Incrementalization of Vertex-Centric Programs

Timothy A. K. Zakian
Dept. of Computer Science
University of Oxford
Oxford, United Kingdom
timothy.zakian@cs.ox.ac.uk

Ludovic A. R. Capelli
School of Informatics
The University of Edinburgh
Edinburgh, United Kingdom
l.capelli@ed.ac.uk

Zhenjiang Hu
Information Systems Architecture Research Division
National Institute of Informatics
Tokyo, Japan
hu@nii.ac.jp

***Abstract*— As the graphs in our world become ever larger, the need for programmable, easy to use, and highly scalable graph processing has become ever greater. One such popular graph processing model—the vertex-centric computational model—does precisely this by distributing computations across the vertices of the graph being computed over. Due to this distribution of the program to the vertices of the graph, the programmer "thinks like a vertex" when writing their graph computation, with limited to no sense of shared memory and where almost all communication between each on-vertex computation must be sent over the network. Because of this inherent communication overhead in the computational model, reducing the number of messages sent while performing a given computation is a central aspect of any efforts to optimize vertex-centric programs.**

While previous work has focused on reducing communication overhead by directly changing communication patterns—by altering the way the graph is partitioned and distributed, or by altering the graph topology itself—in this paper we present a different optimization strategy based on a family of complementary *compile-time program transformations* in order to minimize communication overhead by changing both the messaging and computational structures of programs. Particularly, we present and formalize a method by which a compiler can automatically *incrementalize* a vertex-centric program through a series of compile-time program transformations—by modifying the on-vertex computation and messaging between vertices so that messages between vertices represent patches to be applied to the other vertex's local state. We empirically evaluate these transformations on a set of common vertex-centric algorithms and graphs and achieve an average reduction of 2.7X in total computational time, and 2.9X in the number of messages sent across all programs in the benchmark suite. Furthermore, since these are compile-time program transformations alone, other prior optimization strategies for vertex-centric programs can work with the resulting vertex-centric program just as they would a non-incrementalized program.

***Index Terms*—Compilers, Vertex-Centric Computation**

I. Introduction

As the scale of real-world graphs that need to be computed over rapidly grows, with graphs many times being made of billions of nodes and edges, the need for *highly-parallel*, and *scalable* graph processing solutions has grown ever greater. One such model for computing over large-scale graphs is the *vertex-centric* model [1]. The vertex-centric model of graph computations has been the center of an intense amount of recent research and development efforts, and has lead to popular vertex-centric frameworks such as Pregel [2], Pregel+ [3], GraphLab [4], Apache Giraph [5] and many more [6], [7], [8]. These vertex-centric programming systems have allowed programmers to easily express computations over graphs at the largest scales [9], [2], and have become not just a useful, but crucial part of a programmer's arsenal when working with graph data at scale.

Each of these modern vertex-centric frameworks are based on the Bulk Synchronous Parallel (BSP) model of computation [10], and the computation is performed as a series of iterative *supersteps*. Each superstep is comprised of a parallel execution of a user-defined program—which takes in messages from other vertices along incoming edges and sends messages along outgoing edges—that is placed on each vertex of the graph being computed over as it is distributed amongst computation nodes or *workers*. This is followed by communication between vertices, and finally a global barrier synchronization is performed to ensure that all messages have been delivered before the start of the next computational step, or *superstep*.

The vertex-centric model of computation sacrifices memory and computational locality for scalability, thus while the model is incredibly scalable it is by definition incredibly communication intensive. Due to this inherent communication overhead, since the original Pregel paper [2] popularized the computational model there has been an explosion of research in trying to speedup computations in this setting by hybridizing the model to be a subgraph-centric computational model [11], [12], and determine better partitioning of the graph amongst the computational nodes [3], [13]. All with the eventual goal of minimizing the number of messages sent in the computation as a whole, and minimizing communication bottlenecks.

The techniques that we present in this paper to incrementalize programs can be seen as being motivated by this same view towards optimizing vertex-centric programs by minimizing communication. However, while previous research has sought to minimize communication by changing the computational model visible to the programmer, or by changing the graph structure of the program, we take a different view. Particularly, we start our research quest by first asking the following question:

> How do we make sure that every message that is sent during the computation is *meaningful*?

Where we intuitively call a message sent from a vertex meaningful if it differs from the most recently sent message from that vertex.

Once our program has the property where every message sent between vertices is meaningful, we can *incrementalize* the computation as a whole: since messages are only sent when a message value has changed, each previous "whole message" can be converted to a “Δ-message”, where the the delta from the previous message is sent instead, and where we cache intermediate values relating to these messages locally within the on-vertex computation. Thus in this setting a lack of a message from a vertex u can be seen as representing an affirmation of cache coherency with u, and a Δ-message from u represents a “patch” to be applied to the cache on the receiving vertex in order to keep it coherent with the updated message value from u.

However, incrementalizing the computation in this manner requires that we change the way the computation within each vertex interacts with the messages it receives. We therefore must specify a number of transformations that need to be performed in order to allow not just sending meaningful-only messages, but also sending and interacting with Δ-messages. Since we need to alter the computational and messaging structure of the program in order to have these properties, we depart from the majority of previous work which has taken a library-oriented or shallow embedding view towards vertex-centric computing and instead move to a deeply-embedded setting where we have access to the program AST and can easily analyze and manipulate it.

Changing the world in which vertex-centric computations are expressed from the shallowly-embedded computational-model-as-library realm into one in which users write programs that are *compiled* down to a vertex-centric framework such as Pregel+ means that our guiding question can be refined a bit more:

> What *compile-time transformations* can be performed in order to ensure that every message sent by the compiled vertex-centric program is meaningful?

Since we are compiling to a vertex-centric framework—Pregel+ in our case—the optimizations and transformations that we perform in order to ensure the above property could be obtained by-hand by anyone in any of the library-based frameworks we have mentioned thus far. However, taking the language-based approach presented in this paper, the programmer does not have to think about any of these optimizations themselves when writing their program and instead writes a normal vertex-centric algorithm, which is both shorter, easier to understand, boiler-plate free, and faster than the equivalent non-“ninja”’d algorithm in Pregel+.

In this paper we make the following contributions:

- We present a novel message passing policy for vertex-centric computations that ensures that only new (“meaningful”) results are communicated, and where the default state for a vertex is to be halted.
- We present the first vertex-centric language with automatic incrementalization of messages and vertex-computations.
- We evaluate the suite of transformations that we present, and show that they can reduce the total execution time by up-to 4.4X and the number of messages sent by up-to 5.8X on PageRank. Furthermore, we attain an average speedup of 2.7X and reduce the number of messages sent by 2.9X across our entire suite of benchmarks comprised of a set of common vertex-centric programs.

II. Vertex-Centric Computation

As mentioned in Section I Pregel-like vertex-centric computational frameworks are designed around the BSP model of computation, where different vertices of the graph are distributed or associated to different machines in the cluster, and where each vertex u keeps track of its adjacency list (*i.e.*, the neighbors of u) along with a *vertex state* which can contain any number of user-defined fields that the user can access, and whose values persist from one superstep to another. In order to define a computation over a graph in this model, the programmer implements a `compute()` function, which runs on each vertex in the graph and proceeds in iterations called *supersteps*. In each superstep the `compute()` function is called on each currently *active* vertex u in the graph.[1] The `compute()` function then performs the task specified by the user for each active vertex u. Each `compute()` function has access to the messages from incoming edges at the previous superstep, can read and write to its own vertex state, can send messages which are to be received in the next superstep to the vertex’s neighbors, and can make the vertex inactive by calling a `vote_to_halt()` function. Once a vertex is halted, it is no longer considered for computation unless it is “reactivated” by receiving an incoming message. Once all vertices in a computation have halted, and there are no more messages pending, the computation as a whole terminates.

Since the computation is distributed amongst many different machines m_i with often-times multiple workers per-machine, vertex-centric frameworks allow users to implement message *combiners* which combine messages from (all the vertices on) one machine m_j to another vertex u on a different machine m_i. These combiners are crucial to reducing inter-worker communication; by using them instead of sending n messages from a one machine to a single vertex, we can instead combine all of the n local messages on the machine into a single message before sending that *single* message to the destination vertex. However, since there is no guarantee about the order in which messages are combined, the combiner operation is restricted to be both commutative and associative.

For a more detailed and comprehensive description of the programming model and the various aspects and features of it, we refer the reader to [2].

III. Building some Intuition

Many times during the execution of a vertex-centric program, a given vertex may compute the same value to send as a message from one superstep to the next. We’ll call such messages “meaningless” since a repeated message with the same value does not provide any new information to the computation that could not be remembered from previous (vertex-local) information.[2] This section provides the intuition and motivation for the transformations that we will be performing in Section VI in order to prevent sending meaningless messages and to incrementalize the compiled

[1]At the beginning of the computation all vertices start out *active*.

[2]In other words, by accurately *memoizing* the vertex computation.

```
1  void compute(const MessageContainer& msgs) {
2    if(step_num() == 1) {
3      value().pr = 1.0 / get_vnum();
4    } else {
5      double sum = 0;
6      for(double message : msgs) {
7        sum += message;
8      }
9      value().pr = 0.15 + 0.85 *
10       (sum / get_vnum());
11   }
12   if(step_num() < 30) {
13     double msg = value().pr /
14       value().neighbors.size();
15     for (VertexID v : value().neighbors) {
16       send_message(v, msg);
17     }
18   } else { vote_to_halt(); }
19 }
```

Fig. 1: The `compute()` function for PageRank in Pregel+.

program. We'll use PageRank [14] as it is defined in Figure 1 as a running example throughout this section.

1) Meaningful Messages: We can imagine many cases in which the `compute()` function calculates the same pagerank value (`msg`) from one superstep to another. To prevent sending the same message twice in a row, we modify the code to first check whether the value of `msg` has changed from the previous superstep before sending. To do this we replace lines 15-17 of Figure 1 with the following:

```
...
// Determine if we have a new value to send
if (msg != value().old_msg) {
  for (VertexID v : value().neighbors) {
    send_message(v, msg); // New value, so send
  }
  // Update most recently sent value
  value().old_msg = msg;
} // Otherwise, don't send
...
```

where the old value of `msg` is remembered in a new vertex field `value().old_msg`. However after this transformation we run into a problem: previously each vertex received *all* of its neighbors pageranks at *every* superstep, however in the transformed program, messages are no longer sent unless the message value has changed. Because of this, the summation of the neighbors pageranks in lines 6-8 of Figure 1 is no longer correct since (possibly) not all of the neighbors pageranks will be sent at every superstep. We therefore need change how we interact with the messages sent by our neighbors, and change the way we calculate their sum.

2) Memoizing Aggregations: In order to resolve this issue, we memoize the summation of our neighbors pageranks from one superstep to the next. This transformation is possible since while we have lost the invariant that we receive the pagerank for all of our neighbors, we have gained the following invariants:

1) If we receive a message it is non-equal to the previous message sent from that vertex; and
2) If we haven't received a message from a vertex, then the value of the message it *would* have sent is the same as the most recent message from that vertex.

Using these, we can memoize the summation from one superstep to the next by adding a field to our vertex—`value().sum`—that records the sum computed at the previous superstep. Then, once we receive a set of messages we use this field to get the sum for the current superstep:

```
...
for (double message : msgs) {
  value().sum += message;
}
...
```

But there's a problem with this method: we still don't calculate the correct sum. While we are only sending changing values, we are sending *whole messages i.e.*, our messages carry the *new* pagerank for a vertex as opposed to the *delta from the previous pagerank*. Thus when we memoize the aggregation as above, we wind up double-counting the previous values for the message that we received in the sum. In order to fix this, we need to change the values that we send between vertices, from *whole messages* to *Δ-messages* that tell us how to change the already computed value.

3) Δ-Messages: To send modified messages, we introduce a function `computeDelta` that given an old message value, and a new message value computes the delta between the two:

```
double computeDelta(double oldMsg, double newMsg) {
  return newMsg - oldMsg;
}
```

Calls to `send_message` are then updated to send Δ-messages (computed with `computeDelta`) instead of directly sending `msg` as was done previously. Thus the final code for lines 15-17 in Figure 1 is

```
...
for (VertexID v : value().neighbors) {
  // Determine if we have a new value to send
  if (msg != value().old_msg) {
    // New value, so compute the delta
    double delta = computeDelta(value().old_msg, msg);
    send_message(v, delta); // send the _delta_
    // Update most recently sent value
    value().old_msg = msg;
  } // Otherwise, don't send
}
...
```

At this point, our transformed program has the following new properties:

1) We only send a message when the value being sent is a meaningful update;
2) The "boundary" of the vertex-function has been incrementalized so it only aggregates over the set of changed values at each step instead of all neighbors; and
3) The messages that we send represent *changes* to previous message values as opposed to *new message values*.

Each of these properties is crucial to the incrementalization of the program: incrementalization could not be performed without changing the messages that are sent to Δ-messages, and both of these optimizations are only possible in conjunction with a meaningful-only messaging policy.

The series of program transformations that we have performed to our PageRank program in order to arrive at these properties can be mechanized as compiler passes. Furthermore, the memoization and incrementalization transformations can be generalized to arbitrary commutative and associative operations. In the rest of this paper we formalize the properties about vertex-centric computations, messages, and program transformations that we have discussed in this section and demonstrate the effectiveness of these program transformations empirically.

IV. Making Things Precise

Now that we have built our intuition about why, and how we transform programs we turn our attention to formalizing these notions.

A. The Meaning of Meaningful Messages

Earlier on we appealed to the reader's intuition regarding what constitutes a "meaningful" message. We now make this notion precise.

Definition 1 (Meaningful Messages): A message m_1 sent from a vertex u_1 to another vertex u_2 at superstep n, is *meaningful* if any of the following statements hold:

1) $n = 0$; or
2) If m_2 is the most recent message sent from u_1 to u_2, then $m_1 \neq m_2$; or
3) No message was sent from u_1 to u_2 for all supersteps k, $k < n$.

Less formally, at the beginning of the computation every message is considered "meaningful", and after that we never want to send two messages with the same value in a row between the same two vertices. This definition of what constitutes a meaningful message is meant to encapsulate the idea of memoization in the vertex-centric context: a message is meaningful iff there is no efficient way for the other (receiving) vertex to cache that messages value from the most recent value sent by that vertex.

In this meaningful-messages only setting, the absence of a message to a vertex is meaningful in itself, since it tells us that the previous value of the *sending* vertex has not changed *i.e.*, that the *cache is coherent* for that vertex (as viewed from the receiving vertex), and we therefore do not need to invalidate the local "cache" for that vertex's value on the receivers side. Thus in this setting messages between vertices not only represent new values to be used, but also in a sense *cache invalidations* for the receiving vertex's cache. However, due to the distributed nature of the computation, how we update that cache after we have "invalidated" it with a new message is a crucial. This is where an *incrementalization policy* is brought into play.

B. Incrementalization

1) Memoization: The Inefficient Approach: When we think of caching for vertices, the first idea that comes to mind is to cache the individual value for each of the vertex's neighbors locally within the vertex's state based on an approach where we keep a lookup table from vertex id to that vertex's value. When we receive a message we update this lookup table with the new mapping from the sending vertex's identifier to value, and instead of directly iterating over the `messages` within the vertex function, we instead use this lookup table as a proxy for the messages to that vertex.

While this lookup-table-based method of caching neighboring vertices message values allows us to fulfill the requirement of only sending meaningful messages throughout the computations, it has a number of issues that make this method impractical and inefficient. In particular: each message sent must be tagged with the sending vertex's id which in many cases can double the size of each message; and each vertex keeping a local lookup table from vertex id to value can increase the size of the vertex state and hence memory footprint of the computation considerably.

As the reader has probably surmised after reading Section III, this is not the approach we take in this paper. While we are able to get rid of meaningless messages through the above memoization this reduction in messaging comes at such a considerable cost that the resulting computation can run even slower than the original. The key realization to resolving these issues lies in blurring the interaction between the messages we receive and the computation that we perform within the vertex function. In other words, we don't want to simply cache neighboring message values, but instead *incrementalize* the *interactions* between vertices.

2) Efficiency Through Incrementalization: Automatically incrementalizing programs is a well-studied and active field of research, however in this setting we are only interested in methods which allow us to *statically* incrementalize the program using compile-time program transformations. While a standard way of performing static incrementalization of programs is by deriving a transformation ∂ [15] such that given a value a, a function f and a change from that a, da, the following equation holds: $f(a \oplus da) \simeq (f\,a\,) \oplus (\partial f\ a\ da)$.[3] However, in our case this version of incrementalization—while static and only using program transformations—will not work since our goal is to not necessarily incrementalize the vertex-program itself, but instead incrementalize the vertex's *interactions with other vertices* and through this incrementalize the vertex computation as a whole.

Beyond simply needing to incrementalize the interactions as opposed to the vertex-program itself, we also want this transformation to be subject to the conditions that the resulting transformed program is still efficient: the incrementalized program should use a minimal amount of additional memory per vertex compared to the non-incrementalized program, and further, the size of the messages should not change. Furthermore, due to the highly distributed nature of vertex-centric computations, given a new message m' to send, and the previous message m we need to be able to determine the incrementalized, or Δ-message, for m' on the senders side (*i.e.*, only using the senders vertex state). This locally determinable incrementalization of messages is a crucial property since it helps ensure the efficiency of the incrementalization policy; without this property we would incur massive communication

[3]Where $\simeq$ is meant to denote denotational equality of expressions, and $\oplus$ is a binary operation subject to certain conditions that they outline in [15].

$$\begin{array}{lll}
u & ::= & \mathit{vertexName} \\
\mathit{uop} & ::= & - \mid \texttt{not} \\
\mathit{pop} & ::= & \texttt{min} \mid \texttt{max} \\
\mathit{op} & ::= & + \mid - \mid * \mid / \mid \&\& \mid || \mid < \mid > \mid \geq \mid \leq \mid == \\
\boxplus & ::= & + \mid * \mid \texttt{min} \mid \texttt{max} \mid || \mid \&\& \\
g & ::= & \texttt{\#in} \mid \texttt{\#out} \mid \texttt{\#neighbors} \\
a & ::= & \texttt{ref} \mid \underline{\texttt{x}_\texttt{f}} \\
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{float} \\
\mathit{prog} & ::= & \texttt{init}\ \{e\}; \mathit{stmt} \\
\mathit{stmt} & ::= & \texttt{step}\ \{e\} \mid \texttt{iter}\ i\ \{e\}\ \texttt{until}\ \{e\} \mid \mathit{stmt}; \mathit{stmt} \\
e & ::= & x \mid \underline{\texttt{x}_\texttt{f}} \mid b \mid i \mid f \mid e\ \mathit{op}\ e \mid \mathit{pop}\ e_1\ e_2 \mid \mathit{uop}\ e \\
 & \mid & e;e \mid u.a \mid x = e. \mid \texttt{graphSize} \mid \texttt{infty} \\
 & \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & \mid & \texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2 \mid \boxplus\ [e \mid u \leftarrow g] \\
 & \mid & \texttt{for}(u : g)\{e\} \mid \texttt{send}(u, x) \mid \texttt{halt}
\end{array}$$

Fig. 2: The syntax of ΔV. Language forms that are `highlighted` are forms that are not visible to the user, and that are part of the internal representation of the AST used by the compiler.

overheads and have to impose inefficient caching policies similar to the one discussed in Section IV-B1. Thus as we will see in Sections VI-D and VI-E the compile-time transformations performed to the computation *within* the vertex in order to allow us to send incremental messages—or Δ-messages—and the definition of the Δ-messages that we send are defined in terms of (or at least need to take into account) one another.

V. The Language

The syntax of the language that we are compiling from is a small imperative query language that we call ΔV, and is defined in Figure 2. It is designed to be small and not particularly special in any way since we want the transformations we express over it to be easily transferable to other languages. Further, we want the compilation process to be as uncomplicated as possible for expository purposes.

A ΔV program consists of an initialization expression `init`$\{e\}$ that is run at each vertex at the start of the computation, and before any communication has taken place. After the initialization expression has been run, each statement is run sequentially, in-order, and homogeneously across all vertices. Each statement can either be a single `step`$\{e\}$ which represents a computation to be run for one single superstep, or an `iter` i $\{e_1\}$ `until` $\{e_2\}$ which keeps iterating e_1 until the expression e_2 is satisfied.

Being a small imperative language, the expressions in ΔV are largely straightforward, with the only two interesting aspects being the $\boxplus[e|u \leftarrow g]$ form which functions as an aggregation operation over the vertices represented by g, and the vertex-field access form $u.a$ which states what field of *the other vertex u* we are accessing within the aggregation expression e. Thus if our vertex-state had a field $\underline{\texttt{amt}}$, and we wanted to compute the sum of the $\underline{\texttt{amt}}$s of our neighboring vertices we could do so with the following aggregation operation

$$+\ [u.\underline{\texttt{amt}} \mid u \leftarrow \texttt{\#neighbors}] \qquad (1)$$

Where we *always* aggregate over a graph-expression g with `#in` and `#out` corresponding to in- and out-vertices in the case that the graph is directed, and `#neighbors` referring to the neighboring vertices in the case that the graph is undirected.[4]

This aggregation operation represents a useful—but not crucial for our optimization techniques—difference from the normal programming model exposed for vertex-centric computation: the vast majority are *push-based* where the programmer directly sends and receives messages, whereas in ΔV the language is *pull-based* where the (user-visible) vertex function directly operates over the value of its neighbors, as opposed to the messages sent to it by other vertices *e.g.*, `#in`, or `#out` neighbors.

The difference between the push-based and pull-based programming models can be seen by comparing the push-based version of PageRank that we saw earlier in Figure 1 where the programmer directly sends messages on one superstep that will be received as `messages` on the next superstep, and the following definition of PageRank in ΔV

```
init {
  // initialize the values
  local vl : float = 1 / graphSize;
  local pr : float = vl / |#neighbors|
};
iter i {
  // sum neighbors PageRanks
  let sum : float = + [ u.pr | u <- #neighbors ] in
  // calculate new value and new pagerank for
  // neighbors to see next superstep
  vl = 0.15 + 0.85 * (sum / graphSize);
  pr = vl / |#neighbors|
} until {
  i >= 30 // stop after 30 iterations
}
```

As we will see when we start compiling the language, specifically demarcating aggregation operations in this manner will make it easier for us to determine the best places to incrementalize the program. However, having these forms is not crucial to the optimizations that we present, and the aggregation points could be determined through flow-analysis on the program.

VI. Incrementalizing Programs

a) Notation: Since the transformations that we will be performing on the AST are traversal independent, we use a context-based rewriting notation to detail the program transformations that we perform. Since they only take place over expressions, the contexts can be viewed as expressions with "holes". Thus when we write $C[]$ we will mean an expression with a hole in it such as $C[] = 1 + []$, and $C[e]$ represents a filling of this hole with the expression e: $C[e] = 1 + e$.

The holes in our contexts are restricted to only appearing in places where an expression is allowed (*i.e.*, expression context). We say that $C[e_1] = e$ if there exists a valid context $C[]$ such

[4]In the case that the graph is undirected, `#in` and `#out` mean the same thing as `#neighbors`.

that $C[e_1] = e$, and we will also say in this case that $e_1 \in e$. Using this notation, we define the context-based compilation of an expression e by saying that $C[e_1] \rightsquigarrow C[e_1']$ if, for every $e_1 \in e$, e_1 is transformed into e_1'. Thus after this $e_1 \notin e$. Furthermore, before all passes are run a type-annotation pass has been run on our source program, so we can always get the type of any expression e by calling the *typeOf*(e) function.

A. Aggregation Conversion

In this pass we convert communication that is based on the pull-based model of ΔV to the push-based model that we need for Pregel. To do this we employ the same technique as other pull-based vertex-centric languages such as Palgol [16], Green-Marl [17], and Fregel [18], where we use the fact that since the computation is homogeneous across the graph a "pull" by a vertex, is a "push" by its neighbors on the previous superstep. Since this is a standard compilation technique we don't go into the details here and simply summarize the process as follows:

- We first convert all aggregation expressions that are not the immediate right hand side of an assignment (or `let`) and bind the result of the aggregation to a new variable and then substitute this variable in for occurrences of that aggregation.[5] After this *all* aggregations that we will encounter are of the form $x = \boxplus[e \mid u \leftarrow g]$.[6]
- At the first superstep (*i.e.* immediately after initialization of the vertex state) send the data from the neighbors perspective: field accesses of a vertex within an aggregation expression such as $u.\underline{\mathtt{amt}}$ in Equation (1) become sends of that field ($\underline{\mathtt{amt}}$) to that vertex's neighbors.
- At subsequent steps every vertex inspects the message list `messages` (`msgs` in Figure 1) that it receives in order to obtain values for its neighbors, and then executes the aggregation, so the following transformation is performed:

$$C[x = +[u.\underline{\mathtt{amt}} | u \leftarrow g]] \rightsquigarrow C\left[\begin{array}{l}\mathtt{let}\ tmp : \tau = \mathtt{default\_init}(+, \tau)\ \mathtt{in} \\ \quad \mathtt{for}(m : \mathtt{messages})\{tmp{+}= m.\underline{\mathtt{amt}};\}; \\ \quad x = tmp\end{array}\right] \tag{2}$$

where τ is the type of $\underline{\mathtt{amt}}$, and $\mathtt{default\_init}(\boxplus, \tau)$ returns the default initialization for the type τ and aggregation operation $\boxplus$. For the above example, since the aggregator is $+$, we get that $\mathtt{default\_init}(+, \tau) = 0$.[7]

B. Adding Vertex State

In order to remember the previous message sent from one superstep to the next, each expression e that appears within a $\mathtt{send}(u, e)$ is added as a field to the vertex state.[8] This can be viewed as a type of A-normalization [19] except instead of binding to `let`-bound variables we are binding to fields in our vertex state:

$$C[\mathtt{send}(u, e)] \rightsquigarrow C\left[\begin{array}{l}\underline{\mathtt{freshVar_e}} = e; \\ \mathtt{send}(u, \underline{\mathtt{freshVar_e}})\end{array}\right] \tag{3}$$

where $\underline{\mathtt{freshVar_e}}$ is a unique variable name and is associated with the type $\tau_e = \mathit{typeOf}(e)$. We then add, or "bind" each of these new fields to our vertex state:

```
VertexState { τe freshVare; ... }
```

and remember each of these fields of our vertex that is accessed within `sends` for the next pass.

C. Inserting Change Checks

In the previous pass we made sure that we only send fields that are unmodified from our vertex state, and we kept track of all of the fields that were ever sent. We'll call these the *externally visible fields* of the vertex, since they are the only things that other vertices can know about any other vertices.

We now add a "dirty-bit" to the vertex state to track whether any of these externally-visible fields has changed during a computational step. Since we haven't sent anything at the first superstep, the dirty-bit is pre-set in the initial vertex state:

```
VertexState { bool dirty = true; ... }
```

At the beginning of each superstep (except the first) we save the current state of each externally visible field $\underline{\mathtt{x_f}}$ in a temporary variable o_f. Then whenever we encounter an assignment to that field we possibly dirty this dirty-bit by comparing the new value to our saved value of that field at the beginning of the superstep:

$$C[\underline{\mathtt{x_f}} = e] \rightsquigarrow C\left[\begin{array}{l}\underline{\mathtt{x_f}} = e; \\ \underline{\mathtt{dirty}} = \underline{\mathtt{dirty}} \parallel (\underline{\mathtt{x_f}} \neq o_f)\end{array}\right] \tag{4}$$

Now, when we encounter sends of a field to neighboring vertices, we check whether the dirty-bit has been set

$$C[\mathtt{send}(u, \underline{\mathtt{x_f}})] \rightsquigarrow C\left[\begin{array}{l}\mathtt{if}(\underline{\mathtt{dirty}}) \\ \mathtt{then}\ \mathtt{send}(u, \underline{\mathtt{x_f}})\end{array}\right] \tag{5}$$

Furthermore, when we encounter such a send in a broadcast setting we can lift the if-expression outside of the loop and prevent execution of the loop entirely if the message to be broadcast has not changed:

$$C[\mathtt{for}(u : g)\{\mathtt{send}(u, \underline{\mathtt{x_f}})\}] \rightsquigarrow C\left[\begin{array}{l}\mathtt{if}(\underline{\mathtt{dirty}}) \\ \mathtt{then} \\ \quad \mathtt{for}(u : g)\{ \\ \quad\quad \mathtt{send}(u, \underline{\mathtt{x_f}}) \\ \quad \}\end{array}\right] \tag{6}$$

D. Incrementalizing Aggregations

Having altered the program so messages are sent only when the value has changed from the most recently sent value, we turn to incrementalizing the interaction between the vertices in the graph computation as a whole. To do this, we first alter aggregations while also taking into account how the $\Delta_u^m(m')$ function in the next pass will be determined.

We alter the aggregations by hoisting each aggregation operation so that it is memoized in the vertex state. We thus go back

[5] *i.e.*, we A-normalize [19] with respect to aggregations.

[6] Or `let` bound, but the reasoning for these is identical so we will WOLOG only handle assignments in this section.

[7] It is important to pass in the type along with the aggregation operation to `default_init` since for certain aggregation operations we would need to return the maximum or minimum value for τ.

[8] Unless e is already a field of the vertex.

and change the operation that was performed in Equation (2), and instead perform the following transformation:

$$C[x = \boxplus[e|u \leftarrow g]] \rightsquigarrow C\left[\begin{array}{l}\texttt{for}(m : \texttt{messages})\{\\ \quad \underline{\texttt{aggAcc}} = \underline{\texttt{aggAcc}} \ \boxplus \ e\\ \};\\ x = \underline{\texttt{aggAcc}}\end{array}\right] \tag{7}$$

where $\underline{\texttt{aggAcc}}$ is a unique variable name associated with that particular aggregation expression. We then add this accumulator to the vertex state so that the previous value is remembered from one superstep to the next.

```
VertexState {
  τ aggAcc = default_init(⊞, τ);
  ...
}
```

where τ and `default_init` are just as in Section VI-A. Thus this can be seen as performing the same function as the transformation that was done in Equation (2) with the exception that instead of `let`-binding to a temporary variable as is standard when converting to A-normal form, we instead use a vertex field so that we can use this to memoize the aggregations. Thus each aggregation within the vertex function starts with the value from the previous superstep. This then means that the messages that are sent only need to say how to change the previous value to get the new one, or in other words how to calculate the delta from the previous value.

1) Dealing with multiplicative operations: Memoizing things in the above manner works well in most cases, however in multiplicative contexts, such as with $\times$ or && aggregation operators, if we receive a "nullary-message" of 0 or `false` our accumulator is now nulled out, and there is no way to recover to a correct non-null accumulator while still keeping a meaningful-messages only policy—even if the offending nullary-message subsequently becomes a non-nullary value. In order to solve this problem, if we encounter a multiplicative aggregation operation instead of tracking one value we in fact track three values locally within the vertex's state. These consist of the non-nulled result ($\underline{\texttt{nnAcc}}$), the number of nullary expressions ($\underline{\texttt{aggNulls}}$) not included in the $\underline{\texttt{nnAcc}}$ field, and a final value for the aggregation at that superstep ($\underline{\texttt{aggAcc}}$). When we have no nullary expressions in the aggregation (*i.e.*, $\underline{\texttt{aggNulls}}$ `== 0`) we set the value of $\underline{\texttt{aggAcc}}$ to the value of the non-null result $\underline{\texttt{nnAcc}}$, however if we still have nullary expressions in the aggregation ($\underline{\texttt{aggNulls}}$ `> 0`) then we set the final value of $\underline{\texttt{aggAcc}}$ to the nullary value for the type and operation of the aggregation. Thus when we receive a nullary-message we increment the number of nullary expressions in our aggregation but we do not do anything else.[9] On the other hand, when we receive a non-nullary message, we check the tag of the message to see if it was a previously null message, and decrement the number of nullary elements in the aggregation ($\underline{\texttt{aggNulls}}$) if so. We thus add the following fields to the vertex state

[9]Due to the meaningful-messages only policy, we know that it was a non-previously null message, and we therefore increment the number of nullary results.

```
VertexState {
  int aggNulls = 0;
  τ nnAcc = default_init(⊞, τ);
  τ aggAcc = default_init(⊞, τ);
  ...
}
```

and perform the following transformation for multiplicative aggregations

$$C[x = \boxplus\ [e \mid u \leftarrow g]] \rightsquigarrow$$
$$C\left[\begin{array}{l}\texttt{for}(m : \texttt{messages})\{\\ \quad \texttt{if}\ (\texttt{is\_nullary}(\texttt{m}))\\ \quad \texttt{then}\ \underline{\texttt{aggNulls}} = \underline{\texttt{aggNulls}} + 1;\\ \quad \texttt{else}\\ \qquad \underline{\texttt{nnAcc}} = \underline{\texttt{nnAcc}} \ \boxplus \ e;\\ \qquad \texttt{if}\ (\texttt{prev\_nullary}(\texttt{m}))\\ \qquad \texttt{then}\ \underline{\texttt{aggNulls}} = \underline{\texttt{aggNulls}} - 1;\\ \}\\ \texttt{if}\ (\underline{\texttt{aggNulls}} == 0)\\ \texttt{then}\ \underline{\texttt{aggAcc}} = \underline{\texttt{nnAcc}};\\ \texttt{else}\ \underline{\texttt{aggAcc}} = \texttt{nullary\_elem}(\boxplus, \tau);\\ x = \underline{\texttt{aggAcc}}\end{array}\right] \tag{8}$$

The $\Delta_u^m(m')$ needs to track nullary messages values and tag non-nullary messages with whether or not the previous message (m) from that vertex was a nullary message or not.

E. Δ-Message Insertion

Now that we have memoized the aggregation operations between supersteps, we now turn to determining the $\Delta_u^m(m')$ function that computes the incrementalized messages. However before we do this, we insert calls to calculate Δ-messages. Since we already saved the values of our externally visible fields by the transformation in Section VI-C we use this saved value when computing the Δ-message. We thus perform the following transformation

$$C[\texttt{send}(u, \underline{\texttt{x}_\texttt{f}})] \rightsquigarrow C[\texttt{send}(u, \Delta_u^{o_f}(\underline{\texttt{x}_\texttt{f}}))] \tag{9}$$

Now that calls to our message incrementalization function have been inserted, we determine what exactly this function should be by the following equation:

$$x \boxplus m' \simeq (x \boxplus m) \boxplus \Delta_u^m(m') \tag{10}$$

where x is an accumulator, $\boxplus$ is an aggregation operation, and $\simeq$ is (as before) denotational equality. In more intuitive language, for each aggregation we track the fields that are sent to that aggregation, and the aggregation operation performed (*e.g.*, for the PageRank example we would get $\{+ \mapsto \texttt{pr}\}$). Then for an aggregation operation $\boxplus$, we synthesize $\Delta_u^m(m')$ such that Equation (10) holds.

In the case that the aggregation is multiplicative (subject to possible nulling) we tag each message with whether or not the previous message was nullary or not. Tagging for multiplicative operations is very useful here, since *e.g.*, in the $\boxplus = \times$ case, we need to avoid division by zero in the $\Delta_u^m(m')$

TABLE I: Graph datasets used in our evaluation.

Dataset	Type	|V|	|E|
Wikipedia[11]	Directed	18.27M	136.54M
LiveJournal-DG[12]	Directed	4.85M	68.48M
Facebook[13]	Undirected	59.22M	185.04M
LiveJournal-UG[14]	Undirected	3.99M	34.68M

function, and in this case we calculate the incrementalized message by the following:

$$\Delta_u^m(m') = \begin{cases} m'/m & \text{if } m \neq 0 \\ tag(m') & \text{if } m = 0 \end{cases}$$

F. Addition of Halts

While this pass is not necessary for correctness, the transformations that we have performed in order to ensure a meaningful-only messaging policy exposes a powerful characteristic that we can take advantage of. Recall that in our meaningful-only messaging policy, messages also serve as cache invalidations, thus once a vertex has computed a specific value and sent its messages, the only way the values of the messages that it sends can change is by it receiving new messages.[10] However any *halted* vertex will be woken up if it receives any new messages. Since we only send meaningful messages, this means that after sending its messages, *every vertex can halt after every superstep.*

As opposed to the previous transformations which all applied to expressions e in ΔV, this pass is instead a straightforward transformation on statements in ΔV which simply inserts `halts` at the end of every statement given by the following:

$$\begin{aligned} &\texttt{step}\{e\} \rightsquigarrow \texttt{step}\{e;\texttt{halt}\} \\ &\texttt{iter } i\ \{e\} \texttt{ until } \{e\} \rightsquigarrow \begin{array}{l}\texttt{iter } i\ \{e;\texttt{halt}\} \\ \texttt{until } \{e;\texttt{halt}\}\end{array} \end{aligned} \tag{11}$$

VII. Evaluation

In this section we evaluate the usefulness of the transformations that we have presented by evaluating them on a number of common vertex-centric benchmarks. In particular we compare the total computation time, number of messages sent, and total bandwidth of the program for PageRank (PG) [14], Single-Source-Shortest Paths (SSSP), Connected Components (CC), and a non-converging version of Hyperlink-Induced Topic Search (HITS) [20] where we perform the hub and authority updates simultaneously. For each of these benchmarks, we compare ΔV to reference implementations for each benchmark written in Pregel+ [3], and compare ΔV to itself *without* message-reduction optimizations— $\Delta V^\star$. All experiments were conducted on an Amazon EC2 cluster with 8 m4.xlarge nodes, each having 4 vCPUs and 16GB of memory running two workers per node and connected by 750Mbps ethernet. The datasets that we use are commonly used real-world graphs representing different social networks (Facebook, LiveJournal-DG, and LiveJournal-UG) and the hyperlink network of Wikipedia. Their details are listed in Table I.

[10]We assume here that all operations are deterministic, and we don't have access to functions such as `rand()`.

TABLE II: Size (in bytes) of vertex state for ΔV, and ΔV without incrementalization ($\Delta V^\star$) as compared to Pregel+ and Palgol vertex state.

Variant	PageRank	SSSP	CC	HITS
ΔV	48 B	48 B	48 B	80 B
$\Delta V^\star$	40 B	40 B	40 B	64 B
Palgol	40 B	64 B	40 B	64 B
Pregel+	32 B	40 B	32 B	56 B

A. Change in Vertex State Size

Since we want the compiled and incrementalized program to use as little additional memory as possible as compared to a non-incrementalized version, we compare the size of the vertex states generated by both ΔV and $\Delta V^\star$, and compare this to the vertex states generated by Palgol [16]—another language that compiles to Pregel+—in addition to the size of the vertex state in the reference implementations in Pregel+.

As we see from Table II, the compiled vertex state sizes in general are larger than their comparable vertex state in a hand-written Pregel+ program. This difference is due to the fact that source programs in ΔV, $\Delta V^\star$, and Palgol all need to be compiled to a state machine within the Pregel+ `compute()` function, and additional internal state-tracking variables are required in order to track these states and transitions from one superstep to the next. Thus, while in general the hand-written Pregel+ vertex-state sizes are smaller than our compiled vertex states, we are not as interested in this comparison; the state tracking that leads to the majority of this increase in vertex-state size is needed by any vertex-centric compilation scheme. Instead the more useful comparison is between ΔV and the two non-incrementalized languages; $\Delta V^\star$ and Palgol. In this case we see that while incrementalization adds some additional space overhead, this additional space is fairly minimal compared to the overall size of the vertex state.

B. Performance and Message Reduction

In order to evaluate the viability of the transformations that we have presented, we not only want to show that we speed-up programs where the optimizations can be useful, but also that we don't necessarily slow down computations by performing these transformations when they are not useful, or required. In order to evaluate both of these properties we compare ΔV to itself without the optimizations that we have presented—$\Delta V^\star$—along with the same algorithms implemented in Pregel+. The results of these benchmarks are presented in Figure 3 and are obtained by taking the average of three runs (with the slowest and fastest runs marked with error bars). PageRank was run for 30 iterations, and HITS for 7 (5 after 2 initialization steps).

As we see from Figure 3, while Pregel+ is always faster than $\Delta V^\star$, on certain iterative algorithms (PG & HITS) the optimizations that we have performed in ΔV can

[14]http://konect.uni-koblenz.de/networks/dbpedia-link
[14]https://snap.stanford.edu/data/soc-LiveJournal1.html
[14]https://archive.vn/tVl1G
[14]https://snap.stanford.edu/data/com-LiveJournal.html

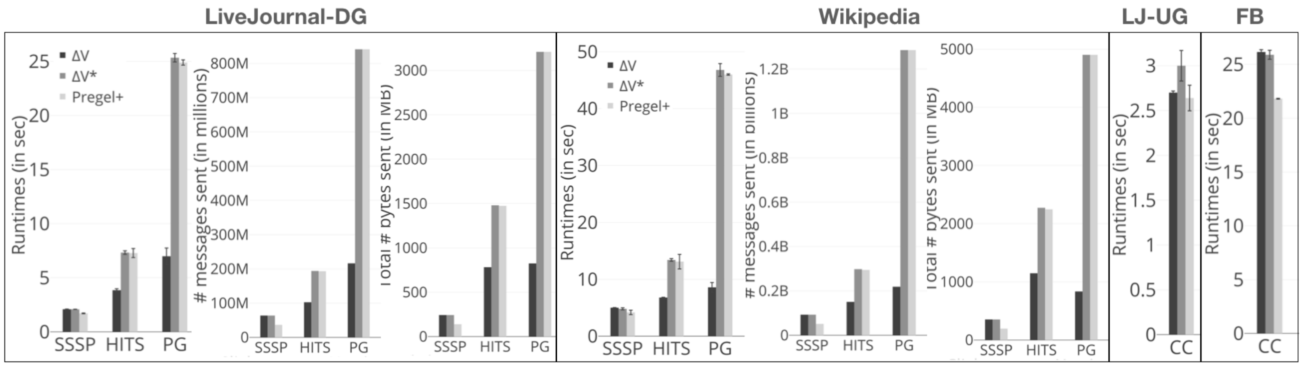

Fig. 3: Execution time, number of messages sent, and total number of bytes sent for PageRank, SSSP, and HITS (left two). And execution time for Connected Components on LiveJournal-UG and Facebook graphs (right two). Total bytes sent and message numbers have been elided for CC since all variants sent the same number and size of messages.

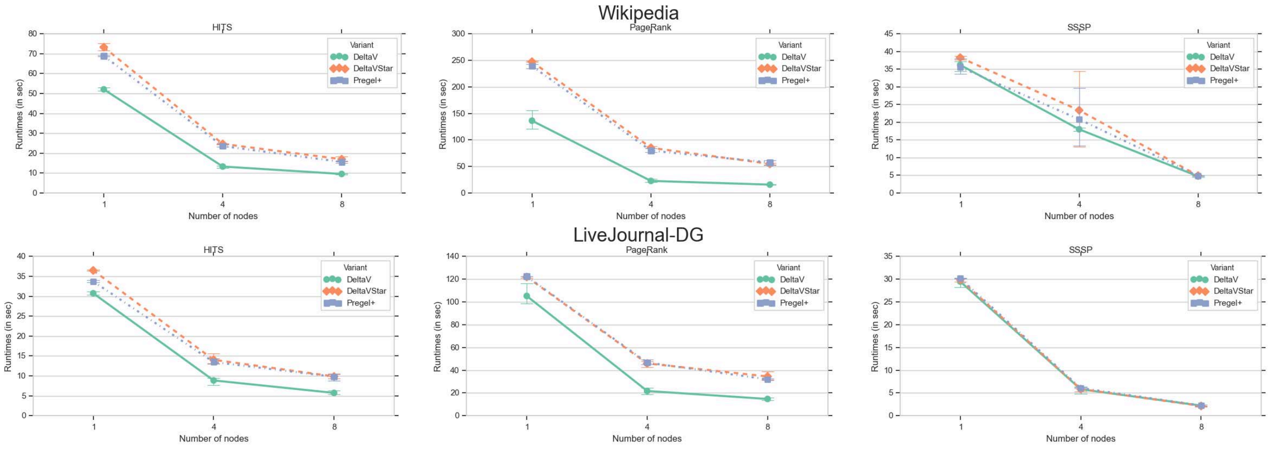

Fig. 4: Scaling graphs for HITS, PageRank, and SSSP on Wikipedia (top) and LiveJournal-DG (bottom). All nodes are run with two workers per node.

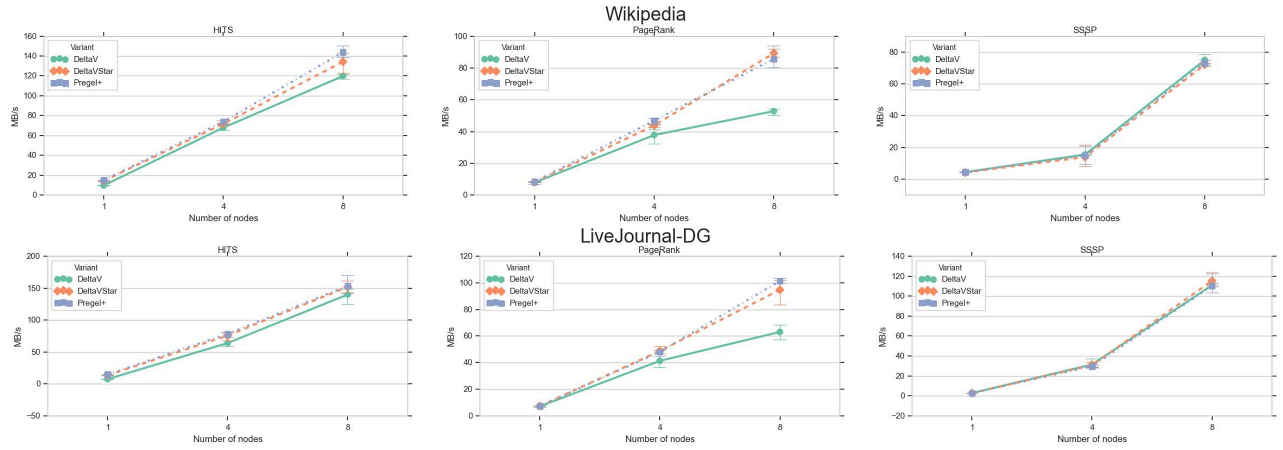

Fig. 5: Average MB/s for HITS, PageRank, and SSSP on Wikipedia (top) and LiveJournal-DG (bottom) graphs on 1, 4, and 8 nodes. All nodes are run with two workers per node.

increase performance significantly. With PageRank having an average speedup of 4.4X across the Wikipedia and LiveJournal graphs and HITS showing an average speedup of 1.9X as well, as compared to Pregel+. Furthermore, we see a 5.8X and 1.9X reduction in the number of messages sent for the PageRank and HITS programs respectively.

On other programs that are less amenable to incrementalization, such as CC (Figure 3) and SSSP (Figure 3), we do not see any performance improvement at all. With $\Delta V^\star$ and ΔV in fact sending the exact same number of messages in both cases.[15] The lack of a speedup from our optimizations for these programs makes sense; in both CC and SSSP the standard algorithm is in a sense "pre-incrementalized": the `compute()` function only sends new messages when a new value has been attained at that vertex. The most important thing then for these two benchmarks is that the optimizations that we have performed in ΔV have not slowed the computation down.

C. Scaling and Messaging Overheads & Bottlenecks

This section explores how the incrementalization strategy in ΔV scales across different numbers of computational nodes. This is particularly interesting with respect to our incrementalization strategy, since as we decrease the number of nodes, the ratio of on-node work and communication to inter-network message-passing increases from what we saw in the previous section where we used 8 nodes. To look at these scaling properties, we compare the runtimes of ΔV,$\Delta V^\star$, and Pregel+ programs for HITS, SSSP, and PageRank on the same graphs as in the previous section, and running on 1, 4, and 8 nodes, with each node assigned two workers. The results of these scaling experiments can be seen in Figures 4 and 5.

As we see from these graphs, while the runtime reductions remain relatively similar across node counts, as the number of nodes increases and the graph computation starts becoming more distributed across the network, the incrementalization approach shows some benefits with respect to the bandwidth needed to perform the computation, while the runtime speedup performance remains relatively constant. We see this particularly in the graph for PageRank, in which while overall runtime speedup relative to $\Delta V^\star$ and Pregel+ decreases slightly from 4 to 8 nodes, we in fact see a marked decrease in bandwidth needed for the computation.

While it may seem odd at first that the decrease in MB/s of the computation is not reflected in the runtime of PageRank, it is in fact perfectly understandable, and surfaces an important point: due to the BSP nature of vertex-centric computations, vertices with high degree can create communication bottlenecks—since the entire computation must wait for all messages to be delivered before the next superstep can start. Because of this, even though we reduced the total number of messages and amount of data being sent across the computation as a whole, there still may exist communication hotspots that slow down the rest of the computation in a non-linear way due to the BSP nature of the computation. Removing these bottlenecks from vertex-centric algorithms is a research topic in its own right and was one of the core contributions of [3], where so-called *ghost vertices* were added as a variant of core Pregel+. While we are not able to explore these bottleneck removal optimizations as they effect ΔV programs at this time, doing so should be relatively straightforward; since the optimization is a graph-load/runtime-based solution, adding this should entail only a slight modification of the code-generation phase and runtime for ΔV.

[15]For CC ΔV, $\Delta V^\star$, and Pregel+ sent the exact same number of messages so this graph has been elided.

VIII. Related Work

The majority of previous work for vertex-centric computation has taken a library-oriented, or shallow embedding view such as Pregel [2], Pregel+ [3], Apache Giraph [5], Graphlab [4] and many others [6], [7], [8]. While this library-based approach has the benefit of being lightweight and portable, a library-based system is unable to gather information about the program as a whole, or transform it. Thus a library-oriented approach is not suitable for determining and preventing non-meaningful messages, and we need to instead compile, or deeply-embed our language.

For compilation, we have followed the approach of other deeply-embedded, or language-based approaches to vertex-centric programming such as Palgol [16], Green-Marl [17], and Fregel [18] in which the programmer writes a vertex-centric algorithm kernel which is then compiled to a vertex-centric (library-based) framework. However, the approach that we have taken towards incrementalization of the language can be seen as an extension of different techniques in the wider library-based vertex-centric community over the past years. Particularly, Apache Flink [8] has supported delta iterations—which are very similar to our idea of an incrementalization of a vertex-centric program—for a number of years, however the user still needs to hand-write and transform their code in order to use these capabilities. Other compiled DSLs for vertex-centric computations have also worked on communication optimizations as well [16], [17], however none have sought to automatically incrementalize the program during the compilation process for vertex-centric computations, or to reduce the number of messages sent in the compiled program in this manner.

IX. Conclusions & Future Work

In this paper we have introduced a family of program transformations that together incrementalize the interaction between vertex computations and through this incrementalize the computation as a whole (Section VI). We have demonstrated the usefulness of these transformations (Section VII), and shown that programs written in our prototype language ΔV that have been incrementalized by these transformations can achieve improved performance on iterative algorithms. Furthermore, these transformations are straightforward, readily applicable to other (compiled) DSLs for vertex-centric computation, and not only do not restrict other standard optimization techniques for vertex-centric programs but open up other low-level optimization techniques. Indeed, while we have demonstrated the effectiveness of incrementalization there are still a number of future directions and possibilities to explore that are directly related to the method of automatic incrementalization that we have presented.

Future Work

One future direction for this work is to extend the incrementalization policy to allow removing vertices from the graph during the computation; while ΔV is a query language and thus cannot manipulate the graph, the incrementalization approach that we have presented can be extended to handle these manipulations. A vertex could be deleted from the graph in the same manner as described in Pregel [2], however with the addition that the vertex being deleted first broadcasts a message that zeros out the value of the vertex to its neighbors before the deletion of the edges is performed (*i.e.* the vertex needs to set the neighbors value for that vertex's most recent message to a unitary value.).

Another future direction is to allow the programmer to define an "allowable slop" parameter ϵ to the program where a message value is counted as changed if it is no longer within ϵ of the old message. Thus we wouldn't send a message from a vertex as long as the new message is within the ϵ of the most recently sent message, but the change to the outgoing message value is tracked internally within the vertex's state from one superstep to another. Once the message value differs by at least ϵ from the most recently sent message it is counted as changed, and a message is sent. With this view, the work described in this paper could be seen as a degenerate case of this where $\epsilon = 0$.

The vertex-centric framework (Pregel+) that we compile to does not currently take advantage of the halt-by-default policy in a compiled ΔV program. However, we believe that there is significant performance benefits that could be realized by changing the way Pregel+ determines (and schedules) which vertices should be run at any given superstep under this policy. In a none halt-by-default setting, at each superstep *each vertex in the graph* needs to be accessed in order to determine if it should be run (since the vertex may be active). However, in a halt-by-default setting, with the exception of the first superstep, the vertices that run at any given step are determined by the messages that are received at the previous superstep. Using this knowledge, we could use the message passing framework to build a work-queue based scheduler for the computation, as opposed to how it currently stands where we need to loop through each and every vertex in the graph at every superstep, in order to determine which vertices should run.

Acknowledgements

This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099. And by the International Internship Program of the National Institute of Informatics of Tokyo and the UK Engineering and Physical Sciences Research Council [grant number EP/L01503X/1, CDT in Pervasive Parallelism]. Timothy Zakian was supported by the Clarendon Fund.

References

[1] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 25, 2015.

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[3] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1307–1317.

[4] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[5] A. S. Foundation, "Giraph," http://giraph.apache.org/, 2012.

[6] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 22:1–22:12.

[7] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: a system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 169–182.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[9] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.

[10] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[11] A. Guerrieri, A. Montresor, and S. Centellegher, "Etsch: Partition-centric graph processing," in *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences*. IEEE, 2016, pp. 706–713.

[12] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From think like a vertex to think like a graph," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.

[13] R. Dindokar, N. Choudhury, and Y. Simmhan, "A meta-graph approach to analyze subgraph-centric distributed programming models," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 37–47.

[14] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[15] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann, "A theory of changes for higher-order languages: Incrementalizing lambda-calculi by static differentiation," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 145–155.

[16] Y. Zhang, H.-S. Ko, and Z. Hu, "Palgol: A high-level dsl for vertex-centric graph processing with remote data access," in *Programming Languages and Systems*, B.-Y. E. Chang, Ed. Cham: Springer International Publishing, 2017, pp. 301–320.

[17] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 349–362.

[18] K. Emoto, K. Matsuzaki, Z. Hu, A. Morihata, and H. Iwasaki, "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*, ser. ICFP 2016. New York, NY, USA: ACM, 2016, pp. 200–213.

[19] A. Sabry and M. Felleisen, "Reasoning about programs in continuation-passing style." in *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, ser. LFP '92. New York, NY, USA: ACM, 1992, pp. 288–298.

[20] J. M. Kleinberg, "Hubs, authorities, and communities," *ACM Comput. Surv.*, vol. 31, no. 4es, Dec. 1999.