Articulation Points Guided Redundancy Elimination for Betweenness Centrality

Lei Wang, Fan Yang, Liangji Zhuang, Huimin Cui, Fang Lv, Xiaobing Feng

State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Science
{wlei, yangfan2013, zhuangliangji, cuihm, flv, fxb}@ict.ac.cn

Abstract

Betweenness centrality (BC) is an important metrics in graph analysis which indicates critical vertices in large-scale networks based on shortest path enumeration. Typically, a BC algorithm constructs a shortest-path DAG for each vertex to calculate its BC score. However, for emerging real-world graphs, even the state-of-the-art BC algorithm will introduce a number of redundancies, as suggested by the existence of articulation points. Articulation points imply some common sub-DAGs in the DAGs for different vertices, but existing algorithms do not leverage such information and miss the optimization opportunity.

We propose a redundancy elimination approach, which identifies the common sub-DAGs shared between the DAGs for different vertices. Our approach leverages the articulation points and reuses the results of the common sub-DAGs in calculating the BC scores, which eliminates redundant computations. We implemented the approach as an algorithm with two-level parallelism and evaluated it on a multicore platform. Compared to the state-of-the-art implementation using shared memory, our approach achieves an average speedup of 4.6x across a variety of real-world graphs, with the traversal rates up to 45 ~ 2400 MTEPS (Millions of Traversed Edges per Second).

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; G.2.2 [*Discrete Mathematics*]: Graph Theory—Graph algorithms

General Terms Algorithms, Performance

Keywords Partial Redundancy Elimination; Parallelism; Betweenness Centrality

1. Introdection

Betweenness centrality (BC) [1] is an important metrics in graph analysis which indicates critical vertices in large-scale networks based on shortest path enumeration. It has been widely used in many areas for decades, such as lethality in biological networks [2] [3], analysis of transportation networks [4], study of sexual networks and AIDS [5], and contingency analysis for power grid component failures [6]. Nowadays, it is playing a significant role in studying the emerging social networks, e.g., in community detection [7] and in identifying key actors in terrorist networks [8][9]. Unfortunately, even the state-of-the-art algorithm for calculating BC scores runs in O(|V||E|) time for unweighted graphs, making the analysis of large graphs challenging. As the data volume is getting more tremendous, it is critical to improve the efficiency of BC algorithms in processing large-scale graphs.

We observed that for a large number of real-world graphs, existing BC algorithms have redundant computations. Let us take Brandes' algorithm for illustration, which is the fastest known algorithm for calculating BC [10], and is the basis of a number of parallel BC algorithms [11][12][13][14][15][16]. The algorithm firstly constructs a shortest-path DAG for each vertex by performing a breadth-first search (BFS) from that vertex, and then it traverses all these DAGs backward to accumulate the dependencies and update the BC scores of all vertices. Redundancy exists when there are articulation points in the graph. A vertex is an articulation point iff removing it disconnects the graph. Consider a graph G with one articulation point *a*, which divides G into two sub-graphs SG_1 and SG_2. For all vertices in SG_1, their DAGs will have a common sub-DAG which starts from the articulation point *a* and spans the sub-graph SG_2. However, existing BC algorithms do not leverage such information and they traverse the entire DAG for each vertex in SG_1, thus the common sub-DAG will be traversed multiple times, causing significant redundant computations.

In this paper, we propose an articulation-points-guided redundancy elimination approach, called APGRE, to eliminate the above redundancy. The key idea is to identify the common sub-DAGs shared between the shortest-path DAGs for multiple vertices using articulation points, and to remove partial and total redundancies in these sub-DAGs. Our approach consists of the following three steps. First, a graph G is decomposed into multiple sub-graphs (SG_1, …, SG_k) by a set of articulation points (a_1, …, a_{k-1}), where each sub-graph is associated with at least one articulation point. Second, for each sub-graph SG_i, a sub-DAG starting from each articulation point is constructed. Each sub-DAG is a common sub-DAG shared between the DAGs for those vertices not in SG_i. We count the size of a common sub-DAG and the number of the DAGs which share this sub-DAG. Finally, we calculate the BC scores for each sub-graph SG_i according to the dependency information of its corresponding sub-DAGs. As such, our approach can reuse the results in traversing the common sub-DAGs and eliminate redundant computations.

This paper makes the following contributions:

- We propose an articulation-point-guided redundancy elimination (APGRE) approach for Brandes' algorithm. It leverages the articulation points to classify the recursive relations for the dependency of a source vertex on all the other vertices into four possible cases of shortest paths, which are identified when calculating the BC scores of an individual sub-graph. Our approach reuses the results of those already visited sub-DAGs in computing the BC scores.
- Since the sub-graphs are independent, we implemented the APGRE approach as an algorithm with two-level parallelism on a shared memory platform, i.e., coarse-grained asynchro-

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.
PPoPP '16, March 12-16, 2016, Barcelona, Spain
© 2016 ACM. ISBN 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851154

nous parallelism among sub-graphs and fine-grained synchronous parallelism among vertices within one sub-graph. The two-level parallelism makes it possible to extract large amount of parallelism.

- Our experimental results show that compared to the state-of-the-art implementation, our APGRE algorithm achieves an average speedup of 4.6x for a variety of real-world graphs, ranging from 1.59x to 32.83x, with traversal rates up to 45 ~ 2400 MTEPS (Millions of Traversed Edges per Second) on a multi-core machine.

2. Background and Motivation

2.1 Definition and Brandes' Algorithm

Given graph G(V, E) and vertices *s, t* ∈ V, let σ_{st} be the number of the shortest paths from *s* to *t* in G, and $\sigma_{st}(v)$ be the number of the shortest paths that pass through a specified vertex *v*. The *betweenness centrality* of a vertex *v* is defined as: $BC(v) = \sum_{s\neq v\neq t\in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. Using this definition, a naive method can enumerates the all-pairs shortest-paths in $O(|V|^3)$ time [1].

Brandes [10] presented an efficient sequential algorithm which computes BC in O(|V||E|) time and O(|V| + |E|) space for unweighted graphs. Brandes' algorithm is based on a new accumulation technique. By defining the *dependency* of a source vertex *s* on any vertex *v* as $\delta_s(v) = \sum_{t\in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, the betweenness centrality of a vertex *v* can be expressed as follows:

$$BC(v) = \sum_{s\neq v\in V} \delta_s(v) \quad (1)$$

The key insight is that given pairwise distances and shortest paths counts, $\delta_s(v)$ satisfies the following recursive relation:

$$\delta_s(v) = \sum_{w:\ v\in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w)) \quad (2)$$

where $P_s(w)$ is the set of immediate predecessors of vertex *w* on the shortest paths from *s*.

In equation 2, vertex *v* is an immediate predecessor of vertex *w* on the shortest paths from *s*, or vertex *w* is an immediate successor of vertex *v*. If vertex *v* does not have an immediate successor, $\delta_s(v)$ is zero; otherwise $\delta_s(v)$ is the sum, $\frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$ of all immediate successors *w* of *v*.

Brandes' algorithm uses this insight to accumulate the dependencies and update the BC scores. For unweighted graphs, it performs breadth-first search (BFS) to count the number of the shortest paths and constructs a directed acyclic graph (DAG) of the shortest paths starting from each vertex (loop in line 4 in Figure1). The algorithm then traverses all these DAGs backward to accumulate the dependencies and add the dependencies to BC scores (loop in line 6 in Figure1).

The fine-grained parallel BC algorithms [11][12][13][14][15][16] perform the per-source vertex computations in parallel, and

```
Input: G(V,E)
Output: bc[]
1.  foreach s ∈ V {
2.    forall v ∈ V      // Initialize
3.      clear σst(v), Ps(v) , δs(v)
4.    forall v ∈ V      //Construct shortest-path DAG
5.      compute σst(v) and Ps(w)
6.    forall v ∈ D      // Traverse DAG backward
7.      compute δs(v)
8.      bc[v] += δs(v)
9.  }
```

Figure 1. Pseudocode for Brandes' algorithm

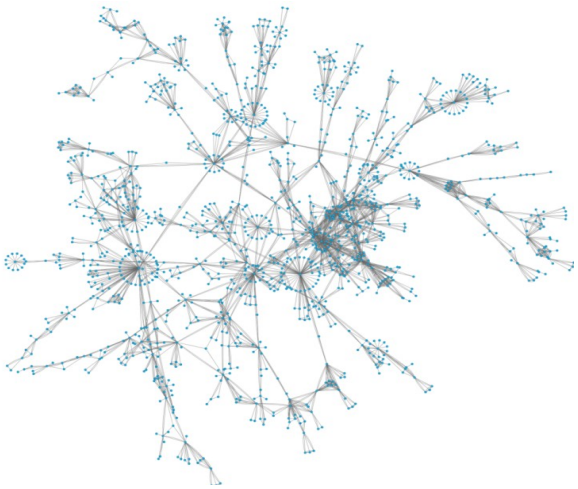


Figure 2. Human Disease Network (1419 vertices and 3926 edges)

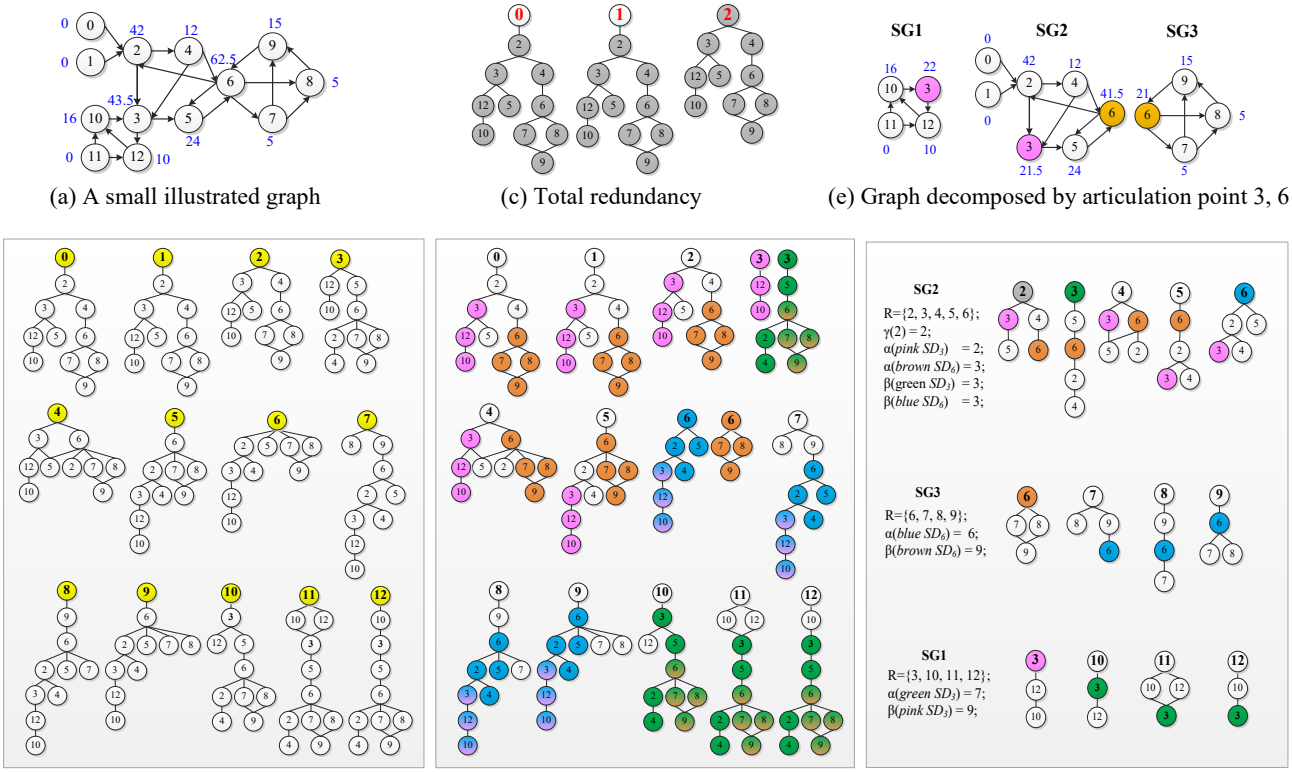
they are more space-efficient since they only need to maintain a single graph instance. Recent work on fine-grained parallel BC algorithms includes level-synchronous parallelization based on a BFS DAG, asynchronous parallelization by relaxing the order in computing BC, and hybridization by combining a top-down BFS algorithm and a bottom-up BFS algorithm.

2.2 Redundant Computation

By studying a large number of graphs, including social networks, web graphs, co-author networks, email networks, protein networks and road graphs, we observed that there are a large number of articulation points and a large number of vertices with a single edge, as Figure2 [29] shows. Real-world graphs typically have the power-law degree distributions, which implies that a small subset of the vertices are connected to a large fraction of the graph [30][31], and there are many vertices with a single edge.

Articulation points suggest considerable redundancies in calculating BC scores of unweighted graphs. A vertex in a connected graph is an articulation point iff removing it disconnects the graph. For example, in Figure3 (a), vertex 2, vertex 3 and vertex 6 are articulation points in the directed graph. When calculating BC scores of this graph, Brandes' algorithm constructs 13 DAGs rooted at each vertex (D_0~D_{12}), as shown in Figure3 (b). Figure3 (d) shows that there are four common sub-DAGs. The DAGs D_6, D_7, D_8, and D_9 share a common blue sub-DAG rooted at vertex 6, called *blue* SD_6 with vertices {6, 2, 5, 3, 4, 12, 10}. D_3, D_{10}, D_{11}, and D_{12} share a common green sub-DAG rooted at vertex 3, named *green* SD_3 with vertices {3, 5, 6, 2, 7, 8, 4, 9}. D_0, D_1, D_2, D_4 and D_5 have two common sub-DAGs, the *pink* SD_3 with vertices {3, 12, 10} and the *brown* SD_6 with vertices {6, 7, 8, 9}. In addition, a common sub-DAG can contain other smaller sub-DAGs, such as *blue* SD_6 contains *pink* SD_3, and *green* SD_3 contains *brown* SD_6. DAGs rooted at articulation points are comprised of common sub-DAGs, such as D_3 comprised by *pink* SD_3 and *green* SD_3, and D_6 comprised by *blue* SD_6 and *brown* SD_6. These common sub-DAGs lead to **partial redundancies**.

When articulation points have predecessors with no incoming edge and a single outgoing edge, **total redundancies** exist in calculating BC scores. For example, in Figure3 (a), vertex 2 is an articulation point, the degree of vertex 0 and vertex 1 are one. And the DAG D_2 is the sub-DAG of D_0 and D_1, as shown in Figure3 (c). The dependency δ_0 of vertex 0 on all other vertices can be derived from the dependency δ_2 of vertex 2 on all other vertices, so D_0 need not to be constructed. D_0 and D_1 are total redundancies.



(a) A small illustrated graph (c) Total redundancy (e) Graph decomposed by articulation point 3, 6

(b) DAGs in Brandes' algorithm (d) Partial redundancy (f) DAGs in APGRE algorithm

Figure 3. Illustration of partial and total redundancy in BC calculation.

2.3 Our Approach

In this paper, we focus on leveraging articulation points to reduce redundant computations. We leverage the articulation points to classify the recursive relation for the *dependency* of a source vertex on all other vertices into four possible cases of the shortest paths, which are identified when calculating the BC scores of an individual sub-graph. As a result, when computing the BC scores, our algorithm can reuse the results of the sub-DAGs already visited.

In our approach, a shortest-path DAG D_s rooted at a source s is divided by articulation points into sub-DAGs (SD_{a1}, …, SD_{ak}). Let $\alpha(SD_{ai})$ be the size of the sub-DAG SD_{ai} excluding the root vertex ai, and $\beta(SD_{ai})$ be the number of DAGs which contain the common sub-DAG SD_{ai}. When calculating the BC scores of an individual sub-graph SG_i, the triplet of source s, vertex v and target t can be categorized into four classes, and the dependency of s on any vertex v is classified into four recursive relations accordingly.

1. ***in2in***, $s, v, t \in SG$i, the sub-DAG SD_s rooted at s is in SG_i and is located at the top of D_s. All the shortest paths in SD_s are involved. The *in2in dependency* $\delta_{i2i(s)}(v)$ becomes a sub-problem of BC in a smaller graph.
2. ***in2out***, $s, v \in SG$i, $t \notin SGi$, then the paths from v to t must contain an articulation point a which connects SG_i to t. In this case, we just need to calculate the path s->v->a, because the shortest path from s to t must contain the path from s to a. A sub-DAG SD_a rooted at a is out of SG_i and is located at the bottom of D_s. The number of the shortest paths from s to t containing the path s->v->a in D_s equals to the size of SD_a excluding a. So only the shortest paths from s to a in SD_s are involved, and $\alpha(SD_a)$ is applied to the path s->v->a to accumulate the *in2out dependency* $\delta_{i2o(s)}(v)$.
3. ***out2in***, $v, t \in SG$i, $s \notin SGi$, then the paths from s to v must contain an articulation point a which connects s to SG_i. In this case, we just need to calculate the path a->v->t, because the shortest path from s to t must contain the path from a to t. A sub-DAG SD_a rooted at a is in SG_i and is located at the bottom of D_s. SD_a is a common sub-DAG shared among multiple DAGs rooted at different s. $\beta(SD_a)$ is the number of D_s that contain SD_a. All the shortest paths in SD_a are involved, and $\beta(SD_a)$ is applied to all paths in SD_a to accumulate the *out2in dependency* $\delta_{o2i(a)}(v)$.
4. ***out2out***, $v \in SG$i, $s, t \notin SGi$, they are in three different sub-graphs, and must be connected by two articulation points $a1$ and $a2$ in SG_i. In this case, we just need to calculate the path $a1$->v->$a2$, because all the shortest paths from s to t containing v must contain the shortest path $a1$->v->$a2$. SD_{a1} rooted at $a1$ is in SG_i and is located in the middle of D_s. SD_{a2} rooted at $a2$ is out of SG_i and is located at the bottom of D_s. So only the shortest paths $a1$->v->$a2$ in SD_{a1} are involved, and $\beta(SD_{a1})$ and $\alpha(SD_{a2})$ are applied to $a1$->v->$a2$ to accumulate the *out2out dependency* $\delta_{o2o(a1)}(v)$.

In a word, an articulation point a has $\delta_{i2i(a)}$, $\delta_{i2o(a)}$, $\delta_{o2i(a)}$ and $\delta_{o2o(a)}$, and a non-articulation point u has $\delta_{i2i(u)}$ and $\delta_{i2o(u)}$. Therefore, the BC scores of an individual sub-graph SG_i can be computed as follows:

$$BC_{SGi}(v) = \sum_{s,a \neq v \in SGi} (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v) + \delta_{o2i(a)}(v) + \delta_{o2o(a)}(v))$$

The articulation point a is shared by several sub-graphs, so BC(a) is the sum of its local BC scores in sub-graphs.

We can further reduce total redundancies based on the above four types of dependency. Let vertex u be a vertex with no incoming edges and a single outgoing edge u->s, then the DAG D_s rooted at s is the sub-DAG of D_u rooted at u. The dependency δ_u of u on all other vertices can be derived from the dependency δ_s of s on all other vertices, so D_u does not need to be constructed. Let $\gamma(D_s)$ be the number of D_u, and $\gamma(s)$ be the number of vertex s' neighbours which have a single outgoing edge to s, and with no incoming edges. Let R be a set of root vertices from which DAGs are constructed. Vertex u is removed from R, and $\gamma(s)$ is increased by one. When calculating the BC scores of an individual sub-graph SG_i, $\delta_{i2i(u)}$ and $\delta_{i2o(u)}$ can be derived from $\delta_{i2i(s)}$ and $\delta_{i2o(s)}$, because u is not an articulation point. Therefore, the BC scores of an individual sub-graph SG_i can be expressed as follows:

$$BC_{SGi}(v) = \sum_{s,a\neq v;s,a\in R_{sgi}} ((1+\gamma(s)) * (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v)) + \delta_{o2i(a)}(v) + \delta_{o2o(a)}(v)) + \sum_{s=v;s\in R_{sgi}} \gamma(s) * (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v))$$

The key insight of our approach is to identify common sub-DAGs shared among multiple DAGs using articulation points, and to remove partial and total redundancies in these sub-DAGs. Our approach consists of the following three steps: decomposing an unweighted graph through articulation points, counting the size of a common sub-DAG and the number of DAGs which share this sub-DAG, and calculating the BC scores of each sub-graph using the four types of dependency. How to recursively accumulate the four types of dependency and how to compute the α and β of a common sub-DAG are introduced in section 3.

Example 1 (DAGs in our approach). *Figure3 (e) and (f) show the DAG construction and dependency accumulation on the decomposed graphs for BC computation.* The graph in Figure3 (a) are decomposed into three sub-graphs (SG_1, SG_2 and SG_3) by articulation points 3 and 6, as shown in Figure3 (e). Figure3 (f) shows the DAGs in our approach. The four common sub-DAGs occur only once: *pink* SD_3 is in SG_1; *brown* SD_6 is in SG_3; part of *green* SD_3 is in SG_2 and the other part is in SG_3, since *brown* SD_6 is a sub-DAG of *green* SD_3; part of *blue* SD_6 is in SG_2 and the other part is in SG_1, since *pink* SD_3 is a sub-DAG of *blue* SD_6. When calculating **the BC scores of SG_3**, *blue* SD_6 is shared among D_7, D_8 and D_9, and *brown* SD_6 is shared among D_0, D_1, D_2, D_3, D_4, D_5, D_{10}, D_{11} and D_{12}. As a result, α(*blue* SD_6) is applied to the shortest paths *7->v->6*, *8->v->6*, *9->v->6* to accumulate $\delta_{i2o(7)}$, $\delta_{i2o(8)}$, and $\delta_{i2o(9)}$, respectively, and β(*brown* SD_6) is applied to all the shortest paths of *brown* SD_6 to accumulate $\delta_{o2i(6)}$. When calculating **the BC scores of SG_2,** since all the shortest paths from vertices in SG_1 to vertices in SG_3 must contain the shortest path *3->v->6*, β(*green* SD_3) and α(*brown* SD_6) are applied to *3->v->6* to accumulate $\delta_{o2o(3)}$; since all the shortest paths from vertices in SG_3 to vertices in SG_1 must contain the shortest path *6->v->3*, β(*blue* SD_6) and α(*pink* SD_3) are applied to *6->v->3* to accumulate $\delta_{o2o(6)}$; *pink* SD_3 is shared among D_2, D_4, D_5 and D_6, and α(*pink* SD_3) is acted on the shortest paths 2->v->3, 4->v->3, 5->v->3, 6->v->3 to accumulate $\delta_{i2o(2)}$, $\delta_{i2o(4)}$, $\delta_{i2o(5)}$, and $\delta_{i2o(6)}$, respectively; since *brown* SD_6 is shared among D_2, D_3, D_4 and D_5, α(*brown* SD_6) is acted on the shortest paths 2-> v ->6, 3->v->6, 4->v->6, 5->v->6 to accumulate $\delta_{i2o(2)}$, $\delta_{i2o(3)}$, $\delta_{i2o(4)}$, and $\delta_{i2o(5)}$, respectively; *green* SD_3 is shared among D_{10}, D_{11} and D_{12}, thus β(*green* SD_3) is applied to all shortest paths of *green* SD_3 to accumulate $\delta_{o2i(3)}$; *blue* SD_6 is shared among D_7, D_8 and D_9, and β(*blue* SD_6) is applied to all shortest paths of *blue* SD_6 to accumulate $\delta_{o2i(6)}$; finally, since D_2 is a sub-DAG of D_0 and D_1, γ(*2*) is 2, and vertices 0 and 1 are removed from R. From Figure3 (f), we can see that an articulation point is a root of two sub-DAGs, if SD_a is in SG_i, β(SD_a) is needed, otherwise α(SD_a) is needed.

3. APGRE: Redundancy Elimination Approach for BC

In this section, we introduce a partial and total redundancy elimination approach for BC, called APGRE. We divide the recursive relation for the dependency of a source vertex on all other vertices into four possible cases of shortest paths, which are identified when calculating the BC scores of an individual sub-graph. As such, our algorithm can reuse the results of those sub-DAGs that are already visited when computing the BC scores for vertices different from the one currently processed. We first describe the new algorithm, then we discuss the correctness of our algorithm.

3.1 Redundancy Elimination Algorithm

Definition 1 (Decomposed Graph). A vertex in a connected graph is an articulation point iff removing it disconnects the graph. A graph G(V, E) can be decomposed into multiple sub-graphs connecting each other through articulation points. A sub-graph is expressed as SGi(V, E, A) or (V_{sgi}, E_{sgi}, A_{sgi}), where A is the set of articulation-points in *SGi*.

We now show how to express common sub-DAGs using three variables of articulation points. An articulation point a connects several sub-graphs (*SGi, SGj, …, SGk*), and it is shared by these sub-graphs. The articulation point a in *SGi* can express two shortest-path sub-DAGs, SD_a in *SGi,* and SD_a out of *SGi,* respectively. Let $\alpha_{SGi}(a)$ be the size of SD_a that is out of *SGi*, excluding the root a; $\beta_{SGi}(a)$ be the number of DAGs containing the SD_a that is in *SGi*; $\gamma_{SGi}(a)$ be the number of DAGs D_s which has an edge s->D_a. For unweighted graphs, $\alpha_{SGi}(a)$ is the number of vertices which a can reach without passing through *SGi* in G, and it can be obtained by BFS; $\beta_{SGi}(a)$ is the number of vertices which can reach a (or which a is reachable from) without passing through *SGi* in G, and it can be obtained by reverse BFS from a; $\gamma_{SGi}(a)$ is the number of vertex a' neighbours with no incoming edges and a single outgoing edge in *SGi*.

The decomposed graph and the shortest paths have the following properties:

1. In an unweighted graph, if *SGi* connects to *SGj* through an articulation point a, a must lie on all the shortest paths between vertices in *SGi* and vertices in *SGj*.
2. If a vertex $v \in V$ lies on the shortest path between vertices $s, t \in V$, then $\sigma_{st}(v) = \sigma_{sv} * \sigma_{vt}$.[10]
3. Any connected graph decomposes into a tree of biconnected components. These biconnected components are attached to each other at shared vertices called articulation points.
4. In the decomposed graph, an articulation point is split across multiple sub-graphs, and an edge in G is assigned to one sub-graph.

3.1.1 Four Types of Dependency

We classify the recursive relation for the dependency of a source vertex on all other vertices into four possible sub-problems, the *in2in dependency*, the *in2out dependency*, the *out2in dependency*, and the *out2out dependency,* respectively. The four types of dependency can reuse the results in traversing common sub-DAGs.

For $s, v, t, a \in SG$i, $a \in A_{sgi}$, given shortest paths counts and shortest-path DAGs in *SGi*, the four types of dependency in SG_i can be expressed as follows. Here $P_s(w)$ is a set of immediate predecessors of a vertex w on shortest paths from s, and $succ_s(v)$

is a set of immediate successors of a vertex v on shortest paths from s.

1. ***in2in dependency*** $\delta_{i2i(s)}(v)$. It is defined as $\delta_{i2i(s)}(v) = \sum_{t \in SGi} \delta_{i2i(s,t)}(v) = \sum_{t \in SGi} \frac{\sigma_{st}(v)}{\sigma_{st}}$, and obeys the following recursive relation,

$$\delta_{i2i(s)}(v) = \begin{cases} 0 & \text{if } succ_s(v) = \emptyset \\ \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * (1 + \delta_{i2i(s)}(w)) & \text{if } succ_s(v) \neq \emptyset \end{cases} \quad \textbf{(3)}$$

$\delta_{i2i(s)}(s)$ is zero. All shortest paths of DAG D_s are involved to accumulate $\delta_{i2i(s)}(v)$. We construct shortest-paths DAGs for each vertex in *SGi*, and accumulate their *in2in dependencies*.

2. ***in2out dependency*** $\delta_{i2o(s)}(v)$. It is defined as $\delta_{i2o(s)}(v) = \sum_{a \in A_{sgi}} \delta_{i2o(s,a)}(v) = \sum_{a \in A_{sgi}} \frac{\sigma_{sa}(v)}{\sigma_{sa}} * \alpha_{SGi}(a)$, and obeys the following recursive relation,

$$\delta^{init}_{i2o(s)}(v) = \begin{cases} \alpha_{SGi}(v) & \text{if } v \in \mathrm{A}_{sgi} \\ 0 & \text{if } v \notin \mathrm{A}_{sgi} \end{cases}$$

$$\delta_{i2o(s)}(v) = \begin{cases} \delta^{init}_{i2o(s)}(v) & \text{if } succ_s(v) = \emptyset \\ \delta^{init}_{i2o(s)}(v) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \delta_{i2o(s)}(w) & \text{if } succ_s(v) \neq \emptyset \end{cases}$$

(4)

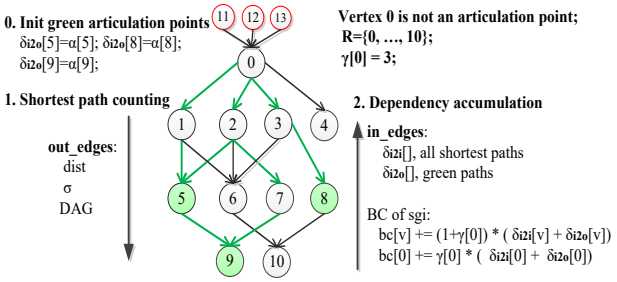
$\delta_{i2o(s)}(s)$ is zero. Only the shortest paths from s to articulation points in DAG D_s are involved to accumulate $\delta_{i2o(s)}(v)$. Each vertex in *SGi* can be as a source vertex to accumulate *in2out dependencies* of it on all other vertices.

In Figure4 (a), vertex 5, 8, 9 are articulation points, and vertices on green paths have *in2out dependencies* $\delta_{i2o(0)}(v)$. For example, vertex 3 lies on the shortest path 0->3->8, and $\delta_{i2o(0)}(3)$ is $\frac{\sigma_{0,3}}{\sigma_{0,8}} * \alpha_{SGi}(8)$. Vertex 5 is an articulation point on the shortest paths from 0 to 9, hence $\delta_{i2o(0)}(5)$ is $\alpha_{SGi}(5) + \frac{\sigma_{0,5}}{\sigma_{0,9}} * \alpha_{SGi}(9)$. Vertex 2 lies on shortest paths from 0 to 9 and from 0 to 5, hence $\delta_{i2o(0)}(2)$ is $\frac{\sigma_{0,2}}{\sigma_{0,5}} * \delta_{i2o(0)}(5) + \frac{\sigma_{0,2}}{\sigma_{0,7}} * \delta_{i2o(0)}(7)$.

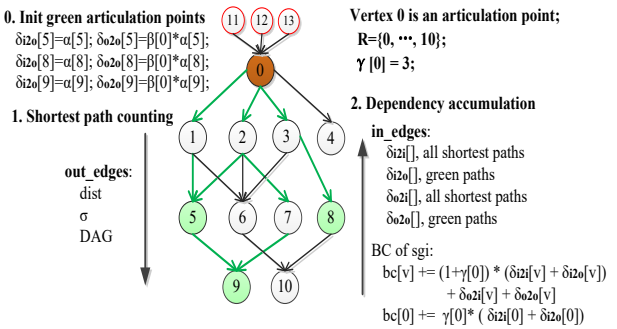
3. ***out2in dependency*** $\delta_{o2i(s)}(v)$. It is defined as if $s \in A_{sgi}$, $\delta_{o2i(s)}(v) = \sum_{t \in SGi} \delta_{o2i(s,t)}(v) = \sum_{t \in SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} * \beta_{SGi}(s)$; otherwise $\delta_{o2i(s)}(v) = 0$. And it satisfies the following relation,

$$\delta_{o2i(s)}(v) = \begin{cases} 0 & \text{if } \mathrm{s} \notin \mathrm{A}_{sgi} \\ \delta_{i2i(s)}(v) * \beta_{SGi}(s) & \text{if } s \in \mathrm{A}_{sgi} \end{cases} \quad \textbf{(5)}$$

$\delta_{o2i(s)}(s)$ is zero. Only articulation point can be a source vertex s to accumulate *out2in dependencies* of s on all other vertices. For example, vertex 0 in Figure4 (b) is an articulation point and has $\delta_{o2i(0)}(v)$; and vertex 0 in Figure4 (a) isn't an articulation point, $\delta_{o2i(0)}(v)$ is zero.



(a) Local DAG rooted at a non-articulation point



(b) Local DAG rooted at an articulation point

Figure 4. Illustration of calculating BC scores of an individual sub-graph in APGRE

4. ***out2out dependency*** $\delta_{o2o(s)}(v)$. It is defined as if $s \in A_{sgi}$, $\delta_{o2o(s)}(v) = \sum_{t \in A_{sgi}} \delta_{o2o(s,t)}(v) = \sum_{t \in A_{sgi}} \frac{\sigma_{st}(v)}{\sigma_{st}} * \beta_{SGi}(s) * \alpha_{SGi}(t)$; otherwise $\delta_{o2o(s)}(v) = 0$. And it obeys the following recursive relation,

$$\delta^{init}_{o2o(s)}(v) = \begin{cases} \beta_{SGi}(s) * \alpha_{SGi}(v) & \text{if } \mathrm{s} \neq v \in \mathrm{A}_{sgi} \\ 0 & \text{if } v \notin \mathrm{A}_{sgi} \end{cases}$$

$$\delta_{o2o(s)}(v) = \begin{cases} \delta^{init}_{o2o(s)}(v) & \text{if s} \in \mathrm{A}_{sgi} \text{ and } succ_s(v) = \emptyset \\ \delta^{init}_{o2o(s)}(v) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \delta_{o2o(s)}(w) & \text{if s} \in \mathrm{A}_{sgi} \text{ and } succ_s(v) \neq \emptyset \\ 0 & \text{if s} \notin \mathrm{A}_{sgi} \end{cases}$$

(6)

$\delta_{o2o(s)}(s)$ is zero. Only articulation points can be a source vertex s to accumulate *out2out dependencies* of s on all other vertices lying on the shortest paths from s to articulation points. For example, vertex 0 in Figure4 (a) is not an articulation point, hence it does not have $\delta_{o2o(0)}(v)$; on the contrary, vertex 0 in Figure4 (b) is an articulation point and it has $\delta_{o2o(0)}(v)$. In Figure4 (b), vertex 0, 5, 8, 9 are articulation points, and vertices on green paths have *out2out dependencies*. For example, vertex 3 lies on the shortest path 0->3->8, and $\delta_{o2o(0)}(3)$ is $\frac{\sigma_{0,3}}{\sigma_{0,8}} * \beta_{SGi}(0) * \alpha_{SGi}(8)$. Vertex 5 is an articulation point on the shortest paths from 0 to 9, and $\delta_{o2o(0)}(5)$ is $\beta_{SGi}(0) * \alpha_{SGi}(5) + \frac{\sigma_{0,5}}{\sigma_{0,9}} * \beta_{SGi}(0) * \alpha_{SGi}(9)$. Vertex 2 lies on the shortest paths from 0 to 9 and from 0 to 5, hence $\delta_{i2o(0)}(2)$ is $\frac{\sigma_{0,2}}{\sigma_{0,5}} * \delta_{o2o(0)}(5) + \frac{\sigma_{0,2}}{\sigma_{0,7}} * \delta_{o2o(0)}(7)$.

3.1.2 BC Scores Based on New Dependencies

We now introduce how to merge these new dependencies to obtain the BC score of a vertex *v*. R_{sgi} and $\gamma(s)$ are used to calculate the BC scores. R_{sgi} is the set of root vertices from which DAGs are constructed, and $\gamma(s)$ indicates the number of dependencies can be derived from the dependency of *s* on all other vertices. In addition, $\gamma_{SGi}(s)$ is the number of vertex *s*' neighbours with no incoming edges and a single outgoing edge in *SGi*. These neighbours are removed from R_{sgi}.

An articulation point a is shared by several sub-graphs, so BC(a) is the sum of its BC scores of sub-graphs. The BC scores of non-articulation points are their sub-graph BC scores. For $s, v \in SGi$, $s \in R_{sgi}$, the BC scores of an individual sub-graph SG_i can be expressed as follows:

$$BC_{SGi}(v) = \sum_{s\neq v, s\in R_{sgi}} ((1+\gamma(s)) * (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v)) + \delta_{o2i(s)}(v) + \delta_{o2o(s)}(v)) + \sum_{s=v, s\in R_{sgi}} \gamma(s) * (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v)) \quad (7)$$

The betweenness centrality of a vertex *v* can be then expressed as follows:

$$BC(v) = \sum_{SGi\in set(SG)} BC_{SGi}(v) \quad (8)$$

For example, in Figure4, since vertices 11, 12 and 13 have one single outgoing edge with no incoming edge, they are removed from R_{sgi}, and $\gamma_{SGi}(0)$ is 3. Vertex 0 in Figur4 (a) is not an articulation point, the dependencies of 0, 11, 12 and 13 on other vertices are $(1+\gamma(0)) * (\delta_{i2i(0)}(v) + \delta_{i2o(0)}(v))$, and the dependencies of 11, 12 and 13 on 0 are $\gamma(0) * (\delta_{i2i(0)}(0) + \delta_{i2o(0)}(0))$, where $\delta_{i2i(0)}(0)$ and $\delta_{i2o(0)}(0)$ are the sum of vertex 0's successors. On the other hand, Vertex 0 in Figur4 (b) is an articulation point, the dependencies of 0, 11, 12 and 13 on other vertices are $(1+\gamma(0)) * (\delta_{i2i(0)}(v) + \delta_{i2o(0)}(v)) + \delta_{o2i(0)}(v) + \delta_{o2o(0)}(v)$, and the dependencies of 11, 12 and 13 on 0 are $\gamma(0) * (\delta_{i2i(0)}(0) + \delta_{i2o(0)}(0))$.

3.1.3 New Algorithm

APGRE leverages articulation points to reduce redundant computations. It uses the four types of dependency to calculate the BC scores of sub-graphs, and reuse the results of those already visited sub-DAGs when computing the BC scores for vertices different from the one currently processed.

Figure5 shows the pseudocode of APGRE algorithm, which includes three steps: 1.decomposing an unweighted graph through articulation points (in line 1 in Figure5); 2.counting the size of a common sub-DAG and the number of DAGs that contain this sub-DAG (loop in line 2 in Figure5); and 3.calculating the BC scores for each sub-graph using four types of dependency in Equation 3-6 and the contribution to BC in Equation 7-8 (loop in line 5 in Figure5). As the sub-graphs are independent, the APGRE algorithm has two-level parallelism on a shared memory platform, i.e., coarse-grained asynchronous parallelism among sub-graphs (loop in line 5 in Figure5) and fine-grained synchronous parallelism among vertices inside one sub-graph (loop in line 6 in Figure5).

3.2 Correctness of APGRE

In this section, we state the main correctness results of the new algorithm APGRE. Theorem 1 is a new BC definition on a decomposed graph, requiring O(|V|^3) time. Theorem 2 is new computation equations, which are four types of dependency, requiring O(|V||E|) time. Theorem 3 is an equation of the contribution to BC scores and is used to reduce total redundancies.

```
Input: G(V,E)
Output: bc[]
1.  decompose G to obtain set(SG), R_sgi , γ_SGi(v)
2.  foreach a ∈ A {
3.     compute α_SGi(a), β_SGi(a)
4.  }
5.  foreach sgi ∈ set(SG) {
6.     foreach s ∈ R_sgi {
7.        forall v ∈ sgi        // Initialize
8.           Initialize σ_st(v),P_s(v), δ_i2i(s)(v), δ_i2o(s)(v),
9.                δ_o2i(s)(v), δ_o2o(s)(v)
10.       forall v ∈ sgi       //Construct shortest-path DAG
11.          compute σ_st(v) and P_s(v)
12.       forall v ∈ D_s       // Traverse DAG backward
13.          compute δ_i2i(s)(v), δ_i2o(s)(v),
14.                  δ_o2i(s)(v), δ_o2o(s)(v)
15.          if v ≠ s then
16.             bc[v] += (1 + γ(s)) * (δ_i2i(s)(v) + δ_i2o(s)(v))
17.                    +δ_o2i(s)(v) + δ_o2o(s)(v)
18.          else
19.             bc[v] += γ(s) * (δ_i2i(s)(v) + δ_i2o(s)(v))
20.    }
21. }
```

Figure 5. Pseudocode for APGRE algorithm

THEOREM 1 (The BC score of a vertex in a decomposed graph). *In an unweighted graph decomposed by articulation points, define the BC score of a vertex v in sub-graph SGi(V, E, A) as, f*or $s, v, t, a \in SGi,\ a, aj, ak \in A_{sgi}$,

$$BC_{SGi}(v) = \sum_{s\neq v\neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s\neq v\neq a} \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} * \alpha_{SGi}(a) + \sum_{a\neq v\neq t} \frac{\sigma_{a,\,t}(v)}{\sigma_{a,t}} * \beta_{SGi}(a) + \sum_{aj\neq v\neq ak} \frac{\sigma_{aj,\,ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$

The betweenness centrality of a vertex v can be then expressed as follows:

$$BC(v) = \sum_{SGi\in Set(SG)} BC_{SGi}(v)$$

Due to lack of space, the proofs are presented in appendix.

THEOREM 2 (The dependency of a decomposed graph). *In an unweighted graph decomposed by articulation points, the BC score of a vertex v in sub-graph SGi can be expressed as follows:*

$$BC_{SGi}(v) = \sum_{s\neq v\in SGi} (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v) + \delta_{o2i(s)}(v) + \delta_{o2o(s)}(v))$$

The four types of the dependency obey recursive relations in equations (3), (4), (5) and (6).

PROOF. Given pairwise distances and the shortest paths counts, the *pair-dependency* is $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$, $\delta_{st}(v) > 0$ only for those $t \in V$ for which *v* lies on at least one shortest path from *s* to *t*; and for two ends of shortest paths, $\delta_{st}(s)$ and $\delta_{st}(t)$ are zero. $\delta_{st}(v,e) = \frac{\sigma_{st}(v,e)}{\sigma_{st}}$, $v \in V$ and $e \in E$, $\sigma_{st}(v,e)$ is the number of the shortest paths from *s* to *t* that contain both *v* and *e*.

Let *w* be any vertex with $v \in P_s(w)$, $P_s(w)$ is a list of immediate predecessors of *w* on the shortest paths from *s* to *w* . Of the shortest paths σ_{sw} from *s* to *w*, σ_{sv} many first go from *s* to *v* and then use *v*->*w*. Consequently, $\frac{\sigma_{sv}}{\sigma_{sw}} * \sigma_{st}(w)$ the shortest paths from *s* to some $t \neq w$ contain *v* and *v*->*w*, that is, $\sigma_{st}(v, v\to w) = \frac{\sigma_{sv}}{\sigma_{sw}} * \sigma_{st}(w)$ and $\sigma_{sw}(v, v\to w) = \sigma_{sv}$. It follows that the pair-dependency of *s* and *t* on *v* and *v*->*w* is

$$\delta_{st}(v, v \to w) = \frac{\sigma_{st}(v, v \to w)}{\sigma_{st}} = \begin{cases} \frac{\sigma_{sv}}{\sigma_{sw}} & \text{if } t = w \\ \frac{\sigma_{sv}}{\sigma_{sw}} * \frac{\sigma_{st}(w)}{\sigma_{st}} & \text{if } t \neq w \end{cases}$$

Inserting this into the below equations,

1. ***in2in***, $s, v, t \in SG$i. For two ends of the shortest paths, $\delta_{st}(s)$ and $\delta_{st}(t)$ are zero. The *in2in dependency* of s on any v in SGi is

$$\delta_{i2i(s)}(v) = \sum_{t \in SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{t \in SGi} \delta_{st}(v)$$
$$= \sum_{t \in SGi} \sum_{w:\, v \in P_s(w)} \delta_{st}(v, v \to w)$$
$$= \sum_{w:\, v \in P_s(w)} \sum_{t \in SGi} \delta_{st}(v, v \to w)$$
$$= \sum_{w:\, v \in P_s(w)} \left(\frac{\sigma_{sv}}{\sigma_{sw}} + \sum_{t \neq w \in SGi} \frac{\sigma_{sv}}{\sigma_{sw}} * \frac{\sigma_{st}(w)}{\sigma_{st}} \right)$$
$$= \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \left(1 + \delta_{i2i(s)}(w)\right)$$

2. ***in2out***, $s, v \in SG$i, $t \notin SGi$, then the paths from v to t must contain an articulation point a which connects SG_i to t. Due to $s \in SG$i and $t \notin SGi$, $\delta_{st}(s) = 0$, $\delta_{st}(a) = 1$ and $\delta_s(a) = \alpha_{SGi}(a)$. The *in2out dependency* of s on any v in SGi is

$$\delta_{i2o(s)}(v) = \sum_{a \in A_{sgi}} \frac{\sigma_{sa}(v)}{\sigma_{sa}} * \alpha_{SGi}(a)$$
$$= \alpha_{SGi}(a) + \sum_{a \neq v, a \in A_{sgi}} \delta_{sa}(v) * \alpha_{SGi}(a)$$
$$= \alpha_{SGi}(a) + \sum_{a \neq v, a \in A_{sgi}} \sum_{w:\, v \in P_s(w)} \delta_{s,a}(v, v \to w) * \alpha_{SGi}(a)$$
$$= \alpha_{SGi}(a) + \sum_{w:\, v \in P_s(w)} \sum_{a \neq v, a \in A_{sgi}} \frac{\sigma_{sv}}{\sigma_{sw}} * \frac{\sigma_{s,a}(w)}{\sigma_{s,a}} * \alpha_{SGi}(a)$$
$$= \alpha_{SGi}(a) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \sum_{a \in A_{sgi}} \frac{\sigma_{s,a}(w)}{\sigma_{s,a}} * \alpha_{SGi}(a)$$
$$= \alpha_{SGi}(a) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \sum_{a \in A_{sgi}} \frac{\sigma_{s,a}(w)}{\sigma_{s,a}} * \alpha_{SGi}(a) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \sum_{t \notin A_{sgi}} \frac{\sigma_{s,t}(w)}{\sigma_{s,t}} * 0$$

This can deduce that $\delta_{i2o(s)}(v)$ satisfies the following recursive relation,

$$\delta^{init}_{i2o(s)}(v) = \begin{cases} \alpha_{SGi}(v) & \text{if } v \in A_{sgi} \\ 0 & \text{if } v \notin A_{sgi} \end{cases}$$

$$\delta_{i2o(s)}(v) = \begin{cases} \delta^{init}_{i2o(s)}(v) & \text{if } succ_s(v) = \emptyset \\ \delta^{init}_{i2o(s)}(v) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \delta_{i2o(s)}(w) & \text{if } succ_s(v) \neq \emptyset \end{cases}$$

3. ***out2in***, $v, t \in SG$i, $s \notin SGi$, then the paths from s to v must contain an articulation point a which connects s to SG_i. As $\delta_{st}(a)$ has computed in $\delta_{i2o(s)}(a)$, $\delta_{st}(t)$and $\delta_{st}(a)$ are zero. The *out2in dependency* of s on any v in SGi is

$$\delta_{o2i(a)}(v) = \sum_{t \in SGi} \frac{\sigma_{at}(v)}{\sigma_{at}} * \beta_{SGi}(a)$$
$$= \sum_{t \in SGi} \delta_{at}(v) * \beta_{SGi}(a)$$
$$= \sum_{t \in SGi} \sum_{w:\, v \in P_a(w)} \delta_{at}(v, v \to w) * \beta_{SGi}(a)$$
$$= \sum_{w:\, v \in P_a(w)} \sum_{t \in SGi} \delta_{at}(v, v \to w) * \beta_{SGi}(a)$$
$$= \sum_{w:\, v \in P_a(w)} \left(\frac{\sigma_{av}}{\sigma_{aw}} + \sum_{t \neq w \in SGi} \frac{\sigma_{av}}{\sigma_{aw}} * \frac{\sigma_{at}(w)}{\sigma_{at}} \right) * \beta_{SGi}(a)$$
$$= \sum_{w:\, v \in P_a(w)} \frac{\sigma_{av}}{\sigma_{aw}} * (1 + \delta_{i2i(a)}(w)) * \beta_{SGi}(a)$$
$$= \delta_{i2i(a)}(v) * \beta_{SGi}(a)$$

If $s \notin A_{sgi}$, $\delta_{o2i(s)}(v)$ is zero.

4. ***out2out***, $v \in SG$i, $s, t \notin SGi$, they are in three different sub-graphs, and must be connected by two articulation points aj and ak in SG_i. v is in SGi, s in SGj, and t in SGk. $\delta_{st}(aj)$ is zero, $\delta_{st}(ak) = \beta_{SGi}(aj) * \alpha_{SGi}(ak)$. The *out2out dependency* of s on any v in SGi is

$$\delta_{o2o(aj)}(v) = \sum_{ak \in A_{sgi}} \frac{\sigma_{aj,\,ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$
$$= \beta_{SGi}(aj) * \alpha_{SGi}(ak) + \sum_{ak \neq v, ak \in A_{sgi}} \delta_{aj,ak}(v) * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$
$$= \beta_{SGi}(aj) * \alpha_{SGi}(ak) + \sum_{ak \neq v, ak \in A_{sgi}} \sum_{w:\, v \in P_s(w)} \delta_{aj,ak}(v, v \to w) \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$
$$= \beta_{SGi}(aj) * \alpha_{SGi}(ak) + \sum_{w:\, v \in P_{aj}(w)} \sum_{ak \neq v, ak \in A_{sgi}} \frac{\sigma_{aj,v}}{\sigma_{aj,w}} \frac{\sigma_{aj,ak}(w)}{\sigma_{aj,ak}} \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$
$$= \beta_{SGi}(aj) * \alpha_{SGi}(ak) + \sum_{w:\, v \in P_{aj}(w)} \frac{\sigma_{aj,v}}{\sigma_{aj,w}} * \sum_{ak \in A_{sgi}} \frac{\sigma_{aj,ak}(w)}{\sigma_{aj,ak}} \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$
$$= \beta_{SGi}(aj) * \alpha_{SGi}(ak) + \sum_{w:\, v \in P_{aj}(w)} \frac{\sigma_{aj,v}}{\sigma_{aj,w}} * \sum_{ak \in A_{sgi}} \frac{\sigma_{aj,ak}(w)}{\sigma_{aj,ak}} \beta_{SGi}(aj) * \alpha_{SGi}(ak) + \sum_{w:\, v \in P_{aj}(w)} \frac{\sigma_{aj,v}}{\sigma_{aj,w}} * \sum_{t \notin A_{sgi}} \frac{\sigma_{aj,t}(w)}{\sigma_{aj,t}} * 0$$

From this we can deduce that $\delta_{o2o(s)}(v)$ satisfies the following recursive relation,

$$\delta^{init}_{o2o(s)}(v) = \begin{cases} \beta_{SGi}(s) * \alpha_{SGi}(v) & \text{if } s \neq v \in A_{sgi} \\ 0 & \text{if } v \notin A_{sgi} \end{cases}$$

$$\delta_{o2o(s)}(v) = \begin{cases} \delta^{init}_{o2o(s)}(v) & \text{if } s \in A_{sgi} \text{ and } succ_s(v) = \emptyset \\ \delta^{init}_{o2o(s)}(v) + \sum_{w:\, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} * \delta_{o2o(s)}(w) & \text{if } s \in A_{sgi} \text{ and } succ_s(v) \neq \emptyset \\ 0 & \text{if } s \notin A_{sgi} \end{cases}$$

Now, we can use these equations and Theorem1 to calculate the BC scores of SGi,

$$BC_{SGi}(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s \neq v \neq a} \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} * \alpha_{SGi}(a) +$$
$$\sum_{a \neq v \neq t} \frac{\sigma_{a,\,t}(v)}{\sigma_{a,t}} * \beta_{SGi}(a) + \sum_{aj \neq v \neq ak} \frac{\sigma_{aj,\,ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$$
$$= \sum_{s \neq v \in SGi} \left(\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v) + \delta_{o2i(s)}(v) + \delta_{o2o(s)}(v) \right)$$

□

THEOREM 3 *In an unweighted graph decomposed by articulation points, the BC score of a vertex v in sub-graph SGi can be expressed as follows:*

$$BC_{SGi}(v) = \sum_{s \neq v, s \in R_{sgi}} ((1+\gamma(s)) * (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v)) + \delta_{o2i(s)}(v) + \delta_{o2o(s)}(v)) + \sum_{s=v, s \in R_{sgi}} \gamma(s) * (\delta_{i2i(s)}(v) + \delta_{i2o(s)}(v))$$

PROOF. Let vertex u be a vertex with no incoming edges and a single outgoing edge, there is the edge u->s, then the DAG D_s rooted at s is the sub-DAG of DAG D_u rooted at u. And then if $u \neq s \neq v$, $\delta_u(v) = \delta_s(v)$; if $u \neq s = v$, $\delta_u(s) = \delta_s(s)$, where $\delta_s(s)$ is the sum of s' successors. Consequently, when traversing D_s, we can obtain $\delta_u(v)$ and $\delta_s(v)$ at the same time. $\gamma_{SGi}(s)$ is the number of u. As u is not an articulation point, u has $\delta_{i2i(u)}$ and $\delta_{i2o(u)}$, and $\gamma_{SGi}(s)$ can be applied to $\delta_{i2i(s)}$ and $\delta_{i2o(s)}$.

□

4. Implementation

The articulation-point-guided redundancy elimination (APGRE) for BC algorithm leverages the articulation points and reuses the results of the common sub-DAGs in calculating the BC scores, which eliminates redundant computations. As the sub-graphs are independent, we implement the APGRE algorithm using two-level parallelism on a shared memory platform, i.e., coarse-grained asynchronous parallelism among sub-graphs and fine-grained synchronous parallelism among vertices inside one sub-graph.

The pseudocode for APGRE algorithm is shown in Figure5. APGRE includes three steps, decomposing an unweighted graph through articulation points, counting $\alpha_{SGi}(a)$ and $\beta_{SGi}(a)$ for each articulation points and calculating the BC scores for each sub-graph using the four types of dependency in Equation 3-6 and the merging Equation 7-8. The first step uses Tarjan's algorithm in O(|V|+|E|) time to find all articulation points. The second step uses parallel BFS to count $\alpha_{SGi}(a)$ and $\beta_{SGi}(a)$ for each articulation points. For unweighted graphs, $\alpha_{SGi}(a)$ is the number of vertices which a can reach without passing through *SGi* in G, and can be obtained by BFS; $\beta_{SGi}(a)$ is the number of vertices which can reach a (or which a is reachable from) without passing through *SGi* in G, and can be obtained by reverse BFS from a. The third step uses the four types of dependency to compute BC scores in O(|topSG'V||topSG'E|) time by using two-level parallelism.

Algorithm 1 shows graph partition through articulation points. This partition should effectively recognize common sub-DAGs, merge small adjacent sub-graphs for large granularity, and minimize the amount of articulation points. GraphPartition finds all articulation points and all biconnected components (BCC) in a given graph, and then begins to build sub-graphs starting from the maximal biconnected component, called *topBCC*. FINDBCC() function on Line2 finds all biconnected components and all articulation points using Tarjan's algorithm [32], requiring O(|V|+|E|) time. The graph is decomposed into a tree of biconnected components. Sub-graph construction uses depth-first search starting from *topBCC* (Line 4 ~Line 24 in Algorithm 1). If the size of a BCC is less than a threshold, and its father BCC is not *topBCC*, it is merged into its father BCC. If a BCC has only two vertices, and its father BCC is *topBCC*, it is merged into its father BCC. Otherwise, the BCC is built into a sub-graph. BUILDSUBGRAPH() function in Line24 will set γ_{SGi}[] and R_{SGi}[]. If a vertex has a single outgoing edge and no incoming edges, it's removed from R_{SGi}[], and its neighbor's γ_{SGi}[] is increased by one.

Algorithm 2 and Figure4 shows the calculation of BC scores for a sub-graph with fine-grained synchronous parallelism. For each vertex v in sub-graph *SGi*, we maintain a number of variables:

- *dist[v]*, the shortest path distance from root *s*
- $\sigma[v]$, the number of shortest paths from root *s*

Algorithm 1. GRAPHPARTITION(G) ---- *graph decomposed by articulation points*

```
Input: G(V, E)
Output: gG(set(SG))
1.  undirectedG = GETUNDG( G)
2.  myBCC = FINDBCC(undirectedG) //find biconnected components
3.
4.  visited[] = {-1, …, -1}
5.  visited[topBcc] = 1
6.  PUSH(Stack, topBcc)
7.  while Stack != empty do
8.   TOP(Stack, currBcc)
9.   flag = 0
10.  for all neighbor nextBcc of currBcc do
11.   if visited[nextBcc] == -1 then
12.    PUSH(Stack, nextBcc)
13.    visited[nextBcc] = 1
14.    flag = 1
15.    break
16.  if flag == 0 then
17.   POP( Stack, currBcc)
18.   TOP( Stack, prevBcc)
19.   if prevBcc != topBcc && SIZE(currBcc) < THREHOLD
    then
20.    VSet[prevBcc] += VSet[currBcc]
21.   else if prevBcc == topBcc && SIZE(currBcc) <= 2 then
22.    VSet[prevBcc] += VSet[currBcc]
23.   else
24.    BUILDSUBGRAPH(currBcc,G,gG) //set γ_SGi[] and R_SGi[]
25.
26. //last sub-graph for unconnected vertexes
27. sgVSet = NULL
28. for all i ∈ all Bcc do
29.  if visited[i] == -1 then
30.   sgVSet + = VSet[i]
31. if sgVSet != NULL then
32.  BUILDSUBGRAPH(sgVSet, G, gG)
33. return gG
```

- $\delta_{i2i(s)}(v)$, the *in2in dependency* of root *s* on any a vertex *v*
- $\delta_{i2o(s)}(v)$, the *in2out dependency* of root *s* on any a vertex *v*
- $\delta_{o2i(s)}(v)$, the *out2in dependency* of root *s* on any a vertex *v*
- $\delta_{o2o(s)}(v)$, the *out2out dependency* of root *s* on any a vertex *v*
- $\gamma_{SGi}[s]$, the number of total redundant DAGs

Algorithm 2 includes the following three phases.

1) Initializing these dependencies (Line 10 ~ Line 18 in Algorithm 2): when source vertex *s* is an articulation point, set *sizeO2I* as $\beta_{SGi}[s]$, and set all articulation points' $\delta_{o2o(s)}$[i] as $\beta_{SGi}[s]$* $\alpha_{SGi}[i]$. For any source vertex *s*, set all articulation points' $\delta_{i2o(s)}[i]$ as $\alpha_{SGi}[i]$ and set all vertices' $\delta_{i2i(s)}$[] as zero.

2) Computing the shortest paths (Line 20 ~ Line 33 in Algorithm 2): counting σ[] and *dist*[] use a forward BFS from a source vertex *s*. All vertices at level *i* are processed in parallel, and Levels[] is a buckets to save the vertices of each level.

3) Accumulating dependencies and contributing to BC scores (Line 35 ~ Line 49 in Algorithm 2): we use successor method to parallelize this phase, which accumulates these dependencies of vertices within level *i* and updates the BC scores of vertices within level *i* in parallel. The four dependencies are calculated by new computation equations in section 3.1.1, and the contribution to BC scores is used by merge equation in section 3.1.2.

Algorithm 2: BCinSG(sgid, gG) ---- *BC calculation in a sub-graph*

Input: sgid, gG(set(SG))
Output: $bc_{SGi}[]$

```
1.  bc_SGi[] = {0.0, …, 0.0}
2.  for all s ∈ R_SGi do
3.    // clean
4.    σ[] = {0, …, 0}
5.    δ_i2i[] = {0, …, 0}
6.    δ_i2o[] = {0, …, 0}
7.    δ_o2o[] = {0, …, 0}
8.    dist[] = {-1, …, -1}
9.    levels[] = empty multiset
10. Phase0: initialize these dependencies
11.   sizeO2I = 0
12.   if s == Articulation-point then
13.     sizeO2I = β_SGi[s]
14.     for all i ∈ A_SGi && i != s do
15.       δ_o2o[i] = β_SGi[s] * α_SGi[i]
16.   for all i ∈ A_SGi && i != s do
17.     δ_i2o[i] = α_SGi[i]
18.
19. Phase1: compute the number of shortest paths in the forward traversal
20.   σ[s] = 1
21.   dist[s] = 0
22.   currLevel = 0
23.   PUSH(Levels[currLevel], s)
24.   while SIZE(Levels[currLevel]) != 0 do
25.     for all v ∈ Levels[currLevel] in parallel do
26.       for all neighbor w of v in parallel do
27.         if dist[w] < 0 then
28.           PUSH(Levels[currLevel + 1], w)
29.           dist[w] = dist[v] + 1
30.         if dist[w] == dist[v] + 1 then
31.           σ[w] += σ[v]
32.       currLevel = currLevel + 1
33.
34. Phase2: accumulate dependency and bc in the backward traversal
35.   currLevel = currLevel -1
36.   while currLevel >= 0 do
37.     for all v ∈ Levels[currLevel] in parallel do
38.       for all neighbor w of v do
39.         if dist[w] == dist[v] + 1 then
40.           δ_i2i[v] += σ[v]/σ[w] * ( 1 + δ_i2i[w])
41.           δ_i2o[v] += σ[v]/σ[w] * δ_i2o[w]
42.           δ_o2o[v] += σ[v]/σ[w] * δ_o2o[w]
43.       if v != s then
44.         bc_SGi[v] += (1+γ_SGi[s]) * (δ_i2i[v] + δ_i2o[v])
45.                      + sizeO2I*δ_i2i[v]+ δ_o2o[v]
46.       else
47.         bc_SGi[v] += γ_SGi[s] * (δ_i2i[v] + δ_i2o[v])
48.     currLevel = currLevel – 1
49.
50. return bc_SGi[]
```

5. Experimental Evaluation

In this section, we compared the performance of our APGRE algorithm against a number of BC algorithms for unweighted graphs on multi-core machines.

5.1 Experimental Methodology

Our algorithm is implemented in C++ and CilkPlus, and the graphs are stored in Compressed Sparse Row (CSR) format. We compared its performance against a number of publicly available versions of BC algorithms for unweighted graphs. The compiler used was GCC 4.8.1. We report results from two multicore server systems, a 6-core 2.4GHz Intel Xeon E5645 with 12 hyper threads processor and 32GB of DRAM, and a four 8-core 2.13GHz Intel Xeon E7-8830 processors with 256GB memory. The 6-core system is used for most of the evaluation. And the four 8-core system is used to evaluate the parallel scalability of our approach up to 32 threads.

Table1 summarizes the graph inputs we used. These graphs were taken from the Stanford Network Analysis Platform (SNAP) [36], the DIMACS shortest paths challenge [18], and a variety of web crawls [17]. These benchmarks are real-world instances of graphs that correspond to a wide variety of practical applications and network structures.

We evaluated a number of BC algorithms for unweighted graphs. We describe each of them and give details about the parallelization strategy as below.

- **APGRE** Our algorithm is a two-level parallelization, a sub-graph-level asynchronous among sub-graphs, and a fine-grained level synchronous in a sub-graph. The implementation uses cilk_for to parallelize the loop of a sub-graph-level and uses reduction bag [40] to parallelize BFS in a sub-graph.
- **preds/preds-serial** This algorithm is a fine-grained level-synchronous parallelization presented in [12]. The implementation is part of the SSCA v2.2 benchmark. It also serves as the serial baseline that we compare all algorithms against.
- **succs** This algorithm uses successors instead of predecessors to eliminate locks of the second phase. It is a fine-grained, level synchronous parallelization presented in [13].
- **lockSyncFree** This algorithm is a fine-grained parallel approach without lock synchronization[14]. The implementation uses OpenMP directives to parallelize the loops.
- **async** This algorithm is a fine-grained asynchronous parallelization presented in [11]. The algorithm is implemented in the Galois system and is able to extract large amounts of parallelism. This version only deals with undirected graphs.
- **hybrid** This algorithms is a fine-grained level synchronous parallelization presented in [25], and use a hybrid BFS which combines a top-down algorithm along with a bottom-up algorithm to reduce the number of edges examined. This algorithm is implemented in Ligra system.

To benchmark performance, we present the running time and the search rate (or the traversal rate) in Millions of Edges Traversed per Second (MTEPS) of all algorithms. For the exact computation of betweenness centrality, the number of TEPS has been defined as [35], $TEPS_{BC} = \frac{nm}{t}$. In this equation, n is the number of vertices, m is the number of edges, and t is execution time of the BC computation.

5.2 Algorithm Efficiency

We present the running time of all algorithms for the exact BC computation in Table2, the speedup in Figure6 and the search rate MTEPS in Table3. Our *APGRE* algorithm achieves large speedups across all of the graphs：the speedups range from 1.59x to 32.83x across a variety of real-world graphs over the best previous shared memory implementation, with an average speedup of 4.6x.

We can see that the *APGRE* algorithm achieves approximately 45 ~ 2400 MTEPS for all of these graphs, whereas other algo-

Table 1. Real-world graphs used for evaluation

Graph	Description	#Vertices	#Edges	Directed
Email-Enron	Enron email network	36,692	367,662	N
Email-EuAll	Email network of a large European Research Institution	265,214	420,045	Y
Slashdot0811	Slashdot Zoo social network	77,360	905,468	Y
soc-DouBan	DouBan Chinese social network	154,908	654,188	Y
WikiTalk	Communication network of Wikipedia	2,394,385	5,021,410	Y
dblp-2010	DBLP collaboration network	326,186	1,615,400	Y
com-youtube	Youtube online social network	1,134,890	5,975,248	N
NotroDame	University of Notre Dame web graph	325,729	1,497,134	Y
web-BerkStan	Berkely-Stanford web graph from 2002	685,230	7,600,595	Y
web-Google	Webgraph from the Google programming contest	875,713	5,105,039	Y
USA-roadNY	Road network	264,346	733,846	N
USA-roadBAY	Road network	321,270	800,172	N

Table 2. Performance in execution time (**second**) on 12-threads

Graph	*serial*	*APGRE*	*preds*	*succs*	*lockSyncFree*	*async*	*hybrid*
Email-Enron	130	**46**	133	102	180	84	100
Email-EuAll	1826	**53**	844	1203	1145	-	1501
Slashdot0811	846	**246**	587	496	-	-	348
soc-DouBan	1993	**182**	1458	1434	2430	-	979
WikiTalk	90496	**4931**	32298	-	29527	-	281931
dblp-2010	8015	**988**	5518	3261	7136	-	5948
com-youtube	219925	**19258**	-	108860	-	52261	71747
NotroDame	1198	**291**	10237	1347	1606	-	4341
web-BerkStan	31099	**7929**	-	17824	38709	-	54839
web-Google	69744	**11883**	60856	24856	49729	-	58534
USA-roadNY	6788	**4213**	-	5595	-	7159	24164
USA-roadBAY	10450	**4951**	-	7284	-	10585	27163
Average speedup = algorithm/serial	1	**8.9x**	1.43x	1.68x	1.28x	1.93x	1.2x

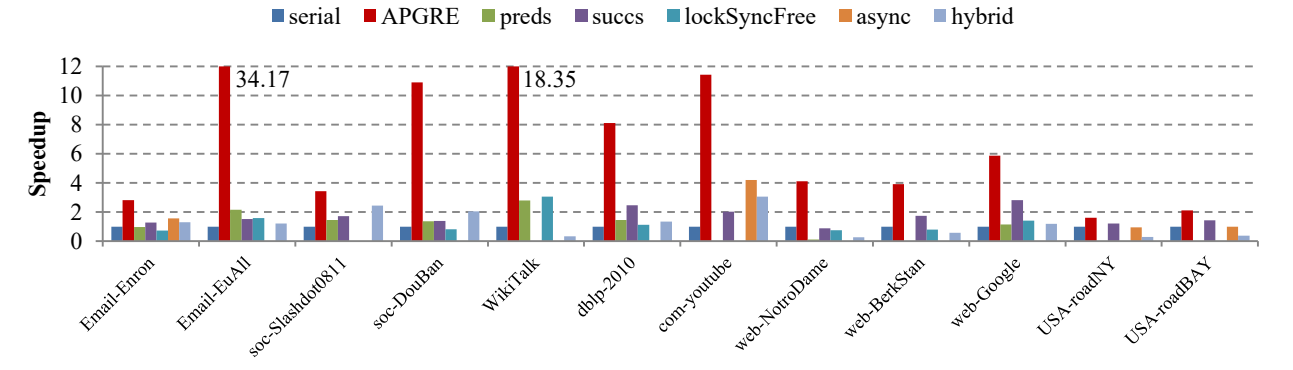


Figure 6. Speedups on the 12 threads relative to *serial*.

rithms have 8 ~ 400 MTEPS. The TEPS metric is a standardized performance metric, and is similar in spirit to measuring MFLOPS for optimized matrix multiplication in terms of the classic $O(N^3)$ algorithm. A *sampling* approach of BC is the highest published performance for GPU [34], and its search rate for single GPU have approximately 40 ~ 400 MTEPS for various unweighted graphs. Our *APGRE* algorithm on multicore machines overcomes the *sampling* on GPU.

5.3 Explaining the Performance Improvement

The speedup of the APGRE approach demonstrated in Figure6 is due to partial and total redundancy elimination (Figure7). Brandes' algorithm constructs DAGs for each vertex, and traverses these DAGs in backward (in non-increasing distance order) to accumulate the dependency values and update the BC scores. As shown in Figure7, partial redundancy coming from common sub-DAGs exists the BC calculation of various graphs, e.g., 80% in WikiTalk, 64% in web-NotrdDame, 49% in dblp-2010, 35% in soc-Slashdot0811. And total redundancy coming from leaf vertices with one-degree exists the BC calculation of these benchmarks except soc-Slashdot0811, e.g., 71% in Email-EuAll, 67% in soc-DouBan, 53% in com-youtube, 31% in Email-Enron. Even in road graphs, whose degree distributions are not power-law distribution, there are also redundant computation, e.g., 5% partial redundancy and 16% total redundancy in USA-roadNY, 13% partial redundancy and 23% total redundancy in USA-roadBAY. Our APGRE can eliminate these partial and total redundant computations to accelerate BC calculation.

APGRE introduces extra computations to eliminate these redundancies; the extra computations include graph partition and

counting $\alpha_{SGi}(a)$ and $\beta_{SGi}(a)$ of articulation points. Figure8 is execution time breakdown of APGRE, the extra computations take 25.7%, 23%, 20.9%, 19.8%, 1.59% and 1.87% of the total execution time in com-youtube, dblp-2010, soc-DouBan, web-NotroName, web-BerkStan, USA-roadNY, respectively, using 12 threads. The BC calculation of the top sub-graph is the majority of the total execution time, because the top sub-graph is larger than other sub-graphs (Table 4).

5.4 Algorithm scalability

We evaluate the scalability of our APGRE algorithm on the 6-core system and the four 8-core system. Our approach observes parallel speedup with each additional core (Figure9 and Figur10). Our algorithm is a two-level parallelization, a sub-graph-level asynchronous among sub-graphs, and a fine-grained level synchronous in a sub-graph. Two-level parallelism makes it possible to extract large amounts of parallelism. But in these benchmarks, the scalability of APGRE does come from fine-grained level synchronous in the top sub-graph. The top sub-graph is larger than other sub-graphs (Table 4), and the BC calculation of the top sub-graph is the majority of the total execution time (Figure 8).

Though APGRE is better than the other algorithms on the 6-core SMT system, Figure 9 shows that it loses the scaling beyond 6 cores. Our studies indicate that SMT limits performance. When we run two threads on two physical cores, 1.62x speedup is obtained over one thread. But when we run two threads on one physical core, which uses SMT technique, 1.18x speedup is obtained over one thread. The Linux scheduler will not allocate two threads on one physical core unless there is no other physical core available. This experiment suggests that SMT limits the performance and scalability of the parallel execution of APGRE. And on multi-socket systems, many studies show that NUMA architectures and memory system may limit performance of graph applications [37][38][39][40][41][42][43].

6. Related Work

Parallel algorithms: a number of parallel BC algorithms are developed based on Brandes' algorithm. Bader et al. [12] present

Table 3. Performance in search rate (**MTEPS**, Millions of Traversed Edges per Second) on 12-threads

Graph	*serial*	*APGRE*	*preds*	*succs*	*lockSyncFree*	*async*	*hybrid*
Email-Enron	103.52	291.02	100.83	131.14	74.85	161.52	134.67
Email-EuAll	60.99	2084.03	131.92	92.59	97.24		74.19
Slashdot0811	82.75	284.59	119.23	141.25	-		201.12
soc-DouBan	50.84	554.25	69.50	70.66	41.69		103.51
WikiTalk	132.86	2437.89	372.26	-	407.19		42.65
dblp-2010	65.74	532.92	95.49	161.59	73.84		88.58
com-youtube	30.83	352.13	-	62.29	-	129.76	94.52
NotroDame	406.83	1673.36	47.63	361.87	303.59		112.34
web-BerkStan	167.47	656.81	-	292.20	134.54		94.97
web-Google	64.10	376.21	73.46	179.85	89.90		76.37
USA-roadNY	28.58	46.04	-	34.67	-	27.10	8.03
USA-roadBAY	24.60	51.92	-	35.29	-	24.29	9.46
Average speedup = algorithm/serial	101.59	778.45	126.29	142.13	152.86	85.67	86.70

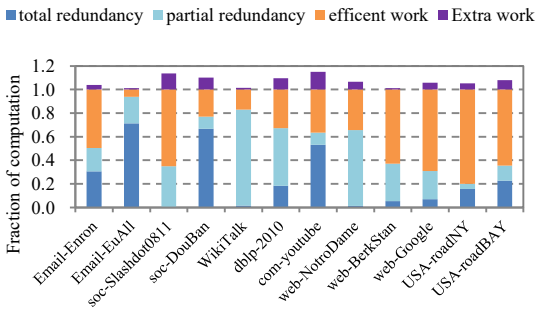


Figure 7. Breakdown of BC computation

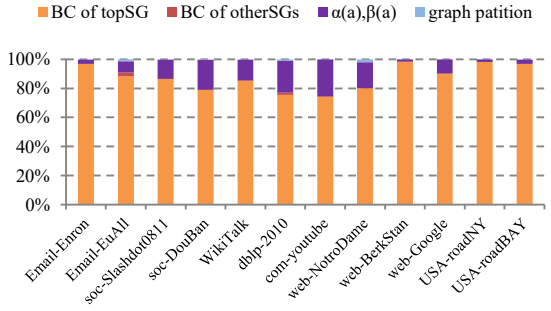


Figure 8. Breakdown of execution time of APGRE on 12-threads

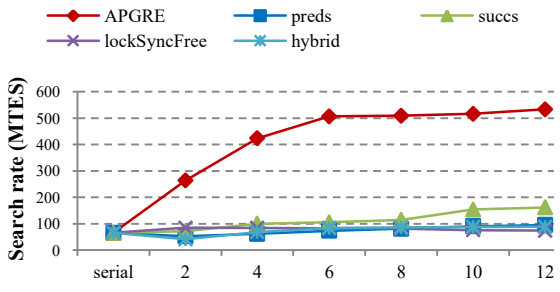


Figure 9. Parallel scaling of all algorithms on the 6-core system for dblp-2010 (326,186 vertices and 1,615,400 edges)

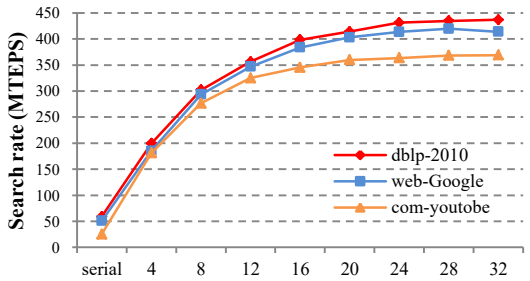


Figure 10. Parallel scaling of APGRE on the four 8-core system

the first parallel implementation for computing BC, which exploits fine-grained and coarse-grained parallelism. Madduri et al [13] use successors instead of predecessors to improve the algorithm. Oded Green et al. [16] use neighbour traversal operations instead of ancestor-list data structure to improve the space of the algorithm. Tan et al. [14][15] use graph partition approach to eliminate access conflicts. Cong et al. [24] perform prefetching and appropriate re-layout of the graph nodes to improve locality. Prountzos et al. [11] present an asynchronous parallel algorithm which can extract large amounts of parallelism through a fast global scheduler. They relax the order in computing BC and focus on global scheduling on shared-memory multicores. Shun et al. [33] use hybrid BFS approach to compute BC, which combine a top-down BFS algorithm and a bottom-up BFS algorithm on a multi-socket server. McLaughlin et al. [34] present a work-efficient approach which combines vertex-parallel and edge-parallel on GPU. Edmonds et al. [22] present a space-efficient distributed parallel algorithm for computing BC. This work breaks the shortest paths counting into three sub-phases to deal with weighted and unweighted graphs. Buluc and Gibert [23] use algebraic computation to compute BC and use MPI to exploit coarse-grained parallelism. All the works are based on Brandes' algorithm, and have redundancies on real-world graphs.

Approximation algorithms: a number of approximation algorithms have been proposed [19] [20][21] to reduce the amount of computation. They perform the shortest path computations for only a subset of vertices. In this paper, we focus on exact algorithms, and present APGRE approach to eliminate partial and total redundancy of exact algorithm.

Graph-parallel processing framework for shared memory: a number of graph-parallel processing frameworks have been proposed for supporting shared memory [25][26][27][28][37]. In our evaluation, *async* algorithm [11] is implemented in the Galois system, and *hybrid* algorithm [25] is implemented in Ligra system.

7. Conclusion

This paper presents a redundancy elimination approach, which identifies the common sub-DAGs shared between multiple DAGs rooted at different vertices using articulation points and calculates the BC scores using four types of dependency, thus our approach can reuse the results in traversing the common sub-DAGs and eliminate redundant computations. APGRE uses graph decomposition by articulation points to recognize common sub-DAGs, counts the size of common sub-DAGs and the number of DAGs that share this common sub-DAG, and calculates the BC scores for each sub-graph using four types of dependency. We implemented our approach with two-level parallelism and evaluated our approach on a multicore platform, achieving an average speedup of 4.6x across a variety of real-world graphs over the best previous shared memory implementation, with traversal rates up to 45 ~ 2400 MTEPS.

Acknowledgments

We thank all the reviewers for their valuable comments and suggestions. This work was supported in part by the National High Technology Research and Development Program of China (2015AA011505), the National Natural Science Foundation of China (61402445, 61303053, 61202055, 61221062, 61502452).

Appendix

PROOF of theorem 1. Shortest paths have two properties:

- If a vertex $v \in V$ lies on the shortest path between vertices $s, t \in V$, then $\sigma_{st}(v) = \sigma_{sv} * \sigma_{vt}$.[10]
- In an unweighted graph, if *SGi* connects *SGj* through an articulation point *a*, *a* must lie on all the shortest paths between vertices in *SGi* and vertices in *SGj*.

Recall that the *betweenness centrality* of a vertex *v* is defined as: $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$. To calculate the BC score of a vertex *v*, the source *s* and the target *t* can be categorized into four classes:

1. ***in2in***, *s* and *t* are both in the same sub-graph of *v*, and then the *in2in BC score* of vertex *v* is $\sum_{s \neq v \neq t \in SGi} \frac{\sigma_{st}(v)}{\sigma_{st}}$.
2. ***in2out***, *s* is in the same sub-graph of *v*, but *t* is not, then the paths from *v* to *t* must contain an articulation point *a* which connects the sub-graph to *t*, and then $\sigma_{st} = \sigma_{sa} * \sigma_{at}$ and $\sigma_{st}(v) = \sigma_{sa}(v) * \sigma_{at}$. And thus $\frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{\sigma_{sa}(v) * \sigma_{at}}{\sigma_{s,a} * \sigma_{a,t}} = \frac{\sigma_{s,a}(v)}{\sigma_{s,a}}$.

   Further for all $t \in (G - SGi)$ which *v* can reach via *a*, there are $\sum_{t \in G-SGi \text{ via } a} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{t \in G-SGi \text{ via } a} \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} = \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} * \alpha_{SGi}(a)$; and further for all $t \in (G - SGi)$ which *v* can reach via A_{sgi}, there are $\sum_{t \in G-SGi \text{ via } A_{sgi}} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{a \in A_{sgi}} \sum_{t \in G-SGi \text{ via } a} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{a \in A_{sgi}} \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} * \alpha_{SGi}(a)$.

   Therefore, the *in2out BC score* of *v* is computed as follows: $\sum_{s \neq v \in SGi, t \in G-SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{s \neq v \neq a \in SGi, a \in A_{sgi}} \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} * \alpha_{SGi}(a)$.
3. ***out2in***, $v, t \in SG\text{i}, s \notin SGi$, then the paths from *s* to *v* must contain an articulation point *a* which connects *s* to SG_i. This case is similar with *in2out*, the *out2in BC score* of *v* is $\sum_{s \in G-SGi, v \neq t \in SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{a \neq v \neq t \in SGi, a \in A_{sgi}} \frac{\sigma_{a,t}(v)}{\sigma_{a,t}} * \beta_{SGi}(a)$.
4. ***out2out***, $v \in SG\text{i}, s, t \notin SGi$, they are in three different sub-graphs, and must be connected by two articulation points *aj* and *ak* in SG_i. There are $\sigma_{st} = \sigma_{s,aj} * \sigma_{aj,ak} * \sigma_{ak,t}$ and $\sigma_{st}(v) =$

Table 4. The size of sub-graphs for various graphs

Graph	#*SG*	Top sub-graph				Second sub-graph				Third sub-graph	
		#V	#E	V/G.V	E/G.E	#V	#E	V/G.V	E/G.E	#V	#E
Email-Enron	706	20,416	326,964	55.64%	88.93%	2,996	6,040	8.17%	1.64%	1,229	2,518
Email-EuAll	1,559	36,395	188,490	13.72%	44.87%	40,382	24,775	13.72%	44.87%	8,649	8,925
Slashdot0811	5,314	54,207	840,198	70.07%	92.79%	268	800	0.35%	0.09%	246	734
soc-DouBan	7,801	52,061	448,474	33.61%	68.55%	56	110	0.04%	0.02%	51	100
WikiTalk	18,672	634,355	3,247,833	26.49%	64.68%	53,814	53,813	2.25%	1.07%	15,890	16,084
dblp-2010	15,637	148,485	1,161,496	45.52%	71.90%	99,773	182,480	30.59%	11.30%	83	290
com-youtube	84,767	522,582	4,731,188	46.05%	79.18%	7,614	15,274	0.67%	0.26%	4,179	8,434
NotroDame	10,731	140,317	1,198,268	43.08%	80.04%	5,329	5,332	1.64%	0.36%	2,635	13,940
web-BerkStan	4,273	491,370	6,664,542	71.71%	87.68%	30,448	101,170	4.44%	1.33%	17,664	94,723
web-Google	25,815	661,197	4,682,434	75.50%	91.72%	19,911	38,197	2.27%	0.75%	661	896
usa-roadNY	7,034	232,844	665,946	88.08%	90.75%	1,155	3,352	0.44%	0.46%	183	444
usa-roadBAY	12,717	252,897	654,710	78.72%	81.82%	251	564	0.08%	0.07%	244	572

$\sigma_{s,aj} * \sigma_{aj,ak}(v) * \sigma_{ak,t}$, and thus $\frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{\sigma_{s,aj} * \sigma_{aj,ak}(v) * \sigma_{ak,t}}{\sigma_{s,aj} * \sigma_{aj,ak} * \sigma_{ak,t}} = \frac{\sigma_{aj,ak}(v)}{\sigma_{aj,ak}}$.

Further for all s reaching v via aj and all t which v can reach via ak, there are

$\sum_{s \in G-SGi \text{ via } aj,\ t \in G-SGi \text{ via } ak} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{s \in G-SGi \text{ via } aj,\ t \in G-SGi \text{ via } ak} \frac{\sigma_{aj,ak}(v)}{\sigma_{aj,ak}} = \frac{\sigma_{aj,ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$.

And further for all $s \in (G - SGi)$ reaching v via A_{sgi} and all $t \in (G - SGi)$ which v can reach via A_{sgi}, there are

$\sum_{s,t \in G-SGi \text{ via } A_{sgi}} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{aj \neq ak \in A_{sgi}} \sum_{s \in G-SGi \text{ via } aj,\ t \in G-SGi \text{ via } ak} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{aj \neq ak \in A_{sgi}} \frac{\sigma_{aj,ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$.

Therefore, the *out2out BC score* of v is computed as follows:

$\sum_{v \in SGi, s \neq t \in G-SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{aj \neq ak \in A_{sgi}, v \in SGi} \frac{\sigma_{aj,ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak)$.

Put it all together, and the *BC score* can be expressed as:

$\mathrm{BC}(v) = \sum_{SGi \in Set(SG)} \left(\sum_{s \neq v \neq t \in SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s \neq v \in SGi, t \in G-SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s \in G-SGi, v \neq t \in SGi,} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{v \in SGi, s \neq t \in G-SGi} \frac{\sigma_{st}(v)}{\sigma_{st}} \right)$

$= \sum_{SGi \in Set(SG)} \left(\sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} + \sum_{s \neq v \neq a, a \in A_{sgi}} \frac{\sigma_{s,a}(v)}{\sigma_{s,a}} * \alpha_{SGi}(a) + \sum_{a \neq v \neq t, a \in A_{sgi}} \frac{\sigma_{a,\ t}(v)}{\sigma_{a,t}} * \beta_{SGi}(a) + \sum_{aj \neq v \neq ak, aj, ak \in A_{sgi}} \frac{\sigma_{aj,\ ak}(v)}{\sigma_{aj,ak}} * \beta_{SGi}(aj) * \alpha_{SGi}(ak) \right)$

□

References

[1] L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 1977.

[2] A. Del Sol, H. Fujihashi, and P. O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 2005.

[3] H. Jeong, S. P. Mason, A. L. Barab´asi, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411, May 2001.

[4] R. Guimer`a, S. Mossa, A. Turtschi, and L. A. N. Amaral. The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles. In *NAS*, 2005.

[5] F. Liljeros, C. Edling, L. Amaral, H. Stanley, and Y. A° berg. The web of human sexual contacts. *Nature*, 2001.

[6] S. Jin, Z. Huang, Y. Chen, D. G. Chavarr´ıa-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *IPDPS*, 2010.

[7] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. In *NAS*, 2002.

[8] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47, 2004.

[9] V. Krebs. Mapping networks of terrorist cells. *Connections*, 2002.

[10] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 2001.

[11] Dimitrios Prountzos and Keshav Pingali. Betweenness centrality: algorithms and implementations. In *PPoPP*, 2013.

[12] D. Bader, K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *ICPP*, 2006.

[13] K. Madduri, D. Ediger, K. Jiang, D. Bader, D. Chavarria-Miranda. A Faster Parallel Algorithm and Efficient Multithreaded Implementations for Evaluating Betweenness Centrality on Massive Datasets. In *IPDPS*, 2009.

[14] G. Tan, D. Tu, N. Sun. A parallel algorithm for computing betweenness centrality. In *ICPP*, 2009.

[15] G. Tan, V. Sreedhar, G. Gao. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *Journal of Supercomputing*, 2011.

[16] O. Green, D. Bader. Faster Betweenness Centrality Based on Data Structure Experimentation. In *ICCS*, 2013.

[17] I. Leskvovec, K. Lang, A. Dasgupta, and M. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics 6*, 1 (2009), 29–123.

[18] 9th DIMACS Implementation Challenge. *http://www.dis.uniroma1.it/challenge9/download.shtml*.

[19] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *WAW*, 2007.

[20] U. Brandes and C. Pich. Centrality Estimation in Large Networks. *International Journal of Birfucation and Chaos*, 17(7), 2007.

[21] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *ALENEX*, 2008.

[22] N. Edmonds, T. Hoefler, and A. Lumsdaine. A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In *HiPC*, 2010.

[23] A. Buluc and J. Gilbert. The combinatorial BLAS: design, implementation, and applications. In *INT J HIGH PERFORM C*, 2011.

[24] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS*, 2011.

[25] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[26] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, 2012.

[27] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[28] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[29] Goh K-I, Cusick ME, Valle D, Childs B, Vidal M, and Barabási A-L. The Human Disease Network. In *NAS*, 2007.

[30] Albert-Laszlo Barabasi and Reka Albert. Emergence of Scaling in Random Networks. *Science*, 286:509–512, 1999.

[31] M. Faloutsos, P. Faloutosos, and C. Faloutsos. On Power-law Relationships of the Internet topology. In SIGCOMM, 1999.

[32] Hopcroft, J. and Tarjan, R. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM* 16 (6): 372–378.

[33] Scott Beamer Krste Asanovi´c David Patterson. Direction-Optimizing Breadth-First Search. In *SC*, 2012.

[34] Adam McLaughlin and David A. Bader. Scalable and High Performance Betweenness Centrality on the GPU. In *SC*, 2014.

[35] A. E. Sarıyu¨ce, E. Saule, K. Kaya, and U¨. C¸ atalyu¨rek, Regularizing Graph Centrality Computations. In *JPDC*, 2014.

[36] SNAP: Stanford Network Analysis Platform. *snap.stanford.edu/snap/index.html*

[37] Kaiyuan Zhuang, Rong Chen, Haibo Chen. NUMA-Aware Graph-Structured Analytics. In *PPoPP*, 2015.

[38] J. Zhao, H. Cui, J. Xue, X. Feng. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis. In *TPDS*, 2015.

[39] Lei Liu, Yong Li, Zehan Cui, Yungang Bao, Mingyu Chen, Chengyong Wu. Going Vertical in Memory Management: Handling Multiplicity by Multi-policy. In *ISCA*, 2014.

[40] Leiserson CE, Schardl TB. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA*, 2010.

[41] Y. Zhang, M. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *MICRO*, 2014.

[42] L. Liu, Y. Li, C. Ding, H. Yang, C. Wu. Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? In *TC*, 2016.

[43] J. Zhao, H. Cui, J. Xue, X. Feng, Y. Yan, and W. Yang. An Empirical Model for Predicting Cross-core Performance Interference on Multicore Processors. In *PACT*, 2013.