An Empirical Analysis on Expressibility of Vertex Centric Graph Processing Paradigm

Siyuan Liu
Nanyang Technological University, Singapore
slu019@e.ntu.edu.sg

Arijit Khan
Nanyang Technological University, Singapore
arijit.khan@ntu.edu.sg

Abstract—We study the vertex-centric (VC) paradigm for distributed graph processing, popularized by Google’s Pregel system. Since then, there were several attempts to implement many graph algorithms in a vertex-centric framework, as well as efforts to design optimization techniques for improving their efficiency. Many follow up works experimentally compared the efficiency and scalability of existing VC systems. However, to the best of our knowledge, there has not been any systematic study to analyze the expressibility of the VC paradigm itself.

Our work addresses this gap in the following ways. We consider multiple distributed algorithms for two important graph problems: single-source shortest path and betweenness centrality. We investigate, with thorough experiments, whether all these algorithms can be efficiently implemented in a VC framework. We find that all distributed algorithms (often the more efficient ones) cannot be effectively implemented in the VC paradigm. We conclude by discussing our recommendations on the road ahead.

Index Terms—vertex-centric (VC) paradigm; expressibility; single-source shortest path; betweenness centrality;

I. INTRODUCTION

In order to achieve low latency and high throughput over massive graph datasets, distributed solutions were proposed in which the graph and its data are partitioned horizontally across cheap commodity servers in a cluster. The vertex-centric (VC) distributed programming model for large graphs has been popularized by Google’s Pregel framework [29]. It hides distribution related details such as data partitioning, communication, underlying system architecture, and fault tolerance behind an abstract API. Also known as the think-like-a-vertex model, it requires that the user expresses the computation from the perspective of a single vertex, by providing a high-level vertex-compute() function.

In Pregel, which was inspired by the Bulk Synchronous Parallel (BSP) model [41], graph algorithms are expressed as a sequence of iterations called supersteps. Each superstep is an atomic unit of parallel computation. During a superstep, Pregel executes a user-defined function for each vertex in parallel. The user-defined function specifies the operation at a single vertex \(v\) and at a single superstep \(S\). The supersteps are globally synchronous among all vertices, and messages are usually sent along the outgoing edges from each vertex. In 2012, Yahoo! launched the Apache Giraph [11] as an open-source project, which clones the concepts of Pregel.

With the inception of the Pregel framework, vertex-centric distributed graph processing has become a hot topic in the database community (for a survey, see [20], [25], [30], [43]). Although Pregel provides a high-level distributed programming abstract, it suffers from efficiency issues such as the overhead of global synchronization, large volume of messages, imbalanced workload, and straggler problem due to slower machines. Therefore, more advanced vertex-centric models (and its variants) have been proposed, e.g., asynchronous (GraphLab [27]), asynchronous parallel (GRACE [42]), barrierless asynchronous parallel (Giraph Unchained [19]), gather-apply-scatter (PowerGraph [16]), data parallel (GraphX [17], Pregelix [6]), and subgraph centric frameworks (NScale [35], Giraph++ [40]). Various algorithmic and system-specific optimization techniques were also designed, e.g., graph partitioning and re-partitioning [26], [46], combiners and aggregators [29], vertex scheduling [27], superstep sharing [19], finishing computations serially [37], among many others.

In this work, we investigate some of the fundamental limitations of the vertex-centric paradigm itself, by implementing a few distributed graph algorithms that are difficult to be expressed in the VC paradigm. Several graph problems have multiple distributed algorithms, which vary in their efficiency. We observe that not all distributed algorithms of a graph problem can be implemented effectively in the vertex-centric framework. In particular, we carefully limit the scope of the paper and focus on two specific classes of algorithms: bucketing-based and multi-phased.

Bucketing-based refers to an algorithm that relies on a bucket structure to define priorities of vertices. Vertices are processed in batch from the highest priority vertices to lowest priority vertices. Vertices can be added and removed from buckets during execution to update their priorities. Example algorithms in this class include \(\Delta\)-Stepping algorithm [32] for single-source shortest path, weighted BFS [12], k-core [11], and approximate set cover [9].

Multi-phased implies those algorithms which consists of several distinct phases. Each phase itself may be iterative. However, overall the algorithm’s execution follows the predefined order of phases. Example algorithms that fall under this category include distributed Brandes’ algorithm [5] for betweenness centrality, finding bi-connected and strongly connected components [45].

We find that often the more efficient distributed algorithm of a graph problem under the above two categories cannot be effectively implemented in the vertex-centric paradigm.
Notice that not all vertex-centric frameworks can implement these two classes of algorithms. For bucketing-based algorithms, synchronous systems like Giraph [1] and GPS [37] are necessary. This is because bucketing-based algorithms require synchronization after each superstep to determine the next bucket for processing. For multi-phased algorithms, systems which support global synchronization in certain but not necessarily in all supersteps (in order to synchronize between phases) are sufficient, such as Giraph Unchained [19].

In summary, we systematically study the expressibility of the vertex-centric framework, so to equip graph system researchers and practitioners with a good understanding of the trade-offs of this paradigm, compared to other distributed frameworks, such as the classic message passing interface (MPI) for distributed computing. Our contributions can be stated as follows.

- To the best of our knowledge, for the first time we implement bucketing-based \(\Delta\)-Stepping (for single-source shortest path problem) and multi-phase Edmond et al.’s (for betweenness centrality computation) algorithms in the vertex-centric framework.
- We highlight the key expressibility limitations (e.g., more work, many active vertices, large message overheads, etc.) of VC paradigm in regards to both bucketing-based and multi-phase algorithms. We conduct extensive experiments to demonstrate these expressibility challenges specific to vertex-centric paradigm, while also presenting that such bottlenecks do not exist in traditional MPI distributed systems.

The rest of our paper is organized as follows. We introduce preliminaries and related work in Section II, expressibility concerns in Section III, coupled with experimental results. We then summarize our findings and conclude in Section IV.

II. BACKGROUND AND RELATED WORK

A. Vertex-Centric Paradigm

For ease of programmability of graph algorithms in parallel and distributed environments, various programming models have been proposed, including vertex-centric (VC) [29], sparse matrix operations [8], Map-Reduce based [7], [14], [23], graph domain specific languages [21], declarative programming [39], and task-based models [34]. Among them, the VC paradigm has received unprecedented interests in databases, data mining, machine learning, and systems communities. While the exact programming APIs for different VC systems [16], [17], [19], [27], [29], [35], [36], [40], [42] somewhat vary, they all require that the user expresses the computation from the perspective of a single vertex, by providing a higher-order \texttt{vertex-compute()} function.

Figure 1 depicts VC programming in Google’s Pregel system. A Pregel computation consists of a sequence of supersteps separated by global synchronization barriers until the algorithm terminates (Figure 1(b)). Within each superstep, the vertices compute in parallel, each executing the same user-defined function that expresses the logic of the algorithm. A vertex can modify its state (active or inactive), receive messages sent to it in the previous superstep, and send messages to other vertices (to be received in the next superstep) (Figure 1(a)). The algorithm, as a whole, terminates when all vertices are inactive and there are no messages in transit.

The execution of a Pregel program initiates running many copies of the user program on a cluster of machines. One of these copies acts as the master. It is not assigned any portion of the graph, but it coordinates worker activities. The master, in fact, determines how many partitions the graph will have, and then assigns one or more partitions to each worker machine. The master also instructs each worker to perform a superstep. The worker iterates over its active vertices, using one thread for each partition. The worker calls \texttt{vertex-compute()} for each active vertex. Messages are sent asynchronously, to enable overlapping of computation, communication, and batching, but are delivered before the end of the current superstep. When the worker finishes the current superstep, it responds to the master, informing how many vertices would be active in the next superstep. These steps are repeated as long as some vertices are active, or some messages are in transit.

The main benefit of VC programming model is that the users only require to write a single \texttt{vertex-compute()} function for a specific graph algorithm, without the knowledge of distributed programming. The Pregel system has been cloned by many open source projects, e.g., Apache Giraph [1], Apache Hama [2], and GPS [36]. In addition to the basic VC paradigm as demonstrated in Figure 1, several optimization techniques have been included in these open source systems, to improve the performance, as well as to support more complex distributed graph algorithms. In the following, we discuss three optimization techniques, which we shall employ in our experiments.

**Combiner.** Sending a message to a vertex on another machine causes overhead. This cost can be reduced if the messages intended for a vertex can be aggregated (via commutative and associative operations), and only that aggregated value...
matters for the vertex. As an example, messages in case of a single-source shortest path algorithm (e.g., Bellman-Ford [29]) consist of potential shorter distances to a receiving vertex. Since this receiving vertex is only interested in the minimum among them, such algorithm greatly benefits by using a combiner.

**Aggregator.** An aggregator ensures global communication, monitoring, and coordination. Each vertex can provide a value to the aggregator at the end of the current superstep, the aggregator combines those values with a reduction operation, and the resulting value is made available to all vertices before the next superstep. For algorithms in which a global coordination is necessary (e.g., distributed Δ-Stepping algorithm [32] for the single-source shortest path problem), an aggregator is required.

**Master-Compute.** In GPS [36] and in later versions of Giraph [1], an optional master-compute() function is executed by the master between supersteps to perform serial computation, and for coordination in algorithms that are composed of multiple phases (e.g., distributed Brandes’ algorithm [5] for betweenness centrality problem).

**B. Performance Comparison of VC Systems**

With the explosion of distributed vertex-centric graph processing systems, many follow up works empirically compared the efficiency and scalability of the existing vertex-centric systems. We survey bulk of these experimental works as follows.

**Han et al.** [20]. One of the early works that empirically compared several VG systems by measuring the running time, maximum memory usage, total network usage, and scalability of different graph algorithms in these systems.

**Lu et al.** [28]. For existing VC systems, they measured the effects of individual optimizations, as well as experimented with five categories of graph algorithms, and graphs having different characteristics (power law, small world, large diameter, etc.).

**Guo et al.** [18]. This work employed additional performance metrics, e.g., raw processing power, resource utilization, setup overhead, horizontal/vertical scalability, to experimentally compare several VC systems.

**Satish et al.** [38]. The authors compared the performance of VC systems with respect to hand-optimized MPI algorithms as a reference, and also provided quantitative suggestions for the improvement of these systems.

**Gao et al.** [15]. They focused on five experimental design decisions, e.g., data distribution, data organization, programming model, synchronization, and message model.

**Capota et al.** [10]. This work analyzed choke points of VC systems, coupled with benchmark datasets generation and an advanced benchmarking harness.

In addition to the aforementioned experimental works, there are many surveys and tutorials, e.g., [22], [25], [30], [43], comparing state-of-the-art VC systems. Yan et al. [45] studied VC implementations of connected component-based graph problems with the notion of balanced practical Pregel algorithms. McSherry et al. [31] measured the running times of single-threaded implementations of several graph algorithms using a high-end 2014 laptop, and compared them with the published results for state-of-the-art distributed graph processing systems. We earlier reported time-processor based complexity results of fifteen VC algorithms [24]. Our current work is different from all these prior works, since we investigate a fundamental limitation of the vertex-centric paradigm itself — from the perspective of expressibility, that is, often the more efficient distributed algorithm of a graph problem cannot be effectively implemented in the VC paradigm.

**C. Experimental Setup**

**1) Datasets:** We summarize our datasets in Table I. Many of these datasets have been extensively used in past research on VC graph processing systems, including [15], [17], [18], [20], [37], [40].

**Road-US:** We download the full USA road network dataset from the 9th DIMACS Challenge (http://www.diag.uniroma1.it/challenge9). An edge weight denotes physical distance from source to target vertex in the road network.

**Web-UK:** The uk-2005 Web graph (http://law.di.unimi.it/datasets.php) is a collection of UK Web pages, which are represented as vertices. A hyperlink in page $u$ to page $v$ is denoted by an edge between them.

**Orkut:** Orkut is an on-line social network where users form friendship with each other. We obtain the dataset from [33].

**KKI:** We download the Kki2009_679_1-bg brain network from http://openconnectome.me/graph-services. The edge weight represents the number of fiber tracts that connect one vertex to another.

**Synthetic:** We generate synthetic graphs using GTgraph [4]. Random graphs are generated following the Erdős-Rényi (E-R) model, since the Δ-Stepping algorithm has a solid theoretical analysis of its complexity over random graphs [32]. The edge generation probability is adjusted so that the number of vertices and edges are proportional to the number of servers. Only the statistics of the largest synthetic graph is shown in Table I.

**2) Cluster Configuration:** We perform experiments on a cluster of 6 servers, each having dual E5-2620 V3 CPUs (2.4GHz), 12 physical cores, and 24 threads (with hyperthreading), 64GB RAM, OS is Ubuntu 16.04 LTS, interconnected by 10Gbits/s Ethernet. We use all 12 cores and 24 threads in each server during our experiments.

---

**Table I: Properties of datasets**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>Characteristics</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road-US</td>
<td>23,974,347</td>
<td>58,333,344</td>
<td>directed, weighted</td>
<td>road network</td>
</tr>
<tr>
<td>Web-UK</td>
<td>39,459,925</td>
<td>93,636,282</td>
<td>directed, weighted</td>
<td>social network</td>
</tr>
<tr>
<td>Orkut</td>
<td>3,072,606</td>
<td>223,534,301</td>
<td>directed, unweighted</td>
<td>biological network</td>
</tr>
<tr>
<td>KKI</td>
<td>1,827,240</td>
<td>40,876,288</td>
<td>directed, weighted</td>
<td>E-R random graph</td>
</tr>
<tr>
<td>Synthetic</td>
<td>6,000,000</td>
<td>600,016,480</td>
<td>directed, weighted</td>
<td></td>
</tr>
</tbody>
</table>
A. Single-Source Shortest Path Computation

are further magnified in this setting. SSSP evaluations, expressibility concerns of the V vertices, and Edmond et al.'s VC systems. Next, we consider both the paths with minimal weight from the source vertex (i.e. no distance update over the graph). Since BC requires multiple iterations, and in each iteration the edges can be traversed in any arbitrary order, it can be easily parallelized. Although it exposes a great amount of parallelism, it is not efficient: the sequential version runs in $O(mn)$ time.

To demonstrate expressibility limitations, we show that the former is more efficient, however cannot be effectively implemented with the VC paradigm, thereby demonstrating expressibility limitations of VC compared to traditional message-passing (MPI) distributed systems. Next, we consider both VC and MPI implementations of Edmond et al.'s distributed algorithm [13] for betweenness centrality (BC) computation. Since BC requires multiple SSSP evaluations, expressibility concerns of the VC paradigm are further magnified in this setting.

A. Single-Source Shortest Path Computation

An edge-weighted, directed graph $G = (V, E, W)$ consists of a set $V$ of $n$ vertices, $E \subseteq V \times V$ is a set of $m$ directed edges, and $W : E \to \mathbb{R}_{\geq 0}$ assigns a non-negative, real-valued weight $W(e)$ to every edge $e \in E$. Given a source vertex $s \in V$, the single-source shortest path (SSSP) problem finds the paths with minimal weight from $s$ to all other vertices, if such a path exists. The weight of a path is defined as the sum of edge weights along this path, and we only report minimum distances from $s$ to all other vertices.

We consider two most notable distributed algorithms for the SSSP computation: Bellman-Ford and $\Delta$-Stepping.

Algorithm 1 Distributed Bellman-Ford Algorithm

Require: source vertex $s$ in graph $G = (V, E, W)$
Ensure: minimum distances from $s$ to all other vertices
1: initialize distance array, $distance[v] = \infty$, for all vertices $v \in V$
2: $distance[s] = 0$ [relax $s$]
3: $s$ sends ($distance[s] +$ edge weight) along all outgoing edges
4: for $i$ from 1 to $|V| - 1$ do
5:  $updated = false$
6:  receive messages along incoming edges of all vertices
7:  for $M$ in messages do
8:    $v = M.destination$
9:    if $M.distance < distance[v]$ then
11:       $updated = true$
12:       $v$ sends ($distance[v] +$ edge weight) along all outgoing edges
13:    end if
14:  end for
15:  if $updated$ then
16:    terminate if no vertex is updated (i.e., $updated = false$)
17: end for

1) Bellman-Ford Algorithm: The Bellman-Ford algorithm is originally a sequential algorithm. However, as it runs in $(n-1)$ iterations, and in each iteration the edges can be traversed in any arbitrary order, it can be easily parallelized. Although it exposes a great amount of parallelism, it is not efficient: the sequential version runs in $O(mn)$ time.

Therefore, distributed Bellman-Ford is often optimized (1) by only performing update over outgoing edges if the distance of the source vertex has just been updated (line 9), and (2) by terminating early if no update is performed in the last iteration (line 15). The pseudocode for the optimized distributed Bellman-Ford algorithm in the MPI paradigm is depicted in Algorithm 1. The message contains the destination vertex id and its new distance. This algorithm can also be easily implemented in the VC paradigm (e.g., Pregel [29]). In fact, it is even simpler in the VC paradigm because the user does not have to write code to manually check if no distance update is carried out over the entire graph in some iteration, and the algorithm should be terminated early. Instead, the user simply calls voteToHalt() on all vertices, and the algorithm will terminate when no message is generated in a superstep (i.e. no distance update over the graph).

Although the optimized distributed Bellman-Ford algorithm is already more efficient than its original form, it still wastes a lot of work by multiple relaxation of the same vertex, as shown below.

Example 1: Consider the graph in Figure 2, and the source vertex is 0. After vertex 1 is relaxed in iteration 1, it sends message to vertex 3 and vertex 3 will be relaxed to the distance 20 in iteration 2. However, 20 is clearly not the shortest distance from vertex 0 to vertex 3. Indeed, vertex 3 will be relaxed again in iteration 3 when it receives a message of distance 6 from vertex 4.

2) $\Delta$-Stepping Algorithm: To reduce the amount of wasted work in Bellman-Ford, while still maintaining sufficient amount of parallelism, the $\Delta$-Stepping algorithm can be used. In this method, vertices are placed into buckets according to
their priority during relaxation. The bucket id is calculated as \([\text{distance}[v]/\Delta]\), where \(\text{distance}[v]\) is the tentative distance of \(v\) from the source vertex \(s\), and \(\Delta\) is a user-defined input parameter. Vertices that are in lower buckets are processed before the ones that are in higher buckets. Therefore, even if a vertex’s distance is updated in a round, the vertex may not send this new distance to its neighbors if it does not belong to the non-empty bucket with the lowest id.

**Example 2:** By assigning a priority based on the tentative distance, the \(\Delta\)-Stepping algorithm avoids the unnecessary relaxation from vertex 1 to vertex 3 in Figure 2 (which happens with Bellman-Ford). Consider the case when we set \(\Delta = 3\). When vertex 1 and 2 are relaxed from vertex 0, vertex 1 is in bucket 3 (i.e., \([10/\Delta] = 3\)), and vertex 2 is in bucket 0 (i.e., \([2/\Delta] = 0\)). After that, vertex 4 will be added to bucket 1 via relaxation. Now when vertex 4 relaxes its neighbor, vertex 3’s distance will be directly set to 6. Note that vertex 1 never gets the chance to set vertex 3’s tentative distance to 20 because its bucket id is too high.

Another important optimization in the \(\Delta\)-Stepping algorithm is the introduction of light and heavy edges. Light edges refer to edges whose weights are smaller or equal to \(\Delta\), while heavy edges refer to those whose weights are larger than \(\Delta\). The idea is that when relaxing along light edges, previously removed vertices may be added again to the current bucket with a shorter distance. By sending along heavy edges only at the end (lines 24-26, Algorithm 2), it is guaranteed that the vertices’ distances in the current bucket are settled, and they need to send along heavy edge only once.

The value of \(\Delta\) can be adjusted to find a balance between efficiency and the amount of parallelism. When \(\Delta = 1\) it is same as Dial’s implementation of Dijkstra’s algorithm, and when \(\Delta = \infty\) it coincides with the Bellman-Ford algorithm.

Similar to Bellman-Ford, the \(\Delta\)-Stepping algorithm can be easily parallelized because the vertices in the same bucket can be processed in any order. Algorithm 2 shows how it could be implemented in the MPI paradigm. The message contains the destination vertex id and its new distance.

**\(\Delta\)-Stepping in VC Paradigm.** When implementing \(\Delta\)-Stepping over VC, we realize that it is not straightforward. Since the users are not allowed to create any global data structure, the bucket structure cannot be implemented as described in the original algorithm. Instead, the users can store each vertex’s current bucket id as a vertex attribute. To calculate the id of the lowest non-empty bucket, a minimum aggregator (i.e., that reduces its input values to the minimum value) is used.

---

**Algorithm 2** Distributed \(\Delta\)-Stepping Algorithm

**Require:** source vertex \(s\) in graph \(G = (V, E, W)\), the value of \(\Delta\)

**Ensure:** minimum distances from \(s\) to all other vertices

1: initialize distance array, \(\text{distance}[v] = \infty\), for all vertices \(v \in V\)
2: initialize bucket array, \(\text{bucket}[i] = \emptyset\), for \(1 \leq i \leq |V|\)
3: add \(s\) to bucket[0] [relax \(s\)]
4: \(\text{distance}[s] = 0\)
5: \(i = 0\)
6: while not all buckets are empty do
7:   initialize an empty deleted set
8:   while bucket[i] is not empty do
9:     for \(v\) in bucket[i] do
10:        \(v\) sends \((\text{distance}[v] + \text{edge weight})\) along light edges
11:        remove \(v\) from bucket[i]
12:        add \(v\) to deleted
13:     end for
14:     receive messages along incoming edges of all vertices
15:     for \(M\) in messages do
16:       \(v = M\).destination
17:       if \(M\).distance < \(\text{distance}[v]\) then
18:         remove \(v\) from bucket[\([\text{distance}[v]/\Delta]\)] [relax \(v\)]
19:         add \(v\) to bucket[\([M\).distance/\Delta]\)]
20:        \(\text{distance}[v] = M\).distance
21:     end if
22:   end while
23:   for \(v\) in deleted do
24:     \(v\) sends \((\text{distance}[v] + \text{edge weight})\) along heavy edges
25:   end for
26:   for \(M\) in messages do
27:     \(v = M\).destination
28:     if \(M\).distance < \(\text{distance}[v]\) then
29:       remove \(v\) from bucket[\([\text{distance}[v]/\Delta]\)] [relax \(v\)]
30:       add \(v\) to bucket[\([M\).distance/\Delta]\)]
31:      \(\text{distance}[v] = M\).distance
32:   end if
33:   end for
34:   \(i = i + 1\)
35: end while

---

**Algorithm 3** \(\Delta\)-Stepping+VC: vertex-compute() of \(\Delta\)-Stepping in VC

**Require:** source vertex \(s\) in graph \(G = (V, E, W)\), the value of \(\Delta\), minimum aggregator \(agg\), current superstep id \(\text{superstep_id}\)

**Ensure:** minimum distances from \(s\) to all other vertices

1: if \(\text{superstep_id} = 0\) then
2:   if \(\text{vertex.id} == s\) then
3:     initialize distance to 0
4:     send distance to all neighbors
5:   else
6:     initialize distance to \(\infty\)
7:   end if
8:   initialize bucket to \(\infty\)
9:   vote to halt
10: else if \(\text{superstep_id} \mod 2 == 1\) then
11:   for \(M\) in messages do
12:     if \(M\).distance < \(\text{distance}\) then
13:       \(\text{distance} = M\).distance
14:       \(\text{bucket} = [\text{distance}/\Delta]\)
15:     end if
16:   end for
17:   if \(\text{bucket} == \infty\) then
18:     vote to halt
19:   else
20:     send bucket to aggregator \(agg\)
21:   end if
22: else
23:   get \(\text{minimal_bucket}\) from aggregator \(agg\)
24:   if \(\text{bucket} == \text{minimal_bucket}\) then
25:     send distance to all neighbors
26:     \(\text{bucket} = \infty\)
27:     vote to halt
28:   end if
29: end if
Expressibility issues with VC paradigm: $\Delta$-Stepping and $\Delta$-Stepping+VC are faster than Bellman-Ford in MPI, but $\Delta$-Stepping+VC becomes slower than Bellman-Ford in Giraph. Furthermore, $\Delta$-Stepping+VC does not finish in 5 hours per vertex in the Road-US dataset over Giraph. We show speedup of different algorithms with respect to Bellman-Ford, where speedup = running time of Bellman-Ford/running time of the algorithm.

![Graph showing speedup comparison](image)

Fig. 3: Expressibility issues with VC paradigm: $\Delta$-Stepping and $\Delta$-Stepping+VC are faster than Bellman-Ford in MPI, but $\Delta$-Stepping+VC becomes slower than Bellman-Ford in Giraph. Furthermore, $\Delta$-Stepping+VC does not finish in 5 hours per vertex in the Road-US dataset over Giraph. We show speedup of different algorithms with respect to Bellman-Ford, where speedup = running time of Bellman-Ford/running time of the algorithm.

In the vertex-compute() function, a vertex only processes itself if it belongs to the current bucket being processed. The pseudocode for the vertex-compute() function is shown in Algorithm 3. Since in the VC paradigm communication happens only between supersteps, the pattern of Send-Receive-Relax in Algorithm 2 has to be broken into 2 supersteps (lines 10-17 for one superstep, and lines 18-25 for another superstep) in order to be implemented in the VC paradigm. Superstep 0 (lines 1-9) is a special superstep used to initialize the algorithm as all computations have to be embedded into the vertex-compute() function, and the user cannot write a separate preprocessing function to initialize it. The vertex stores its distance and bucket, and the message contains only the new distance of an edge’s destination vertex (its id is specified when sending, but not visible on the receiver’s end as per the design of the VC paradigm).

![Graph showing Stepping Working vertices percentage](image)

Fig. 4: Working vertices percentage for $\Delta$-Stepping+VC is close to 0 in many initial supersteps, indicating that much work is wasted in VC.

Note that unlike Algorithm 2, here we do not distinguish between light and heavy edges for efficiency reasons. If we are to separate the relaxation along light and heavy edges, a master-compute() function has to be implemented to handle more complicated algorithm state coordination. For example, when the minimum bucket id changes, the master would detect the change and instruct workers to process heavy edges instead of going directly for light edges in the new minimum bucket. While this reduces the amount of messages along the heavy edges, it also increases the running time of the algorithm. Thus, for $\Delta$-Stepping in VC paradigm, we do not distinguish between light and heavy edges.

Although such an implementation preserves the semantics of the $\Delta$-Stepping method, it loses certain efficiency due to the lack of the actual bucket structure. The efficiency loss happens in two ways.

**First**, suppose in superstep $i$, vertices receive messages, and update their bucket ids. Now they would need to compute the updated minimum bucket id to proceed. To do that, they need to send their ids to the aggregator. As the aggregator only works between supersteps, the vertices have no choice but to stay active going into superstep $(i + 1)$. Only in superstep $(i + 1)$, they can decide whether they should send message to their neighbors based on the updated minimum bucket id. In other words, each iteration in the original $\Delta$-Stepping algorithm now requires two supersteps in VC, in the first superstep vertices having the minimum bucket id are identified (lines 18-25), and in the next superstep, their neighboring vertices are relaxed (lines 10-17).

**Second**, when a vertex has a bucket id larger than the current minimum, it still has to remain active until the turn comes to process that bucket. Consider the graph in Figure 2 with $\Delta = 3$, vertex 1 and 2 will receive messages in the second superstep, and set their bucket id to 3 and 0, respectively. Although vertex 1 will not be processed in the next superstep, it has to stay active. This is because if it is inactive, it can only be activated again by a message from vertex 0. However, as vertex 0 is inactive after it is removed from the buckets and that it has no incoming edge, it will never be able to activate vertex 1. In both situations, remaining active means that we cannot call voteToHalt() on such vertices.

Note that in the VC paradigm, the voteToHalt() function not only helps with algorithm termination, but it also helps improve algorithm performance. When voteToHalt() is called on a vertex, and no message is sent to it in a superstep, the vertex-compute() function will not be called on it in the next superstep, and its vertex state does not need to be read from memory. As many graph algorithms are data-intensive, the less it reads from memory, the better the performance will be. In the implementation of $\Delta$-Stepping over VC, to ensure the correctness of the algorithm, many vertices require to be active...
TABLE II: Single-source shortest path results: MPI vs. VC. We show memory usage and network traffic of different distributed algorithms. ∆-Stepping+VC does not finish in 5 hours per vertex in the Road-US dataset over Giraph.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MPI</th>
<th>Giraph</th>
<th>MPI</th>
<th>Giraph</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory (GB)</td>
<td>Memory (GB)</td>
<td>Network traffic (GB)</td>
<td>Network traffic (GB)</td>
</tr>
<tr>
<td></td>
<td>Bellman-Ford +VC</td>
<td>Bellman-Ford +VC</td>
<td>Bellman-Ford +VC</td>
<td>Bellman-Ford +VC</td>
</tr>
<tr>
<td>Road-US</td>
<td>153.3</td>
<td>148.4</td>
<td>63.2</td>
<td>386.3</td>
</tr>
<tr>
<td></td>
<td>162.8</td>
<td>63.2</td>
<td>14.4</td>
<td>14.6</td>
</tr>
<tr>
<td></td>
<td>15.0</td>
<td>15.0</td>
<td>0.18</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>64.2</td>
<td>64.2</td>
<td>0.18</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>8.2</td>
<td>8.2</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>386.3</td>
<td>386.3</td>
<td>4.2</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>Synthetic</td>
<td>Synthetic</td>
<td>Synthetic</td>
<td>Synthetic</td>
</tr>
<tr>
<td></td>
<td>78.1</td>
<td>55.4</td>
<td>61.8</td>
<td>61.8</td>
</tr>
<tr>
<td></td>
<td>57.9</td>
<td>57.9</td>
<td>82.3</td>
<td>82.3</td>
</tr>
<tr>
<td></td>
<td>8.2</td>
<td>8.2</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>24.9</td>
<td>24.9</td>
<td>4.2</td>
<td>4.0</td>
</tr>
</tbody>
</table>

1 Figures 3(c) and 3(d) report weak scalability, since we vary the number of vertices and edges in the Synthetic graph proportional to the number of servers.
source vertex $s$, as given below.

$$\delta_{so}(v) = \sum_{t \in V} \delta_{st}(v) = \sum_{w : v \in P_s(w)} \frac{\sigma_{sw}}{\sigma_{sw}} \cdot (1 + \delta_{so}(w)).$$

The pseudocode for Brandes’ algorithm is shown in Algorithm 4. Consider the graph in Figure 2. When the source vertex is vertex 0, after the SSSP phase is performed, vertex 3 would have vertex 4 as its predecessor, and both vertices have a $\sigma$ value of 1, i.e., $\sigma_{03} = 1$, and $\sigma_{04} = 1$. Next, when calculating dependency, vertex 4’s dependency is updated by vertex 3 to be 1 (vertex 3 has a dependency of 0 as it is a leaf vertex in the shortest path DAG without any successors, and dependency values are initialized to 0). Indeed, among all shortest paths from source vertex 0, only the path to vertex 3 passes through vertex 4, and that is the only shortest path from vertex 0 to vertex 3.

2) Distributed Edmond et al.’s Algorithm: The Brandes’ algorithm can be easily parallelized by running it from different source vertex $s$ simultaneously. Although such an algorithm can achieve perfect linear speedup due to its embarrassingly parallel nature, the size of the input graph is ultimately limited by the capacity of each machine. Therefore, here we instead consider Edmond et al.’s algorithm [13], that partitions the graph across machines, and is space-efficient. Edmond et al.’s algorithm is based on the sequential Brandes’ algorithm, and consists of four phases. In the first phase, it uses the $\Delta$-Stepping algorithm to find all predecessors of each vertex. In the second phase, the successors are computed based on the predecessors. In the third phase, it counts the number of shortest paths for each vertex using the set of successors. In the last phase, it traverses the shortest path DAG in the reverse direction to calculate the dependency $\delta_{so}(v)$ for all vertices. For the last two phases, the graph is traversed in a level-synchronous fashion. An overview of Edmond et al.’s algorithm over message-passing (MPI) paradigm is shown in Algorithm 5.

Consider the graph in Figure 2 with source set to 0. After $\Delta$-Stepping, vertex 3 has vertex 4 as its predecessor. In the second phase, vertex 4 has vertex 3 as its successor. Then, in the third phase, vertex 3’s $\sigma_{03}$ value is computed to be 1, because vertex 4 is its only predecessor, and vertex 4 has a $\sigma_{04}$ value of 1. In the last phase, Edmond et al.’s algorithm accumulates dependency for vertex 3’s predecessors, and vertex 4’s dependency is set to 1. Essentially, Edmond et al.’s algorithm splits Brandes’ algorithm into more phases for better efficiency in a distributed setting.

Edmond et al.’s Algorithm in VC Paradigm. Similar to the case of implementing $\Delta$-Stepping in the vertex-centric paradigm, implementing Edmond et al.’s algorithm in this paradigm also introduces inefficiencies.
First, due to the dependency on the $\Delta$-Stepping algorithm, its inefficiency is inherited.

Second, extra inefficiencies are incurred due to the multi-phase nature of Edmond et al.’s algorithm. As mentioned in Section III-A, algorithms in VC paradigm terminates by calling voteToHalt(). Consider the SSSP phase in Edmond et al.'s algorithm. If we directly copy an implementation of SSSP algorithm into our implementation of Edmond et al.’s algorithm, it would certainly compute the correct result for this phase. However, if we do so, all vertices will call voteToHalt(), and the entire Edmond et al.’s algorithm will terminate prematurely right after SSSP finishes, which is only the first phase. As a result, we have to modify the original SSSP implementation by removing all calls to voteToHalt(), which leads to the same inefficiency issue on memory reads as described in Section III-A. The performance impact in this case is even more severe than $\Delta$-Stepping, because in $\Delta$-Stepping only unprocessed vertices need to stay active, while in Edmond et al.’s algorithm all vertices have to remain active.

Third, another possible inefficiency, which depends on how a particular VC system is implemented, is related to message transfers. For example, in Pregel and Giraph, only a single vertex-compute() function can be defined for an algorithm. This would mean that a single message type is used across different phases of a multi-phase algorithm. In some multi-phase algorithms, this may not be an issue. However, in other multi-phase algorithms, different phases require different message types with different number of fields and different data types. For example, in the case of Edmond et al.’s algorithm, the first phase needs to send two integers. The second and third phases need to send one integer. The last phase needs to send one integer and one float. If we are to encapsulate all possibilities in one message type, we have to use at least two integer fields and one float field. This means that the fields in a message would never be fully utilized in any phase and a great deal of network bandwidth could be wasted by sending unnecessary values. In particular, during the second and third phases, two thirds of the network bandwidth would be essentially used for sending garbage data. If a particular VC implementation supports implementing each phase of a multi-phase algorithm with different vertex-compute() functions, this inefficiency could be avoided.

3) BC Experimental Results: We present our experimental results for the betweenness centrality problem in Figure 5. Due to large size of our graphs, employing every vertex as a source vertex is almost infeasible during BC computation. We, therefore, follow the heuristic technique in [3] and sample uniformly at random, 100 source vertices. In Figure 5(a), one can observe that Edmond et al.’s algorithm in MPI is several orders of magnitude faster than that in Giraph. This is primarily because Edmond et al.’s algorithm executes $\Delta$-Stepping for multiple times, thus the expressibility limitations of the VC paradigm get magnified in BC computation.

In addition, we find in Figure 5(b) that the working vertices percentage is smaller than 20% across all supersteps. Note that we only show the supersteps in the first phase of Edmond et al.’s algorithm (i.e., SSSP computation via $\Delta$-Stepping). Compared to Figure 4 in original SSSP, the working vertices percentage is even smaller here. As explained earlier, for multi-phase Edmond et al.’s algorithm, we have to modify the original SSSP implementation by removing all calls to voteToHalt(), which further demonstrates the expressibility issues with VC paradigm while implementing a multi-phase algorithm.

IV. Conclusion

We investigated expressibility challenges of the VC paradigm. As a distributed framework, while the VC paradigm improves programmability by hiding distribution related details, it suffers from expressibility limitations: often the more efficient distributed algorithm of a graph problem cannot be effectively implemented in the VC paradigm. We empirically demonstrated this with two important classes of algorithms: bucketing-based and multi-phased. Although our experiments are conducted in Giraph, the limitations we discovered apply to all systems in which these two classes of algorithms can be implemented.

We realize that the scope of the current voteToHalt() mechanism is quite limited. It only allows each vertex to be in either active or in inactive state during a (synchronous) superstep, however it does not provision for defining priorities among vertices. In future, it would be interesting to directly incorporate priority-based data structures (e.g., buckets) in each worker node, that permit defining and dynamically updating priorities of vertices, thereby enabling more efficient implementation of bucketing-based algorithms in the VC framework. In contrast to voteToHalt(), one can design and invoke functions such as insertIntoBucket(), deleteFromBucket(), and updateInBucket(). Such bucketing data structures, together with the master node, could identify only the working set of vertices at each superstep, resulting in higher efficiency.

In regards to multi-phase algorithms, we recommend providing more control to the master node. Currently, the master node can only terminate an algorithm, but it cannot re-activate vertices when required. Therefore, vertices which finished their computations in the current phase, still needs to remain active for the next phase of a multi-phase algorithm. Providing more control to the master node, e.g., writing simple logic that can identify phase transitions, and thereby help the master node re-activating all vertices for the next phase, would be an interesting future direction.

We highlight that our suggestions above, while enhances the scheduling capabilities of the VC framework, remains compatible with the existing systems. This is because current algorithms, which rely on voteToHalt(), can be considered special cases of bucketing-based algorithms with only a single bucket. A system which implements these changes could still offer the existing interface as a wrapper of the new design for compatibility reasons.
ACKNOWLEDGEMENT

The research is supported by MOE Tier-1 RG83/16 and NTU M4081678. Any opinions, findings, and conclusions in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES