Optimizing a Distributed Graph Data Structure for K-Path Centrality Estimation on HPC

Lance Fletcher[1,2], Trevor Steil[1], Roger Pearce[1,2]
[1]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory (LLNL)*
[2]*Department of Computer Science and Engineering, Texas A&M Univeristy*
{fletcher28, steil1, rpearce}@llnl.gov

***Abstract*—K-Path centrality is based on the flow of information in a graph along simple paths of length at most K. This work addresses the computational cost of estimating K-path centrality in large-scale graphs by introducing the random neighbor traversal graph (RaNT-Graph). The distributed graph data structure employs a combination of vertex delegation partitioning and rejection sampling, enabling it to sample massive amounts of random paths on large scale-free graphs. We evaluate our approach by running experiments which demonstrate weak scaling on R-MAT graphs and strong scaling on large real-world graphs. The RaNT-Graph approach achieved a 56,544x speedup over the baseline 1D partition implementation when estimating K-path centrality on a graph with 89 million vertices and 1.9 billion edges.**

***Index Terms*—centrality, distributed graph processing, vertex delegation, random paths, random walks**

I. Introduction

With the growing amount of data being collected and processed, the importance of scalable algorithms becomes more evident. Large amounts of data are often represented as graphs, enabling insightful information to be extracted in a variety of applications. A common objective associated with processing graphs is the concept of centrality that assigns a value or ranking to vertices or edges to quantify their importance. κ-path centrality is a relatively new centrality metric which assigns each vertex v a value based on the sum of the probabilities a simple path of length at most κ originating from all other vertices passes through v [1]. The method mimics the concept of a message being propagated in a social network, where it can only traverse along a simple path through users (vertices) who share a connection (edge). More important users are likely to propagate more messages. An edge variant of κ-path centrality has also been introduced and is based on the same idea of information flowing through a network [11].

κ-path centrality has been utilized in an assortment of graph problems such as community detection and link prediction [3], [9], [10]. In [4], Blackburn et al. use κ-path centrality as a substitute for betweenness centrality to evaluate the behavior of cheaters in an online gaming network.

As shown in [1] and [16], a large motivation behind κ-path centrality is its ability to identify vertices which have high betweenness centrality. The betweenness centrality of a vertex v is found by determining what fraction of shortest paths between all vertex pairs does v participate in [13]. The best exact algorithm for betweenness centrality takes $O(nm)$ time [6], which quickly becomes computationally infeasible as the size of the graph increases.

In [1], Alahakoon et al. introduce a randomized approximation algorithm, RA-κpath, to estimate κ-path centrality. The algorithm samples a number of simple paths and assigns a vertex v a value based on the number of paths v participates in. The approximation algorithm runs in $O(\kappa^3 n^{2-2\alpha} \ln n)$ time, where n is the number of vertices in the graph and α is a hyperparameter which adjusts the tradeoff of computation time and accuracy. Despite being less computationally taxing than any exact betweenness centrality algorithms, RA-κpath still requires a large amount of paths to be sampled for accurate results on large networks. Therefore, an algorithm which can handle both huge networks and sample a large number of paths is necessary for κ-path applications.

In the present paper, we introduce the random neighbor traversal graph (RaNT-Graph), a distributed graph data structure enabling the sampling of massive numbers of random walks and paths. The data structure combines *vertex delegation* partitioning and *rejection sampling*. Partitioning via vertex delegation takes high-degree vertices or *hubs* and splits their adjacency lists amongst all processors, helping balance communication, computation, and storage [22].

Rejection sampling is a common technique used to sample from a (often complex) distribution. To sample a random simple path, a path continually traverses to neighboring vertices until a termination condition is met. However, since the path is simple, the path cannot step to any previously visited vertex. Thus, rather than calculating the set of unvisited neighbors, employing rejection sampling to select an unvisited vertex can greatly reduce compute time. We demonstrate the weak scaling capability of our RaNT-Graph approach on R-MAT graphs. Additionally, we exhibit strong scaling on real world graphs and show up to a $56,544\times$ speedup over the baseline 1D partitioned implementation.

The remainder of this paper consists of notation and definitions (Section II), our approach to distributing RA-κpath by using RaNT-Graph (Section III), threshold and scaling experiments (Section IV), related work (Section V), and concluding remarks (Section VI).

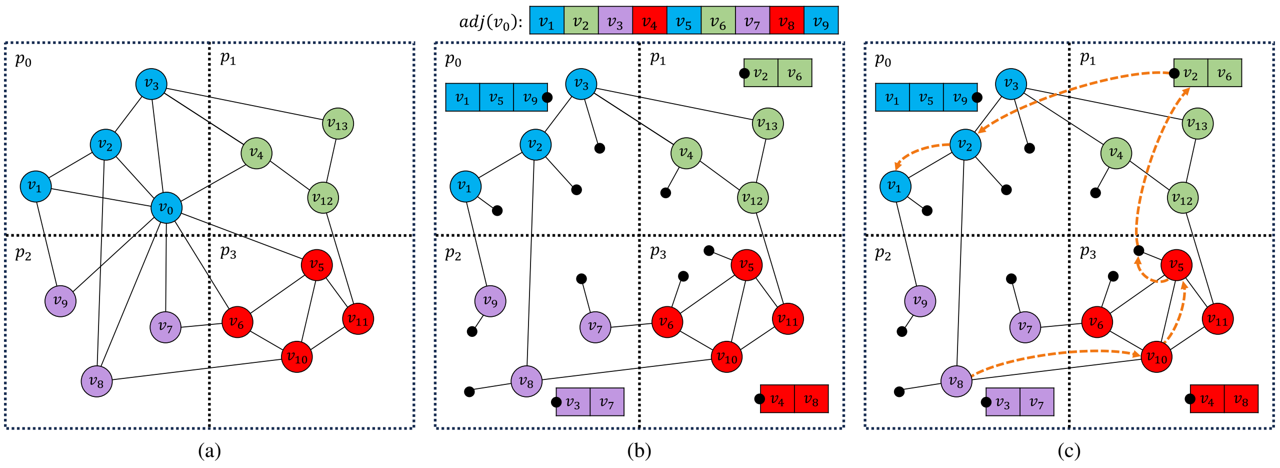

$adj(v_0)$: v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
p_0
p_1
p_2
p_3
(a)
(b)
(c)

Fig. 1: A 1D partitioning (a) of a graph with a hub vertex v_0 stored on processor p_0. A vertex delegation partitioning (b) of the same graph which shows the adjacency list $adj(v_0)$ delegated amongst all processors. The smaller adjacency list contained in each processor's partition represents the portion $adj_{local}(v_0)$ of $adj(v_0)$ owned by each processor. In (c), the dashed orange arrows are the steps of a path taken in a vertex delegation partitioned graph. The order of the vertices visited is $v_8 \rightarrow v_{10} \rightarrow v_5 \rightarrow v_0 \rightarrow v_2 \rightarrow v_1$, the order of the processors visited is $p_2 \rightarrow p_3 \rightarrow p_3 \rightarrow p_1 \rightarrow p_0 \rightarrow p_0$. When the path steps to v_0, DelegatedStep (see Algorithm 3) is asynchronously executed by p_1 which owns the edge from v_0 to v_2.

II. Preliminaries

A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ represents relational data between vertices in the vertex set $\mathcal{V}$ through edges $(i, j) \in \mathcal{E}$, where $i, j \in \mathcal{V}$. Each vertex $v \in \mathcal{V}$ has a neighborhood $\mathcal{N}(v) = \{u \mid (v, u) \in \mathcal{E}\}$ which is the set of vertices adjacent to v. The degree of a vertex v is defined as $d(v) = |\mathcal{N}(v)|$. Throughout this work we also denote a vertex's adjacency list as $adj(v)$ which is an indexable equivalent to $\mathcal{N}(v)$. Further, we denote the number of vertices and edges in the graph as $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ respectively.

A *simple path* $\mathcal{S}$, which has a source vertex s, is a path where vertices cannot be repeated. Given a path $\mathcal{S}$ and vertex v, the set of unvisited neighbors is defined as $\mathcal{U}(v) = \mathcal{N}(v) \backslash \mathcal{S}$. Throughout this work we will often refer to simple paths as paths for the sake of brevity. First introduced by Alahakoon et al. in [1], κ-path centrality is defined as follows:

Definition 1: ***κ-Path Centrality*** *– Given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and a maximum path length κ, the κ-path centrality $\mathcal{C}_\kappa(v)$ of each vertex $v \in \mathcal{V}$ is*

$$\mathcal{C}_\kappa(v) = \textstyle\sum_{s \in \mathcal{V} \backslash \{v\}} \Pr(\mathcal{S}_{\kappa,s}(v))$$

where $\Pr(\mathcal{S}_{s,\kappa}(v))$ is the probability a simple path of length at most κ originating from vertex s will traverse through v.

κ-path centrality can be applied to weighted graphs, where an edge's weight affects the probability it is traversed along. Throughout this work we assume all graphs to be unweighted and undirected. Therefore, when determining the next vertex in a path, an unvisited neighbor of the path's current vertex is chosen uniformly at random.

III. Approach

To approximate κ-path centrality, Alahakoon et al. introduce RA-κpath, a randomized approximation algorithm which samples $2\kappa^2 n^{1-2\alpha} \ln n$ random simple paths, where $\alpha \in [-\frac{1}{2}, \frac{1}{2}]$ is a hyperparameter which determines the number of paths sampled thus affecting the accuracy of the approximation [1]. With probability at least $1 - 1/n^2$, the algorithm provides an approximation with an additive error up to $\pm n^{1/2+\alpha}$. When evaluating the algorithm's ability to identify vertices with high betweenness centrality, it was found that smaller values of α produced better results [1], [16]. This comes with the computational cost of sampling a large amount of paths T, where $T \gg n$. For large graphs, depending on the value of α, generating T random paths takes significant compute time and memory.

To sample the large amount of paths required to estimate κ-path centrality, we propose RaNT-Graph. The graph data structure is capable of performing large amounts of random walks or paths on massive scale-free graphs. The graph data structure provides the ability to quickly choose an unvisited neighbor to visit and helps balance computation amongst all processors via vertex delegation partitioning.

A. Vertex Delegation

The imbalances of storage, compute, and communication are problems often associated with graph algorithms due to the non-uniform topology present in scale-free graphs. We employ vertex delegation partitioning to mitigate these issues. Vertex delegation distributes the adjacency lists of high-degree vertices or hubs amongst all processors [22]. This partitioning technique has been employed in a variety of graph algorithms and has proven to help scaling capabilities [7], [19], [20], [24], [28], [29].

RaNT-Graph utilizes a simplified version of vertex delegation partitioning which does not optimize co-located edges as done in [22]. RaNT-Graph makes use of the threshold d_{thresh} to calculate the set of delegated vertices $\mathcal{D} = \{v \mid d(v) \geq d_{thresh}\}$. Using 1D partitioning, each undelegated vertex v and its adjacency list $adj(v)$ is owned by a processor, denoted as $p_{owner}(v)$. With a delegated vertex v, its original undelegated adjacency list $adj(v)$ must be split amongst multiple processors, where each processor p stores a portion $adj_{local}(v)$ of $adj(v)$. Figure 1b provides an example of this partitioning, where v_0's adjacency list is divided amongst all four processors. Determining which elements of $adj(v_0)$ each processor owns is done in a round robin fashion. With this method, given a global index i_{global} and the total number of processors $|P|$, the processor p_{dest} which owns the element and its local index i_{local} can quickly be calculated as shown in Algorithm 4. For example, in Figure 1b, $i_{global} = 5$ corresponds to p_1's $adj_{local}(v_0)[1]$ element (using zero-based indexing). When using this round robin method, if the same processor is always given the first element of an adjacency list, imbalances can occur. Therefore, an offset is applied to the round robin ordering to help balance the number of elements each processor contains after dividing numerous adjacency lists. We leave out the concept of an offset in our pseudocode to simplify the algorithms.

B. Rejection Sampling

Rejection sampling is a probabilistic method used to generate samples from a target distribution by accepting or rejecting samples based on a comparison with a proposal distribution. It involves generating samples from the proposal distribution and accepting those that fall within the target distribution, while rejecting the ones that do not. The method is commonly employed by random walk frameworks to randomly sample neighbors where the probability distribution is based on edge weights [21], [25], [27].

Recall, when generating a random simple path, the next vertex must be unvisited i.e. it cannot already be in the path. Therefore, an unvisited neighbor must be chosen uniformly at random to be next in the path. For a given vertex v, iteratively constructing $\mathcal{U}(v)$ takes $O(d(v))$ time. When sampling massive amounts of random paths, this calculation at each step is computationally costly. Alternatively, rejection sampling can be applied by randomly selecting any neighbor and accepting it if not present in the path or rejecting it if it is. Consequently, a new neighbor is sampled until one is accepted. The probability of selecting an unvisited vertex follows a geometric distribution. Given a vertex v, the expected number of neighbors sampled until an unvisited one is chosen is

$$M = \frac{d(v)}{d(v) - |\mathcal{W}(v)|}, \tag{1}$$

where $\mathcal{W}(v) = \mathcal{N}(v) \setminus \mathcal{U}(v)$ is the set of visited neighbors of v.

RaNT-Graph takes advantage of rejection sampling when choosing a random neighbor of either a delegated vertex or an undelegated vertex. To sample an unvisited neighbor of a delegated vertex v, first an index i_{global} in the range $[0, d(v) - 1]$ is randomly chosen. As discussed previously, processor p_{dest} and index i_{local} can be derived from i_{global} (see Algorithm 4). Next, if p_{dest} finds $adj_{local}(v)[i_{local}]$ is already in the path, a new neighbor of v is sampled. Otherwise, the path is traversed to the sampled neighbor. Lines 3-6 of Algorithm 3 show this rejection sampling process in a recursive form.

For an undelegated vertex v, there are two ways to sample an unvisited neighbor. The first is calculating $\mathcal{U}(v)$ and sampling from it, and the second is rejection sampling until an unvisited neighbor is chosen from $adj(v)$. Deciding which method to use is determined by a rejection sampling threshold r_{thresh}. If M is greater than r_{thresh} then the neighbor is sampled from $\mathcal{U}(v)$. Conversely, if M is less than or equal to r_{thresh} then the neighbor is rejection sampled. However, calculating M at each step is costly due to $\mathcal{W}(v)$ being derived from $\mathcal{U}(v)$. This negates the potential benefit of rejection sampling. Therefore, the worst case scenario of every vertex in the path $\mathcal{S}$ being a neighbor of v is assumed. Replacing $|\mathcal{W}(v)|$ with $|\mathcal{S}|$ in Equation 1 yields the worst case rejection sampling value $M_{\mathcal{S}}$ as shown in line 3 of Algorithm 2. In addition, to guarantee a neighbor of v is eventually chosen from rejection sampling, $d(v)$ must be greater than $|\mathcal{S}|$. When $d(v)$ is less than $|\mathcal{S}|$, $\mathcal{U}(v)$ is constructed. If $\mathcal{U}(v) = \emptyset$, i.e. all neighbors have been visited, then the path terminates early. Lines 3-16 of Algorithm 2 show the pseudocode of sampling an unvisited neighbor of an undelegated vertex.

Algorithm 1 SampleKPaths

```
Input: RaNT-Graph $\mathcal{R}(\mathcal{V}, \mathcal{E}, \mathcal{D}, d_{thresh}, r_{thresh})$,
    Max Path Length $\kappa$, Total Paths $T$
Output: Count of paths traversed over each vertex $count$
1:  for each $v \in \mathcal{V}$, $count[v] \leftarrow 0$
2:  parfor $t \leftarrow 1$ to $T$ do
3:      $l \leftarrow$ path length chosen uniformly at random from $[1, \kappa]$
4:      $v_{source} \leftarrow$ vertex chosen uniformly at random from $\mathcal{V}$
5:      if $v_{source} \in \mathcal{D}$ then
6:          $p_{dest}, i_{local} \leftarrow$ ChooseDelegateEdge($d(v_{source})$)
7:          async execute DelegateStep($v_{source}, l, \emptyset, i_{local}$) on $p_{dest}$
8:      else
9:          async execute Step($v_{source}, l, \emptyset$) on $p_{owner}(v_{source})$
10:     end if
11: end for
12: return $count$
```

C. YGM

YGM is an asynchronous communication library built on top of MPI that abstracts communication through fire-and-forget semantics [23]. A YGM message instructs another processor to execute some function typically using data contained in the message. Messages are sent by a sender without interacting with the receiver which lends to irregular communication patterns. Additionally, the library employs message buffers which increase throughput by reducing the number of individual messages sent. Messages sent using YGM can contain data of varying type and length and are serialized on departure and deserialized on arrival. RaNT-Graph is built using multiple YGM distributed containers.

In the pseudocode presented throughout this paper the syntax **async execute** $f(x)$ **on** p refers to the YGM model of asynchronously instructing another processor p to execute a function f using data x.

Algorithm 2 Step

```
Input: Vertex $v$, Path Length $l$, Path $\mathcal{S}$
 1: $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
 2: if $|\mathcal{S}| < l$ then
 3:     $M_{\mathcal{S}} \leftarrow d(v)/(d(v) - |\mathcal{S}|)$
 4:     if $d(v) > |\mathcal{S}|$ and $M_{\mathcal{S}} < r_{thresh}$ then
 5:         $v_{next} \leftarrow$ random vertex from $adj(v)$
 6:         while $v_{next} \in \mathcal{S}$ do
 7:             $v_{next} \leftarrow$ random vertex from $adj(v)$
 8:         end while
 9:     else
10:         $\mathcal{U}(v) \leftarrow \mathcal{N}(v) \setminus \mathcal{S}$
11:         if $\mathcal{U}(v) \neq \emptyset$ then
12:             $v_{next} \leftarrow$ random vertex from $\mathcal{U}(v)$
13:         else
14:             for each $v \in \mathcal{S}$, increment $count[v]$
15:         end if
16:     end if
17:     if $v_{next} \in \mathcal{D}$ then
18:         $p_{dest}, i_{local} \leftarrow$ ChooseDelegateEdge($d(v_{next})$)
19:         async execute DelegateStep($v_{next}, l, \mathcal{S}, i_{local}$) on $p_{dest}$
20:     else
21:         async execute Step($v_{next}, l, \mathcal{S}$) on $p_{owner}(v_{next})$
22:     end if
23: else
24:     for each $v \in \mathcal{S}$, increment $count[v]$
25: end if
```

Algorithm 3 DelegateStep

```
Input: Vertex $v$, Path Length $l$, Path $\mathcal{S}$, Local Index $i_{local}$
 1: $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
 2: if $|\mathcal{S}| < l$ then
 3:     $v_{next} \leftarrow adj_{local}(v)[i_{local}]$
 4:     if $v_{next} \in \mathcal{S}$ then
 5:         $p_{dest}, i_{local} \leftarrow$ ChooseDelegateEdge($d(v)$)
 6:         async execute DelegateStep($v, l, \mathcal{S}, i_{local}$) on $p_{dest}$
 7:     else
 8:         if $v_{next} \in \mathcal{D}$ then
 9:             $p_{dest}, i_{local} \leftarrow$ ChooseDelegateEdge($d(v_{next})$)
10:             async execute DelegateStep($v_{next}, l, \mathcal{S}, i_{local}$) on $p_{dest}$
11:         else
12:             async execute Step($v_{next}, l, \mathcal{S}$) on $p_{owner}(v_{next})$
13:         end if
14:     end if
15: else
16:     for each $v \in \mathcal{S}$, increment $count[v]$
17: end if
```

Algorithm 4 ChooseDelegateEdge

```
Input: Degree of Delegated Vertex $d$
Output: Processor $p_{dest}$, Local Index $i_{local}$
 1: $i_{global} \leftarrow$ index chosen uniformly at random from $[0, d-1]$
 2: $i_{local} \leftarrow \lfloor i_{global}/|P| \rfloor$
 3: $p_{dest} \leftarrow i_{global} \mod |P|$
 4: return $p_{dest}, i_{local}$
```

D. Applying RaNT-Graph to κ-Path Centrality

Our application of RaNT-Graph to κ-path centrality provides a method of sampling a random simple path by continually stepping to an unvisited vertex in a recursive manner until a termination condition is met. Each processor executes Algorithm 1, initiating the recursive path traversals which call two major functions, Step and DelegatedStep shown in Algorithms 2 and 3 respectively. To prevent revisiting a vertex, each step taken requires the entire path be contained in the YGM message.

The function Step is asynchronously called when a path is starting at or is traversing to an undelegated vertex v and is executed by $p_{owner}(v)$. Recall, all undelegated vertices are 1D partitioned. Depending on r_{thresh}, the subsequent vertex in the path is either rejection sampled, or chosen from $\mathcal{U}(v)$.

DelegatedStep is asynchronously called when a path is starting at or is traversing to a delegated vertex v and is executed by the processor p_{dest} storing the randomly chosen neighbor $u \in \mathcal{N}(v)$ in its portion $adj_{local}(v)$ of $adj(v)$. Therefore, before stepping to a delegated vertex, the subsequent vertex u in the path —located at $adj_{local}(v)[i_{local}]$ on processor p_{dest}— must be chosen as done in the ChooseDelegateEdge function. When eventually executed on p_{dest}, if u is already in the path, then we resample another neighbor of v and asynchronously execute DelegationStep again, as done in lines 3-6 of Algorithm 3. In either function, a path halts when the length of the path has reached it's specific limit or it has no unvisited neighbors to step to.

When the next vertex in a path is selected, it must be determined whether it is delegated or not. Therefore, each processor stores the entire set of delegated vertices $\mathcal{D}$ but not their full adjacency lists. Additionally, to sample a random index of a delegated vertex's adjacency list, each processor also stores the degree of each delegated vertex.

IV. Experiments

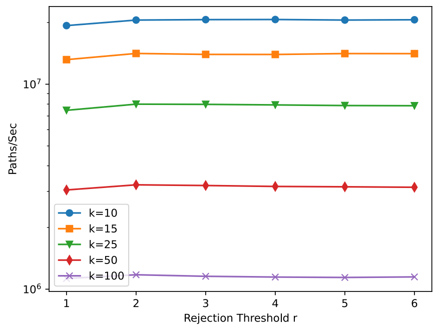
A. Experimental Setup

All experiments were conducted on LLNL's *Catalyst* cluster where each compute node is equipped with dual Intel Xeon E5-2695v2 processors totaling 24 cores and 128GB of DRAM. The network uses an Infiniband QDR interconnect. Our implementation utilizes YGM [23] and was written in C++.

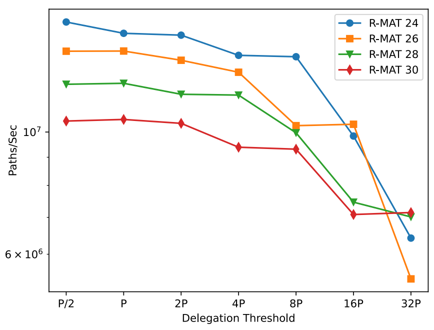
B. Threshold Evaluations

As previously described, RaNT-Graph has two threshold parameters r_{thresh} and d_{thresh}. For κ-path centrality, the value of κ can be thought of as a threshold for the maximum length of a random path. We evaluate altering each of these thresholds to gain insight into what configuration yields the best performance, measured by paths completed per second. All threshold tests were run on 32 compute nodes i.e. 768 processors. Assume a configuration of $d_{thresh} = 768$, $r_{thresh} = 2$, and $\kappa = 20$ unless the value is being altered for evaluation reasons.

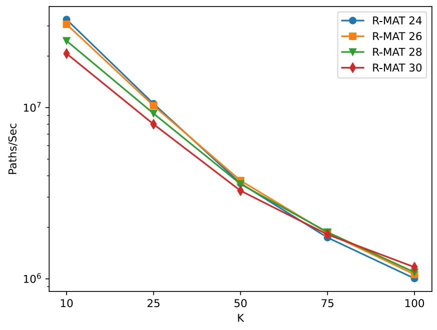
Figure 2a shows altering r_{thresh} has little affect on performance regardless of the parameter κ. The only notable difference is when r_{thresh} equals one. When this is the case, no rejection sampling is performed when sampling an undelegated vertex's neighbor (delegated vertices always rejection sample). This is because the expected number of samples M_S can never be less than one. Since no rejection sampling is performed, the set of unvisited neighbors $\mathcal{U}(v)$ of a vertex v is constructed in $O(d(v))$ time at each step. However, an undelegated vertex's



(a)



(b)



(c)

Fig. 2: Evaluation of the effect the parameters r_{thresh} (a), d_{thresh} (b), and κ (c) have on the performance of sampling paths utilizing RaNT-Graph. All tests were conducted on 32 compute nodes (768 processors). All r_{thresh} tests (a) were run on an R-MAT graph of scale 30. In (b), the x-axis shows varying values of d_{thresh} proportional to the number of processors $P = 768$.

degree is bounded by d_{thresh}, thus constructing $\mathcal{U}(v)$ takes at most $O(d_{thresh})$ time. This indicates why the performance of RaNT-Graph is not greatly impacted by the value of r_{thresh}.

The results of varying d_{thresh} proportional to the number of processors P is shown in Figure 2b. Values of d_{thresh} ranging from $P/2$ to $2P$ result in similar performance, but as d_{thresh} increases the performance is hindered. This is likely caused by greater imbalances in computation and communication inherent to larger d_{thresh} values.

Lastly, adjusting the κ-path centrality parameter κ is shown in Figure 2c and as expected, as κ increases the performance decreases. The increased compute time associated with taking more steps in a path, lowers the number of completed paths per second. Additionally, when taking a step in a path, the entire path must be stored in the message that is sent to the next processor. Therefore, the amount of data sent greatly increases as κ increases. This combination of increased compute and communication degrades performance.

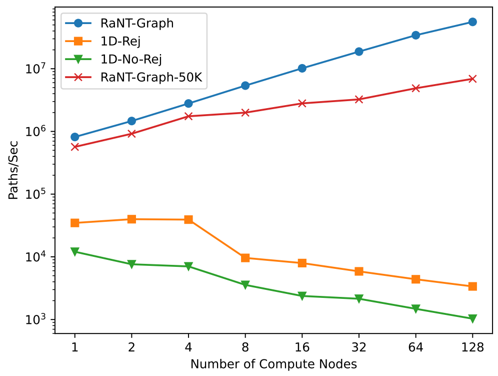


Fig. 3: Weak scaling of RaNT-Graph, 1D-Rej, and 1D-No-Rej based on the number of completed paths per second. Starting with an R-MAT graph of scale 26 for one compute node, up to a scale of 33 for 128 compute nodes. The number of paths to be sampled also scales. For RaNT-Graph the paths to compute node ratio is 10M:1, whereas for 1D-Rej, 1D-No-Rej, and RaNT-Graph-50K the ratio is 50K:1.

C. Weak Scaling

To evaluate our RaNT-Graph based approach, we compare to two 1D partitioned implementations, one which uses rejection sampling (1D-Rej) and one which does not (1D-No-Rej). We perform a weak scaling experiment which measures the amount of paths sampled per second on R-MAT graphs [8] starting at scale 26 and incrementally increasing up to 33. In many graph problems, such as triangle counting or k-core decomposition, the amount of work required scales as the size of the graph increases. When sampling a constant number of paths, the work required does not change with the size of the graph. Therefore, to scale the work as the number of compute nodes increases, we employ a ratio of paths to compute nodes. For RaNT-Graph this ratio is 10M:1, whereas for 1D-Rej and 1D-No-Rej it is 50K:1. This difference is due to the large amount of compute time it would require to sample 10 million paths per compute node using either 1D partitioned implementation. To ensure this ratio is not biased towards RaNT-Graph, we also evaluate RaNT-Graph with a ratio of 50K:1. The methods were configured using $\kappa = 10$, $r_{thresh} = 2$, and $d_{thresh} = |P|$. Figure 3 shows the results of the weak scaling experiment. It can be seen that RaNT-Graph is able to sample multiple orders-of-magnitude more paths than its 1D counterparts and scales with the number of compute nodes.

TABLE I: Graphs used in strong scaling experiments.

Graph	n	m	d_{max}	T	κ
Orkut [26]	3M	117M	33K	74M	18
LiveJournal [2]	4.85M	43M	20K	102M	18
Twitter [17]	42M	1.2B	3M	580M	21
Friendster [26]	66M	1.8B	5.2K	857M	22
web-cc12-hostgraph [18]	89M	1.9B	3M	1B	22
uk-2007-05 [5]	106M	3.3B	975K	1.2B	22

D. Strong Scaling

We provide multiple strong scaling experiments conducted on large-scale real-world graphs of varying sizes. Table I shows the size of each graph, along with their maximum degree, d_{max}. It also shows the total number of paths sampled, T, in each experiment as well as the value of κ used. These values are derived by setting the parameter α to 0.2 in the equations $T = \lfloor 2\kappa^2 n^{1-2\alpha} \ln n \rfloor$ and $\kappa = \lfloor \ln(n + m) \rfloor$ presented in [1]. This configuration of RA-κpath proved to

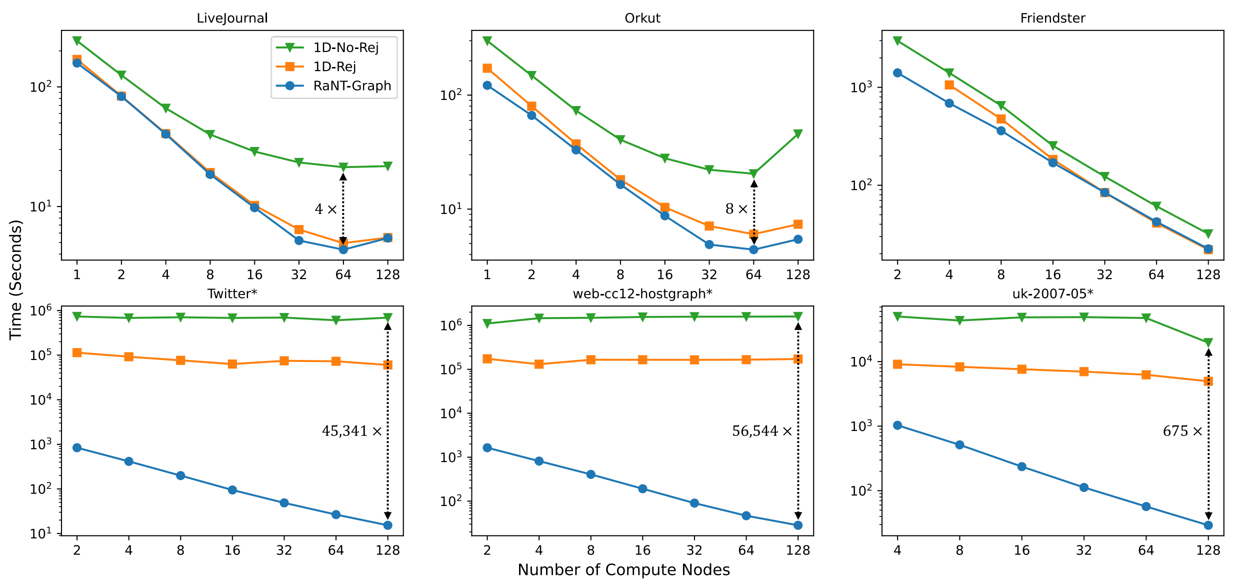


Fig. 4: Strong scaling of RaNT-Graph, 1D-Rej, and 1D-No-Rej on various real world graphs where T paths were sampled (see Table I). *Due to the large compute time required, all values of 1D-Rej and 1D-No-Rej were estimated for the Twitter, web-cc12-hostgraph, and uk-2007-05 networks. These values were estimated by timing the sampling of 1M paths and extrapolating this value based on the desired amount of paths to be sampled T. The time to ingest and construct each graph is not included in the recorded times.

find vertices with high-betweenness centrality which is an important feature of κ-path centrality.

Similar to the weak scaling experiment, we compare RaNT-Graph to 1D-Rej and 1D-No-Rej with a configuration of $d_{thresh} = |P|$ and $r_{thresh} = 2$. The strong scaling results are shown in Figure 4. Except for between 64 and 128 compute nodes on the LiveJournal and Orkut graphs, RaNT-Graph shows a decrease in compute time when strong scaling. Since LiveJournal and Orkut are smaller graphs that require less paths to be sampled, the communication cost outweighs the advantage of distributing the work amongst 128 nodes.

Due to the large compute time required, the run times of 1D-Rej and 1D-No-Rej were estimated for the Twitter, uk-2007-05, and web-cc12-hostgraph networks (See caption of Figure 4 for estimation details). Notice in Table I that these networks all have extremely large maximum degrees. This highlights the RaNT-Graph approach's ability to spread computation associated with high degree vertices amongst all processors. Despite Friendster being a very large graph, all three methods performed similarly. This reveals the benefit of using RaNT-Graph is more dependent on a graph's degree distribution rather than its size.

V. Related Work

To the best of our knowledge, this work is the first to estimate κ-path centrality in distributed memory. KnightKing is a distributed memory general random walk framework that also takes advantage of rejection sampling to greatly reduce sampling time [27]. The framework allows users to define a walker's state which can store application specific data for the walker to use. A path's previously visited vertices can be stored in the walker's state, meaning KnightKing could be utilized to estimate κ-path centrality. However, the framework employs 1D partitioning and would likely result in similar compute and communication imbalances that decrease performance as shown in our experiments.

In [14], an adaptive algorithm is proposed as an alternative to RA-κpath. The algorithm reduces the number of sample paths and samples them faster by computing two subsets of the vertex set for each vertex. The first subset contains vertices where a path's source vertex can be sampled from. The second subset defines the vertices which can be sampled while sampling a path.

VI. Conclusion

Estimating κ-path centrality can require sampling large amounts of paths when applied to large-scale graphs. We introduce RaNT-Graph, a novel graph data structure optimized for sampling massive amounts of simple paths. It combines vertex delegation partitioning with rejection sampling to reduce compute, storage, and communication imbalances caused by high-degree vertices. Our RaNT-Graph approach to estimating κ-path centrality shows good weak scaling on R-MAT graphs of various scale. Additionally, we demonstrate the strong scalability of RaNT-Graph on multiple large-scale real-world graphs. When compared to the baseline 1D partitioned implementations, our approach yields up to a $56,544\times$ speedup.

In future work, we plan to extend RaNT-Graph to algorithms which utilize random walks such as Personalized PageRank [15] and Meta-Path [12]. In addition, enabling RaNT-Graph to account for edge weights when sampling neighbors would introduce new applications such as graph embedding. Further optimizations of RaNT-Graph could be explored such as attempting to make use of co-located edges to further reduce communication.

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL- CONF-854338). Funding from LLNL LDRD project 21-ERD-020 was used in this work.

References

[1] T. Alahakoon, R. Tripathi, N. Kourtellis, R. Simha, and A. Iamnitchi, "K-path centrality: A new centrality measure in social networks," in *Proceedings of the 4th Workshop on Social Network Systems*, ser. SNS '11. New York, NY, USA: Association for Computing Machinery, 2011.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 44–54.

[3] A. Biswas and B. Biswas, "Community-based link prediction," *Multimedia Tools and Applications*, vol. 76, no. 18, pp. 18 619–18 639, Sep. 2017.

[4] J. Blackburn, R. Simha, N. Kourtellis, X. Zuo, M. Ripeanu, J. Skvoretz, and A. Iamnitchi, "Branded with a scarlet "C": cheaters in a gaming social network," in *Proceedings of the 21st international conference on World Wide Web*, ser. WWW '12. New York, NY, USA: Association for Computing Machinery, Apr. 2012, pp. 81–90.

[5] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, 2011, pp. 587–596.

[6] U. Brandes, "A faster algorithm for betweenness centrality," *The Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[7] H. Cao, Y. Wang, H. Wang, H. Lin, Z. Ma, W. Yin, and W. Chen, "Scaling graph traversal to 281 trillion edges with 40 million cores," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 234–245.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[9] P. De Meo, E. Ferrara, G. Fiumara, and A. Provetti, "Enhancing community detection using a network weighting strategy," *Information Sciences*, vol. 222, pp. 648–668, Feb. 2013.

[10] ——, "Mixing local and global information for community detection in large networks," *Journal of Computer and System Sciences*, vol. 80, no. 1, pp. 72–87, Feb. 2014.

[11] P. De Meo, E. Ferrara, G. Fiumara, and A. Ricciardello, "A novel measure of edge centrality in social networks," *Knowledge-Based Systems*, vol. 30, pp. 136–150, 2012.

[12] Y. Dong, N. V. Chawla, and A. Swami, "metapath2vec: Scalable representation learning for heterogeneous networks," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 135–144.

[13] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.

[14] M. Haghir Chehreghani, A. Bifet, and T. Abdessalem, "Adaptive algorithms for estimating betweenness and k-path centralities," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1231–1240.

[15] T. H. Haveliwala, "Topic-sensitive pagerank," in *Proceedings of the 11th international conference on World Wide Web*, 2002, pp. 517–526.

[16] N. Kourtellis, T. Alahakoon, R. Simha, A. Iamnitchi, and R. Tripathi, "Identifying high betweenness centrality nodes in large social networks," *Social Network Analysis and Mining*, vol. 3, no. 4, pp. 899–914, jul 2012.

[17] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[18] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web—revisited: a trick of the heavy tail," in *Proceedings of the 23rd international conference on World Wide Web*, 2014, pp. 427–432.

[19] B. A. Page and P. M. Kogge, "Scalability of Hybrid SpMV with Hypergraph Partitioning and Vertex Delegation for Communication Avoidance," *International Conference on High Performance Computing & Simulation (HPCS 2020)*, Mar. 2021.

[20] Y. Pan, R. Pearce, and J. D. Owens, "Scalable Breadth-First Search on a GPU Cluster," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 1090–1101, iSSN: 1530-2075.

[21] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-saw: A framework for graph sampling and random walk on gpus," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[22] R. Pearce, M. Gokhale, and N. M. Amato, "Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2014, pp. 549–559, iSSN: 2167-4337.

[23] B. Priest, T. Steil, G. Sanders, and R. Pearce, "You've got mail (ygm): Building missing asynchronous communication primitives," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 221–230.

[24] T. Reza, C. Klymko, M. Ripeanu, G. Sanders, and R. Pearce, "Towards Practical and Robust Labeled Pattern Matching in Trillion-Edge Graphs," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 1–12, iSSN: 2168-9253.

[25] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li, "Thunderrw: An in-memory graph random walk engine," *Proc. VLDB Endow.*, vol. 14, no. 11, p. 1992–2005, jul 2021.

[26] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth. corr abs/1205.6233 (2012)," *arXiv preprint arXiv:1205.6233*, 2012.

[27] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: A fast distributed graph random walk engine," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 524–537.

[28] J. Zeng and H. Yu, "A Distributed Infomap Algorithm for Scalable and High-Quality Community Detection," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP '18. New York, NY, USA: Association for Computing Machinery, Aug. 2018, pp. 1–11.

[29] ——, "A Scalable Distributed Louvain Algorithm for Large-Scale Graph Community Detection," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 268–278, iSSN: 2168-9253.