Fast, Exact Graph Diameter Computation with Vertex Programming

Corey Pennycuff Tim Weninger
University of Notre Dame
{cpennycu, tweninge}@nd.edu

ABSTRACT

In graph theory the diameter is an important topological metric for understanding size and density of a graph. Unfortunately, the graph diameter is computationally difficult to measure for even moderately-sized graphs, insomuch that approximation algorithms are commonly used instead of exact measurements. In this paper, we present a new algorithm to measure the exact diameter of unweighted graphs using vertex programming, which is easily distributed. We also show the practical performance of the algorithm in comparison to other, widely available algorithms and implementations, as well as the unreliability in accuracy of some pseudo-diameter estimators.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Vertex-centric computing, graph analytics, diameter

1. INTRODUCTION

As datasets increase in size, we need new algorithms and paradigms in order to take advantage of advances in computer architecture and systems. Vertex computation is one such paradigm in which algorithms are constructed from the perspective of a vertex within a graph. In the vertex programming paradigm, the algorithm is not omniscient in regards to the graph structure, but rather vertices perform computations using messages that are exchanged with their neighbors along their incident edges. The advantage to this vertex-centric approach is that the algorithms running on each vertex may be easily parallelized across a network without knowing the overall structure of the graph of which it is a member.

The diameter of a graph is a foundational metric of graph theory that is routinely considered when analyzing real-world networks (see, for example, [3]). It is a global (single-valued) statistic of the graph which, when taken singularly or in combination with other metrics, help to distill and communicate information about the graph. It exists for all graphs, directed or undirected, real-world or generated. It has been used to evaluate graph generation models [2], and studied for its relation to other graph metrics[12, 6].

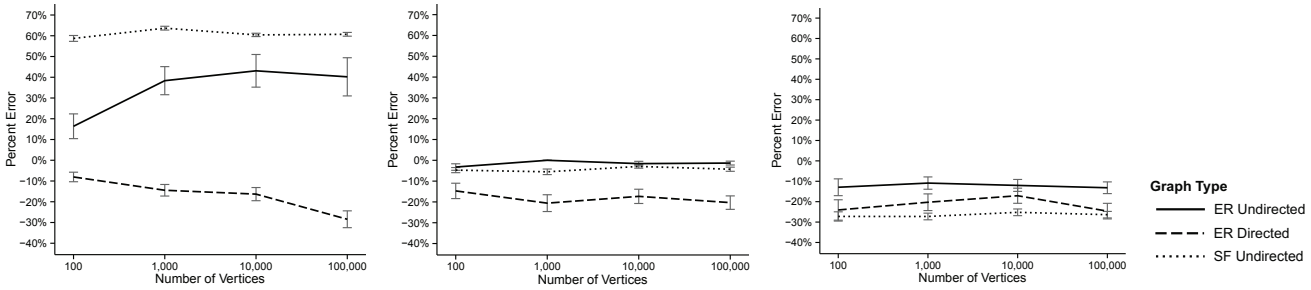
Measuring the Diameter of a graph is computationally expensive. Exact solutions are often implemented using either Floyd-Warshall or Johnson's algorithm, which have complexities of $O(|V|^3)$ and $O(|V|^2 \log |V| + |V||E|)$ respectively, and are not easily distributable. Because of the computational complexity, the diameter of medium and large-sized graphs can only be estimated using pseudo-diameter measurements, which have varying degrees of accuracy. Thus, it is not practical (and in some cases, impossible) to know the exact diameter of such large graphs as social networks, the Web, etc. We begin to address these problems by developing a vertex programming algorithm for measuring the exact diameter of a graph.

In graph theory, the eccentricity $\epsilon(v)$ of a vertex v is the greatest geodesic distance between v and any other vertex in the graph. It may also be viewed as the depth of a breadth first search, rooted at v. The graph diameter d is defined as the largest graph eccentricity for all nodes in a graph (i.e., the largest shortest path within a graph) and may be expressed as $d = \min_{v \in V} \epsilon(v)$. All exact algorithms solve the all pairs shortest paths (All-Pairs Shortest Path (APSP)) problem, and it is still an open problem as to whether or not a diameter may be exactly measured without calculating APSP [4]. As a graph metric, the diameter may be combined with other metrics to indicate the overall structure of the graph. As such, the accuracy of the measurement is of importance to network scientists.

Graph diameter measurement algorithms fall into two categories: exact and approximate, and can be further categorized as sequential or parallel in nature. Exact, serial algorithms include Floyd-Warshall and Johnson's algorithms. The Floyd-Warshall algorithm uses dynamic programming, while Johnson's combines Bellman-Ford and Dijkstra's single source shortest path (Single-Source Shortest Path (SSSP)) algorithm and generally performs better than Floyd-Warshall especially on sparse graphs. Both algorithms are defined serially, although parts of them may be parallelizable.

Parallel algorithms for exact diameter measurement have been developed using parallel graph libraries [7] and shared-memory strategies [14], however all of the algorithms presented are variants and extensions of Dijkstra's original SSSP

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License
http://dx.doi.org/10.5821/hpgm15.2
HPGM'15, August 10, Sydney, Australia.



(a) HADI (b) Reverse Cuthill-McKee (c) Random BFS

Figure 1: Algorithm Error by Graph Type. Algorithms are explained in Section 3.

with no improvement in computational complexity. A parallel solution does exist for SSSP on graphs with non-negative edge weights using Δ-stepping on specialized hardware architectures [11], but we are not aware of such implementations available for commodity systems.

As opposed to exact diameter algorithms, approximation algorithms are substantially less computationally intensive. Simple approximation schemes, like reverse Cuthill-McKee [5], sample paths starting from peripheral vertices (*i.e.*, vertices at the edge of the graph), as paths between peripheral vertices are an intuitive estimation for the graph diameter [8]. An even simpler estimation is to iteratively find path-lengths between random pairs of vertices and record the longest such path as the approximate diameter. The recent popularity of network science and graph theory as resulted in a slew of new approximation schemes, including an algorithm with time complexity of $\tilde{O}(EV^{2/3})$ (where V and E are the number of vertices and edges) [4]. Despite the improved performance, the resulting approximations can be wildly inaccurate, although each pseudo-diameter algorithm typically defines a bounded error rate.

For perspective, Figure 1 shows the average error rate observed between the pseudo-diameter's value and the actual, exact diameter. This behavior is more varied when viewed by the graph type (e.g., Erdős-Rényi (ER) and Scale-free (SF)). These results show the wildly inconsistent performance of the various algorithms as they operate on various graph structures, and further underscores the necessity for faster, exact diameter measurements.

In this paper we present a fast, exact method for calculating the diameter of a graph using vertex computing. The algorithm leverages the near-linear scaling of vertex computing to decrease the algorithmic time complexity by a factor of V.

1.1 Vertex-Centric Computing

Vertex-centric computing is a computational paradigm that iteratively executes a user-defined program simultaneously over all vertices of a graph. The vertex program is designed from the perspective of a vertex, receiving as input the vertex's data as well as data from adjacent vertices and incident edges. The vertex-centric programming model is less expressive than conventional graph-omniscient algorithms, but is easily scalable with more opportunity for parallelism.

Vertex-centric computing is heavily influenced by distributed algorithm theory. Indeed, many graph problems can be solved by both a sequential, shared-memory algorithm as well as a distributed, vertex-centric algorithm. For example, both Dijkstra's and the Bellman-Ford algorithms iteratively replace distance estimates with more accurate values until eventually reaching the solution. Both variants have a superlinear time complexity: Djisktra's runs in $O(E \log E + V)$ and Bellman-Ford's runs in $O(E \times V)$. Perhaps more importantly, both procedural, shared-memory algorithms keep a large state matrix resulting in a space complexity of $O(V^2)$ for both variants. The requirement for a shared state matrix is a drawback for the common algorithms in that, even though parts of their inner loops may be parallelized, each process must access and modify the same (shared) memory structure.

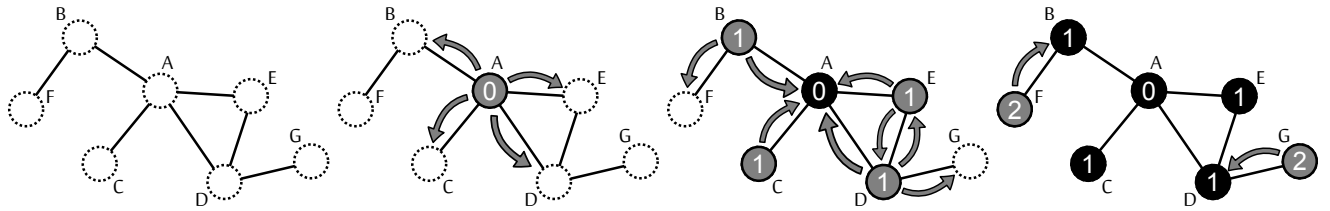
In contrast, our algorithm allows each vertex to manage their own history, which makes it easy to distribute the algorithm across many machines. To solve the same single-source shortest path problem in the vertex computation model (VC-SSSP) we perform the following for each vertex in parallel: (1) a vertex determines the minimum value among all messages received (initially, the source vertex receives a message of '0'), (2) the vertex adopts the minimum value as its shortest path length, and (3) the vertex sends the new path length plus respective edge weights to its outgoing neighbors. If a vertex does not receive any new messages, then it becomes inactive. When all vertices are inactive the process halts.

The number of superstep-iterations required for this SSSP implementation gives the eccentricity of the source vertex $\epsilon(v)$ and sends at least E messages to all V vertices. We measure the complexity of VC-SSSP by the number of messages sent, which is on the order of $O(E)$, even though there were only $\epsilon(v)$ iterations. Because the diameter of a graph can be defined as the largest vertex eccentricity, it is possible to repeat the SSSP process V-times to find the largest eccentricity and, by definition, the exact diameter of the graph.

Note that we do not consider the complexity of the Vertex Computing framework itself, as the complexities for the different stages may vary depending on implementation for all algorithms using that framework.

2. FAST EXACT GRAPH DIAMETER

Rather than finding the largest vertex eccentricity one-by-one, vertex-centric computing provides a framework for parallel computing. Thus, the diameter algorithm works by computing the eccentricity of every vertex simultaneously.



(a) Initial State (b) Superstep 0 (c) Superstep 1 (d) Superstep 2

Figure 2: Computing the eccentricity of vertex A.

Algorithm 1: Diameter Vertex Program Apply Function

Data: *incoming* = A set of message IDs received from neighbors

```
if superstep = 0 then
    outgoing ← {self.id};
    history.add(self.id);
else
    outgoing ← {};
foreach id ∈ incoming do
    if id ∉ history then
        history.add(id);
        outgoing.add(id);
send(outgoing);
```

2.1 The Vertex Computation Exact Diameter Algorithm

The vertex's compute() function is described in Alg 1, and operates with the following process: 1) initially, each vertex adds it's own unique id to the outgoing-message and history sets, which reside in vertex-local memory; 2) after the initial superstep, the algorithm operates by iterating through the set of received ids, which correspond to the vertex that sent the original message; 3) the receiving vertex then constructs a set of *outgoing* messages by adding each element of the *incoming* set which has not yet been seen. The reason for keeping a history of the originating ids that have been received is to prevent a message from being re-propagated to the same vertices. If the vertices did not keep a history set, then all messages would be repeatedly passed back and forth among all pairs of adjacent vertices without halting. The history set also serves to prune the set of total messages by eliminating message paths that would never result the vertex's eccentricity.

Certain frameworks allow for asynchronous or synchronous supersteps. However, the exact diameter algorithm must use the synchronous approach to assure that a message is propagated from its source in a breadth-first manner. Asynchronous timing would allow random messages to reach vertices out of order, thereby eliminating the distance guarantees. Each vertex has knowledge of the global superstep, but has no other knowledge of the overall graph structure other than its immediate neighbors and the messages that are exchanged along their incident edges.

2.1.1 Example: Computing the Eccentricity of a Single Vertex

Without loss of generality, Figure 2 shows execution of the above algorithm from the *perspective of a message originating from a single vertex* (A) in an undirected graph. The numbers of a node indicate the superstep on which the vertex received a message originating at vertex A; the colors of a node, white, grey, black, indicate if the node is unseen, active, or finished respectively in terms of the message originating from A. The initial state of the graph is shown in Fig. 2a, and the message that is passed will originate from vertex A. In superstep 0 (Fig. 2b), vertex A initializes itself by recognizing that the first superstep is taking place, creating a message which contains the unique identifier of that vertex. Because the vertex originates the first message, the message is added to its history and outgoing queue. The message is then sent to its neighbors (B, C, D, and E). (Note: The presence or absence of the message value within the history set is sufficient for the algorithm to function. Additional information is present as an aide for the reader.)

In superstep 1 (Fig. 2c), vertices B, C, D, and E have received the message from A. They have not seen this message, so they add it to their respective histories and outgoing queues. They then pass the message to their respective neighbors.

In superstep 2 (Fig. 2d), vertices A, D, and E have received the message, but because the message already exists in their history, they disregard it. Vertices F and G, however, have not seen the message, so it is added to their respective history and outgoing queue. In superstep 3 (not shown), vertices B and D receive the message, but have already seen it, and therefore no further propagation occurs. Thus, all processing stops after superstep 3, and we are guaranteed that the eccentricity of vertex A is the number of supersteps (3) except the final non-processing superstep (1) of this example ($3 - 1 = 2$, so $\epsilon(A) = 2$).

2.1.2 From Eccentricity to Diameter

While the previous example computed the eccentricity of vertex A, the same algorithm be followed to compute the eccentricity of vertex B at the same time. The only changes are that both A and B will originate a unique message on superstep 0, and that all of the vertices will now have to store 2 pieces of information in their respective histories: Whether or not they have seen the message from vertex A, and whether or not they have seen the message from vertex B. In this particular example, the algorithm will run for one superstep longer than before, because the $\epsilon(B) = 3$, whereas the $\epsilon(A) = 2$ (as stated previously).

We can further extend this idea by having all vertices originate a unique message in superstep 1, all vertices maintaining a history of which messages they have and have not seen, and continuing the algorithm until there are no more

messages to propagate. If the graph is connected, then each vertex will process a message from each originating vertex exactly once. Because the computation completes when the largest eccentricity is calculated, the diameter of the graph is equal to the number of supersteps (minus 1, for the final, non-processing superstep). In the same example as above, the message originating from vertex G will arrive at F after 4 supersteps, (and vice versa in this undirected graph, *i.e.*, the message originating at F will arrive at G after 4 supersteps as well). Thus, the program will halt after 5 supersteps indicating a full, exact diameter of $(5-1=)4$.

Special considerations are required for directed graphs because it is possible for a vertex in a directed graph to have no outgoing edges (i.e., it is a sink). In such a case, it is necessary to transform the graph by adding a self-loop on any sink. The addition of a self-loop on a sink is necessary because it is possible that the message originating at the vertex with the maximum eccentricity may terminate at a sink, causing the algorithm to complete 1 superstep early. The simple addition of self-loops to sinks allow the algorithm to proceed normally.

Because the process works by counting supersteps, graphs with weighted edges are not supported for this algorithm. In weighted graphs, counting supersteps cannot ensure that the synchronous propagation of messages through the network will reveal the actual diameter in all cases, although further exploration in this area is possible.

2.2 Analysis

From this simple example, we find that the diameter of the graph can be calculated by having all vertices originate a unique message on superstep 0, and then let the algorithm proceed iteratively until no more messages are sent. There are therefore $\Theta(|V|)$ unique messages present in the graph, each originating from a single vertex. For all graphs, each message will be passed $O(|E|)$ times. In an undirected graph a message will be passed along each edge exactly twice (once for the initial message, once for the return that is disregarded) resulting in a total message complexity of $O(|V||E|)$. In a directed graph, each message will be passed along an edge exactly once, as messages are only sent along outgoing edges, also resulting in a message complexity of $O(|V||E|)$.

The entirety of the messages will propagate through the graph in $O(d)$ supersteps, where d is the diameter of the graph. Although the number of messages that each vertex relays is not constant from one superstep to the next, and will vary according to the topology of the graph, the *total* number of messages that will be passed is constant.

Given that each vertex must store a history of the messages received, each vertex must store $O(|V|)$ vertex IDs. The total memory requirement of the algorithm, then, is $O(|V|^2)$. Because vertex programming frameworks often permit the vertices to be distributed across multiple machines, the memory requirement may also be partitioned across these machines.

2.3 Batch Processing To Reduce Memory Use

The $O(|V|^2)$ memory requirement is not insignificant. It is possible, however, to reduce the parallelism and therefore reduce the concurrent memory requirement. This reduction is achieved by breaking apart the simultaneous eccentricity measurements in a series of batches. Assume, for example, that the memory requirement to compute the diameter of a graph is $M = O(|V|^2)$ and that this computation is to be performed in b batches, where $\bigcup_{i=0}^{b} B_i = V$, and $|B_0| = |B_1| = \ldots = |B_b| = \frac{|V|}{b}$. During each batch B_i, only $v \in B_i$ will initiate its message in superstep 0 of that batch. Each vertex, then, only has to maintain a history of size $|B_i|$. Batches are processed serially, in which the history of a previous batch may be discarded. It is simple to see that the total memory usage is $|V||B_i|$, or equivalently, $\frac{M}{b}$.

While this change does not reduce the runtime complexity of the algorithm, it provides a practical relief of memory use in the implementation. The memory used in B_i is, in effect, recycled for use by B_{i+1}, and thoughtful implementation of this feature requires limited additional computation. The only performance penalty with such an approach is in the overhead of each batch's creation; there is no increase in the overall total number of messages that will be passed or processed in the `compute()` stage.

2.4 Correctness

To show the correctness of the algorithm for computing eccentricity, we demonstrate that the propagation of a message from a single vertex in the manner described in the algorithm is equivalent to a breadth-first traversal of the graph from the originating vertex. As per the algorithm description from Alg. 1, a vertex may only send a message which it has received from a neighboring vertex, save for the special case in which it originates its own message on superstep 0. A vertex may only send messages to another vertex with whom it has an incident edge in an undirected graph, or along its outgoing edges in a directed graph. This stepwise propagation ensures that, for any vertex that is a distance of d from the message-origin, the vertex will send a message to its neighbors. If the neighbor has not seen the message before, then the neighbor is guaranteed to be a distance of $d+1$ from the message-origin. If the neighbor has already seen the message, then it has a distance $\leq d+1$ from the origin, so the message is discarded.

If the message originator has a distance of 0 from itself, then its neighbors will have a distance of 1. Thus, we see by induction that the message propagates outward from the message origin in a breadth-first manner, and because of the breadth first nature of the propagation, the depth of the BFS tree (i.e., the number of synchronous supersteps) must be equal to the eccentricity of the originating vertex.

2.5 Implementation Details

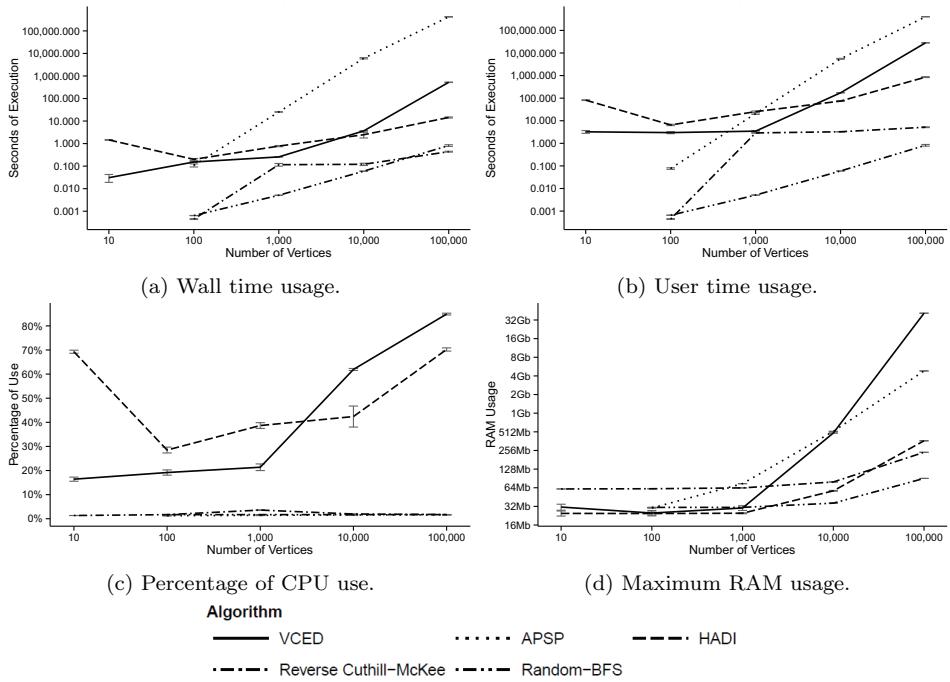
The Vertex Computation Exact Diameter (VCED) algorithm, described above, was implemented in the PowerGraph 2.2 framework[1] using the gather, apply, scatter model [9]. All scripts, source code, and test graphs are available at our GitHub repository[2].

3. EXPERIMENTS AND RESULTS

To test the comparative performance of VCED, we measured its performance with an analysis of several types of graphs of different sizes, and compared this to the performance of diameter calculations of other established algorithms. In terms of graph topology, we created two classes of

[1]Available at `https://github.com/graphlab-code/graphlab/tree/v2.2`

[2]Available at `https://github.com/nddsg/graphlab/tree/diameter-v1.0`



(a) Wall time usage.

(b) User time usage.

(c) Percentage of CPU use.

(d) Maximum RAM usage.

Figure 3: Algorithm Performance on Undirected Erdos-Reyni Graphs ($p = 32\%$).

graphs: ER undirected, and SF undirected. All graphs were constructed using node counts of 10, 100, 1000, 10000, and 100000. For ER graphs, we created graphs with node:edge ratios of 1:32. For SF graphs, we created graphs using a k parameter of 2.0. We created five random examples of each graph.

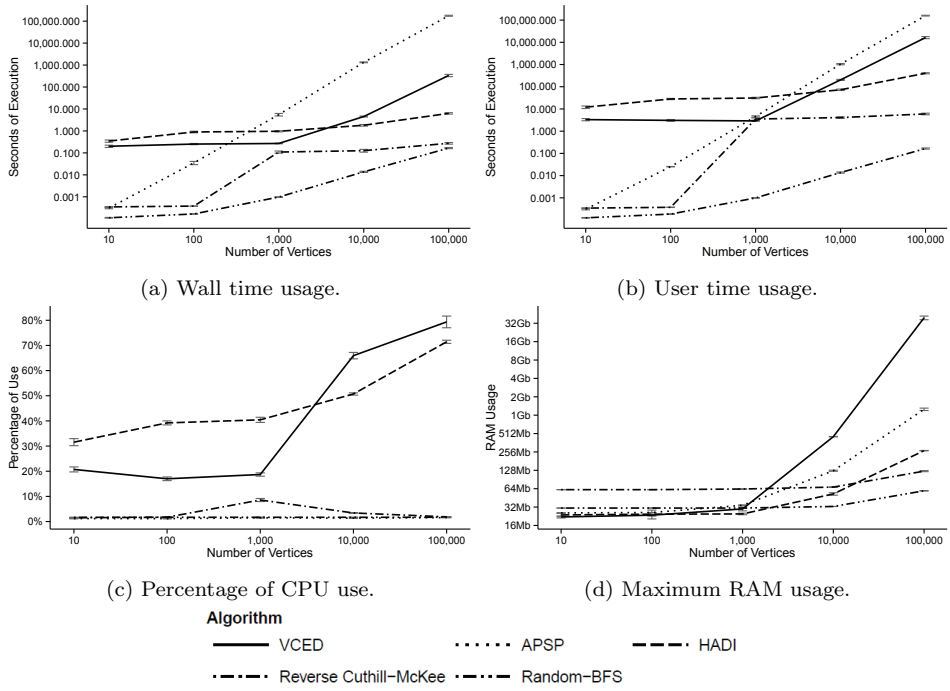
We compared VCED to the exact graph measurement algorithms APSP as implemented by NetworkX. We also compared it to pseudo-diameter algorithms HAdoop DIameter and radii estimator (HADI) [10], reverse Cuthill-McKee [5] and random BFS as implemented by PowerGraph, Graph-Tool, and SNAP respectively. Experiments were performed on a single virtual machine with 58 cores at 2.4GHz each and 256GB of RAM. We measured User Time and Wall-Clock time for all tests as well as maximum RAM usage. All variants of ER and SF graphs had similar performance profiles, with vertex and edge count being the only factors significantly affecting the results. It should be noted that we fully expect all pseudo diameter methods to be much faster and more efficient than exact measurements. It should also be noted that algorithms from NetworkX failed when graphs were not connected, which is a frequent occurrence in smaller, randomly generated graphs.

Figures 3a and 4a show the *wall time* performance of the algorithm on a log-log axis (lower is better). Wall time, defined as the amount of time that passes as measured by a clock on the wall, highlights the advantage of parallelism provided by our algorithm as opposed to the serial nature of APSP. We expected our algorithm to perform slightly better than shown here due to its better runtime complexity, however some anticipated performance gains were thwarted by the framework's implementation. Most of the performance overhead when exchanging messages could be attributed to the overabundance of very small memory allocations and deallocations when compiling and sending messages between vertices (especially in the `gather()` step). Furthermore, some of the backing structures for message passing had $O(logN)$ performance, which may have further inflated the time. We acknowledge, however, that dealing with such large memory structures is inherently slow due to the heavily fragmented memory access patterns and poor locality of reference. Future research may be able to improve the memory performance of this algorithm.

Figures 3b and 4b show the *user time* performance of the algorithm, also graphed on a log-log axis. User time is the sum of the processing time for all processors used by a program (lower is better). The advantage of user time is that it forces all processor usage to be accounted for, and reveals whether or not performance gains are merely the benefit of adding more cores to an easily-parallelizable problem. This graph, which is based on the same graph set as figure 3a and 4a, shows the order-of-magnitude performance gain of our algorithm over APSP in large graphs. Figures 3b and 4b also demonstrate the overhead of the vertex-computing framework when applied to smaller graphs in both vertex-centric approaches, i.e., VCED and HADI.

Figures 3c and 4c shows the efficiency of each algorithm in its use of the available processing power. Single threaded solutions are at an obvious disadvantage here. Vertex-centric computing (used in VCED and HADI) utilize the available CPU power effectively in large graphs, but this might also indicate that vertex-centric frameworks have a significant amount of overhead as graph sizes grow. These graphs also indicate that VCED and HADI have a large initial overhead in setup, which dominates the percentage of use when processing the smaller graphs.

Figures 3d and 4d show a log-log graph of the RAM usage by the different algorithms. All algorithms use more RAM as the graph grows, and VCED memory requirements grow faster than the APSP algorithm, which is expected. The



(a) Wall time usage.

(b) User time usage.

(c) Percentage of CPU use.

(d) Maximum RAM usage.

Figure 4: Algorithm Performance on Undirected Scale Free Graphs ($k = 2$).

						Pseudo-Diameter (Error±StdDev)		
Graph	Nodes	Edges	Diameter	Wall Time (m:s)	Mem (Mb)	Rand BFS	Rev C-K	HADI
Facebook	4,039	88,234	8	00:01	63	0 ± 0	0 ± 0	$+5.2 \pm .84$
ca-HepPh	12,008	118,521	13	01:22	2,772	0 ± 0	0 ± 0	$-3.6 \pm .54$
ca-CondMat	23,133	93,497	15	01:38	7,518	-1 ± 0	0 ± 0	-5.0 ± 0
email-Enron	36,692	183,831	13	00:44	3,500	-2 ± 0	0 ± 0	$-4.6 \pm .54$
loc-gowalla	196,591	950,327	16	23:23	118,481	-2 ± 0	0 ± 0	-6.0 ± 0

Table 1: Results of Diameter processing on 5 real world graphs. Results from Pseudo-diameter algorithms are shown at right. Each pseudo-diameter algorithm was executed 5 times, the error of the mean result is shown as well as the standard deviation of the five executions.

nature of VCED requires each vertex to keep a record of which messages it has received, which is a $O(|V|^2)$ space requirement. Although the memory requirement is a clear drawback to our algorithm, recall that the memory burden may be shared by multiple machines, and that the VCED algorithm may also be run in several small batches if needed. For these experiments, we did not use multiple batches in order to reduce the memory footprint.

We further used VCED to compute the diameters of five reasonably sized graphs randomly chosen from the SNAP datasets[3]. The wall time and memory usage of VCED are reported in Table 1 showing that the performance results obtained on artificial graphs do indeed generalize to real world graphs. The right three columns in Table 1 show how much the mean diameter result of five repetitions differs from the exact diameter (± the standard deviation). The pseudo-diameter algorithms, all of which ran in less than 10 seconds on our test graphs, show reasonable accuracy in some cases and poor performance in others.

4. OTHER APPLICATIONS

[3]`http://snap.stanford.edu/data/`

The benefits of our algorithm is not only in determining the exact diameter, but also in the ability to examine the individual histories of each vertex. The first, obvious use is that the eccentricity for each node is easily determined. If the history contains a record of when (on which superstep) an individual message is received, then a more nuanced picture of the graph emerges.

The *neighborhood* of each vertex (the number of nodes that are reachable within a certain number of hops) is trivial to calculate. If the graph is undirected, then the node must only look into its own history to count the nodes whose messages it received within the designated number of hops. If the graph is directed, then a map-reduce job can inspect all the nodes to determine if the message from the source node arrived within the designated number of hops. In either case, the query is $O(|V|)$ after the exact diameter has been determined. While approximation functions have been developed for this task[1, 13], an exact solution is still computationally intensive.

Our algorithm could also be modified to act as an approximation algorithm. One approximation would be to only activate a small subset of the graphs (similar to the batch approach discussed earlier) and take the largest eccentricity

as the approximation. The overall benefit to this approach is that immense graphs could be analyzed, while each vertex is only required to store a small history representing the subset of nodes which initialized a message.

5. CONCLUSIONS

Currently, computing the exact diameter of even medium-sized graphs is often viewed to be impractical. Indeed most requests to determine this metric is typically discarded prima facie because of the computational difficulty posed by the task.

As a remedy to this problem, this paper presents a parallel algorithm able to compute the exact diameter of a non-weighted graph using a manageable $O(|V||E|)$ messages. We report the performance of our algorithm as compared to other exact and approximation algorithms, and we show the accuracy of the pseudo-diameter measurement algorithms and its volatility in relation to the type of graph being measured.

We finally note that the PowerGraph framework used to implement the VCED did posses severe memory allocation issues pertaining to mallocs during growth in the vertex data, which is further exacerbated by the immense memory requirements of analyzing large graphs. Nevertheless, vertex-centric platforms are in their infancy; progress in system design and future updates are sure to alleviate these performance issues and improve the running time of VCED and other related vertex-centric methods significantly.

In terms of practicality, we observed that, for smaller graphs, a naïve, non-parallelized APSP can be faster than the VCED approach, due to the framework overhead. We also see the speed at which larger graphs can be processed, however at the cost of large memory requirements.

Further research will involve decreasing the memory footprint of the algorithm, as well as exploring vertex partitioning algorithms to better group vertices and reduce inter-process and inter-machine communication.

6. ACKNOWLEDGEMENTS

We thank Baoxu Shi and Garrett McGrath for their help and discussion. This research is sponsored by the Air Force Office of Scientific Research FA9550-15-1-0003.

7. REFERENCES

[1] P. Boldi, M. Rosa, and S. Vigna. Hyperanf: Approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th international conference on World wide web*, pages 625–634. ACM, 2011.

[2] B. Bollobás and O. Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[3] U. Brandes and T. Erlebach. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[4] S. Chechik, D. H. Larkin, L. Roditty, G. Schoenebeck, R. E. Tarjan, and V. V. Williams. Better approximation algorithms for the graph diameter. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1041–1052. SIAM, 2014.

[5] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference of the ACM*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[6] A. H. Dekker and B. D. Colbert. Network robustness and graph topology. In *Proceedings of the 27th Australasian conference on Computer science-Volume 26*, pages 359–368. Australian Computer Society, Inc., 2004.

[7] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-source shortest paths with the parallel boost graph library. *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ*, pages 219–248, 2006.

[8] N. E. Gibbs, J. Poole, William G., and P. K. Stockmeyer. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX.

[10] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *ACM Trans. Knowl. Discov. Data*, 5(2):8:1–8:24, Feb. 2011.

[11] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel shortest path algorithms for solving large-scale instances. 2006.

[12] B. Mohar. Eigenvalues, diameter, and mean distance in graphs. *Graphs and combinatorics*, 7(1):53–64, 1991.

[13] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: A fast and scalable tool for data mining in massive graphs. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 81–90. ACM, 2002.

[14] G. Vaira and O. Kurasova. Parallel bidirectional dijkstra's shortest path algorithm. In *Proceedings of the 2011 Conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, pages 422–435, Amsterdam, The Netherlands, The Netherlands, 2011. IOS Press.