Parallel Betweenness Centrality in Dynamic Graphs

Nathan Bowness
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
nbown088@uottawa.ca

December 18th, 2020

Abstract

Dynamic graphs are an important tool to model constantly changing data for applications such as social networks, roads networks and biological networks. Finding correlations and relationships programmatically in these graphs is essential, as the amount of information has grown far beyond what humans can process manually. Betweenness centrality shows a user how influential a specific node is in connecting the other nodes around them. This paper implements Batch-iCENTRAL; a program based on a batch update algorithm proposed by Shukla et al. [5] that computes betweenness centrality on dynamic graphs. Batch-iCENTRAL is parallelizable on a graph's affected nodes, resulting in improved performance on multi-threaded machines. The algorithm leverages graph theory to reduce the number of repeated breadth-first search and reverse breadth-first searches when a batch of edges are applied. After experimentation, it is determined that Batch-iCENTRAL typically offers increased performance on batch sizes of 25 or more when compared to other non-batch algorithms.

1. Introduction

As the 21st Century progresses, humans are becoming more dependant on technology; whether that is a smart-phone, DNA sequencer, or even a bank machine, they all result in the creation of data. The mass abundance of data has led to tremendous research efforts into the fields of Data Science and Big Data. Through this research, it has become apparent that sequential computing is insufficient for performing computational tasks on massive data sets. As a result, parallel computing has been adopted to speed-up the computation process. Parallel computing allows programmers to push past the limitations of sequential computing, which are often restricted by hardware, by splitting sections of a large task into independent steps that can be executed simultaneously [1]. Parallel computing allows these independent steps to run on multiple processors at the same time. Once the individual steps are completed, the output can be interpreted by the main process which combines the different outputs, thereby defining a single solution to the large task. This parallelization will reduce the overall compute time for the task. Parallel computing cannot be used for all applications; particularly in applications that require steps to be completed in a specific sequence, but where it is applicable, users will see large gains in performance.

Graphs have become instrumental in modelling relationships in applications such as biological, social, and transportation networks. The applications mentioned, and many others, encompass massive data sets that require parallel computing for graph analysis within a reasonable period. One core metric for graph analysis is to evaluate the centrality of all nodes. Depending on the information desired, centrality can be measured in multiple ways including degree-, closeness-, betweenness-, Eigenvector- centrality

and many others, each offering a different view into the data [2]. In this paper, the focus will be solely on betweenness centrality (BC). BC for a node *v* in a graph *G* can be defined as "the fraction of the shortest paths between all pairs of nodes that pass through *v*" [3],[4]. Taking a social network as an example, BC values measure how influential a person is in connecting the network around them; someone with a high value will have the most influence.

There are many different algorithms for calculating BC, differing based on graph size and whether updates are expected. This paper's focus will be directed towards recomputing centrality in dynamic graphs; graphs that change over time by removing or adding edges. These graphs are better representations of real-world applications that are continuously changing. For example, in a transportation network, routes are constantly being added or removed. A social media network where people may be friending and unfriending multiple people every minute is another example. In these scenarios, algorithms are required to recompute existing values quickly, rather than recomputing all values again, thereby unnecessarily wasting time. The goal of this paper is to evaluate a cutting-edge parallel algorithm produced by Shukla et al. [5] for computing BC on dynamic graphs and evaluate its performance on additional datasets to verify the massive performance gain claimed.

2. Literature Review

Betweenness centrality can be computed using many algorithms. The "best" algorithm for a certain case generally depends on two factors: whether the graph is expected to change, and the size of it. Using these factors, the algorithms can be grouped into three main subsections: static graphs, massive graphs (100s of millions to billions of nodes/edges) and dynamic graphs. This paper will briefly touch on the first two sections with a focus on state-of-the-art algorithms for dynamic graphs.

2.1 Betweenness Centrality in Static Graphs

Computing the betweenness centrality for each node in a static graph is required as a preliminary step for most dynamic approaches. The dynamic algorithms leverage information such as all-pairs shortest paths and the number of shortest paths, both of which are stored during a preliminary run. This is done to speed up calculations for a future update. Algorithms for static graphs are often used for comparison to see how much a dynamic algorithm improves performance, rather than having to recompute the values for the entire graph again. The fastest known algorithm for computing betweenness centrality in a static graph was found by Brandes [6] and has a runtime of $\vartheta(|V||E|)$. Recently, researchers have attempted to increase the performance of the Brandes algorithm, but they were shown to only improve in some situations, and theoretically, the algorithms do not offer any computation advantage [7], [8]. Additionally, there has been lots of work in computing BC for static graphs in parallel, this work is outlined in many papers including [9], [10], [11], [12]. These parallel algorithms offer a decrease in computational time, with a downside of requiring a large amount of memory, so as graphs grow, they surpass the memory requirements of some machines. This is why approximation algorithms are used for massive graphs, to bypass the required memory needed for exact computations.

2.2 Betweenness Centrality Approximation in Massive Graphs

Calculating the exact betweenness centrality of a graph with hundreds-of-millions to billions of nodes and edges is slow and can be resource-intensive. To increase the speed of computation, users can sacrifice accuracy to get quick results using approximation algorithms. These are especially useful for applications where some chance of error is acceptable, but results are required quickly. Similarly, to calculating the exact betweenness, approximation research has been split into approximating static graphs and dynamic graphs. Approximating the betweenness centrality in static graphs has been researched in depth for social networks, some key papers include [13], [14]. More recent research has turned toward approximating the BC values in dynamic graphs [15], [16], [17]. Considering dynamic graphs, approximation gives a large speed up over exact calculations. To give an example, Hayashi et al. [16] mentioned their algorithm can "reflect a graph change in less than a millisecond on an average large-scale web graph with 106 [million] vertices and 3.7 [billion] edges". To contrast, the exact algorithm by Shukla et al. [5] for a graph with 325 thousand nodes and 1.082 million edges could process a batch update of 25 nodes in approximately 700 seconds.

2.3 Betweenness Centrality in Dynamic Graphs

Recomputing betweenness centrality for each node in a dynamic graph is a common scenario when modelling real-world networks. Significant research has been conducted to improve processing speed, thereby reducing the space required to store graph information and parallelize the algorithms. Some key papers on the topic include [18], [19], [20], [21], [22]. Most notably, Jamour et al. introduced iCENTRAL [22], an algorithm that offers a large speed improvement without a large space requirement that is also parallelizable. iCENTRAL was based on a few key concepts to reduce run time: limiting breadth-first search (BFS), using biconnected components (BCC), as well as identifying redundant nodes. Shukla et al. [5] have improved on the iCENTRAL algorithm by introducing the concept of a batch update to address its main flaw of being limited to sequential updates.

2.3.1 Calculating Betweenness Centrality for Dynamic Graphs

Considering a Graph G defined with a set of edges E and nodes V, in general the betweenness centrality (BC) for a node *v* in a graph can be defined as [6]:

$$BC_G[v] = \sum_{\substack{s,t \in V, \\ s \neq t \neq v}} \frac{\sigma_{st}(v)}{\sigma_{st}} = \sum_{\substack{s,t \in V, \\ s \neq t \neq v}} \frac{\delta_{st}(v)}{\delta_{st}} \qquad (1)$$

Where:

s, t	are also nodes in the graph G
$\sigma_{st}(v)$	is the number of shortest paths from s to t that pass through *v*
σ_{st}	is the number of shortest paths from s to t

The ideal algorithm proposed by Brandes [6], which is leveraged by dynamic graphs algorithms, uses pair and source dependencies denoted by $\delta_{st}(v)$ and $\delta_{s\bullet}(v)$ respectively to calculate the betweenness centrality. Pair and source dependencies are represented by equations (2) and (3) respectively, as follows:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \qquad (2) \qquad \delta_{s\bullet}(v) = \sum_{t \in V, t \neq v} \delta_{st}(v) \qquad (3)$$

The algorithm implements a BFS from node s to compute both σ_{sw} and $P_s(w)$ for all nodes w∈V with s≠w. The second step uses a reverse-BFS to find the source dependencies, $\delta_{s\bullet}(v)$, using Equation (4).

$$\delta_{s\bullet}(v) = \sum_{v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \left(1 + \delta_{s\bullet}(w)\right) \qquad (4)$$

Where:

σ_{ij} is the number of shortest paths from i to j

$P_s(w)$ is the list of parents of w in the BFS of s

$$BC_G[v] = \sum_{s \in V, s \neq v} \delta_{s\bullet}(v) \qquad (5)$$

Brandes' algorithm [6] allows for the computation of betweenness centrality by only performing a BFS and reverse-BFS. This combined with other techniques, offers a large speed improvement for dynamic graph algorithms. Brandes also keeps properties about the BFS traversal from each node so that the properties do not need to be re-computed from scratch each time. The information kept with regards to a source node, s, includes: S_s the order of nodes visited from the source node, D_s the distance from the source node, σ_s number of paths from s, and P_s the parent nodes of s.

2.3.2 Biconnected Components

Large graphs can be sectioned into their biconnected components. A BCC of a graph "is a maximal biconnected subgraph"[22]. The BBCs within a graph are only connected by articulation points – a node that would disconnect the graph if removed. The BBC sections allow for the graph to be logically split ensuring the following claims: a node can be a part of multiple BCCs, but an edge can only be part of one BCC. Figure 1 below shows a graph split into different BCCs and the articulation points connecting them.

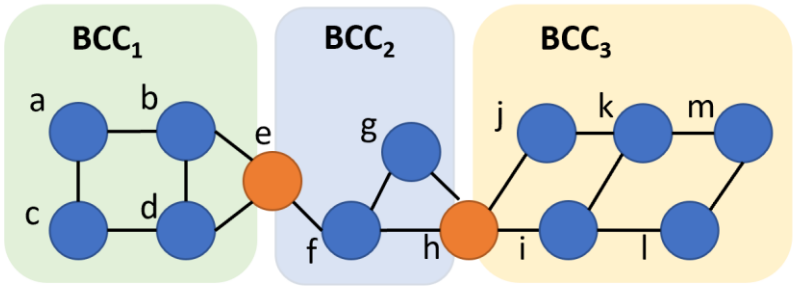

Figure 1: Graph displaying different biconnected components (BCC), and articulation points e and h in orange.

BCCs are a critical piece of dynamic BC algorithms as they limit the scope of the BFS required by the Brandes algorithm to re-compute BC. Jamour et al. [22] show that the BC can be computed completely within the BCC of the affected edge, thereby reducing the number of nodes the BFS must be run from drastically and offering a large speed improvement.

2.3.3 Batch Update for Betweenness Centrality

Previous approaches by [19], [22], [20], [23] to updating betweenness centrality were restricted to processing only one update at a time. The disadvantage to that approach is if multiple updates occur during a short period, there are consecutive updates to all nodes. As well, if multiple updates affect a single node *v*, recomputing the BC for each incremental update is wasteful. Equation (6) below shows the formula used by *i*CENTRAL [22] to update the BC by removing and adding source dependencies for each added or removed edge one by one. Further information about the *i*CENTRAL algorithm can be found in the Appendix.

$$BC_{G'}[v] = BC_{G'}[v] - \sum_{s \epsilon Q, s \neq v} \delta_{s\bullet}(v) + \sum_{s \epsilon Q, s \neq v} \delta'_{s\bullet}(v) \qquad (6)$$

Where:

- $BC_{G'}$ is the updated betweenness centrality value for a node *v* in Graph G
- Q is the set of all nodes for where $\delta_{s\bullet}(v)$ has changed after the insertion of edge e

Shukla et al. [5] improved upon *i*CENTRAL to allow for batch updates, circumventing the disadvantage of current approaches and improving performance dramatically. Shukla et al. propose removing all old source dependencies and then adding all new source dependencies for the entire batch of updated edges. Figure 2 below shows the improvements of calculating the betweenness centrality in a batch rather than sequentially for each edge.

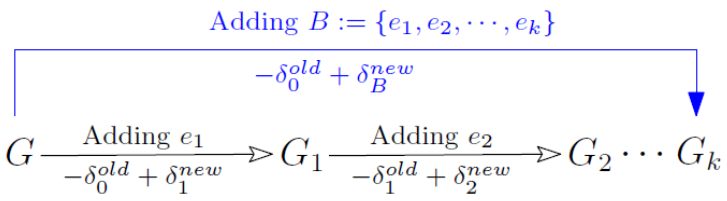

Figure 2: Diagram taken from Shukla et al. [5] showing their batch update approach in blue, and comparing it to iCENTRAL's [22] approach of sequential updates in black.

3. Problem Statement

The goal of this paper is to implement the batch update algorithm designed by Shukla et al. [5] and evaluate the parallel performance of their model on additional datasets. The original paper [5] only analyzed the algorithm's performance with one configuration of 48 threads on a CPU. This paper looks to evaluate the algorithm's performance under different processor configurations to give others more insight into how the algorithm parallelizes. In certain graph applications, it is reasonable to assume only a small number edge updates will ever occur. Thus, another goal is to evaluate if the batch update algorithm is always faster than iCENTRAL, or if there is additional overhead on small batch sizes. Finally, this paper will compare the batch update's performance on different graphs, grouped by various traits, with a goal of defining which graph properties impact the performance.

4. Parallel Batch-iCENTRAL

This section covers the algorithm implemented by this project, Batch-iCENTRAL. Batch-iCENTRAL is based on Shukla et al.'s batch update model [5], with some additional modifications to reduce complexity. Batch-iCENTRAL does not use the concept of chains in a graph or remove redundant nodes from the BFS and reverse-BFS steps. Both techniques offer minimal speedup to the algorithm, so they have been disregarded for this implementation. Shukla et al. mention that each technique offers a speedup of around 1.1 times for a batch size of 25 edges when compared to the original iCENTRAL. Since the batch update algorithm has a speedup of 3 to 15 times depending on the dataset, this is negligible. Batch-iCENTRAL can be used to update betweenness centrality on dynamic graphs, provided that Brandes-BFS [6] has been run on the original static graph beforehand. The algorithm leverages the properties stored during BFS and R-BFS that were discussed in Section 2.3.3, which include S_s, D_s, σ_s and P_s to increase speed.

4.1 Batch Update for Betweenness Centrality

The main method for Batch-iCENTRAL, seen below in Algorithm 1, requires a few input arguments: a graph G with set of edges E, vertices V, and Brandes-BFS data, as well as a batch of edges B, the existing betweenness centrality values of graph G and the operation to perform with the batch of edges. For the preliminary implementation, all edges are considered either inserted or deleted for the entire batch. On lines 5-6, the algorithm looks for all biconnected components that have edges either inserted or deleted from them. If the operation is an insertion, the BCCs of G' will be returned, whereas if the operation is a deletion, the BCCs of G will be returned. The reasoning for the difference is detailed in section 4.2. Once all affected BCCs are returned, all affected nodes within each BCC must be found. Affected nodes can be found by running a BFS from each side of a new edge and checking if the distances are equal. Any node u, is affected by an edge (v,w) if $d(u, v) \neq d(u, w)$.

```
1   Algorithm 1: Batch-iCENTRAL(G, B, BC, operation)
2
3     G′ = G ∪ B
4     BC′[v] = BC[v] foreach v∈V
5     bccs = findAffectedBccs( G, G′, B )
6     findAffectedNodesInBccs( bccs )
7
8     foreach affected bcc ∈ bccs in G do
9       foreach node s ∈ bcc do in parallel
10        AtomicICentral(s, bcc, BC′, -1)
11      end
12    end
13
14    foreach affected bcc ∈ bccs in G′ do
15      foreach node s ∈ bcc′ do in parallel
16        if operation is DELETION
17          removeNewEdges( bcc′ )
18        AtomicICentral(s, bcc, BC′, +1)
19      end
20    end
```

After the affected nodes and BCCs have been found, lines 8-12 deal with removing the original source dependencies for the BC array. This section of the algorithm is equivalent to the $-\sum_{s\epsilon Q, s\neq v} \delta_{s\bullet}(v)$ portion of Equation (6). Each of the affected nodes within a BCC of G are divided amongst the available machines or processors and AtomicICentral is run in parallel. When the operation is insertion, the BBCs being iterated through correspond to the graph G'. To ensure only source dependencies related to G are removed, any nodes that do not have previous Brandes-BFS properties are ignored during the subtraction steps. This ensures that all newly inserted edges are ignored.

After the betweenness centrality array has had its old source dependencies removed, lines 14-20 deal with adding all new source dependencies to the array. This section is equivalent to the $+\sum_{s\epsilon Q, s\neq v} \delta'_{s\bullet}(v)$ portion of Equation (6). For this section, all affected nodes for a BCC of G' are divided amongst processors to call AtomicICentral in parallel. Since the affected BCCs are supposed to correspond to G', if the operation is deletion, all edges must be added to the BCC before running AtomicICentral. This ensures that the source dependencies are added from all affected nodes in G' for both types of operations.

4.2 Finding Affected Biconnected Components

Finding the affected biconnected components of the graph is essential to the Batch-iCENTRAL algorithm. The affected BCCs are defined as any BCC (sub-section) of the graph that contains one of the modified edges. The array of BCCs returned by the algorithm either correspond to the BCCs from the original graph G, or the BCCs from the modified graph G'. If the operation is deletion, then the BCCs of the original graph G are returned. This covers any case where removing an edge would separate one biconnected component into two separate BCCs. However, the opposite is true if the operation is insertion. In this case, the affected BCCs of the modified graph G' are returned. This supports cases where an inserted edge can combine two different biconnected components into one. Algorithm 2

below shows this logic in a more distinct manner and ensures that all affected BCCs are returned for the corresponding graph G or G'.

```
1   Algorithm 2: findAffectedBccs( G, G′, B, operation )
2
3     affectedBccs = []
4     graphToSearch = G
5
6     if operation is INSERTION
7       graphToSearch = G′
8
9     foreach edge ∈ B do
10      affectedBcc = findBccThatEdgeChanges( graphToSearch, edge)
11      if affectedBcc ∉ affectedBccs
12        affectedBccs ⇐ affectedBcc
13    end
14
15    return affectedBccs
```

4.3 Atomic iCENTRAL

Atomic iCENTRAL, seen in Algorithm 3 below, was taken directly from the pseudocode outline by Shukla et Al [5]. It leverages the theory proposed by Jamour et Al. to allow recomputing updated betweenness centrality values from within a BCC [22]. Based on that theory, the arguments for the algorithm do not include the entire graph. Instead, they are the node to traverse from, the biconnected component to traverse, the current BC array and lastly, the factor determining if it should subtract or add dependencies. Lines 5-17 of Algorithm 3 use a reverse BFS and theory proposed by Jamour et Al. [22] to update the BC by either adding or subtracting source dependencies atomically. If the factor is -1, the source dependencies are subtracted, and if the factor is 1, then source dependencies are added. All changes to the array storing BC values are atomic. This guarantees that no writes occur at the same time and ensures that correct values are calculated for BC.

Algorithm 3: *atomicICentral*(s, bcc, BC′, factor)

find $\sigma_s[v]$and $P_s[v]$ for $v \in$ bcc
foreach $v \in$ bcc, set $\delta_s[v] = 0$ and $\delta_{G_s}[v] = 0$
foreach w ∈ reverse BFS of s **do**
  if s, w are articulation points
    $\delta_{G_s}[v] = |G_s| \cdot |G_w|$
  foreach p ∈ $P_s[w]$ **do**
    $\delta_s[p] = \delta_s[p] + \frac{\sigma_s[p]}{\sigma_s[w]}(1 + \delta_s[w])$
    if s is an articulation point
      $\delta_{G_s}[p] = \delta_{G_s}[p] + \delta_{G_s}[w] \cdot \frac{\sigma_s[p]}{\sigma_s[w]}$
  if s≠w **do atomic**
    $\boldsymbol{BC}'[w]$ += factor$\cdot \frac{\delta_s[w]}{2}$
  if s is an articulation point **do atomic**
    $\boldsymbol{BC}'[w]$ += factor$\cdot \delta_s[w] \cdot |G_s|$
    $\boldsymbol{BC}'[w]$ += factor$\cdot \frac{\delta_{G_s}[w]}{2}$
end

5. Experimental Evaluation

5.1 Experimental Setup

All experiments were performed on a Linux Virtual Machine (VM) running Ubuntu 18.04, that was hosted on a computer running Windows. The Windows computer was using an Intel Xeon W-2175 CPU, that had 14 cores, 28 logical processors, all clocked at 2.50 GHz. The Linux VM itself was provided 96 GB of RAM and 20 processors. The VM's performance when trying to allocate more than 20 processors was quite sluggish due to the Windows operating system requirements on the original computer. All algorithms were implemented in C++ 11 and all testing was done on a single machine by creating threads.

Code for the original iCentral algorithm, designed by Jamour et al. [22], was available on GitHub [24]. For this project, some modifications were made to the original iCentral code to compare results with Batch-iCentral. Brandes-BFS and the algorithms for finding BCCs were reused from the available code on GitHub [24].

5.2 Datasets

Table 1: The datasets used for performance testing of Batch-iCENTRAL.

	Graph Dataset	|V|	|E|	Average Degree	Source
Web Networks	web-NotreDame	3.25×10^5	1.50×10^6	9	[25]
	web-Stanford	2.82×10^5	2.31×10^6	16	[26]
	rec-Amazon	9.18×10^4	1.26×10^5	1	[27]
	soc-Epinions	2.66×10^4	1.00×10^5	7	[28]
	web-EPA	4.77×10^3	9.00×10^3	4	[29]
Road Networks	road-Luxembourg	1.15×10^5	1.20×10^5	1	[30]
	road-Minnesota	2.64×10^3	3.30×10^3	2	[31]
	road-Euroroad	1.17×10^3	1.42×10^3	1	[32]
Biologic Networks	PP-Pathways	2.16×10^4	3.42×10^5	32	[33]
	Bio-grid Human	9.53×10^3	6.24×10^4	4	[34]
	Bio-grid Mouse	1.46×10^3	3.72×10^3	13	[35]

Datasets from three different applications were used for performance testing: web networks, road networks and biological networks. All graphs were treated as undirected and unweighted. For any detached graphs, additional edges were added to ensure that the graph was connected. For analysis, all graphs were grouped into two categories based on network size and average degree. For network size, the graphs were classified as small, medium, and large. Small networks were defined as graphs with less than ten-thousand nodes, medium networks were defined as having between ten-thousand and one hundred-thousand nodes, and finally, large networks contain over one hundred-thousand nodes. The graphs' average degrees were grouped into 3 categories as well: low degree networks, medium degree networks, and high degree networks. Low degree networks are marked as having an average degree of less than 2, medium degree networks are marked as having an average degree between 3 and 10, and high degree networks are classified as having an average degree of above 10.

5.3 Results

5.3.1 Parallel Performance

It is important to evaluate the parallel performance of algorithms to ensure that they can take advantage of available hardware. Speedup graphs offer insight to how an algorithm performs across multiple processors for a given dataset. To evaluate the parallel performance of Batch-iCENTRAL, it was run with configurations of 1, 10 and 20 cores on the different datasets mentioned in Section Datasets5.2. To evaluate each of the configurations accurately, the results from trials with five deletions and five insertions were averaged for each dataset. The results were grouped by the two graph properties discussed above. The grouping of various graph sizes can be seen in Figure 3, and the graphs grouped by average degree in Figure 4.

The graphs in Figure 3 show the results for performance, where the parallelization on large graphs is much better than on small graphs. All parallelization is done on affected nodes within any affected biconnected components, therefore in general, larger graphs have more nodes to spread amongst processors. With a configuration of 20 processors, the performance on a large network is about 1.5 times better than the performance on 10 cores. This is not ideal, but still offers a large performance

boost. For the same large datasets, more processors would increase performance but would likely decrease the relative speedup per processor. For graphs larger than the ones tested, with a million or more nodes, having more than 20 processors would be best. The graph showing the performance on small networks shows the limitation of parallelizing Batch-iCENTRAL. If there are not enough affected nodes within an affected BCC, then processors are sitting idle waiting for others to finish. In Figure 3, it can be seen that the ideal number of processors for small graphs appears to be around 10 processors. As part of future work, more trials should be run to find the ideal number of processors for different graph sizes, split into more granular group sizes.

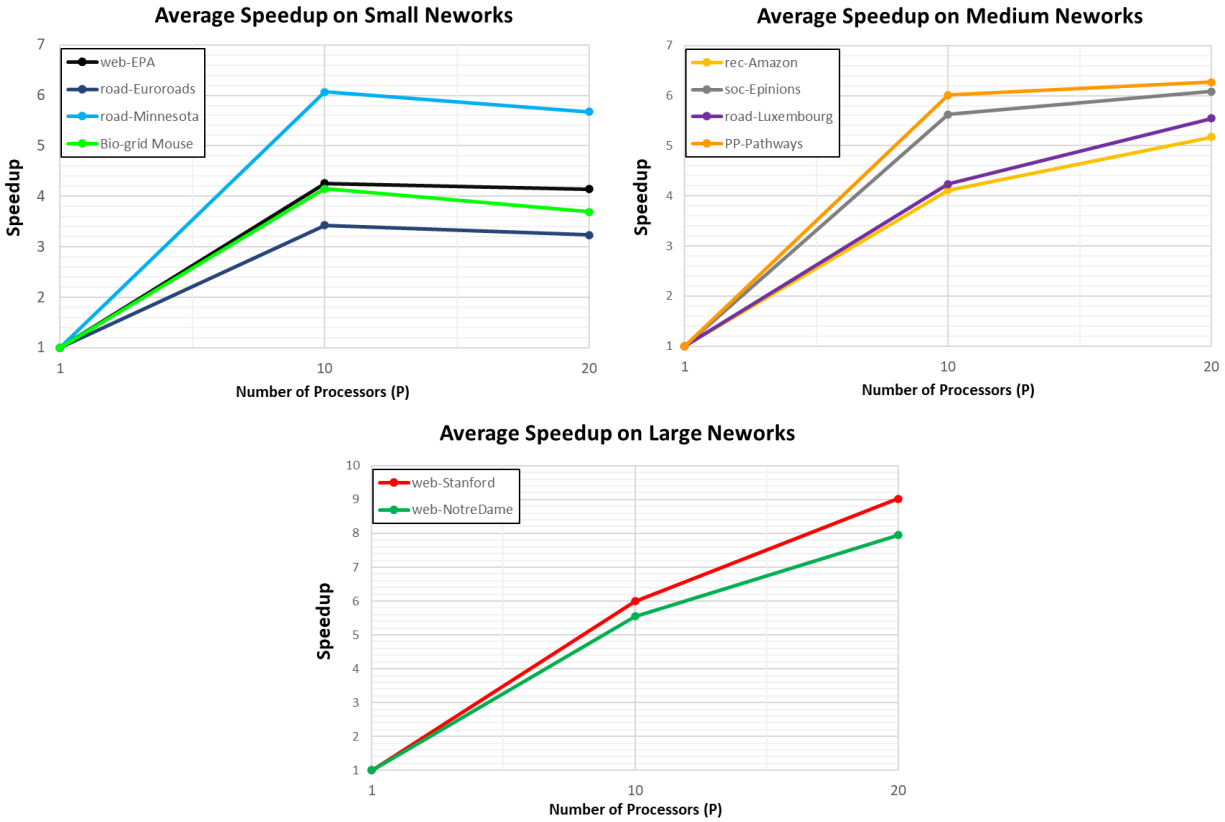

Figure 3: Graphs of the average parallel speedup for Batch-iCENTRAL versus the number of processors used on networks of various sizes.

The parallel performance of Batch-iCENTRAL grouped on networks with various average degrees are shown in Figure 4. The graphs do not have an evident correlation like with network size. The average degree does not directly impact the number of nodes available to parallelize on, however graphs with a higher average degree should have less biconnected components. This is not necessarily the case, but in general, a graph of the same size with a higher average degree should have more edges between nodes resulting in less BCCs and more nodes in each one. The graph with higher average degrees does appear to have a slightly higher speedup compared to the graphs for medium or small average degrees. This difference does not give conclusive evidence that the average degree impacts the parallel performance.

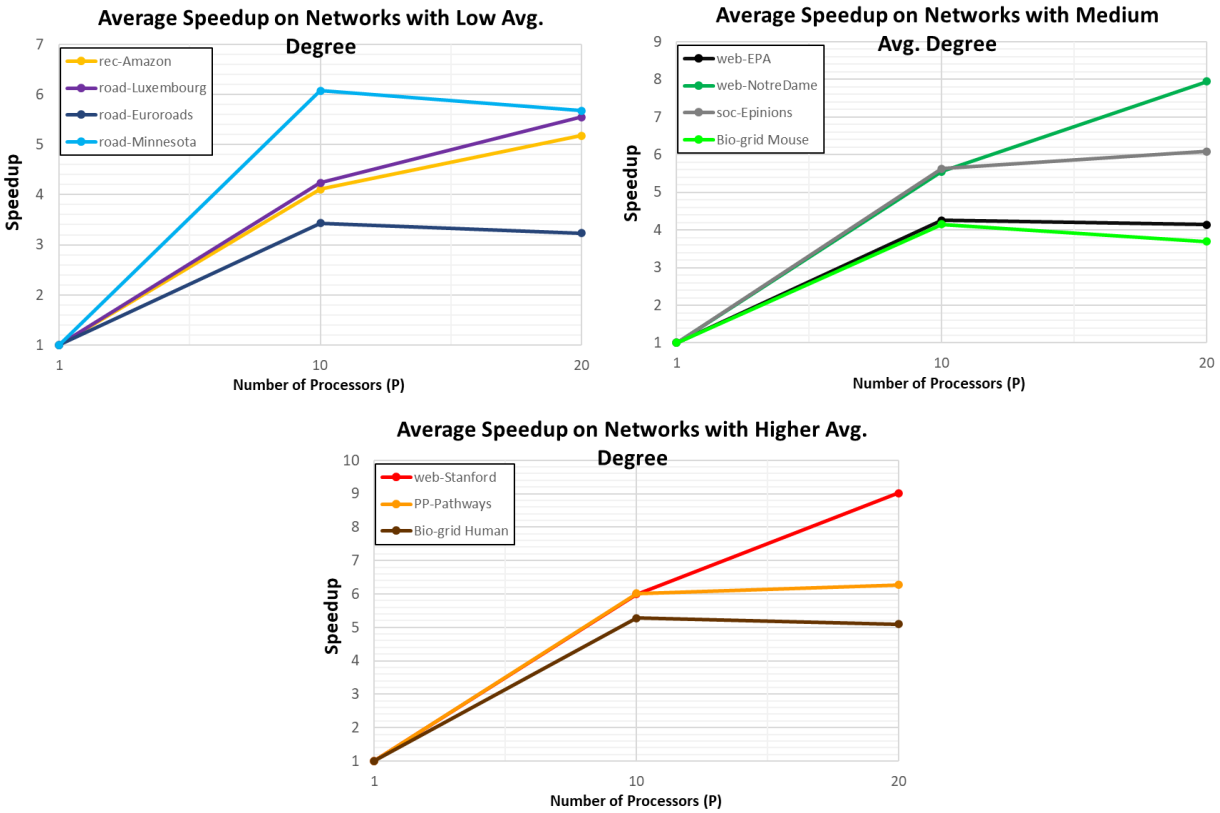

Figure 4: Graphs of the average parallel speedup for Batch-iCENTRAL versus the number of processors used on networks with various average degrees.

5.3.2 Comparison of iCentral to Batch-iCentral

Shukla et al. designed their batch update model to offer large performance gains when calculating betweenness centrality on a batch of edges. The original paper by Shukla et al. only compared their batch update model to iCENTRAL on batch sizes of 25, 50 and 100 [5]. From their test results, the speedup of the batch model versus the original iCENTRAL algorithm was large; it was measured to be 63 times faster on average for a batch size of 100. This paper uses the same batch sizes of 25, 50 and 100 to allow for comparisons on similar datasets. The results were once against grouped based on two graphs traits: the graph size and the average degree to see if they impact the results.

Analysis of Figure 5 aligns with Shukla et al.'s batch update theory. The speedup of Batch-iCENTRAL for smaller networks is higher than on large networks for the same number of edges. This is because there are more edges affected per each of the biconnected components, which result in less repeated traversals when using Batch-iCENTRAL. For larger graphs, edges are spread out amongst more BCCs and adding edges sequentially results in only a few repeated traversals by iCENTRAL, lowering the speedup. If a batch size proportional to the number of nodes in a graph was used, likely the speedups for all datasets seen in Figure 5 would be similar – assuming they have similar average degrees.

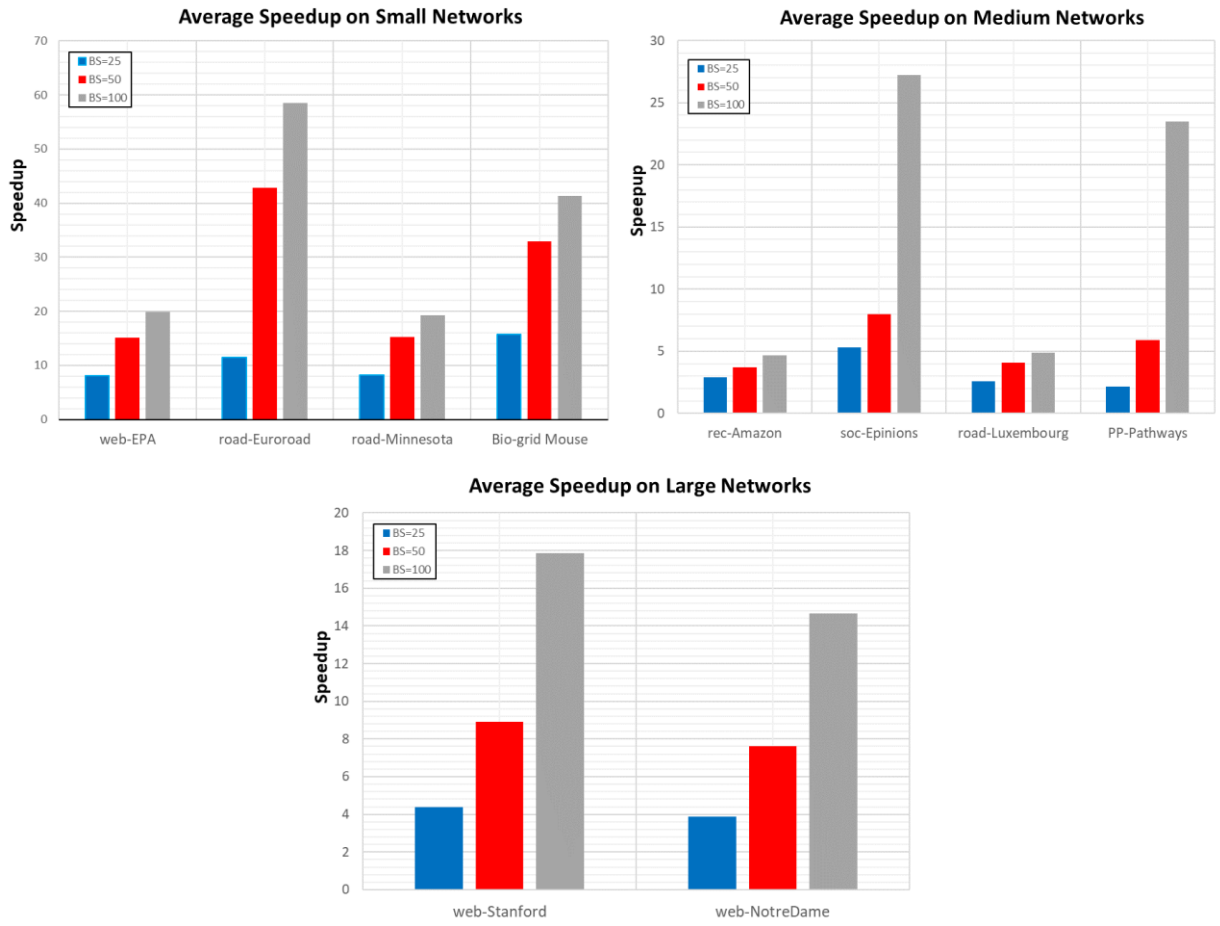

Figure 5: Graphs comparing the speedup of Batch-iCENTRAL over iCENTRAL, for various network sizes, on batch sizes (BS) of 25, 50 and 100.

The general statement that smaller graphs will have a higher speedup when using Batch-iCENTRAL over iCENTRAL does not apply in all scenarios. In particular, the rec-Amazon and road-Luxembourg have a much smaller speedup compared to the other datasets. This can be better explained by looking at the average degree for datasets in Table 1 and looking at the graphs in Figure 6. Both rec-Amazon and road-Luxembourg have a very low average degree of around 1, which causes the graph to have more BCCs. These BCCs will have less edges in each, resulting in iCENTRAL having improved performance with less repeated traversals. Graphs with medium to high average degrees, outlined as values from 3 to 32, have a much higher speedup. In general, it appears that both the graph size and the average degree of nodes correlate to how affective Batch-iCENTRAL is over iCENTRAL.

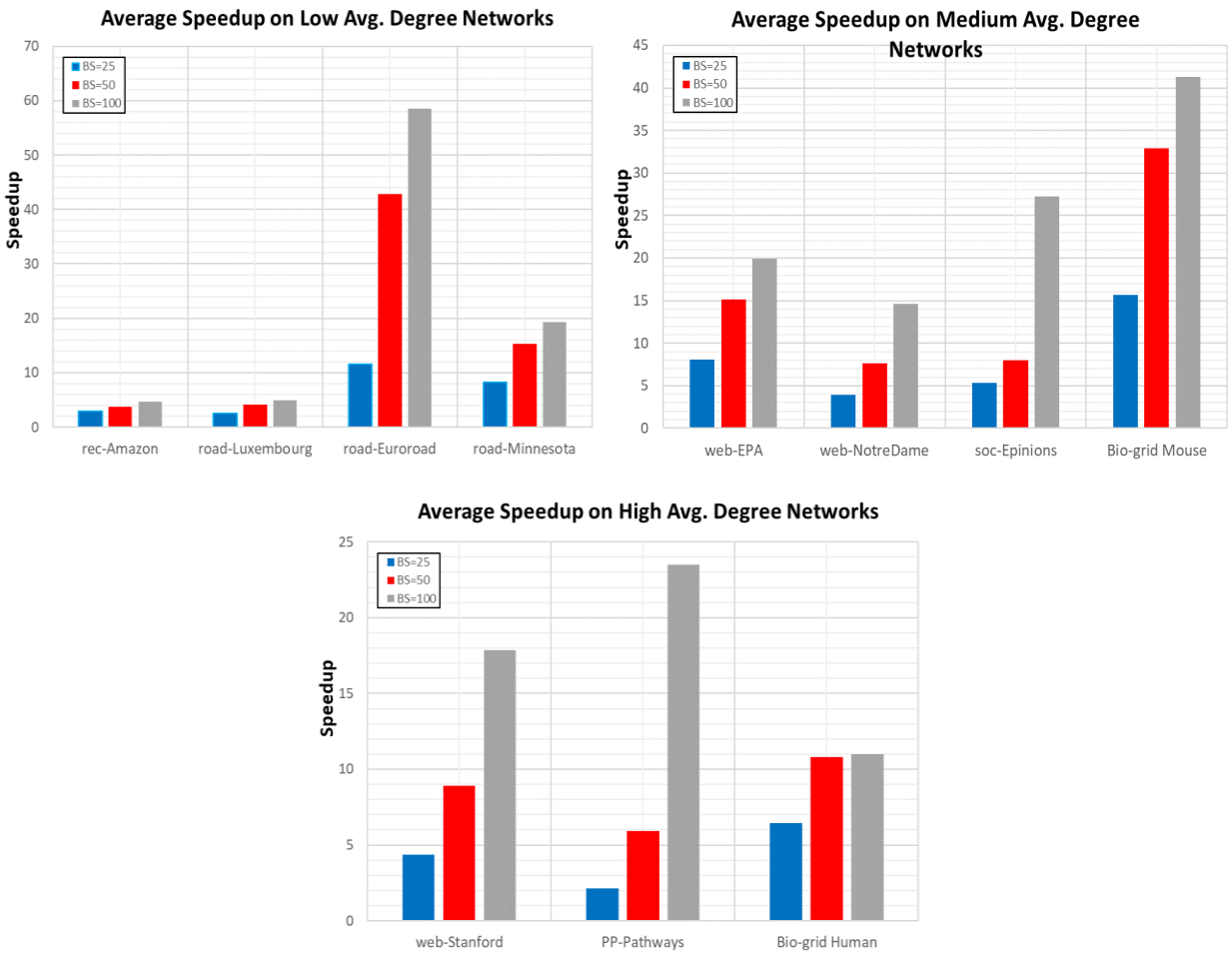

Figure 6: Graphs comparing the speedup of Batch-iCENTRAL over iCENTRAL, for graphs with varying average degrees, on batch sizes (BS) of 25, 50 and 100.

Shukla et al. did not mention evaluating the performance of their algorithm on smaller batch sizes of 1, 2, etc. [5]. To fill this gap, Batch-iCENTRAL was run with batch sizes of 1, 2, 3, and 5 and compared to the performance of iCENTRAL on similar batch sizes. The performance results were almost identical. The only significant performance difference was when multiple edges from the batch were added to the same biconnected component. This results in only one BFS and reverse-BFS being required with Batch-iCENTRAL. On smaller graphs that only contain a few BCCs, 5 edges was enough to start seeing some performance difference, however on larger graphs, even batches with 10 edges did not result in any noticeable speedup of Batch-iCENTRAL over iCENTRAL. This shows that the distribution of edges over the BCCs is imperative for evaluating the performance gain Batch-iCENTRAL can offer. Batch-iCENTRAL's main downside is the increased memory requirement. The algorithm keeps all affected BCCs and nodes in memory at once, rather than storing one BCC and one set of affected nodes like in iCENTRAL. This is important to note when evaluating a small set of edges on very large graphs as the memory requirement is larger with negligible speedup since each edge will affect separate biconnected components.

5.3.3 Comparison to Literature

The performance results collected by Shukla et al. for the speedup of their batch update compared to regular iCENTRAL were much larger than the results gathered when comparing Batch-iCENTRAL to iCENTRAL [5]. On the overlapping datasets: PP-Pathways, web-Stanford and web-NotreDame, Shukla et al. found a speedup of 90, 50 and 64 respectively on a batch size of 100 edges. With Batch-iCENTRAL the results showed the speedups to be 23.5, 18 and 15. Therefore, Batch-iCENTRAL seems to be 65-75% slower than the batch algorithm proposed by Shukla et al. This large discrepancy can likely be attributed to how the edges were inserted into the graph. If more edges are inserted into the same biconnected component, the speedup is larger when compared to iCENTRAL. Shukla et al. did not discuss how they inserted or deleted edges from the graph, however in this project all edges were randomly inserted or deleted. As well, the discrepancy can also be attributed to different hardware being used; Shukla et al. utilized hardware with 128 GB of RAM and 48 threads whereas this project uses 96 GB of RAM and 20 processors in the largest configuration.

6. Conclusion

The goal of this paper was to implement and study Shukla et al.'s batch update algorithm for computing betweenness centrality on dynamic graphs [5]. The algorithm uses theory proposed by Jamour et al. to allow for updating BC values by only traversing a biconnected component rather than the entire graph [22]. As a result of this paper, Batch-iCENTRAL was created based on Shukla et al.'s batch update algorithm. Batch-iCENTRAL was tested to evaluate its parallel performance and its speedup over the iCENTRAL algorithm that processes edges sequentially [22]. The tests to find the parallel performance and speedup were run against multiple graph networks as outlined in **Table *1***. The results were then grouped based on both graph size and the graph's average degree. The results show that parallelization is much better on larger graphs since more nodes are available in each of the BCCs to spread amongst processors. As well, the results showed that the speedup of Batch-iCENTRAL over iCENTRAL is dependent on both the graph size and average degree of a graph. The results align with the findings made by Shukla et al. and confirm that the batch update algorithm's speedup over iCENTRAL is dependent on the number of traversals needed through each affected BCC. Batch-iCENTRAL was not as performant as the original batch update algorithm designed by Shukla et al, but was able to demonstrate the advantage of batching groups of edges when recomputing BC in dynamic graphs.

Future work would include evaluating the datasets shown in Table 1 on additional processor configurations, including configurations with more processors, to find exactly which configurations work best for datasets with certain properties. This evaluation would allow for less general statements and allow for others to estimate their expected performance. In the future, it would also be beneficial to see how Batch-iCENTRAL compares to the estimation algorithms outlined in Section 2.2 when recomputing betweenness centrality on massive graphs.

7. Appendix:

7.1 iCENTRAL:

Jamour et al. [22] proposed an algorithm *i*CENTRAL that computes updated betweenness centrality values after an edge insertion or deletion. Although it requires each update to be processed sequentially, their algorithm was a large improvement over [23]. *i*Central allows for updating BC values to be computed incrementally by only considering the affected biconnected components, rather than the entire graph. This decreases the amount of BFS and reverse-BFS required to compute BC as it is limited to only the affect BCC. The algorithm uses Equation (6) to sequentially remove and then add source dependencies as each edge is added one by one, calculating BC by using Brandes algorithms. Equation (7) below shows a more in-depth formula for the calculation used in the iCENTRAL algorithm. For further information on the proof and theorem please refer to [22].

$$BC_{G'}[v] = BC_G[v] - A[v] + A'[v] - B[v] + B'[v] - C[v] + C'[v] \qquad (7)$$

Where:

A[v], A'[v]	is the contribution of nodes s and t that are in the affected BCC, to the source dependencies of v
B[v], B'[v]	is the contribution of node s that is in the affected BCC and t that is not in the affected BCC, to the source dependencies of v
C[v], C'[v]	is the contribution of nodes s and t that are not in the affected BCC, to the source dependencies of v

Full derivations and definitions of the equations for A[v], A'[v], B[v], B'[v], C[v], C'[v] can be found in [22].

$$A[v] = \sum_{\substack{s,t \in B'_e \\ s \neq t \neq v}} \delta_{st}(v) \quad , \quad A'[v] = \sum_{\substack{s,t \in B'_e \\ s \neq t \neq v}} \delta'_{st}(v) \qquad (8), (9)$$

$$B[v] = \sum_{\substack{s \in G_i, t \in B'_e \\ s \neq t \neq v \\ i=1 \ldots k}} \delta_{st}(v) \quad , \quad B'[v] = \sum_{\substack{s \in G_i, t \in B'_e \\ s \neq t \neq v \\ i=1 \ldots k}} \delta'_{st}(v) \qquad (10), (11)$$

$$C[v] = \sum_{\substack{s \in G_i, ts \in G_j \\ s \neq t \neq v, i=1 \ldots k \\ j=1 \ldots k, i \neq j}} \delta_{st}(v) \quad , \quad C'[v] = \sum_{\substack{s \in G_i, ts \in G_j \\ s \neq t \neq v, i=1 \ldots k \\ j=1 \ldots k, i \neq j}} \delta'_{st}(v) \qquad (12), (13)$$

Where:

G'	is the graph constructed by added an edge *e* to graph *G*
B'_e	is the biconnected component of G' that edge *e* belongs to
$a_1, \ldots, a_k$	are the articulation points of BCC B'_e
$G_1, \ldots, G_k$	are the subgraphs of G' connected to B'_e through $a_1, \ldots, a_k$ respectively

8. References

[1] S. Rastogi and H. Zaheer, “Significance of Parallel Computation over Serial Computation Using OpenMP, MPI, and CUDA,” *ResearchGate*, Oct. 2018. https://www.researchgate.net/publication/320213267_Significance_of_Parallel_Computation_over_Serial_Computation_Using_OpenMP_MPI_and_CUDA (accessed Oct. 03, 2020).

[2] E. Yan and Y. Ding, “Applying centrality measures to impact analysis: A coauthorship network analysis,” *Journal of the American Society for Information Science and Technology*, vol. 60, no. 10, pp. 2107–2118, 2009, doi: 10.1002/asi.21128.

[3] L. C. Freeman, “A Set of Measures of Centrality Based on Betweenness,” *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977, doi: 10.2307/3033543.

[4] J. M. Anthonisse, “The rush in a directed graph,” Art. no. BN 9/71, Jan. 1971, Accessed: Oct. 09, 2020. [Online]. Available: https://ir.cwi.nl/pub/9791.

[5] K. Shukla, S. C. Regunta, S. H. Tondomker, and K. Kothapalli, “Efficient parallel algorithms for betweenness- and closeness-centrality in dynamic graphs,” in *Proceedings of the 34th ACM International Conference on Supercomputing*, New York, NY, USA, Jun. 2020, pp. 1–12, doi: 10.1145/3392717.3392743.

[6] U. Brandes, “A faster algorithm for betweenness centrality,” *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, Jun. 2001, doi: 10.1080/0022250X.2001.9990249.

[7] R. Puzis, P. Zilberman, Y. Elovici, S. Dolev, and U. Brandes, *Heuristics for Speeding Up Betweenness Centrality Computation*. 2012, p. 311.

[8] D. Erdos, V. Ishakian, A. Bestavros, and E. Terzi, “A Divide-and-Conquer Algorithm for Betweenness Centrality,” Jun. 2014, doi: 10.1137/1.9781611974010.49.

[9] D. A. Bader and K. Madduri, “Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks,” in *2006 International Conference on Parallel Processing (ICPP’06)*, Aug. 2006, pp. 539–550, doi: 10.1109/ICPP.2006.57.

[10] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, “A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–8, doi: 10.1109/IPDPS.2009.5161100.

[11] G. Tan, D. Tu, and N. Sun, “A Parallel Algorithm for Computing Betweenness Centrality,” in *2009 International Conference on Parallel Processing*, Sep. 2009, pp. 340–347, doi: 10.1109/ICPP.2009.53.

[12] N. Edmonds, T. Hoefler, and A. Lumsdaine, “A space-efficient parallel algorithm for computing betweenness centrality in distributed memory,” in *2010 International Conference on High Performance Computing*, Dec. 2010, pp. 1–10, doi: 10.1109/HIPC.2010.5713180.

[13] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, “Approximating Betweenness Centrality,” in *Algorithms and Models for the Web-Graph*, Berlin, Heidelberg, 2007, pp. 124–137, doi: 10.1007/978-3-540-77004-6_10.

[14] R. Geisberger, P. Sanders, and D. Schultes, “Better Approximation of Betweenness Centrality,” in *2008 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 0 vols., Society for Industrial and Applied Mathematics, 2008, pp. 90–100.

[15] E. Bergamini, H. Meyerhenke, and C. L. Staudt, “Approximating Betweenness Centrality in Large Evolving Networks,” in *2015 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*, 0 vols., Society for Industrial and Applied Mathematics, 2014, pp. 133–146.

[16] T. Hayashi, T. Akiba, and Y. Yoshida, “Fully dynamic betweenness centrality maintenance on massive networks,” *Proc. VLDB Endow.*, vol. 9, no. 2, pp. 48–59, Oct. 2015, doi: 10.14778/2850578.2850580.

[17] S. K. Maurya, X. Liu, and T. Murata, “Fast Approximations of Betweenness Centrality with Graph Neural Networks,” in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, New York, NY, USA, Nov. 2019, pp. 2149–2152, doi: 10.1145/3357384.3358080.
[18] M.-J. Lee, J. Lee, J. Park, R. Choi, and C.-W. Chung, “QUBE: A quick algorithm for updating betweenness centrality,” *WWW’12 - Proceedings of the 21st Annual Conference on World Wide Web*, Apr. 2012, doi: 10.1145/2187836.2187884.
[19] O. Green, R. McColl, and D. A. Bader, “A Fast Algorithm for Streaming Betweenness Centrality,” in *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, Sep. 2012, pp. 11–20, doi: 10.1109/SocialCom-PASSAT.2012.37.
[20] M. Pontecorvi and V. Ramachandran, “A Faster Algorithm for Fully Dynamic Betweenness Centrality,” Jun. 2015, Accessed: Oct. 17, 2020. [Online]. Available: https://arxiv.org/abs/1506.05783v3.
[21] N. Kourtellis, G. De Francisci Morales, and F. Bonchi, “Scalable online betweenness centrality in evolving graphs,” in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, May 2016, pp. 1580–1581, doi: 10.1109/ICDE.2016.7498421.
[22] F. Jamour, S. Skiadopoulos, and P. Kalnis, “Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 659–672, Mar. 2018, doi: 10.1109/TPDS.2017.2763951.
[23] M.-J. Lee, S. Choi, and C.-W. Chung, “Efficient algorithms for updating betweenness centrality in fully dynamic graphs,” *Information Sciences*, vol. 326, pp. 278–296, Jan. 2016, doi: 10.1016/j.ins.2015.07.053.
[24] F. Jamour, *fjamour/icentral*. 2020. https://github.com/fjamour/icentral
[25] “SNAP: Network datasets: Notre Dame web graph.” https://snap.stanford.edu/data/web-NotreDame.html (accessed Nov. 04, 2020).
[26] R. Rossi and N. Ahmed, “Stanford | Web Graphs | Network Data Repository.” http://networkrepository.com/web-Stanford.php (accessed Dec. 17, 2020).
[27] R. Rossi and N. Ahmed, “amazon | Recommendation Networks | Network Data Repository.” http://networkrepository.com/rec-amazon.php (accessed Dec. 17, 2020).
[28] R. Rossi and N. Ahmed, “epinions | Social Networks | Network Data Repository.” http://networkrepository.com/soc-epinions.php (accessed Dec. 17, 2020).
[29] R. Rossi and N. Ahmed, “EPA | Web Graphs | Network Data Repository.” http://networkrepository.com/web-EPA.php (accessed Dec. 17, 2020).
[30] R. Rossi and N. Ahmed, “luxembourg-osm | Road Networks | Network Data Repository,” *Network Repository*. http://networkrepository.com/road-luxembourg-osm.php (accessed Dec. 17, 2020).
[31] R. Rossi and N. Ahmed, “minnesota | Road Networks | Network Data Repository,” *Network Repository*. http://networkrepository.com/road-minnesota.php (accessed Dec. 17, 2020).
[32] R. Rossi and N. Ahmed, “euroroad | Road Networks | Network Data Repository,” *Network Repository*. http://networkrepository.com/road-euroroad.php (accessed Dec. 17, 2020).
[33] “BioSNAP: Network datasets: Human protein-protein interaction network.” https://snap.stanford.edu/biodata/datasets/10000/10000-PP-Pathways.html (accessed Dec. 17, 2020).
[34] R. Rossi and N. Ahmed, “grid-human | Biological Networks | Network Data Repository.” http://networkrepository.com/bio-grid-human.php (accessed Dec. 17, 2020).
[35] R. Rossi and N. Ahmed, “grid-mouse | Biological Networks | Network Data Repository,” *Network Repository*. http://networkrepository.com/bio-grid-mouse.php (accessed Dec. 17, 2020).