Efficient Processing of Large Graphs via Input Reduction

[1]Amlan Kusum [1]Keval Vora [1]Rajiv Gupta [2]Iulian Neamtiu

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

CCS 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_{phase1}) pays off as it is significantly lower than the savings achieved by the second phase ($T_{original} - T_{phase2}$); hence the overall processing time reduces from $T_{original}$ to $T_{phase1} + T_{phase2}$.

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 multinodes. 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-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

© 2016 ACM. ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907312

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, ans 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×-2.14× 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 simplifies to handle undirected graphs.

Given an iterative vertex-centric graph algorithm $i\mathcal{A}$ and a large input graph $\mathcal{G}$, the accurate results of vertex values $V_\mathcal{G}$ can be computed by applying $i\mathcal{A}$ to $\mathcal{G}$, that is:

$$V_\mathcal{G} \;=\; i\mathcal{A}(\mathcal{G})$$

To accelerate this computation, we use the following steps:

- Reduce input $\mathcal{G}$ to $\mathcal{G}'$: we transform the large input graph $\mathcal{G}$ into a smaller graph $\mathcal{G}'$ via multiple applications of an input reduction transformation $\mathcal{T}$.
- Compute results for $\mathcal{G}'$: we apply $i\mathcal{A}$ to $\mathcal{G}'$ to compute $V_{\mathcal{G}'}$. Computing on $V_{\mathcal{G}'}$ takes lesser time than on $V_\mathcal{G}$.
- Obtain results for $\mathcal{G}$: using simple mapping rules $m\mathcal{R}$s, we convert the results $V_{\mathcal{G}'}$ to $V^1_\mathcal{G}$. Then, via multiple application of update rules in $i\mathcal{A}$, we reduce the error in $V^1_\mathcal{G}$ and obtain the result $V^2_\mathcal{G}$.

Thus, our approach replaces computation $V_\mathcal{G} = i\mathcal{A}(\mathcal{G})$ by:

$$\begin{array}{ll} \text{[INPUT REDUCTION]} & \mathcal{G}' \;=\; \mathcal{T}^\Delta(\mathcal{G}) \\ \text{[PHASE 1]} & V_{\mathcal{G}'} \;=\; i\mathcal{A}\,(\mathcal{G}') \\ \text{[MAP RESULTS]} & m\mathcal{R}:\; V_{\mathcal{G}'} \rightarrow V^1_\mathcal{G} \\ \text{[PHASE 2]} & V^2_\mathcal{G} \;=\; i\mathcal{A}\,(V^1_\mathcal{G},\, \mathcal{G}) \end{array}$$

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

2.1 Efficient Input Reduction Transformations

Given the iterative nature of algorithms considered, applying $i\mathcal{A}$ to $\mathcal{G}'$ as opposed to $\mathcal{G}$ 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 $\mathcal{G}$ to $\mathcal{G}'$.

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

$$\mathcal{G}_1 \leftarrow \mathcal{T}(v_1, \mathcal{G}); \quad \mathcal{G}_2 \leftarrow \mathcal{T}(v_2, \mathcal{G}_1) \quad \cdots$$
$$\cdots \quad \mathcal{G}_{\Delta-1} \leftarrow \mathcal{T}(v_{\Delta-1}, \mathcal{G}_{\Delta-2}); \quad \mathcal{G}' \leftarrow \mathcal{T}(v_\Delta, \mathcal{G}_{\Delta-1})$$

B. Non-interfering sequence. $\mathcal{T}^\Delta$, a sequence of Δ applications of local transformation $\mathcal{T}$ as shown above is *non-interfering* if and only if: vertices $v_1 \cdots v_\Delta$ are distinct vertices in $\mathcal{G}$; and each subGraph($\mathcal{T}(v_i, \mathcal{G})$) is contained in $\mathcal{G}$. 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 $\mathcal{G}$ is only examined when considering the application of $\mathcal{T}$ to v_i or v_j; and
- Any vertex or edge created during one application of $\mathcal{T}$ cannot be involved in any other application of $\mathcal{T}$.

Thus, the cost of applying the transformation sequence is linear in the size of $\mathcal{G}$, 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 $\mathcal{G}$. In contrast, those computations over the transformed portions of $\mathcal{G}$ 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 TPiA ( input G )
2:     G' ← ReduceGraph ( G, T, Δ )
3:     V_G^1 ← iA(G')
4:     V_G^2 ← iAP2(V_G^1, G)
5:     return V_G^2
6: end function

7: function ReduceGraph ( G, T, Δ )
8:     G' ← G
9:     for ( Vertex v : G ) do
10:        if ( NI ( subGraph(T(v,G)) ) then
11:            G' ← T(v,G')
12:            Δ ← Δ − 1
13:            if ( Δ == 0 ) then break end if
14:        end if
15:    end for
16:    return G'
17: end function

18: function iA ( input G )
19:    Initialize V_G & WorkQ
20:    while ( ! WorkQ.empty ) do
21:        v ← WorkQ.getFirst()
22:        if ( UpdateVals (v, V_G) ) then
23:            WorkQ.add ( outNeighbors (v) )
24:        end if
25:    end while
26:    return V_G
27: end function

28: function UpdateVals ( v, V_G )
29:    Updated ← false
30:    if  updateCheck ( v, inNeighbors (v) ) then
31:        update V_G[v]
32:        Updated ← true
33:    end if
34:    return Updated
35: end function

36: function iAP2 ( V_G^1, G )
37:    Initialize WorkQ
38:    for ( Vertex v : G ) do
39:        if ( v ∈ G' ) then
40:            V_G^2 ( v ) ← V_G^1 (v)
41:        else
42:            V_G^2 ( v ) ← initval ( )
43:            WorkQ.add ( v )
44:        end if
45:    end for
46:    WorkQ.add ( Vertex v s.t. v is affected by
47:         addition / deletion of edges)
48:    while ( ! WorkQ.empty ) do
49:        v ← WorkQ.getFirst()
50:        if ( UpdateVals (v, V_G) ) then
51:            WorkQ.add ( outNeighbors (v) )
52:        end if
53:    end while
54:    return V_G
55: end function
```

Algorithm 2 SSSP Algorithm.

```
1: function TwoPhaseSSSP ( input G; srcVertex )
2:     ▷ V_G of a vertex v = length of the
3:     ▷       shortest path from srcVertex to v
4: end function

5: function ReduceGraph ( G, T, Δ, srcVertex )
6:     ▷ srcVertex is not part of applied T's
7: end function

8: function InitializeSSSP ( input G; srcVertex )
9:     ▷ Initialize V_G
10:    for ( Vertex v : G ) do V_G[v] ← ∞
11:    end for
12:    V_G[srcVertex] ← 0
13:    ▷ Initialize WorkQ
14:    WorkQ.add( outNeighbors(srcVertex) )
15: end function

16: function UpdateVals ( v, V_G )
17:    Updated ← false
18:    for ( Vertex v' : inNeighbors (v) ) do
19:        if V_G[v] > V_G[v'] + wt(v', v) then
20:            V_G[v] ← V_G[v'] + wt(v', v)
21:            Updated ← true
22:        end if
23:    end for
24:    return Updated
25: end function

26: function Phase2SSSP ( V_G', G )
27:    initval() assigns ∞ or results from phase 1
28: end function
```

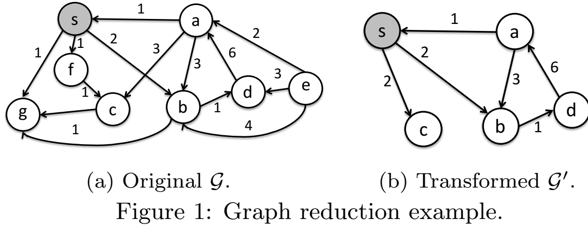
2.2 Original and Two-Phased Algorithms

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 `iA` represents the original algorithm whose application to graph $\mathcal{G}$ produces the accurate ($V_{\mathcal{G}}$) result. Function `TPiA` is the two phased version that calls `iA` and `iAP2` in first and second phases. Note that the processing logic in `iAP2` (lines 48-53) is exactly the same as that in `iA` (lines 20-25). The result ($V_{\mathcal{G}}^2$) is obtained from the application of `TPiA` to $\mathcal{G}$. The result obtained from `TPiA` might not be accurate; we discuss this in Sections 4.1 and 4.2.

ReduceGraph examines the vertices in $\mathcal{G}$ one at a time and if $\mathcal{T}(v, \mathcal{G}')$ is non-interfering with transformations already applied, then it is applied on v. The function `NI` enforces non-interference by ensuring that all vertices and edges in subGraph($\mathcal{T}(v, \mathcal{G})$) are being examined for the first time. The algorithm terminates after applying Δ transformations. The function `iAP2` copies results from vertices in $\mathcal{G}'$ to vertices in $\mathcal{G}$ for each vertex that is present in both graphs. The vertices in $\mathcal{G}$ that were eliminated in the process of creating $\mathcal{G}'$ are assigned initial values by `initval()`. Then, similar to `iA`, UpdateVals is applied to $V_{\mathcal{G}}^2$ until convergence.

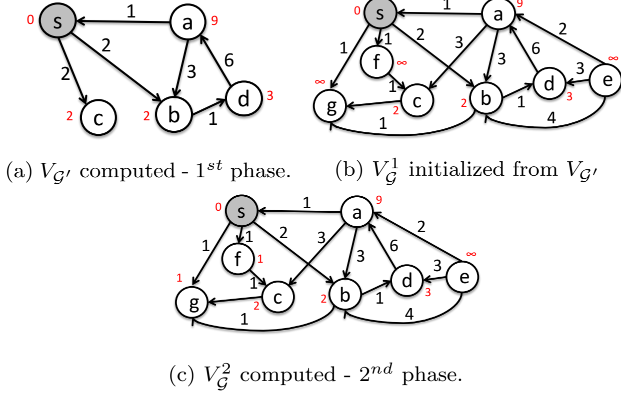
2.3 Example: Single Source Shortest Paths

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.



(a) Original $\mathcal{G}$. (b) Transformed $\mathcal{G}'$.

Figure 1: Graph reduction example.



(a) $V_{\mathcal{G}'}$ computed - 1st phase. (b) $V_{\mathcal{G}}^1$ initialized from $V_{\mathcal{G}'}$

(c) $V_{\mathcal{G}}^2$ computed - 2nd phase.

Figure 2: Two-phased SSSP processing on $\mathcal{G}$ & $\mathcal{G}'$.

Figure 1 illustrates graph reduction by converting $\mathcal{G}$ to $\mathcal{G}'$ and Figure 2 illustrates how the two-phased SSSP algorithm works on the example graph by first computing $V_{\mathcal{G}'}$ (Figure 2-a), then feeding these computed results to $V_{\mathcal{G}}^1$ (Figure 2-b), and then computing $V_{\mathcal{G}}^2$ (Figure 2-c). In this case a single application of UPDATEVALS in the second phase yields precise results (i.e., $V_{\mathcal{G}}^2 = V_{\mathcal{G}}$). 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

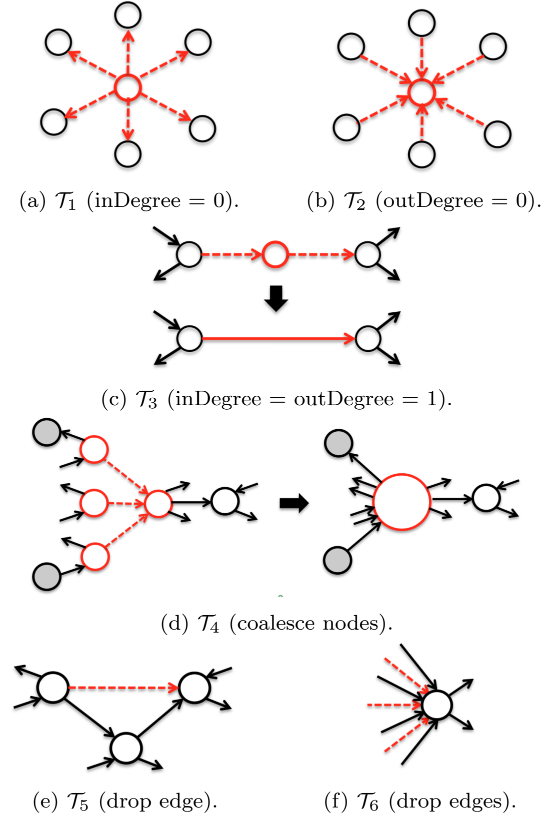
Since many graph algorithms are super-linear in the number of edges, the goal of graph reduction is to reduce the number of edges in the graph. If all edges involving a node are eliminated, then so is the node. Figure 3 shows the transformations. The red dashed edges are the ones that are eliminated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

$\mathcal{T}_1/\mathcal{T}_2$. If vertex v has no incoming/outgoing edges, its outgoing/incoming edges are removed and v is dropped.

$\mathcal{T}_3$. For every vertex v with a single incoming and a single outgoing edge, transformation $\mathcal{T}_3$ 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 $\mathcal{T}_3$ ensures that a path between two vertices v and w is preserved even though direct edges or intervening nodes are dropped.

$\mathcal{T}_4$. For a vertex v with high number of incoming edges,[1] transformation $\mathcal{T}_4$ merges the vertices for those incoming edges with v. $\mathcal{T}_4$ achieves input graph reduction by coalescing directly connected nodes so that the edges connecting them are eliminated and a reduced graph with fewer edges is obtained. This approach does not reduce connectivity, rather it can introduce new directed paths that were not present in the original graph, hence increasing connectivity. As seen in Figure 3, $\mathcal{T}_4$ adds a path between the two gray vertices which is not present in the original graph.

[1] An indegree threshold can be set while using $\mathcal{T}_4$ and $\mathcal{T}_6$. Based on our experiments, we set this threshold to be 1000.



(a) $\mathcal{T}_1$ (inDegree = 0). (b) $\mathcal{T}_2$ (outDegree = 0).

(c) $\mathcal{T}_3$ (inDegree = outDegree = 1).

(d) $\mathcal{T}_4$ (coalesce nodes).

(e) $\mathcal{T}_5$ (drop edge). (f) $\mathcal{T}_6$ (drop edges).

Figure 3: Transformations for Input Reduction.

$\mathcal{T}_5$. This transformation drops edge $v \rightarrow w$, if there exists a u such that $v \rightarrow u$ and $u \rightarrow w$. Effectively, for vertex v, $\mathcal{T}_3$ drops the outgoing edge $v \rightarrow w$ if a neighboring vertex of v is directly connected to w. As in $\mathcal{T}_3$, $\mathcal{T}_5$ ensures path preservation; however, $\mathcal{T}_5$ increases the hops/distance between connected vertices.

$\mathcal{T}_6$. Transformations $\mathcal{T}_1$-$\mathcal{T}_5$ can only be applied when their preconditions are satisfied. Thus, the amount of reduction obtained will depend upon the input graph's structural characteristics. In fact, in our experiments the input graph FT is greatly reduced by $\mathcal{T}_1$-$\mathcal{T}_5$ compared to the other graphs. Hence, we introduce transformation $\mathcal{T}_6$ which randomly eliminates incoming edges for a given vertex with high indegree.[1] In this case, the edges are dropped in proportion to the vertex's indegree. Since $\mathcal{T}_6$ can aggressively eliminate edges, it is applied when none of the previous transformations ($\mathcal{T}_1$-$\mathcal{T}_5$) can be used because the vertex does not satisfy their corresponding preconditions.

We classify $\mathcal{T}_5$ and $\mathcal{T}_6$ as *aggressive* transformations mainly because they do not fully preserve the structural similarity

Algorithm 3 Graph Reduction Algorithm.

```
1:  Algorithm TRANSFORM ( G(V, E) )
2:    E' ← E
3:    for ∀v ∈ V do
4:      if ( inDegree(v) = 0 ) then
5:            ▷ apply T1 : drop v → ∗
6:               E' ← E' \ outEdges(v)
7:      elseif ( outDegree(v) = 0 ) then
8:            ▷ apply T2 : drop ∗ → v
9:               E' ← E' \ inEdges(v)
10:     elseif ( inDegree(v) = outDegree(v) = 1 ) then
11:           ▷ apply T3 : bypass v
12:              E' ← (E' \ {u → v, v → w}) ∪ {u → w}
13:             where {u → v, v → w} ⊆ E'
14:     elseif ( all inNeighbors(v) are unchanged ) then
15:           ▷ apply T4 : coalesce v and inNeighbors(v)
16:              E' ← COALESCE(G, E', v)
17:     end if
18:   end for
19:   if ( G requires further reduction ) then
20:     for ∀v ∈ V s.t. v is unchanged do
21:       if ( w ∈ outNeighbors(v) s.t. w is unchanged and
22:          outNeighbors(v) ∩ inNeighbors(w) ≠ φ ) then
23:               ▷ apply T5 : drop v → w
24:                  E' ← E' \ {(v → w)}
25:       elseif ( inDegree(v) > threshold ) then
26:               ▷ apply T6 : drop some ∗ → v
27:                  E' ← E' \ R where R ⊆ inEdges(v)
28:       end if
29:     end for
30:   end if
31:   return E' of G'
32: end algorithm
33:
34: Algorithm COALESCE ( G(V, E), E', v )
35:   for ∀(w → v) ∈ inEdges(v) do
36:     E' ← E' \ {w → v}
37:     for ∀(u → w) ∈ inEdges(w) do
38:       E' ← E' \ {u → w}
39:       E' ← E' ∪ {u → v}
40:     end for
41:     for ∀(w → u) ∈ outEdges(w) do
42:       E' ← E' \ {w → u}
43:       E' ← E' ∪ {v → u}
44:     end for
45:   end for
46:   return E' of G'
47: end algorithm
```

between the transformed graph and the original graph. In particular, $\mathcal{T}_5$ 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. $\mathcal{T}_6$, 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 $\mathcal{T}_1$-$\mathcal{T}_4$ 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).

Trans.	[V-ADD]	[V-SUB]	[E-ADD]	[E-SUB]	[C-MERGE]	[C-SPLIT]
$\mathcal{T}_1$	✗	✓	✗	✓	✗	?
$\mathcal{T}_2$	✗	✓	✗	✓	✗	?
$\mathcal{T}_3$	✗	✓	✓	✓	✗	✗
$\mathcal{T}_4$	✗	✓	✓	✓	✗	✗
$\mathcal{T}_5$	✗	✗	✗	✓	✗	✗
$\mathcal{T}_6$	✗	✗	✗	✓	✗	?

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.

3.2 Transformation Properties

We consider each transformation and deduce strong guarantees about various properties of the transformed graph $\mathcal{G}'$ compared to that of the original graph $\mathcal{G}$. 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 $\mathcal{G}'$ with $\mathcal{G}$ when transformation $\mathcal{T}_k$ ($1 \leq k \leq 6$) is applied.

[V-ADD]: $\mathcal{T}_k$ results in vertex v s.t. $v \in \mathcal{G}', v \notin \mathcal{G}$.

[V-SUB]: $\mathcal{T}_k$ results in vertex v s.t. $v \in \mathcal{G}, v \notin \mathcal{G}'$.

[E-ADD]: $\mathcal{T}_k$ results in edge e s.t. $e \in \mathcal{G}', e \notin \mathcal{G}$.

[E-SUB]: $\mathcal{T}_k$ results in vertex e s.t. $e \in \mathcal{G}, e \notin \mathcal{G}'$.

[C-MERGE]: $\mathcal{T}_k$ results in a new component c s.t. $c_1 \in \mathcal{G}, c_2 \in \mathcal{G}, c = c_1 \cup c_2, c \in \mathcal{G}'$.

[C-SPLIT] $\mathcal{T}_k$ results in new components c_1 and c_2 s.t. $c_1 \in \mathcal{G}', c_2 \in \mathcal{G}', c = c_1 \cup c_2, c \in \mathcal{G}$.

It is easy to follow that $\mathcal{T}_1$ and $\mathcal{T}_2$ 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 $\mathcal{T}_3$ and $\mathcal{T}_5$ guarantee occurrence of [E-SUB] and non-occurrence of [V-ADD], [C-MERGE], [C-SPLIT]. $\mathcal{T}_3$ also guarantees occurrence of [E-ADD] and [V-SUB], whereas $\mathcal{T}_5$ also guarantees non-occurrence of [E-ADD] and [V-SUB]. Transformation $\mathcal{T}_4$ guarantees occurrence of [V-SUB], [E-ADD], [E-SUB] and non-occurrence of [V-ADD], [C-MERGE], [C-SPLIT]. Finally, $\mathcal{T}_6$ guarantees occurrence of [E-SUB] and non-occurrence of [V-ADD], [V-SUB], [E-ADD], [C-MERGE]. While dropping edges using $\mathcal{T}_6$, [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 $\mathcal{T}_3$ and $\mathcal{T}_4$ guarantee occurrence of [E-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 $\mathcal{G}'$ with that of $\mathcal{G}$ when transformation $\mathcal{T}_k$ ($1 \leq k \leq 6$) is applied. In the following expressions, $a \implies b$ means $b \in \mathcal{G}'$ is resulted from $a \in \mathcal{G}$.

[E-EQUAL] $\mathcal{T}_k$ results in edges e_1 and e_2, both with weights $w(e)$ s.t. $e_1 \in \mathcal{G}, e_1 \notin \mathcal{G}', e_2 \in \mathcal{G}', e_2 \notin \mathcal{G}, e_1 \implies e_2$.

[E-FUNC] $\mathcal{T}_k$ 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. $\{e_1, e_2\} \in \mathcal{G}, \{e_1, e_2\} \notin \mathcal{G}', e_3 \in \mathcal{G}', e_3 \notin \mathcal{G}$, $w(e_3) = func(w(e_1), w(e_2)), (e_1, e2) \implies e_3$.

[E-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-SUM], [E-MIN], or [E-MAX] respectively).

Transformation $\mathcal{T}_3$ guarantees occurrence of [E-FUNC] and non-occurrence of [E-EQUAL]. For transformation $\mathcal{T}_4$, both [E-EQUAL] and [E-FUNC] can occur. As we will see in Section 4.1, we use [E-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 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 $\mathcal{G}'$, and later on the original graph $\mathcal{G}$. For illustration, we use two versions of the SSSP vertex functions, SSSP-IN and SSSP-SIN,

Algorithm 4 Variants of SSSP vertex functions.

```
 1: function SSSP-IN ( Vertex v )
 2:     if ( v = source ) return 0; end if
 3:     minPath ← ∞
 4:     for ( Vertex u : inNeighbors (v) ) do
 5:         if ( u.path + wt(u, w) < minPath ) then
 6:             minPath ← u.path + wt(u, w)
 7:         end if
 8:     end for
 9:     return minPath
10: end function
11:
12: function SSSP-SIN ( Vertex v )
13:     if ( v = source ) return 0; end if
14:     minPath ← v.path
15:     for ( Vertex u : inNeighbors (v) ) do
16:         if ( u.path + wt(u, w) < minPath ) then
17:             minPath ← u.path + wt(u, w)
18:         end if
19:     end for
20:     return minPath
21: end function
```

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 $minPath$ (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-SUB] and [E-SUB]: [E-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-SUB] since the effect of vertex deletion is viewed as edge deletion by its neighbors, reducing to [E-SUB]. In both of these cases, SSSP-IN and SSSP-SIN produce an over-approximation of path distance when applied on $\mathcal{G}'$, compared to the precise distance computed on $\mathcal{G}$, i.e., $minPath(\mathcal{G}') \geq minPath(\mathcal{G})$. In the second phase when missing vertices and edges become available in $\mathcal{G}$, this approximation automatically gets corrected.

(B) [E-ADD], [E-EQUAL], and [E-FUNC]: Transformations resulting in [E-ADD] are introduced in order to preserve the connectivity in the graph which is essential for various traversal-based graph algorithms. Moreover, both [E-EQUAL] and [E-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-

Alg.	Vertex Function
SSSP	$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.path + e.wt)$
SSWP	$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.wt))$
CC	$v.component \leftarrow \min_{e \in \text{edges}(v)} (e.other.component)$
PR	$v.rank \leftarrow 0.15 + 0.85 \times \sum_{e \in \text{inEdges}(v)} e.source.rank$

Alg.	Vertex Function
GC	$change \leftarrow \bigvee_{e \in \text{edges}(v)} (v.color == e.other.color)$ **if** $change == true$ **then:** $v.color \leftarrow c$: where $\forall_{e \in \text{edges}(v)} (e.other.color \neq c)$
CD	$\forall_{e \in \text{edges}(v)} frequency[e.other.community]$ += 1 $v.community \leftarrow c$: where $frequency[c] = \max_{i \in frequency} (frequency[i])$

Table 2: Various vertex-centric graph algorithms. SSSP, SSWP, CC, PR, and GC produce 100% accurate results.

inal graph in the second phase. However, care must be taken to ensure that algorithms which cannot tolerate such newly added relationships do run correctly; in such cases, the newly added edges can be eliminated dynamically from the computation. When [E-ADD] results from eliminating intermediate vertices such that there is a path between the end vertices in $\mathcal{G}$ (as in $\mathcal{T}_3$), correctness of both SSSP-IN and SSSP-SIN is guaranteed by [E-SUM].

However, $\mathcal{T}_4$, which results in [E-EQUAL], can add an edge between two vertices across which a directed path did not exist in $\mathcal{G}$. In this case, the approximation computed by SSSP-IN and SSSP-SIN can include calculated paths that are smaller than the true shortest paths. During the second phase using $\mathcal{G}$, SSSP-IN recovers from such approximation since the computation of a path does not depend on its own previous value, resulting in 100% accurate results.[2] On the other hand, computation in SSSP-SIN relies on the previously computed path value for the given vertex, and hence SSSP-SIN cannot recover from such approximate solution. In this case, instead of directly using [E-EQUAL], the edge weight for such newly added edges resulting in new paths can be set to ∞ (**[E-INF]**) which can guarantee 100% accurate results for SSSP-SIN as well.

(C) [C-SPLIT]: Finally, transformations resulting in [C-SPLIT] typically do not impact correctness since computations are performed locally at vertex-level. If the algorithm requires collaborative tasks at component level, they can be performed correctly in the second phase on the original graph. In our examples, both SSSP-IN and SSSP-SIN remain unaffected by [C-SPLIT].

Transformations beyond $\mathcal{T}_1$-$\mathcal{T}_6$. Note that our transformations can be used as fundamental building blocks to create more complicated transformations which can be applied to reduce the graph size. Conversely, the correctness of graph algorithms while using any new transformation $\mathcal{T}_x$ ($x > 6$) can be argued by reducing the new transformation to one or many of the proposed set of transformations. If there exists a sequence of transformations among $\mathcal{T}_1$-$\mathcal{T}_6$ which produces the same transformed subgraph as that produced by $\mathcal{T}_x$, correct answers can be guaranteed at the end of computation using the transformed graph produced by $\mathcal{T}_x$. For some $\mathcal{T}_x$ which cannot be expressed as a sequence of proposed transformations, arguments using their structural and non-structural properties can be used to ensure correctness of results. Note that this relationship is *transitive* and hence, the newly proved $\mathcal{T}_x$ can be further used along with $\mathcal{T}_1$-$\mathcal{T}_6$ to prove correctness of results while using other new transformations.

[2]This is true for graph structures consisting of loops as well.

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 $\mathcal{G}'$, 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, SSWP 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 calculated path width in $\mathcal{G}'$ is always at most that of the equivalent path in $\mathcal{G}$.

(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 $\mathcal{G}'$, which were disconnected in $\mathcal{G}$. 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 $\mathcal{G}$ become disconnected. This can cause the vertices to be assigned

[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 $\mathcal{G}'$. However, during the second phase, these edges become available in $\mathcal{G}$ which reprocesses 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 $\mathcal{G}'$ provide an approximation of the final results, these results, when fed as initialization values for $\mathcal{G}$, 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 $\mathcal{G}$ 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 $\mathcal{G}'$ to converge to its final solution before moving on to $\mathcal{G}$. This is because the intermediate values produced while processing $\mathcal{G}'$ 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:

$$I_1.P_T + I_2.P_G^T < I.P_G \tag{1}$$

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:

$$I_1.P_T < I.P_G - I_2.P_G^T \tag{2}$$

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:

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

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

$$\implies I_1.P_T < \frac{3}{4}.I.P_G \quad \rightsquigarrow \quad |G_T| < \frac{3}{4}.|G| \tag{3}$$

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 $\mathcal{T}_4$ and $\mathcal{T}_6$ 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 Konect 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:

$$ERP = \frac{|E_{\mathcal{G}'}|}{|E_{\mathcal{G}}|} \times 100$$

where $|E_{\mathcal{G}}|$ and $|E_{\mathcal{G}'}|$ are the number of edges in original graph $\mathcal{G}$ and the reduced graph $\mathcal{G}'$. 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^{TM} 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×, 1.27×, 1.41×, 1.51× and 1.53× 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 = 30% the performance degrades and the average speedup drops to 1.41×. This is because the reduced graph with very low ERP becomes too small (i.e.,

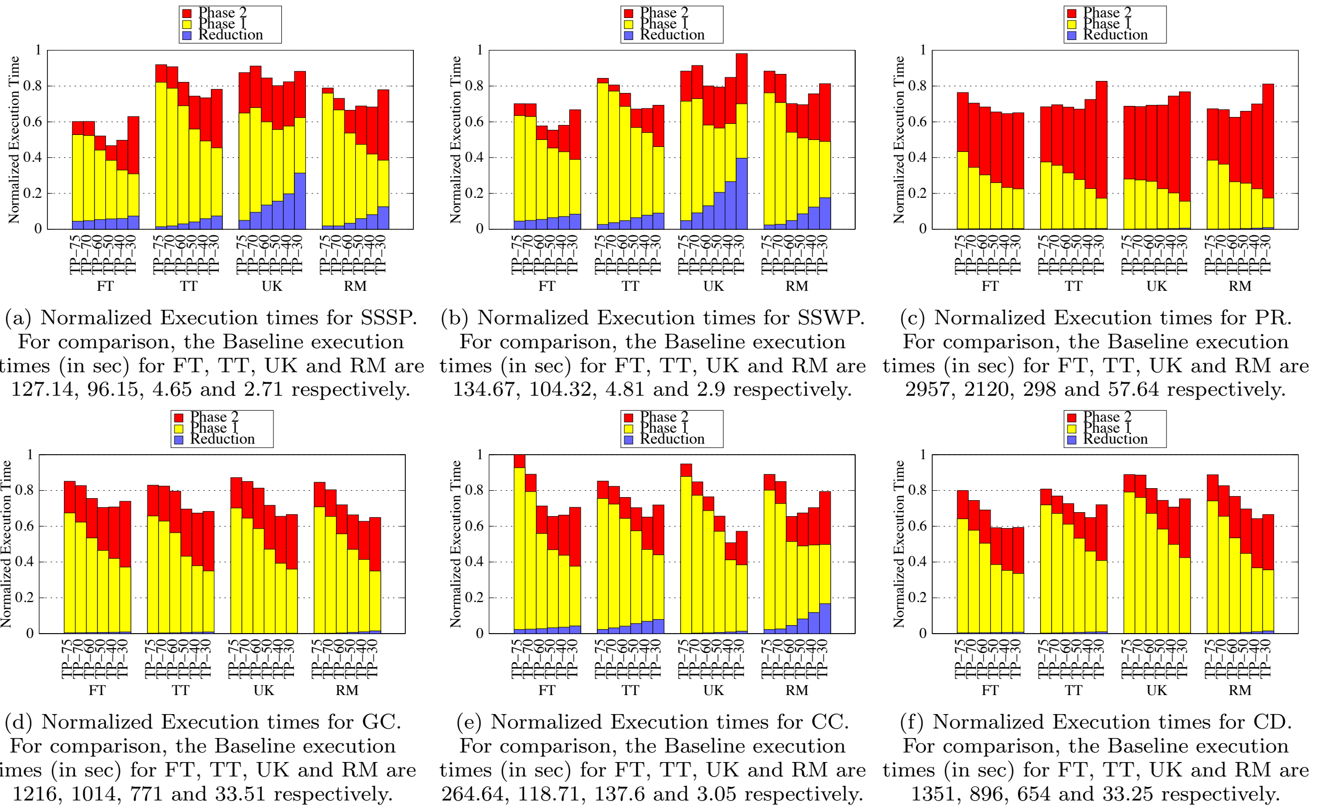
[4]Real-word graphs from various domains like social network analytics, web anaytics, mining, etc. are highly irregular.

Benchmark	Type
Single Source Shortest Path (SSSP) Single Source Widest Path (SSWP) PageRank (PR) Graph Coloring (GC) Connected Components (CC)	Accurate
Community Detection (CD)	Approximate

Table 3: Graph Algorithms.

Input Graph		Graph Size		Reduction
		#Nodes	#Edges	Time (sec)
Friendster (FT)	Original	68.3M	2.6B	5.63-9.37
	Reduced	41.9-51.8M	0.78-1.9B	
Twitter (TT)	Original	41.7M	1.5B	1.31-7.13
	Reduced	23.4-30.8M	0.4-1.1B	
UKDomain (UK)	Original	39.5M	936.4M	1.31-7.13
	Reduced	27.6-32.1M	280.9-702.3M	
RMAT-24 (RM)	Original	17M	268M	0.05-0.34
	Reduced	11.6-13.5M	80.4-201M	

Table 4: Input Graphs.



(a) Normalized Execution times for SSSP. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 127.14, 96.15, 4.65 and 2.71 respectively.

(b) Normalized Execution times for SSWP. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 134.67, 104.32, 4.81 and 2.9 respectively.

(c) Normalized Execution times for PR. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 2957, 2120, 298 and 57.64 respectively.

(d) Normalized Execution times for GC. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 1216, 1014, 771 and 33.51 respectively.

(e) Normalized Execution times for CC. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 264.64, 118.71, 137.6 and 3.05 respectively.

(f) Normalized Execution times for CD. For comparison, the Baseline execution times (in sec) for FT, TT, UK and RM are 1351, 896, 654 and 33.25 respectively.

Figure 4: Normalized execution time of two-phased execution for each benchmark-graph-ERP value.

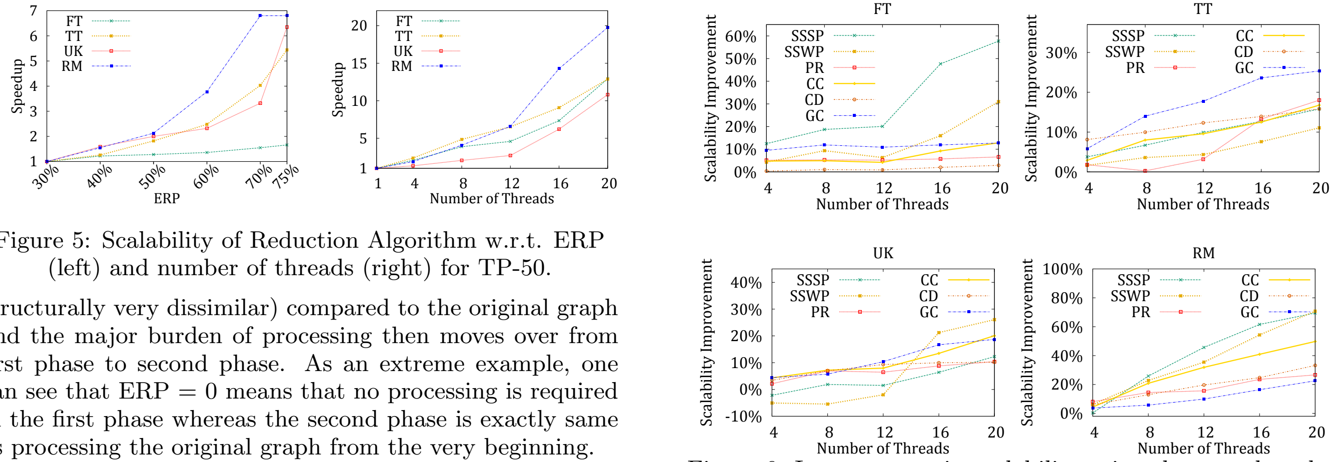


Figure 5: Scalability of Reduction Algorithm w.r.t. ERP (left) and number of threads (right) for TP-50.

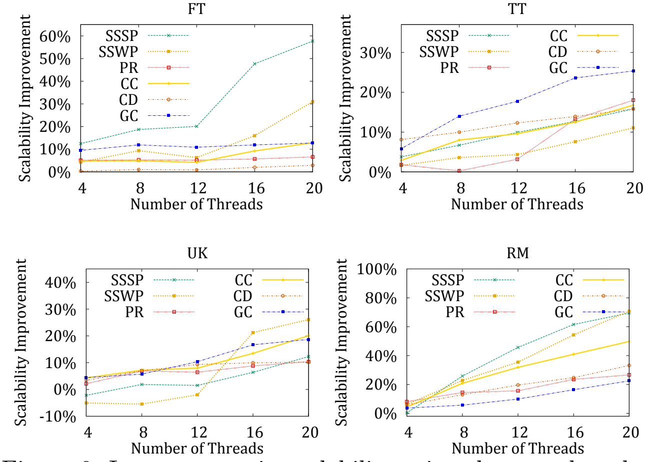


Figure 6: Improvement in scalability using the two-phased model with varying number of threads. For comparison, the Baseline execution times (in sec) for PR/SSSP with 1 thread for FT, TT, UK and RM are 24577/1674.43, 22307/1016.88, 3014.39/53.64 and 934.43/12.07.

structurally very dissimilar) compared to the original graph and the major burden of processing then moves over from first phase to second phase. As an extreme example, one can see that ERP = 0 means that no processing is required in the first phase whereas the second phase is exactly same as processing the original graph from the very beginning.

It is interesting to note that the benefits achieved from our two-phased approach are greater for FT graph (1.37-1.69×) mainly because it is larger than TT and UK graphs.

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 reduction 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. number 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%).

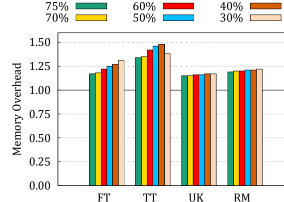


Figure 7: Increase in memory footprint.

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.

Input	TP-30	TP-40	TP-50	TP-60	TP-70	TP-75
FT	0.017	0.002	0.001	<1E-5	<1E-5	<1E-5
TT	0.049	0.041	0.036	0.021	0.019	0.017
UK	0.065	0.023	0.017	0.013	0.012	0.011
RM	0.043	0.034	0.021	0.018	0.012	0.01

Table 5: Relative Error for CD.

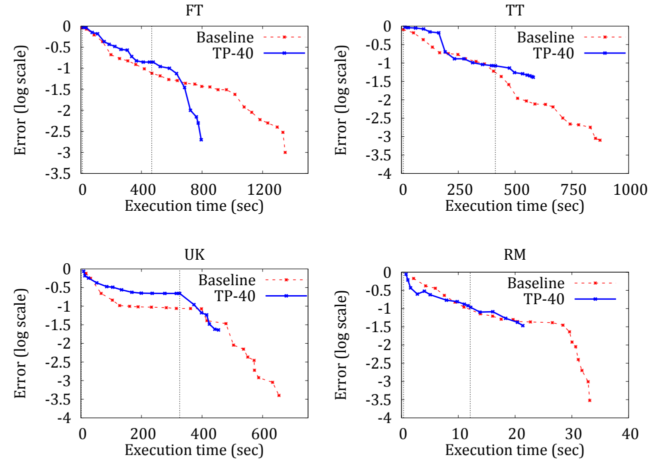


Figure 8: Relative Error (log scale) vs. Execution Time (sec) for CD: Baseline and TP-40. Note that the point at which the Baseline version terminates, i.e., relative error becomes zero, is not plotted due to use of log scale.

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 $\mathcal{T}_{1-k}$ as the set of transformations starting from $\mathcal{T}_1$ up to $\mathcal{T}_k$. Hence, the transformation set $\mathcal{T}_{1-4}$ includes $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ and $\mathcal{T}_4$ whereas $\mathcal{T}_{1-1}$ only includes $\mathcal{T}_1$.

The reduced graphs for this set of experiments are generated using different transformation sets $\mathcal{T}_{1-k}$ $(1 \geq k \leq 4)$. 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, $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ collectively reduce only small portion of TT, UK and RM graphs; $\mathcal{T}_{1-3}$ 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 $\mathcal{T}_4$ 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 $\mathcal{T}_2$ and $\mathcal{T}_3$, allowing 50% ERP to be achieved for $\mathcal{T}_{1-2}$ and $\mathcal{T}_{1-3}$ too. Hence, the speedups achieved for those transformation sets are ∼2.15×.

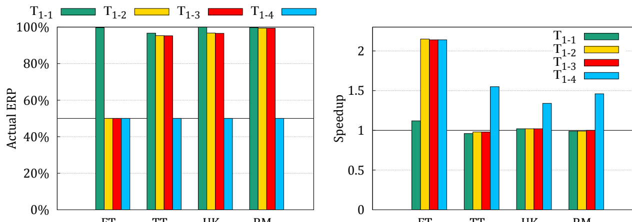


Figure 9: Actual ERP achieved (left) and speedups achieved (right) while using different transformation sets when ERP is set to 50%.

7. RELATED WORK

Graph processing has gained a lot of attention due to its applicability across various domains. Many graph processing frameworks have been developed for distributed ([16, 14, 31, 34, 24]), shared memory ([19, 28, 32]) and GPU based environments ([11, 26]). These frameworks include a parallel runtime that iteratively processes the input graph until all the graph values convergence. The computation is based on asynchronous or bulk synchronous model [30]. This traditional style of processing includes a single processing phase.

Multilevel transformation techniques. There is a body of work [7, 17, 3, 18, 23, 35, 9, 8, 10, 33] that reduces the size of graphs to accelerate processing. These works mainly rely on algorithm-specific reduction techniques and mostly operate of regular meshes. [7] presents 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. It uses edge contraction where neighbors are unified into a single vertex which is suitable for relatively regular meshes. [17] uses the same three phases and relies on quality functions of the reduced (coarse) grids based on aspect ratio. Moreover, the reduction algorithm operates on the dual graph and uses maximal independent set computation which requires non-trivial processing. [3, 9, 8, 10] also aim to partition graphs via recursive edge contraction using maximal independent set computation and edge contraction to generate multinodes. In contrast, our work identifies light-weight, local and non-interfering transformations that are *general* (i.e., not algorithm-specific) and effective for *irregular* 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. [33] processes queries by providing multiple levels of abstractions and refining the query to these abstraction levels. [23] is specifically designed for distance based algorithms like SSSP where they aim to achieve gate vertex sets which allow traversals to be constructed on the reduced graphs. [35] reduces by pruning weakest edges based on cost functions and which adhere to specific constraints related to connectivity maintenance. These works require path- or component-level transformations that are computationally expensive whereas our transformations are light-weight and hence effective for large graphs.

Beyond these works, various optimization techniques have been developed which attempt to accelerate processing at the cost of achieving approximate results. We divide the literature encompassing such approximation based graph 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× to 2.14×.

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

[1] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating Betweenness Centrality. WAW, 2007.

[2] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation. *PLDI*, pages 198-209, 2010.

[3] S. T. Barnard and H. D. Simon. Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice and experience*, 6(2):101–117, 1994.

[4] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. *PLDI* 2013.

[5] B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the Minimum Spanning Tree Weight in Sublinear Time. *International Colloquium on Automata, Languages and Programming*, 2005.

[6] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward. Authority Rankings from HITS, Pagerank, and SALSA: Existence, Uniqueness, and Effect of Initialization. *SIAM J. Scientific Computing*, 27(4):1181–1201, 2005.

[7] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. *ACM/IEEE Conference on Supercomputing*, 1995.

[8] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. *ICPP (3)*, pages 113–122, 1995.

[9] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[10] G. Karypis and V. Kumar. Multilevel K-way Partitioning Scheme for Irregular Graphs. *JPDC*, 48(1):96–129, 1998.

[11] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable SIMD-Efficient Graph Processing on GPUs. *PACT*'15.

[12] J. Kunegis. KONECT: The Koblenz Network Collection. WWW Companion, pages 1343-1350, 2013.

[13] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. *OSDI*, pages 31-46, 2012.

[14] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *VLDB Endowment*, 2012.

[15] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. *arXiv preprint arXiv:1408.2041*, 2014.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. *SIGMOD International Conf. on Management of Data*, 2010.

[17] I. Moulitsas and G. Karypis. Multilevel Algorithms for Generating Coarse Grids for Multigrid Methods. *Supercomputing*, pages 45–45, 2001.

[18] D. Nanongkai. Distributed Approximation Algorithms for Weighted Shortest Paths. *STOC*, pages 565-573, 2014.

[19] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. *PLDI*, pages 12-25, 2011.

[20] M. Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations that Discard Tasks. *ICS*, pages 324-334, 2006.

[21] M. C. Rinard. Using Early Phase Termination to Eliminate Load Imbalances at Barrier Synchronization Points. *OOPSLA*, pages 369-386, 2007.

[22] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric Graph Processing using Streaming Partitions. *SOSP*, pages 472–488, 2013.

[23] N. Ruan, R. Jin, and Y. Huang. Distance Preserving Graph Simplification. *ICDM*, pages 1200–1205, 2011.

[24] S. Salihoglu and J. Widom. GPS: A Graph Processing System. *SSDBM*, pages 22:1-22:12, 2013.

[25] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning Approximation for Graphics Engines. *MICRO-46*, pages 13-24, 2013.

[26] D. Sengupta, S. L. Song, K. Agarwal and K. Schwan. GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems. *SC*, pages 28:1–12, 2015.

[27] Z. Shang and J. X. Yu. Auto-approximation of Graph Computing. *VLDB Endow.*, pages 1833-1844, 2014.

[28] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *PPoPP*, pages 135-146, 2013.

[29] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. *ESEC/FSE* 2011.

[30] L. G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8):103–111, 1990.

[31] K. Vora, S. C. Koduru, and R. Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms using a Relaxed Consistency Based DSM. *OOPSLA*, pages 861-878, 2014.

[32] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous Large-scale Graph Processing Made Easy. *CIDR* 2013.

[33] K. Wang, G. Xu, Z. Su and Y. D. Liu. GraphQ: Graph Query Processing with Abstraction Refinement—Scalable and Programmable Analytics over Very Large Graphs on a Single PC. *USENIX ATC*, pages 387–401, 2015.

[34] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. *International Workshop on Graph Data Management Experiences and Systems*, 2013.

[35] F. Zhou, S. Malher, and H. Toivonen. Network Simplification with Minimal Loss of Connectivity. *ICDM*, pages 659–668, 2010.