, J. E., Low, Y., Gu, H., Bickson, D. & Guestrin, C. (2012) Powergraph: Distributed graph-parallel computation on natural graphs. In 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8–10, 2012, pp. 17–30.

Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J. & Stoica, I. (2014) Graphx: Graph processing in a distributed dataflow framework. In 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, Broomfield, CO, USA, October 6–8, 2014, pp. 599–613.

Guo, Y., Biczak, M., Varbanescu, A. L., Iosup, A., Martella, C. & Willke, T. L. (2014) How well do graph-processing platforms perform? An empirical performance evaluation and analysis. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS 2014, Phoenix, AZ, USA, May 19–23, 2014, pp. 395–404.

Hamana, M. (2010) Initial algebra semantics for cyclic sharing tree structures. *Log. Methods Comput. Sci.* **6**(3), 1–23.

Han, M. & Daudjee, K. (2015) Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB* **8**(9), 950–961.

Han, M., Daudjee, K., Ammar, K., Özsu, M. T., Wang, X. & Jin, T. (2014) An experimental comparison of pregel-like graph processing systems. *PVLDB* **7**(12), 1047–1058.

Hidaka, S., Asada, K., Hu, Z., Kato, H. & Nakano, K. (2013) Structural recursion for querying ordered graphs. In ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, MA, USA - September 25–27, 2013, pp. 305–318.

Hong, S., Chafi, H., Sedlar, E. & Olukotun, K. (2012) Green-marl: A DSL for easy and efficient graph analysis. In Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3–7, 2012, pp. 349–362.

Kalavri, V., Vlassov, V. & Haridi, S. (2018) High-level programming abstractions for distributed graph processing. *IEEE Trans. Knowl. Data Eng.* **30**(2), 305–324.

Kang, U., Tong, H., Sun, J., Lin, C. & Faloutsos, C. (2012) GBASE: An efficient analysis platform for large graphs. *VLDB J.* **21**(5), 637–650.

Kang, U., Tsourakakis, C. E. & Faloutsos, C. (2011) PEGASUS: Mining peta-scale graphs. *Knowl. Inf. Syst.* **27**(2), 303–325.

Kato, N. & Iwasaki, H. (2019) Eliminating unnecessary communications in the vertex-centric graph processing by the fregel compiler. *Comput. Software* **36**(2), 2_28–2_46. In Japanese.

Khan, A. (2017) Vertex-centric graph processing: Good, bad, and the ugly. In Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21–24, 2017, pp. 438–441.

Khan, A. & Elnikety, S. (2014) Systems for big-graphs. *PVLDB* **7**(13), 1709–1710.

Liu, S. & Khan, A. (2018) An empirical analysis on expressibility of vertex centric graph processing paradigm. In IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10–13, 2018, pp. 242–251.

Liu, Y., Zhou, C., Gao, J. & Fan, Z. (2016) Giraphasync: Supporting online and offline graph processing via adaptive asynchronous message processing. In Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24–28, 2016, pp. 479–488.

Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C. & Hellerstein, J. M. (2012) Distributed graphlab: A framework for machine learning in the cloud. *PVLDB* **5**(8), 716–727.

Lu, Y., Cheng, J., Yan, D. & Wu, H. (2014) Large-scale distributed graph computing systems: An experimental evaluation. *PVLDB* **8**(3), 281–292.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N. & Czajkowski, G. (2010) Pregel: A system for large-scale graph processing. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6–10, 2010, pp. 135–146.

McCune, R. R., Weninger, T. & Madey, G. (2015) Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* **48**(2), 25:1–25:39.

Morihata, A., Emoto, K., Matsuzaki, K., Hu, Z. & Iwasaki, H. (2018) Optimizing declarative parallel distributed graph processing by using constraint solvers. In Functional and Logic Programming – 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9–11, 2018, pp. 166–181.

Nguyen, D., Lenharth, A. & Pingali, K. (2013) A lightweight infrastructure for graph analytics. In ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP 2013, Farmington, PA, USA, November 3–6, 2013, pp. 456–471.

Oliveira, B. C. d. S. & Cook, W. R. (2012) Functional programming with structured graphs. In ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, September 9–15, 2012, Thiemann, P. & Findler, R. B. (eds). ACM, pp. 77–88.

Prountzos, D., Manevich, R. & Pingali, K. (2012) Elixir: A system for synthesizing concurrent graph programs. In Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented

Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21–25, 2012, pp. 375–394.

Prountzos, D., Manevich, R. & Pingali, K. (2015) Synthesizing parallel graph programs via automated planning. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, Portland, OR, USA, June 15–17, 2015, pp. 533–544.

Quamar, A. & Deshpande, A. (2016) Nscalespark: Subgraph-centric graph analytics on apache spark. In Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016, pp. 5:1–5:8.

Quamar, A., Deshpande, A. & Lin, J. J. (2014) Nscale: Neighborhood-centric analytics on large graphs. *PVLDB* **7**(13), 1673–1676.

Quamar, A., Deshpande, A. & Lin, J. J. (2016) Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *VLDB J.* **25**(2), 125–150.

Salihoglu, S. & Widom, J. (2013) GPS: A graph processing system. In Conference on Scientific and Statistical Database Management, SSDBM 2013, Baltimore, MD, USA, July 29–31, 2013, pp. 22:1–22:12.

Salihoglu, S. & Widom, J. (2014) Optimizing graph algorithms on pregel-like systems. *PVLDB* **7**(7), 577–588.

Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z. & Dubey, P. (2014) Navigating the maze of graph analytics frameworks using massive graph datasets. In International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22–27, 2014, pp. 979–990.

Sengupta, D., Song, S. L., Agarwal, K. & Schwan, K. (2015) Graphreduce: Processing large-scale graphs on accelerator-based systems. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15–20, 2015, pp. 28:1–28:12.

Seo, J., Park, J., Shin, J. & Lam, M. S. (2013) Distributed socialite: A datalog-based language for large-scale graph analysis. *PVLDB* **6**(14), 1906–1917.

Simmhan, Y., Kumbhare, A. G., Wickramaarachchi, C., Nagarkar, S., Ravi, S., Raghavendra, C. S. & Prasanna, V. K. (2014) Goffish: A sub-graph centric framework for large-scale graph analytics. In Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25–29, 2014. Proceedings, pp. 451–462.

Song, S., Liu, X., Wu, Q., Gerstlauer, A., Li, T. & John, L. K. (2018) Start late or finish early: A distributed graph processing system with redundancy reduction. *Proc. VLDB Endow.* **12**(2), 154–168.

Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S. & McPherson, J. (2013) From "think like a vertex" to "think like a graph". *PVLDB* **7**(3), 193–204.

Valiant, L. G. (1990) A bridging model for parallel computation. *Commun. ACM* **33**(8), 103–111.

Verma, S., Leslie, L. M., Shin, Y. & Gupta, I. (2017) An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB* **10**(5), 493–504.

Wang, G., Xie, W., Demers, A. J. & Gehrke, J. (2013) Asynchronous large-scale graph processing made easy. In Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6–9, 2013, Online Proceedings.

Watts, D. J. & Strogatz, S. H. (1998) Collective dynamics of 'small-world' networks. *Nature* **393**(6684), 440–442.

Xie, C., Chen, R., Guan, H., Zang, B. & Chen, H. (2015) SYNC or ASYNC: Time to fuse for distributed graph-parallel computation. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7–11, 2015, pp. 194–204.

Yan, D., Bu, Y., Tian, Y., Deshpande, A. & Cheng, J. (2016) Big graph analytics systems. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, San Francisco, CA, USA, June 26–July 01, 2016, pp. 2241–2243.

Yan, D., Bu, Y., Tian, Y. & Deshpande, A. (2017) Big graph analytics platforms. *Found. Trends Databases* **7**(1–2), 1–195.

Yan, D., Cheng, J., Lu, Y. & Ng, W. (2014a) Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB* **7**(14), 1981–1992.

Yan, D., Cheng, J., Lu, Y. & Ng, W. (2015) Effective techniques for message reduction and load balancing in distributed graph computation. In Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18–22, 2015, pp. 1307–1317.

Yan, D., Cheng, J., Xing, K., Lu, Y., Ng, W. & Bu, Y. (2014b) Pregel algorithms for graph connectivity problems with performance guarantees. *PVLDB* **7**(14), 1821–1832.

Yuan, P., Xie, C., Liu, L. & Jin, H. (2016) Pathgraph: A path centric graph processing system. *IEEE Trans. Parallel Distrib. Syst.* **27**(10), 2998–3012.

Zhou, J., Xu, C., Chen, X., Wang, C. & Zhou, X. (2017) Mermaid: Integrating vertex-centric with edge-centric for real-world graph processing. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14–17, 2017, pp. 780–783.

Zhuo, Y., Chen, J., Luo, Q., Wang, Y., Yang, H., Qian, D. & Qian, X. (2020) Symplegraph: Distributed graph processing with precise loop-carried dependency guarantee. In Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15–20, 2020, Donaldson, A. F. & Torlak, E. (eds). ACM, pp. 592–607.