Optimizing Declarative Parallel Distributed Graph Processing by Using Constraint Solvers

Akimasa Morihata[1(✉)], Kento Emoto[2], Kiminori Matsuzaki[3], Zhenjiang Hu[4], and Hideya Iwasaki[5]

[1] University of Tokyo, Tokyo, Japan
morihata@graco.c.u-tokyo.ac.jp
[2] Kyushu Institute of Technology, Kitakyushu, Japan
[3] Kochi University of Technology, Kami, Japan
[4] National Institute of Informatics, Tokyo, Japan
[5] The University of Electro-Communications, Chofu, Japan

Abstract. Vertex-centric graph processing is a promising approach for facilitating development of parallel distributed graph processing programs. Each vertex is regarded as a tiny thread and graph processing is described as cooperation among vertices. This approach resolves many issues in parallel distributed processing such as synchronization and load balancing. However, it is still difficult to develop efficient programs requiring careful problem-specific tuning. We present a method for automatically optimizing vertex-centric graph processing programs. The key is the use of constraint solvers to analyze the subtle properties of the programs. We focus on a functional vertex-centric graph processing language, Fregel, and show that quantifier elimination and SMT (Satisfiability Modulo Theories) are useful for optimizing Fregel programs. A preliminary experiment indicated that a modern SMT solver can perform optimization within a realistic time frame and that our method can significantly improve the performance of naively written declarative vertex-centric graph processing programs.

1 Introduction

Nowadays big graphs are ubiquitous. Nearly every interesting data set, such as those for customer purchase histories, social networks, and protein interaction networks, consists of big graphs. Parallel distributed processing is necessary for analyzing big graphs that cannot fit in the memory of a single machine. However, parallel distributed processing is difficult due to such issues as communications, synchronizations, and load balancing.

Vertex-centric graph processing (abbreviated to VcGP) [1] is a promising approach for reducing the difficulties of parallel distributed graph processing. VcGP is based on the "think like a vertex" programming style. It regards each vertex as a tiny thread and describes graph processing as cooperation among vertices, each of which updates its value using information supplied from other vertices.

J. P. Gallagher and M. Sulzmann (Eds.): FLOPS 2018, LNCS 10818, pp. 166–181, 2018.
https://doi.org/10.1007/978-3-319-90686-7_11

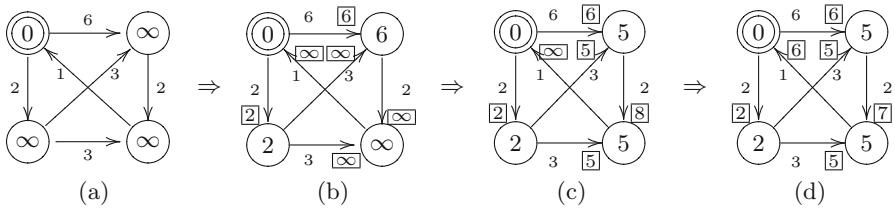

Fig. 1. Vertex-centric SSSP: doubly-circled vertex denotes source, and framed numbers denote messages.

As an example, consider the following algorithm illustrated in Fig. 1 for the single-source shortest path problem (SSSP).

- First, the source vertex is assigned 0, and the other vertices are assigned ∞. The values are the estimated distances from the source vertex (Fig. 1 (a)).
- Then, each vertex sends the estimated distances to its neighbors and updates its value if it receives a shorter distance (Fig. 1(b–c)).
- The previous step is repeated until all the values are no longer changed (Fig 1(d)).

A VcGP framework executes programs in a distributed environment. Vertices (and accordingly edges) are distributed among computational nodes. At every step, every computational node simultaneously updates the values of its vertices in accordance with the specified computation. Computational nodes exchange messages if information of other nodes is necessary. The VcGP approach releases programmers from such typical difficulties as communication, synchronization, and load balancing, and makes it easier to write *runnable* parallel distributed graph processing programs.

Although the VcGP approach is beneficial, it is still difficult to achieve *efficiency*. Natural VcGP programs tend to be slow. For instance, there is room for improvement in the above SSSP algorithm.

- It is unnecessary to process all vertices at every step; it is sufficient to process only those vertices for which values are updated. Similarly, it is unnecessary for all vertices to communicate their neighbors at every step.
- We have adopted *synchronous* execution: each vertex is processed exactly once at each step. One can instead adopt *asynchronous* execution, which processes vertices without synchronization barriers. The relative efficiencies of these two approaches depend on the situation. For SSSP, both executions lead to the same solution, and combining the two approaches may improve performance.
- The algorithm is essentially the Bellman-Ford algorithm, whose work is $O(n^2)$, where n is the size of the graph. Processing near-source vertices prior to distant ones, like Dijkstra's algorithm, may reduce the amount of work because the work of Dijkstra's algorithm is $O(n \log n)$.

These inefficiencies have already been known, and several frameworks have been proposed to remove them [1–7]. For instance, the Pregel framework [1] enables

us to *inactivate* vertices so that they are ignored by the runtime system until they receive a new message. However, it remains the programmer's responsibility to be aware of these inefficiencies and to mitigate them by using available functionalities. This is fairly difficult because these inefficiencies and potential improvements may be hidden by nontrivial problem-specific properties.

We have developed a method for automatically removing such inefficiencies from naive programs written in a functional VcGP language, Fregel [8]. The key is the use of modern constraint solvers for identifying potential optimizations. The declarative nature of Fregel enables optimizations to be directly reduced to constraint solving problems. We focused on four optimizations.

- Eliminating redundant communications (Sect. 3.1).
- Inactivating vertices that do not need to be processed (Sect. 3.2).
- Removing synchronization barriers and thereby enabling asynchronous execution (Sect. 3.3).
- Introducing priorities for processing vertices (Sect. 3.4).

Our approach is not specific to these optimizations for Fregel programs. Nontrivial optimizations on declarative VcGP languages will be implemented similarly if they are formalized as constraint solving problems.

We considered the use of two different constraint solving methods: quantifier elimination (QE) [9] and satisfiability modulo theories (SMT) [10]. 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 less practical because of its high computational cost. We thus use SMT solvers as a practical implementation method that captures typical cases. An experiment using a proof-of-concept implementation demonstrated that a modern SMT solver can perform the optimizations within a realistic time frame and that the optimizations led to significant performance improvement (Sect. 4).

2 Fregel: Functional VcGP Language

2.1 Pregel

Pregel [1] is a pioneering VcGP framework. We review it first because the following systems were strongly influenced.

A Pregel program essentially consists of a function that is invoked by each vertex at each step. It usually updates the values stored in vertices using the following functionalities.

- A vertex can send messages usually to adjacent vertices. The message is available on the destination at the next step.
- A vertex can inactivate itself. The runtime system skips processing of inactive vertices. An inactive vertex is reactivated if it receives a message.
- A vertex can read a summary showing the total sum, average, etc. of all active vertices. This functionality is called an *aggregator*.

```
sssp g = let init v = if v is the source vertex then 0 else ∞
             step v prev = let m = minimum [prev u + e | (e,u) <- is v]
                           in min (prev v) m
         in fregel init step Fix g
```

Fig. 2. Fregel pseudo-program for single-source shortest path problem.

Pregel is based on the bulk synchronous parallel (BSP) model [11]. Computations on the BSP model consists of a series of supersteps. A superstep is a local computation (i.e., invocation of the function by each vertex) followed by a synchronization barrier that guarantees message arrival. Because of the adoption of the BSP model, computation of a Pregel program is deterministic[1]: every vertex can be processed simultaneously without any race conditions. A Pregel program terminates when all vertices become inactive.

2.2 Fregel

Fregel [8] is a functional VcGP language. It is a subset of Haskell and enables VcGP programs to be easily written using graph-processing higher-order functions that conceal side effects including communication and vertex inactivation.

Figure 2 shows a pseudo-program for SSSP. For readability, we focus on the core functionality of Fregel and accordingly use a simplified syntax.

The core of Fregel is the graph processing higher-order function `fregel`. Its first parameter, `init :: Vertex -> Int` in Fig. 2, is applied to each vertex at the initial step. The second one, `step :: Vertex -> (Vertex -> Int) -> Int`, is used at the subsequent steps. The function `step` takes vertex `v` and a table, `prev :: Vertex -> Int`, which stores the results of the previous step. A vertex may access results of neighbor vertices using the table and a special function called a *generator*. The program in Fig. 2 uses `is :: Vertex -> [(Edge,Vertex)]`, which enumerates every neighbor with an edge that leads to the neighbor. Other generators can express other communication patterns including aggregators. Since information read from the neighbors essentially forms a multiset, the information should be summarized not by using a conventional list operation but by using an associative commutative binary operation such as `sum` and `minimum`. The operation should have the unit needed for dealing with an isolated vertex. The third parameter of `fregel`, namely `Fix`, shows that the computation terminates when the result of the current step is the same as that of the previous one. Note that Fregel has no construct for inactivating vertices.

The Fregel compiler translates a Fregel program into a Java program runnable on Giraph[2]. The functionalities of Giraph are nearly the same as those of Pregel. An access to a neighbor's previous value using the `prev` table and a generator is compiled to message exchange if the target vertex is located in a different computational node. Calculating new vertex values using neighbors' previous values naturally corresponds to a superstep in the BSP model.

[1] Except for the order of arrival messages.
[2] Apache Giraph: http://giraph.apache.org/.

$$
\begin{aligned}
\texttt{step v prev = let } c_1 &= \textstyle\bigoplus_1 \texttt{[}f_1\texttt{(e,prev u) | (e,u) <- is v, } p_1\texttt{(prev u)]} \\
&\vdots \\
c_n &= \textstyle\bigoplus_n \texttt{[}f_n\texttt{(e,prev u) | (e,u) <- is v, } p_n\texttt{(prev u)]} \\
\texttt{in } & g\texttt{(prev v,}\, c_1, \ldots, c_m\texttt{)}
\end{aligned}
$$

Fig. 3. Target program for optimization

3 Optimizing Fregel Programs

Here we describe optimizations for programs written using a `fregel` function. We refer to the second parameter, i.e., the one invoked at the non-initial steps, as `step`, and assume that it is written in the form shown in Fig. 3. In the program, f_i, p_i, and $\bigoplus_i$ $(1 \leq i \leq n)$ respectively represent computation over each neighbor's information, the condition showing the necessity of sending the information, and the operator used to summarize the information. g denotes the calculation of the new value of the 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 Sect. 3.5.

We use SSSP as a running example. For SSSP in Fig. 2, $n = 1$, $f_1 = (+)$, $g = (\oplus_1) = \min$, and $p_1(x) = \text{True}$.

3.1 Reducing Communication

Since accesses of a neighbor's information are compiled to message exchange, modifying the condition p_i and thereby avoiding unnecessary accesses reduces the amount of communications. In the following discussion, we focus on reducing communications caused by the k-th access expressed by f_k, p_k, and $\bigoplus_k$. Our strategy is to formalize the situation in which optimization is possible and then to use constraint solvers to implement the optimization.

Formulation. Let $\dot{u}$ be the value of the message-sending vertex, and consider formulating the necessity of sending $\dot{u}$. The following property naturally formulates that sending $\dot{u}$ does not affect the computation on the destination vertex.

$$
\begin{aligned}
&\forall v, e, w_1, \ldots, w_n. \\
&\quad g(v, w_1, \ldots, w_n) = g(v, w_1, \ldots, w_{k-1}, w_k \oplus_k f_k(e, \dot{u}), w_{k+1}, \ldots, w_n) \qquad (1)
\end{aligned}
$$

Though correct, this property is not sufficient in practice, as the following example shows.

Example: SSSP. For SSSP, Property (1) is instantiated as

$$\forall v, e, w.\ \min(v, w) = \min(v, \min(w, e + \dot{u})).$$

This is equivalent to $\dot{u} = \infty$, which means that a vertex can skip message sending if its value is ∞. This result is not satisfactory because a vertex can skip message sending if its value is unchanged from the previous step.

For capturing the case of SSSP, we need a more general 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 the idea. Here, $\dot{u}$ and $\ddot{u}$ respectively denote the current and previous values of the vertex.

$$\begin{aligned}&\forall v, e, w_1, \ldots, w_n, w'_1, \ldots, w'_n.\\ &\quad g(v', w'_1, \ldots, w'_n) = g(v', w'_1, \ldots, w'_k \oplus_k f_k(e, \dot{u}), w'_{k+1} \ldots, w'_n)\\ &\qquad \textbf{where}\ \ v' = g(v, w_1, \ldots, w_k \oplus_k f_k(e, \ddot{u}), w_{k+1} \ldots, w_n)\end{aligned} \tag{2}$$

This is a generalization of Property (1). The necessity of $\dot{u}$ is checked on the basis of the premise that the message-receiving vertex (which has value v') took into account the previous value $\ddot{u}$ of the message-sending vertex.

Example: SSSP (Contd). Property (2) is instantiated as

$$\begin{aligned}\forall v, e, w, w'.\ \min(v', w') = \min(v', \min(w', e + \dot{u}))\\ \textbf{where}\ \ v' = \min(v, \min(w, e + \ddot{u})).\end{aligned}$$

This is equivalent to $\dot{u} \geq \ddot{u}$: a vertex can skip communication when the current value is not smaller than the previous one. Since the current value is always not larger than the previous one, this is equivalent to $\dot{u} = \ddot{u}$.

Implementation Using Constraint Solvers. We could implement the optimization by checking Property (2) dynamically for each vertex. However, since Property (2) consists of quantifiers, its evaluation is likely impossible or very slow. To obtain efficient codes, we need a method for *synthesizing a simple, especially quantifier-free, formula* that is equivalent to (or expressing a sufficient condition of) the property. For this purpose, we use constraint solvers.

QE translates a formula into a quantifier-free equivalent. 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, there are three reasons that using QE solvers may be impractical. 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 $\forall x, a.\ x^2 + ax + a^2 \geq 0$. Recently efficient SMT solvers are intensively developed and used in many applications.

There are two problems in using SMT for checking Property (2). The property contains free variables, $\dot{u}$ and $\ddot{u}$, and moreover, SMT solvers are unable to

```
sssp g =
    let init v = (if v is the source vertex then 0 else ∞, False)
        step v prev = let m = minimum [fst (prev u) + e |
                                       (e,u) <- is v, not (snd (prev u))]
                          v' = min (fst (prev v)) m
                      in (v', v' == fst (prev v))
    in fregel init step Fix g
```

Fig. 4. Fregel SSSP program obtained by communication reduction, where `fst` and `snd` respectively denote extraction of the first and second components from a pair.

synthesize a simple formula. To overcome these problems, we prepare templates of simple formulae, such as $\dot{u} = \ddot{u}$. If the SMT solver guarantees that a template is a sufficient condition of Property (2), we insert the negation of the template into p_k. The effectiveness of this approach relies on the generality of the template. We believe $\dot{u} = \ddot{u}$ captures most practical cases. Other useful templates include natural comparisons on $\dot{u}$ and $\ddot{u}$, such as $\geq$ and/or $\leq$ on numbers and lexicographic orders on tuples.

Example: SSSP. We instruct an SMT solver to check the following formula.

$$\forall \dot{u}, \ddot{u}, v, e, w, w'.\ (\dot{u} = \ddot{u}) \Rightarrow (\min(v', w') = \min(v', \min(w', e + \dot{u})) \\ \textbf{where}\ v' = \min(v, \min(w, e + \ddot{u})))$$

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. Figure 4 shows the optimized program. Each node value is a pair, $\dot{u} = (d, b)$, where d and b respectively denote the estimated distance from the source and whether the value has been changed.

3.2 Inactivating Vertices

Next we discuss inactivating vertices. Inactive vertices do nothing (including any message sending) unless they receive a new message. This optimization should be applied after communication reduction optimization described in Sect. 3.1 because we cannot inactivate vertices that send a message.

A vertex is inactivated if the following condition holds: unless the vertex receives a message, its value does not change and it does not need to send a message. The optimization condition is thus formalized as

$$\left(\bigwedge_{1 \leq i \leq n} \neg p_i(\dot{u}) \right) \wedge (g(\dot{u}, \iota_1, \ldots, \iota_n) = \dot{u}), \tag{3}$$

where each ι_i $(1 \leq i \leq n)$ is the unit of $\oplus_i$ and corresponds to the absence of messages. Since Property 3 contains no quantifier, this optimization can be implemented without the use of a constraint solver.

Example: SSSP. For the SSSP in Fig. 4, Property (3) is instantiated to $b \wedge (\min(d, \infty) = d)$ **where** $\dot{u} = (d, b)$, which is equivalent to b **where** $\dot{u} = (d, b)$. In short, a vertex can be inactivated if its value is the same as the previous one.

3.3 Removing Barriers

Recall that the execution of Fregel is based on the BSP model. Each local computation is followed by a synchronization barrier. Though this makes program behaviors deterministic and more understandable, barriers may make execution slower especially when there are many computational nodes. For most graph algorithms including SSSP, asynchronous barrier-less execution and synchronous barrier-full execution yield the same result; thus, barriers are unnecessary.

The flexibility of asynchronous execution enables further optimizations such as vertex splitting (also known as vertex mirroring) [12,13]. Practical graphs often contain vertices that have too many edges, and such vertices form a bottleneck in VcGP. 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 phase for every step to merge the messages sent to the split vertices. With asynchronous execution, the additional phase 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 optimization is related to subgraph-centric (or neighborhood-centric) approaches [4,5] in which not vertices but subgraphs are the target of parallel processing.

We have developed a method that automatically guarantees equivalence between synchronous and asynchronous execution. We first present the following lemma. Its proof is obvious and thus omitted.

Lemma 1. *For functions h, h' and a binary relation $\preceq$, three conditions are assumed:*

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

Then, $h^*(x) = h^*(h'(x))$ holds for any x, where h^* is defined by $h^*(x) = z \iff (h(z) = z) \wedge (z = h(h(\cdots(h(x))\cdots)))$. □

We apply Lemma 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 processing. Lemma 1 guarantees that a partial processing does not change the result; then, by induction, asynchronous execution does not as well.

Lemma 1 requires an appropriate binary relation, $\preceq$. From the ordering between h and h', a natural candidate is the 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 issue for using Lemma 1 is 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 `step` function contains only one access of neighbor's information.

Lemma 2. *For the* `step` *function, let* $\preceq$ *be a binary relation defined by* $x \preceq y \iff (\exists m.\ y = g(x, m))$. *Three conditions are assumed:*

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

Then, $h_{\texttt{step}}$, $h'_{\texttt{step}}$, *and* $\preceq_{\mathsf{G}}$ *satisfy the premise of Lemma 1, where the first two are respectively complete and partial one-step processing over the graph by* `step` *and the last one compares graphs based on vertex-wise comparison using* $\preceq$.

Proof (sketch). The first condition and the definition of $\preceq$ guarantee the ordering between $h_{\texttt{step}}$ and $h'_{\texttt{step}}$. The antisymmetry of $\preceq_{\mathsf{G}}$ easily follows from the second condition. The third condition together with the first one and the commutativity of $\oplus$ guarantees the monotonicity of $h_{\texttt{step}}$. □

The first condition of Lemma 2 can be taken to mean that message delay is not harmful. This is a natural requirement for asynchronous execution.

Example: SSSP. For SSSP, the definition of the relation $\preceq$ is instantiated as $x \preceq y \iff \exists w.\ \min\{x, w\} = y$, which is equivalent to $x \geq y$. Therefore, confirming the three conditions is easy.

Implementation. The first and second conditions can be checked using either QE or SMT. Note that the second is equivalent to $\forall x, m, w.\ (g(g(x, m), w) = x) \Rightarrow (g(x, m) = x)$, 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.

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

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

3.4 Prioritized Execution

Another interesting optimization asynchronous execution enables is prioritized execution [3,6,7]. For example in SSSP, a prioritized execution may more intensively process vertices nearer to the source, like Dijkstra's algorithm.

Prioritized execution typically focuses on vertices whose values are *nearer* to the final outcome and thus likely contribute to the final outcome of other vertices.

Therefore, it is natural to use $\preceq$ defined in Lemma 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 cannot be used for processing vertices efficiently using priority queues. A practical solution to this problem is to check whether a known linear order, $\geq$ for example, is consistent with $\preceq$; i.e., $\forall x, y.\ (x \preceq y) \Rightarrow (x \geq y)$. If it is, the linear order can be used for prioritization. Note that an SMT solver can check the condition.

3.5 Limitation and Generalization

We have assumed that information reading 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. Aggregator's result may change regardless of message arrival. Therefore, if the k-th communication is an aggregator, the following condition should be checked instead of Property (3).

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

Namely, the vertex value should not change regardless of the aggregator's value if the vertex receives no message. Since it contains a quantifier, unless QE is used, an executable sufficient condition is needed. A natural candidate is to check the following condition instead.

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

If it holds, a vertex having $\dot{u}$ can be inactivated if $(\bigwedge_{1 \leq i \leq n} \neg p_i(\dot{u}))$ 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 the other graph processing higher-order functions were neglected. The restriction is theoretically not essential. The Fregel compiler normalizes other forms of programs into exactly the one in Fig. 3. Nevertheless, from the practical perspective, since the normalization complicates programs, it is questionable whether normalized programs can be effectively optimized.

4 Implementation and Evaluation

The feasibility of our method was evaluated by implementing it in the Fregel compiler.

Table 1. Graphs for experiments

Name	#Vertices	#Edges
WebBerkStan	685,230	7,600,595
RoadNet-PA	1,088,092	1,541,898
Rand-1M10M	1,000,000	10,000,000

4.1 Implementation

The implementation uses the Z3 SMT solver[3] (version 4.3.2). Because the current Fregel backend, Giraph, does not support asynchronous execution, we implemented only communication reduction and vertex inactivation. There is no conceptual difficulty in implementing the other optimizations.

The implementation was mostly straightforward. It may be worth noting that the units for minimum and maximum, respectively $-\infty$ and ∞, are necessary for vertex inactivation. We prepared numerals with $-\infty$ and ∞ and used them instead of the one conventionally used, such as `Int`.

4.2 Setup of Experiments

We applied our optimizations to three Fregel programs:

- SSSP: the SSSP program shown in Fig. 2.
- PageRank: a program that calculates PageRank by repeatedly calculating a weighted sum of the surrounding vertices' values (30 iterations).
- SCC: a program that calculates strongly connected components by repeatedly finding a strongly connected component by a `fregel` function, which propagates the maximum vertex id, and then removing that component from the graph [14].

For each of them, we considered four programs: the original Fregel program, one to which only the communication reduction was applied (CR), one to which the communication reduction and vertex inactivation were applied (CR+VI), and a handwritten Giraph program.

We prepared three graphs: WebBerkStan, RoadNet-PA, and Rand1M-10M (Table 1). The first two were obtained from the Stanford Large Network Dataset Collection [4]. The former is a web graph; the latter is a road network. The last one

[3] Z3 Solver: https://z3.codeplex.com/.
[4] https://snap.stanford.edu/data/.

Table 2. Performances of programs (unit: seconds)

	Original	CR	CR+VI	Handwritten
SSSP/WebBerkStan	233.2	54.2	46.0	26.8
SSSP/RoadNet-PA	146.4	70.8	47.2	32.1
SSSP/Rand-1M10M	16.9	7.3	7.4	4.6
PageRank/WebBerkStan	20.8	–	–	12.7
PageRank/RoadNet-PA	12.6	–	–	7.6
PageRank/Rand-1M10M	26.1	–	–	17.1
SCC/WebBerkStan	1765.2	1413.1	–	254.9
SCC/RoadNet-PA	326.7	154.9	–	55.5
SCC/Rand-1M10M	35.4	28.1	–	12.6

is a graph generated by randomly connecting vertices. All graphs are directed. RoadNet-PA is symmetric, i.e., each edge accompanies the reverse edge.

The environment for the experiment was a PC cluster consisting of 16 computational nodes. Each node consisted of Intel Core i5 CPUs (nine of them were Core i5-2500, and the other seven were i5-7600), 8-GB memory, and a 128-GB SSD. As the backend of Fregel, we used Giraph 1.3.0, Hadoop 1.2.1, and Java 1.8.0_141 running on Debian 4.9.6-3. We used 16 workers for each experiment.

4.3 Results

For all programs, optimizations were performed immediately (within 0.1 s). For SSSP, both communication reduction and vertex inactivation were possible. For PageRank, both optimizations were impossible. For SCC, although the optimizer guaranteed that both optimizations were possible, the Fregel compiler could not introduce vertex inactivation because Giraph does not support vertex reactivation after all the vertices become inactive. If vertices are inactivated based on Property 3, after finding a strongly connected component, they should be reactivated to find another strongly connected component. The handwritten Giraph program instead inactivates vertices that are removed from the graph. This optimization is impossible based on Property 3 because its justification requires an analysis beyond a single `fregel` function.

As shown in Table 2, the original program for SSSP were significantly slower than the handwritten program. Our optimization removed most of the inefficiencies, leading to a program that were roughly only 1.5 times as slow as the handwritten one. Although our method was not able to optimize PageRank, the difference between the original and the handwritten programs was relatively small. For SCC, while the communication reduction was effective, absence of the vertex inactivation make the optimized program less efficient than the case of SSSP. The program is especially slow for WebBerkStan. We guess that the inefficiency comes from the iterative nature of the SCC algorithm, which requires

a lot of supersteps (thereby synchronization barriers) for analyzing graphs that contain many strongly connected components.

Possibility of Other Optimizations. For SSSP and SCC, the other optimizations, barrier removal and prioritized execution, are theoretically applicable. They may be effective especially for SCC. The maximum vertex id is intensively propagated without being interrupted by barriers. Lemma 2 cannot be applied to PageRank. Existing asynchronous implementations of PageRank use the previous messages of neighbors if new ones have not yet arrived. Lemma 2 considers processing computations using arrived messages only.

5 Related Work

As mentioned in the introduction, the optimizations discussed are not new. Vertex inactivation is a part of the core functionality of Pregel [1]. The communication reduction technique for SSSP was also reported [1]. Many VcGP frameworks use asynchronous execution [15–17]; moreover, some combine asynchronous and synchronous execution to further improve efficiency [2,3]. Some frameworks [3,6,7] support prioritized execution as well. The effectiveness of these optimizations has been intensively studied. Our contribution is their automation using constraint solvers.

Modern constraint solving techniques including QE and SMT have been used for program analysis and synthesis [18]. A typical application is optimization of nested loops, especially *stencil* loops [19–21]. Our optimization can be understood as a variant of such loop optimizations. For instance, introduction of asynchronous execution is essentially an exchange of the inner and outer loops, which is a typical application of constraint solving techniques. From this perspective, the distinctive feature of our study is that it deals with graph manipulation programs. Graph manipulation tends to form irregular complex loops and may not be captured by formalisms supported by constraint solvers, e.g., a system of linear inequalities. Our study focused on VcGP rather than general graph processing and provided a supporting lemmas (Lemmas 1 and 2) that enable constraint solvers to perform optimizations.

Most related systems are Elixir [6,22] and Distributed SociaLite [23]. Elixir automatically derives efficient distributed graph processing from the logical specifications of the output graph. It uses an SMT solver to specify the vertices that should be processed at each step. Distributed SociaLite is a graph processing language similar to Datalog. It accelerates SSSP-like computation by using the generalized Δ-stepping algorithm [24], in which vertices are processed according to a special priority, if a certain kind of monotonicity property is detected. Both start from declarative programs and apply nontrivial optimizations by analyzing certain properties. Unfortunately, both require programmers to provide some clues for optimizations. For instance, with Elixir, programmers should specify the conditions for sending messages and the priorities for processing vertices. With Distributed SociaLite, the generalized Δ-stepping is applied only if programmers

use certain operators. In addition, both frameworks are based on asynchronous execution. We have shown that intensive use of constraint solvers enables many interesting optimizations to be applied to nearly annotation-free deterministic programs.

6 Conclusion and Future Work

We have developed a method of automatically applying nontrivial optimizations to declarative VcGP programs. The key is the use of constraint solvers to reveal the program properties. In our experiments, optimizations were achieved within a realistic time frame and led to significant performance improvement.

We are developing another backend of Fregel based on Pregel+[5]. The new backend will enable more rigorous evaluation of our method.

Our approach to optimize Fregel programs can be used for other declarative graph processing frameworks [6,7,22,23,25]. These frameworks generally require users to write specific programs (e.g., adding annotations and/or invoking certain API functions) in order to apply nontrivial optimizations. It would be interesting if constraint solvers enabled these optimizations to be applied to naively written programs.

Acknowledgements. The authors are grateful to Shigeyuki Sato for discussion with him about related work. This study is partly supported by JSPS Kakenhi JP26280020 and JP15K15965.

References

1. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Elmagarmid, A.K., Agrawal, D. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 135–146. ACM (2010)
2. Xie, C., Chen, R., Guan, H., Zang, B., Chen, H.: SYNC or ASYNC: time to fuse for distributed graph-parallel computation. In: Cohen, A., Grove, D. (eds.) Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 194–204. ACM (2015)
3. Liu, Y., Zhou, C., Gao, J., Fan, Z.: Giraphasync: supporting online and offline graph processing via adaptive asynchronous message processing. In: Mukhopadhyay, S., Zhai, C., Bertino, E., Crestani, F., Mostafa, J., Tang, J., Si, L., Zhou, X., Chang, Y., Li, Y., Sondhi, P. (eds.) Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, pp. 479–488. ACM (2016)
4. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "think like a vertex" to "think like a graph". PVLDB **7**(3), 193–204 (2013)
5. Quamar, A., Deshpande, A., Lin, J.J.: NScale: neighborhood-centric large-scale graph analytics in the cloud. VLDB J. **25**(2), 125–150 (2016)

[5] Pregel+: www.cse.cuhk.edu.hk/pregelplus/.

6. Prountzos, D., Manevich, R., Pingali, K.: Elixir: a system for synthesizing concurrent graph programs. In: Leavens, G.T., Dwyer, M.B. (eds.) Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, pp. 375–394. ACM (2012)
7. Cruz, F., Rocha, R., Goldstein, S.C.: Declarative coordination of graph-based parallel programs. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, pp. 4:1–4:12. ACM (2016)
8. Emoto, K., Matsuzaki, K., Hu, Z., Morihata, A., Iwasaki, H.: Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing. In: Garrigue, J., Keller, G., Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 200–213. ACM (2016)
9. Caviness, B.F., Johnson, J.R. (eds.): Quantifier Elimination and Cylindrical Algebraic Decomposition. Springer, Vienna (1998). https://doi.org/10.1007/978-3-7091-9459-1
10. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
11. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
12. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: Gangemi, A., Leonardi, S., Panconesi, A. (eds.) Proceedings of the 24th International Conference on World Wide Web, WWW 2015, pp. 1307–1317. ACM (2015)
13. Verma, S., Leslie, L.M., Shin, Y., Gupta, I.: An experimental comparison of partitioning strategies in distributed graph processing. PVLDB **10**(5), 493–504 (2017)
14. Salihoglu, S., Widom, J.: Optimizing graph algorithms on pregel-like systems. PVLDB **7**(7), 577–588 (2014)
15. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Thekkath, C., Vahdat, A. (eds.) Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, USENIX Association, pp. 17–30 (2012)
16. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
17. Han, M., Daudjee, K.: Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. PVLDB **8**(9), 950–961 (2015)
18. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017)
19. Größlinger, A., Griebl, M., Lengauer, C.: Quantifier elimination in automatic loop parallelization. J. Symb. Comput. **41**(11), 1206–1221 (2006)
20. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 132–146. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78791-4_9

21. Pouchet, L., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: convexity, pruning and optimization. In: Ball, T., Sagiv, M. (eds.): Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 549–562. ACM (2011)
22. Prountzos, D., Manevich, R., Pingali, K.: Synthesizing parallel graph programs via automated planning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 533–544. ACM (2015)
23. Seo, J., Park, J., Shin, J., Lam, M.S.: Distributed socialite: a datalog-based language for large-scale graph analysis. PVLDB **6**(14), 1906–1917 (2013)
24. Meyer, U., Sanders, P.: [Delta]-stepping: a parallelizable shortest path algorithm. J. Algorithms **49**(1), 114–152 (2003)
25. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: graph processing in a distributed dataflow framework. In: Flinn, J., Levy, H. (eds.) Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2014, pp. 599–613. USENIX Association (2014)