Efficient Processing of Large Graphs via Input Reduction

Amlan Kusum  Keval Vora  Rajiv Gupta  Iulian Neamtiu

1Department of Computer Science, University of California, Riverside
{akusu001, kvora001, gupta}@cs.ucr.edu
2Department of Computer Science, New Jersey Institute Of Technology
ineamtlu@njit.edu

ABSTRACT
Large-scale parallel graph analytics involves executing iterative algorithms (e.g., PageRank, Shortest Paths, etc.) that are both data- and compute-intensive. In this work we construct faster versions of iterative graph algorithms from their original counterparts using input graph reduction. A large input graph is transformed into a small graph using a sequence of input reduction transformations. Savings in execution time are achieved using our two phased processing model that effectively runs the original iterative algorithm in two phases: first, using the reduced input graph to gain savings in execution time; and second, using the original input graph along with the results from the first phase for computing precise results. We propose several input reduction transformations and identify the structural and non-structural properties that they guarantee, which in turn are used to ensure the correctness of results while using our two phased processing model. We further present a unified input reduction algorithm that efficiently applies a non-interfering sequence of simple local input reduction transformations. Our experiments show that our transformation techniques enable significant reductions in execution time (1.25×-2.14×) while achieving precise final results for most of the algorithms. For cases where precise results cannot be achieved, the relative error remains very small (at most 0.065).

CSC Concepts
• Computing methodologies → Parallel computing methodologies; Parallel programming languages;

Keywords
Graph Processing; Input Reduction; Iterative Algorithms

1. INTRODUCTION
With the proliferation of data, parallel graph analytics has become a difficult task because it involves executing iterative algorithms on very large graphs. This has led researchers to explore acceleration strategies, some of which produce approximate results. There are two main strategies for approximate computing: algorithmic [5, 18, 1] and code-centric [21, 20, 29, 25, 2, 27]. The algorithmic approach is application specific and thus the ideas for one application may not transfer to others. The code-centric approach transforms the application so that at runtime it switches between code versions or skips computations to save time, albeit sacrificing accuracy. However, for applications whose behavior is input sensitive, intelligent skipping is difficult as the program lacks global view of input characteristics.

In this paper we present a general approach for accelerating parallel vertex-centric iterative graph algorithms that repeatedly process large graphs until convergence. Even though these algorithms are parallel, their execution times can be large for real-world inputs. Thus there is a great deal of benefit in approximating them to save processing time. The novel aspect of our two-phased approach is that it is input data-centric. In the first phase, the original (unchanged) iterative algorithm is applied on a smaller graph which is representative of the original large input graph; this step yields savings in execution time. In the second phase, the results from the smaller graph are transferred to the original larger graph and, via application of the original graph algorithm, error reduction is achieved, possibly converging to the final accurate results. The additional time required to process the reduced graph in the first phase ($T_{\text{phase 1}}$) pays off as it is significantly lower than the savings achieved by the second phase ($T_{\text{original}} - T_{\text{phase 2}}$); hence the overall processing time reduces from $T_{\text{original}}$ to $T_{\text{phase 1}} + T_{\text{phase 2}}$.

To reduce the size of input graphs, we propose light-weight vertex-level input reduction transformations whose application is guided by their impact on graph connectivity (i.e., the global structure of the graph). While there exist works [7, 17, 3, 18, 23, 35, 9, 8, 10] that reduce the size of graphs to accelerate processing, they mainly present algorithm-specific reduction techniques and mostly operate of regular meshes. [7, 17] present a multilevel graph partitioning algorithm where first a hierarchy of smaller graphs is created, then the highest level graphs are partitioned and then, these partition results are carefully propagated back down the hierarchy to achieve partitioning of the original graph. They use edge contraction or maximal independent set computation over dual graph which are suitable for relatively regular mesh structures but can be computationally expensive. [3, 9, 8, 10] also partition graphs via recursive edge contraction using maximal independent set computation to generate multipoles. In contrast, we identify light-weight, local and non-interfering transformations which are general (i.e., not algorithm-specific) and are suitable for reducing large irreg-
ular input graphs. Moreover, our reduction strategy is not hierarchical (multi-level) since our transformations are designed from the vertex’s perspective and are applied at most once on each vertex. Other works like [23, 35] are specifically designed for certain problems (e.g., shortest paths) and require path-level or component-level transformations that involve computationally intensive pre-processing. Our transformations are vertex-level, light-weight, and suitable for large irregular graphs.

Upon carefully studying various characteristics of vertex-centric algorithms and properties of input reduction transformations, we show that it is possible to achieve fully accurate results for a subclass of graph algorithms, while remaining algorithms produce approximate solutions. In comparison to algorithmic works our approach is more general and in contrast to code-centric our approach has two advantages:

- **Input Data-centric Approximation**: via graph reduction, we achieve the effect of skipping computations like the code-centric approach. However, since skipping is achieved as a consequence of input graph reduction that is performed as a preprocessing phase, the decision of what to skip is sensitive to the structure of the input graph – graph connectivity guides the application of transformations.

- **Uncompromised Processing Algorithm**: our approach requires no changes to the core graph analysis algorithm. The original algorithm is used until convergence on the reduced graph and then on the full graph for error reduction. With careful choice of input transformations, the algorithm’s capability can remain uncompromised, i.e., upon convergence the error reduction phase can give precise results.

We evaluate our two-phased processing technique in a shared memory environment using Galois [19], a state-of-the-art parallel execution and graph processing framework. Our experiments with six graph algorithms and multiple real-world graphs show that our techniques achieve an average speedup of $1.25\times$ to $2.14\times$ while achieving precise final results for five benchmarks and approximate results for one with very low relative error (at most 0.065).

2. OVERVIEW OF OUR APPROACH

This section provides an overview of our two phased processing model. While graph reduction based processing strategies are used in various works like [7, 17], we focus on iterative general purpose graph algorithms and operate on a single reduced graph along with the original input graph (i.e., there are no multiple levels in the hierarchy). We use the vertex-centric programming model as it is intuitive and commonly used by many graph processing systems like GraphLab [15], GraphX [34], and Galois [19]. We consider directed graphs in our discussion; our approach easily sim-plies to handle undirected graphs.

Given an iterative vertex-centric graph algorithm $iA$ and a large input graph $G$, the accurate results of vertex values $V_0$ can be computed by applying $iA$ to $G$, that is:

$$V_0 = iA(G)$$

To accelerate this computation, we use the following steps:

- **Reduce input $G$ to $G'$**: we transform the large input graph $G$ into a smaller graph $G'$ via multiple applications of an input reduction transformation $T$.

- **Compute results for $G'$**: we apply $iA$ to $G'$ to compute $V_{G'}$. Computing on $V_{G'}$ takes less time than on $V_0$.

- **Obtain results for $G$**: using simple mapping rules $mR$s, we convert the results $V_{G'}$ to $V_G$. Then, via multiple application of update rules in $iA$, we reduce the error in $V_G$ and obtain the result $V_G^2$.

Thus, our approach replaces computation $V_G = iA(G)$ by:

$$V_G = iA(G')$$

where $\Delta$ is a parameter that controls the degree of reduction performed as it represents the number of applications of $T$ to $G$. Thus, the greater the value of $\Delta$, the smaller the size of the reduced graph $G'$. Depending on various properties of input reduction transformations $T$ (Section 3.2) and the nature of iterative algorithm $iA$, the computed values will be accurate, i.e., $V_G^2 = V_G$. However, we identify cases in which $V_G^2$ may not be the same as $V_G$ (Section 4) — the computed results are approximate for those cases.

2.1 Efficient Input Reduction Transformations

Given the iterative nature of algorithms considered, applying $iA$ to $G'$ as opposed to $G$ is expected to result in execution time savings. However, these savings can be offset by the extra overhead due to application of input reduction transformations and result converting rules. Therefore we must ensure that these steps are simpler than the iterative computation that they aim to avoid. We do so by placing the following restrictions on the kind of transformation that is allowed (local) and the sequence of its application (non-interfering) permitted for reducing $G$ to $G'$.

A. Local transformation. Transformation $T(v, G)$, where $v$ is a vertex in $G$, is a local transformation if its application only examines edges directly connected to $v$. The subgraph involving $v$ and its edges is denoted as $\text{subGraph}(T(v, G))$.

$$G_0 \leftarrow T(v_0, G_0); G_1 \leftarrow T(v_1, G_1); \cdots; G_{\Delta-1} \leftarrow T(v_{\Delta-1}, G_{\Delta-2}); G' \leftarrow T(v_{\Delta}, G_{\Delta-1})$$

B. Non-interfering sequence. $T^{\Delta}$, a sequence of $\Delta$ applications of local transformation $T$ as shown above is non-interfering if and only if: vertices $v_1 \cdots v_{\Delta}$ are distinct vertices in $G$; and each subGraph($T(v_i, G)$) is contained in $G$. Note that the above restrictions (local and non-interfering) ensure that input reduction is performed via a single pass over the original graph because:

- An edge $v_i \rightarrow v_j$ from $G$ is only examined when considering the application of $T$ to $v_i$ or $v_j$; and

- Any vertex or edge created during one application of $T$ cannot be involved in any other application of $T$.

Thus, the cost of applying the transformation sequence is linear in the size of $G$, i.e., the number of vertices and edges in it. Moreover, the cost of converting results is proportional to the size of the transformed portions of $G$. In contrast, those computations over the transformed portions of $G$ that we avoid would have required repeated passes due to the iterative nature of graph algorithms considered.

In conclusion, the restrictions on transformations and sequences ensure that the cost of applying them will be less than the cost of the computation they avoid, leading to net savings in execution time.
Algorithm 1 Iterative Vertex-Centric Graph Algorithm.

1: function TP1A ( input $G$ )
2: 
3: $G' \leftarrow$ REDUCEGRAPH $(G, T, \Delta)$
4: $V_0^1 \leftarrow 1A(G')$
5: $V_0^2 \leftarrow 1AP2(V_0^1, G)$
6: return $V_0^2$
7: end function

8: function REDUCEGRAPH $(G, T, \Delta)$
9: 
10: for ( Vertex $v : G$ ) do
11: if ( $|T_v|$ ( subGraph($T(v, G)$) ) then
12: $G' \leftarrow T(v, G')$
13: $\Delta \leftarrow \Delta - 1$
14: if ( $\Delta == 0$ ) then break end if
15: end if
16: end for
17: return $G'$
18: end function

19: function 1A ( input $G$ )
20: Initialize $V_0$ & WorkQ
21: while ( ! WorkQ.empty ) do
22: $v \leftarrow$ WorkQ.getFirst()
23: if ( UPDATEVALS ( $v, V_0$ ) ) then
24: WorkQ.add ( outNeighbors ( $v$ ) )
25: end if
26: end while
27: return $V_0$
28: end function

29: function UPDATEVALS ( $v, V_0$ )
30: Updated $\leftarrow$ false
31: if updateCheck ( $v$, inNeighbors ( $v$ ) ) then
32: update $V_0[v]$
33: Updated $\leftarrow$ true
34: end if
35: return Updated
36: end function

37: function 1AP2 ( $V_0^1, G$ )
38: Initialize WorkQ
39: for ( Vertex $v : G$ ) do
40: if ( $v \in G'$ ) then
41: $V_0^2 ( v ) \leftarrow V_0^1 ( v )$
42: else
43: $V_0^2 ( v ) \leftarrow \text{initial}( )$
44: WorkQ.add ( $v$ )
45: end if
46: end for
47: WorkQ.add ( Vertex $v$ s.t. $v$ is affected by addition / deletion of edges )
48: while ( ! WorkQ.empty ) do
49: $v \leftarrow$ WorkQ.getFirst()
50: if ( UPDATEVALS ( $v, V_0$ ) ) then
51: WorkQ.add ( outNeighbors ( $v$ ) )
52: end if
53: end while
54: return $V_0$
55: end function

Algorithm 2 SSSP Algorithm.

1: function TwoPhaseSSSP ( input $G$; srcVertex )
2: \begin{itemize}
3: \item $V_0$ of a vertex $v$ = length of the
4: \item shortest path from srcVertex to $v$
5: \end{itemize}
6: end function

7: function REDUCEGRAPH $(G, T, \Delta, srcVertex)$
8: \begin{itemize}
9: \item srcVertex is not part of applied $T$'s
10: \end{itemize}
11: end function

12: function INITIALIZESSSP ( input $G$; srcVertex )
13: \begin{itemize}
14: \item $V_0[srcVertex] \leftarrow 0$
15: \end{itemize}
16: end function

17: function UPDATEVALS ( $v, V_0$ )
18: \begin{itemize}
19: \item Updated $\leftarrow$ false
20: \item if ( $v' : inNeighbors ( v )$ ) do
21: \end{itemize}
22: end if
23: return Updated
24: end function

25: function Phase2SSSP ( $V_0^2, G$ )
26: \begin{itemize}
27: \item initial() assigns $\infty$ or results from phase 1
28: \end{itemize}
29: end function

2.2 Original and Two-Phased Algorithms

Next we summarize our approach by presenting the general form of an original iterative vertex-centric graph algorithm and its corresponding two phased version. In Algorithm 1, function 1A represents the original algorithm whose application to graph $G$ produces the accurate ($V_0^2$) result. Function TP1A is the two phased version that calls 1A and 1AP2 in first and second phases. Note that the processing logic in 1AP2 (lines 48-53) is exactly the same as that in 1A (lines 20-25). The result ($V_0^2$) is obtained from the application of TP1A to $G$. The result obtained from TP1A might not be accurate; we discuss this in Sections 4.1 and 4.2.

REDUCEGRAPH examines the vertices in $G$ one at a time and if $T(v, G')$ is non-interfering with transformations already applied, then it is applied on $v$. The function 2I enforces non-interference by ensuring that all vertices and edges in subGraph($T(v, G)$) are being examined for the first time. The algorithm terminates after applying $\Delta$ transformations. The function 1AP2 copies results from vertices in $G'$ to vertices in $G$ for each vertex that is present in both graphs. The vertices in $G$ that were eliminated in the process of creating $G'$ are assigned initial values by initial(). Then, similar to 1A, UPDATEVALS is applied to $V_0^2$ until convergence.

2.3 Example: Single Source Shortest Paths

Algorithm 2 presents the two-phased version of the Single Source Shortest Paths (SSSP) algorithm. Only the code sequences that are specific to SSSP are shown while other code sequences from Algorithm 1 remain the same. The function UPDATEVALS() computes the shortest path for a vertex $v$ based on its incoming edges.
Based on our experiments, we set this threshold to be 1000.

We present six transformations to reduce input graph and accelerate the second phase. 

3. INPUT REDUCTION

We discuss their properties to gain useful programming insights.

3.1 Transformations for Input Reduction

Since many graph algorithms are super-linear in the number of edges, the goal of graph reduction is to reduce the number of edges in the graph. If all edges involving a node are eliminated, then so is the node. Figure 3 shows the transformations. The red dashed edges are the ones that are eliminated by the transformations. Algorithm 3 presents the algorithm which examines every vertex of the input graph \(G\), and considers applicability of transformations.

\(T_1/T_2\). If vertex \(v\) has no incoming/outgoing edges, its outgoing/incoming edges are removed and \(v\) is dropped.

\(T_3\). For every vertex \(v\) with a single incoming and a single outgoing edge, transformation \(T_3\) eliminates \(v\) and adds a direct edge between the other end vertices of \(v\)’s edges. Thus, in a single step we bypass multiple nodes; however, for simplicity we consider bypassing a single node only. Note that \(T_3\) ensures that a path between two vertices \(v\) and \(w\) is preserved even though direct edges or intervening nodes are dropped.

\(T_4\). For a vertex \(v\) with high number of incoming edges,\(^1\)

\(^1\) An indegree threshold can be set while using \(T_4\) and \(T_6\). Based on our experiments, we set this threshold to be 1000.

Figure 1 illustrates graph reduction by converting \(G\) to \(G’\) and Figure 2 illustrates how the two-phased SSSP algorithm works on the example graph by first computing \(V_0\) (Figure 2-a), then feeding these computed results to \(V_1\) (Figure 2-b), and then computing \(V_2\) (Figure 2-c). In this case a single application of UpdateVals in the second phase yields precise results (i.e., \(V_2 = V_0\)). In general, for large complex graphs and different applications, this may not be the case; however, the results computed in the first phase will accelerate the second phase.

3. INPUT REDUCTION

We present six transformations to reduce input graph and discuss their properties to gain useful programming insights.

3.1 Transformations for Input Reduction

Since many graph algorithms are super-linear in the number of edges, the goal of graph reduction is to reduce the number of edges in the graph. If all edges involving a node are eliminated, then so is the node. Figure 3 shows the transformations. The red dashed edges are the ones that are eliminated by the transformations. Algorithm 3 presents the algorithm which examines every vertex of the input graph \(G\), and considers applicability of transformations.

\(T_1/T_2\). If vertex \(v\) has no incoming/outgoing edges, its outgoing/incoming edges are removed and \(v\) is dropped.

\(T_3\). For every vertex \(v\) with a single incoming and a single outgoing edge, transformation \(T_3\) eliminates \(v\) and adds a direct edge between the other end vertices of \(v\)’s edges. Thus, in a single step we bypass multiple nodes; however, for simplicity we consider bypassing a single node only. Note that \(T_3\) ensures that a path between two vertices \(v\) and \(w\) is preserved even though direct edges or intervening nodes are dropped.

\(T_4\). For a vertex \(v\) with high number of incoming edges,\(^1\)

\(^1\) An indegree threshold can be set while using \(T_4\) and \(T_6\). Based on our experiments, we set this threshold to be 1000.
Algorithm 3 Graph Reduction Algorithm.

1. Algorithm TRANSFORM (G(V, E))
2. \( E' \leftarrow E \)
3. for \( \forall v \in V \) do
4.   if (inDegree(v) = 0) then
5.     \( E' \leftarrow E' \setminus \text{outEdges}(v) \)
6. else
7.     \( E' \leftarrow E' \setminus \text{outEdges}(v) \)
8.   end if
9. end for
10. if (outDegree(v) = 0) then
11.   \( E' \leftarrow E' \setminus \text{inEdges}(v) \)
12. end if
13. if (inDegree(v) = outDegree(v) = 1) then
14.   \( \text{apply } T_2 : \text{drop } v \rightarrow v \)
15. end if
16. end algorithm

Algorithm COALESCE (G(V, E), E', v)
17. for \( \forall (w \rightarrow v) \in \text{inEdges}(v) \) do
18.   \( E' \leftarrow E' \setminus \{w \rightarrow v\} \)
19. end for
20. for \( \forall (u \rightarrow w) \in \text{inEdges}(w) \) do
21.   \( E' \leftarrow E' \setminus \{u \rightarrow w\} \)
22. end for
23. for \( \forall (w \rightarrow u) \in \text{outEdges}(w) \) do
24.   \( E' \leftarrow E' \setminus \{w \rightarrow u\} \)
25. end for
26. for \( \forall (w \rightarrow u) \in \text{outEdges}(w) \) do
27.   \( E' \leftarrow E' \setminus \{w \rightarrow u\} \)
28. end for
29. return \( E' \) of \( G' \)
30. end algorithm

between the transformed graph and the original graph. In particular, \( T_5 \) can effectively increase the diameter of the input graph by spreading out vertices which are close to each other in the original graph, far apart in the transformed graph and hence, increasing the traversal cost. \( T_6 \), on the other hand, randomly drops edges from high-degree vertices which are typically important locations defining the graph structure. Care must be taken while reducing the graph using these transformations since the computed values from first phase using structurally dissimilar graphs can prove to be useless and hence, demand significant computation on the original graph in the second phase. Algorithm 3 achieves our objective of applying a non-interfering sequence of transformations as it efficiently applies the transformations by making a pass over the vertices in the graph. Since this is a conservative approach, we can run the algorithm to completion applying as many transformations from \( T_1 \) to \( T_4 \) as possible in one pass (lines 3-18). If the expected reduction is not achieved, we use the aggressive transformations for further reduction (lines 19-30).

3.2 Transformation Properties

We consider each transformation and deduce strong guarantees about various properties of the transformed graph \( G' \) compared to that of the original graph \( G \). These guarantees are categorized into two types: a) Structural Guarantees that determine a relation of structural properties, i.e., edges, vertices and components; and b) Non-Structural Guarantees that determine a relation of edge-weights.

Structural Guarantees. Consider six transformational properties that determine the relation of structural properties of \( G' \) with \( G \) when transformation \( T_k \) (1 \( \leq k \leq 6 \)) is applied.

<table>
<thead>
<tr>
<th>Trans.</th>
<th>[V-ADD]</th>
<th>[V-SUB]</th>
<th>[E-ADD]</th>
<th>[E-SUB]</th>
<th>[C-MERGE]</th>
<th>[C-SPLIT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>( T_4 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>( T_5 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>( T_6 )</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
</tbody>
</table>

Table 1: Structural guarantees for each transformation. ✓ and ? indicate occurrence and non-occurrence of the corresponding property respectively, whereas ? indicates that the corresponding property may or may not occur.

It is easy to follow that \( T_1 \) and \( T_2 \) guarantee occurrence of \( [V-SUB] \), \( [E-SUB] \) and non-occurrence of \( [V-ADD] \), \( [E-ADD] \), \( [C-MERGE] \). Also, \( [C-SPLIT] \) can occur when these two transformations are applied. Transformations \( T_3 \) and \( T_4 \) guarantee occurrence of \( [E-SUB] \) and non-occurrence of \( [V-ADD] \), \( [E-ADD] \), \( [C-MERGE] \). \( T_4 \) also guarantees occurrence of \( [E-ADD] \) and \( [V-SUB] \), whereas \( T_5 \) also guarantees non-occurrence of \( [E-ADD] \) and \( [V-SUB] \). Transformation \( T_6 \) guarantees occurrence of \( [V-SUB] \), \( [E-ADD] \), \( [E-SUB] \) and non-occurrence of \( [V-ADD] \), \( [C-MERGE] \), \( [C-SPLIT] \). Finally, \( T_6 \) guarantees occurrence of \( [E-SUB] \) and non-occurrence of \( [V-ADD] \), \( [V-SUB] \), \( [E-ADD] \), \( [C-MERGE] \). While dropping edges using \( T_6 \), \( [C-SPLIT] \) can occur.

Table 1 overviews all structural properties guaranteed by each of the transformations. Note that all transformations guarantee non-occurrence of \( [V-ADD] \) and occurrence of \( [E-SUB] \) which result in reduction of transformed graph sizes.
Non-Structural Guarantees. Since transformations $T_3$ and $T_4$ guarantee occurrence of $[E\text{-ADD}]$, correct edge weights need to be assigned to newly added edges for weighted graphs.

We define two transformational properties which determine the relation of edge weights of $G'$ with that of $G$ when transformation $T_k$ $(1 \leq k \leq 6)$ is applied. In the following expressions, $a \iff b$ means $b \in G'$ is resulted from $a \in G$.

- $[E\text{-EQUAL}]$: $T_k$ results in edges $e_1$ and $e_2$, both with weights $w(e_1), w(e_2)$ respectively s.t. $w(e_1) = w(e_2) \iff e_1 \leftrightarrow e_2$.

- $[E\text{-FUNC}]$: $T_k$ results in edges $e_1, e_2$ and $e_3$, with weights $w(e_1), w(e_2)$ and $w(e_3)$ respectively s.t. $w(e_3) = func(w(e_1), w(e_2)) \iff e_3$.

$[E\text{-FUNC}]$ represents the weight of the newly added edge as a function of weights of edges from the original graph that resulted in this new edge. For example, the new weight can be set as the sum, minimum, or maximum of the original edge weights ($[E\text{-SUM}],[E\text{-MIN}],[E\text{-MAX}]$ respectively).

Transformation $T_3$ guarantees occurrence of $[E\text{-FUNC}]$ and non-occurrence of $[E\text{-EQUAL}]$. For transformation $T_2$, both $[E\text{-EQUAL}]$ and $[E\text{-FUNC}]$ can occur. As we will see in Section 4.1, we use $[E\text{-SUM}]$ to benefit the exploratory and traversal based graph algorithms.

4. PROGRAMMING FOR TRANSFORMED GRAPHS

Using the transformation properties described in Section 3.2, we discuss properties of vertex-centric graph algorithms that permit them to benefit from the two-phased model.

4.1 Impact of Transformations on Vertex Functions

Since the aforementioned transformations change the structural and non-structural properties of the graph, it is important to determine the impact of these changes on how programmers should correctly express graph algorithms. Even though custom algorithms can be written so that computations performed on transformed graphs always lead to correct values, we eliminate this programming overhead by supporting the popular vertex centric programming model for our two-phased processing model.

Vertex-centric programming. In this model, algorithms are expressed in a vertex-centric manner, i.e., computations are written from the perspective of a single vertex. These computations, called vertex functions, are iteratively executed on all vertices in parallel, until all the vertex values in the graph stabilize. Vertex functions typically use the values coming from its incoming edges as inputs for computation. Hence, the newly computed value of a vertex depends on the values coming from its incoming edges. Moreover, the asynchronous nature of the graph algorithms requires computations over updates coming from incoming edges to be commutative and associative — this way, updates coming from different incoming edges can be processed in any order, e.g., the order of their arrival.

To guarantee correct answers at the end of computation, we need to reason about the behavior of vertex functions, first when applied on the transformed graph $G'$, and later on the original graph $G$. For illustration, we use two versions of the SSSP vertex functions, SSSP-IN and SSSP-SIN, shown in Algorithm 4. Computations in SSSP-IN only depend on values coming from incoming neighbors, whereas those in SSSP-SIN depends on the previous value of the vertex in addition to the values coming from neighbors. The only difference between SSSP-IN and SSSP-SIN is the initialization of $\text{minPath}$ (line 3 and 14 marked in red); the rest of the functions are identical. Note that both of these variants produce correct results when used in the traditional vertex centric processing model. However, they behave differently when used in our two-phased processing model, in which only SSSP-IN leads to accurate results.

Let us evaluate each of the structural and non-structural properties which are affected by our transformations.

(A) $[V\text{-SUB}]$ and $[E\text{-SUB}]$: $[E\text{-SUB}]$ leads to computations being performed even when all the incoming edges of a vertex are not available. Such computations are equivalent to that in the staleness-based (i.e., relaxed consistency) computation model [31] where the edges can potentially contain stale values; in this case, missing edges can be viewed as edges with no new contribution. The same argument also holds true for $[V\text{-SUB}]$ since the effect of vertex deletion is viewed as edge deletion by its neighbors, reducing to $[E\text{-SUB}]$. In both of these cases, SSSP-IN and SSSP-SIN produce an over-approximation of path distance when applied on $G'$, compared to the precise distance computed on $G$, i.e., $\text{minPath}(G') \geq \text{minPath}(G)$. In the second phase when missing vertices and edges become available in $G$, this approximation automatically gets corrected.

(B) $[E\text{-ADD}]$, $[E\text{-EQUAL}]$, and $[E\text{-FUNC}]$: Transformations resulting in $[E\text{-ADD}]$ are introduced in order to preserve the connectivity in the graph which is essential for various traversal-based graph algorithms. Moreover, both $[E\text{-EQUAL}]$ and $[E\text{-SUM}]$ attempt to create edge-weights of newly added edges to represent an approximation of the distance between corresponding vertices in the original graph. This allows traversal algorithms to proceed with computations based on those newly added edges since the results for transformed graphs are close to the results for the original graph, and hence can accelerate processing over the orig-
Table 2: Various vertex-centric graph algorithms. SSSP, SSWP, CC, PR, and GC produce 100% accurate results.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Vertex Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP</td>
<td>[v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.path + e.wt)]</td>
</tr>
<tr>
<td>SSWP</td>
<td>[v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.wt))]</td>
</tr>
<tr>
<td>CC</td>
<td>[v.component \leftarrow \min_{e \in \text{edges}(v)} (e.other.component)]</td>
</tr>
<tr>
<td>PR</td>
<td>[v.rank \leftarrow 0.15 + 0.85 \times \sum_{e \in \text{inEdges}(v)} e.source.rank]</td>
</tr>
</tbody>
</table>

4.2 Graph Algorithms

We now discuss how each of the graph algorithms used in this work will perform using our technique. Table 2 shows details about each of the seven vertex functions considered in this work. We will argue that PR, SSSP, SSWP, GC, and CC produce 100% accurate results whereas the same accuracy cannot be ensured by CD.

(A) Shortest & Widest Paths: As discussed in Section 4.1, when shortest path (SSSP) is computed on \(G'\), the transformations lead to an approximate solution which gets corrected in the second phase of processing when using SSSP-IN. For the widest path (SSWP), recall that [E-SUM] is a specialization of [E-FUNC] which can support a wide range of such traversal based algorithms. Hence, SSSP can be supported by ensuring that the weight of any newly added edge is the minimum of the edges whose removal caused the addition of this new edge ([E-MIN]). In this case, [E-MIN] ensures that the computed path width in \(G'\) is always at most that of the equivalent path in \(G\).

(B) Connected Components: Since the main idea behind CC is that vertex values within a component are the same and those in different components are different, we determine its correctness using [C-MERGE] and [C-SPLIT] properties. All the transformations guarantee non-occurrence of [C-MERGE]; hence, values flowing in different components of the original graph will always be different in the transformed graph. When [C-SPLIT] occurs, vertices within the same component of the original graph can now belong to different components of the transformed graph, leading to different values flowing in the same original component. This approximation gets corrected when these vertices are re-grouped together into the same component in the second phase; the computation simply picks one of the vertex values to flow across the entire component.

(C) Graph Coloring: The underlying idea behind GC is to assign different colors to the end vertices of every edge while using minimal \(^3\) set of colors to color all vertices. Hence, we determine its correctness using [E-ADD] and [E-SUB] properties. When [E-ADD] occurs, an edge connects two vertices in \(G'\), which were disconnected in \(G\). Even though this causes the two vertices to be assigned different colors, it does not violate the correctness of the solution: when the edge is removed in the second phase, the color assignment for one of these two vertices gets updated and is propagated throughout the graph. When [E-SUB] occurs, vertices which are connected by an edge in \(G\) become disconnected. This can cause the vertices to be assigned different colors.

\(^3\)Graph coloring is NP-complete and hence the constraint is usually relaxed to minimal colors which can be solved in polynomial time.
the same color when processing on \( G' \). However, during the second phase, these edges become available in \( G \) which re-processes the vertices and hence, the self-correcting nature of the algorithm detects and corrects the coloring inconsistency. This in turn ensures that different colors are assigned to connected vertices. Note that different executions of the same original graph coloring algorithm on the same graph can result in different color assignments and minimal number of colors, i.e., the set of correct solutions is not a singleton and hence, the solution computed by our two-phased approach is one of the solutions in the correct set because it adheres to the two constraints of the problem.

(D) PageRank: As shown in [6], PR converges to the correct solution regardless of the initial vertex values. With different initializations, the path to convergence changes. Since computations over \( G' \) provide an approximation of the final results, these results, when fed as initialization values for \( G \), cause the second phase to converge faster.

(E) Community Detection: CD detects communities in the graph by propagating labels that are most frequent among the immediate neighborhood of the vertices. Both \([E-SUB]\) and \([E-ADD]\) influence this computation since the frequency of labels get affected by edge addition/deletion, which leads to an approximation at the end of first phase. During the second phase when \( G \) becomes available, this approximation may not be fully corrected because individual corrections due to availability of original edges might not affect the highly approximate frequency calculated in previous iterations. This can lead to results which are not accurate.

Early Termination in First Phase. A key advantage of our approach is that none of the algorithms require processing over \( G' \) to converge to its final solution before moving on to \( G \). This is because the intermediate values produced while processing \( G' \) also represent a valid approximation of the final solution. Hence, to speed up the computation even further, we can employ early termination of first phase, where the computation does not wait to reach to its converged solution, and the available computed values are directly used in the second phase to process the original graph.

5. ANALYSIS & GENERALITY

We first theoretically analyze the performance benefits that can be achieved by our two-phased model and then discuss the generality of our approach to achieve similar benefits in different scenarios.

5.1 Analysis

Let \( P_G \) and \( P_T \) be the average execution times of a single iteration over \( G \) (original graph) and \( G_T \) (reduced graph) respectively. Further, let \( P_G^T \) be the average execution time of a single iteration over \( G \) in the second phase using computed results fed from \( G_T \) to \( G \). Note that \( P_G^T < P_G \). Moreover, since \(|G_T| < |G|\), i.e., \( G_T \) has fewer edges than \( G \), we know that \( P_T < P_G \). In order to accelerate processing using the two-phased approach, we require:

\[
I_1.P_T + I_2.P_G^T < I.P_G
\]

(1)

where \( I_1 \), \( I_2 \), and \( I \) are the number of iterations in which \( G_T \) is processed in the first phase, \( G \) is processed in the second phase when computed results are fed from \( G_T \), and \( G \) is processed in the original processing model, respectively. Upon rearranging Eq. 1 we get:

\[
I_1.P_T < I.P_G - I_2.P_G^T
\]

(2)

which conveys that in order to achieve benefits from our technique, the savings from the second phase \((I.P_G - I_2.P_G^T)\) should be larger than the time spent in the first phase \((I_1.P_T)\).

For example, if we want to accelerate the overall processing by 25%, we should have:

\[
I_1.P_T + \frac{1}{4} I.P_G = I.P_G - I_2.P_G^T
\]

\[
\implies I_1.P_T = \frac{3}{4} I.P_G - I_2.P_G^T
\]

\[
\implies I_1.P_T < \frac{3}{4} I.P_G \quad \sim \quad |G_T| < \frac{3}{4}|G|
\]

(3)

The above implication from processing times to graph sizes \(|G| \) and \(|G_T|\) is an approximation that holds true as \( G_T \) is created primarily by dropping vertices and edges from \( G \) and hence, \( I_1.P_T \) reduces proportionately compared to \( I.P_G \).

Eq. 3 shows that if we want to accelerate the overall processing by 25% using our two phased processing technique, we must ensure that the reduced graph is reduced to at least 75% of the original graph. As we will see in our evaluation (Section 6), reducing the original graph by a quarter to a half of its original size practically allows up to 32% savings in execution times.

5.2 Generality

From the above analysis, it can be clearly seen that the savings in the overall processing times are largely dependent on \(|G|\) and \(|G_T|\), i.e., size of original and transformed graphs. This allows us to argue that the our technique is independent of the underlying processing environments, iterative algorithms, and input graphs.

Processing Environments. Processing large graphs in different environments incurs different overheads and since our technique eliminates significant amount of processing on the entire large graph, it can help alleviate some of these overheads. For example, processing large graphs on GPUs would require frequent transfer of subgraph information and computed values between host-memory and device-memory which is a significant overhead [11, 26]. Since our transformed graph is much smaller, bulk of this transfer gets eliminated in the first phase and is only performed for remaining few iterations in the second phase. Moreover, if the transformed graph fully fits in the GPU memory, absolutely no transfers are required in the first phase.

In a distributed processing environment, the overall performance is largely dependent on the communication of vertex updates between nodes [31]. Again, using our technique, much of the communication can be avoided in the first phase, hence reducing the overall communication overheads. Moreover, the transformed graph in the first phase can be processed on the subset of nodes in the cluster to reduce synchronization and communication overheads.

The applicability is similar in an out-of-core processing environment where the graph is resident on secondary storage [13, 22]. The first phase eliminates costly disk read and writes while reducing them in the second phase due to reduction in number of iterations.

Iterative Algorithms. The two-phased processing is suitable for iterative graph algorithms whose convergence is dependent on the values being computed. As shown in Section 6, the performance benefits are noteworthy for different
kinds of graph algorithms: on one hand, traversal algorithms like SSSP/SSWP which require lesser computation and on other hand, algorithms like PR/GC/CD which require more computation to compute final solution. Also, the benefits achieved are higher for asynchronous graph algorithms [31] because correctness guarantees are stronger for those cases. Again, as deduced in the above analysis, the performance benefits of our technique are mainly due to reduction in the data-size that needs to be processed and is independent of the kind of processing being performed on the data.

Input Graphs. The proposed reduction and processing techniques are best suited for irregular graphs where the degree distribution across vertices is spread across a wider range, allowing various pre-conditions for our transformations to be satisfied 4. As long as the input graph is large enough that reduction in its size achieves perceivable reduction in processing time, the two-phased processing model can be used to accelerate processing. As shown in Table 4, we use large real-world input graphs which are highly irregular and sparse for our evaluation on which our technique achieves reasonable benefits. Moreover, our transformations \( T_0 \) and \( T_e \) are tunable so that they can be applied even to a graph on which no other transformations can be applied.

6. EVALUATION

We thoroughly evaluate our two-phased processing technique to show that our approach is efficient (savings in execution time), scalable (higher savings in execution with higher number of threads) and produces accurate results for most of the graph applications with low time overhead.

Benchmarks, Inputs and System. We consider six popular vertex centric graph algorithms, as shown in Table 2 and Table 3. We implemented the baseline and the two-phased version of each of the benchmarks in Galois [19], a state-of-the-art parallel execution framework.

Table 4 shows the details of the input graphs, their reduced versions and time taken for reduction. We use 4 input graphs, 3 of which are real-world graphs (Friendster, Twitter and UKDomain) from publicly available Konekt repository [12]. The synthetic graph (RMAT-24) is a scalefree graph \((a = 0.5, b = c = 0.1, d = 0.3)\) similar to the one used in [19]. To transform these graphs, we define a tunable parameter Edge Reduction Percentage (ERP) as:

\[
ERP = \frac{|E_G|}{|E_G'|} \times 100
\]

where \( |E_G| \) and \( |E_G'| \) are the number of edges in original graph \( G \) and the reduced graph \( G' \). We generate the reduced graphs with varying ERP (75%, 70%, 60%, 50%, 40% and 30%) using our transformation tool based on Algorithm 3.

Experiments were performed on a machine with 4 six-core AMD™ 8431 processors (total 24 cores) and 32 GB RAM running Ubuntu 14.04.1 (kernel version 3.19.0-28-generic). The programs were compiled using GCC 4.8.4, optimization level -O3.

We evaluate the performance of following versions of the benchmark implementations:

- **Baseline**: based on the traditional processing model.
- **TP-X**: based on our two-phased processing model using reduced graphs with ERP = X%. Note that the execution times include the graph reduction times which are already presented in Table 4.

Unless otherwise specified, the benchmarks were run with 20 software threads.

Efficiency of Two-Phased Processing. Figure 4 show the speedups achieved by TP-X over Baseline for \( X \in \{30\%, 40\%, 50\%, 60\%, 70\%, 75\%\} \). As we can see, the speedups increase as ERP decreases from 75% to 40%; on an average, TP-75, TP-70, TP-60, TP-50 and TP-40 achieve a speedup of \(1.23 \times, 1.27 \times, 1.41 \times, 1.51 \times \) and \(1.53 \times \) respectively. This is because of the high savings achieved in the second phase while processing the original graphs. On an average for TP-75, TP-70, TP-60, TP-50, and TP-40, the savings achieved in the second phase are 84.88%, 83.08%, 80.65%, 77.99% and 73.71% respectively. These high savings allow tolerating the execution times of reduction and first phase over reduced graphs; the execution times normalized w.r.t. Baseline for the first phase of TP-75, TP-70, TP-60, TP-50, and TP-40 are 0.66, 0.61, 0.50, 0.42, and 0.36 respectively and for the reduction are as low as 0.01, 0.02, 0.03, 0.03, and 0.04 respectively. Since our reduction transformations are local and non-interfering, the cost of performing the input reduction is much lower than the savings achieved in processing.

As expected, the time taken to process the reduced graph in the first phase decreases as ERP decreases simply because the work done is typically proportional to the size of graph. On the other hand, the execution time in second phase increases as ERP decreases. This is mainly because an aggressively reduced graph with lower ERP is structurally less similar to the original graph compared to that reduced with a higher ERP. Hence, the values which are fed from reduced graph with lower ERP require more computation in the second phase in order to reach to maximum possible accuracy for the original graphs.

The savings achieved by our two-phased processing model increases as ERP decreases up to a certain limit. Across each of our benchmark-input-ERP combination, the maximum savings are observed for ERP around 40-50%. However, note that further decreasing ERP reduces the amount of savings achieved; with ERP \(\approx 30\%\) the performance degrades and the average speedup drops to \(1.41 \times\). This is because the reduced graph with very low ERP becomes too small (i.e.,

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Source Shortest Path (SSSP)</td>
<td></td>
</tr>
<tr>
<td>Single Source Widest Path (SSWP)</td>
<td></td>
</tr>
<tr>
<td>PageRank (PR)</td>
<td>Accurate</td>
</tr>
<tr>
<td>Graph Coloring (GC)</td>
<td></td>
</tr>
<tr>
<td>Connected Components (CC)</td>
<td></td>
</tr>
<tr>
<td>Community Detection (CD)</td>
<td>Approximate</td>
</tr>
</tbody>
</table>

Table 3: Graph Algorithms.

<table>
<thead>
<tr>
<th>Input Graph</th>
<th>Graph Size</th>
<th>Reduction Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friends (FT)</td>
<td>Original</td>
<td>68.3M</td>
</tr>
<tr>
<td></td>
<td>Reduced</td>
<td>41.9-51.8M</td>
</tr>
<tr>
<td>Twitter (TT)</td>
<td>Original</td>
<td>41.7M</td>
</tr>
<tr>
<td></td>
<td>Reduced</td>
<td>23.4-30.8M</td>
</tr>
<tr>
<td>UKDomain (UK)</td>
<td>Original</td>
<td>49.9M</td>
</tr>
<tr>
<td></td>
<td>Reduced</td>
<td>27.6-32.1M</td>
</tr>
<tr>
<td>RMAT-24 (RM)</td>
<td>Original</td>
<td>17M</td>
</tr>
<tr>
<td></td>
<td>Reduced</td>
<td>11.6-13.5M</td>
</tr>
</tbody>
</table>

Table 4: Input Graphs.
them to be executed at vertex-level in parallel.

The transformations to be local and non-interfering allowing

shows that the reduction algorithm is scalable w.r.t. num-

faster than for ERP=30% mainly because there are fewer

Scalability of Input Reduction. We study the scalability

of our input reduction algorithm while 1) varying ERP

from 30% to 75% with 20 threads; and, 2) varying number

of threads from 1 to 20 for TP-50\(^5\). As we can see in Figure 5

(left), with increase in ERP the reduction algorithm runs

faster than for ERP=30% mainly because there are fewer

edges to be removed for higher ERP, and hence, the reduc-

tion algorithm only needs to traverse certain percentage of

the graph to achieve the expected ERP. Moreover, Figure 5

shows that the reduction algorithm is scalable w.r.t. num-

ber of threads; this naturally follows from the requirement of

the transformations to be local and non-interfering allowing

them to be executed at vertex-level in parallel.

Scalability of Two-Phased Processing. As shown in

[19], the Baseline system scales well with increase in number

of threads. To show the impact of our approach, Figure 6

shows the improvement in scalability achieved by TP-50\(^5\)

over Baseline while varying the number of threads from 1 to

20. Note that in Figure 6 the Baseline is also parallel, i.e., a

data-point with \(t\) threads represents improvement achieved

by our technique using \(t\) threads compared to baseline using

\(t\) threads. As we can see in most cases, the improvements

\(^5\) Since ERP = 50 performs best across most cases in our

previous experiments, we only consider TP-50 to save space.
slowly increase as number of threads increase and the maximum improvements are achieved with 20 threads. We believe this is because the reduced graphs become denser compared to the original graphs and hence, the probability of the same vertex to be scheduled multiple times by different threads increases rapidly in TP-50 with increase in threads compared to that in Baseline. This in turn allows more merging of such multiple schedule requests of same vertices to single vertex computations. Moreover, the second phase mostly performs value corrections, hence, less contention is expected since probability of all neighboring vertices to be scheduled simultaneously is greatly reduced.

**Memory Overhead.** While the two phases can be processed separately, feeding values from the first phase to the next can incur expensive reads and writes which can offset the performance benefits achieved by our technique. Hence, it is crucial to maintain the reduced and the original graph in memory and eliminate the explicit intermediate feeding by incorporating a unified graph which leverages the high structural overlap across the two graphs. Figure 7 shows the increase in memory when using a unified graph. On an average, the memory consumption increases by 1.25×; it goes higher for TT (1.34×-1.48×) mainly because the percentage of newly added edges in the transformed graphs is much higher (25%-40%) for TT compared to other graphs (2.7%-23%).

It is interesting to note that the overhead increases as ERP decreases. This is due to the impact of increase in the structural dissimilarity between the original and transformed graphs that requires representing the dissimilar components (i.e., newly added edges) separately for both graphs. Note that these overheads are tolerable compared to those incurred by representing both the graphs separately in memory which can be as high as 1.75×.

**Relative Error for CD.** As discussed in Section 4.2, the accuracy of results for CD could not be guaranteed. In order to determine how good the calculated results are, we define relative error as the ratio of vertices whose computed community values are different compared to the ideal results. Table 5 shows the relative error for CD across all input-ERP combinations. As we can see, the relative error is very small; the average relative error across all cases is 0.02 and the maximum relative error is only 0.065. In fact, the relative error for FT across ERP-60, ERP-70 and ERP-75 is very low (<1E-5). It is interesting to note that the error values decrease as ERP increases. This is mainly because with fewer reduction transformations being applied for higher values of ERP, the probability of merging communities in reduced graphs decreases.

<table>
<thead>
<tr>
<th>Input</th>
<th>TP-30</th>
<th>TP-40</th>
<th>TP-50</th>
<th>TP-60</th>
<th>TP-70</th>
<th>TP-75</th>
</tr>
</thead>
<tbody>
<tr>
<td>FT</td>
<td>0.017</td>
<td>0.002</td>
<td>0.001</td>
<td>&lt;1E-5</td>
<td>&lt;1E-5</td>
<td>&lt;1E-5</td>
</tr>
<tr>
<td>TT</td>
<td>0.049</td>
<td>0.041</td>
<td>0.036</td>
<td>0.021</td>
<td>0.019</td>
<td>0.017</td>
</tr>
<tr>
<td>UK</td>
<td>0.065</td>
<td>0.023</td>
<td>0.017</td>
<td>0.013</td>
<td>0.012</td>
<td>0.011</td>
</tr>
<tr>
<td>RM</td>
<td>0.034</td>
<td>0.034</td>
<td>0.021</td>
<td>0.018</td>
<td>0.012</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 5: Relative Error for CD.

We further study how the relative error changes during execution by plotting it for TP-40 in Figure 8. The vertical dotted lines indicate different phases of execution; the first line (close to 0) indicates end of reduction process and the second line (in the middle) indicates the end of the first phase and the beginning of the second phase. As we can see, the relative error remains high during the first phase mainly because of vertices which are missing in the reduced graph. However, the relative error drops rapidly during the second phase due to availability of missing vertices and edges in the original graph. At the end of the first phase, the relative error for FT, TT, UK and RM remain at 0.014, 0.084, 0.22 and 0.12 respectively.

**Contribution of Individual Transformations.** Finally, we evaluate the effect of applying individual transformations one after the other on the overall performance. We define a transformation set $T_{1-k}$ as the set of transformations starting from $T_1$ up to $T_k$. Hence, the transformation set $T_{1-4}$ includes $T_1$, $T_2$, $T_3$ and $T_4$ whereas $T_{1-1}$ only includes $T_1$.

The reduced graphs for this set of experiments are generated using different transformation sets $T_{1-k}$ ($1 \leq k \leq 4$). To clearly present the impact of transformations on both, the size of reduced graphs and the savings in execution time, we select ERP = 50% and only consider the SSSP benchmark. Figure 9 shows the speedups achieved for each of the graphs transformed using the transformation sets, compared to the Baseline. Since the transformations being applied have their pre-conditions which need to be satisfied, the actual ERP using a smaller transformation set can be higher than the requested ERP of 50%. Hence, we also present the actual ERP obtained using the transformation sets.

As we can see in Figure 9, $T_1$, $T_2$ and $T_3$ collectively reduce only small portion of TT, UK and RM graphs; $T_{1-3}$ achieves 95.26%, 96.58% and 99.39% ERP for TT, UK and RM respectively. Due to this, little to no savings are achieved until $T_4$ is included in the transformation sets for which speedups of up to 1.34-1.55× are achieved. FT graph, on the other hand, is amenable to $T_2$ and $T_3$, allowing 50% ERP to be achieved for $T_{1-2}$ and $T_{1-3}$ too. Hence, the speedups achieved for those transformation sets are ~2.15×.
processing techniques into two categories, discussed below. None of these techniques provide correctness guarantees and hence the results of these techniques are always approximate.

Algorithm-specific approximation. Chazelle et al. proposed a technique for approximating the weight of minimum spanning tree in sublinear time by approximating the number of connected components [5]. The technique approximates the weight of the minimum spanning tree but it does not find the tree. Nanongkai [18] proposed an approximation technique to find SSSP and all-pair shortest path (APSP) by bounding the diameter of the graph. Bader et al. [1] proposed the approximation of Betweenness Centrality (BC) by employing an adaptive sampling technique. The algorithm samples a subset of vertices and performs SSSP on them selected, thus reducing the number of SSSP operations to determine BC. In contrast to sampling, our approach to input graph reduction is smarter as it considers graph connectivity and is more general as it is applied to a class of graph algorithms. In fact all of the above approximation techniques were developed for a single specific graph application. In contrast, our technique applies to many iterative graph algorithms all using the same input reduction transformations.

Compiler-based approximation. Researchers have focused on trading accuracy for execution time by skipping a task’s execution or by choosing a specific implementation from multiple ones provided by the developer. Rinard proposed early termination [21] and task skipping [20] that are applied during execution. These techniques use a distortion model based on sampling to estimate the error introduced due to early termination or task skipping. The work does not provide an empirical justification for the distortion model and thus it is unclear if it will work for input graphs with different characteristics. Green [2] selects a specific implementation out of many different implementations provided by the developer while maintaining the quality of output. Hoffman et al. [29] proposed loop perforation where certain iterations of a loop are skipped to trade off accuracy for faster execution. The loops that are perforated are chosen with the help of training input and the error bound set by the user. This technique is not useful for different graph applications since it requires perforated loops to fall into one of the specified categories of the global patterns. Our technique does not require loops to follow any such pattern and it does not perform any static or dynamic analysis of the application to achieve approximation.

The Sage [25] compiler generates CUDA kernels that exploit GPUs to achieve approximation using different optimizations. The runtime system includes a tuning phase which selects the best optimization technique and a calibration phase to help maintain quality. Although we tested our methodology only for CPU systems, it can be easily applied for GPUs as we achieve approximation by reducing the input graph. Shang et al. [27] proposed auto-approximation of vertex-centric graph applications by automatically synthesizing the approximate version of an application. They combined different approximation techniques such as task skipping, sampling, memorization, interpolation and system function replacement for synthesizing the approximate version. Carbin et al. [4] proposed a language to specify approximate program transformations. Our approach works without modifying the original implementation. Moreover input reductions, guided by impact on graph connectivity, customize the skipped computations to input characteristics.
8. CONCLUSION
We proposed input reduction transformations and faster iterative graph algorithms that run in two phases: first, using the reduced input graph, and second using the original graph along with the results from first phase. We evaluated our two-phased model using Galois; our experiments with multiple algorithms and large graphs show that our technique reduces execution time by $1.25\times$ to $2.14\times$.

Acknowledgments
The authors thank their shepherd, Shuaiwen Song, for his guidance in preparing the final version of this paper, as well as the anonymous reviewers for their feedback and suggestions. This work is supported by NSF grants CCF-1524852 and CCF-1318103 to the University of California Riverside.

9. REFERENCES